A Middleware Approach to Building Content-Centric Applications

Gareth Tyson B.Sc. (Hons) M.Sc.

A Thesis Submitted for the Degree of Doctor of Philosophy

Department of Computer Science
Lancaster University
UK

May, 2010

Acknowledgements

I would like to thank everybody who has helped me during the compilation of this thesis. First and foremost I would like to thank Andreas Mauthe for his invaluable contributions to my research. I could not have asked for a more insightful and dedicated supervisor, who always found time to help regardless of his own workload. I realise this is a rare characteristic and I consider myself very lucky to have worked with him.

In addition, I would also like to thank a number of other people who have made a range of immeasurable contributions to my work over the years: Sebastian Kaune, Yehia El-khatib, Paul Grace, Thomas Plagemann, Gordon Blair, Magnus Skjegstad, Rubén Cuevas Rumín, as well as everybody from the CONTENT and INTERSECTION projects. It was a privilege to work with you all. Alongside these people, I would also like to acknowledge David Hutchison, Edmundo Monteiro and Michael Luck for making a number of helpful suggestions regarding the final version of this thesis.

I would also like to thank everybody at Infolab21 and Lancaster University who made my time here so enjoyable, including Ali, Andi, Andreas, Andy, Colin, Eddie, Frodo, Jamie, Jason, Jayne, Kirsty, Matt, Mu, Rach, Rick, Sean, Sue and Yehia. In particular, I'd like to thank Zoë for keeping me sane (and smiling) whilst writing up!

Last of all, I'd like to thank my parents for supporting me during every stage of my education and always being there to lend a helping hand.

Abstract

Recent years have seen a huge proliferation in the use of content in distributed applications. This observation has been exploited by researchers to construct a new paradigm called *content-centric networking*. Within this paradigm, applications interact with the network using a simple content request/reply abstraction. The network is then responsible for routing this request towards the 'nearest' provider that can offer the content. This process therefore exploits the observation that applications rarely have a vested interest in *where* their content is obtained from. However, it currently ignores the fact that many applications similarly have no interest in *how* their content is obtained, as long as it falls within certain requirement bounds (e.g. performance, security etc.). Consequently, existing content-centric interfaces offer no support for stipulating such abstract requirements. This thesis therefore proposes an extension of the content-centric abstraction to include the concept of *delivery-centricity*. A delivery-centric abstraction is one that allows applications to associate (high-level) delivery requirements with content requests. The purpose of this is to offer access to content in a specialised and adaptable way by exploiting an application's ambivalence towards the underlying means by which it is acquired. Through this, applications can simply issue abstract requirements that are satisfied by the underlying system. These requirements can range from performance needs to more diverse aspects such as overheads, anonymity, monetary cost and the ethical policies of providers.

Using the above principles, this thesis proposes the design of a new system that can offer a delivery-centric abstraction. This process is guided by key design goals, which dictate a solution should be interoperable with existing sources, highly deployable and extensible. Specifically, a middleware approach is taken, which results in the development of the *Juno* middleware. Juno operates by using configurable protocol plug-ins that can interact with various third party providers to discover and access content. Using these plug-ins, Juno can dynamically (re-)configure itself to deliver content from the sources that are most conducive with the application's requirements.

The thesis is evaluated using a number of techniques; first, a detailed study of

real-world delivery protocols is performed to motivate and quantify the benefits of using delivery-centricity. Alongside this, Juno's functional aspects (discovery and delivery) are also evaluated using both simulations and a prototype deployment to understand the performance, overheads and feasibility of using a delivery-centric abstraction. Throughout the thesis, *performance* is focussed on as the primary delivery requirement. It is shown that utilising a delivery-centric abstraction can dramatically increase the ability to satisfy this requirement, and that Juno's approach fully supports such improvements. It is then concluded that placing delivery-centricity in the middleware-layer is a highly effective approach, and that it can be performed in a feasible manner to ensure that delivery requirements are met. The key contributions of this thesis are therefore, (*i*) the introduction of a new delivery-centric abstraction, (*ii*) the design and implementation of a middleware solution to realise this abstraction, and (*iii*) the development of novel technologies to enable content-centric interoperation with existing (and future) content providers.

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university.

The prototype implementation of the proposed system was written entirely by myself, although the OpenCOM component model used to develop it was implemented by Paul Grace, who also kindly provided some performance measurements. The BitTorrent measurement study detailed in this thesis was performed in conjunction with Sebastian Kaune and Rubén Cuevas Rumín.

Gareth Tyson.

Publications

During the compilation of this thesis, the following related articles* have been published by the author.

- Sebastian Kaune, **Gareth Tyson**, Konstantin Pussep, Andreas Mauthe, Aleksandra Kovacevic and Ralf Steinmetz. The Seeder Promotion Problem: Measurements, Analysis and Solution Space. In Proc. 19th IEEE Intl. Conference on Computer Communication Networks (ICCCN), Zurich, Switzerland (2010).

- Sebastian Kaune, Ruben Cuevas Rumin, **Gareth Tyson**, Andreas Mauthe, Carmen Guerrero and Ralf Steinmetz. Unraveling BitTorrent's File Unavailability: Measurements and Analysis. In Proc. 10th IEEE Intl. Conference on Peer-to-Peer (P2P), Delft, Netherlands (2010).

- **Gareth Tyson**, Paul Grace, Andreas Mauthe, Gordon Blair and Sebastian Kaune. A Reflective Middleware to Support Peer-to-Peer Overlay Adaptation. In Proc. 9th Intl. IFIP Conference on Distributed Applications and Interoperable Systems (DAIS), Lisbon, Portugal (2009).

- **Gareth Tyson**, Andreas Mauthe, Sebastian Kaune, Mu Mu and Thomas Plagemann. Corelli: A Dynamic Replication Service for Supporting Latency-Dependent Content in Community Networks. In Proc. 16th ACM/SPIE Multimedia Computing and Networking Conference (MMCN), San Jose, CA (2009).

- **Gareth Tyson**, Paul Grace, Andreas Mauthe and Sebastian Kaune. The Survival of the Fittest: An Evolutionary Approach to Deploying Adaptive Functionality in Peer-to-Peer Systems. In Proc. ACM/IFIP/USENIX Middleware: Workshop on Adaptive and Reflective Middleware (ARM), Leuven, Belgium (2008).

*A comprehensive list is available at http://eprints.comp.lancs.ac.uk

- **Gareth Tyson**, Andreas Mauthe, Thomas Plagemann and Yehia El-khatib. Juno: Reconfigurable Middleware for Heterogeneous Content Networking. In Proc. 5th Intl. Workshop on Next Generation Networking Middleware (NGNM), Samos Islands, Greece (2008).

Contents

Chapter 1

Introduction

1.1 Overview

Recent years have seen a huge proliferation in the use of content in distributed applications. This has resulted in a large percentage of Internet traffic being attributed to various types of content distribution [39][60][128]. Consequently, its delivery has become a well studied research topic from both providers' and consumers' perspectives. One such avenue of recent investigation is that of *content-centric networking*, which proposes the radical overhaul of current Internet technologies to make the discovery and delivery of content an explicit network-level function.

The proposed benefits of content-centric networking are wide. Within such a system, content would be uniquely addressed with the ability to be accessed from any location using a simple request/reply paradigm. This would make application development simpler and allow an array of network-level optimisations to be performed relating to the storage and distribution of content. However, the placement of traditionally application-level functionality at the network-level has resulted in a number of identifiable limitations. Specifically, this refers to (*i*) poor interoperability between existing distribution schemes, (*ii*) complicated deployment, and (*iii*) an inability to configure or adapt deliveries.

This thesis investigates content-centric networking, with a focus on addressing the above issues. To achieve this, the principle of content-centric networking is extended to include the concept of *delivery-centricity*. This is the ability for an application to associate content requests with delivery requirements (e.g. performance, security, access preferences etc.) that are dynamically resolved and satisfied by the content-centric network. To investigate this, a new delivery-centric abstraction is designed and implemented, with a focus on ensuring easy deployment and high levels of interoperability with existing delivery schemes.

The remainder of this chapter provides an introduction to content-centric networking. First, an overview of content-centric networking is given, looking at both its potential benefits as well as its current limitations. Following this, the aims of this thesis are explored, looking at the important research goals. Last, an overview of the thesis is given.

1.2 Content-Centric Networking

1.2.1 Overview

Broadly speaking, content-centric networking refers to any type of network that views content as a central tenet of its operation [116]. However, recently the term has been used to describe a new type of clean-slate network, which follows a content publish/subscribe paradigm [81]. Providers can *publish* their ability to serve an item of uniquely identified content, which consumers can then *subscribe* to accessing. This is done through a common operating system abstraction [54], leaving the network to manage the intermediate delivery and optimisation.

A number of designs have emerged in the last few years, including PARC's Assurable Global Network (AGN) [82] and U.C. Berkeley's Data-Oriented Network Architecture (DONA) [93]. These propose the deployment of routing infrastructure within the Internet, alongside the introduction of a new type of networking stack to interact with it. DONA, for instance, proposes the use of DNS-like routing infrastructure that can redirect consumers towards the closest providers. Similarly, AGN uses content-centric routers that operate alongside IP to deliver both Interest packets (subscriptions) and Data packets (provisions). Alongside this, both approaches propose new types of content identification as a conceptual replacement for traditional host addresses.

1.2.2 The Case for Content-Centric Networking

Content delivery has emerged as one of the primary uses of the Internet today. This has been driven by recent improvements in access connectivity, as well as the evolution of technologies and business models to adapt to new user demands. A content-intensive application is one that heavily relies on some form of content delivery; examples therefore include e-mail clients, IPTV systems, file sharing applications and software update mechanisms.

Currently, content-intensive applications must implement a content delivery protocol (e.g. HTTP), alongside a mechanism to discover the location of one or more available sources. For instance, a software update management system would likely use a server that provides periodic announcements regarding newly available packages. In such an environment, any client applications would be configured with the location (domain name) of the server, alongside the necessary

protocol support to interact with it. Although this seems simplistic, this creates a wide range of unnecessary complications. This is because the application only desires access to a given item of content; it does not have a vested interest in *where* the content comes from or *how* it arrives. The management of such details is therefore an unnecessary burden for developers. This burden often leads to sub-optimality because developers without a vested interest in content delivery rarely invest time (and money) in improving the process.

Content-centric networking offers an effective solution to these problems by deploying an integrated content discovery and delivery system that can operate behind a single standardised abstraction. Evidently, this approach would offer a simpler mechanism by which applications could access content. However, the introduction of a single integrated system would further allow a range of other optimisations to be performed that are not possible when dealing with multiple diverse location-based solutions. First, performance advantages could be achieved by allowing each item of content to be uniquely addressed at the network-level; exploitations of this including easy caching, replication and request profiling. Alongside this, the current problems of availability and resilience are addressed because it becomes difficult for flash crowds to be directed towards particular hosts. Instead, packets are solely addressed using content identifiers, thereby resulting in requests being forwarded to a variety of potential sources (based on suitable proximity and loading metrics). Last, many security issues could also be resolved as the current approach of securing communication channels (e.g. through SSL) would become obsolete; instead, the content itself could be secured using self-certifying content identifiers based on cryptographic hashing.

1.2.3 Critique of Existing Approaches

Content-centric networking is still a very young research area and, as such, there are a number of weaknesses in existing designs. Within this thesis, three core problems are identified and investigated. These are interoperability, deployability and delivery specialisation.

Interoperability. Most content-centric approaches promote themselves as 'clean-slate' solutions that wish to reform how the Internet operates. Whilst this is an interesting challenge, it is highly undesirable in regard to interoperability with existing content infrastructure. When an application utilises a content-centric abstraction it likely wishes to have the opportunity to interact on a large-scale with any available provider. Unfortunately, however, any users of a current content-centric design would not be able to interoperate with providers offering content using traditional discovery and delivery. This is because current proposals introduce new bespoke infrastructure and protocols that require both provider-side and consumer-side modifications. Consequently, current de-

signs only promote access to content across multiple (controlled) replicas, and not across multiple systems. This observation means that the majority of content in the Internet would not be available to those using content-centric networking.

Deployability. Most existing content-centric networks involve the deployment of a network infrastructure (e.g. [82][93]). This infrastructure largely focuses on the routing of requests based on content identifiers. Solutions such as NetInf [33] also promote the deployment of storage infrastructure in the network. In practice, this makes the uptake of the content-centric paradigm extremely unlikely, as building new infrastructure is both slow and expensive. Further, due to their network-level position, content-centric protocols are susceptible to a lack of support by ISPs. This has been frequently encountered when deploying new network technologies with examples such as quality of service (QoS) and multicast [34]. Evidently, this requirement is therefore related to interoperability, as such support also improves the deployment process (e.g. through backwards compatibility).

These hardware deployment challenges also lead to software deployment problems caused by the need for both applications and operating systems to be adapted. The cost of re-engineering existing applications to use a content-centric paradigm is high; similarly, there is an extremely high risk associated with developing new applications using a content-centric paradigm. Consequently, it is likely that most applications would not utilise any newly deployed content-centric network until it had global support.

Delivery Configuration and Specialisation. At present, the primary defining property of a content-centric network is the separation of location and identifier when requesting content. The purpose of this is to allow the network to route requests based on content rather than hosts. However, as of yet, there is no consideration for the specialisation or adaptation of the delivery procedure to match the requirements of the consumer (or provider). In fact, there is not even the ability to stipulate these requirements as part of the abstraction [54]. This is a wasted opportunity as the abstraction acknowledges the fact that the consumer does not care *where* content is coming from, yet ignores the fact that the consumer similarly does not care *how* the content is being received (as long as it falls within certain requirement bounds, e.g. performance). This therefore allows a huge array of 'behind-the-scenes' adaptation to be performed based on these application-level requirements. Within this thesis, the ability to do this is described as being *delivery-centric*.

Delivery-centricity is a concept that would be manifested through the ability for an application to stipulate delivery requirements when requesting an object. There exists a number of a possible delivery requirements that could be issued

by applications at both the provider's and consumer's side. Currently, these are satisfied through the use of different delivery protocols (e.g. BitTorrent, HTTPS, etc.). However, if a single unified content-centric system were to be defined, the need must be satisfied in a different way. The most evident delivery requirement is performance; this includes things such as throughput, delay and jitter. Due to a lack of network-level support, this is currently performed (if at all) at the application-level using techniques such as multi-source selection [92] and swarming [49]. In a traditional content-centric network, however, this must be handled entirely by the network, as it would not be possible to use these approaches. This is because the consumer would be unable to differentiate between different sources.

Although performance is probably the most important, there are also a number of other delivery requirements that an application might wish to stipulate. Whilst it is possible that performance concerns might be introduced into content-centric designs (e.g. COMET [6]), it is unlikely that this more extended set will be. Collectively, these requirements define the quality of service that an application requires. If all requirements are met by the delivery-centric system, then an application can be considered to be satisfied with the service received from its content provider. A range of potential requirements can feed into this, including,

- *Security:* What degree of security should be employed on the connection (e.g. encryption strength)?
- *Timeliness:* Is the content required immediately? Must it be live or can it be buffered?
- *Anonymity:* Must the request be anonymous?
- *Monetary Cost:* How much is the user/application prepared to pay for the content?
- *Overheads:* What local and remote overhead can be acceptably incurred (e.g. upload bandwidth, memory)?
- *Ethical Policies:* What are the energy costs involved? What policies does the provider have regarding employment practices?

The reason that these requirements cannot be easily satisfied is that the delivery protocol utilised by current content-centric networks is static. It therefore cannot be adapted and configured in the same way that traditional application-level delivery systems can be. Currently, in the simplest case, a consumer might choose not to download a particular item of content from BitTorrent because of anonymity fears; instead, they might choose to use FreeNet [48]. Because consumers cannot individually select their delivery protocol or their preferred

sources, this becomes impossible in a content-centric network. Thus, many powers are removed from the consumer, making it impossible to enforce personal criteria over that of the network policy.

1.3 Research Goals

The key objective of the thesis is to investigate the notion of content-centric networking and to extend its definition to fully exploit the potential of (existing and future) content systems. This objective can be further decomposed into three core research goals,

1. To define an extended notion of a content-centric abstraction encompassing both discovery and delivery, capturing the requirements of (existing and future) heterogeneous content systems

2. To design and implement an end-to-end infrastructure that realises this (new) abstraction in a flexible manner

3. To show that this concept is feasible and can be deployed alongside existing systems in an interoperable way

From these high-level goals, a number of more specific research steps can be derived. First, it is necessary to explore the state-of-the-art in content discovery and delivery systems, to understand how they could be capitalised on using a shared abstraction. This includes a detailed analysis of current content distribution systems to ascertain whether or not the above heterogeneity regarding delivery-centricity actually exists. Using this information, the next step must be to design and implement a system that can provide applications with a new delivery-centric abstraction, whilst building explicit support for deployment and ensuring interoperability with existing (and future) content systems. Last, this design must be evaluated based on the above goals, namely it must be shown that the system offers support for (*i*) a content-centric and delivery-centric abstraction, (*ii*) interoperation with existing (and future) third-party content systems, and (*iii*) explicit deployment support to enable wide-spread uptake.

1.4 Thesis Overview

Chapter 2 provides a background overview of content networking, as well as an overview of technologies that could be used to achieve interoperation between different content systems. Following this is a detailed analysis of related content-centric solutions, looking at their ability to satisfy the key requirements of this thesis. *Chapter 3* then performs a detailed quantitative analysis of delivery system dynamics. This specifically looks at how the abilities of three popular delivery

systems to satisfy performance requirements varies over time. To achieve this, simulations, emulations and real-world measurement studies are used to show that such requirements can only be satisfied using runtime adaptation, thereby making it difficult for applications to provide such support natively.

Chapter 4 proposes a new delivery-centric interface that can be used to abstract applications away from the complexity of handling the previously discovered dynamics. This interface is then realised by a middleware design called Juno. Following this, *Chapter 5* explores the deployment challenges of Juno whilst proposing techniques that can improve both its discovery and delivery aspects.

Chapter 6 then evaluates Juno, looking at its ability to achieve both content-centricity and delivery-centricity. First, the discovery process is evaluated using simulations, before deploying a prototype implementation on a testbed to validate its ability to achieve the performance benefits highlighted in Chapter 3.

Finally, *Chapter 7* highlights the main contributions of the thesis, as well as detailing potential future work. The thesis is then concluded, reviewing how the primary research goals have been achieved.

Chapter 2

Background and Related Work

2.1 Introduction

The previous chapter has introduced the key goals of this thesis, highlighting limitations of existing content-centric networks, alongside important research tasks regarding their future. The central tenet of this thesis is to investigate the potential of extending the existing content-centric abstraction to include the concept of delivery-centricity. An important stage in achieving this goal is therefore understanding the capabilities and operation of existing content systems that might be involved in this process.

This chapter provides a detailed background overview of the principles of content networking, with a focus on understanding the various forms of heterogeneity that can be observed between different systems. This underpins this thesis by providing a state-of-the-art understanding of the paradigms and protocols that can potentially be exploited by a delivery-centric network. Following this, is a survey of potential technologies that can be used for achieving interoperation between these multiple content systems. Last, the related work to this thesis is explored; this involves analysing three closely related systems based on the core goals detailed in Chapter 1.

2.2 Defining Content-Centricity

Content-centric networking (CCN) is a term that has the potential to cover many topics. By its nature, the term can be used to encompass all networks that view content as a central and paramount entity in their operation. This follows a similar definition to Plagemann et. al. [116]. This generic definition, however, could lead to a number of networks being classified as being content-centric. Examples include peer-to-peer systems and the Akamai Content Distribution

Network (CDN) [1]. This section seeks to clarify this definition by exploring the criteria that a system must uphold to be classified as truly content-centric from the perspective of this thesis. To define a content-centric network, this thesis uses three measures. Each of these is now investigated in turn.

A content-centric network should provide a network abstraction that offers a publish/subscribe-like interface. This implicitly creates a location/identifier split, detaching the network abstraction away from the use of any specific location (e.g. IP address). Currently, when an application requires an item of content it is necessary to make a connection to one or more IP locations so that it can be requested using an application-level protocol (e.g. HTTP [14]). This has resulted in the development of a range of discovery techniques to allow applications to ascertain the best location to request a given item of content from. Examples of this include Akamai's DNS redirection infrastructure [1] and Coral's lookup clusters [63]. A content-centric abstraction, however, must implicitly offer this service without the need for the application to provide any details about content location.

Table 2.1 provides an overview of the content-centric API proposed by Demmer et. al. [54]. Such an API would typically be offered by a middleware or operating system. It offers the capability to publish an object (`create`) and then fill that publication with data (`put`). This therefore allows subscribers to discover and access the content by opening a handle to the publication (`open`) then requesting data from it (`get`). Publications are accessed using a handle in a similar way to a TCP connection being accessed using a socket handle. It is important to note, however, that the open command needs only a publication identifier as opposed to a socket's need for a host address and port.

This stipulation is the primary defining factor in the design of a content-centric network. Interestingly, however, it deviates from the standing definition offered by Van Jacobson et. al. as being a "communications architecture built on named data...[it] has no notion of host at its lowest level - a packet address names content, not location" [83]. The important difference is therefore that existing content-centric networks fulfil this criterion based on their underlying functionality, whilst this thesis attempts to fulfil the criteria solely using an abstraction (i.e. it does not necessarily have to be content-centric behind the abstraction). This means that this thesis places no prerequisites on the underlying method of implementation.

A content-centric network should attempt to fulfil the delivery requirements of the consumer(s). Different users and applications have different delivery requirements. Currently, this observation is addressed through the use of a range of divergent (application-level) delivery systems. However, the deployment of a single, unified content-centric network would also need to satisfy this requirement to receive widespread uptake. An application's needs can vary from the

Parameter	Returns	Description
open(pub_id, flags)	handle	Opens a stream to a given publication
close(handle)	void	Closes a stream to a given publication
create(handle, attrs)	Status	Creates a new (empty) publication
stat(handle)	Status, Attrs, Statistics	Provides an up-to-date set of statistics relating to a publication
update(handle, attrs)	Status	Updates a previous publication
get(handle)	Status, Message	Requests data from a publication
put(handle, message)	Status	Puts data into a given publication
seek(handle, seqid)	Status	Moves to a particular location in a publication

Table 2.1: Overview of Content-Centric Networking Interface [54]

way it accesses the content (stored, streamed, interactive) to non-functional requirements such as reliability and security. Within this thesis, a content-centric network that supports the fulfilment of these various on-demand requirements is termed *delivery-centric*.

Delivery-centricity can, generally speaking, be achieved through one of two mechanisms: (*i*) intelligent source selection and (*ii*) protocol adaptation. Intelligent source selection is a discovery process that attempts to locate sources that offer content in a way that is conducive with the requirements of the application; whilst, protocol adaptation involves adapting the behaviour of sources that aren't already operating in a way that is conducive with the requirements. The former offers a passive way in which delivery-centricity can be achieved whilst the latter constitutes a more pro-active approach.

Current content-centric networks do not place a focus on the delivery process. DONA [93], for instance, just performs a point-to-point sequential delivery after a source has been found. LIPSIN [85] takes a push approach and therefore the delivery is defined by the provider. AGN [83] provides a greater degree of control over the delivery with the ability to request the parts of the content (data packets) in different orders. This, however, is essentially a multi-source TCP connection. None of these systems therefore support the stipulation of more diverse requirements relating to issues such as security, anonymity, resilience, monetary cost and performance. Due to this unfulfilled requirement, existing content-centric systems cannot be classified as truly content-centric from the perspective of this thesis. To address this, the term *delivery-centric* is introduced; this refers to a system that fulfils this requirement.

A content-centric network should offer content security based on securing the content itself and not the communications channel. By its nature, a content-

centric network should remain agnostic to any particular location. As such, it is not important to secure any particular channel but, instead, to secure the content received from that channel. This need is increased by the promotion of caching [140] in content-centric networks [33][118], which means it is often impossible to ascertain the original source of the data.

To secure the content it is important for consumers to be able to validate every part that they receives (e.g. each chunk or packet). This must also be implicit to the content without the need to contact remote locations. The most intuitive approach is therefore to generate content naming based on the data, e.g. identify the content using a SHA1 hash of its data. This subsequently allows any consumer to validate the integrity of the data by re-hashing it and comparing it to its identifier. This is the approach taken by current content-centric networking designs.

2.3 Principles of Content Distribution

2.3.1 Overview

The central tenet of this thesis is to explore the potential of deploying a new content-centric abstraction by unifying the resources of existing popular content systems. The purpose of this is two-fold; first, it would allow a practical deployment of content-centricity using the resources and content of existing systems; and, second, it would allow the intelligent selection of different available sources and protocols to allow delivery-centricity to be achieved. The former provides a foundation for building a new content-centric system with access to a wide range of content whilst the latter exploits this foundation to further offer optimisation on behalf of the application. Collectively, these offer a strong motivation for application developers to incorporate content-centric networking in their designs.

As a foundation to this research, it is therefore necessary to (*i*) understand the predominant protocols and paradigms utilised in widely deployed systems, and (*ii*) investigate how their non-functional properties can vary to support delivery-centricity. In its simplest form, content distribution can consist of a point-to-point connection between a consumer and a provider. This, however, can be (and often is) extended to exploit a range of sophisticated technologies that attempt to optimise the delivery process. As such, a content distribution system can be defined as "a networked infrastructure that supports the distribution of content" [116]. At its core, it is necessary for a content distribution system to offer at least two functions:

- *Content Discovery:* The ability for consumers to discover sources of a given item of content, based on some form of identifier

- *Content Delivery:* The ability for consumers to subsequently access the content by transporting it from the remote source(s) to the chosen location

This section now explores the major paradigms used to build these two functions in modern deployed systems.

2.3.2 Content Discovery

Content discovery is the process by which a node discovers a set of available sources for a specified item of content. This is a necessary first step before a content delivery can take place. In a location-oriented network (e.g. IP) this generally involves performing some form of lookup to map a content identifier to a host address. In contrast, in a content-centric network this involves routing a request to an available source. Despite this, it is evident that both mechanisms perform the same essential function. Content discovery mechanisms can be categorised into three main groups,

- *Client-Server:* A logical server maintains a database that can be queried by potential consumers
- *Peer-to-Peer:* A set of cooperating peers collectively maintain a database that they openly provide to each other
- *Decentralised Infrastructure:* A set of cooperating servers maintain a database that can be queried by potential consumers

All of these approaches have been successfully deployed in the Internet and offer various advantages and disadvantages. Each is now discussed through the use of prominent real-world examples.

Client-Server Model

The client-server model is the oldest and simplest distributed systems paradigm available. It involves setting up a single (logical) high capacity node that can serve as a centralised accessible index of information. This index is then contacted by clients that request this information, as shown in Figure 2.1. Such an approach offers a flexible mechanism by which clients can query information using a range of different criteria.

There are many client-server discovery systems deployed in the Internet with most web sites operating in a client-server fashion. For instance, websites such as rapidshareindex.com and rapidmega.info offer the ability to perform keyword searching on the Rapidshare [25] content host. These services operate as servers that maintain a database that contains the mapping of keywords → content identifier, as well as content → location, e.g.,

'animals' → 'Monkey3-12a' → http://rapidshare/files/217935/72B2Y11.jpeg

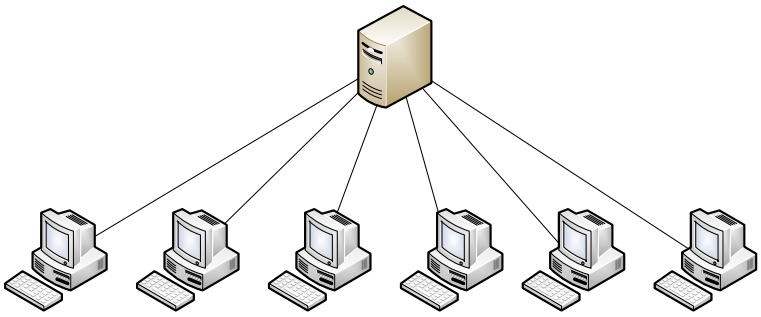


Figure 2.1: The Client-Server Paradigm

The client-server model is highly attractive for a number of reasons. It is fast and simple to deploy. Further, it allows queries to be resolved quickly based on a range of flexible request fields as all data and computation resides at a single location. However, it has significant scalability issues that create a limit on the number of requests it can handle. This means that increases in demand must be responded to with an increase in the amount of provisioned resources. This process is both slow and costly, making the client-server model unattractive to systems with limited resources or large variations in demand.

Peer-to-Peer

The peer-to-peer paradigm has become a popular alternative to the client-server model. Peer-to-peer discovery services operate by sharing and exploiting the resources of the peers utilising the service. As such, peers self organise to cooperatively offer content indexing. There are two primary approaches to building peer-to-peer discovery systems: unstructured and structured. These two approaches are now outlined.

Unstructured discovery overlays were the first widespread deployment of peer-to-peer networking following Napster [23]. The most popular example of this is Gnutella [10], which offers a file sharing service between users. The Gnutella topology is totally unstructured with peers connected to a set of n arbitrary peers, as shown in Figure 2.2. All peers are considered equal and operate in homogeneous roles. When a peer wishes to search the network it forwards a query to all of its neighbours, which in turn forward it to all of their neighbours with a Time to Live (TTL). Any node that receives this query subsequently responds if it possesses a shared file matching the query.

The process of flooding queries to all neighbouring nodes obviously creates significant overhead and makes Gnutella largely unscalable. A number of supe-

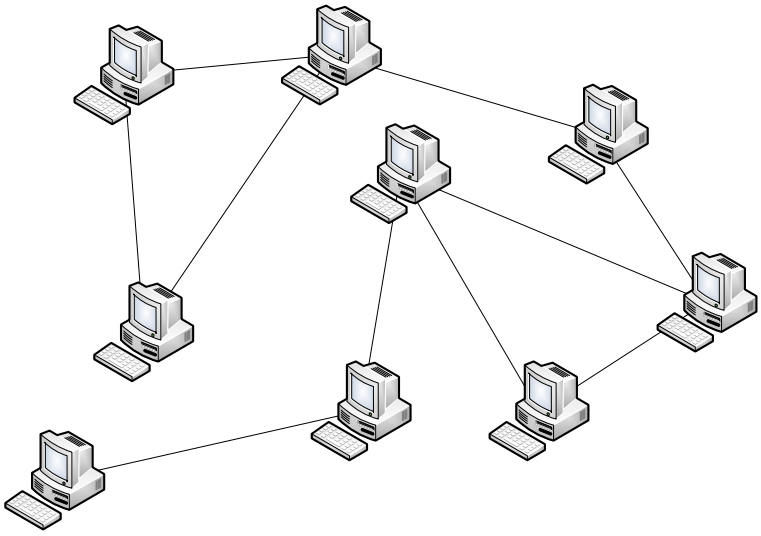


Figure 2.2: An Unstructured Overlay Topology

rior variations therefore also exist, such as Gnutella 0.6 [91] and FastTrack [98], which introduce the concept of *super-nodes*. These are high capacity peers that can solely perform the indexing role for the network, as shown in Figure 2.3. This leaves less reliable peers in a client-like role making the approach more scalable. Such an approach is utilised by popular applications such as Limewire [18] (Gnutella 0.6) and Shareaza [27] (Gnutella2).

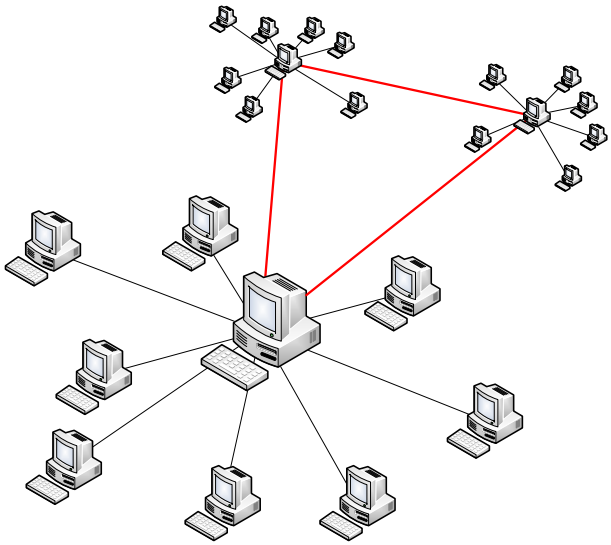


Figure 2.3: A Super-Peer Overlay Topology

Structured discovery overlays are the second type of widely deployed peer-to-peer discovery system with large-scale systems such as Azureus, eMule and Skype all using structured overlays. They were developed to addresses the scalability and reachability issues of unstructured alternatives. Structured discovery overlays offer an efficient and deterministic routing process by which queries can be propagated to the node(s) that are responsible for the search information. Structured overlays encompass any topology with a predefined structure, such as trees (e.g. BATON [84]) and skip lists (e.g. SkipNet [75]). However, of most relevance is the example of Distributed Hash Tables (DHTs) [126]. These allow applications to retrieve information using a hash table abstraction - information can be stored under a given unique identifier and then subsequently retrieved at a later date using the same identifier. Behind this abstraction exists a fully decentralised set of peers that have self organised into a structure that allows Distributed Object Location and Routing (DOLR). In essence, each peer is given a unique identifier; DOLR allows messages to be routed to any member peer by addressing it with this unique identifier. Most currently deployed systems claim that this can be done in $log(N)$ hops, where N is the number of nodes. For instance, Pastry [126] creates an ordered ring topology that is supplemented by a tree topology connecting all the nodes. DHTs have become hugely popular with a number of instances being deployed such as Chord [131] for OpenDHT and KAD [106] for eMule and BitTorrent's decentralised tracker system.

Peer-to-peer content discovery services offer a means by which a large amount of content can be indexed cheaply with little investment by developers. In contrast to client-server models, it also allows the system to scale with an increase in the number of users. This is because an increase in the number of users also results in an increase in the available resources. Despite this, peer-to-peer systems suffer from efficiency and flexibility problems. These arise because it is necessary to distribute data and computation over a large set of nodes. This makes coordination difficult and expensive, thereby restricting the number of tasks that can be performed (e.g. searching based on flexible criteria). Similarly, this need to distribute the indexing process makes the system far slower than client-server counterparts. For instance, in theory, a Pastry DHT would, on average, requires 4 hops to locate an item of content in a 100,000 node network [126], whilst a Gnutella query can contact over 400 nodes based on its default settings [125]. In practice, these values are generally higher due to routing failures and stale information; Azureus' KAD, for instance, has been shown to contact over 50 nodes during a lookup, often taking well over a minute [61]. Alternative, more aggressive, lookup algorithms improve performance, however, lookup times are in the order of seconds, rather than milliseconds. Peer-to-peer discovery systems therefore often struggle to offer the performance of client-server systems in the real-world.

Decentralised Infrastructure

Decentralised infrastructure has been in use since the inception of the Internet. In fact, the Internet is by its nature a decentralised infrastructure, i.e. a collection of routers and end hosts working in cooperation. In essence, it is a combination of both client-server and peer-to-peer principles to exploit the benefits of both. It consists of a set of autonomous items of infrastructure; these are usually servers, however, they can also be specialised hardware such as routers. This infrastructure is then connected in a peer-to-peer manner using an overlay network.

There are two types of decentralised infrastructure. The first is open infrastructure that is spread over many different autonomous domains and is open to the convenient addition of new resources. This generally requires some form of protocol standardisation and, as such, is limited to older protocols and services. The second is closed infrastructure that is controlled by one or more private operators. This is usually owned by a small number of commercial entities and therefore does not require the same type of protocol standardisation as open infrastructure. Instead, it is possible for a company to deploy its own protocols. These two instances are now discussed.

Open infrastructure is defined by a relatively open policy of admission, alongside the use of standardised open protocols allowing anybody to potentially contribute. The most prominent example of a public decentralised discovery infrastructure is that of the Domain Name System (DNS). This consists of a hierarchical collection of DNS servers that operate to resolve domain names to IP addresses. This is analogous to the mapping of content identifiers to source locations. In fact, a prominent content-centric network, DONA [93] (c.f. page 32), considers itself similar to a DNS service. DNS servers are interconnected using a tree topology based on the structure of domain names. When a client wishes to perform a lookup using the DNS system, it queries its local DNS server, which traverses the query through the tree until an entry is found. An alternative prominent example is that of content-based routing designs such as LIPSIN [85] and CBCB [44], which allow content requests to be directly routed to sources. These build decentralised routing infrastructure that maintain content routing tables; using these, routers forward requests through the network to the 'nearest' source of content.

Closed infrastructure is defined by relatively closed ownership that limits management of the infrastructure to a controlled few. Generally, it is initiated by an individual organisation that wishes to replicate functionality (e.g. lookups) for increased performance and fault tolerance. Due to the specific nature of the infrastructure it also offers the potential to support a richer set of functionality that is finely tuned to the organisation's needs [137]. Content intensive websites such as YouTube [32] often build private infrastructure due to their access to large amounts of resources [94]. This gives them the flexibility to define a large

number of important factors; most importantly, it allows them to select their own peering points to reduce costs and optimise performance.

Decentralised infrastructure combines many of the benefits of both client-server and peer-to-peer designs. By distributing the load over multiple servers it increases scalability and resilience; further, by using dedicated infrastructure it becomes possible to increase control and maintain higher levels of performance. The primary limitation is therefore the cost and complexity of deploying the infrastructure. Most examples of decentralised discovery infrastructure are either legacy systems that have evolved alongside the Internet (e.g. DNS) or, alternatively, systems that have been deployed by extremely well provisioned companies such as Google and Microsoft. Subsequently, open infrastructure is extremely slow to deploy due to the need for protocol standardisation and cooperation between multiple organisations; whilst, the extreme cost of deploying closed infrastructure is prohibitory for most organisations.

2.3.3 Content Delivery

Content delivery is the process by which an item of content is transferred from one or more providers to a consumer. This is an extremely important step as it generally consumes the greatest amount of time and resources. Before it can take place, some form of content discovery must have taken place to ascertain a set of locations. A delivery mechanism must therefore offer the interface,

$$\mathsf{get}(\mathsf{Locations}, \mathsf{ContentID}) \rightarrow \mathsf{Content}$$

It is important to note that this type of function call will not be directly invoked by an application when operating in a content-centric manner. This is because the method signature uses a location reference, thereby invalidating the requirements in Section 2.2. Instead, an intermediate interface would exist that interacts first with the discovery system before passing the location information to the delivery service.

The content can be returned to the caller in whatever form desired (e.g. file reference, data stream). The there are two primary types of content delivery; *stored delivery* involves viewing the content as a discrete entity that must be downloaded in its entirety before it becomes available to the application. An example of this is downloading a software package, which cannot be compiled until all data has been received. The second type is *streamed delivery* which views the content as a progressive (potentially infinitive) stream of data. With such a paradigm, the stream generally can be immediately accessed without waiting for its complete transfer. An example of this is streaming a video, which allows users to begin to watch it as it is received.

From an infrastructural perspective, content delivery technologies can be cat-

egorised into three main groups. They each provide a generic approach by which content can be transferred from one or more providers to a consumer in both stored and streamed manners,

- *Client-Server:* A server stores a copy of the content which is remotely requested by clients
- *Peer-to-Peer:* A set of cooperating peers share and distribute an item of content amongst themselves
- *Decentralised Infrastructure:* A set of servers cooperate to best deliver the content

Each of these paradigms has been successfully deployed in the Internet through a number of different popular delivery systems. These delivery systems offer a range of access mechanisms, such as content push/pull and publish/subscribe. This section now explores these three approaches through the use of prominent examples.

Client-Server

The simplest way in which a provider can transfer content to a consumer is to utilise a client-server model. Within such a model, a single (logical) high capacity node stores a copy of the published content. Any user that wishes to access content must therefore connect to this server and download it. In practice, however, many client-server systems actually operate with server farms, which consist of multiple servers co-located (e.g. Rapidshare [35]).

Popular examples of client-server distribution systems are the Hyper-Text Transfer Protocol (HTTP) [14] for stored content delivery and the Real-Time Transfer Protocol (RTP) [26] for streaming. When using HTTP, for instance, a client first discovers a location of a desired item of content. This location is identified using a Uniform Resource Locator (URL), which contains the protocol, IP address and remote path of the content, e.g. http://www.lancs.ac.uk/file.html. This URL is then used to access the content; first, the client makes a TCP connection to the server (by default on port 80); following this, the client sends a GET request using the HTTP protocol that stipulates the remote path of the desired content (e.g. /file.html). The server then responds by sending a data stream of the content over the TCP connection.

The advantages and limitations of client-server content delivery are similar to those of client-server content discovery. They are, however, exacerbated by the resource intensive nature of transferring content. A client-server delivery system is fast and simple to setup with the ability to maintain high degrees of provider control. The primarily limitations, however, are cost and scalability; when demand increases beyond the currently provisioned resources, it is necessary to

perform costly upgrades to the server as well as its network connection. These problems still remain even when operating server farms as they simply serve to increase the resources by a further finite level. Last, client-server models are also heavily restricted by their uni-sited nature as they are highly vulnerable to network failures; further, this means they cannot adapt to variations in geographic demand. This is particular crucial when considered the delay-sensitive nature of TCP/IP networks [38]. It is these vulnerabilities that make client-server delivery systems highly susceptible to security attacks, such as denials of service (DoS).

Peer-to-Peer

The peer-to-peer paradigm has become hugely popular for distributing content. During its early deployment this was largely attributable to the widespread availability of copyrighted content. However, increases in access link capacities have made peer-to-peer content distribution highly effective and hugely scalable.

Early peer-to-peer file sharing applications such as Gnutella [91] utilised very simple distribution mechanisms that involved finding individual sources and downloading the content from them in a client-server fashion. Unfortunately, this approach suffers greatly from the often limited resources available at the chosen 'server' peer. This is exacerbated further by possible churn that can result in download failure. To remedy this, simple multi-source downloads were developed (e.g. Limewire [18]). These made connections to multiple peers to exploit their resources and aggregated reliability. This generally involved splitting the file into ranges and then requesting each range from a different source. This was largely effective, however, the emergence of a culture of free-riding meant that often such approaches would fail [78]. This is analogous to the tragedy of the commons [87] in which users attempt to consume resources without contributing them in return. In response to this, systems were developed that integrated the necessary incentive mechanisms into the core of the distribution protocols. The most prominent example of this is BitTorrent [49], which has been measured as contributing over 66% of all peer-to-peer traffic [128]. Unlike previous distribution mechanisms, BitTorrent employs a direct reciprocation incentive scheme called tit-for-tat (TFT) [62]. Peers separate the file they are downloading into small parts called *chunks* (default 256 KB), which are then requested from other peers possessing that part of the file. The decision to provide a remote node with a chunk is made based on the performance that has been received from that node. Therefore, it becomes difficult for nodes to download chunks unless they are also prepared to upload chunks to others [40]. This approach has been hugely successful and has acted as a foundation for a number of further designs [24][148].

It is also possible to stream content using peer-to-peer; early approaches used tree-based topologies, e.g. NICE [36] and ZigZag [136]. These built a tree-based overlay structure connecting the consumers so that receivers also forward

the stream to their children. Unfortunately, tree structures often result in long delays between the source and the receiver; further, they also are heavily affected by churn. To address this, researchers began looking into building BitTorrent-like protocols that operate using chunk exchange but in a streamed manner. Popular examples of this are CoolStreaming [148] and PRIME [104]. These both operate using sliding windows that stipulate ranges of the content that a peer is interested in (e.g. in windows of 30 seconds). Chunks nearer to the peer's playback point are therefore considered more important than ones that are further away.

Peer-to-peer content delivery offers a powerful solution for providers wishing to perform large-scale delivery without the associated costs of using a client-server model or paying commercial CDNs such as Akamai. This is because the clients that consume resources also contribute resources, thereby pushing the costs of the delivery onto the tier-2 and 3 networks [46], as shown in Figure 2.4. It can be seen that the content is only provided from the original source once; following this, the content is only passed through tier-2 and 3 networks.

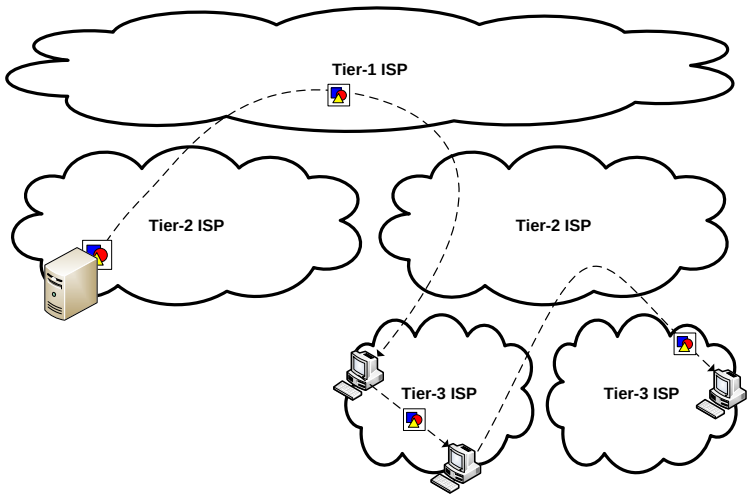


Figure 2.4: Peer-to-Peer Content Distribution over Different ISPs

Despite these benefits for the provider, peer-to-peer systems often struggle to compete with the over-provisioned resources of infrastructural alternatives. Many research papers have claimed through simulation that the opposite is the case [89], however, in practice this is not always true. Experiments have shown that well-provisioned client-server providers can offer vastly superior performance when compared to peer-to-peer systems. Atoniades et. al. [35] investigated the performance when accessing content using both BitTorrent and the client-server content provider Rapidshare. This was done by downloading a random set of 38 files using both systems. The average download rate using BitTorrent was 600 Kbps with only the top 10% achieving over 2.4 Mbps. In contrast, Rapidshare

offered in excess of 8 Mbps for 50% of users, with a notable percentage gaining over 20 Mbps. Similarly, the reliance that peer-to-peer systems have on user behaviour means that content delivery systems often suffer from large variations in performance and reliability when compared to infrastructural alternatives. In the most extreme of cases, peer-to-peer delivery systems sometimes are incapable of distributing certain content [73][88].

Decentralised Infrastructure

Decentralised infrastructure consists of the coordinated use of multiple servers to best distribute an item of content. To achieve this, a set of servers are placed at various strategic points in the network and loaded with content that they are responsible for providing to their region. This content is often dynamically defined and can vary based on observed demand trends. When a consumer desires an item of content, it is redirected towards the server that can best provide it. The predominant metric driving this is locality, as low delay usually results in superior performance and reduced economic cost for the underlying network infrastructure.

Generally, two types of decentralised delivery infrastructure exist. *Public* infrastructure that offers its servers for (paid) third party utilisation and *closed* infrastructure that is owned and used by a single controlled organisation. Note that public infrastructure is different to *open* infrastructure in that it is not possible for third parties to integrate their own resources.

Popular examples of public infrastructure include Akamai [1] and Mirror Image [22]. These services place their customers' content onto their content servers and subsequently allow customers to redirect requests into their network. Closed infrastructure, in contrast, is owned and managed by an individual organisation and is solely used for their needs. This can often be seen in larger-scale web sites (such as YouTube [32]), which distribute their servers to improve performance. Regardless of the model, when a consumer wishes to access content in any decentralised infrastructure, the discovery process selects one or more servers for the consumer to access the content from. The sources that are returned are optimised for certain desirable parameters such as distance and cost. This is achieved by resolving the requester's IP address based on one of these characteristics; this is trivial using GeoIP services and BGP maps. Subsequently the returned sources are then utilised by the consumer to access the content in a traditional client-server manner, usually using a protocol such as HTTP.

The benefits of using decentralised delivery infrastructure can be significant. First, it allows greater performance as it becomes possible to (*i*) serve clients from geographically closer servers, and (*ii*) load balance across multiple servers. It also improves fault tolerance significantly as there no longer exists a single point of failure. Through the use of public infrastructure it also becomes possible

to address the scalability limitations of traditional client-server models. This is achieved by sharing the resources of the public infrastructure between many different content providers in the hope that only a subset of those providers will need it at any given time. This therefore allows providers that only occasionally endure spikes in demand to quickly scale their resources for a short period.

The primary limitation of using decentralised infrastructure is its heavy cost. Unfortunately, building private infrastructure is unfeasible for all but the richest providers; examples of companies that have done this are Google and Microsoft, which have been observed to build far-reaching wide area networks covering much of the U.S as well as Europe, Asia and South America [66]. An alternative is to simply purchase the use of existing infrastructure such as Akamai. This approach has the advantages of reduced maintenance for content providers as well as improved performance (through intelligent edge server selection). These advantages, however, are expensive with an average charge of between $0.3-0.6 per GB [122]. Assuming a relatively low-use provider that has a mean average of 10 Mbps traffic, the costs per month would reach approximately $1,000. This cost would obviously increase dramatically for large scale websites that reach Gbps traffic. As such, many providers without strong financial backing are unable to afford to use commercial CDNs, leading to the use of cheaper solutions [42].

2.3.4 Summary

This section has detailed the principles and paradigms used in modern content distribution over the Internet. Two separate bodies of functionality have been investigated: discovery and delivery. Content discovery refers to the process by which a consumer can resolve a content identifier to a set of content locations. Content delivery is then the subsequent process by which a consumer can utilise those locations to access the content.

Current content discovery and delivery systems can be categorised as either client-server, peer-to-peer or decentralised infrastructure. Table 2.2 provides a summary of the different paradigms and examples of popular systems. All three approaches are widely deployed in the Internet with large numbers of users. Importantly, it can be observed that there is a plethora of diverse content-based mechanisms that are used today.

From this study, it is clearly evident that a large degree of heterogeneity exists between this multitude of content systems. This heterogeneity has been shown to come in a number of forms, including performance, overhead, reliability and security. It can therefore be derived that a significant benefit could be gained from integrating access to these systems, in terms of both content and resource availability. Further, however, it is also evident that sufficient heterogeneity exists to enable delivery-centricity through the informed selection of different providers and protocols. Consequently, the primary challenge that can be concluded from

Paradigm	Examples	Pros and Cons
Client-Server	HTTP RTP SQL	+ Good content management + High performance possible + Predictable performance − Not optimised for divergent consumers − Not scalable − Expensive − Inflexible to demand
Peer-to-peer	Gnutella, BitTorrent, ZigZag	+ Highly Scalable + Inexpensive + Resilient − Complicated to build − Poor content management − Slow content querying − Unpredictable performance − Higher overhead for consumers
Decentralised Infrastructure	Akamai, DNS	+ Good content management + Relatively scalable + Very high performance + Resilient − Extremely expensive − Highly complicated to deploy and maintain

Table 2.2: Overview of Content Discovery and Delivery Paradigms

this study is how a future content-centric network could (*i*) unify and integrate the different protocols and addressing schemes used by each provider to exploit these resources, and (*ii*) make informed decisions as to which provider/protocol best fulfils a given set of delivery-centric requirements.

2.4 Principles of Networked System Interoperation

2.4.1 Overview

The previous section has explored the diversity of currently deployed content systems to highlight their trade-offs in terms of both providers and consumers. A key conclusion from this study is that the exploitation of this diversity can contribute heavily towards both content and resource availability, as well as delivery-centricity. However, as discussed in Chapter 1, to gain this, it is necessary to achieve interoperation between the different protocols. This section explores three common approaches taken to interoperation between multiple networked

systems,

- *Protocol Standardisation:* This involves designing a standard protocol that is expected to replace all previous protocols and be used globally
- *Protocol Bridging:* This involves using an intermediary to convert protocol messages between two or more incompatible systems
- *Interoperability Middleware:* This involves placing a software layer between the network and the system to adapt it somehow

This section investigates the background to these three approaches; as such, the basic principles are explored before outlining prominent examples and analysis.

2.4.2 Protocol Standardisation and Uptake

The most obvious manner in which a content network can achieve interoperation is through protocol standardisation and large-scale uptake. This, for instance, has been performed with protocols such as HTTP [14] and BitTorrent [49]. It involves a protocol being defined and then deployed with the acceptance of (ideally) all parties. Therefore, in the ideal world this is the optimal approach to interoperation between distributed systems.

Many protocols such as HTTP [14] and RTP [26] have undergone traditional protocol standardisation through bodies such as OASIS, the world wide web Consortium and the IETF. This involves an extensive review process that starts with issuing a Request for Comments (RFC) document that precisely details the workings of the protocol. After many iterations the protocol is formally standardised, allowing developers to confidently build an implementation. In contrast to this, protocols such as BitTorrent and Gnutella began as non-standardised protocols that gained increasing popularity until becoming de-facto standards. Neither approach, however, supports interoperation between different protocols; instead, it simply allows interoperation between different implementations of a single protocol.

A different approach to protocol standardisation is therefore to attempt to persuade system administrators/developers to cease using their individual protocols and replace them with a new standard to enable interoperation. Depending on the diversity of the existing protocols this might have to be performed without any backwards compatibility. A prominent example of this form of protocol standardisation is the introduction of new inter-domain routing protocols. As new variations of the Border Gateway Protocol (BGP) [123] are developed (currently version 4), it becomes necessary for domains to install new software . This is generally a process that must be led from the top by the more powerful ISPs. Although this has proved feasible in this field, it is unlikely that it is applicable to most other areas. This is because inter-domain routing is already a global process

that operates in an open manner that is fundamental for the continuation of the Internet. In contrast, content distribution systems often operate as independent organisations that do not require the assistance of others. As such, the existing relationships that ISPs have, do not exist between content distribution systems. This is exacerbated by commercial competition observed between content networks; for instance, what benefits would Akamai gain through integrating their protocols with other content distribution systems?

Another variation is to define a new protocol that allows existing systems (using their own protocols) to interact. This treats each individual system as a black box, allowing each instance to interact through the protocol. A pertinent example of this comes from the IETF Content Distribution Internetworking (CDI) workgroup, as detailed in RFC 3750 [53]. This has investigated the potential of interconnecting multiple content delivery networks. The aim of this is to design a protocol to allow different CDNs to communicate with each other, to share resources. To achieve this, the Content Network Advertisement Protocol (CNAP) [43] has been defined, allowing CDNs to advertise content that they possess to each other alongside information relating to properties such as topology, latency and geography. CDN nodes can therefore continue to use their own propriety protocols whilst using CNAP to communicate with other CDNs. A similar approach is also taken in [137], which defines a further CDI architecture.

The standardisation and deployment of protocols such as CNAP has been feasible due to their application-level nature. They therefore support progressive deployment that does not require modification to intermediate infrastructure. This is one of the reasons for the rapid and widespread uptake of peer-to-peer systems such as BitTorrent. In contrast, lower levels protocols such as Ethernet have far stricter deployment policies as progressive deployment is often complicated. This is even more difficult for layer 3 protocols such as IPv6 that require global uptake [58]. This is exemplified by the deployment difficulties of network layer multicast and Quality of Service (QoS) due to the reluctance of intermediate networks (e.g. Tier-1 ISPs) [34]. Therefore, protocol standardisation can generally only be used to solve deployment issues when the support of third parties (e.g. providers, ISPs) is not required. This can be seen in Figure 2.5, in which the clients wish to subscribe to the multicast stream provided by the server but cannot as it must traverse a network that does not support the multicast protocol.

2.4.3 Protocol Bridging

A protocol bridge is an entity that acts as a mediator between two (or more) parties that communicate using different protocols. A bridge is therefore required to be aware of both protocols and to be able to perform the necessary mappings between the two. This is shown in Figure 2.6 with a client and server interacting with each other despite the fact that they are utilising different underlying

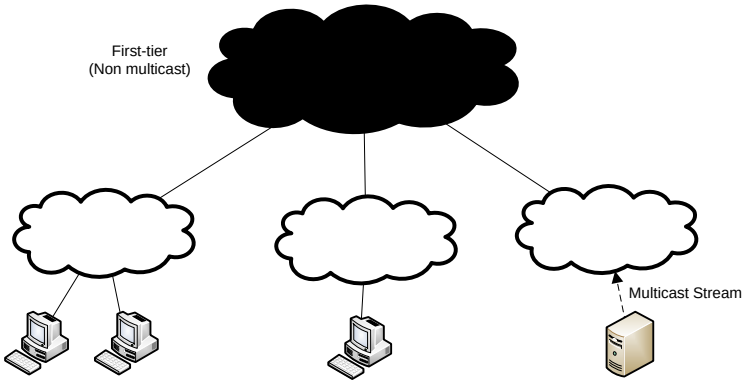


Figure 2.5: A Multicast Deployment Problem; the black cloud does not support the multicast protocol, whilst the white clouds do

protocols. To allow a new protocol to interoperate with existing protocols, such an approach therefore involves the new protocol being simultaneously deployed with a bridge that can convert its interactions with existing systems. This can take place within a local software layer or, alternatively, at a remote point (e.g. a router). This simple approach has been used extensively in the past as it offers a simple way in which incompatible systems can interoperate without modification. This is especially helpful when interacting with multiple parties that operate outside of a single domain (e.g. two companies).

Prominent examples of protocol bridging are network bridges and middleware bridges. A network bridge generally operates on the edge between two different networks and converts layer 2 and 3 protocols (e.g. IPv4 → IPv6 [100]). There also exists higher level bridges for interoperable services; for instance, Multi Protocol Discovery and Access (MDSA) [121] is a service discovery bridge that allows applications using different service discovery protocols to interact. The Object Management Group has also created the DCOM/CORBA interworking specification that provides the necessary mappings between DCOM and CORBA [70]. Similarly, SOAP2CORBA [29] provides the same functionality for bridging SOAP and CORBA applications.

A bridge can also be considered as a type tunnel; tunnelling has been used extensively in lower level networking deployments that require intermediate support for a protocol. Unlike bridging, however, tunnelling does not intend to achieve interoperability between the two end hosts but, instead, to achieve interoperability with the two end hosts and the intermediate network, e.g. 6to4 and ISATAP [58]. Each end of the tunnel therefore utilises the same protocol but hides it from any intermediate infrastructure; this is usually done by packaging messages within the payload of existing compatible messages (e.g. IP packets).

Protocol bridging is attractive due to its simplicity. It also offers a means by

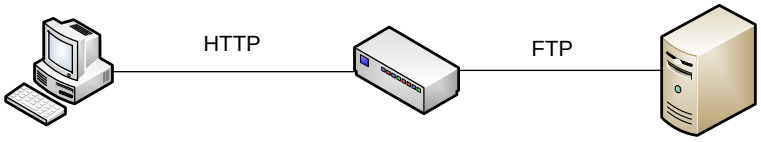


Figure 2.6: A HTTP → FTP Protocol Bridge

which a system can elegantly extract interoperability functionality and place it in an external entity. Importantly, this can even take place without the awareness of either party by transparently re-directing output and input through the bridge. There are, however, also a number of limitations. Most importantly, two protocols can only be bridged in a scalable manner if they possess a core set of synonymous protocol messages, therefore allowing the bridge to operate in a stateless manner. If two protocols perform the same (or similar) high level function but in totally different ways, the process of bridging often becomes difficult because the bridge is required to maintain state for both protocols.

When dealing with a content distribution system there are also heavy resource requirements if it becomes necessary to channel data through the bridge. This can be mitigated if the control and data planes can be separated to allow the data transfer to operate externally to the bridge. This is most evident in a protocol such as FTP, which utilises entirely separate TCP connections for the two planes. In contrast, a BitTorrent mechanism would need constant bridging as it uses protocol messages to request each chunk. When converting BitTorrent to HTTP, a bridge would therefore need to request each chunk individually (in order) before passing them to the other party over the HTTP connection, as shown in Figure 2.7. This makes bridging such protocols complicated due to the high level of functionality required, as well as the frequent memory copies.

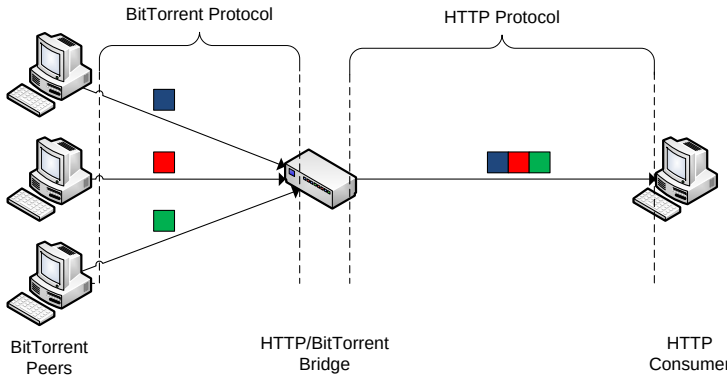


Figure 2.7: A HTTP → BitTorrent Protocol Bridge Converting Chunks Received from a BitTorrent Swarm into a HTTP Data Stream

These problems are further exacerbated by the need for N^2 bridges to be fully interoperable, where N is the number of protocols. It further can require application modification so that messages are redirected towards the bridge. Collectively, this makes the bridge a dangerous point of failure that can leave any dependent applications non-functioning.

2.4.4 Interoperability Middleware

Middleware has emerged as a vital component in distributed systems; it can be defined as,

> *"A layer of software residing on every machine, sitting between the underlying operating system and the distributed applications, whose purpose is to mask the heterogeneity of the cooperating platforms and provide a simple, consistent and integrated distributed programming environment"* [50]

In essence, middleware resides as an abstract constant that protects applications from underlying heterogeneity. This heterogeneity comes in many forms; popular examples include programming languages [51], network infrastructure [69] and mobility [68]. Middleware therefore deals with a variety of important deployment complexities for application developers.

Middleware is underpinned by the principle of abstraction; this is the process of defining *what* a function does rather than *how* it does it. This is realised by developing interfaces that present a set of methods that allow a body of functionality to be interacted with regardless of how it is implemented. Middleware exploits this to bind distributed applications together without the need for each instance of the application to understand the underlying operations taking place behind the abstraction. A popular example of this is the Common Object Request Broker Architecture (CORBA) [8]; this middleware abstracts the application away from both the location and programming language of a method call. As such, it allows applications written in different languages and at different locations to seamlessly interact, therefore handling language and location heterogeneity. Similar examples include R-OSGi [124] and J-Orchestra [135], which offer the ability to transparently partition and distribute an application over multiple locations.

One particular limitation of middleware such as CORBA is the need for dual-sided deployment. Therefore, all parties involved are required to host the middleware so that it can handle interoperability on the application's behalf. This therefore creates similar problems to that of protocol standardisation, i.e. all systems need to install and utilise a compatible middleware. To remedy this, uni-sided middleware have also been pioneered. A prominent example of this is ReMMoC [68], which offers transparent client-side support for interoperation with diverse service implementations. ReMMoC achieves this by adapting its underlying im-

plementation to support whatever service protocols are available in its current environment. This is done transparently without the application's involvement. As such, an application can simply interact with ReMMoC's standardised service abstraction without the need to handle the complexity of operating with different service protocols. Importantly, this occurs solely at the client-side without the need for server-side support. Therefore, the deployment can take place incrementally without support from existing infrastructure. A similar (albeit less sophisticated) approach is also taken by a limited set of peer-to-peer applications such as Shareaza [27] and TrustyFiles [31], which provide support for connecting to multiple peer-to-peer networks such as Gnutella2, BitTorrent, eMule etc.

To build an interoperability middleware it is necessary to implement an abstract-to-concrete mapping so that applications can issue abstract requests that are converted into system-specific requests. This is therefore a transformation function similar to that found in a protocol bridge. However, the primary difference is that neither party continues to operate using a particular protocol. Instead, they begin to use a new abstraction that protects them from the underlying nature of the protocol used. This means that extra functionality can potentially be added to the abstraction to enable the utilisation of a wider range of protocols. This clearly makes the approach non-transparent, although, this can be considered acceptable for a paradigm such as content-centric networking, which requires the introduction of a new network abstraction anyway.

Beyond possible deployment difficulties, the key limitation of using interoperability middleware is that of overhead. This arises from the need to perform abstract-to-concrete mappings, as well as handling potentially memory-intensive operations. For instance, using ReMMoC's abstract interface creates a 54% decrease when being mapped to CORBA method calls and a 11% decrease when mapped to SOAP [28] method calls (for a null operation). Clearly, the process of interoperation comes at a cost that must therefore be traded-off against the overall system requirements.

2.4.5 Summary

This section has detailed three important approaches to achieving interoperation between different networked systems: standardisation, bridging and middleware. These take systems with the same (or similar) functional goals and offer some form of additional support to allow them to interact in either a passive or active manner. Table 2.3 provides an overview of the properties associated with the mechanisms investigated.

From this study, it is evident that deploying systems in an interoperable manner is an important issue. It has been shown that there are a set of proven approaches that have looked at similar goals in different domains. However, only limited work has been performed in the area of content distribution and therefore

this background offers a platform over which interoperability between content-centric systems can be explored. Most closely related is the existence of content inter-networking protocols that allow CDNs to interact with each other. However, these do not allow consumers to interact with different content protocols, nor do they allow interactions to occur in a transparent manner. These principles must therefore be built upon to fulfil the requirements of a content-centric network based on interoperation.

Paradigm	Examples	Pros and Cons
Standardisation	Akamai, HTTP, DNS, SOAP	+ High performance + Simple − Not transparent − Often not feasible − Requires multi-party cooperation − Often requires external third party support (e.g. ISPs)
Protocol Bridging	MDSA, ISATAP	+ Transparent to applications + Simple − Expensive − Bottleneck − Difficult to deploy − Not scalable for some protocols − Not always possible
Middleware	ReMMoC, CORBA	+ Eases application development + Doesn't require network deployment + Can be uni-sided (doesn't require multi-party support) + Flexible due to operating within software − Requires application awareness − Increases end-host resource requirements

Table 2.3: Overview of Approaches to Interoperability

2.5 Related Work

2.5.1 Overview

The previous sections have investigated a range of existing content discovery and delivery systems as well as common paradigms that could be used to gain interoperability between these systems. The approaches discussed in these sections have shaped heavily the formation of most content-centric networking solutions.

The purpose of this section is now to explore the related work to content-centric networking, seeing how these principles are utilised in such designs. From the outset, it can be said that no current content-centric designs offers interoperability due to their clean-slate nature. This means that such systems cannot interoperate with each other, or any existing content providers/consumers at the protocol-level. Therefore, in relation to the primary goals of this thesis, it is important to inspect this related work based on two remaining criteria,

- *Deployability:* Is it feasible to deploy this system in the current Internet? What are the costs?
- *Delivery-Centricity:* What is the support for expressing and satisfying diverse and extensible delivery requirement?

To do this, three popular content systems have been selected. The first is DONA [93], which is a tree-based infrastructural content-centric network proposed in 2007. It promotes the replacement of the current domain name → location mapping, with a unique content name → location mapping instead. The second system looked at is AGN [83], which was proposed in 2009 as a content-centric networking solution based on deploying new routers that can operate above a range of connection-less protocols (including IP and UDP). Finally, the Akamai Content Distribution Network (CDN) [1] is investigated; this is not a content-centric system as it does not utilise content-centric addressing, however, it is currently the largest and most powerful content distribution organisation and, as such, shares many similar goals to the other systems.

2.5.2 Data-Oriented Networking Architecture (DONA)

Design

The Data-Oriented Networking Architecture (DONA) [93] is one of the first systems to emerge that labels itself as 'content-centric'*. It is built as a replacement to the existing Domain Name System (DNS) infrastructure. Its purpose is therefore to allow location resolution for named content. Unlike alternate designs, it does not aim to install content-centric routing at the network layer; instead, it aims to deploy indirection infrastructure (similar to i3 [130]) at the application layer. This involves the distribution of servers capable of routing requests over various autonomous systems.

It operates using two primitives: REGISTER and FIND. When a node wishes to publish an item of content, it sends a REGISTER message to the DONA infrastructure, which registers the particular node as being capable of providing the item of content. When a node wishes to access an item of content it then

*This term is synonymous with data-oriented

sends a FIND message to the DONA infrastructure which, in turn, returns the 'nearest' instance of the content. In essence, this is therefore an anycast service that routes based on proximity metrics.

For DONA to work correctly, it is obvious that a new naming mechanism must be employed to uniquely identify content. Content identifiers in DONA are organised around principals; these are providers that are responsible for the content. As such, each item of content is associated with a given principal. Each principal is associated with a public-private key pair that uniquely identifies it. Therefore, content names are of the form, $P : L$ where P is the cryptographic hash of the principal's public key and L is a label that identifies the content (chosen by the principal). Naming in DONA can therefore be considered to be structured as a two-tier hierarchy.

The next question is how DONA performs routing. DONA requires that each domain installs a DNS-like server termed a *Resolution Handler* (RH). RHs are responsible for maintaining the entirety of DONA's redirection service. They are linked together in a tree structure that represents the BGP topology of the underlying network; finer grained topologies can also be built (e.g. departments in a university). Each domain registers all of its local content with its Resolution Handler. This information is then passed on to the RH's parents and peers in the tree if (i) the content has not been encountered before, or (ii) the new registration is 'closer' than the existing one. The neighbouring RHs then decide whether to forward the information further depending on local policy (e.g. depending on whether they are prepared to serve as a transit for the content). This process is outlined in Figure 2.8 with the client (on the right) publishing an item of content; the solid arrows show how this information is propagated. Evidently, the broadcast nature of this propagation results in huge content routing tables that must be maintained in the network, thereby creating significant scalability issues. This problem is exacerbated further when considering any degree of churn that results in previous REGISTER messages becoming invalid. When a REGISTER message is received by a RH it is first authenticated by issuing a nonce to the originator so that it can be signed with P's private key. If this is valid, the RH then signs the message so that it can be passed to other RHs without the repeated need for verification.

When a FIND is generated by a node, it is first passed to its local RH, which checks its *registration table* to ascertain if it possesses a pointer to a RH responsible for a replica. A registration table is a routing table that maps content addresses to a next hop RH. Registration tables contain entries for both P:* as well as individual items of content (P:L). If there is an entry that matches the FIND message, it is passed onto the next hop RH. If there are multiple choices, the closest one is selected based on a (currently undefined) distance metric. If no entry is found, the request is passed up the tree in the hope of locating an

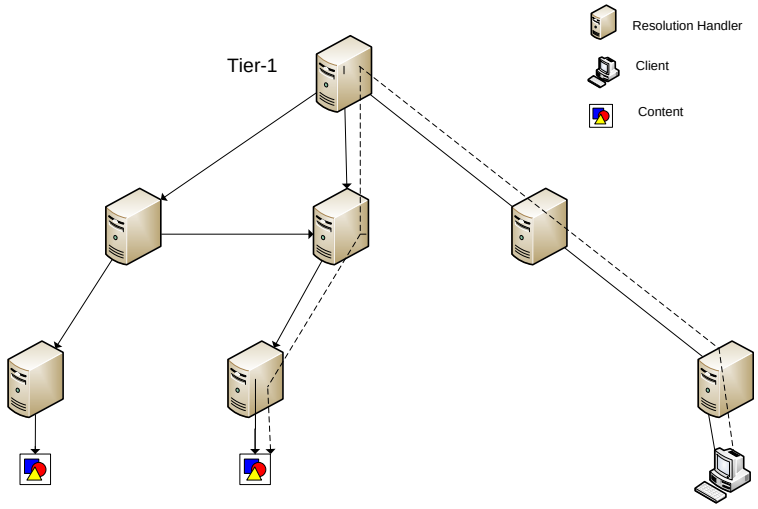


Figure 2.8: DONA REGISTER and FIND Processes; propagation of registration state (solid arrows) and routing of FIND message (dashed arrow).

entry. This, however, is an unguided process as the content addressing does not following the same hierarchical structure of the tree topology. This process can therefore be highly inefficient for less popular content [47]. An example of this is shown in Figure 2.8; the client (on the right) issues a request for an item of content. Its local RH performs a lookup to find the next hop; in this example, it does not have an entry and therefore passes the message up the tree. This process continues until a entry is found at the first-tier. The FIND message then begins to be routed back down the tree to the nearest copy located at the central branch.

Deployment Support

There are two deployment challenges that must be successfully addressed in DONA. The first is dealing with the physical hardware deployment (RHs) whilst the second is achieving wide-spread software up-take.

DONA's functionality is mainly implemented through a distributed set of Resolution Handlers (RHs). These operate in a similar way to DNS servers and therefore exist entirely above the network-layer. As such, deployment does not involve the modification of existing infrastructure but, instead, simply the deployment of new infrastructure. This eases the challenge, however, it still leaves the need to deploy a large body of new hardware. DONA takes a protocol standardisation approach to its system deployment, i.e. all domains must adopt its protocol. This is possible because its application-layer position allows it to enjoy progressive deployment without the need for every autonomous system to

immediately take up the system. A problem with this, however, is that it becomes difficult (if not impossible) for users operating in networks without support to use DONA unless they can find a nearby surrogate network that they can route REGISTER and FIND messages through.

The second issue is that of software modification; it is necessary for operating systems and applications to extend their code to use the content-centric routing. This is a step that will have to be taken by any new content-centric system. DONA, however, does provide support for this by offering the ability to perform protocol bridging between legacy protocols and the new content-centric system. Proxies have been built for bridging HTTP, SIP and RSS. There are few details, however, provided about how these operate and it is not clear how traditional HTTP requests (URLs) can be converted into DONA's content-centric addressing scheme. Currently, therefore, DONA offers little support for interoperating with other content systems.

The above two issues relate to the theoretical deployment issues. However, there are also a number of practical deployment challenges. These centre on the massive increase in overhead related to routing in this manner. For instance, a first-tier RH would likely need to handle around 84,000 registration messages per second (assuming a registration lifetime of two weeks) [93]; if each message were 1 KB, this would generate ≈680 Mbps of traffic, even excluding the transit of any data. Beyond this, it also would be necessary to perform expensive cryptographic operations, which can take in excess of 100 ms. Consequently, this 680 Mbps of traffic would require at least 40 3 Ghz processors to handle the load in real-time. Alongside these bandwidth and processing costs, the non-hierarchical nature of DONA's identifiers would also create a large memory cost. First, assume that each routing table entry would require 42 bytes (40 for the name and 2 for the next-hop). Taking the estimated number of web pages from 2005 [71] and increasing it by an order of magnitude to 10^{11} results in an estimate of approximately 4 TB of storage required for the routing information required at a first-tier ISP (excluding data structure overhead). Considering the expense of memory, this would therefore likely require secondary storage, which has lookup times of ≈2.5 ms. Consequently, to handle a 1 Gbps link (20,000 requests per second), it would be necessary to operate 50 disks in parallel to achieve the necessary throughput. Alternative studies [47] have also found DONA's routing procedure to be non-scalable, with the routing table size of Resolution Handlers increasing linearly with content numbers. Evidently, this can be considered vastly inferior to the logarithmic increase as observed in other structures such as Pastry [126]. The inefficiencies of DONA's design therefore produce significant hardware costs, introducing a direct incentive not to adopt DONA.

Delivery-Centric Support

A delivery-centric system makes the delivery process a primary element of its operation. It provides an application with the ability to stipulate diverse and extensible requirements that must be satisfied by the system. Delivery-centricity can be achieved using either (*i*) intelligent source selection (discovery aspect) or (*ii*) protocol adaptation (delivery aspect). The delivery-centricity of a system can be looked at in two ways; first, through its current support and, second, through its potential support. Currently, DONA cannot be considered to be delivery-centric as its abstraction and protocol simply do not offer any mechanisms to explicitly state delivery requirements. However, there is potential scope to introduce it.

DONA's delivery process operates through the use of FIND messages that request items of content. When a FIND is generated by a consumer, DONA attempts to route it to the 'nearest' copy based on the number of hops from the requesting RH (i.e. the RH that is responsible for the requesting node). When a FIND is received at an end point possessing the content, it automatically initiates the content transfer. This simply involves replying with a standard transport-level message to the requester. Unfortunately, like many aspects of DONA, this is not properly defined. However, this is likely to be a process similar to TCP but with the use of content identifiers rather than the host:port identifiers currently used. Importantly, however, this occurs over the location-based IP network and not through the RH infrastructure (unless some sort of caching is taking place).

It is evident that delivery-centricity in DONA is fuelled by the network's view of a node's requirements rather than that of an individual application. Unsurprisingly, there is no support for differentiation and configuration in the delivery process, as the network uses a fixed set of metrics for routing FIND requests. It is possible for nodes to request that its FIND packet is routed to a more distant source in the situations that it finds previous requests have not been fulfilled. However, it still maintains no control over what source will receive the FIND. The primary goal in DONA is therefore to simply route requests to the nearest source - an identical goal to that of existing systems such as Akamai. It is therefore evident that delivery-centricity can not be achieved in DONA using any form of intelligent source selection.

As previously mentioned, DONA might be able to achieve delivery-centricity through protocol adaptation. This could be achieved by sending delivery requirements directly to the source once it has received the FIND message. Although this is not stipulated in the DONA specification, it would be possible as the transport mechanism is currently very loosely defined. This could involve sending a direct message to the source, requesting the use of a particular protocol or network QoS reservation. One obvious limitation might be that the source selected by the RHs does not support the needs of the consumer or, alternatively,

the runtime conditions prevent it from achieving the requirements. In such a circumstance, the consumer would have to issue the FIND again whilst stipulating the need for a more distant source. Depending on the number of times this cycles for, this could increase the delay dramatically. This also makes the use of multi-sourcing difficult as it would be necessary for multiple FIND messages to be sent out each asking for more and more distant replicas. However, there is currently no functionality available for requesting only a subset of the content. DONA makes heavy use of identifiers for such tasks and therefore a unique content identifier would have to exist for each chunk of data being requested from the different sources. For instance, if a file were to be separated into chunks in a similar way to BitTorrent, an identifier would have to be generated for each one and then separately requested, creating a even greater burden on the routing infrastructure. An item of content not published in this fashion would not be capable of delivery in this manner. Further, more sophisticated functionality such as BitTorrent's tit-for-tat incentive scheme would not be able to operate in this way. In fact, many current peer-to-peer principles would cease to operate as it would be impossible to control and show preference for different sources.

Summary

DONA is a content-centric network that operates at the application-layer. It constructs a DNS-like tree topology over a number of Resolution Handler (RH) servers throughout the Internet structure (i.e. within different domains). Providers within each domain register their content with their local RH using unique content identifiers. RHs then distribute content advertisements between themselves so that when consumers wish to access the content, they can forward their requests to a nearby source. Once a source receives a request it can subsequently send a transport protocol response to the consumer so that the data can be exchanged.

DONA does not suffer from compatibility issues with existing infrastructure because it operates at the application-layer. Any interoperability issues, instead, arise through software integration, i.e. the need to update applications and operating systems. Consequently, there is little support for interacting with any existing content delivery protocols. DONA also has significant deployment barriers due to its need to deploy extremely powerful RHs on a large-scale. Further, these must adhere to a single protocol thereby reducing the chances of other organisations developing their own equivalents.

DONA also does not offer delivery-centric support for applications. This is most evident because its protocol does not offer the support for carriage of delivery-centric information. However, DONA's poor transport specification and the fact that data exchange does not occur through the DONA infrastructure leaves significant scope for introducing this functionality through some form of protocol re-configuration.

In summary, DONA offers a basic approach to realising content-centric networking that is unlikely to be successful in the real-world. Despite this, it does offer feasible deployment as long as substantial resources are available. Further, there is also scope for introducing delivery-centricity. Unfortunately, however, currently DONA is poorly defined without a substantial evaluation leaving many conclusions theoretical.

2.5.3 Assurable Global Networking (AGN)

Design

Assurable Global Networking[†] (AGN) is a research project developed at PARC [83]. Like DONA, it aims to introduce infrastructural content-centric networking, offering the ability to route based on content identifiers. It does, however, take a far more comprehensive approach than DONA with considerations ranging from scalability to deployment. Further, a lower level approach is taken, including integration with existing protocols such as BGP [123] to allow routers to operate using both IP and AGN.

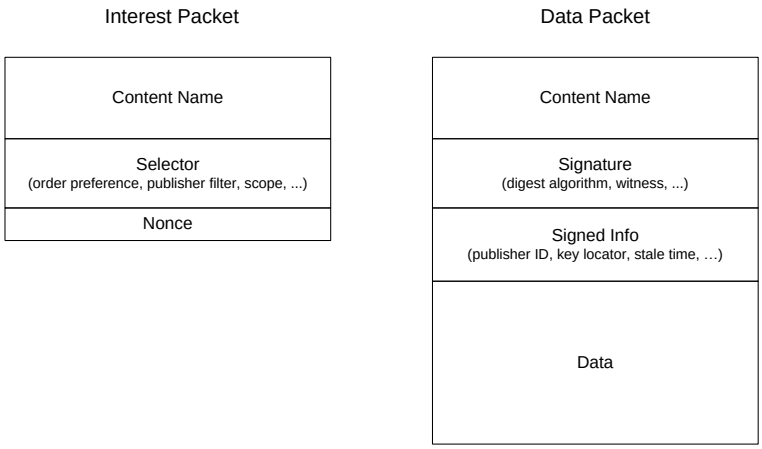


Figure 2.9: AGN Packet Types

Two packet types exist in AGN: Data and Interest, as shown in Figure 2.9. When a node wishes to access content, it broadcasts an Interest message over all available connectivity. This packet is detailed in Table 2.4; it contains a unique content identifier (ContentName), a set of parameters such as the order preference (Selector), and a Nonce to prevent the packet from looping. The most important field from these is the ContentName; this is a hierarchical identifier much in the same way that IP addresses are (net, subnet, host). This is necessary to enable scalable global routing to be achieved (addressing an obvious criticism of DONA's

[†]No name currently exists for the system developed and therefore this name is taken from an earlier PARC white paper [82]

flat addressing). Each identifier consists of a number of components, which consist of an arbitrary number of octets. Figure 2.10 shows an example content identifier. The first part of the identifier provides the global routing information; the second part contains the organisational routing information (i.e. the locally selected name); finally, the last part shows the versioning and segmentation functionality of the identifier.

Field	Description
ContentName	Unique hierarchical identifier of content
Selector	Order preference, publisher filter, scope, etc.
Nonce	Used to prevent looping in the network

Table 2.4: Overview of Interest Packet in AGN

Figure 2.10: AGN Address Format

When an Interest message is sent into the network, it is routed towards the 'closest' instance of the content, identified by the ContentName. The routing process is managed by a set of content-centric routers that operate in a similar manner to traditional IP routers (due to the similarities in the addressing scheme). An AGN router receives a packet on a network *face* (or interface), performs a longest-match lookup in a routing table, then performs some action based on the result (e.g. forwarding). Each AGN router contains three core data structures: a Forwarding Information Base (FIB), a Content Store (cache) and a Pending Interest Table (PIT). The FIB is used to forward Interest packets towards potential source(s) of data. The Content Store acts as a buffer space;

however, unlike IP, CCN data packets are idempotent and self-identifying thereby allowing one packet to potentially be used for many different consumers. This means the router buffer actually becomes a cache. Last, the PIT keeps a log of the Interest packets that have been forwarded so that returned Data packets can follow the reverse route. In essence, this creates a set of breadcrumbs so that Data packets are not required to be routed using any complicated routing information.

The way that these data structures are propagated with information is, once again, very similar to traditional IP networking. This is because of their shared hierarchical addressing structure. As such, AGN exploits existing routing protocols for its implementation. Routing can be separated into two types: intra and inter domain; these are now discussed in turn.

Intra-domain routing is the process of routing packets within an individual domain. This involves describing both local connectivity ('adjacencies') and directly connected resources ('prefix announcements'). However, unlike IP, a prefix announcement relates to given items of content rather than a host. Despite this, both IS-IS [113] and OSPF [108] algorithms can be used with AGN. AGN content prefixes, however, are greatly different to IP prefixes and therefore cannot be used ordinarily with these protocols as they currently stand. To address this, fortunately, both protocols support the advertisement of directly connected resources (in this case, content) via a general Type Label Value (TLV) scheme. This field can be used to advertise the presence of content to the other routers, allowing the AGN routers to construct their link-state routing tables based on this data. Any traditional routers simply ignore this information and pass it on. In Figure 2.11, the CCN routers A, B, E and F can therefore exchange information between each other whilst IP routers C and D simply ignore it.

The inter-domain routing in AGN is based on traditional BGP [123]; this allows each domain to announce its connectivity and the prefixes that it can route to. Subsequently, in AGN these announcements will advertise accessible content rather than IP addresses.

Once an Interest packet has been routed to a content store, the data is sent back using a Data packet. For each Interest packet, there is only a single Data packet that represents a segment of data from the content. As such, Data packets can be cached and reused. Unlike DONA, this is done through the content-centric routers and not as a separate process over IP; as such, AGN can be considered a far 'purer' example of a content-centric network. However, this also means the AGN routers must be able to achieve a suitably high throughput. Data Packets contain four fields, as shown in Table 2.5. The first two fields provide the content identifier and the actual data. The last two fields, alternatively, provide the necessary content security. Security in AGN is performed by authenticating the binding between names and content. Content names are therefore not self-

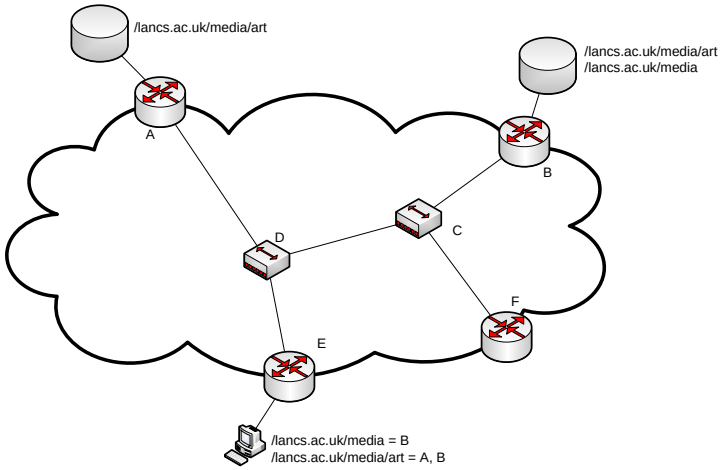


Figure 2.11: AGN Routing Process; square routers are IP, circular routers are AGN

Parameter	Description
ContentName	Unique hierarchical identifier of content
Data	The data
Signature	A hash signature of the data
Signed Info	Supporting information for data verification (e.g. hash function)

Table 2.5: Overview of Data Packet in AGN

certifying. Instead, each Data packet contains a Signature field that contains a public key signature of the entire packet. This therefore allows user or application meaningful names as opposed to using simple data hash values. The signature algorithm used is selected by the publisher based on the requirements of the content (e.g. latency and computational cost). The Signed Info field (shown in Table 2.5) is then used to allow parties to retrieve the necessary information to validate the content using the signature. This information includes the cryptographic digest of the public key and a key locator indicating where the key can be obtained from (the key could even be an item of content itself). This therefore allows a consumer to validate the content even on a per-packet basis.

Deployment Support

AGN has been designed with deployment in mind, as detailed in [83]. The primary challenge to AGN deployment is the distribution of new CCN routers as well as their progressive integration with existing network technologies (e.g. IP). AGN is not built to necessarily operate at Layer 3 of the network stack. It places

few requirements on the layer below it and therefore can operate over a number of different technologies including IP, UDP and peer-to-peer overlays. Therefore, AGN can actually be deployed as an overlay network. Unfortunately, such an approach is often plagued by performance limitations though.

AGN supports progressive deployment alongside existing non-CCN aware routers. As discussed previously, intra-domain routing can be handled by popular protocols such as IS-IS and OSPF, whilst inter-domain routing can be performed by BGP. The greatest challenge is therefore successfully operating these protocols without affecting the operation of the existing IP routing. Luckily, the Type Label Value (TLV) scheme aids the deployment greatly, as routers that do not recognise particular types simply ignore them. This means that both IP and AGN routers can co-exist without adversely affecting each other. An AGN router therefore first learns the physical topology of the network and announces its own location (using the standard protocols) before flooding the network with the content prefixes that it can resolve (using an AGN TLV). This subsequently allows a fully directed graph to be constructed for all content.

Although progressive deployment on an intra-domain level is often helpful, generally, the progressive deployment on an inter-domain level is far more important as this involves dealing with routers that are outside of an individual organisation's control. Unless this is addressed, AGN can only operate in islands. This is addressed through use of the existing BGP protocol. It does this by exploiting BGP's support for an equivalent of the TLV scheme. This therefore allows a topology map to be constructed at the Autonomous System level (as opposed to the prefix level in intra-domain routing). This map can then be annotated with information about content prefixes by any supporting router.

As of yet, the developers of AGN do not provide any evaluation regarding the system's actual deployability. Therefore, the performance and scalability of this deployment approach remains untested in the real-world. Assuming the deployment works, the largest barrier is therefore the cost of deploying the necessary routing infrastructure. [83] offers some incentives regarding the uptake of the approach. Specifically, this refers to the advantages that can be gained from caching Data packets. Unfortunately, however, these advantages are mitigated by the presence of legacy routers that cannot perform caching. Last, as with DONA, it is necessary for applications and operating systems to modify their software to introduce support for interacting with AGN. Clearly, this is endemic to introducing a new networking paradigm. This means that there is no backwards compatibility with existing content protocols (e.g. HTTP).

Delivery-Centric Support

Content in AGN is delivered using the Data packet; consumers send Interest packets containing the desired content address (which implicitly defines a spe-

cific segment) and, in return, receive Data packets that contain that segment. Transport in AGN is connectionless and client-side driven, i.e. the consumer must request every segment using an Interest packet. Therefore, unlike TCP, the consumer is responsible for re-sending an Interest packet if it has not been satisfied with a Data packet after a given timeout. Subsequently, the consumer is responsible for reliability and flow control by adjusting the frequency of Interest packets; this is analogous to TCP's window size.

First, considering the definition in Section 2.2, AGN is clearly not delivery-centric because it does not support the stipulation of delivery requirement in its protocol. Instead, the AGN routers simply route any Interest packets to the nearest source based on the distance metric (e.g. hops). As with DONA, source selection is managed entirely within the network's routing fabric and therefore consumers cannot shape the sources that are utilised. This is exacerbated further by the fact that transport occurs entirely within the content-centric routing infrastructure and, as such, a consumer cannot even ascertain where the content is coming from. Clearly this can be considered as an implicit property of a content-centric network when considering more clean-slate designs. However, this prohibits a consumer even making an out-of-band connection to a provider to attempt a delivery-centric transfer. The sources selected by the network therefore only constitute the ones considered optimal by the routing metric.

The second mechanism for achieving delivery-centricity is through protocol adaptation. Within DONA, the transfer protocol is not defined but within AGN this has become an inherent aspect of the protocol. A consumer issues an Interest packet for a particular segment of an item of content and, in return, receives a Data packet containing the data. A strong black-box abstraction therefore exists between data requests and reception. Unfortunately, with black-box systems it is impossible for adaptations to take place beyond that supported by the abstraction. As such, it is impossible for adaptation requests to be sent or any reflective information about the consumer to be provided.

Summary

AGN [83] is a content-centric network designed to be able to operate over a range of connectionless protocols, including IP and UDP. It offers content-centric routing using hierarchical identifiers alongside in-network content retrieval, i.e. content is transported through the content-centric network rather than through other means as with DONA. AGN is based on a set of special routers that can co-exist with traditional IP routers to offer the necessary routing functionality to forward requests (Interest packets) towards the nearest source. During this process, 'bread crumbs' are also left so that Data packets can be routed back to the requester.

AGN places significant focus on enabling deployment alongside existing in-

frastructure. AGN itself is designed to operate over IP, and both IP and AGN routers are designed to be co-located. To allow progressive deployment, AGN also utilises existing IP routing protocols for both intra-domain (IS-IS or OSPF) and inter-domain (BGP) routing. This is through the use of general type label schemes supported by the routing protocols, which allow content-centric prefixes to be distributed. Routers that do not support the labels simply ignore them.

Delivery-centricity in AGN is not currently supported as there is no protocol ability to stipulate delivery requirements. Instead, consumers must send Interest packets whilst providers must simply respond with Data packets. The selection of which sources provide the content is entirely managed within the routing fabric of the network and therefore consumers cannot shape this process. Further, unlike DONA, the exchange of messages occurs through AGN's own infrastructure, thereby preventing any form of protocol adaptation taking place out-of-band. These two facts make AGN non-configurable in terms of the delivery operation.

In summary, AGN can currently be considered as the purest existing content-centric network proposal available. This is because it performs both content discovery and delivery within the network without the external use of existing networks (although, obviously, it operates above an existing network). Further, AGN also offers attractive real-world solutions to deployment issues as well as feasible large-scale routing algorithms. However, despite these facts, its deployment is still slow and complicated, meaning that it is unlikely to receive wide-spread uptake. This is worsened by the use of the AGN infrastructure for forwarding data as this dramatically increases the hardware costs of the routers. This design choice similarly prevents AGN from offering true delivery-centricity, as all control is removed from the consumer and placed in the network, without the ability for individual consumers to inject personalised delivery requirements. These observations mean that AGN does not fulfil any of the key goals explored in this thesis.

2.5.4 Akamai

Design

Akamai [1] is a widely deployed content distribution network (CDN) that currently maintains a significant market share (64% [77]). It is not a content-centric network as defined by [81]; similarly, it is not a content-centric network as defined in Section 2.2. This is because it does not utilise content-centric identifiers or location-agnostic security. Instead, content addresses are location dependent; importantly, however, the delivery itself is *not* location dependent. This means that an item of content's address is based on the location of the original provider but it can subsequently be delivered from a variety of different locations through DNS redirection. It is therefore possible to argue that Akamai offers

a publish/subscribe-like abstraction, allowing providers to publish their content and consumers to access it (albeit not in a strictly content-centric way).

Akamai is a relatively straight-forward system; it acts as an augmentation to existing (web) content hosts by placing their content on its set of distributed servers. When a provider that uses Akamai receives a request, it can optionally redirect it into the Akamai network. This allows a provider to achieve superior performance, resilience and handle flash-crowds effectively. Akamai claims to have 56,000 edge servers, distributed in over 70 countries [1]. From these, 60% reside in the U.S.A and a further 30% in nine other countries [77]. Figure 2.12 provides a geographical overview of the location of Akamai's content servers. It is clear that the scale of Akamai is huge with 10-20% of all web traffic being routinely delivered by Akamai servers, sometimes reaching over 650 Gbps [1].

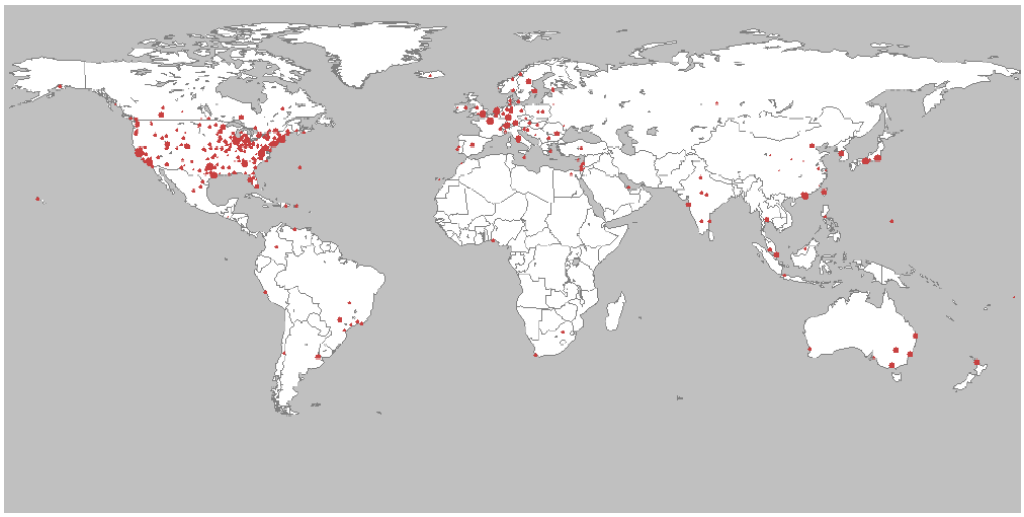


Figure 2.12: Location of Akamai Edge Servers [77]

As usual, there are two stages involved in utilising Akamai; the first is publishing content and the second is consuming it. Unlike DONA and AGN, publishing content is limited to users that are prepared to pay, i.e. Akamai is not an open system. Importantly, the process is also not an implicit function of the network; instead, it is a commercial transaction achieved through business interactions. Similarly, the content consumption process also follows a different paradigm to DONA and AGN. This is because content requests are always location-oriented, i.e. a request must be sent to a specific provider that has subscribed to the use of Akamai's infrastructure. Despite this, Akamai is by far the largest integrated content distribution infrastructure in the world and it does share many similar goals to content-centric networking.

Before an item of content can be accessed, it is first necessary for a provider to purchase the services of Akamai. Once this has taken place, it becomes possible

to 'inject' its content into the CDN. Akamai operates a multicast network based on a three-tier hierarchy, as shown in Figure 2.13. Providers first transfer their content to *entry points*; this can be done statically for traditional media or, alternatively, in real-time for streamed media [133]. Once this has taken place, the entry points pass the content to a set of *reflectors*, which subsequently are responsible for distributing the content to an optimal set of *edge servers*. These are the strategically placed content servers that actually provide the content to end hosts. To make the process more scalable, reflectors only pass content to an edge server once it has been requested in that region; this ensures that content is not pushed to unnecessary locations.

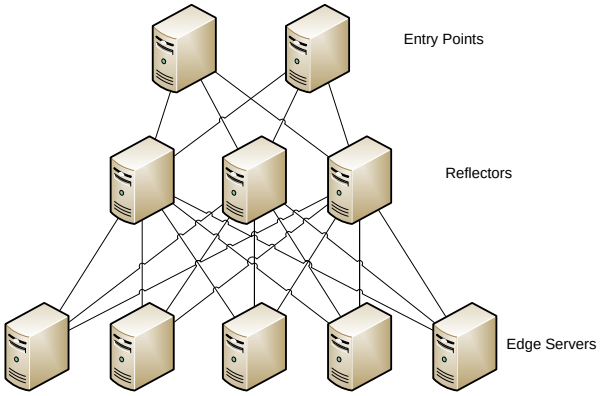


Figure 2.13: Publishing Content in Akamai through the Multicast Network

The process of accessing content in Akamai is slightly more complicated. This is because Akamai has been deployed as a transparent augmentation to existing protocols. As such, it is necessary for such CDNs to 'trick' consumers into using them without explicit knowledge. This is achieved using DNS redirection, which involves manipulating the IP addresses returned by DNS servers so that content can be acquired through Akamai rather than the original source. This process is relatively simple and is shown in Figure 2.14. When a website such as tu-darmstadt.de uses Akamai, it is provided with a unique Akamai hostname (e.g. a1964.g.akamai.net). This hostname can then be used to redirect requests to. For instance, when a user at lancs.ac.uk attempts to download an item of content from tu-darmstadt.de, its domain name is first resolved using the DNS server residing at tu-darmstadt.de (steps 1). This server then redirects the client to the Akamai infrastructure rather than resolving the query to the Darmstadt server. This is done by generating a CNAME response containing the Akamai hostname (i.e. a1964.g.akamai.net) (step 2). This results in the client generating a second DNS query for a1964.g.akamai.net, which enters the Akamai private DNS network (step 3). These servers then locate the client's closest Point of Presence (POP) based on its IP address. Using this, the query is resolved to a set of the closest

edge servers (default 2) that possess the content of tu-darmstadt.de; these IP addresses are then returned to the consumer using the DNS protocol (step 4). Finally, the consumer then connects to one of the edge servers and downloads the content using a protocol such as HTTP (step 5).

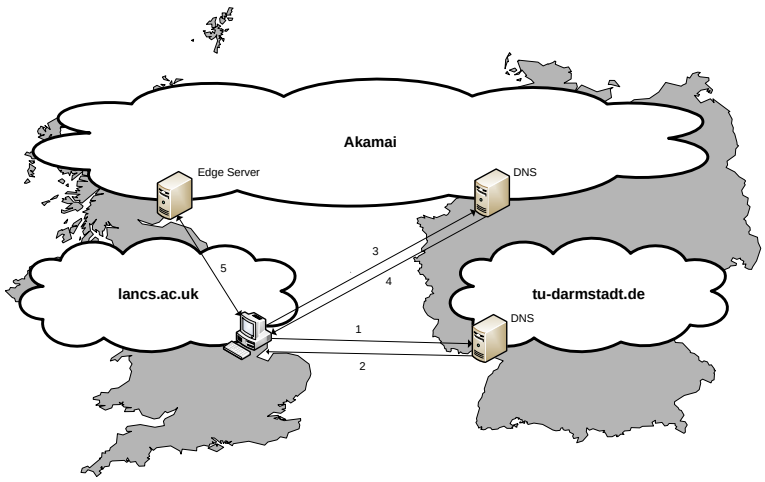


Figure 2.14: Performing a Lookup in Akamai

Deployment Support

Deployment has always been a primary concern for Akamai because it is a commercial organisation that needs wide-spread uptake to be successful. Akamai therefore has slightly different deployment challenges to DONA or AGN. This is driven by the fact that (*i*) Akamai is fully controlled by one organisation, and (*ii*) it is inherently designed to operate seamlessly with existing web distribution infrastructure (e.g. DNS, HTTP). It therefore does not offer the revolutionary approach of DONA or AGN but, instead, takes an evolutionary approach to improving existing content systems.

The biggest challenge to Akamai is hardware deployment. Unlike DONA and AGN, Akamai is primarily focussed on content delivery rather than content discovery. This is obviously a much more resource intensive process and therefore the expense of setting up an Akamai-like CDN is phenomenal. Further, it is difficult to set an Akamai-CDN up on a small scale because it's purpose is to optimise latency and increase the capabilities of existing providers. This requires that Akamai has both sufficient resources and geographical reach. This is exacerbated by the private nature of the Akamai network, thereby preventing any other organisation from contributing resources. Other, CDNs such as Limelight [17] mitigate these costs by primarily uni-locating their content servers, however, the necessary bandwidth and infrastructure for this is still extremely expensive.

Whereas, DONA and AGN are systems that demand many different third parties incorporate their protocols, Akamai is entirely private. In essence, this means that Akamai is solely responsible for its own deployment without the need to persuade any other organisations to become involved. This is therefore a type of protocol standardisation. Of course, this means that Akamai is also solely responsible for the costs. This, however, results in an alternative challenge as it becomes necessary for Akamai to (*i*) place content servers in third party networks, and (*ii*) ideally persuade third party networks into settlement-free peering [65]. Some CDNs such as Limelight [17] and Level 3 [16] are owned and managed by a particular network, which eases this burden as most servers are hosted within their own autonomous system. In contrast, however, Akamai alongside others such as Mirror Image [22] and CacheFly [5], are not owned by a particular network and therefore must distribute their servers over many different autonomous systems. The task is therefore to persuade these third party networks to install the servers or, alternatively, to peer with Akamai. Peering is the process of two networks mutually accepting each others' traffic directly without charge. This arrangement can increase delivery performance and resilience whilst reducing the transit cost for both parties. A major challenge, however, is achieving the necessary business relationships; a problem that is largely synonymous with persuading other networks to utilise a particular content-centric protocol. Akamai has a large number of peering arrangements with 63 officially listed [2]; this is due to their open policy as well as their dominant market position. Newer CDNs would not be able to achieve this acceptance, however, as peering generally requires both parties to prove their commercial worth.

Usually, the final difficulty encountered during networked system deployment is the end host software uptake. Akamai, however, also does not require modification to consumer-side software. It has been designed from the outset with backwards compatibility in mind. From the consumer's perspective no new protocols or operations are utilised; discovery is performed using transparent DNS redirection whilst delivery is seamlessly performed using HTTP. This best highlights Akamai's difference from content-centric networks such as DONA and AGN because Akamai operates in conformance with the existing IP nature of the Internet rather than attempting to introduce new protocols.

Delivery-Centric Support

As previously discussed, a delivery-centric systems makes the delivery process a primary element of its operation. In this respect, Akamai takes a largely similar approach to DONA and AGN, in that support for this is removed from the consumer and placed in the network. However, Akamai is far more flexible in its behaviour because it operates exclusively at the application-level. For instance, it offers support for HTTP, FTP, streaming, as well as digital rights and software

management.

The first possible approach to delivery-centricity is the use of intelligent source selection. Like the previous systems discussed, this is implemented within the network, however, unlike AGN and DONA, this is done through a lookup function rather than content-based routing. The lookup function returns one or more IP addresses, which implicitly provides a significantly greater degree of flexibility. By default, Akamai returns a set of two possible nearby content servers, however, these servers provide the same functionality as each other and there is little benefit in dynamically selecting between them. It is, instead, a load balancing function, leaving consumers to randomly select between the two. The ability for intelligent source selection to enable delivery-centricity therefore does not exist in Akamai.

The other approach for achieving delivery-centricity is to use protocol adaptation to manipulate delivery behaviour. Currently, Akamai offers no support for this; however, unlike AGN there is significant scope for it to take place. This is because consumers gain the address of the specific provider, thereby allowing out-of-band interactions to take place (e.g. a re-configuration request). This is also something that could be easily provisioned in the Akamai infrastructure because it is owned by a single organisation, subsequently allowing global software updates to be deployed. As such, when a lookup is performed in Akamai, a consumer could send delivery requirements to the possible sources so that their behaviour can be re-configured to address the requirements. Although this would introduce delivery-centricity, this has the downside of needing software modification to take place, thereby preventing Akamai from operating transparently. It would be easy for such infrastructure, however, to support both approaches to allow backwards compatibility. Despite this, it is also likely that Akamai would provide such support in a provider-centric manner considering that commercial income is generated via providers rather than consumers.

Summary

Akamai is a widely deployed content distribution network (CDN) that is believed to be responsible for 10-20% of all web traffic being routinely delivered [1]. It uses DNS redirection to redirect clients to nearby servers that can best serve their requested content. This is done by requiring providers to upload content into the Akamai network, addressed by a unique domain name (e.g. a1964.g.akamai.net). Therefore, whenever a provider receives a DNS lookup for its content, it can simply reply with a CNAME response that redirects the consumer to query a1964.g.akamai.net from Akamai's DNS infrastructure. Subsequently, Akamai can resolve the DNS request using the available content server that is closest to the consumer.

Akamai has clearly already shown itself to be a highly deployable system.

From the outset, there was a strong focus on achieving transparent deployment through the use of existing protocols. Akamai combines existing DNS and HTTP functionality to successfully operate without client-side modification. The biggest deployment barrier to Akamai-like systems is therefore not technical but commercial. The cost of setting up a large-scale decentralised CDN is huge, prohibiting nearly all organisations from achieving it.

Although Akamai is a delivery focussed network with significant resources solely employed in improving delivery performance, it cannot be considered as delivery-centric as it does not offer the necessary protocol support for applications to stipulate delivery requirements. Instead, the delivery protocol is a predefined element of the address (usually HTTP) and the source selection process is managed by Akamai.

In summary, Akamai cannot be considered as a true content-centric network because (*i*) it does not use location-independent addressing, and (*ii*) it does not offer any location-independent security. Instead, Akamai offers many of the facilities of a content-centric network without introducing the principles of unique content addressing. However, Akamai is not restricted to delivering content from a particular location even though the identifier stipulates it. This is achieved through transparent DNS redirection, which therefore means that Akamai does not try to introduce any new protocols making deployment and interoperability far easier. Currently, however, Akamai does not offer a delivery-centric abstraction; this is because Akamai's need for transparent interactions prevents it from providing support beyond that of existing traditional protocols such as HTTP.

2.5.5 Summary

This section has investigated the related work to content-centric networking. The purpose of this has been to ascertain to what extent existing approaches fulfil the research goals stated in Section 1.3. Three systems have been inspected: DONA, AGN and Akamai. DONA is an application-layer content-centric network that is designed to operate as a replacement to the existing Domain Name System. It allows nodes to resolve content identifiers to the closest available source using some network metric. Following this, the source can respond with a transport message to initiate the content delivery over the traditional IP network. In contrast, AGN offers a more revolutionary approach in which routers are placed throughout the networks to augment the functionality of IP. Both requests and responses are routed through the content-centric infrastructure. Finally, Akamai is a content distribution network that has already been successfully deployed. It is not, however, a content-centric network because it does not introduce unique content identification. Table 2.6 summarises the details of each system, whilst Table 2.6 provides a comparison between them.

	DONA	AGN	Akamai
Addressing	Flat	Hierarchical	URL
Routing	Tree	AS-Topology	DNS redirection
Security	Hashing	Hashing	SSL
Delivery	Out-of-bands over IP	Data Packets	HTTP
Layer	Application	Above IP/ UDP/ P2P	Application
Deployment - Cost - Speed	 High Slow	 High Slow	 Very High Slow
Interoperability - Between CCNs - Between Protocols	 None Limited bridging	 None None	 None Possible
Delivery-Centricity - Source Selection - Protocol Adaptation	 Within Network Potential	 Within Network None	 Within Network Potential

Table 2.6: Summary of Related Work

2.6 Conclusions

This chapter has investigated the background and related work to the research goals discussed in Section 1.3. First, the principles of content distribution were investigated, looking at the state-of-the-art in content discovery and delivery. Following this, technologies for achieving interoperation were also detailed to provide an overview of possible techniques to unify existing delivery systems. Last, the related work to content-centric networks was investigated based on their ability to support delivery-centricity, as well as real-world deployment. To summarise, the following conclusions can be drawn from this chapter,

- There are a large range of diverse discovery and delivery systems used in the Internet
  - Providing unified access to these different systems would increase content availability and ease the development burden on applications
  - Heterogeneity in each system makes its suitability vary when facing different requirements at different times (e.g. performance, security, reliability etc.)
- It is feasible to build interoperability into diverse networked systems

  - Protocol standardisation is largely unfeasible without commercial or sociological incentives
  - Bridging is feasible in certain cases but often cannot operate with complicated or divergent protocols
  - Middleware can provide interoperability effectively but it requires a new independent abstraction to be conformed with at certain strategic points in the system (e.g. consumer, provider or both)

- Content-centric proposals and similar content distribution systems do not satisfy the research goals
  - DONA and AGN are difficult to deploy and do not offer interoperability with existing content systems
  - Akamai is deployable (with enough resources) but it does not offer interoperability with other systems (in fact, it is an instance of one system)
  - None of the related work handles delivery-centricity as defined in this thesis

- The discovery, delivery and content-centric systems contribute to diversity rather than managing it

So far, this chapter has dealt with important background issues relating to content distribution, interoperability and content-centricity. It has also been established that a primary tenet of this thesis is the exploitation of diversity between delivery protocols and providers. A large degree of heterogeneity has been identified within this chapter, however, to feasibly exploit this heterogeneity, it is necessary to take a more detailed, quantitative approach. The following chapter builds on this chapter to take an in-depth look at the dynamic characteristics that make certain delivery systems more appropriate than others at a given time.

Chapter 3

Analysis and Modelling of Delivery Protocol Dynamics

3.1 Introduction

The previous chapter has detailed a number of possible mechanisms by which users can access content over the Internet. This includes a diverse set of protocols as well as a range of independent providers that utilise them. The primary goal of this thesis is to design and build a content-centric and delivery-centric abstraction that can be successfully deployed to interoperate with existing content systems. One of the key potential advantages behind this is the ability to dynamically select between these various different providers and protocols to best fulfil the application's requirements. This, however, is complicated by the fact that the ability of a provider to fulfil a set of requirements will often vary dynamically during runtime. This obviously strengthens the case for pushing complexity into the middleware-layer and abstracting content requests away from the underlying mechanism by which they are accessed. However, it is important to first understand how these dynamic variations take place, as well as offering mechanisms to capture and model such heterogeneity to allow informed decisions to be made.

This chapter provides a detailed quantitative analysis of the delivery system dynamics observed in three popular protocols: HTTP, BitTorrent and Limewire. These are focussed on rather than other systems and paradigms (e.g. live streaming protocols [26]) due to their predominance - they collectively make up 75% of all traffic in the Internet [128]. This is approached by investigating certain key parameters that affect the behaviour of each protocol. These parameters can be categorised as either *static* or *dynamic*. Static parameters can easily be inspected and taken into consideration at design-time, however, dynamic parameters often

fluctuate, thereby requiring decisions to be made during runtime. Consequently, it is vital to be able to understand and model how such parameters vary (*i*) over time (*temporal variance*), and (*ii*) between different consumers (*consumer variance*). This is because collectively these define the divergence that a node accessing an item of content can potentially observe. To allow a detailed analysis, this chapter therefore focusses on the dynamics of the most important delivery requirement: *performance*.

This chapter now explores the analysis and modelling of content delivery protocols. In the following sections, the three delivery systems are investigated in turn using simulations, emulations and measurement studies. Each section looks at an individual system based on both resource and protocol-specific parameters that are found to fluctuate during runtime. With this information, techniques for modelling and predicting performance are then presented to allow accurate decisions to be made dynamically relating to the best protocol to utilise.

3.2 HTTP

3.2.1 Overview of HTTP

Delivery Protocol Overview

The Hyper-Text Transfer Protocol (HTTP) is a text-based application layer client-server protocol that is primarily used to transfer content from a single provider to potentially many consumers in a point-to-point fashion. When a consumer desires an item of content it issues a GET request to a known provider. As such, HTTP is entirely based on location without any separation between content and location.

To request an item of content, a consumer first makes a TCP connection with the known server (by default on port 80). The identification of the required content is stipulated through a Uniform Resource Locator (URL). A URL offers the location of the server alongside the subsequent remote file path of the content. For instance, http://www.rapidshare.com/files/513245/music.mp3 indicates that the protocol used is HTTP, the location of the server is www.rapidshare.com (i.e. 195.122.131.14), and the remote path to the content on that server is files/513245/music.mp3.

Once a connection has been made, the consumer sends the GET request containing the URL (i.e. GET www.rapidshare.com/files/513245/music.mp3). The provider then replies with a data stream of the file. Due to this, the performance of HTTP is largely defined by the performance of TCP, which subsequently manages this data stream delivery. This is particularly prevalent for large transfers, in which the initial HTTP protocol exchange is dwarfed by the subsequent data transfer. This is highlighted in Figure 3.1, which provides an overview of the

protocol messages exchanged during a HTTP delivery.

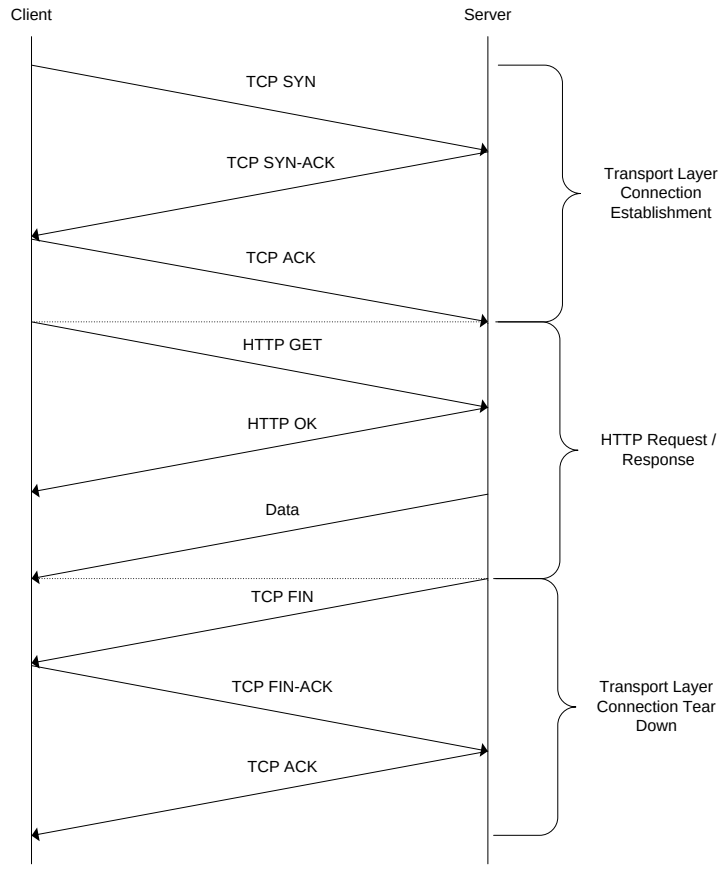


Figure 3.1: Overview of Protocol Exchange for HTTP Delivery

Discovery Protocol Overview

Discovery refers to the ability of a system to index and discover sources for a particular item of content. HTTP does not possess any native mechanism for this because there is no separation between content and location, i.e. a HTTP identifier (a URL) already implicitly contains the content source, thereby removing the need for any indirect source discovery. Instead, HTTP is simply concerned with accessing an item of content relative to a given location.

Due to this facet of HTTP, it is necessary to provide discovery as a supplementary service. This can be done through a variety of means, however, the most popular approach is to *crawl* web servers to discover what content they possess. This is done by a range of companies such as Google [11], which periodically download web content to extract pertinent elements of information. This information is then offered through a search interface.

An alternative to this is the use of user contributions. This, in essence, is a use of the peer-to-peer paradigm as exploited by systems such as Grub [12] and Harvest [13], which perform progressive web crawling in a decentralised manner. A variation of this is for users to actively upload references to content with a web interface. This is a popular approach to indexing one-click hosting sites such as Rapidshare [25].

3.2.2 Methodology

The next step is to investigate the behaviour and performance of HTTP. To do this, a number of experiments have been set up in Emulab [144]. Emulab is a testbed containing a number a dedicated computers that can be remotely accessed. Each computer possesses an emulated connection to the testbed that can be modified based on a range of parameters such as bandwidth, packet loss and delay. By using Emulab, it is possible to test a distributed system in a realistic, reproducible setting that can be subjected to many real-world factors including congestion and real protocol stack implementations.

Within the experiments, two nodes are focussed on: a client and a server. In each experiment the server possess 100 Mbps upload capacity whilst the capacity of the client is varied. Importantly, two further nodes also exist in the experiment with the purpose of creating contention on the server's resources. These two nodes maintain 40 HTTP connections to the server and are constantly downloading throughout each experiment. To emulate different server loads, the bandwidth capacity of these contention nodes is varied whilst the performance of the client is monitored. To investigate the performance of HTTP, the pertinent parameters are varied before files are transferred from the server to the client; the average application layer throughput is then calculated and logged using, $\frac{downloadtime}{filesize}$. All nodes in the experiment run Red Hat 8.0 with a 2.6 Kernel using TCP Reno; the HTTP server used is BareHTTP (available from `http://www.savarese.org`), whilst the HTTP client running is implemented in Java using the java.net package (v1.6). The HTTP server is hosted on a high capacity Emulab node with a 3 Ghz 64 bit Xeon processor; 2 GB of RAM; and a 10,000 RPM SCSI disk.

3.2.3 Resources

The predominant factor in any delivery system's performance is the available resources. This section investigates HTTP's resource usage; first, an analysis is provided before moving on to study measurements taken from Emulab.

Analysis

HTTP follows a client-server model; this makes its performance largely based on the resources available at the server. This, however, is not a static metric;

instead, it is dynamically defined by the loading that the server observes. With an increase in the number of clients being served, these resources become depleted. If this occurs beyond a certain threshold then the server will not have sufficient resources to serve all clients satisfactorily. This is obviously a variance that occurs over time, making it impossible to predict before a request is performed. There are several resources of interest when designing and provisioning servers; namely, bandwidth capacity, processing capacity, I/O capacity and memory size.

Although all resources are of interest, the one that acts as the greatest bottleneck is the bandwidth capacity. The processing capacity, although important, rarely has the significance of bandwidth capacity; [86] found that even a low-capacity 450 MHz Pentium II CPU could achieve well over 1 Gbps when serving static content items. Similarly, the I/O speed and memory size will often not act as a bottleneck as increased server load is generally associated with demand for only a particular subset of objects due to usual skewed Zipf distribution of popularity [80][147]. In contrast, the consumption of upload resources is far faster; this is most noticeable in multimedia servers that have strict bandwidth requirements. For instance, a 100 Mbps upload capacity can theoretically serve only 200 simultaneous users viewing a typical 500 Kbps encoded video. Thus, it can be concluded that the resource of most interest is, by far, the (available) upload capacity of the server.

As with any resource, each server is restricted by a finite upload capacity, denoted by up^S for server S. Whilst every client is similarly restricted by a finite *download* capacity, denoted by $down^C$ for client C. If, at a given time, n clients wish to download content from the server at their full capacity, the maximum number of satisfied connections is defined by,

$$\frac{up^S}{avg(down)} \tag{3.1}$$

where $avg(down)$ is given by,

$$avg(down) = \frac{\sum_{i<n} down^i}{n} \tag{3.2}$$

Beyond this, an admission policy must reject further requests or, alternatively, each connection must be degraded. Generally, the policy utilised is a combination of both these approaches with rate allocation being managed by TCP, which attempts to allocate bandwidth fairly (although this is often ineffective [96]).

Measurements

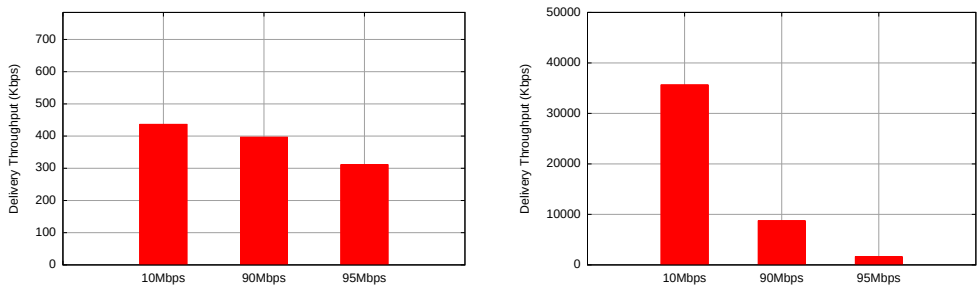
To investigate the effect of different server workloads, a 4 MB delivery is performed from the server to the individual monitor clients when operating under various loads. Two monitor clients are used: a low capacity 784 Kbps client and

a high capacity 100 Mbps client. 4 MB is used because it is large enough to reach the achievable peak throughput (after TCP slow-start); a larger 700 MB delivery was also performed with similar results. Three different workloads are utilised in the experiments to emulate different available resources; each load is defined by the application layer congestion at the server (excluding the monitor client's request). The first represents a *low* workload and consists of 10 Mbps traffic. The second is a *medium* workload of 90Mbps traffic that leaves sufficient resources for the client to access the content. Last, a *heavy* workload of 95 Mbps is used to emulate a fully saturated server (note that 100 Mbps is not an achievable saturation level due to protocol overheads). Importantly, by generating traffic at the application layer, the server also suffers from processor consumption and not just bandwidth consumption. This therefore allows the effect of different resource levels to be monitored in a controlled environment.

Figure 3.2a shows the average download throughput for a client with a 784 Kbps download capacity. It can be seen that the client's performance is relatively unchanged under the three different loadings. This is attributable to the client's low bandwidth requirements, which place only a very light load of the server. This can be contrasted with a high capacity 100 Mbps client, which suffers greatly from any degree of loading at the server, as shown in Figure 3.2b. The effects become even more extreme once the server's resources are saturated, with a 95% performance decrease for the 100 Mbps client. This occurs because during saturation it becomes necessary to share the server's upload capacity between all clients. The HTTP implementation does not perform any intelligent resource management and therefore bandwidth allocation is managed by TCP, which attempts to allocate a fair share between all streams. By providing each client with a 'fair share' of bandwidth, low capacity clients can easily saturate their downlinks without significant degradation. In contrast, high capacity nodes suffer greatly from this algorithm as their fair share is far lower than their desired throughput. Consequently, the effects of resource variations are observed in vastly different ways when perceived from different clients: this is evidently a *consumer variance*.

The experiments have so far highlighted that as the server's finite resources are consumed, the performance perceived by the clients is reduced. Further, it has also been shown that there is a marked difference between the effects on divergent consumers. Subsequently, it can be derived that two clients operating at identical times can get totally different qualities of service (consumer variance). The next step is to ascertain whether or not these fluctuations also occur over time (temporal variance). To achieve this, it is necessary to inspect real-world studies to understand how resource demand varies over time.

The presence of temporal variance is validated by a range of studies. Yu et. al. [147] found that the access patterns of users vary greatly over time;



(a) 784 Kbps Download Capacity Client (b) 100 Mbps Download Capacity Client

Figure 3.2: HTTP Download Performance

by studying the PowerInfo Video on Demand (VoD) service, it was found that server loading varies on both a daily and weekly basis. Unsurprisingly, usage drops during the early morning (12AM-7AM) and in the afternoon (2PM-5PM), whilst it peaks during the noon-break (12PM-2PM) and after work (6PM-9PM). It was found that this can result in variations between 0 and 27 requests per 5 seconds. Similarly, [80] found that access rates vary significantly based on the time of the day and the time of the week. Obviously, the effects of these variations are most noticeable with low capacity servers, which seem highly prevalent; for instance, the University of Washington probed a set of 13,656 web servers to discover that more than 65% had under 10 Mbps upload capacity [112]. With such limited resources, the effects of high load can obviously result in significant problems. However, even in well resourced environments such as Rapidshare [25], these issues can still take effect. Antoniades et. al. [35] performed a number of Rapidshare downloads from an individual site. They found that the performance significantly varies over time. With a premium account, the capacity can range from as little as 1 Mbps to over 30 Mbps, with a 50:50 split between those achieving under 8 Mbps and those achieving more. Clearly, these results therefore confirm the existence of *temporal variance* in HTTP based on the availability of resources.

3.2.4 Protocol

The purpose of this section is to investigate how dynamic variations endemic to the protocol implementation can affect performance regardless of the available resources. Table 3.1 provides an overview of the relevant network parameters for HTTP, as detailed in [38]. It should be noted that there are also a number of other higher-level parameters, relating to both TCP and HTTP (e.g. maximum transmission unit, TCP/HTTP version). However, these are not considered because they are parameters that can be better configured by the providers and

Parameter	Description
Server Bandwidth	The amount of available upload bandwidth at the server
Packet Loss	The percentage of packets lost during the delivery
Link Asymmetry	The difference between a link's upload and download characteristics
Propagation Delay	The delay between the client and the server

Table 3.1: Overview of Relevant HTTP Parameters

consumers (i.e. they are not outside of their control).

The first parameter, *server bandwidth*, is a resource metric as previously described. The second parameter is *packet loss*. This references the percentage of lost packets that a stream endures when operating over a given link. High levels of packet loss obviously affect performance because it becomes necessary to re-send the lost data. TCP's behaviour, however, also exacerbates this by using packet loss as a congestion metric. The third closely linked parameter is *link asymmetry*; this can make it difficult for TCP's congestion algorithms to correctly ascertain the bottleneck capacity, thereby potentially lowering performance. All of these parameters are clearly important, however, the chosen parameter is *propagation delay*. First, an analysis of the importance of this parameter is provided, followed by details from the Emulab measurements.

Analysis

TCP operates with a closed control-loop. Receivers must acknowledge the receipt of data periodically so that the sender can verify it has been successfully sent (using an ACK packet). The period over which this control-loop operates is defined by the *window size* ($CWIN$); this represents the number of bytes that a sender can transfer before waiting for the next ACK. Subsequently, a sender will pause the download periodically whilst it waits for the receiver's ACK packet. This results in throughput being limited by,

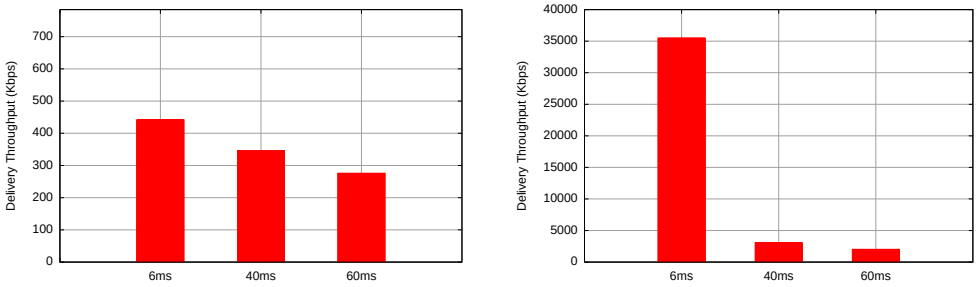
$$throughput <= \frac{CWIN}{delay} \tag{3.3}$$

If a connection has a lengthy delay then this will result in a significant proportion of time being spent in this waiting state. This problem is most noticeable in *long fat pipes* that have the capacity to send at a high data rate but are restricted due to the TCP control-loop [150]. To highlight this, a simple example is used; imagine an up^S of 100 Mbps, a connection delay of 100 ms and a *fixed* window size of 64 KB. The time it takes for the server to transfer 64 KB is ≈5 ms. The server must then wait for the ACK to be received which takes a further

100 ms; during this time a further ≈1.2 MB could have been transmitted. In this example, each control cycle therefore results in a huge wastage of capacity.

Measurements

To study this, several experiments are performed in Emulab whilst varying the delay between the HTTP server and the clients. Figure 3.3 shows the throughput for two clients requesting a 700 MB file from the server with a 10 Mbps background load and different delays. It can clearly be seen that the throughput degrades as the delay increases. This is most noticeable for the high capacity 100 Mbps client, as shown in Figure 3.3b, with the 60 ms delay achieving only 9% of the throughput achieved by the 6 ms delay. This is attributable to the larger Bandwidth-Delay Product (BDP) of this node when compared to the 784 Kbps client; this therefore results in a greater percentage of wasted capacity during high latency transfers. For instance, every 10 ms spent in a wait-state results in the waste of just under 1 KB for a 784 Kbps client whilst the same period spent waiting costs 125 KB for a 100 Mbps client.



(a) 784 Kbps Download Capacity Client

(b) 100 Mbps Download Capacity Client

Figure 3.3: HTTP Download Performance with 10 Mbps Contention and Different Delays

These results show that delay has a significant impact on the performance of a HTTP connection. Recent studies have shown that delay in the real-world can vary dramatically over time by up to four orders of magnitude [95] (*temporal variance*). However, the most likely cause for heterogeneity is the location of the requesting client (*consumer variance*). For instance, two nodes connected via the same network will have a lower delay than two nodes operating at different sides of the globe [87]; this is highlighted in Table 3.2 by the measured delay through both a co-located wired and wireless access links. Some content distribution networks such as Akamai [1] reduce this effect by replicating content at multiple geographical locations, however, the majority of web providers are located at only a single site. Rapidshare, for instance, hosts all its servers at a single site in

Host	Country	Wireless RTT	Wired RTT
bbc.co.uk	UK	10 ms	7 ms
google.com	UK	22 ms	13 ms
rapidshare.com	Germany	43 ms	37 ms
cornell.edu	NY, USA	124 ms	107 ms
heraldsun.com.au	Australia	329 ms	306 ms

Table 3.2: RTT between Lancaster University and Remote Hosts

Germany [35]. Similarly, content distribution networks such as Limelight [17] also host their servers at only a small number of sites. In such situations, the effects of delay are not mitigated through site replication. Interestingly, even when sites are replicated using CDNs such as Akamai, latency is not static or necessarily well controlled. Instead, [94] found that over 20% of clients witness, on average, 50 ms greater delay than other clients operating in the same geographical location. Further, it was also found that 40% of clients suffer from over 200 ms delays even when accessing content provided through CDNs such as Google.

3.2.5 Modelling

The previous sections have shown that the performance of HTTP possesses both consumer and temporal variance that cannot be predicted at design-time. Consequently, to achieve delivery-centricity using HTTP, it is evidently necessary to be able to dynamically select providers. To allow this, however, it is necessary for a runtime model to be built of the system to allow comparison between different alternatives. This section now explores how HTTP can be modelled to predict the performance it will achieve under certain runtime conditions. First, the model is described before showing how the important parameters can be acquired. Following this, a validation of the model is given.

Model

Modelling the behaviour of HTTP is largely a process of modelling TCP. This is because in most situations the performance is defined by the underlying algorithms of TCP. Deliveries can be separated into two groups; first, small downloads that are dominated by delay and, second, larger downloads that are dominated by TCP's congestion control.

When performing small downloads, the primary limiting factor is the propagation delay between the two points. This is because when the data is below a default 64 KB, it is possible for the entire download to take place without the TCP congestion control algorithms being executed. A prime example of this is a picture; a 32 KB file could be transferred using 22 Ethernet frames without the

need for TCP to employ any ACKs. This makes the TCP and HTTP initiation phases the greatest bottleneck. This involves the following steps,

1. *TCP SYN:* Client sends connection initiation
2. *TCP SYN-ACK:* Server acknowledges
3. *TCP ACK:* Client acknowledges the acknowledgement
4. *HTTP GET:* Client sends content request

This means that two round trip times are required before the data can be sent (i.e. $4 \cdot delay$). For instance, a typical delay between Lancaster and Washington University would be 175 ms. In such a circumstance, the initiation phase would last approximately 700 ms (excluding any processing time). In contrast, based on a bandwidth capacity of 5 Mbps, the data transfer period would only last approximately 50 ms. In such circumstances, throughput can therefore be approximated by,

$$\frac{filesize}{delay \cdot 4 + (filesize/bandwidth)} \tag{3.4}$$

where *delay* is the connection delay, $filesize$ is the size of the content being accessed and *bandwidth* is the consumer download bandwidth (assuming the local bandwidth is less than the provider's bandwidth). This subsequently creates a rough estimate of the time required for the data exchanged before adding it to the time required for the connection initiation.

The above represents the simplest scenario because the process does not involve any of the more complicated functionality of TCP. In contrast, larger items of content will be subjected to the congestion control algorithms employed by TCP, which subsequently dwarf the impact of the connection initiation delay. A number of models have been developed for predicting the achievable throughput by a TCP connection. The most predominant of these is the so called square root equation, first defined by Mathis et. al [105] and extended by Padhye et. al. [110]. This model calculates achievable throughput based on two key parameters: delay and packet loss. Within such a model, for the moment, it is therefore necessary to assume that server resources always exceed consumers resources.

Within this model, the throughput is defined as being inversely proportional to latency and the square root of loss rate. Variations also exist for alternative versions such as TCP Reno [111]. The model, taken from [110], is as follows,

$$E[R] = min\left(\frac{M}{T\sqrt{\frac{2bp}{3}} + T_o min(1, \sqrt{\frac{3bp}{8}})p(1+32p^2)}, \frac{W}{T}\right) \tag{3.5}$$

where p is the loss probability and b is the average number of packets acknowledged by an ACK, W is the maximum congestion window, M is the maximum

segment size, T is the average roundtrip time, and T_o is the average duration of a timeout without back-off. Evidently, both b, W and T_o are parameters defined by the operating system, whilst p and T are dynamic parameters that vary between different hosts. The model has been empirically evaluated using real-world traces to show its accuracy. Further studies, have also performed independent analysis of the model to find that the majority of predictions fall within a factor of two in terms of accuracy [117].

Acquiring Parameters

The previous section has provided one mechanism by which throughput between two points can be predicted mathematically. However, before this calculation can be undertaken, it is necessary to acquire the necessary parameters dynamically; namely, delay and packet loss.

Considering the time-sensitive nature of generating predictions, active probing cannot be used. Therefore, to achieve this with a low overhead, network monitoring services must be utilised. These are remote databases that are accessible via some form of interface. They collect and maintain information about system resources to allow a host to query the performance of a particular system aspect. A prominent example of this is the Network Weather Service (NWS) [145], which monitors and predicts various network and computational resources. Network characteristics are periodically measured using a distributed group of hosts that perform active probing whilst computation resources are measured using daemons residing at end hosts. Other alternatives to this include GridMAP [59] and Remos [107], which both offer the ability to generate accurate throughput predictions.

A more recent proposal is that of iPlane [102], which offers the ability to query network characteristics for links between arbitrary hosts. This service exposes an RPC interface that can be accessed by consumers with very low latency. This service simply exposes a method (`iplane.query`), which accepts a list of source/destination pairs, and returns a range of information including both packet loss and delay. This information is collected from a number of network probes, which monitor characteristics over time and index this information so that it can be accessed by remote hosts. This is exposed as a web service and, as such, can be accessed with very low delay. To validate this, a number of queries were generated with increasing numbers of hosts being queried (in batch requests). The delay increases linearly with only ≈13 ms of processing delay for every 10 extra providers queried, making the service highly suitable for generating low latency predictions.

Validation

The previous two sections have shown how one of the many TCP models can be utilised to predict the achievable throughput when accessing content via HTTP. It is important, however, to validate that these predictions are actually accurate and feasible when used in conjunction with iPlane. This can be done using results provided in [102].

To perform this study, active probes were performed between 161 PlanetLab nodes to ascertain their achievable throughput. Following this, iPlane was queried to generate predicted throughput values based on loss and latency using the model from [111] (shown in Equation 3.5). Figure 3.4 compares the two sets of results. It can be observed that the predicted values, on the whole, are extremely accurate. For instance, for over 82% of predictions, iPlane achieves an error range of under 10%.

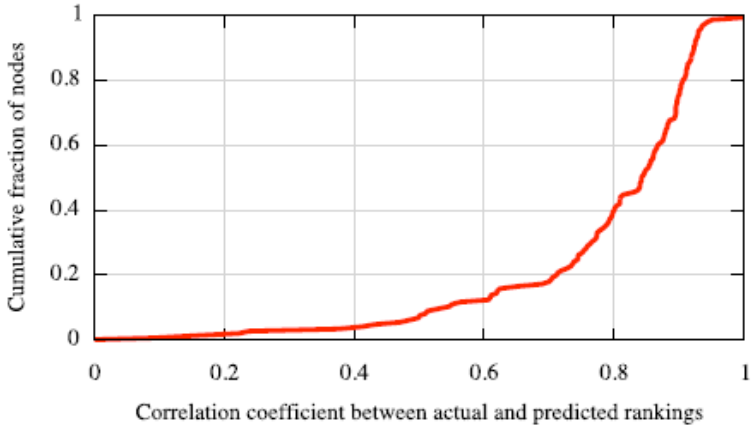


Figure 3.4: Rank Correlation Coefficient between Actual and Predicted TCP throughput [102]

Although, these results are clearly very accurate, the primary issue when using these predictive services is that of real-time variations. For instance, a flash crowd scenario would not be captured by a predictive service unless it occurs during a measurement period. Unfortunately, most services utilise relatively limited numbers of probes (e.g. two a day [103]) and therefore such rapid changes in network behaviour cannot be discovered. Despite this, it is likely that most Internet paths remain relatively similar. For instance, using traceroute operations, Zhang et. al. [149] found over 75% of Internet paths remain identical on a day-to-day period. However, obviously this does not incorporate end-host characteristics such as computational loading. Such information would be perfect for exposure by a reflective interface or, alternatively through a NWS daemon [145].

In summary, the combination of iPlane and existing models can clearly be seen to offer an effective approach to generating predictive throughput values. Impor-

tantly, this is simply one exemplary approach; there are a number of constantly improved throughput models, including [57]. Similarly, alternative network services can be used to acquire the necessary characteristic parameters.

3.2.6 Summary

This section has inspected the dynamic behaviour of HTTP. It has been shown that HTTP exhibits significant runtime variations when operating in different conditions and with different clients. Two aspects have been investigated: available server resources and propagation delay.

It has been shown that both available server resources and connection delay have a significant effect on performance. Importantly, consumer and temporal variance is observable in both aspects. Consequently, when dealing with HTTP, two consumers will likely receive vastly different performance levels if either (*i*) they access the content at different times, or (*ii*) they possess different personal characteristics (e.g. bandwidth capacity, location). From this it can be concluded that performance-oriented delivery-centricity can only be achieved if providers can be dynamically selected on a per-consumer basis. To achieve this, however, it is evidently necessary to provider each consumer with a personalised prediction regarding the achievable performance of each potential provider; therefore, this section has also detailed a mechanism by which important runtime parameters can be acquired with low overhead, and converted into such predictions.

3.3 BitTorrent

3.3.1 Overview of BitTorrent

Delivery Overview

BitTorrent [49] is a peer-to-peer distribution system; it has become the de-facto standard for scalable content delivery with 66.7% of all peer-to-peer traffic being attributable to BitTorrent [128]. It operates by separating files into small pieces called chunks, which are then exchanged between the cooperating peers; this set of peers is collectively referred to as a *swarm*. A swarm is a file-centric entity in which all peers are sharing an individual item of content. Consequently, each swarm can be considered to be an independent provider in itself.

When a new peer wishes to download a file it first connects to a *tracker*; this is a centralised manager that maintains an index of all peers sharing or downloading a particular file. The tracker provides new peers with a list of existing peers that can be contacted. Peers can be categorised into two types: *seeders* and *leechers*. Seeders are those that already possess the entire file and are altruistically sharing it, whilst leechers are those that are still downloading the content.

After receiving a set of peers from the tracker, new users connect to a random subset of the existing peers to request data. Each peer maintains a bit map indicating which chunks it possesses and which chunks it requires. These chunk maps are then exchanged between peers using the Peer Exchange Protocol (PEX) to allow each peer to discover which other peers can provide the required chunks. To improve availability, each peer requests the chunks that appear to be the rarest according to its local view of the swarm.

Each peer maintains a set of five upload slots for which competing peers barter for. This process is managed by BitTorrent's incentive mechanism, termed rate-based *tit-for-tat* [99]. If there is contention over a peer's upload slots, the peer shows preference to the requesters that have been uploading to it at the highest rate, i.e. the ones that have been contributing the most. This metric is calculated locally and subsequently peers cluster into neighbourhoods within which they download and upload to each other. If peer P^1 makes the decision to upload content to peer P^2, it is said that P^2 is unchoked. By default, four peers are unchoked periodically (every 10 seconds) based on the tit-for-tat algorithm. However, to ensure that newly arrived peers that cannot cooperate (due to a lack of chunks) are not prevented from downloading, each peer also randomly selects a fifth peer to unchoke (every 30 seconds); this is termed optimistic unchoking because it is hoped that the peer will reciprocate.

Discovery Overview

Unlike many other peer-to-peer systems, BitTorrent does not natively support the discovery of content within its own peer-to-peer infrastructure. Instead, content is indexed and discovered outside of the swarm. Every swarm is managed by a individual server known as the *tracker*, which is responsible for providing newly joined peers with a list of existing swarm members. It therefore provides a bootstrapping mechanism and acts as an access point for the swarm.

Databases of trackers are kept by a variety of indexing web sites that allow users to search for content. Popular examples are Mininova.org and PirateBay.com, which index millions of objects by providing links to each swarm's tracker. This allows users to query the index using both web interfaces and web services to discover (*i*) items of content, and then (*ii*) the address of the tracker to access the swarm through. This approach allows content to be quickly discovered with response times that are orders of magnitude below traditional peer-to-peer searching. However, there are also peer-to-peer discovery mechanisms such as Azureus' KAD that provide this functionality in a decentralised way

3.3.2 Methodology

BitTorrent, unlike HTTP, is a complex system that cannot be easily studied in a controlled environment. This is because the performance of BitTorrent is largely based on the behaviour and characteristics of the peers operating in the system (which are difficult to accurately capture). Further, logistically, the resources required to emulate a BitTorrent swarm are much greater than a client-server system. Therefore, to address these concerns, two approaches are taken: simulation and large-scale measurements.

Simulation. The simulator utilised is the OctoSim [40] BitTorrent simulator, developed by Carnegie Mellon University and Microsoft Research. To investigate BitTorrent's performance in diverse settings, a number of simulations are performed whilst varying the resources and behaviour of the swarm members. For each experiment, one client is selected as a monitor node. Each experiment is run with a monitor node possessing a small set of different bandwidth capacities to inspect how divergent nodes behave. This allows a detailed analysis to be performed in a flexible, reproducible and controlled manner. The bandwidth of the other clients is distributed using the measurement data provided by [56]. Peers join the swarm in a flash crowd scenario, all at time 0; this allows a more impartial evaluation to be performed based on the varied parameters rather than other aspects such as the arrival rate.

Measurement Studies. The main limitation of utilising simulations to investigate BitTorrent is the difficulty of truly modelling real-world situations. This is because capturing realistic node capacities and behaviour is often impossible. To remedy this, two detailed measurement studies have also been performed that offer insight into BitTorrent's real-world performance.

The first measurement study inspects high level characteristics of swarm behaviour by periodically probing a number of trackers. These measurements are termed *macroscopic* as they investigate behaviour on a torrent-wide basis. To achieve this, a crawler was developed that listens for newly created torrents published on the Mininova website [21]. After this, it followed these torrents to collect information about their member peers. This ran for 38 days, starting on the 9th December, 2008; it collected information about 46,227 torrents containing 29,066,139 peers. This process allowed the crawler to log peer arrival patterns alongside the seeder:leecher ratio within the torrents over time.

To complement the information gained through the macroscopic measurements, a second *microscopic* study was also performed. This involved developing and deploying a crawler that can investigate swarms on a per-peer level as well. The crawler operated from the 18th July to 29th July, 2009 (*micros-1*) and then again from the 19th August to 5th September, 2009 (*micros-2*). It collected sets

of online nodes from the trackers indexed by the Mininova.org website, requesting information using the Peer Exchange Protocol (PEX). Periodically (every 10 minutes) the crawler requested a chunk bitmap from each peer in the swarm. This information was then logged, generating 7 GB and 12 GB of data from each trace respectively. For the *micros-1* study, the crawler followed 255 torrents appearing on Mininova after the first measurement hour. In these torrents, 246,750 users were observed. The *micros-2* dataset contains information from 577 torrents and 531,089 users. The logs consist of the chunk bitmaps and neighbour table entries for each peer sampled with a resolution of every 10 minutes.

After the study had completed, the logs were post-processed to remove all peers that were not recorded for their entire download lifetime; this left approximately 12,000 peers in *micros-1* and 92,000 peers in *micros-2*. The average download throughput of these peers was then recorded by performing the following calculation, $throughput = \frac{filesize}{downloadtime}$. Following this, the peak download rate of each peer was calculated by sequentially inspecting the chunk maps of each peer. This was then assumed to be the maximum download capacity of the peer therefore allowing the downlink saturation to be calculated, $\frac{throughput}{peakthroughput}$. This information was then logged alongside various parameters.

3.3.3 Resources

Analysis

BitTorrent is a peer-to-peer system and therefore the available resources are defined by those contributed by peers. Subsequently, the available resources can be modelled by looking at the contributed resources minus the required resources. If this is negative, clearly there are insufficient resources in the system to satisfy peer requirements. In BitTorrent these two factors can generally be measured by looking at the number of seeders compared to the number of leechers, defined by the seeder:leecher (S:L) ratio. In this section, it is represented with a single decimal figure; this is calculated using, $\frac{n^s}{n^l}$ where n^s and n^l are the number of seeders and leechers, respectively.

Seeders play a very important role in BitTorrent. On the one hand, they ensure that the complete file remains available [88] by acting as a source of every possible chunk. Whilst, on the other hand, they also improve download speeds by providing resources without consuming any [40].

Every torrent, x, can be identified as having a particular service capacity (up_T^x); this is the aggregate of available upload capacity and is defined for torrent x at a given time as,

$$up_T^x = \sum_{i<n^s} up_S^i + \sum_{i<n^l} up_L^i \tag{3.6}$$

where up_S^i and up_L^i are the upload capacities of seeder i and leecher i, respec-

tively. Equally important is the *required* service capacity of the swarm; this is the aggregate download capacity of the leechers and is defined at a given time for torrent x as,

$$down_T^x = \sum_{i<n^l} down_L^i \tag{3.7}$$

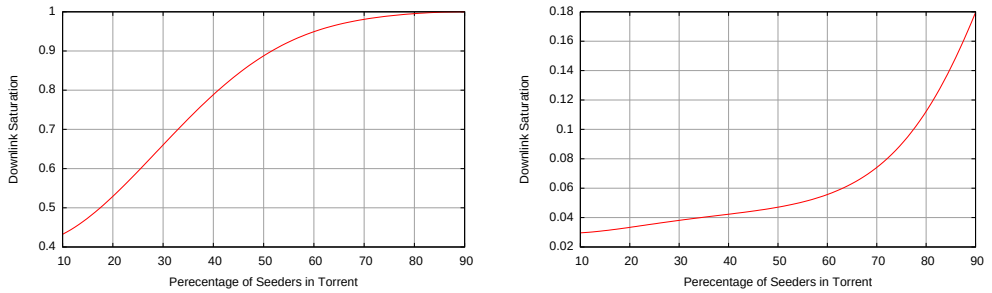
Assuming that all chunks are available, in a homogeneous environment it is therefore easy to model the percentage downlink saturation for torrent x by,

$$min\left(\frac{up_T^x}{down_T^x}, 1\right) \tag{3.8}$$

Unfortunately, however, due to the asynchronous nature of most Internet connections [56], it is generally impossible for a swarm of solely leechers to achieve downlink saturation as $\frac{up_L^i}{down_L^i} < 1$. Saturation can therefore only be achieved by the presence of seeders as seeders increase service capacity without introducing any service requirement.

Measurements

To investigate the effect that the S:L ratio has on download performance, a number of experiments are executed in the OctoSim simulator. Ten torrents are created and to each one, 100 nodes are added in a flash crowd scenario (i.e. all at time 0). Each torrent operates with a different percentage of seeders in the swarm (10-90%) to measure how performance differs.



(a) A 784 Kbps Download / 128 Kbps Upload Client

(b) A 100 Mbps Download / 100 Mbps Upload Client

Figure 3.5: Download Saturation with Different Seeder:Leecher Ratios (Simulated)

Figure 3.5a and Figure 3.5b show the downlink saturation for these experiments. In both situations, the increase in the number of seeders results in improved performance. Interestingly, however, the curves of each graph are significantly different. When compared to the 784 Kbps client, the 100 Mbps client is

more sensitive to the S:L ratio; it can be seen that the saturation continues to increase even when the torrent is made up of 90% seeders. This occurs because of the higher capacity client's very high service requirement (its optimal download time is only 56 seconds). It can therefore easily continue to consume the resources of increasing numbers of seeders. In contrast, the 784 Kbps client is far less sensitive to variations in the S:L ratio because its service requirement is far lower [114]. In essence, due the observation that usually $up_L^i < down_L^i$, peers that have download capacities that are above the swarm average require seeders to achieve higher download rates whilst those that operate below, can generally receive high performance regardless. This is a prominent example of *consumer variance*. This is exemplified by the number of uploaders that a 100 Mbps client would require to achieve downlink saturation; assuming that the five unchoked users get an equal share of bandwidth (as assumed by [101])), this would be modelled by

$$\frac{100Mbps}{avg(up_L) \cdot 0.2} = \frac{100000Kbps}{105Kbps} = 380 \tag{3.9}$$

This occurrence can also be investigated by inspecting the measurement results. To validate the findings, users from the measurement logs are grouped into intervals (of size 0.05) based on the S:L ratio of the swarm when each user *first* joins. Figure 3.6 shows the average downlink saturation achieved by each S:L range. It can be seen that the plot follows a similar trend to the simulation results with a steady increase in downlink saturation, until it begins to curve off. Whilst the S:L ratio is below two, there is a strong linear increase, as the peers eagerly consumer the increased resources. As the ratio increases beyond two, however, this trend tails off as many of the lower capacity peers reach their saturation (99% of measured peers had a downlink capacity of under 10 Mbps).

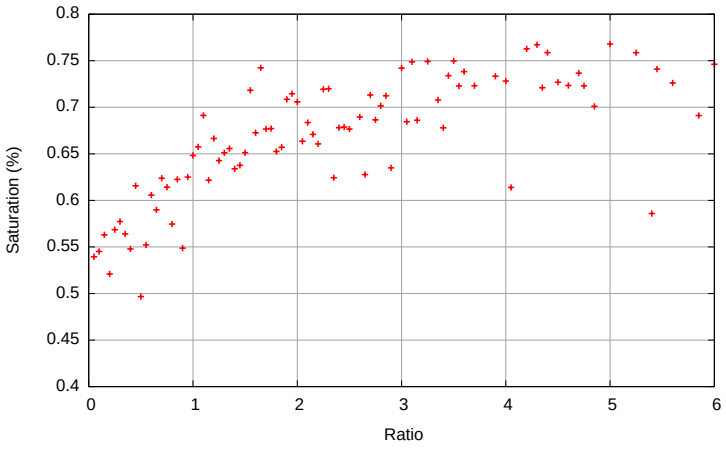


Figure 3.6: Downlink Saturation with Different Seeder:Leecher Ratios (Measured)

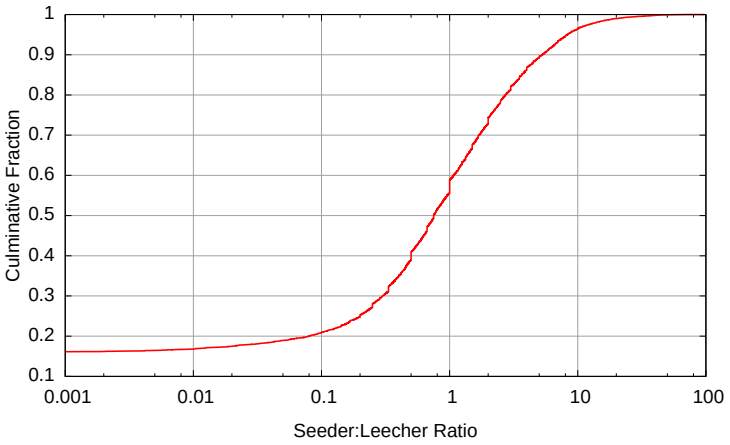


Figure 3.7: Cumulative Distribution of Seeder:Leecher Ratios

	Anime	Books	Games	Movies	Music	Others	Pics	TV
S:L	0.78	2.74	0.66	0.42	1	0.58	3.3	0.8
Exp. Sat	61%	71%	58%	54%	64%	60%	74%	57%

Table 3.3: The Median Seeder:Leecher Ratio and Expected Downlink Saturation for Different Content Categories

So far, it has been shown that a change in the S:L ratio of a torrent generally translates into a change in the achievable performance witnessed by consumers. Further, it has similarly been shown that different nodes undergo this change in different ways (*consumer variance*). The next important step is therefore to validate these large inter-torrent variations actually exist in the real-world. To explore this, Figure 3.7 shows the cumulative distribution of S:L ratios in all torrents at a snapshop 2 weeks into the macroscopic traces (note the log scale). It can be observed that there are a huge range of S:L ratios, varying from 0.001 to 100, with the majority (74%) falling below 2. Clearly, variations in the S:L ratio are therefore common-place showing that the performance received by a consumer is entirely dependent on the item of content being accessed. To gain an understanding of how this might translate to performance variations, Table 3.3 also shows the observed median S:L ratios for the various content types, as well as the average downlink saturation levels that can be expected for each category (based on Figure 3.6). This shows, for instance, that a client downloading pictures will likely gain far higher performance than a client downloading TV shows.

The previous results show that consumer variance is highly prevalent in BitTorrent with large variations between consumers accessing different content, as well as consumers possessing different downlink capacities. It is also important, however, to ascertain if BitTorrent similarly possesses high levels of *temporal*

variance in terms of resource availability. This can be done by inspecting how torrents' S:L ratios evolve over time, i.e. are the S:L ratios largely static or do they vary dynamically? Figure 3.8 shows the number of seeders and leechers for a representative torrent* over time, taken from the macroscopic measurements. Evidently, the population size can clearly be seen to change dramatically. For instance, between minute 0 and minute 173, the number of leechers increases from 1 to 91. Similarly, the number of seeders can change rapidly with a ten-fold increase taking place over only 33 minutes (minute 173 - 206). The most noticeable change occurs after 213 hours when the number of seeders drops to zero. This indicates that during this period the file will be unavailable [88]. Interestingly, however, on day 5 the number of seeders increases again meaning that, once again, the content will be available. Consequently, two identical consumers would achieve entirely different levels of performance when downloading the same item of content at different times, highlighting a strong *temporal variance*.

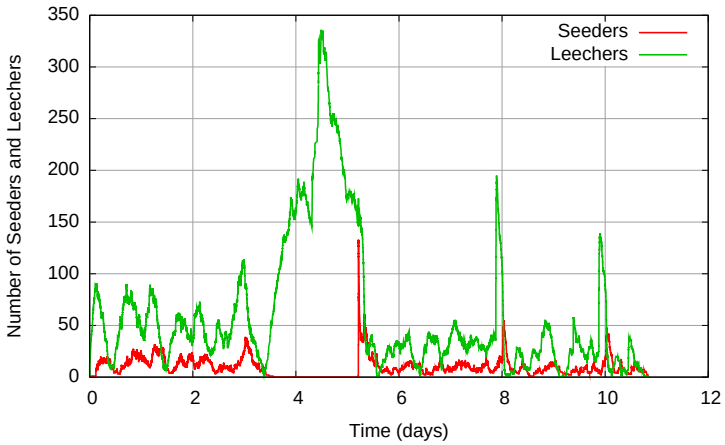


Figure 3.8: Seeder:Leecher Ratio of Representative Torrent over Time

3.3.4 Protocol

This section investigates how specific facets of the BitTorrent protocol can affect the performance perceived by consumers. Due to the complexity of BitTorrent, a large number of parameters can be observed to have dynamic variations. Table 3.4 provides an overview of important parameters in BitTorrent.

The first important parameter is the average peer *upload capacity*. This is an important resource parameter as its sum represents the swarm capacity. This can also be contrasted with the average peer *download capacity*, which defines the resource requirements of the users. This has already been exemplified previously by varying the S:L ratio. These parameters, however, also have an extended

*Similar behaviour has been observed in most torrents.

Parameter	Description
Average Peer Download Rate	The average download rate in the swarm
Average Peer Upload Capacity	The average upload capacity of peers
Client Upload Capacity	The upload capacity of the inspected client
S:L Ratio	The number of seeders compared to leechers
Churn Rate	The degree of churn observed in the swarm
Swarm Size	The number of users in a swarm

Table 3.4: Overview of Relevant BitTorrent Parameters

effect when combined with the consumer's *upload capacity*. It has been found that peers tend to cluster together into bandwidth symmetrical neighbourhoods [97]. Consequently, a consumer's upload capacity when compared against the swarm average largely defines its ability to achieve a high performance. The next important parameter is the *churn rate*; it has been found that swarms with low churn are far more resilient and have better availability [88]. It was also found that larger swarms better ensure the availability of files. However, swarm size does not similarly have a significant effect on download performance [40].

The chosen parameter to investigate is the *client upload capacity* as this can be easily identified by each consumer (to assist in later modelling). The rest of this section now focusses on investigating how performance varies with this parameter. Information regarding any of the other parameters can be found in papers such as [40][62][73][101].

Analysis

Within BitTorrent, peers compete with each other for the finite number of upload slots within the swarm (default 5 per peer). Periodically (every 10 seconds), peers decide which neighbours they will upload chunks to. To achieve this, each peer periodically ranks all the peers it is downloading from based on the observed bit rates. Using this ranking, the top n peers in this list (default $n = 4$) are *unchoked* and their requests accepted. This competition therefore means that a leecher i with up_L^i will generally receive content only from peers with $up_L <= up_L^i$, thereby resulting in the clustering of nodes based on bandwidth capacities [97].

Measurements

To study this, simulations are solely used as the measurement studies cannot capture the upload rate of individual users. A movie torrent containing 100 peers is simulated using the median S:L ratio from the macroscopic measurements (0.42); all peers join in a flash crowd. Several experiments are executed whilst varying the upload capacity of peers.

Figure 3.9 shows the download time of a 700 MB file by four different clients. Each client has a different download capacity: 784 Kbps, 5.5 Mbps and 100 Mbps. Experiments were then executed whilst varying each node's upload capacity as shown on the y-axis. The effect that decreasing the upload capacity has is significant. The average upload capacity in the swarm is 526 Kbps; when exceeding this average (i.e. 5.5 and 100 Mbps), the download time improves significantly. Conversely, when the node's upload capacity drops below the average (i.e. 128 and 512 Kbps), the opposite occurs. This indicates that the performance of a client operating in BitTorrent does not rely on its download capacity but, rather, its upload capacity; this is confirmed by [120]. Clearly, this is another example of *consumer variance*, which is based on the individual consumer's upload capacity, as well as the average upload rate of the swarm. Considering that this average will also change as peers come and go, a *temporal variance* similarly exists.

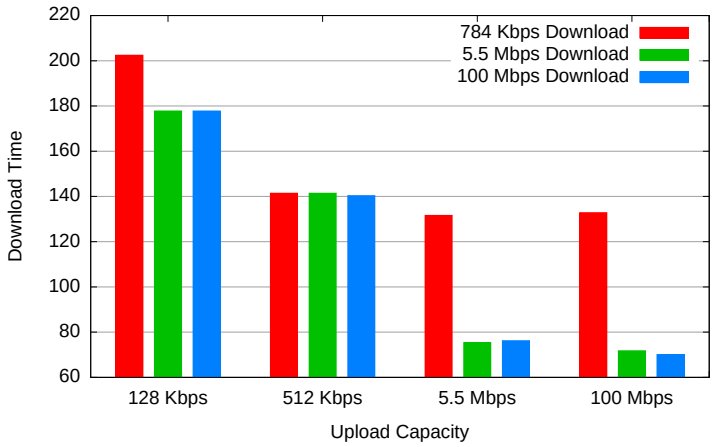


Figure 3.9: Download Times for Clients with Different Upload Capacities

These assertions are best exemplified by comparing the 5.5 Mbps client with the higher capacity 100 Mbps client. It can be seen that the download times are almost identical despite the vast differences in their download capacity. This is because both peers are clustered around the same neighbours (and therefore getting the same unchoke rate). An interesting point to also note is that even when both the 5.5 and 100 Mbps peers possess a huge upload capacity of 100 Mbps, their download times remain (almost) the same. In theory, the tit-for-tat algorithm should, instead, allow the higher capacity client to utilise its download capacity better. This does not occur, however, because of the far lower average upload capacity in the swarm, preventing the nodes from saturating their downlink (as discussed earlier). Importantly, as observed in both consumer access links [56] and specific BitTorrent studies [115][120], there is a huge range of upload capacities, thereby leading to two conclusions. First, these different users will receive a different performance based on their upload capacity; and, second,

an arbitrary user with a given capacity will receive a different performance based on the other peers that have chosen to join its swarm. These are both therefore prominent examples of *consumer variance*.

3.3.5 Modelling

The previous sections have shown how BitTorrent's performance can vary when operating under different conditions; this includes both swarm-wide aspects (S:L ratio) and local aspects (upload capacity). This section explores how the performance of BitTorrent can be modelled using the available upload capacity of a client.

Model

It has been shown how the seeder:leecher ratio (resource aspect) and the local upload capacity (protocol aspect) heavily affect the performance of BitTorrent. The latter is a factor that is introduced when the resource capacity of the swarm is not plentiful enough to fully serve all peers (i.e. $\frac{up_T^x}{down_T^x} < 1$), thereby resulting in resource competition. When this occurs, it becomes necessary for peers to trade their upload resources in return for the upload resources of other peers. In such situations, a node chooses to upload chunks to nodes that, in return, upload chunks at the highest rate. Unfortunately, the vast majority of swarms operate competitively due to poor seeder:leecher ratios (74% below a ratio of 2) and therefore it has been found that a peer's upload capacity can be used as a primary factor in predicting its download rate [79][115].

BitTorrent's tit-for-tat incentive mechanism does not offer perfect fairness in which contributions are matched with improved performance linearly. Instead, imperfect fairness is achieved in which increases in contributions result in a sub-linear monotonic increase in performance [115]. This means that a consumer who contributes a greater upload capacity will achieve a higher performance but not necessary at the same level that it contributes. The ability for a peer to compete can simply be ascertained by comparing its upload capacity with that of other peers in the swarm; this principle can be utilised to build a model of the achievable download rate of a given peer. This has already been shown to be possible in Section 3.3.4 through the observation that varying a node's upload capacity modifies its download rate, regardless of its download capacity.

Piatek et. al. [115] performed an extensive measurement study of BitTorrent, observing over 300k peers. With this, they constructed a model to predict the download rate (reward) received from a given upload rate (contribution). Table 3.5 details the necessary parameters required to calculate this value; these are based around general shared parameters such as the number of transfer slots, as well as runtime torrent-specific parameters such as the upload rate of each peer.

Parameter	Default	Description
ω	2	Number of simultaneous optimistic unchokes per peer
b(r)	runtime	Probability of upload capacity rate r
active(r)	$\lfloor\sqrt{0.6r} - \omega\rfloor$	Number of peers in active transfer set for upload capacity r
split(r)	$\frac{r}{active(r)+\omega}$	Per-connection upload capacity for upload capacity r
S(r)	runtime	cumulative probability of an equal-split rate r

Table 3.5: Functions and Parameters used in BitTorrent model [115]

The rest of this section details the model.

It is first necessary to calculate the probability by which a peer C will be unchoked by another peer P (i.e. consumer and provider). This is simple to do in BitTorrent as the decision to unchoke another peer is based on its upload rate when compared against the other members of the swarm. This can be modelled by inspecting the cumulative distribution of per-slot upload rates within the swarm; this is represented as $S(r)$, where r refers to a particular upload rate. Consequently, $S(r)$ refers to the fraction of peers that offer an upload slot that is less than or equal to r. This can then be used to calculate the probability by which the consumer's upload rate (rC) will be sufficient to compete for the potential provider's upload rate (rP). This can be modelled as,

$$p\_recip(rC, rP) = 1 - \Big((1 - S(rC))^{active(rP)}\Big) \tag{3.10}$$

The next step is to calculate the download rate that can be achieved by a consumer based on its upload rate. Evidently, this is a function of its competitiveness, which is shown using $p\_recip(rC, rP)$. Assuming no optimistic unchokes, this is subsequently modelled by multiplying the probability of receiving an upload slot with the upload capacity of the upload slot. The probability of receiving reciprocation from a node with an upload rate of r can be modelled by,

$$b(r) \cdot p\_recip(rC, rP) \tag{3.11}$$

where $b(r)$ is the probability of a node in the swarm having an upload capacity of r and $p\_recip(rC, rP)$ is the probability of such a node accepting a chunk request.

This can then be converted into an upload rate by multiplying it by the per-slot upload capacity for a given peer, denoted as $split(r)$ for a peer with an upload rate of r. This is simply calculated by dividing the peer's upload capacity by its number of slots.

Now it becomes possible to calculate the upload rate received from P by C using the default number of slots at each peer, denoted by $active(r)$ for a client with an upload rate of r. This can be calculated as,

$$active(r) \cdot b(r) \cdot p\_recip(rC, rP) \cdot split(r) \tag{3.12}$$

A similar process can also be performed for modelling the upload rate gained from optimistic unchokes with the removal of $p\_recip(rC, rP)$,

$$\omega \cdot b(r) \cdot split(r) \tag{3.13}$$

The sum of these values therefore defines the download rate a peer with a given upload rate, r, will receive. This is because a node's download rate is simply a product of its neighbours' upload rates. A peer's predicted download rate can therefore be calculated by combining the upload rates received from both normal and optimistic unchokes using,

$$\begin{aligned} D(r) = \quad & active(r) \quad \left[\int \cdot b(rP) \cdot p\_recip(rC, rP) \cdot split(rP) d \cdot rP \right] + \\ & \omega \qquad \left[\int \cdot b(rP) \cdot split(rP) \cdot d \cdot rP \right] \end{aligned} \tag{3.14}$$

Acquiring Parameters

The previous model requires a small number of important parameters to work, listed in Table 3.5. Some of these parameters are static configurable values that are set at each client, namely $active(r)$ and ω, which represent the number of active upload slots and optimistic unchokes, respectively. These are therefore default values that can be set statically based on the current standards.

There are also two runtime parameters that are based on the swarm that a peer is operating in. The first is $\mathrm{b}(r)$, which models the probability of a client existing in the swarm with an upload capacity of r. The second runtime parameter is $\mathrm{S}(r)$, which is the cumulative probability of the equal split rate of r. This is obviously a derivative of $\mathrm{b}(r)$ based on the $\mathrm{active}(r)$ sizing. These parameters are required so that a peer can ascertain how well it will be able to compete in the swarm. Further, they allow a resource model to be constructed to predict download rate; this is obviously because the collective upload rate in a swarm is also equal to the collective download rate.

To calculate these dynamic parameters it is necessary to generate a representative distribution function of the upload bandwidth available within the swarm. This is a similar challenge to acquiring the upload capacity of a HTTP server, however, there are a number of more stringent requirements. First, it is not possible to assume that one peer would have a greater upload capacity than another in the same way it can be assumed that a server has a greater capacity than a client. Second, scraping a torrent for peers can return as many as 200 members making any form of active probing unscalable.

An important factor that can be used to address this issue is the observation that a strong correlation exists between the upload rates of nodes sharing the same IP prefix [115]. Subsequently, given an accurate data set, a peer's IP

address can be mapped to an upload capacity using a longest match lookup. Evidently, this requires an up-to-date data set, which must also be representative of the bandwidth capacities allocated to BitTorrent and not the raw link capacities as provided by traces such as [56]. This is because often clients utilise traffic shaping to limit their connection speeds (most BitTorrent implementations support this). Such a data set is openly provided by the University of Washington, which performs periodic monitoring of BitTorrent users from a number of vantage points as detailed in [79]. This involves discovering peers from a large number of BitTorrent swarms and downloading chunks from them using optimistic unchoke slots. This then can be used to compile a profile of the upload capabilities of peers operating with different autonomous systems and IP prefixes.

This data set (available from [15]) contains approximately 87k samples; each entry consists of a /24 prefix and the median access link bandwidth measured by the BitTorrent vantage points. Importantly, the measurement study verifies that the values are similar for end-hosts that share the same /24 prefix. This data set is only 1.8 MB and can therefore reside at each Juno host with periodic updates being performed. Consequently, a new consumer can acquire a list of peers from the tracker and then calculate a distribution function for the bandwidth by performing lookups for each peer's IP address in the data set. This is therefore an extremely low overhead process that can be performed in well under a second.

Validation

It has already been shown in Section 3.3.4 that nodes with vastly different download capacities receive almost identical download throughputs when operating with the same upload capacity. However, it is also important to validate the presented model on a large-scale. To achieve this, results are taken from [115]. These results compare the values generated by the model against real-world measurements taken from over 63k peers. To acquire this data, a number of monitor points downloaded chunks from each peer being measured (using optimistic unchokes) in order to calculate each peer's split(r) parameter (i.e. its per-slot upload rate). Whilst doing this, the download rate of each peer is also monitored using similar techniques to those detailed in Section 3.3.2 (i.e. requesting PEX chunk maps). This information is then correlated to map a peer's upload rate to its achieved download rate.

Figure 3.10 shows the expected download performance as a function of upload capacity; this is solely based on the model presented in the previous section. This can be contrasted with Figure 3.11, which shows the actual measured download throughput as a function of upload capacity (obtained using the previously described technique). Each point represents an average that has been taken over all sampled peers.

The first observation is that the model correctly predicts the sub-linear growth

that is observed in the real-world. This means that increasing the upload capacity of a peer will not result in a one-to-one improvement in download performance. However, it can be also be observed that the model offers slightly more optimistic predictions. For instance, using the model, an upload capacity of ≈2 Mbps should result in a download rate of ≈1.6 Mbps. However, in practice the measurements show that the download rate achieved is only ≈1.2 Mbps. This can likely be attributed to more conservative active set sizes than assumed in the model. Unfortunately, this is difficult to resolve as it is not possible to dynamically collect each peer's active set size. Instead, the most common standard must be used, as shown in Table 3.5. Despite this, clearly the model offers a low overhead mechanism by which a relatively accurate download prediction can be gained.

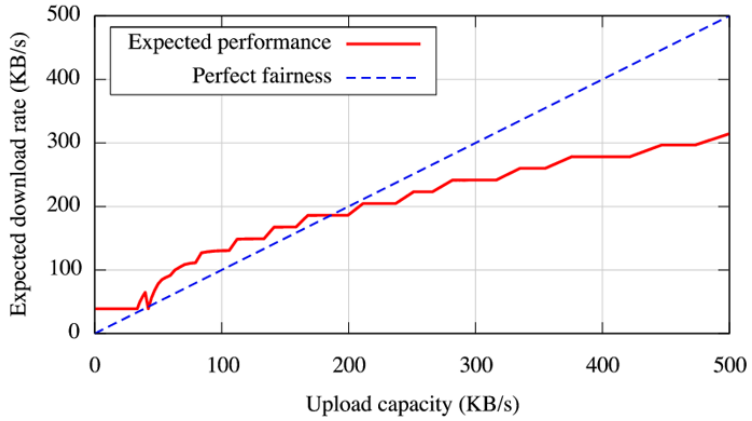


Figure 3.10: Predicted Download Performance as a Function of Upload Capacity based on [115] Model

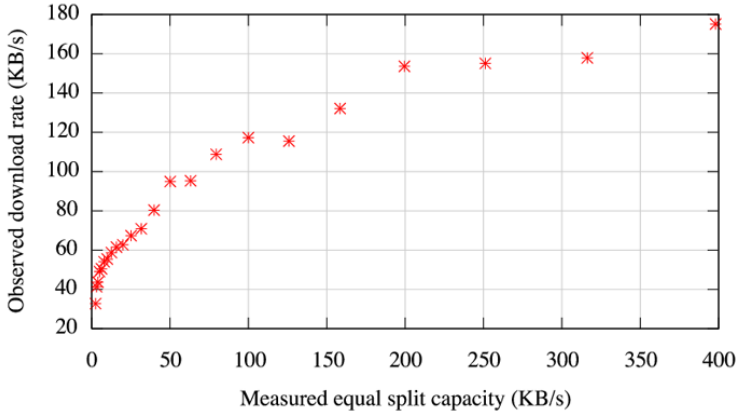


Figure 3.11: Measured Validation of Download Performance as Function of Upload Capacity [115]

3.3.6 Summary

In summary, BitTorrent has been shown to have large performance fluctuations when operating in different environments. Two parameters have been inspected: the seeder:leecher (S:L) ratio and the upload capacity of peers. First, it has been observed that the S:L ratio defines the resource availability within a swarm and that peers operating in well resourced torrents will gain higher performance. Further, those peers operating in swarms without any seeders are likely to find the content often unavailable. Second, it has also been shown that the upload capacity offered by a peer has a direct impact on its received performance when compared against the capacity of other peers in the swarm. This is because peers operating in torrents without sufficient resources must compete for upload slots. Consequently, those peers that contribute the greatest upload capacity get the largest download rate. This means the download performance of a consumer will vary based on its own upload capacity as well as the upload capacities of other members of the swarm.

These observations prove that the performance of BitTorrent varies both over time and between different clients. A client operating in a torrent can get a very high performance but later return to the same torrent to receive a much worse performance due to variations in the S:L ratio. Similarly, two clients operating in the same torrent at the same time are likely to receive different download throughputs if they have different upload capacities. This means that selecting the use of BitTorrent at design-time will result in large variations in performance observed by the different application instances.

Based on the above findings, a model has also been detailed (from Piatek et. al. [115]) that can be used to generate runtime predictions of performance based on the upload capacities of swarm members. Using a publicly available data set, it is possible to efficiently estimate the upload capacity of any BitTorrent peer based on its /24 prefix. Consequently, this allows predictive performance meta-data to be generated dynamically based on the information provided by a BitTorrent tracker.

3.4 Limewire

3.4.1 Overview of Limewire

Delivery Overview

Limewire is a popular open source peer-to-peer file sharing application. Limewire operates a delivery mechanism based on simple multi-source swarming. When a Limewire peer begins a download it first creates a local representation of the entire file in which each region is defined by one of three colours:

- *Black* regions indicate the data range has already been downloaded
- *Grey* regions indicate the data range is currently being downloaded but it has not yet been completed
- *White* regions indicate the data hasn't been downloaded yet

When a source is discovered it attempts to 'grab' responsibility for a region based on a predefined range size. If it is successful it then makes a connection to the source using HTTP and requests the data range using the HTTP range field (e.g. Content-Range: bytes 21010-47021). This process continues with each source grabbing an available data range until the entire file is downloaded. If a source is available, yet there are no white regions remaining, the latter half of a grey region is 'stolen' and allocated to the newly available source. Obviously, the download from the original source is terminated half way through to account for this (this is easy to do as HTTP downloads occur sequentially). This therefore accommodates sources having different upload capacities or, alternatively, new sources being discovered mid-delivery.

In contrast to BitTorrent, Limewire does not operate any sophisticated incentive mechanisms. Instead, by default all requests are accepted by a Limewire peer until a configurable limit is reached. Subsequently, performance is equally shared between the different requesters based on the underlying TCP implementation.

Discovery Overview

Content discovery in Limewire is currently performed by the Gnutella 0.6 network [91]. Gnutella is a decentralised limited-scope broadcast overlay that maintains an unstructured set of connections between the member peers. It is used to allow users to query each other to ascertain whether or not they possess a given file (based on meta-data). More details can be found in Section 2.3.2.

The Gnutella protocol is decentralised and, as such, is highly resilient. However, the protocol has three major limitations: reachability, overhead and delay. This means that queries generated in Gnutella are not guaranteed to return all possible results from the network. This makes it only suitable for popular content. Perhaps more importantly, queries are slow to yield results and generate a high level of overhead [125]. Consequently, accessing small files is ill-advised as an application requesting the download of a 1 KB picture for instant display to the user, does not wish to wait 30 seconds for the discovery process to be accomplished. In such a circumstance, it would obviously be superior to use a fast discovery mechanism, even at the cost of a slower delivery mechanism.

3.4.2 Methodology

To investigate Limewire, a small-scale measurement study has been performed. A Limewire client has been hosted on a high capacity node connected via a 100 Mbps synchronous connection. This node has then been used to download a number of files from other users in the Limewire network. After every download, information about the file is logged and the throughput achieved is recorded. Through this, a profile can be constructed of the achievable performance when utilising the Limewire delivery mechanism.

3.4.3 Resources

As with BitTorrent, the resources available to Limewire are defined by the contributions made by peers. Unlike BitTorrent, however, there is not an explicit separation between users that are downloading a file and those that are in the process of doing so. Hashing in Limewire is not performed on a per-chunk level and therefore only users that possess the entire file are discoverable through Gnutella.

Analysis

The resources available to a Limewire download are defined by the number of sources and their respective upload capacities. Assume a set of n available sources, $\{S^1, S^2...S^n\}$ with respective upload capacities of $\{up^1, up^2...up^n\}$. The achievable download performance for consumer C at a given time is therefore defined by,

$$max\left(down^C, \sum_{i<n} up^i\right) \tag{3.15}$$

Therefore, as the number of sources increase, clearly so does the amount of resources available to a peer. Evidently, sources are transient and therefore depending on up^S, performance can significantly vary if a client witnesses churn within its source set.

Measurements

To investigate the effect that the number of available sources has on performance, a number of Limewire downloads have been performed. Queries were generated for various popular music files with sizes ranging from 3 MB to 7 MB. The chosen range of source numbers is from 1 to 9; this is because over 95% of files are hosted by under 10 sources. The number of sources of a file are identified based on the search results. Importantly, it is possible for a node to discover new sources after the download has begun by gossipping information between the nodes it is downloading from. Similarly, it is possible for nodes to go offline through churn. Therefore, the experiments also incorporate real-world variations in the source

set. For each number of sources, 10 downloads were executed using different files. The average was then calculated over this set.

Figure 3.12 shows the average throughput of a download based on the number of sources available. It can be seen that the performance clearly increases with larger numbers of sources, in-line with Equation 3.15. This occurs for two reasons; first, there are more resources available for the download, and, second, the greater the number of sources, the greater the probability of finding a high capacity peer. The downloads were performed using a 100 Mbps connection and therefore such sources can easily be taken advantage of. This is exemplified by the steepness of the curve showing that increasing the number of sources can have a significant impact.

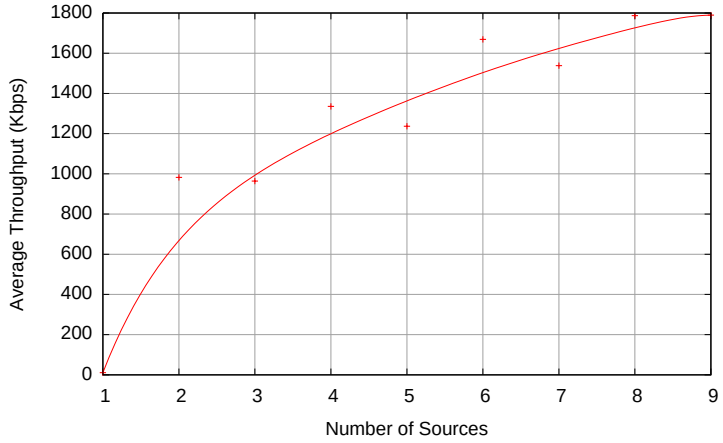


Figure 3.12: Download Performance based on the Number of Sources

This means that a strong *consumer variance* exists, based on the number of sources discovered by the peer; this is a facet of both file popularity and the node's reachability in the Gnutella network. To understand this better, however, it is also necessary to ascertain the performance of individual peers. The reason for this is two fold; first, the majority of files are served by only a single peer; and second, the performance of a multi-source download is actually just the sum of multiple individual connections. To investigate this, 100 uni-sourced downloads have also been performed. Figure 3.13 shows the cumulative distribution of the throughput achieved during these downloads. Importantly, this therefore also offers the upload capacity of the remote source at that time. It can be observed that the majority of users serve files at under 600 Kbps with an average upload rate of only 264 Kbps. Consequently, users accessing unpopular, single-sourced files are likely to fail to achieve downlink saturation.

The previous results have shown that consumer variance emerges when accessing different files, based on the degree of replication. This degree of replication is largely based on popularity, which evidently also results in a temporal vari-

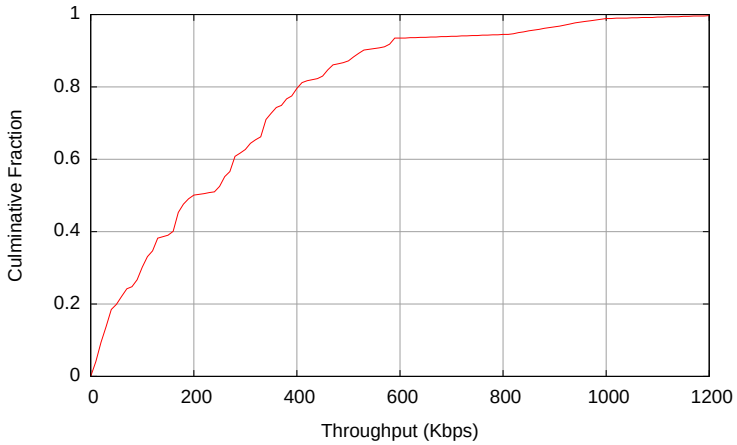


Figure 3.13: Cumulative Distribution of Individual Peer Upload Capacities in Limewire

ance as the popularity of content evolves. To exemplify this, a number of queries were generated for the Top 10 music charts from both November 2009 and May 1985 (tests were performed during November 2009). Unsurprisingly, the recent music had a high replication rate with, on average, 60.7 copies being located by Gnutella. In contrast, however, only an average of 6.2 replicas were discovered for the 1985 music. This is because, as detailed in [151], the files (and number of replicas) constantly evolve based on user trends. Consequently, the resources available for a user downloading a file are constantly in flux.

3.4.4 Protocol

The previous section has investigated the important resource parameters relating to Limewire's performance. Limewire is a variation on the HTTP protocol; in essence, it operates as a collection of HTTP connections to different sources with a simple management mechanism so that each connection knows what to request. As such the resource parameters are different in Limewire (due to multiple sources) but the protocol parameters are the same as HTTP, discussed in Section 3.2.

3.4.5 Modelling

So far, it has been shown that the performance of Limewire is largely based on the number of available sources. This section now builds on this study to investigate how history-based predictions can be performed to model performance at runtime.

Model

Modelling the behaviour of Limewire can be considered as a process of modelling the performance of multiple HTTP streams. Section 3.2.5 detailed an approach for generating performance meta-data relating to HTTP. However, the assumptions made for HTTP providers do not hold for Limewire; namely, that the performance bottleneck is the TCP congestion control algorithm rather than the provider's upload bandwidth. Consequently, such a model can only be utilised if it possible to acquire the resource availability of remote sources to perform a *min* function that can compare the theoretical maximum capacity achievable by the TCP connection with the available upload bandwidth at the source.

To address this, the resources of Limewire peers can be modelled using history-based prediction [76]. This involves extrapolating future performance based on past experiences. The biggest challenge to this is the collection and aggregation of information. The simplest way this can be done is to utilise a linear predictor in which future values are estimated as a linear function. This involves collecting the throughputs achieved via previous downloads and calculating a predicted value based on the number of sources available. The simplest approach to this, is the use of a moving average, as calculated by,

$$\hat{X}_{i+1} = \frac{1}{n} \sum_{k=i-n+1}^{i} X_k \tag{3.16}$$

in which $\hat{X}_i$ is the predicted value and X_i is the observed value at time i. This function therefore consumes all previous throughput values and aggregates them as a moving avergage. Consequently, if n is too low, it is impossible to smooth out the noise of potential outlier. However, on the other hand, if n is too large, it will become slow to adapt to changes in the environment. Evidently, the former case cannot be addressed in any other way than to wait until more downloads have been performed. To address this, alternate smoothing functions also exist that offer better reaction to recent changes (e.g. exponentially weighted moving average). A detailed overview can be found in [76].

Acquiring Parameters

The previous section has detailed a history-based approach to generating throughput meta-data. The acquisition of the required parameters is therefore simply a process of gathering and storing the throughput for each delivery. The key question is therefore how this information is categorised; traditional history-based prediction mechanisms generate moving averages for each remote source interacted with. This is often possible when utilising a limited set of servers (e.g. Akamai edge servers). However, when operating in a peer-to-peer system, the chances of repeat interactions are extremely low [90]. To address this, through-

put information is gathered and collated based on the number of sources, rather than the individual peers. This therefore involves creating a moving average for $1...n$ sources, where n is the maximum number of sources previously encountered for an individual download.

This information can be stored within an array; each element contains the current moving average of the throughput for deliveries from that number of sources. When a delivery has completed, its throughput can be calculated simply by,

$$throughput = \frac{filesize}{downloadtime} \tag{3.17}$$

Following this, the value can be added to the moving average as detailed in the previous section.

Validation

The previous sections have detailed how predictive meta-data can be generated for Limewire deliveries using past interactions. It is now important to validate whether or not these claims are accurate. This is done using the previous measurement data presented, which contains the throughput for each Limewire download performed in a time series; this allows the logs to be post-processed to ascertain the predictions that would have been generated. This involved performing a linear moving average prediction (c.f. Equation 3.16) sequentially over each download so that the predicted values can be compared against the achieved throughput. The rest of this section now validates this approach by inspecting the results.

Figure 3.14 shows the standard deviation of the predictions when compared against the actual throughput for each group of downloads (grouped by number of sources). The throughput of highly sourced downloads can clearly be accurately predicted; for instance, the results show that downloads with over 7 sources gain under a 0.2 factor of deviation. This level of accuracy is achieved because such deliveries can have any anomalous results averaged out by the other sources. Subsequently, the use of moving averages is perfectly suited to this circumstance. Unsurprisingly, it can be observed that downloads from fewer nodes have significantly greater inaccuracy. This is particularly prevalent in single source deliveries. This occurs because such downloads are highly susceptible to anomalies, e.g. particularly high or low capacity peers. Consequently, there can be a high deviation in the accuracy of predictions when downloading from a single peer.

To study this, Figure 3.15 presents the cumulative distribution of the deviations from the predicted value for single sourced downloads. It can be seen that 58% of predictions are within one factor of accuracy whilst 45% gain better than a factor of 0.6. In contrast, 29% gain extremely inaccurate predictions that are over two factors out. The most extreme example is a delivery that has a throughput that is 43 times less than the predicted value (achieving only 6

Kbps). Subsequently, a large number of the predictions are actually more accurate than the averages, with a small number of extreme outliers. Unfortunately, however, there is no true way of improving the prediction for these occasional anomalies and, as such, they will generally be predicted inaccurately unless the particular sources have been previously encountered. Clearly, however, the linear function performs an effective task of smoothing out these occasional anomalies, with accurate predictions for the remaining deliveries.

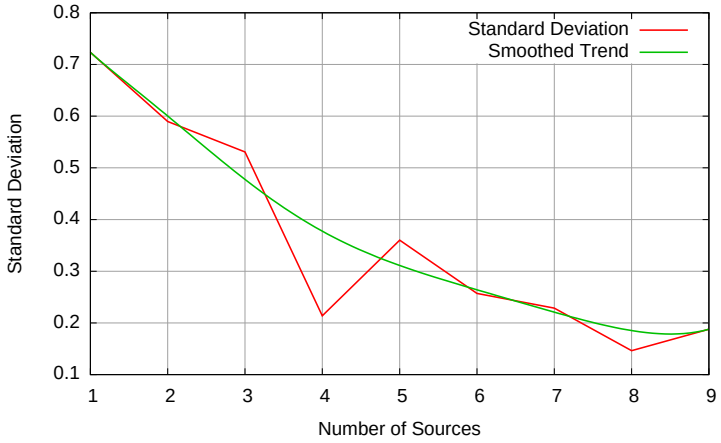


Figure 3.14: Standard Deviation of Prediction from Achieved Throughput

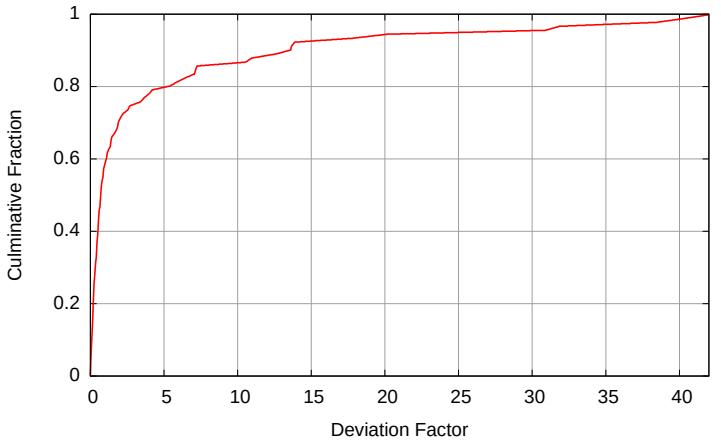


Figure 3.15: Cumulative Distribution of Deviations for Single Sourced Limewire Downloads

The previous data has shown that for most downloads, relatively accurate meta-data predictions can be generated with only a small number of samples. These results were gained using only 10 downloads and therefore it is easy to perceive the accuracy increasing with larger data sets (i.e. taking place over

longer periods of time). It is now necessary to validate that the occasional inaccuracies previously discussed are not related to a lack of past experience (i.e. not enough previous downloads). This is because such a finding would indicate that the approach would be unsuitable for applications that do not perform many downloads. To study this, Figure 3.16 shows the time series for deliveries from single sources. This shows the deviations of the predictions for each download over time. It immediately becomes evident that the extremely inaccurate results are not correlated over time; instead, inaccuracy occurs throughout the node's lifetime. The vast majority of predictions fall within one factor of the actual throughput achieved and the distribution of these results is spread evenly over the entire time series. In fact, the first extreme deviation from the prediction occurs after 9 downloads. From this it can therefore be concluded that the history-based predictions can be quickly and accurately generated and that the deviations are not caused by general problems with the linear function but, rather, from the existence of highly unpredictable extreme cases. These highly unpredictable cases, however, can be rapidly detected by connecting to such nodes before re-selecting another source.

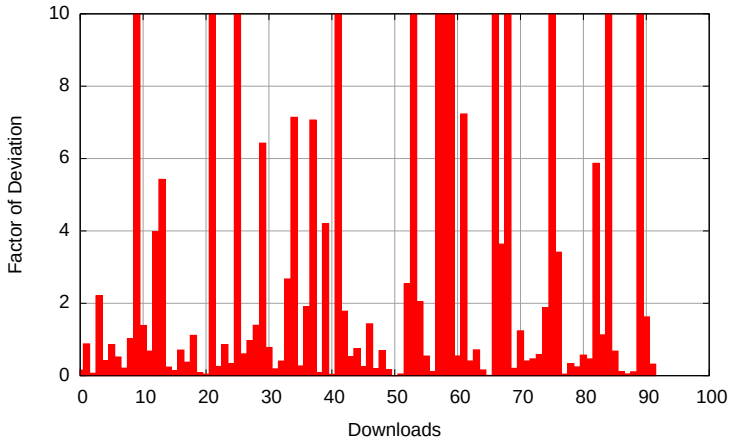


Figure 3.16: Deviations of Predictions over Time for Single Source

3.4.6 Summary

In summary, Limewire has been shown to have large variations in performance when operating in different environments. This performance is defined by two parameters, (i) the number of available sources, and (ii) the HTTP characteristics of each source. Generally, a download with a greater number of sources will achieve higher performance than a download with fewer. However, this is also dependent on the quality of each available source based on the HTTP parameters discussed earlier.

These observations prove that the performance of Limewire will vary when accessing different items of content. Popular content will generally be highly accessible with many sources, whilst unpopular 'long-tail' content will be difficult to access in a reliable manner. Further, considering that deliveries are performed between peers using HTTP, any variations in the parameters detailed in Section 3.2 will also result in changes in performance. Both of these observations therefore result in *consumer* and *temporal* variance in Limewire.

It has also been shown that using history-based predictions can offer an accurate method of modelling future throughput based on the number of available sources. By performing a time-series analysis of the measurement data, a strong correlation has been found between the different downloads, which can be calculated using simply a moving average.

3.5 Conclusions

This chapter has explored the diversity of three prominent delivery protocols utilised in the Internet. This has focussed on identifying and inspecting important runtime parameters that make the design-time selection of delivery protocols suboptimal. HTTP, BitTorrent and Limewere were investigated to ascertain how their performance varies with certain key parameters. It was confirmed that each protocol could be modelled using important runtime parameters that could be easily collected by nodes. Consequently, when faced with a choice between multiple providers using these protocols, it was shown that a node could only make an optimal decision if it were allowed to do so on a per-node and per-request basis. To summarise, the following conclusions can be drawn from this chapter,

- HTTP performance is highly dependent on (*i*) the available server resources and (*ii*) the connection delay; these parameters can be captured and modelled to predict the performance that could be gained through utilising a given HTTP provider
  - The variability of HTTP has been measured and quantified, before detailing and validating a suitable model
- BitTorrent performance is highly dependent on (*i*) the seeder:leecher ratio and (*ii*) the peer upload capacity; these parameters can be captured and modelled to predict the performance that could be gained through utilising a given BitTorrent swarm
  - The variability of BitTorrent has been measured and quantified, before detailing and validating a suitable model

- Limewire performance is highly dependent on (*i*) the content replication level and (*ii*) the HTTP parameters for each connection to the sources; these parameters can be captured and modelled to predict the performance that could be gained through utilising a given set of Limewire sources
  - The variability of Limewire has been measured and quantified, before detailing and validating a suitable model
- Many parameters change (*i*) between different consumers (consumer variance), and (*ii*) over time (temporal variance)
  - An optimal decision regarding the use of a delivery protocol with parameters that change between different consumers cannot be made on a system-wide basis (i.e. it must be made on a per-node basis)
  - An optimal decision regarding the use of a delivery protocol with parameters that change over time cannot be made before request-time
- When an item of content is provided through multiple means, both consumer and temporal variance mean an application cannot be optimised by statically selecting providers/protocols at design-time
  - It is necessary to allow an application to make such decisions at request-time on a per-consumer basis
  - It is necessary to allow an application to re-make such decisions after the delivery has started to address any later temporal variance

This chapter has identified and quantified the extreme levels of heterogeneity that can be observed in delivery systems. Through this, it has been shown that delivery optimisation can only be achieved through allowing provider/protocol selection to be performed at request-time on a per-consumer basis. The next challenge is therefore to devise a mechanism by which this can be exploited to offer delivery-centric access to content. The next chapter builds on the present one to offer a middleware design that supports delivery-centricity through the dynamic selection of protocols and providers.

Chapter 4

A Middleware Approach to Delivery-Centric Networking

4.1 Introduction

Chapter 2 has provided a holistic overview of content network heterogeneity, whilst Chapter 3 has given a more detailed quantitative analysis of delivery system dynamics. These previous chapters have identified that there is a significant number of existing distribution systems that possess a range of divergent characteristics. In regards to content-centricity, two conclusions can therefore be drawn: (*i*) statically selecting providers/protocols at design-time will result in suboptimality due to consumer and temporal variance, and (*ii*) deploying content-centric solutions as stand-alone systems that do not interoperate with existing content systems ignores a huge opportunity in terms of content and resources.

This chapter proposes the use of a new approach to building a content-centric system, which introduces the concept of delivery-centricity into the existing abstraction. Instead of placing functionality in the network, a middleware is developed called *Juno* [141]. Using component-based adaptation, a mechanism is devised that allows an extensible range of divergent content distribution systems to be interoperated with. This ability is then exploited to dynamically resolve and satisfy application-issued delivery requirements by (re-)configuring between the use of different content providers. This therefore offers a mechanism by which content-centric and delivery-centric networking can be instantly deployed alongside current and future content providers.

This chapter explores the design of the Juno middleware. First, the observations of the previous two chapters are explored to construct a set of important design requirements. Following this, an abstract overview of the design is given

based on these requirements. Next, the core design principles and framework are explored, highlighting the fundamental technologies used in the design of Juno. Last, the content-centric and delivery-centric functionality of Juno is explained, showing how it utilises the previously discussed principles to build a content-centric service that fulfils the requirements of this thesis.

4.2 Requirements

The previous section has made two observations regarding current approaches to content-centric networking: (*i*) statically selecting providers/protocols at design-time will result in suboptimality due to consumer and temporal variance, and (*ii*) deploying content-centric solutions as stand-alone systems that do not interoperate with existing content systems ignores a huge opportunity in terms of content and resources. From these, it is possible to derive a set of requirements that can be used to shape a proposed content-centric and delivery-centric solution. This section details these requirements.

First, the solution must offer a content-centric abstraction (such as [54]) that satisfies the functional requirements of content-centric networks as they currently stand. This involves the ability to issue a network request using simply a content identifier, without any reference to potential sources of the data. This abstraction must be provided to applications in a simple, transparent and open manner. As well as this, it must also be offered in a way that can reach a wide audience without the need for modification to extensive software such as operating system network stacks.

Second, the solution must extend the concept of content-centric networking to include delivery-centricity. This means that an abstraction must be designed to allow applications to express delivery requirements to the system. These must then be fulfilled dynamically based on current operating conditions without extensive or complicated involvement by the application.

Third, a solution must be able to exploit the content and resources of existing systems, i.e. the solution should be highly interoperable. It is important that this occurs seamlessly from the application's perspective to avoid introducing complexity into the abstraction or limiting its use to more sophisticated applications. This means that users of the abstraction should not have to provide complicated or extensive information to allow interoperability. Clearly, this must also occur in a low overhead fashion; for instance, it would be unscalable for every content system to install support for every protocol.

Fourth, the solution must avoid the deployment difficulty of previous networked systems such as multicast and quality of service (QoS) [34]. It should therefore not require the immediate global uptake of all parties (i.e. it should be progressive). Similarly, it should not depend on third parties such as first-tier

ISPs that are likely to be uncooperative. Further, it should be designed without the use of expensive infrastructure that will likely act as a financial barrier.

Last, the solution must take a flexible and extensible approach to allow for the later introduction of new technologies. This must be achieved in both the content-centric abstraction and the underlying implementation. The solution must therefore be capable of expanding its capabilities to address the development of new protocols and communication architectures.

In summary, these requirements can be defined as,

1. A solution must offer a content-centric abstraction
    - The abstraction must allow applications to generate network queries for uniquely identified content, without any location-oriented details
    - The abstraction must be openly available to applications
    - The abstraction must not be difficult to deploy or restricted to a small subset of hardware, software or operating systems
2. A solution must provide delivery-centricity by supporting the stipulation of delivery requirements by the application (e.g. relating to performance)
    - Delivery requirements should not need to be associated with individual delivery schemes
    - Delivery-centricity should not need the application to provide any information beyond its abstract requirements
3. A solution must support interoperation with existing content distribution systems
    - Interoperation must be seamless from the application's perspective
    - Interoperation should not require the provision of extensive or obtuse location information from the application
    - Interoperation should not have high resource requirements that are significantly beyond current delivery protocols such as HTTP
4. A solution must be progressively deployable without significant barriers
    - Its deployment must not require ubiquitous uptake by any set of parties (providers, consumers, ISPs)
    - Its deployment must not require the support of (potentially) uncooperative third parties (e.g. ISPs)
    - Its deployment must not require the construction of expensive infrastructure

5. A solution must be flexible and extensible to support future technological changes and advances
    - The abstraction and interoperability mechanism must support the addition of new content systems
    - The abstraction must support the introduction of future access mechanisms (e.g. interactive content)

This rest of the chapter now utilises this requirement set to drive the design of a new middleware solution to handling these challenges.

4.3 Juno Middleware Overview

Before inspecting the strict technical details of the Juno middleware, it is important to gain an understanding of its general design and operation. This section provides a brief abstract and technical overview to provide a foundation for exploring the finer details of Juno.

4.3.1 The Case for Content-Centric Middleware

The previous section has detailed a number of core requirements that must be fulfilled by the design. As previously discussed, existing network-level solutions have a number of flaws. Chapter 2 analysed three prominent content systems to find that none of them successfully satisfy the research goals of this thesis. Namely, relating to delivery-centricity, interoperability and deployability. These problems emerge due to the inflexible and static nature of network-level designs such as DONA [93] and AGN [83]. This means that they often fail to offer the type of capabilities offered by previous application-level solutions.

First, the construction of content delivery at the network-level prevents *delivery-centric* requirements from being satisfied. This is because such approaches cannot be modified to respond to varying user needs, as the content-centric network is viewed largely as a routing infrastructure. This means, for instance, that it cannot be made aware of content quality issues or higher level needs such as monetary cost. Second, the introduction of *interoperation* at the network-level is also largely unfeasible. This is because all major content distribution mechanisms currently operate at the application-level. Consequently, existing content-centric designs would require these systems to modify their behaviour to become visible. Unsurprisingly, large organisations such as Akamai have little incentive to do this, thereby leaving content-centric networking as an unused technology. Finally, this is also exacerbated by the cost of new network infrastructure, making *deployment* extremely difficult.

From this, it is evident that current network-level solutions fail to fulfil the requirements of this thesis. A number of similarities, however, can be drawn be-

tween content-centric networking and existing middleware designs. Both attempt to ease the development burden of applications by offering a simple abstraction that fulfils a set of requirements. Further, both attempt to hide underlying distributed complexities through the use of such an abstraction. A logical question is therefore whether or not there would be any benefit from attempting to introduce the content-centric paradigm at the middleware-layer? Intuitively, the key benefit would be to provide a far greater degree of flexibility through the removal of network-level restrictions. This is because it would be entirely built in software, therefore making it easy to interact with existing (application-level) infrastructure. This would have two advantages; first, it would allow interoperation with existing protocols, thereby making existing content providers instantly visible; and, second, by allowing the use of existing infrastructure, it would allow instant deployment. This flexibility could even potentially allow the introduction of delivery-centricity through the exploitation of the heterogeneity observed in different protocols and sources.

Based on these observations, a middleware approach has been taken to addressing the research goals of this thesis. The rest of this section now explores the design of Juno, which offers a content-centric and delivery-centric service at the middleware-layer.

4.3.2 Abstract Overview

Section 4.2 has provided a set of important requirements to base the solution on. These cover primary functional aspects (*what* the system must do) as well as other non-functional properties (*how* it should be done). In essence, these require that the content-centric solution can interoperate with existing content distribution mechanisms and exploit their diversity to select the one (or more) that best fulfils the delivery-centric requirements of the application. Certain constraints are also placed on this process to ensure that it can be feasibly deployed with limited overhead and support for future advances in technology.

Current approaches that operate within the network cannot achieve this because they consider content-centric networking as simply a routing process that forwards requests to the closest instance of content. This clearly prevents such an approach from interoperating with application-level protocols. This thesis therefore proposes the use of a *middleware* approach for building a content-centric communications paradigm. As such, a content-centric middleware is outlined called *Juno*. Instead of modifying operating system and router capabilities, content-centric functionality is placed within a middleware implementation that resides between the operating system and application. Juno uses this position to transparently translate content-centric requests generated by the application into location-oriented requests that can be passed into the operating system using traditional sockets. To achieve this, Juno maintains connectivity between

itself and an extensible number of available content distribution mechanisms. Through this, it overlays content-centric addressing onto these systems and accesses content from each system using their individual protocol(s) to provide it to the application in an abstracted manner.

Figure 4.1 provides a simple overview of this behaviour. The application interacts with Juno using a content-centric interface; as such, it sends a content-centric request and then receives an abstracted response with the content it desires. However, below this, Juno converts the request into a traditional location-oriented address that is interoperable with the existing widespread content infrastructure in use (e.g. a web server). Juno then interacts with these external parties to gain access to the content so that it can subsequently be returned to the application. This process is made feasible using a number of technologies, which are explained within this chapter.

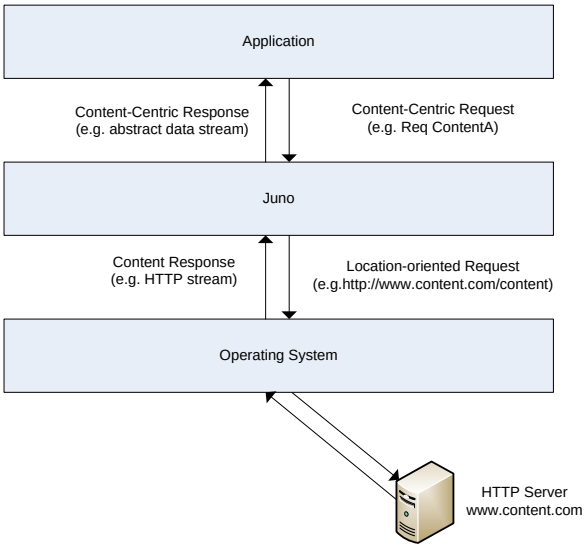


Figure 4.1: An Abstract Overview of Juno's Behaviour

4.3.3 Technical Overview

Juno is a component-based middleware that exploits software (re-)configuration to enable interoperation with a number of different content systems. It exposes a standard content-centric (and delivery-centric) abstraction to applications whilst dynamically re-configuring to download content from existing content infrastructure that offers the desired content in the optimal way. As such, it enables content-centricity to be instantly deployed without the need for new and expensive infrastructure. Considering that Juno is a configurable component-based middleware, the most important aspect of its design is its software architecture.

This is the way in which Juno's constituent components are built and interconnected to successfully offer a content-centric abstraction. The rest of this section provides an architectural overview of Juno alongside a more detailed description of its technical operation.

Juno's design consists of three primary architectural elements: content management, discovery and delivery. These elements are manifested through components capable of independent manipulation, as shown in Figure 4.2. Specifically, these three elements are the Content Manager, the Discovery Framework and the Delivery Framework. These are each managed through the Content-Centric Framework, which interacts with the application to receive requests and return content. These elements are now each discussed in turn.

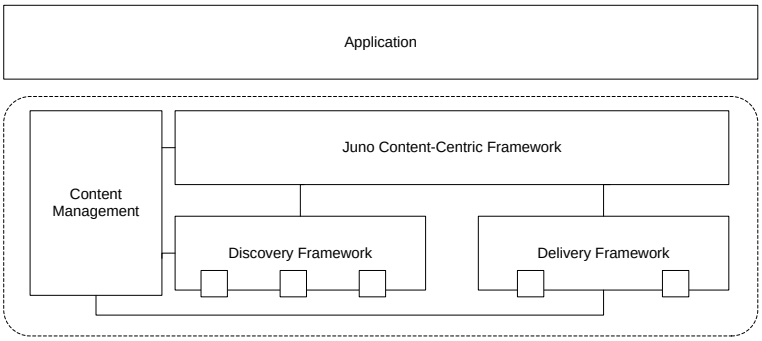


Figure 4.2: An Overview of Juno's Software Architecture

The Content Management component in Juno is responsible for handling the local management of file system content. It provides methods to lookup, retrieve and manipulate local content. To be truly content-centric this must be considered as a primary entity in the system, allowing a full library to be locally shared. When Juno receives requests for content from an application, the Content Manager is first checked to find out whether a local copy is available. Clearly, this is the ideal situation that would occur if a previous application has accessed the content. The Content Manager also provides standard methods to allow Juno to write content to disk when it is receives from remote sources. Similarly, it allows data to be streamed directly to the application without explicit knowledge of the underlying delivery scheme. This therefore makes the development of new delivery protocols easier.

When content is not locally available it is necessary to access it from a remote source. The first step is therefore to locate a set of available sources; in existing content-centric networks this is simply performed by sending a request packet into the network (which is responded to with data from wherever it is routed to). In Juno, however, this process is more intricate and is managed by the *Discovery Framework*. This framework is provided with a content identifier and is then responsible for locating as many valid sources as possible. It does this

by attaching and utilising a set of components called *discovery plug-ins*, shown as small boxes connected to the framework in Figure 4.2. These are components that implement a standard discovery interface that allows the framework to pass it a content request and then receive, in return, a set of sources (host address, port, supported protocol(s), remote identifier(s)). Each plug-in offers the functionality to access the query service of a particular discovery system that indexes content for a set of providers. This process therefore provides the consumer with a choice of possible sources from a number of providers that are addressed by their traditional location (i.e. their IP address).

Once a consumer possesses a number of sources, the next step is to connect to one or more and request the content. This is the responsibility of the *Delivery Framework*. The first step is to select which source(s) to utilise; this decision is driven by the delivery requirements originally issued by the application. In its simplest form, this may be just to get the highest performance, however, this could also extend to other issues such as security, reliability and monetary cost. The Delivery Framework manages a number of *delivery plug-ins* that each offer the functionality to access content from a particular provider or protocol. Plug-ins are required to generate runtime meta-data relating to their ability to perform a particular delivery. As such, the framework can query this meta-data to compare each plug-in against the requirements of the application. Once a decision has been made, the framework dynamically attaches the optimal plug-in and begins the download.

Finally, once this has completed the application is provided with a handle on the content and is allowed to access it through one of Juno's standard abstractions. These abstractions allow the application to view the content using whichever abstraction best suits its needs (e.g. file handle, input stream etc.). This functionality is all underpinned by the *Juno core*; this is a set of important interfaces, classes and components that are required for the middleware to operate. The next section details these fundamental aspects of Juno. Then, following this, a detailed description of the content-centric functionality of Juno is provided.

4.4 Juno Design Principles and Core Framework

This section details the fundamental principles utilised by Juno to support the operation of the content-centric service it offers. First, an overview is given of Juno's core design principles. Following this, the concepts of components and services are detailed, alongside the important interfaces and objects used for building them. Last, these aspects are brought together to show how Juno exploits the unique properties of components and services to perform configuration and re-configuration to adapt system behaviour.

4.4.1 Overview and Design Principles

Juno is designed using three important software engineering principles,

- *Components:* A component is a fixed body of functionality that offers predefined services as well as exposing explicit dependencies
- *Services:* A service is an abstract piece of functionality (as represented by a programming interface) that does not have a fixed underlying implementation
- *Reflection:* Reflection is the ability to dynamically introspect a body of code to ascertain and manipulate its operation

These three principles provide the foundation for Juno's operation. Juno is a component-based configurable middleware. This means that functionality within Juno is separated into independent pluggable entities called components, which collectively build up the system. A component is an element of functionality that offers well-defined services alongside explicit dependencies. This definition makes components a powerful approach for building software systems in a (re-)configurable and reusable manner. This is because, unlike objects, components can be introduced (or removed) safely as long as all dependencies are satisfied [138].

The service(s) a component offers are defined by one or more interfaces; interface naming follows the COM [7] convention by prefixing interface names with the letter 'I' to represent 'Interface' (e.g. IStoredDelivery). These interfaces describe the capabilities of a component and allow external bodies to ascertain the functionality that it supports. The dependencies of a component are similarly defined by interfaces, allowing a component to explicitly state the services it needs to consume. Subsequently, explicit bindings exist that allow one component's dependency to be satisfied by another component's service.

To enable (re-)configuration, these explicit bindings can be manipulated during runtime to change the way components are interconnected. By doing this intelligently, the system can be adapted by introducing or removing components to reflect current operating requirements. To differentiate components that offer the same service, reflection is used. Reflection is the process of introspecting a software implementation and allowing it to make reasoned choices about itself. Of particular interest to Juno is the concept of *structural reflection*; this is the process of introspecting the underlying software structure of a system. In terms of a component-based system, this involves dynamically discovering and manipulating the component architecture of the middleware.

To support this process it is vital that Juno can comprehend not only the service capabilities of a component (i.e. its interfaces) but also its non-functional aspects (i.e. how it operates). To achieve this, reflection also involves the explicit

stipulation of important meta-information relating to each component, as well as the system as a whole. Within Juno, every component is required to expose runtime information about itself regarding its behaviour, performance and overheads. This information is then used to dynamically select the best components to utilise based on requirements issued by the application (and the environment). To make this process less complicated, components are generally grouped together into a *component configuration*. This, in essence, is a description of how a set of components can be interconnected to collectively offer a service. This therefore allows meta-data to be generated and exposed in a coarser-grained manner, making optimisation more scalable and requirement descriptions simpler.

The previous principles are used to optimise system behaviour by selecting the best component implementation to provide a particular desired service. The process of doing this at request time (i.e. when a service is requested) is termed *configuring* a service. However, equally important is the need to address later variations that could potentially make that decision sub-optimal. This process is termed *re-configuring* a service and involves removing an old component and replacing it with a different one. This occurs when something changes in the environment (e.g. network congestion) or, alternatively, when the requirements of the application change (e.g. it requires a different security policy). To enable this, the meta-data exposed by each service is monitored as well as the requirements of the application. If either changes, the selection process is re-executed to find a superior component configuration. Through these principles, it becomes possible to build a flexible, extensible and adaptable system that can evolve with the changing needs of its environment.

4.4.2 Components in Juno

This section details the use of components in Juno. Specifically, it introduces the OpenCOM component model before detailing the important standard objects and interfaces that are used when developing and integrating components in Juno.

Overview of Component Models

Components are the lowest level building blocks in the Juno middleware. A component can be defined as,

> "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*" [134]

From this, it can be seen that a component goes beyond traditional object-oriented programming to introduce explicit bindings between the different soft-

ware entities (i.e. components). Consequently, it is necessary to utilise a *component model* to manage how components are constructed and interact. A component model is a framework used for designing and building software components; this usually consists of a set of standard objects and interfaces that must be used to turn a body of code (e.g. a Java class) into a component. Various component models exist to provide the necessary support for operating in different environments. However, for Juno, a number of key requirements can be identified,

1. *Open and reflective:* The component model must allow the capabilities of components to be (dynamically) discovered and modified to handle variations in the environment

2. *Runtime (re-)configurable:* The component model must support the dynamic (re-)configuration of the constituent components and their interconnections

3. *Lightweight:* The component model must operate alongside applications and therefore it is vital that it is lightweight in terms of processing and memory overheads

A number of component models exist for developing software; to understand the choices made in this thesis, a brief overview of some of the potential models is now given,

- *Component Object Model (COM):* This is a language neutral component model developed by Microsoft; components implement the QueryInterface function to allow external parties to gain access to their functionality (exposed through interfaces). However, COM does not offer any reflective capabilities to allow re-configuration, making it unsuitable [7].

- *.NET:* This is another component model developed by Microsoft; it offers a number of feature including interoperability support, a range of class libraries and security. However, this means .NET is heavyweight, making it unsuitable [20].

- *Koala:* This is a component model developed by Philips targetted at embedded systems. This makes it a lightweight system, however, like many other lightweight models (e.g. nesC [64]) it is only build-time aware, without any support for handling components during runtime [109].

- *OpenCOM:* This is a lightweight component model developed at Lancaster University for building systems software. Components and bindings are treated as explicit runtime entities that are managed by OpenCOM. This allows dynamic re-configuration to take place in an open manner [51].

From this, it is evident that OpenCOM best suits the requirement previously mentioned. As such, it has been selected to form the basis for developing the Juno middleware. The following section now provides an overview of the OpenCOM component model.

The OpenCOM Component Model

OpenCOM [51] is a lightweight reflective component model used for implementing systems software such as middleware. An overview of OpenCOM is given in Figure 4.3. OpenCOM is based on the principles of *interfaces*, *receptacles* and *connections*. An interface expresses the ability of a component to offer a particular service; the component in Figure 4.3 can be seen to offer the ICustomServiceInterface. A receptacle is an explicit dependency that must be satisfied by another component's interface; this can be seen by the ICustomDependency. Last, a connection is an explicit binding between an interface and a receptacle, allowing one component to satisfy the dependency of another. These are managed by the OpenCOM Runtime, which allows components to be created, interconnected and deleted dynamically. A system built using OpenCOM therefore consists of a set of components that are interconnected using their interfaces and receptacles. The OpenCOM Runtime then provides the necessary methods to query these components to retrieve the required functionality.

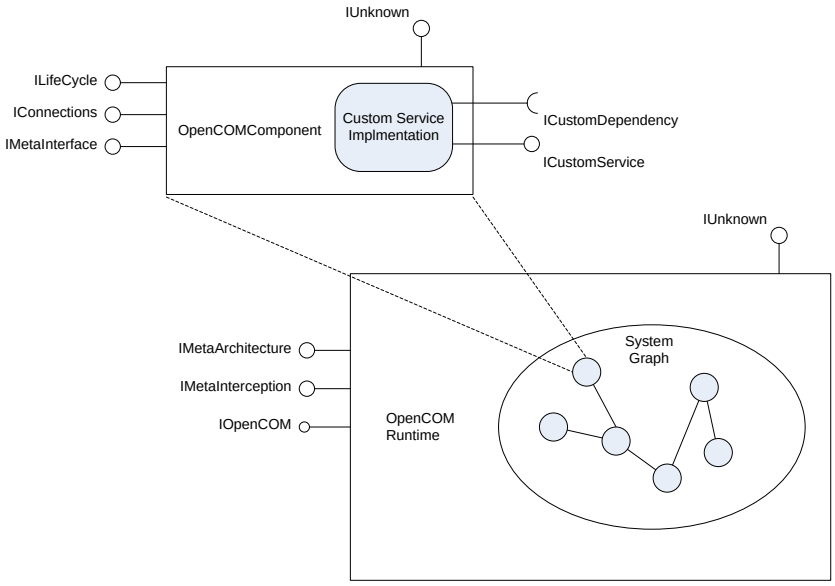


Figure 4.3: An Overview of the OpenCOM Runtime

A key characteristic of OpenCOM is its support for reflection; this is the process of allowing introspection and manipulation of software during runtime. The OpenCOM Runtime achieves this by utilising specific reflective interfaces

implemented by each component. As shown in Figure 4.3, the OpenCOM Runtime maintains a dynamic system graph of the components currently operating within its control. By using explicit bindings this allows the system graph to be manipulated during runtime to modify the behaviour of the system. This graph is termed the *meta layer* and exists as a meta representation of the actual implementation (the base layer). Importantly, these two layers are loosely connected so that changes made to the meta-layer are also reflected in changes in the base layer. To enable the OpenCOM Runtime to achieve this control, it is necessary for each component to implement a set of interfaces. There are four main interfaces that must be implemented by OpenCOM components,

- *IUnknown:* Allows a component to be queried to retrieve a handle on a particular interface it offers
- *ILifeCycle:* Allows a component to be started or shutdown
- *IConnections:* Allows a component's connections to be dynamically modified
- *IMetaInterface:* Allows a component to be inspected to ascertain the values of meta-data associated with it

Through these interfaces, the OpenCOM Runtime can manage a pieces of software to (*i*) configure it by adding or removing components at bootstrap, or (*ii*) re-configure it by adding or removing components during runtime. These principles allow a piece of OpenCOM software to be both adapted and extended through (re-)configuration. This functionality is exposed through the IOpenCOM, IMetaArchitecture and IMetaInterception interfaces, as shown in Figure 4.3. Beyond this, the use of components also offers a management structure so that functionality can be effectively indexed, utilised and extended. OpenCOM is implemented in Java, making it platform independent and easy to distribute. It has a limited overhead of only 32 KB for the Runtime and 36 bytes for a null component.

Core Component Objects

OpenCOM provides the underlying component model to turn traditional Java objects into components. In essence, this solely consists of turning references between objects into explicit bindings that can be manipulated. OpenCOM can therefore be considered as a lightweight model that only provides the base underlying functionality to construct a software system. This makes it necessary to extend certain aspects of OpenCOM to make it appropriate for Juno's requirements. Juno therefore provides a set of important objects that offer some form of utility function. Some of these are optional whilst others are necessary for

interaction with the Juno framework. This section details the primary objects utilised by components operating in the Juno Framework.

StateEntity. In Juno, (re-)configuration takes place by dynamically attaching or detaching components. This, however, can result in errors when re-configuring components that contain state. Therefore, to ease the process, components in Juno only contain explicitly defined state. This allows state to be removed from a component and placed into another without complexity. To support this, Juno uses the StateEntity object; this is an abstract object that represents a collection of state within a component. When developers build a component, they must also build a state object containing all the related state information; this object must extend StateEntity. To accompany these component-specific state objects, there are also service-specific state objects that are associated with particular service interfaces. State therefore becomes a logical extension to the interface definition, allowing components implementing the same interface to exchange state easily. This is vital for re-configuration as it allows a component to be shutdown and replaced without the loss of state. It also allows a State Repository to be built up allowing introspection into any component's current state.

Parameters. Most components require parameters to specialise their behaviour or to help during bootstrap. Parameters is an abstract, reflective class that developers can extend to encapsulate their parameter sets. Developers simply extend the object and create a set of public instance variables. Corresponding files can subsequently be created that contain name:value pairs for each of the public variables. The Parameters object provides a load(String filename) method, which can then load the parameters from the file automatically using native Java reflection. Like with StateEntity objects, this similarly allows a State Repository to be built up containing each component's current parameter set.

Bootstrap. When a component is initiated it usually requires certain pieces of information for bootstrapping. This is encapsulated in the Bootstrap object, described in Table 4.1. It provides a component with a unique identifier, a set of parameters and the appropriate StateEntity object(s). The Bootstrap object is passed to the component when it is initiated.

JunoComponent. Due to the complex and repetitive nature of developing standard framework functionality in a new component, it is necessary to ease the developer's burden. This is achieved using the abstract JunoComponent object, which performs nearly all functionality related to integration with Juno. This generally consists of implementing generic functionality for many of the required interfaces that will be detailed in the following section; namely,

Method	Returns	Description
addStateEntity(StateEntity entity)	void	Adds a StateEntity object to the Bootstrap
addParameters(Parameters p)	void	Adds a Parameters object to the Bootstrap
setServiceID(String id)	void	Provides the component identifier to be used by the component

Table 4.1: Overview of Bootstrap Object (exc. corresponding get methods)

- Service Acquisition: It handles requests for a given service from the component (IUnknown)
- Bootstrap Management: Using native Java reflection it automatically loads all Parameters and StateEntity objects into the component (IBootable)
- Service Publication: When a component is initiated it loads references into Juno's Configuration Repository (c.f. Section 4.4.3).
- Connection Management: It handles requests for the interconnection of components (IConnections)
- Meta-Data Management: It handles the setting and getting of meta-data associated with the component (IMetaInterface)
- State Management: Using native Java reflection it automatically handles all requests to inspect state and parameters (IOpenState)

Components can therefore extend JunoComponent to gain support for the above functionality without having to implement anything. This dramatically decreases the overhead of developing components in Juno.

Core Component Interfaces

The majority of the objects discussed in the previous section are related to a set of framework interfaces that allow components to interact with Juno. This section details the most important of these interfaces. It is important to note that all the interfaces discussed in this section are already implemented by the JunoComponent abstract object and therefore developers can simply extend this with limited effort.

IUnknown. This is a simple interface implemented by all components within Juno, as shown in Table 4.2. It allows a component to be queried for its supported interfaces (i.e. the services it offers). Importantly, every service interface in Juno

Method	Returns	Description
QueryInterface(String intf)	Object	Returns an object reference to the component's requested interface (if available)

Table 4.2: Overview of IUnknown Interface

must extend this interface to allow its integration into the Juno middleware.

IBootable. This interface allows a component in Juno to be initiated by the framework. Its primary function is to offer access to life cycle management. It allows a component to be initiated and shutdown as well as resumed in circumstances in which the StateEntity object already has data in it. There are four steps in a component's life cycle, each controlled by a method in IBootable (shown in Table 4.3) - three at its initiation and one at the end:

1. *Initiation:* Creates a new component and provides it with the necessary Bootstrap object
2. *Configuration:* Performs any inter-component bootstrapping (the components must be interconnected before this can happen). Instead of this, a component can also be *resumed*; this is performed when it is initiated with a StateEntity object that contains existing data
3. *Execution:* Begins the execution of the component's operation (e.g. listening on a port)
4. *Shutdown:* Performs an elegant shutdown of the component

IConnections. Any component that has dependencies on other components must implement the IConnections interface, as shown in Table 4.4. This interface allows a connection to be formed between two components: one offering a service and one consuming it. It also allows the given connection to be later disconnected.

IMetaInterface. Juno relies on various types of reflection to support its operation. The IMetaInterface is implemented by all components to allow meta-data to be associated with them in the form of attribute:value pairs. IMetaInterface provides get and set methods to access this information dynamically.

IOpenState. A further reflective mechanism that components offer is the ability for their state and parameters to be inspected and modified. This is highly beneficial when it becomes necessary to alter the behaviour of a specific component. It is particularly important when re-configuring between the use of

Method	Returns	Description
initiate(Bootstrap boot)	boolean	Initiates a component with the necessary parameters and state for its configuration
configure()	void	This method is used when there must be a delay between initiating the object and it being configured (e.g. if two components depend on each other and therefore must both be initiated before they can be executed)
execute()	void	Begins the component's execution if it has background tasks that it must always perform (e.g. listening on a socket)
resume()	void	This method is used when the component's StateEntity object already has data in and therefore the component must resume using this data (e.g. if it is replacing a previous component)
shutdown()	boolean	Safely shuts the component down

Table 4.3: Overview of IBootable Interface

Method	Returns	Description
connect(IUnknown sink, String intf, long connID	boolean	Connects a component (sink) to this component using a particular interface with a unique connection identifier
disconnect(String intf, long connID)	boolean	Disconnects the component (of type intf) with a given connection identifier

Table 4.4: Overview of IConnections Interface

different components. This is because it allows the state and parameters of the original component to be extracted and put into the new component. IOpenState allows parameters (represented by the Parameters class) and state (represented by the StateEntity class) to be added or modified in a component. Alongside this, it is also possible to dynamically inspect a component to ascertain what types of state and parameters it supports. Details of the IOpenState interface are shown in Table 4.5.

4.4.3 Services in Juno

The previous section has detailed the operation of components in Juno. The second important principle is that of *services*. A component is a self contained body of code that exposes well defined interfaces and dependencies. A service, on the other hand, is an abstract version of a component that only exposes an

Method	Returns	Description
getOpenState()	Vector <StateEntity>	Returns a vector of state objects associated with the component
getOpenState(String stateType)	StateEntity	Returns the StateEntity object associated with the type
setOpenState(String stateType, StateEntity newState)	void	Assigns a given state object to the component
getOpenStateTypes()	Vector <Class>	Returns a vector containing the StateEntity classes of the component
getOpenParameters()	Vector <Parameters>	Returns the parameters associated with the component
getOpenParameters-Types()	Vector <Class>	Returns a vector containing the Parameters classes of the component
setOpenParameters(String parameter-sType, Parameters newParams)	void	Sets the given parameters variable in the component to a new Parameters object
getOpenParameters(String parameterType)	Parameters	Returns the Parameters object of the given type

Table 4.5: Overview of IOpenState Interface

interface without any predetermined implementation. This section now details the concepts behind services in Juno as well as listing the important elements relating to their management.

Overview of Services

A service is an abstract piece of functionality that is open to configuration and re-configuration. It is defined by a fixed service interface that exposes some form of functionality to external entities. It is different to a component in that it does not specify any underlying implementation. Instead, a service can be re-configured by dynamically changing the implementation that resides behind the abstraction. These implementations consist of one (or more) pluggable components. This is shown in Figure 4.4; on the left hand side is a component implementation of ICustomService, which is exposed to external entities through the abstract interface. External entities therefore only interact with this component through the *service*; this therefore allows the underlying component implementation to change without the knowledge or involvement of any external parties. This further allows composite services to be composed by interconnecting multiple services to create

more sophisticated ones.

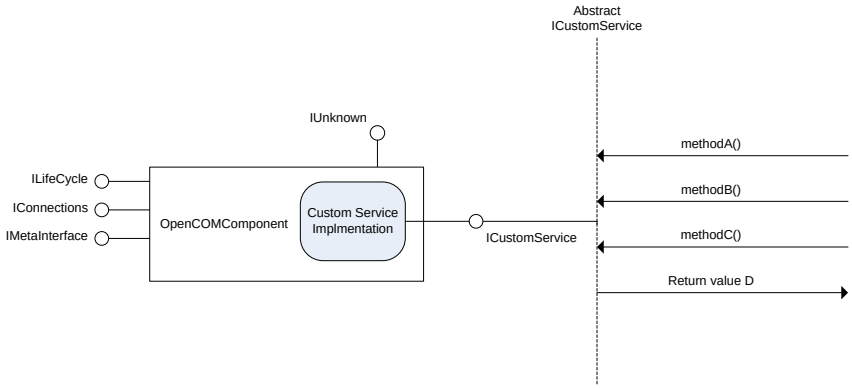


Figure 4.4: The Difference between a Component and a Service

When an application uses the middleware, it exclusively interacts with services as the underlying implementation is not fixed. Instead, Juno manipulates the underlying implementation dynamically to best match the application's requirements. This is achieved by placing one level of indirection between all interactions so that method invocations can be routed to the currently attached component (this is managed by OpenCOM).

The primary object relating the the concepts of services in Juno is the Configurator. This is a container that manages one or more components to offer a predefined service. Importantly, these components are not fixed and can be dynamically changed within the Configurator container without the knowledge of any external parties. This is achieved with the Configuration object, which is used to define how one or more components can be used to offer the desired service. As such, different Configuration objects can be passed into the Configurator container based on application and environmental requirements.

Core Objects, Interfaces and Components

A service, in itself, simply consists of an interface definition, however, surrounding these are three core objects that support their utilisation. This section now outlines these important objects.

The Configuration Object. Often a component only offers a subset of the necessary functionality for a complete service. For instance, a content download service might consist of two components: one that performs the download and one that writes it to file. This allows better code reuse and management, as well as allowing (re-)configuration to take place by modifying the constituent components. To this end, it is necessary to define how a set of components can be interconnected to build an entire service. This description is termed

a *component configuration*; this is represented in Juno using the Configuration object. A Configuration is a Java class that contains the necessary functionality to instantiate and interconnect one or more components to offer a predefined service.

A Configuration object can be accessed and utilised through the IConfiguration interface, as shown in Table 4.6. The execute() method is analogous to a Java constructor; it dynamically builds an instance of the component configuration it represents. When developing a service implementation it is necessary to implement a Configuration by extending the Configuration object and implementing the IConfiguration interface. Within the execute and detach methods, the necessary service-specific code must be inserted to construct and destroy the component configuration. However, the other methods in IConfiguration can be handled by extending the abstract implementations in the Configuration class. The Configuration object also offers IMetaInterface so that each configuration can be associated with meta-data in the same way that components are.

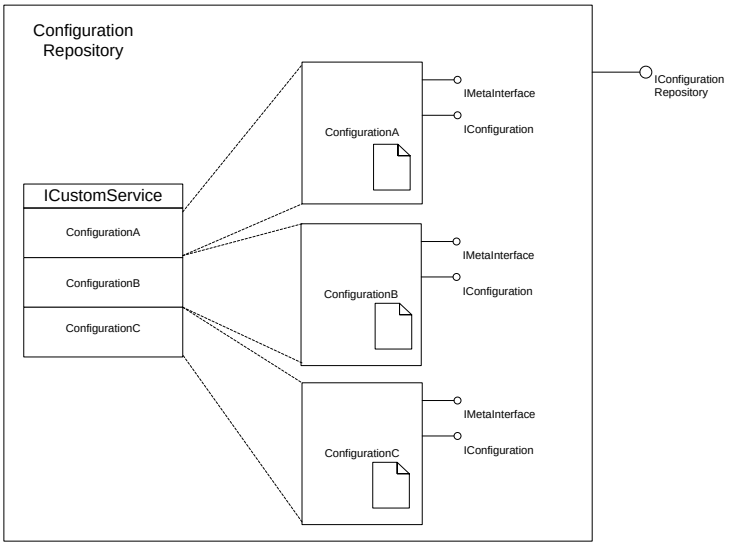


Figure 4.5: The Configuration Repository containing Configuration Objects

The Configuration Repository. The Configuration Repository is responsible for maintaining an index of all the available component configurations, as well as all the implemented components. It provides methods to add and remove components and configurations from the repository. Alongside this, it also offers methods to lookup and retrieve these components and configurations on-demand. A list of all the configurations offering a particular service interface can be retrieved. More sophisticated queries can also be generated for components; these can be looked up based on their interface type, parameter sets and meta-data.

Method	Returns	Description
execute()	boolean	Instantiates and interconnects a set of components and returns the result
detach()	boolean	Detaches and destroys the components that it is responsible for
getExternalInterfaces()	Vector<String>	Returns a Vector of service interfaces that this configuration offers
setConfigurator(Configurator configurator)	void	Sets the Configurator that the Configuration should be built in
getConfigurator()	Configurator	Returns the Configurator that this Configuration is currently associated with

Table 4.6: Overview of IConfiguration Interface

This therefore supports the easy dynamic management of Juno as well as the ability to introduce new components and configurations during runtime.

The Configuration Repository is shown in Figure 4.5. The diagram shows how the repository stores the Configuration objects for a particular service: ICustomService. Each Configuration object is mapped to an entry in an internal database that allows lookups to be performed. The Configuration object exposes the IConfiguration interface as well as the IMetaInterface, allowing each configuration to be queried as to its meta-data.

The Configurator Object. To enable the safe (re-)configuration of services, it is necessary to incorporate domain-specific knowledge into the process. This is the responsibility of Configurator objects, which act as containers for instances of a service to operate in. This is therefore the object that is responsible for implementing the construction and adaptation of the service in a safe and validated manner. Every service, defined by a particular interface, must also be associated with a domain-specific Configurator. When a new service is requested, it is necessary to first instantiate the service's appropriate Configurator, which extends the abstract Configurator object. Following this, a Configuration object can be passed into it, as detailed in Table 4.7. The Configurator is then responsible for instantiating and managing this configuration.

The internal structure of a Configurator object is shown in Figure 4.6; its operation follow a chain in which a Configuration object is passed in and the instantiated service is passed out. When a Configurator receives a component configuration to instantiate, it must first validate that it is correct. This can be done in a variety of ways and is left to the judgement of the developer. If it

is considered safe, the Configurator builds the new component configuration after performing any necessary domain-specific pre-processing. The Configurator's configure(IConfiguration configuration) method can also be called again later with a new Configuration object. The Configurator is then responsible for managing this *re-configuration* by implementing the necessary changes in the service's component connections. Importantly, unlike containers in other component models such as COM [7], the function of a Configurator is solely to safely manage configuration and re-configuration. As such, it is very lightweight without the need to offer any other extended functionality (e.g. persistence, remote access etc.).

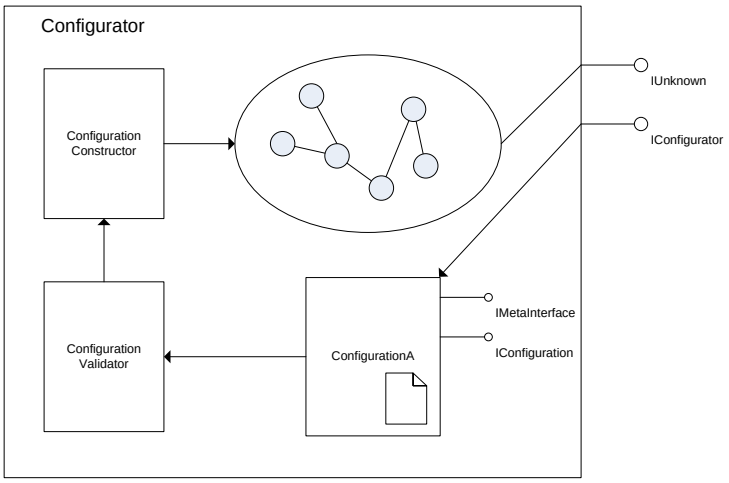


Figure 4.6: The Configurator Container

Once the Configurator has instantiated the necessary components, it is possible for external parties to access its supported functionality. This can simply be performed using the standard IUnknown interface, as detailed in Table 4.2. This allows the Configurator to be requested for a particular interface, which can then subsequently be mapped to a component that can offer it. Importantly, this indirection therefore allows re-configuration to take place without the modification of any object references within external code.

4.4.4 Configuring and Re-Configuring Services in Juno

The previous two sections have detailed the concepts of *components*, *component configurations* and *services*. This section now brings together the principles explained in these sections to show how configuration and re-configuration takes place in Juno.

Method	Returns	Description
getConfiguratorID ()	int	Returns the unique ID of this configurator
getCurrentConfiguration ()	ConfigurationOperation	Returns the current instantiated configuration
configure(IConfiguration configuration)	boolean	Removes the previous configuration (if any) and instantiates the new one
recvConfigurationRequest (ConfigurationRequest req)	void	Informs the Configurator that another node is requesting that its service is adapted (this enables distributed configuration to take place)
getMetaRules()	Collection	Returns the meta-data rules associated with this service
setMetaRules(Collection <MetaDataRule> rules)	void	Sets the meta-data rules associated with this service

Table 4.7: Overview of IConfigurator Interface

Principles of (Re)Configuration

Configuration is the process of dynamically building a service implementation to best meet a set of requirements, whilst re-configuration then later involves modifying this implementation to reflect certain changes in the application or environment. In Juno, both are achieved by dynamically selecting (and re-selecting) which components are utilised to build up a service.

To enable this process, it is necessary for each component configuration to be able to expose its capabilities whilst, equally, it is necessary for an application to represent its requirements. With this information it then becomes the task of the middleware to match its implementation to the requirements provided. This process is managed by a component called the ConfigurationEngine, which accepts requests for particular services alongside the application's non-functional requirements. These requirements are structured as selection predicates for the meta-data associated with the chosen component.

The rest of this section provides a description of each of these elements, showing how they fit together. First, it is detailed how a component or configuration can expose its capabilities, and how an application can then express requirements. Following this, the Configuration Engine is explained, which is responsible for comparing requirements against capabilities to select the optimal configuration.

Representing Capabilities and Requirements

The process of component-based (re-)configuration involves placing different components behind an individual service abstraction so that different implementations can be used in different circumstances. To enable this, it is necessary to be able to (*i*) describe the capabilities of each component/configuration, and (*ii*) describe the requirements of the application (and environment). In Juno this is achieved using *meta-data* and *selection predicates*.

Representing Capabilities. All components and component configurations in Juno are required to expose (*i*) what interfaces (i.e. services) they offer, and (*ii*) related meta-data for each interface that provides non-functional information about the way in which it provides the service (e.g. performance, overheads etc.). This is performed using a standard set of reflective interfaces. All components and component configurations are required to implement the IMetaInterface interface, as shown in Table 4.8. This allows the implementer to associate itself with meta-data in the form of attribute:value pairs. For instance, a component configuration offering a data sorting service might expose, TIME_COMPLEXITY = $O(N)$. Importantly, meta-data is not immutable so it can change dynamically during a component's lifetime.

Alongside the ability to set and retrieve meta-data, it is also necessary for components to generate runtime meta-data. A prominent example is that discussed in Chapter 3, involving the need to dynamically generate performance predictions. This is exposed using IMetaInterface. To assist in this, IMetaInterface also allows a component or component configuration to be pre-loaded with important runtime information so that it can generate its meta-data. Therefore, in the previous example, a HTTP component could be pre-loaded with information such as DELAY = 25. This information can then be used to generate the necessary meta-data that the component should export, e.g. DOWNLOAD_RATE.

Representing Requirements. Up until now, it is only possible for a component or component configuration to expose information about itself. The next step is to allow Juno to exploit this information to drive (re-)configuration. This is achieved by asking applications to couple service requests with meta requirements that inform Juno of their non-functional requirements. In essence, these are selection predicates that dictate the application's preferred values for any meta-data associated with the chosen component configuration that provides the service. For example, an application might request a sort service that has the meta-data, TIME_COMPLEXITY == $O(N)$. This information is represented using the MetaDataRule object. This is a simple object that contains three variables: attribute, comparator and value. It can therefore simply stipulate that the given attribute must have a particular value or range of values based on the comparator; details of the comparators are given in Table 4.9. The attribute

Method	Returns	Description
getAttribute(String intf, String attribute)	String	Returns the value of a specific attribute for a particular exposed interface
setAttribute(String intf, String attribute, Object value)	void	Sets the value of a specific attribute
generateRuntimeMetaData(String intf, Hashtable <String, Object> info)	Hashtable <String, Object>	Generates its runtime meta data for a particular interface based on information provided through a hashtable of meta-data

Table 4.8: Overview of IMetaInterface Interface

Comparator
EQUALS
NOT_EQUAL
GREATER_THAN
GREATER_THAN_OR_EQUAL
LESS_THAN
LESS_THAN_OR_EQUAL
HIGHEST
LOWEST

Table 4.9: Overview of MetaDataRule Comparators

value is always represented with a String object, however, any Java type can be used as the value (e.g. int, double, String etc.). For instance, this can be done with the following code,

```
rule = new MetaDataRule("TIME_COMPLEXITY", EQUALS, "O(N)");
```

Sets of these rules can subsequently be passed to Juno and into its Configuration Engine to describe non-functional requirements of the component configuration chosen to offer the implementation behind a requested service interface.

Context Repository

To assist in the generation of meta-data for components, Juno also maintains a centralised ContextRepository component. This is simply responsible for maintaining up-to-date information about the operating conditions of the node. It offers a hash table abstraction that allows context information to be inserted and retrieved. Further, it allows the attachment of ContextSource components that

can generate context information and insert it into the repository. Last, listeners can also be attached so that they can be informed if any changes occur to particular items of context information.

The purpose of the Context Repository is therefore to offer a means by which individual components can understand the environment they operate in. For instance, when selecting a delivery protocol it is beneficial to take into account the upload capacity of the node; this can simply be ascertained by querying the Context Repository. Further, runtime information such as the available upload bandwidth can also be acquired. This therefore allows the different components and frameworks to utilise IF-DO rules to shape their behaviour.

Configuration Engine

It has previously been shown how Juno represents capabilities and requirements. It is the responsibility of the Configuration Engine to combine these two elements to make a decision as to which component configuration to use in a particular situation.

The Configuration Engine accepts service requests alongside their requirements then selects from the Configuration Repository the best configuration to achieve those requirements; its interface is shown in Table 4.10. This is a simple process that is accessed using the buildService method. This method accepts a request for a particular service (defined by its interface), alongside a set of MetaDataRule objects. First, the Configuration Engine looks up all Configuration objects that offer this interface in the Configuration Repository, before comparing them against the requirements until a match is found. The appropriate Configurator is then constructed and the chosen Configuration object passed to it. This Configurator is then returned by the Configuration Engine to the requester so the service can be accessed.

If the operating requirements of the service change, it is also possible for the Configuration Engine to be updated using the rebuildService method. This allows the rules associated with a particular service instance to be modified. If this modification results in a different component configuration being considered superior then this is subsequently installed, replacing the previous one. Importantly, through the use of dynamic bindings and indirection, it is not necessary for any object references to be changed by the user(s) of the service.

Distributed Configuration Engine

The previous section has shown how a decision engine can compare meta-data exposed by configurations with selection predicates generated by applications. So far, this has been considered from a local perspective. However, in certain circumstances it is also beneficial to perform (re-)configuration in a distributed setting. For instance, if a server becomes heavily loaded with clients, it would be

Method	Returns	Description
buildService(String intf, int configuratorID, Collection<MetaDataRule> rules)	IConfigurator	This method takes a given service interface, a unique identifier and a set of MetaDataRule objects and returns an IConfigurator containing the instantiated service
rebuildService(int configuratorID, Collection<MetaDataRule>)	boolean	This replaces the MetaDataRule set associated with a particular Configurator and re-runs the selection process
select(Collection <IMetaInterface> components, Collection<MetaDataRule> rules)	IConfigurator	This method takes a set of components and executes a given set of rules over them to select the optimal
requestConfiguration(ConfigurationRequest request)	void	This accepts a request (from another node) to re-configure one of its services

Table 4.10: Overview of IConfigurationEngine Interface

advantageous to re-configure the delivery strategy into a more scalable peer-to-peer one. This, however, requires coordination between all parties. The primary use-case for this is therefore when a provider wishes to modify the behaviour of its consumers; this is because, generally, the provider is in a trusted position and consumers will accept its requests to ensure accessibility to the content.

This process is managed by a generic component called the DistributedConfigurationCoordinator (DCC), which augments the functionality of the local Configuration Engine to allow distributed adaptation to take place. This occurs whenever a desired local re-configuration requires the cooperation of a remote party. The DCC is a component that resides on every Juno node and listens for remote configuration requests from other nodes. Similarly, it can also generate requests to be passed to other nodes. The underlying method by which it propagates this information is not fixed (i.e. it can be re-configured using different component implementations). However, currently it is performed using point-to-point connections; alternative propagation techniques include IP multicast and gossip protocols.

DCCs interact with each other by exchanging ConfigurationRequest and ConfigurationResponse objects. When a ConfigurationRequest object is received, the DCC looks up the Configurator responsible for the service in question; this is done using the Configuration Engine. Once the DCC locates the Configurator, it passes the request into it using the recvConfigurationRequest(ConfigurationRequest

Parameter	Description
uniqueID	A unique identifier for the request; this allows responses to be generated that map uniquely to each request
serviceIntf	The canonical name of the service interface being adapted
contentID	The item of content being handled by the service
requestedConfiguration	The new configuration to utilise
remoteConfiguration	The configuration being used at the remote side (i.e. the requester)
source	The node requesting the re-configuration
newRemoteContent	A new RemoteContent object to represent the new point at which the content can be accessed through

Table 4.11: Overview of Remote ConfigurationRequest Object

req) method. The Configurator is then responsible for deciding whether or not to accept the request. If it is accepted, the Configurator re-configures the services as requested. This decision can be made on any criteria; the most likely being whether or not the re-configuration would invalidate the application's selection predicates. Once the decision has been made, the DCC responds to the requester with a ConfigurationResponse object containing either true or false.

Table 4.11 details the ConfigurationRequest object. To identify the service that it wishes to re-configure, the contentID and serviceIntf fields are included. This allows the DCC to discover which service the request refers to, as well as the particular item of content that it is dealing with. Alongside this, it also contains the requestedConfiguration and remoteConfiguration variables; these are String representations of the the configuration being used at the remote side and the configuration that the request would like to have instantiated. Finally, a new RemoteContent object is included in case it changes. This occurs if a provider is re-configuring its provision technique and needs the consumers to access it in a different manner. For instance, if a provider wishes its consumers to access content using BitTorrent rather than HTTP, the newRemoteContent variable would contain a reference to the new BitTorrent tracker.

The previous paragraphs have detailed how the underlying remote interactions of the DCC take place. However, it is also important to understand how the local interactions occur. Table 4.12 details the IDistributedConfigurationCoordinator interface. This interface can be used by any Configurator object to perform distributed re-configuration. The decision to do this is made within the Configuration object as part of its execute method. Therefore, when a new configuration is instantiated, it can also request that other remote configurations are

Method	Returns	Description
requestConfiguration(Collection <Node> nodes, ConfigurationRequest req)	void	Sends a ConfigurationRequest to a set of nodes
sendResponse(Collection<Node> nodes, ConfigurationResponse res)	boolen	Sends a ConfigurationResponse to a set of nodes

Table 4.12: Overview of IDistributedConfigurationCoordinator

instantiated. This allows a distributed service to be constructed, as initiated by a single node.

4.5 Juno Content-Centric Architecture

The previous section has detailed the core elements and principles of the Juno framework. These underlying principles are exploited to build the content-centric functionality of Juno. This section describes how content-centricity is implemented using these principles.

4.5.1 Overview

Juno's content-centric functionality makes up the bulk of Juno's implementation. It operates alongside the core framework to offer a content-centric and delivery-centric paradigm to applications. As shown in Figure 4.7, the service is built from two primary components: the Discovery Framework and the Delivery Framework; as well as a Content Management system. The Discovery Framework is responsible for locating a set of potential sources for an item of content, whilst the Delivery Framework is responsible for subsequently accessing it. Both frameworks are composites that are built up from a set of one or more underlying component implementations. The frameworks' purpose is therefore to manage these underlying components, which are termed *plug-ins*. This management is driven by requirements passed down to Juno from the application. These requirements stipulate how the application wishes to use the middleware and subsequently results in its adaptation.

Figure 4.7 provides an example of Juno's operation. As can be seen, the Content-Centric Framework itself is quite thin, acting primarily as a point of redirection and management for the Discovery and Delivery Frameworks. When Juno receives a content request from the application it first queries the Content Manager to ascertain whether the content is locally available. If it is not, the Content-Centric Framework utilises the Discovery Framework to locate potential sources of the content. The Discovery Framework is built from a number of *discovery plug-ins*. Each plug-in is capable of querying a particular provider's

discovery system to locate available sources of the desired content. In the diagram, the plug-in attached to the Discovery Framework is called 'Content Index'; in practice, however, multiple plug-ins are likely to be attached so to operate in parallel. When multiple plug-ins are operating, the Discovery Framework aggregates all the information discovered and returns it to the Content-Centric Framework.

By the end of this process, the Content-Centric Framework is in possession of a number of sources located through a range of discovery systems that (potentially) offer the content in a variety of ways. The next step is therefore to pass these sources into the Delivery Framework alongside the delivery requirements provided by the application. It is then the responsibility of the Delivery Framework to exploit these potential sources to provide the content in the optimal manner. This is achieved through transparent client-side re-configuration. In essence, this involves adapting the underlying implementation of the Delivery Framework (by attaching plug-ins) to interoperate with the source(s) that best match the delivery requirements. In Figure 4.7, a number of possible methods of delivery exist for content item 'A' including a BitTorrent swarm, a HTTP server and a ZigZag streaming tree. However, due to some requirement (e.g. throughput), the Delivery Framework selects the use of BitTorrent; this plug-in is therefore retrieved by the Configuration Engine and attached to the Delivery Framework. Following this, Juno begins the content download, passing the data to the application via its preferred abstraction. This flexible use of different abstractions allows the application to interact with the content in a way that best suits its requirements; for instance, this can be using a file references, a real-time input stream or a random access lookup. This process can be summarised in the following steps,

1. Receive request formatted as unique content identifier, alongside delivery requirements
2. Check wether or not the Content Manager has a local copy (if so, return it directly to the application)
3. Pass request to Discovery Framework
    - Pass request to each of available discovery plug-ins
    - Receive list of available content sources from one or more plug-ins
    - Aggregate results and return to Content-Centric Framework
4. Pass request and list of sources to the Delivery Framework, alongside original delivery requirements
    - Compare requirements against the meta-data associated with each of the available sources

- Attach the delivery plug-in that can best access the content in a manner that fulfils the requirements
- Pass request and source information to chosen plug-in so that it can begin the download
- Return a content handle to the Content-Centric Framework

5. Return a content handle to the application

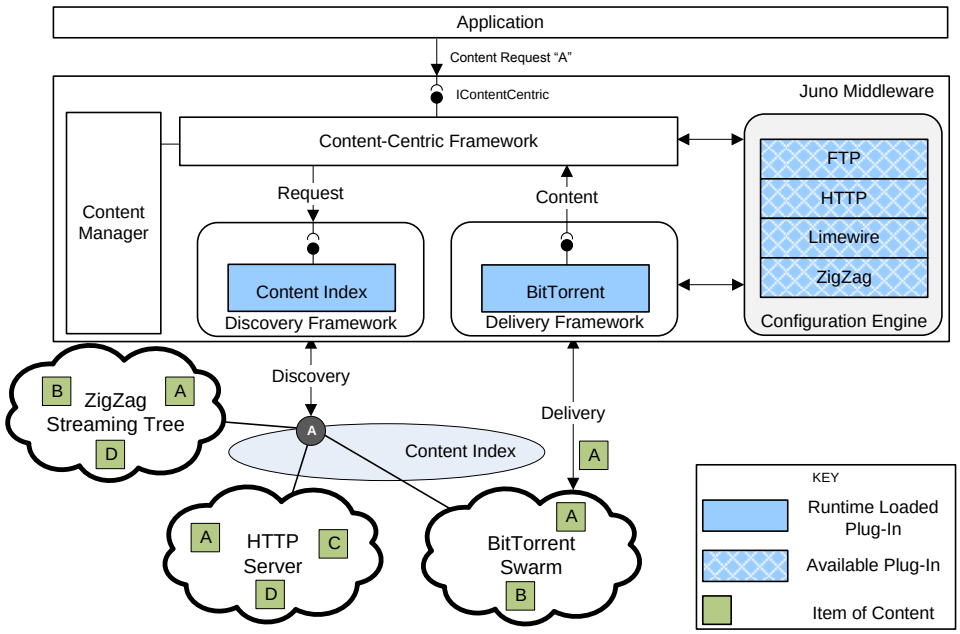


Figure 4.7: Example of Juno Architecture and Operations

This section details the operation of the Juno Content-Centric service. First, it describes the content-centric and delivery-centric abstraction provided to the application. Following this, the Content Management is described, including how content is uniquely identified. The Discovery Framework is then outlined showing how content is discovered in various systems. Last, the Delivery Framework is described, showing the (re-)configuration process for interoperating with different third party content distribution systems.

4.5.2 Content-Centric Framework

The Content-Centric Framework consists of the necessary functionality to (*i*) query the Content Manager, (*ii*) query the Discovery Framework, and (*iii*) pass the discovery information to the Delivery Framework. As such, it is a thin component that simply exposes a content-centric (and delivery-centric) abstraction to

Method	Returns	Description
getContent(String magnetLink, int accessMechanism, Collection<MetaDataRule> rules)	Content	Requests an item of content with a set of requirements structured as rules
updateRuntimeInformation (String magnetLink, Collection<MetaDataRule> rules)	boolean	Changes the requirements of a previously issued request

Table 4.13: Overview of IContentCentric Interface

the application. Using this abstraction, it receives requests from the application before passing them onto the necessary frameworks.

Table 4.13 provides an overview of the IContentCentric interface, which is both content-centric and delivery-centric. The most important method is getContent. It first accepts a content identifier (termed magnetLink), which uniquely identifies the content. It then accepts the preferred access mechanism; currently this can either be stored or streamed (this is extensible though). This dictates through which abstraction the application wishes to view the content (obviously, this also affects the possible underlying delivery protocols that can be used). Last, the application must also provide its delivery requirements as a collection of MetaDataRule objects. A Content object is then returned from this method, which represents the content that is accessed. A number of different Content objects exist for different types of content; these are used to offer access for the application. Alongside this, an update method is also available to modify any of the requirements during the delivery.

The IContentCentric abstraction is extremely simple to use for applications. It is worth noting that unlike previous content-centric abstractions (e.g. [54]), there are no methods to support publishing content. This is because IContentCentric is solely a consumer-side abstraction, which leaves content publication to the discovery systems that Juno interoperates with. A user wishing to publish content must use a different interface: IProvider. A further important observation is that due to Juno's configurable service-oriented nature, it is possible to dynamically replace the IContentCentric component without the modification of application-level code.

4.5.3 Content Management

Overview

Juno abstracts the content management away from any individual delivery plug-in, thus allowing them to share a common content library. The Content Manager is an individual component within Juno that provides this service; it offers methods to lookup, store and manipulate local content. All delivery schemes utilise

the Content Manager, negating the need to transfer content between delivery plug-ins if they are interchanged. Further, this provides a mechanism by which multiple plug-ins can be coordinated without over-writing each others' data. In order to allow convenient usage, the Content Manager also offers multiple interfaces to enable both chunk-based and range-based access to content. Therefore, different schemes can view data in different ways, making implementation and development easier.

Due to the content-centricity of Juno, the Content Manager is considered as a primary entity in its operation. As such, when a request is received by the Content-Centric Framework, the Content Manager is first queried to validate whether or not a local copy exists. If content is found to be locally available, a handle is returned to the application. Depending on the requirements of the application, different content handles can be returned (e.g. file reference, input stream etc.).

This section details the content management aspects of Juno's design. First, the content addressing scheme is described to show how content can be uniquely identified. Following this, the method through which content is represented within Juno is shown, detailing the standard objects used.

Content Addressing

A significant research challenge for overlaying content-centricity onto many existing (non content-centric) discovery systems is that of content addressing. A content-centric system, by its definition, detaches addressing from location and allows the acquisition of content solely using its own unique address. However, the majority of existing delivery systems do not offer this service and therefore this functionality must exist at the middleware layer. Critically, this must be achieved without modifying the behaviour of existing discovery systems and therefore this must be done in a backwards compatible way. Two possible options exist; either, Juno must perform a mapping between global Juno content identifiers and the various location identifiers or, alternatively, an existing deployed content-centric addressing solution must be utilised.

The first option involves generating a mapping between a global location-independent content identifier and a traditional location-dependent content identifier (e.g. Content-A → http://148.88.125.32/ContentA). This is similar to indirection systems such as the Domain Name System and i3 [130]. Unfortunately, this has serious deployment difficulties as it would involve an infeasibly large crawling to generate the mappings for all content. Such a task could perhaps be achieved by massively resourced organisations such as Google, but this would not be feasible for the vast majority. Further, this would have to be repeated periodically to discover new content. An obvious alternative would be to insist providers actively register their content with the system so that the mapping can

Hash Identifier	Supported Protocols
SHA-1	Gnutella, Gnutella2
BitPrint	Gnutella, Gnutella2
ED2K	eD2K
Kazaa	Kazaa
MD5	Gnutella2
BITH	BitTorrent (e.g. Azureus, Mininova, PirateBay etc.)
Tiger Tree Hash	Direct Connect, Gnutella2

Table 4.14: Overview of Discovery Protocols that Support Magnet Links and their Hashing Algorithms

be performed progressively. However, this would significantly reduce the number of potential discoverable sources.

The second option is to use an existing deployed content-centric addressing scheme. Currently, a small number of possible content-centric addressing schemes have been proposed, including DONA [93] and AGN [83]. These, however, have not been successfully deployed as of yet. Similarly, they would not be compatible with existing discovery and delivery systems, subsequently limiting interoperability. There is, in fact, no current globally deployed addressing scheme for content; however, a common property of most current schemes is the use of hash-based content identification. This sees the content's data being passed through a hashing algorithm (e.g. MD5) to generate a unique identifier. This is, for example, used within both DONA and AGN, as well as a number of other deployed content addressing schemes.

Juno therefore follows this example and exploits hashing algorithms to uniquely identify content. A major limitation of this, however, is that different plug-ins generally use different hashing algorithms. Table 4.14 provides an overview of the hashing algorithms used by a range of popular systems. Clearly, interoperation with these different systems would only be possible if Juno were capable of identifying content using all hashing algorithms. Therefore, to address this, Juno does not pre-define a single hashing algorithm for its content identification; instead, the Content Manager utilises a dynamically extensible set of algorithms. To ensure high degrees of interoperability, currently the algorithms detailed in Table 4.14 are all utilised.

Following this, the next challenge is how these different hash-based identifiers can be managed to enable interoperation with the different systems. To handle this concern, Juno stores identifiers as Magnet Links [19]. These are text-based Uniform Resource Identifiers (URIs) that allow multiple hash-based identifiers to be stored within a single content reference. The key benefit of using Magnet Links to 'wrap' the content identifiers is the level of their current uptake, which is far

beyond alternatives such as Extensible Resource Identifiers (XRIs) or Meta Links. Figure 4.8 highlights this by showing the distribution of peer-to-peer protocol usage, as observed in [128]. It can be observed that a massive proportion of peer-to-peer traffic is BitTorrent (67%). Similarly a further 32% is dedicated to eD2K and Gnutella. Importantly, all three protocols support content identification through Magnet Links. Subsequently, the ubiquity of this addressing scheme allows 99% of peer-to-peer traffic to be interoperated with. In fact, only 0.99% of peer-to-peer traffic can be attributed to other protocols; the break-down of this, however, is unavailable which means that it is likely to also consist of protocols such as Direct Connect and Kazaa, which would subsequently allow Juno's reach to extend even beyond this 99% figure.

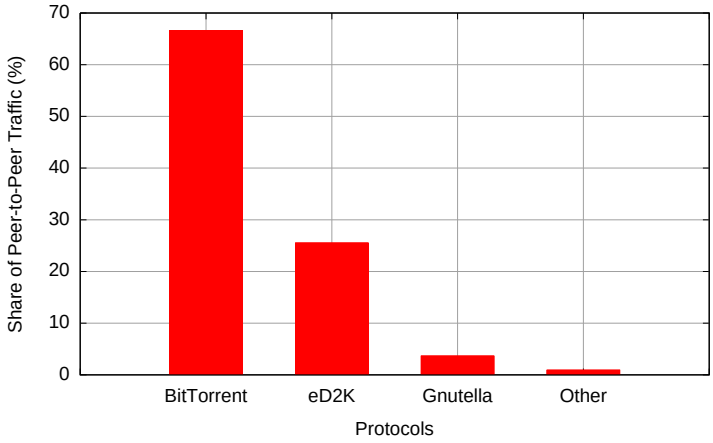


Figure 4.8: The Distribution of Peer-to-Peer Traffic in the Internet [128]

This observation means that a single Magnet Link can be passed into multiple plug-ins to allow seamless interoperation. This simplifies the task for both applications and Juno, as well as the plug-ins. Importantly, however, it is also possible for Magnet Links to be replaced in the future if an alternative (hash-based) scheme becomes more popular. This is because the Magnet Link simply acts as a wrapper for the identifiers. Table 4.15 gives a description of the primary fields in a Magnet Link, which have the format:

magnet:?xt=urn:sha1:FBAAJ3ILW4TRNJIV4WNAE51SJBUK8Q1G.

A Magnet Link is a text-based link similar to a traditional URL. It always starts with the prefix 'magnet:?'. Following this is a set of parameters that define certain properties of the content that it identifies. Parameters are defined as attribute:value pairs with the structure attribute=value. The most common parameter is `xt`, which provides a unique hash identifier of the content based on a given algorithm (e.g. SHA1, MD5 etc.); this is therefore a mandatory field for all

Parameter	Description
xt	*Exact Topic:* This indicates that the link contains the details of a specific item of content. Other parameters can also be used (c.f. Table 4.16)
urn:sha1	*Hash Algorithm:* This indicates the hashing algorithm used by this identifier
FBAAJ3...	*Hash Value:* This is the output of the hashing algorithm when applied to the data

Table 4.15: Overview of Primary Fields in Magnet Links

Parameter	Description
xt	*Exact Topic:* An identifier of the content (based on a given hashing algorithm)
dn	*Display Name:* The human readable title of the content
xl	*Exact Length:* The length of the content in bytes
as	*Acceptable Source:* A location-oriented URL address (e.g. a HTTP link)
xs	*Exact Source:* A host:port reference to a source
kt	*Keyword Topic:* Keywords associated with the content
mt	*Manifest:* A link to a list of further Magnet Links (e.g. a list of Magnet Links for each song on an album)
tr	*Address Tracker:* A location-oriented link to a BitTorrent-like tracker service

Table 4.16: Overview of Parameters in Magnet Links

systems utilising unique addressing. Magnet Links, however, are highly flexible with the ability to stipulate a number of other parameters relating to the content; full details of these are provided in Table 4.16.

Juno and its Content Manager identify content using String objects that contain a Magnet Link. To make the process simpler and to allow future use of alternative hash-based identifiers, it is also possible to identify content simply using one of the identifiers (e.g. "sha1:FBAAJ3..."). At least one hash identifier must be present, however, so that the content can (*i*) be uniquely identified and (*ii*) can be validated through re-hashing it (this is necessary for fulfilling the requirements of a content-centric network, as detailed in Section 2.2).

Content Representation

To ease the interaction between different components, there are a number of standardised mechanisms by which content is represented and accessed in Juno. These are encapsulated in various `Content` objects, which are shown in Figure

4.9. This section details each of these objects.

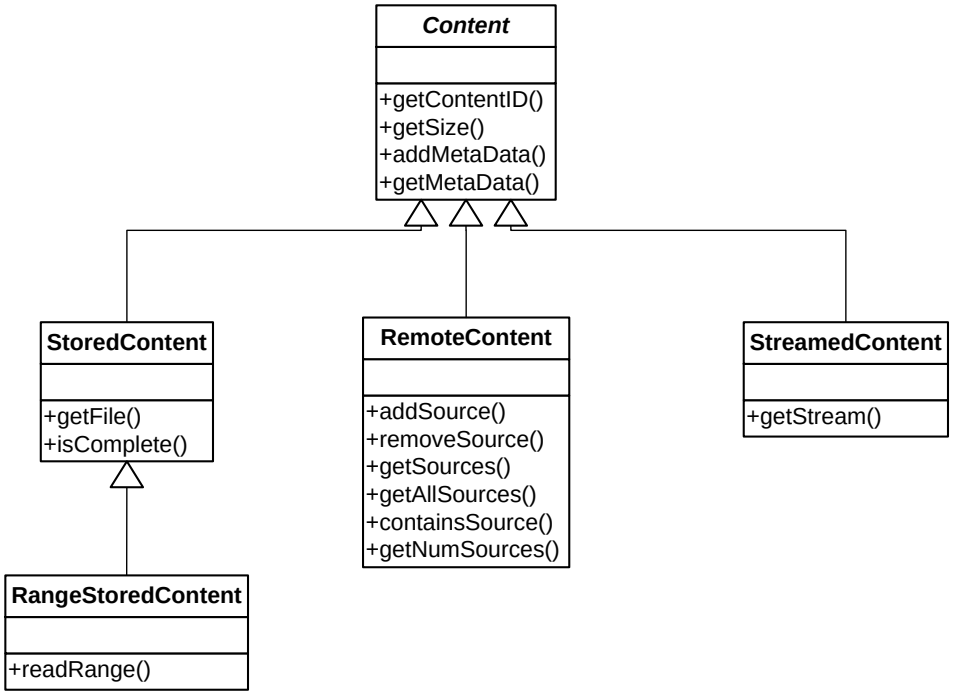


Figure 4.9: A Class Diagram of the Content Representation Classes

Content. The Content object is a base representation of a content item in Juno. It does not contain any data or references to data; instead, it simply represents it. It is abstract and cannot be instantiated. Its details are provided in Table 4.17.

StoredContent. The StoredContent object extends the Content object to represent it when stored locally. The object also provides access control to Juno so that it can be immediately returned to the application without allowing it to access the content. It is generally used when an application requires the content to be stored on disk and accessed afterwards, e.g. for file sharing or a software update. Its details are provided in Table 4.18.

RangeStoredContent. The RangeStoredContent object extends the StoredContent object to provide easy random access to locally stored content. The object also provides access control to Juno so that it can be immediately returned to the application without allowing it to access the content. It is generally used when applications need to skip through the data. Its details are provided in Table 4.19.

Method	Returns	Description
getContentID(HashFunction function)	String	Provides the particular hash identifier of the content
getSize()	long	Provides the size of the content
addMetaData(String attribute, Object value)	void	Adds an item of meta-data to the content
getMetaData(String attribute)	Object	Retrieves an item of meta-data associated with the content

Table 4.17: Overview of Content Object

Method	Returns	Description
getFile()	File	Provides the java.io.File object containing the local path to the content
isComplete()	boolean	Returns whether the file is fully available locally

Table 4.18: Overview of StoredContent Object (N.B. Extends Content object)

Method	Returns	Description
readRange(long start, long end, byte data)	long	Loads the data from the range into the array. Returns number of bytes read

Table 4.19: Overview of RangeStoredContent Object (N.B. Extends StoredContent object)

StreamedContent. The StreamedContent object extends the Content object to offer the ability to access an item of content in a streamed manner. This could be a video stream or, alternatively, simply a data stream. For instance, if an application requests a picture for immediate display from Juno, it is easier to return it as a StreamedContent object than write it to disk first and return it as a StoredContent object. Its details are provided in Table 4.20.

RemoteContent. The RemoteContent object represents content residing at a remote location. It subsequently provides all the necessary information required to access it, i.e. a list of sources. Individual sources are represented using the DeliveryNode object, which contains the host's address as well as its supported protocol(s) and any other information required to access the content (e.g. the remote path). Table 4.21 provides an overview of the methods offered by RemoteContent.

Method	Returns	Description
getStream()	InputStream	Provides a java.io.InputStream object to access the content stream

Table 4.20: Overview of StreamedContent Object (N.B. Extends Content object)

Method	Returns	Description
addSource(DeliveryNode newSource)	void	Adds a source of the content
removeSource(DeliveryNode removeSource)	boolean	Removes a source of the content
getSources(String protocol)	Set	Returns a set of sources that support a given protocol (as DeliveryNode objects)
getAllSources()	Set	Returns all sources of the content (as DeliveryNode objects)
containsSource(DeliveryNode node)	boolean	Returns whether the content has a particular source
getNumSources()	int	Returns the number of sources of the content
getNumSources(String protocol)	int	Returns the number of sources supporting a given protocol

Table 4.21: Overview of RemoteContent Object (N.B. Extends Content object)

4.5.4 Content Discovery Framework

Overview

The Discovery Framework is responsible for locating available sources of the desired content. It is important to note that it is *not* responsible for discovering which item of content an application or user may desire (e.g. through keyword searching). Instead, it assumes the application is already aware of exactly which item of content it wishes to download, uniquely identified through one or more hash identifiers.

There are currently a number of providers that operate in the Internet utilising a range of discovery mechanisms. A discovery mechanism can be defined as any protocol that is used to locate sources of a given item of content. Once an application has decided to access a particular item of content, a provider's discovery mechanism is used to ascertain whether or not it can serve it.

Generally, each discovery mechanism will provide indexing functionality for a small number of providers and delivery systems. For instance, Google provides indexing for a number of websites (HTTP and FTP); Mininova.org provides indexing for BitTorrent swarms; and Gnutella provides indexing for Limewire and

other Gnutella clients. In many situations, a desired item of content is indexed in multiple discovery systems (and available through multiple providers). For instance, sources of the latest Ubuntu ISO are indexed using Google, Mininova.org, Gnutella and eMule, to name a few. Therefore, to locate complete sets of sources it is necessary to interact with multiple discovery systems.

To allow access to content in all these different systems, the Discovery Framework utilises multiple configurable *discovery plug-ins* that each provide access to a given discovery system. The Discovery Framework therefore attaches a set of plug-ins, which it queries whenever an application requests content from Juno. Each plug-in returns its result containing any possible sources of the requested content indexed within that particular discovery system. Requests are formatted as Magnet Links, as described in Section 4.5.3, which allows easy interoperation with plug-ins due to the widespread support for this addressing scheme. Any plug-ins that utilise alternative hash-based addressing schemes can also be used by simply converting the Magnet Link into their desired format (e.g. Meta Link, XRI).

Plug-ins are both dynamically pluggable and extensible, offering an effective means by which applications can gain access to many discovery systems without dealing with the complexity themselves. This also allows new discovery mechanisms to be added at a later date, therefore providing support for the introduction of new protocols (potentially including network-level content-centric lookups) without modification to applications. The Discovery Framework thus provides the foundations for content-centricity in the middleware layer as, through this mechanism, the application is agnostic to content location or the use of any particular discovery system. Instead, the applications views the content-centric network as simply the Juno abstraction.

This section covers the design of the Discovery Framework, alongside the discovery plug-ins. First, the Discovery Framework abstraction is detailed. Following this, the design of the plug-ins is provided. Last, the plug-in management is described; this outlines the core of the Discovery Framework and shows how it selects which plug-ins to query and how they are then interacted with.

Content Discovery Abstraction

The Discovery Framework is directly interacted with only by the Content-Centric Framework. When the Content-Centric Framework receives a request from an application and the content is not locally available, it first passes it to the Discovery Framework. The purpose of the Discovery Framework is then to discover any possible sources of the content. If the Discovery Framework is successful in this task, it returns a list of the sources to the Content-Centric Framework, which then utilises the Delivery Framework to access the content.

Table 4.22 details the IDiscoveryFramework interface; this is used by the

Method	Returns	Description
locateSources(String magnetLink)	RemoteContent	Locates sources for a given content identifier
setTimeout(int timeout)	void	Sets the timeout period for queries
updateSources(RemoteContent remoteContent)	boolean	Allows new source information to be injected by external parties (e.g. the application)

Table 4.22: Overview of IDiscoveryFramework Interface

Content-Centric Framework to interact with the Discovery Framework. This abstraction is very simple and offers access to the necessary functionality to locate content. The primary method used is lookup(String magnetLink), which returns a RemoteContent object. The RemoteContent object is a representative object for an item of content that is remotely accessible (c.f. Section 4.5.3). It can contain one or more sources that the content is accessible through; this can be, for instance, a HTTP URL or, alternatively, the address and port of a peer operating in Gnutella. Therefore, the RemoteContent object returned to the Content-Centric Framework is always an aggregation of all results obtained from the different discovery plug-ins.

There is also a setTimeout(int timeout) method, which allows the Content-Centric Framework to stipulate a maximum length of time the Discovery Framework can spend searching for content. This method dictates that the results must be returned within this period regardless of whether all the plug-ins have returned their results. Setting this value to 0 results in a timeout of ∞.

Discovery Plug-ins

The Discovery Framework maintains an extensible set of discovery plug-ins, as indexed by the Configuration Repository. A plug-in contains the functionality for querying a particular discovery system to ascertain the location (host:port or URL) of any sources of the content indexed by that system. Plug-ins are implemented as self-contained Juno configurations; each plug-in is therefore represented by a Configuration object (c.f. Section 4.4.3) that contains one or more components. As described earlier, a Configuration object is an object that contains the necessary functionality to interconnect one or more components to build a service defined by a given service interface.

The service interface exposed by a discovery plug-in is show in Table 4.23. This interface is very simple, offering only a single method to allow the plug-in to be queried. It receives a Magnet Link and a timeout; after this period the plug-in must return a RemoteContent object containing any sources it has located,

Method	Returns	Description
lookup(String magnetLink, int timeout)	RemoteContent	Locates sources for a given content identifier

Table 4.23: Overview of IDiscovery Plug-in Interface

alongside their supported protocols. Plug-ins solely interact with the Discovery Framework. A plug-in must implement all functionality required to query a particular discovery system in a self-contained way. For instance, a Gnutella plug-in must contain the functionality to connect and interact with the Gnutella overlay; to issue queries to the overlay; and to interpret any results received. As with any Juno component, it is possible for the plug-ins to be configured with any parameters that they might require, using the approach outlined in Section 4.4.2. This allows, for instance, a Gnutella plug-in to be configured to not actively cooperate in the routing of requests to reduce overhead on low capacity devices. An overview of the interconnections between the Discovery Framework and the plug-ins is shown in Figure 4.10; here it can be seen that three plug-ins are connected to the framework (although this can be extended).

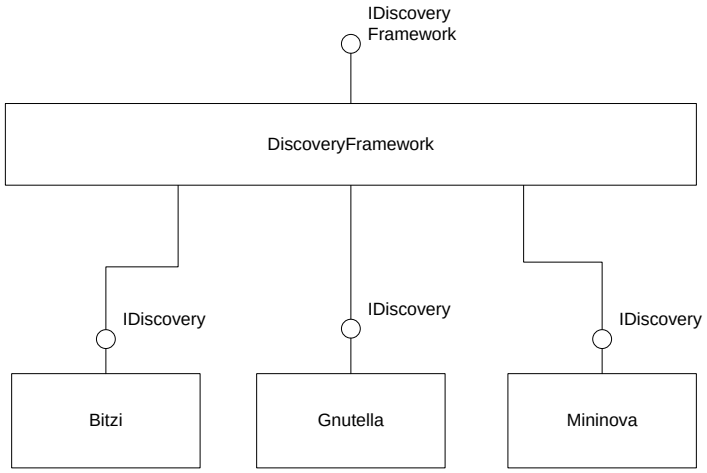


Figure 4.10: Overview of Discovery Framework and Discovery Plug-ins

Discovery Plug-in Management

In essence, the Discovery Framework is a management unit for handling the plug-ins. It involves two steps, (*i*) selecting which plug-ins to query (and in what order), and (*ii*) querying them and aggregating their results. This section now details these two processes.

Method	Returns	Description
getPluginQueryOrder(String magnetLink)	Execution-Order	Returns an ExecutionOrder containing String representations of the plug-ins to query alongside their order and parallelism.
setPluginQueryOrder(String magnetLink, ExecutionOrder)	void	Sets the ExecutionOrder for a particular Magnet Link

Table 4.24: Overview of DiscoveryPolicyMaker Interface

Selecting Discovery Plug-ins. When a request is received by the Discovery Framework it is necessary to first select which plug-in(s) will be queried. This can either be managed automatically by the Discovery Framework or, alternatively, explicitly by the application.

When managed by the Discovery Framework, this is handled by a further (pluggable) component, which is set by the application: the DiscoveryPolicyMaker. This component, shown in Table 4.24, accepts a Magnet Link and returns the order in which the plug-ins should be queried. The DiscoveryPolicyMaker returns an ExecutionOrder object, which represents a sequence of events with each event being associated with a plug-in and a timestamp relative to the request-time. The ExecutionOrder always also stipulates an 'END' point at which the execution ceases; this constitutes the timeout at which the Discovery Framework stops waiting for results.

The purpose of this is to allow applications to influence the discovery process based on their own knowledge. The extent of this knowledge will vary based on how involved the application is with the provision of content. For instance, if the application developers were also responsible for publishing the content, it is likely that they could introduce a DiscoveryPolicyMaker component that could exploit this awareness (e.g. to provide the best locations to search). By supporting this ability, the discovery process can therefore be improved dramatically.

The DiscoveryPolicyMaker component is pluggable and can be re-configured for use with different applications. In the most extreme of cases, it can even be re-configured to use different implementations for each request. Currently, the default DiscoveryPolicyMaker component is implemented using a file that can be used to stipulate a default ExecutionOrder alongside any content-specific orders required. This, however, can be extended to utilise more sophisticated features such as performing remote lookups to decide the best ExecutionOrder. Importantly, the DiscoveryPolicyMaker can, like any other component, be dynamically replaced therefore allowing modifications to be easily made during runtime.

Using Discovery Plug-ins. Once an order has been selected, the Discovery

Framework queries each of the necessary plug-ins using the relative timestamps provided by the DiscoveryPolicyMaker component. The Discovery Framework subsequently collects each of the results returned by the plug-ins. Results are returned using the RemoteContent object, shown in Table 4.21. The RemoteContent object represents any item of content residing at a remote location. This can be a single location or, alternatively, many locations using different protocols.

Due to the diversity of delivery plug-ins, it is usual for the results to arrive progressively. This is because different plug-ins require different periods to obtain their results (e.g. a Gnutella plug-in takes longer than a Pastry one). As each RemoteContent object is received, the data is extracted and inserted into a new, shared RemoteContent object. After the timeout has been reached, the Discovery Framework returns the aggregated RemoteContent object.

4.5.5 Content Delivery Framework

Overview

The Delivery Framework is responsible for accessing content once possible sources have been located by the Discovery Framework. Its duty is to provide the desired content to the application in a way that best suits its requirements. This is achieved through dynamic (re-)configuration of the framework to interoperate with the most suitable provider(s). Generally, an item of content will be available through a multitude of sources operating with different capabilities and protocols. Thus, many different requirements can be satisfied (or unsatisfied) based on the selection of which source(s) and protocol(s) to use.

This process is realised through the use of *delivery plug-ins*. These are pluggable components that offer the functionality to access content from a particular delivery system. They are, however, managed in a different way to the plug-ins used by the Discovery Framework. When the Content-Centric Framework receives a request from the application it first uses the Discovery Framework to locate a set of accessible sources for the data. This information is then passed to the Delivery Framework, which is responsible for accessing the content on behalf of the application. To achieve this, the Delivery Framework parses any requirements for the delivery issued by the application; these can range from relatively static preferences such as the level of security required, to far more dynamic aspects such as performance. Using these requirements, the Delivery Framework selects the source(s) and protocol(s) that offer the content in the most appropriate manner. Once this decision has been made, the Delivery Framework attaches the necessary plug-in to interoperate with the optimal source(s) and then begins the download. Following this, the content can be returned to the application in an abstracted way using the preferred means (using one of the content representation objects, c.f. Section 4.5.3).

Method	Returns	Description
startDownload(RemoteContent content, String intf, Collection <MetaDataRule> rules)	Content	Starts a download for a particular content item using a particular service interface (e.g. stored or streamed) with a set of meta rules
startDownload(RemoteContent content, String intf, Collection <MetaDataRule> rules, File file)	Content	Allows a file location for the download data to be specified
updateRuntimeInformation(RemoteContent content, Collection <MetaDataRule> rules)	boolean	Allows a new set of meta-data requirements to be introduced

Table 4.25: Overview of IDeliveryFramework Interface

The plug-ins are dynamically attachable and extensible, allowing new plug-ins to be acquired if an item of content is published using a currently incompatible protocol. The use of delivery plug-ins therefore offers a powerful mechanism by which applications can request content without having to manage any of the lower level concerns. This section details the design of the Delivery Framework. First, the abstraction used to interact with the framework is detailed. Following this, the delivery plug-ins are described. Finally, the management functionality of the framework is outlined, showing how plug-ins are selected and attached.

Content Delivery Abstraction

The Delivery Framework is solely accessed by the Content-Centric Framework; its interface is detailed in Table 4.25. Once a set of possible sources have been located by the Discovery Framework, they are passed into the Delivery Framework alongside the application's requirements. Requirements are structured as sets of MetaDataRule objects (c.f. Section 4.4.4), which stipulate the preferred values of particular items of meta-data exported by each delivery plug-in. The framework abstraction provides the ability to start a download as well as to store the data in a stipulated file. Importantly, it also offers the ability for the application to update its delivery requirements at any given time using an updateRuntimeInformation method. This allows a new set of MetaDataRule objects to be passed to the framework, thereby driving re-configuration.

Method	Returns	Description
startDownload(RemoteContent content, String file)	StoredContent	Starts downloading a remote item of content
stopDownload(String contentID)	void	Stops a download and deletes all temporary data
pauseDownload(String contentID)	void	Temporarily ceases the download

Table 4.26: Overview of IStoredDelivery Plug-in Interface

Delivery Plug-ins

The Delivery Framework maintains an extensible set of delivery plug-ins. A plug-in is responsible for accessing an item of content on behalf of the framework. Plug-ins are implemented as Juno configurations; each plug-in is therefore represented by a Configuration object (c.f. Section 4.4.3). The service interface exposed by the plug-in depends on its underlying method of delivery. Currently, Juno has defined service interfaces for both downloading and streaming content. An individual plug-in can implement multiple service interfaces; for instance, the HTTP plug-in supports both streamed and stored delivery and therefore offers both interfaces.

A delivery plug-in is not restricted in how it accesses an item of content. Currently, most plug-ins implement an individual protocol (e.g. HTTP) and therefore offer the functionality required to access content in that individual system. However, it is also possible for plug-ins to consist of multiple existing plug-ins operating in a composite manner. By handling multiple delivery systems in co-operation like this, it becomes possible to manage their individual complexities in a more controlled manner. Further, the implementation of functionality in components allows easy reuse.

The stored delivery plug-in interface is defined in Table 4.26. When the startDownload method is invoked, the content is downloaded and the data written to file. The chosen file can be explicitly stipulated or, alternatively, the Content Manager can automatically select the location of it. This method subsequently returns a StoredContent object representing the local instance of the content so that the application can gain a handle on the data. This object also allows access control by preventing the application from retrieving a handle on the data until Juno has authorised it.

The streamed delivery plug-in interface is defined in Table 4.27. It offers the functionality to provide the application with content using a streamed abstraction. Ordinarily, a streamed delivery plug-in contains the functionality to access content in a delivery system that support streaming. However, the nature of

Method	Returns	Description
startStream(RemoteContent content, String file)	StreamedContent	Starts streaming a remote item of content
stopStream(String contentID)	void	Stops a stream and removes temporary data from memory
pauseStream(String contentID)	void	Temporarily ceases the stream

Table 4.27: Overview of IStreamedDelivery Plug-in Interface

the abstraction even allows stored delivery systems to be used by downloading the entire file and then beginning the streaming. This limitation is obviously manifested in the plug-in's meta-data (i.e. a very high start-up delay).

Delivery Plug-in Management

In essence, the Delivery Framework is a management unit for handling the use of multiple plug-ins. This involves two steps, (*i*) selecting the optimal plug-in, and (*ii*) utilising it. Alongside this, it can also be necessary to perform reconfiguration, which involves repeating these steps.

Selecting Delivery Plug-ins. When a request is received by the Delivery Framework it is necessary to select which plug-in should be utilised to access the content. At this point in time, the Delivery Framework possesses a RemoteContent object that provides it with,

- Content Identifier(s): The Magnet Link containing the identifier(s) of the content
- Potential Source(s): A list of sources, including their supported protocol(s) embodied in a RemoteContent object

The first step taken is to extract the protocols that each available source supports. Protocols are identified using the format: protocol:variation. The protocol identifier refers to the particular base protocol that the source utilises (e.g. BitTorrent, HTTP etc.). The variation parameter then stipulates a possible provider-specific variation; this allows a provider to release a specialised plug-in that can exploit some provider-specific functionality (e.g. using network coordinates for locality awareness).

Equivalents of these protocol tags are used by the delivery plug-ins to advertise which protocols (and variations) they support. This information is advertised through the IMetaInterface of each plug-in (c.f. Table 4.8). This allows Juno to match up sources with the plug-ins capable of exploiting them. For each protocol

Provider	Protocol	Indexed by
Limewire	HTTP:Limewire	Gnutella
Rapidshare	HTTP	RapidShareIndex
BitTorrent Swarm	BitTorrent:Tracker	Mininova

Table 4.28: Example Set of Available Plug-ins

found to offer the content, the Delivery Framework invokes a buildService method call on the ConfigurationEngine, passing in the need for a delivery plug-in that supports the identified protocol. This is easily achieved with the following code,

```
Vector rules = new Vector()
rules.add(new MetaDataRule("PROTOCOL", MetaDataRule.EQUALS, "HTTP");
IStoredDelivery plugin =
    (IStoredDelivery) ConfigurationEngine.buildService ("IStoredDelivery", 0, rules);
```

After this process has completed, the Delivery Framework holds a collection of plug-ins; one for each of the sources located. Table 4.28 provides an example set of plug-ins and their protocols. Note that there are two variations of HTTP; Limewire actually utilises HTTP for point-to-point connections with each Limewire peer whilst Rapdishare uses HTTP in the traditional sense.

The next stage is to allow each plug-in to generate and expose its meta-data. Some of this information is static and can therefore easily be provided. However, certain items of meta-data must be dynamically generated, e.g. the predicted download throughput. If the provider has deployed a protocol variation plug-in then it may be capable of interacting with the provider's infrastructure to ascertain this information (e.g. through a remote query interface). However, if the provider is represented through a standard plug-in implementation, then mechanisms must be employed that can generate this information without the explicit cooperation of the provider. This is required to maintain backwards compatibility with traditional delivery schemes.

Any runtime meta-data must be dynamically generated by each plug-in. Each plug-in will generally utilise a different mechanism by which its own meta-data is generated. This functionality is usually embodied in a separate pluggable component that performs the generation on behalf of the plug-in. As such, this allows different prediction mechanisms to be used based on the requirements, environment and resources of each node. Subsequently, any plug-in can use any means it is capable of including active probing, simulation, remote service invocation and modelling. These prediction mechanisms are explored in detail in Chapter 5. To assist in this, RemoteContent objects can also contain meta-data that might be used by the process. Generally, the source information is only required, however,

this ability can also be used to provide extra information such as source location and content properties. The following code allows a plug-in to generate its meta-data,

```
Hashtable info = remoteContent.getMetaData();
plugin.generateRuntimeMetaData("IStoredDelivery", info);
```

Once all the delivery plug-ins have generated their meta-data, the selection process is simple. The Discovery Framework passes all the plug-ins to the Configuration Engine alongside the original meta-data rules provided by the application. This is achieved by the following code,

```
Collection<MetaDataRule> applicationRules = rules from application;
Vector plugins = collection of all plug-ins offering the content;
IStoredDelivery selectedPlugin =
    ConfigurationEngine.select(plugins, applicationRules);
```

The selectedPlugin variable will subsequently reference the plug-in that has meta-data that best matches the rules provided by the application. The selection process is therefore abstracted away from the Delivery Framework and encapsulated in the Configuration Engine. As with any other component, the Configuration Engine can easily be replaced with different implementations. Details of the current implementation is provided in Section 4.4.4.

Using Delivery Plug-ins. Now that the optimal plug-in has been selected, it can be interacted with through one of its service interfaces (e.g. IStoredDelivery). The other instantiated plug-ins are also left active until the delivery has completed; this is because if the currently chosen plug-in later becomes suboptimal, it might become necessary to replace it. As such, the other components must remain in memory so that they can continue to offer runtime meta-data. The chosen plug-in can easily be interacted with using the following code,

```
RemoteContent content = contentToDownload;
String file = locationToStore;
selectedPlugin.startDownload(content, file);
```

The plug-in will then begin to download the content using the compatible source(s) and storing the data in the provided file using the Content Manager. If the download is successful, there is no further function for the Delivery Framework to perform because it has already returned a reference to the content (e.g. a StoredContent object, c.f. Section 4.5.3) when the Content-Centric interface first

invoked the download operation. However, during the download it is possible for either the meta-data of the plug-in to change or, alternatively, the selection predicates issued by the application to change. This will occur if there are runtime variations or if the application decides to change its requirements by issuing new MetaDataRule objects. When this situation occurs, it is necessary to perform *re-configuration*.

Re-configuring Delivery Plug-ins. Delivery re-configuration takes place when the currently utilised plug-in becomes suboptimal for some reason. This can be because of a runtime variation (e.g. network congestion) or, alternatively, because the application changes its requirements (e.g. it needs different security). For instance, if an application issues high security requirements, Juno might configure itself to use BitTorrent whilst running over the Tor network [30] to ensure anonymous, encrypted communications. However, if the Tor connection fails, Juno would have to stop using BitTorrent to ensure the required level of security; this might therefore involve using a different provider offering the content through HTTPS. The actual process of re-configuration in Juno is detailed in Section 4.4.4, however, a brief overview of how this specifically takes place in the Delivery Framework is now given.

Each service in Juno is associated with (*i*) meta-data and (*ii*) and selection predicates that define the required values of the meta-data. This information is maintained in the the service-specific Configurator object, which acts as a container for each service (such as a plug-in). If a delivery undergoes some form of runtime modification, it is necessary for the plug-in to reflect this in its meta-data. Whenever a component's meta-data changes, the Configuration Engine is notified. Similarly, this also occurs if the application changes its selection predicates.

Whenever this occurs, it triggers a rebuildService operation in the Configuration Engine; this is to validate that the currently selected configuration is still optimal for the associated selection predictions. To do this, the Configuration Engine simply extracts the selection predicates from the plug-in's Configurator and then compares them against the current plug-in's meta-data. If the plug-in is no longer suitable, the selection process is re-executed to find the optimal plug-in. This simply involves repeating the previous process by looking at the meta-data of all the compatible components. If this results in a new plug-in being selected, it is simply passed into the existing Configurator using its configure method. This Configurator then extracts the previous plug-in's state before injecting it into the new plug-in and resuming the previous delivery. During this entire process, the higher layers of the middleware, as well as the application, remain unaware of these changes. Further, because content management is always performed external to any given plug-in, there is no need to transfer content between different

plug-ins.

4.6 Conclusions

This chapter has presented the Juno middleware, which offers a content-centric and delivery-centric abstraction for use with existing content systems. First, a number of key requirements were elicited based on the findings of the previous chapters. Following this, a component-based service framework was outlined, before presenting the design of Juno's content-centric functionality. Juno utilises software (re-)configuration to allow consumer-side interoperation with third party content systems, thereby opening up and unifying a huge body of content and resources. Using this position, Juno then intelligently selects between multiple content sources to best fulfil requirements issued by the application. To summarise,

- A middleware has been detailed, *Juno*, that places content-centric and delivery-centric functionality at the consumer-side
- Juno presents applications with a standardised abstraction that allows them to request content whilst stipulating their preferred access mechanism (e.g. stored, streamed etc.), alongside any other delivery requirements they have
- Juno utilises software (re-)configuration to adapt the underlying implementation of its content discovery and delivery protocols to interoperate with a range of existing systems
  - It uses the Magnet Link standard to uniquely discover content in multiple third party systems
  - It uses meta-data comparison to select the optimal provider(s) to serve an application's delivery requirements
- Juno therefore offers a content-centric and delivery-centric abstraction without requiring modifications within the network or the providers
- This can be performed whilst remaining transparent to the application

So far, this chapter has dealt primarily with software and architectural issues of building and designing Juno. It therefore has provided the foundation for offering a middleware-layer content-centric service. It should also be noted that this framework has been used by third party developers to develop discovery protocols [129], thereby further validating its design. However, many important practical real-world deployment issues have not yet been covered. In relation to the Discovery Framework, this involves the required optimisation process for improving the performance and overhead of dealing with many divergent discovery

systems. Further, in relation to the Delivery Framework, is has not yet been discussed how meta-data is generated to inform the source selection process. The next chapter builds on this chapter to explore these issues and offer solutions to make Juno truly deployable in the real-world.

Chapter 5

Addressing Deployment Challenges in Delivery-Centric Networking

5.1 Introduction

The previous chapter has detailed the design of the Juno middleware, which utilises dynamically attachable plug-ins to allow consumers to interoperate with existing discovery and delivery infrastructure. This allows applications to seamlessly utilise the content and resources of existing deployed systems, whilst further offering the ability to achieve delivery-centricity through (re-)configuration between optimal providers and protocols. A major advantage of building content-centricity in the middleware layer is therefore that it can be deployed effectively without major expense or the need for new infrastructure. This, however, creates a trade-off: maintaining client-side interoperability without modification to the provider increases availability and ease of deployment; but, by doing so, efficiency is reduced and client-side complexity is increased.

This chapter describes and addresses key deployment challenges relating to Juno's design philosophy. Specifically, it looks at offering techniques to transparently deploy content-centric and delivery-centric support without requiring the modification of existing providers (although it is hoped providers will begin to adopt explicit support in the future). This involves allowing Juno to more effectively discovery content in third party systems, as well as allowing it to generate dynamic meta-data without explicit provider cooperation.

This chapter now explores these unique issues that challenge Juno's successful deployment. First, these challenges are explored in more detail. Following this, a

mechanism is described by which the information indexed in the different discovery plug-ins can be aggregated and offered as a (higher performance) Juno-specific lookup service. Last, the challenge of generating runtime meta-data for the delivery plug-ins is explored; using the modelling techniques explored in Chapter 3, it is shown how Juno integrates prediction capabilities into its architecture.

5.2 Deployment Challenges

This section briefly explores the key deployment challenges facing Juno's approach to the discovery and delivery of content.

5.2.1 Content Discovery

Content discovery relates to the discovery of possible sources of a particular item of content. Recent content-centric networking proposals [83][93] place this functionality within the network, which involves deploying infrastructure to perform content routing. As discussed in Chapter 2, however, this creates significant barriers to deployment. These barriers centre on cost and complexity, as well as the frequent need for third party cooperation (e.g. ISPs).

The primary deployment challenge relating to content-centric discovery is therefore finding a way in which a content-centric networking abstraction can be exposed to the application without requiring the deployment of new and expensive network infrastructure/protocols. Juno's solution is to build a middleware-based integration layer that maps abstracted content-centric identifiers to traditional location-based identifiers. This is done using Magnet Link addressing, which allows content to be uniquely identified using hash-based identifiers. This allows the use of existing infrastructure without the need to utilise new protocols or introduce expensive hardware. Instead, the middleware layer must manage the interoperation with these existing content systems.

As discussed in Chapter 4, the discovery of content is performed by the Discovery Framework, which utilises dynamically pluggable *discovery plug-ins* to interoperate with external discovery systems. The beauty of this approach is that it allows clients to interoperate with a wide-range of content networks regardless of their individual methods of discovery. This means that Juno can be deployed quickly without expense or the need to modify existing providers. However, a major limitation is that there is an overhead associated with the use of multiple simultaneous plug-ins. This overhead comes in the form of memory and processing costs as well as bandwidth consumption. Perhaps, more importantly, there is also a disparity between the performance of plug-ins, in that that each will take a different length of time to answer queries. This therefore results in a trade-off at the client-side based on how long the application is prepared to wait. This obviously adds complexity to an application specification and can result in

suboptimal decisions being made rather than waiting for more extensive results to be received.

Subsequently, the primary deployment challenge relating to content discovery in Juno is the need to optimise the use of plug-ins to build high performance lookups. Importantly, however, the Discovery Framework must still maintain the advantages of using plug-ins, i.e. (*i*) fast and efficient deployment, and (*ii*) the ability to discover sources in multiple existing content systems. The challenge can therefore be summarised as needing a single integrated (high performance) lookup system that ideally offers all information indexed by all plug-ins.

5.2.2 Content Delivery

Content delivery is the process by which a node accesses an item of content. Generally, recent content-centric proposals such as AGN [83] have built their own delivery protocols within the realms of the overall content-centric network. However, due to their low level nature, these delivery protocols are not flexible or configurable to adapt to changing requirements. This is a key observation that this thesis strives to address. Juno does this by placing support for delivery-centricity within the middleware-layer, alongside the traditional discovery functionality. Once again, it uses this position to interoperate with any given provider/protocol capable of providing the desired content. This ability is exploited to allow a consumer to adapt (at the middleware layer) to be able to interact with any chosen source based on its ability to fulfil the specific requirements issued by the application.

To allow this, it is therefore necessary to allow a node to gain an understanding of each source's ability to fulfil a set of requirements. In Juno, this is done by associating each source with descriptive meta-data that can be compared against requirements structured as selection predicates. Requirements based on static items of meta-data can easily be modelled because such meta-data can be set at design-time within the plug-in. Further, 'relatively' static meta-data (i.e. meta-data that only rarely changes, e.g. the location of a server) can be automatically acquired during a node's bootstrap from a remote source (e.g. centralised repository). However, this cannot be achieved with highly dynamic meta-data, which can be constantly changing. An important example is that of *performance*, which can vary dramatically based on runtime parameters, as explored in Chapter 3. Consequently, to enable deployment, these runtime parameters must be dynamically acquired and modelled without provider-side modification.

Subsequently, the primary deployment challenge in terms of content delivery is allowing this meta-data to be generated dynamically without the explicit cooperation of third party providers. As detailed in Chapter 3, this can be achieved using various models that translate vital runtime parameters into accurate meta-data. The key challenges are therefore (*i*) collecting the necessary parameters,

and (*ii*) converting them into meta-data without provider-side cooperation.

5.3 Juno Content Discovery Service

The previous section has outlined the key deployment challenges facing Juno's design. If deployment were not a focus of this thesis, the easiest approach to discovery would be to force all providers to register references to their content with a global lookup service. Unfortunately, however, it is unlikely that in the near future many providers would support the usage of a single, unified global indexing service for their content. Further, with peer-to-peer systems such as Gnutella, it would be near impossible to achieve this in a scalable manner. Juno's configurable nature supports the later introduction of such global indexing services, however, to resolve this concern in a deployable manner it is necessary for a consumer-side approach to be also taken. This is in-line with the primarily uni-sided deployment of Juno that has so far been described. The challenge in this is therefore to allow the high performance mapping: content identifier → location identifier, without explicit cooperation from the providers.

To achieve this, Juno builds a service called the *Juno Content Discovery Service* (JCDS). This is an indexing service for content so that it can be both passively and actively indexed. Passive indexing occurs when a consumer locates an item of content and then uploads a reference to the JCDS. This can be a new item of content or, alternatively, a new source for an existing item. In contrast, active indexing is where a provider decides to explicitly upload a reference. The JCDS therefore supports immediately instigating passive indexing whilst also offering the potential for providers to later cooperate with active indexing.

This section details the JCDS; first, its principles and design is outlined. Following this, it is shown how providers can actively contribute to the JCDS by using it to index their content. Lastly, the cooperative indexing process is outlined showing how consumers can contribute to the JCDS.

5.3.1 Overview of JCDS

The Juno Content Discovery Service (JCDS) is an indexing service based on a traditional hash table abstraction. It allows content references to be uploaded and retrieved using one or more unique content identifiers. This publication process can be done by both consumers and providers alike.

The JCDS is built as a discovery plug-in that can be attached to the Juno Discovery Framework. It offers the standard plug-in service interface (detailed earlier in Table 4.23), which allows the Discovery Framework to perform queries. The JCDS's indexing method is simple; it works using a hash table abstraction through which it issues requests for content information based on one or more of its content identifier(s). Figure 5.1 provides an exemplary abstract overview

of the JCDS; in this diagram a single lookup server maintains a table mapping 'Content-A' to the four different providers*. When the consumer wishes to access 'Content-A', it sends a request to the JCDS server using the content's unique identifier. In response, it returns a list of all the providers that offer the content. Using this information, the consumer then selects its preferred source and accesses the content. Currently a server-based implementation is utilised to allow easier testing but, in practice, a KAD [106] implementation would be deployed due to its high performance and scalability. Importantly, through Juno's configurable architecture, the existing server-based implementation can be replaced with the KAD implementation without modifying any other code.

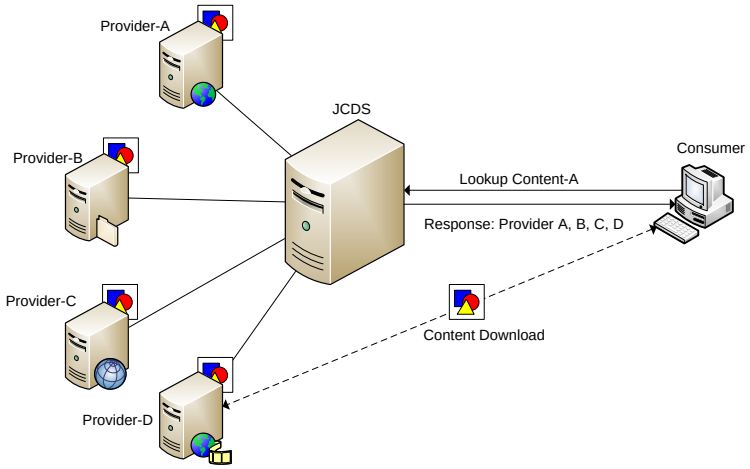


Figure 5.1: An Abstract Overview of the Juno Content Discovery Service (presented as a client-server implementation)

To allow the JCDS to function on a large-scale, it is necessary for its index to be propagated with information regarding content. This can be achieved using two approaches,

- *Provider Publication:* A provider that offers an item of content uploads a reference to it
- *Cooperative Indexing:* A consumer that accesses a previously unindexed item of content uploads a reference to it

Both approaches require that a simple mapping is performed from content identifier → location identifier. As such, content transfers do not take place using the JCDS; instead, the JCDS is used as a redirection service to lookup potential location-addressed sources. Unlike previous designs (e.g. DONA [93]), however,

*This is an exemplary diagram, a real deployment would be unlikely to use a single server

there is no binding to particular delivery systems and protocols. The JCDS is an open service that allows any provider using any protocol to publish their content.

5.3.2 Provider Publication

The most effective means by which the JCDS can be propagated with information is for all providers to voluntarily upload references to their content. For instance, if a provider places a new item of content (Content-A) on a web server (www.content.com), it can upload this information to the JCDS. This would make the source instantly discoverable. Briefly, it is important to note the distinction between the content and the source becoming discoverable. Within Juno, discovery refers to the ability to locate a set of sources for a given item of content. As such, strictly speaking, *content* is not published on Juno; instead, *sources* are published. Therefore when a source is published, the following record is placed on the JCDS: Content-A → http://www.content.com/content-a. The placement of this record on the JCDS takes three steps,

1. *Uploading Content:* The content must first be made accessible through some medium (e.g. a HTTP server, BitTorrent tracker etc.)

2. *Generating Identifiers:* The content identifiers (i.e. hash values) must be generated

3. *Publication:* A RemoteContent object must be built that contains the content meta-data alongside the location(s) it is accessible from. This must then be published on the JCDS using the hash identifier(s) to index it

The first step, *uploading content*, involves the same procedure that any provider must undergo when publishing sources. Traditionally this involves protocols such as FTP, HTTP or BitTorrent, however, any delivery system can be utilised as long as a delivery plug-in exists.

The second step, *generating identifiers*, involves calculating the hash values from the content's data. Following this, the Magnet Link can also be generated by integrating the hash values. Any hash algorithms can be utilised, however, the recommended ones are SHA1, Tiger Hash and ED2K as these are the most frequently used in discovery systems. This entire step can be easily managed using existing, stand-alone software such as BitCollider [3]. This is also packaged with Juno and allows Magnet Links to be generated automatically when content is introduced to the Content Manager. The process is relatively lightweight without excessive computational costs, even for large files (e.g. ≈25 seconds for a 720 MB film).

The third step, *publication*, involves building a RemoteContent object that contains a reference to the location(s) of the content, any meta-data, and the Magnet Link (including the hash values). An overview of this information is

Parameter	Description
IP address(es)	The IP address(es) of the access point of the content
Port(s)	The port(s) of the access point that can be connected to
URL(s)	If necessary the URL path(s) of the content
Protocol(s)	The protocol(s) supported by each source
Meta-Data	Any meta-data related to the content e.g. author, composer etc.

Table 5.1: Overview of Data within RemoteContent Object

Method	Returns	Description
provide()	void	Publishes all content in the Content Manager on the JCDS (as well as making it locally available through the default provider plug-ins)
provide (Collection <StoredContent> content)	void	Publishes a collection of items of content (as well as making it locally available through the default provider plug-ins)
getRemoteContent (String contentID)	void	Retrieves the RemoteContent object for a particular item of content

Table 5.2: Overview of IProvider Interface

provided in Table 5.1. This RemoteContent object is then simply published on the JCDS using the put method; to enable flexible querying, it should be stored under all of its content identifiers. If there is an existing RemoteContent entry for the given file on the JCDS, the two entries are merged.

To assist in the process, Juno also offers the IProvider interface, which allows applications to publish content in this manner without handling the complexities themselves. Table 5.2 details it methods; in essence, this interface allows applications to request items of content are made available. This involves performing the above three steps, as well as initiating a configurable set of provider plug-ins, which then make it locally available. These provider plug-ins can, of course, also upload the content to other remote point (e.g. Amazon S3). The default provider plug-ins are set using a standard configuration file.

5.3.3 Cooperative Indexing

Content sources can easily be published by providers, however, for this to be globally successful it would be necessary for every provider to conform with Juno's usage. This is largely unrealistic and subsequently it becomes impossible to build

a truly content-centric network, as much content would remain inaccessible. In principle, a content-centric network should, instead, provide access to every item of content in the world. To help address this, Juno utilises *cooperative indexing*. This involves using the resources and knowledge of every peer in the system to build up a database of accessible content.

The principles involved are simple; if a node discovers a new item of content, or a new source for an existing item of content, it contributes this information to the JCDS. This subsequently allows other nodes to discover the content, therefore making it visible to Juno. These two processes are now outlined.

Discovering a new item of content. This situation occurs when an item of content enters the Content Manager of a Juno node that isn't already indexed on the JCDS. This occurs when a new item of content or Magnet Link is passed to Juno by the application. A new item of content can be discovered in one of two ways, as now detailed.

First, the application might provide Juno with a new Magnet Link that can be passed into one or more plug-ins. If any of these plug-ins discover the content, it can then be downloaded; if this successfully takes place, the Magnet Link can be verified and uploaded to the JCDS. In this situation, the application must find the Magnet Link through some higher level mechanism, which is not detailed here; an example of this might be through a software management service.

The second way is if the Juno Content Manager encounters a new item of content through alternate means. This would happen if the application utilises an out-of-bands discovery mechanism (i.e. a non Magnet Link). In this case, the Content Manager would generate a new Magnet Link from this content and upload it onto the JCDS alongside the source information.

The former would be the most frequent situation, i.e. the introduction of a new Magnet Link to Juno via an application. As an application would usually be deployed on multiple nodes, it would only take a single node to use the application before the JCDS indexes the appropriate content. When this first node receives the Magnet Link the following steps take place,

1. Acquire a Magnet Link through an application-specific mechanism

2. Query the Discovery Framework with the Magnet Link and download the content (if available)

3. Validate that all the Magnet Link's content identifiers are correct by re-executing the necessary hashing algorithms

4. Augment the Magnet Link with any missing hash algorithms and meta-data extracted from the content

Method	Description
registerSource (RemoteContent)	Publishes an item of content on the JCDS if it is not already indexed. If it is already indexed, the RemoteContent object is checked for any sources that aren't already indexed

Table 5.3: Overview IJCDS Interface

5. Upload the Magnet Link onto the JCDS as well as any sources of the content that have been discovered and verified; it should be stored using each of the content's hash values

The JCDS functionality is implemented within a discovery plug-in. However, to enable the above cooperating indexing procedure to take place, the JCDS plug-in also supports an extended interface, as shown in Table 5.3. The IJCDS interface simply allows a RemoteContent object to be passed into the plug-in. This object can either contain a new item of content or, alternatively, a new source for an existing item of content. The Discovery Framework, however, does not need to handle this; instead, every RemoteContent object encountered by the Discovery Framework can simply be passed into the JCDS plug-in, which will then analyse it to check if there is anything new. Therefore, whenever the Discovery Framework discovers an item of content through a plug-in that is not the JCDS, it is passed into the JCDS to extract any possibly new information.

When the JCDS component receives the RemoteContent object, it first performs a lookup to verify whether the item of content is already indexed. If it is not, the piece of content is retrieved from the Content Manager (when it is available) and validated against the Magnet Link and any missing hash identifiers generated. If this process successfully completes, the RemoteContent object is uploaded onto the JCDS.

Discovering a new source of content. This situation occurs when an item of content is already indexed on the JCDS but a node discovers a new source that is not indexed. Every RemoteContent reference on the JCDS is associated with a list of possible sources (e.g. Limewire peers, HTTP servers etc.). Every time a client queries a plug-in with a Magnet Link, it is possible for a new source to be discovered; in peer-to-peer systems such as Limewire this may occur on a highly frequent basis as peers come and go. The process of contributing content sources is simple and can be outlined as follows,

1. Acquire a Magnet Link though an application-specific mechanism

2. Query the Discovery Framework with the Magnet Link. This results in the plug-ins (including the JCDS) being queried

3. If any sources are discovered that are not also returned from the JCDS, upload the references to the JCDS

4. The JCDS will verify the sources are not already indexed and then add them to the RemoteContent object

The functionality for this is implemented in the JCDS component alongside the functionality for discovering new items of content. Therefore, whenever the Discovery Framework performs a lookup using other plug-ins, the aggregated RemoteContent object is passed to the JCDS (using the IJCDS interface). The JCDS component then compares its own results against the RemoteContent object generated by all the other plug-ins. If any new sources have been discovered, these are then uploaded to the JCDS index. These, however, must be first validated against the Magnet Link's hash identifiers; consequently, only sources that are downloaded from are uploaded to the JCDS. It should also be noted that many delivery protocols such as BitTorrent, Limewire and eD2K support the exchange of source information between peers and therefore the JCDS often will only require a single valid source to act as a gateway into the system. Following this, it is often possible to acquire a larger set of up-to-date sources through this gateway peer. This is most evident in BitTorrent-like protocols that use a centralised gateway (a tracker) that can be used to retrieve a set of current peers.

5.3.4 Managing Plug-ins

So far, the standard approach for querying plug-ins has been to pass a request into all available plug-ins; this is to gain access to as many sources as possible. However, the use of the JCDS means that it is possible for a single JCDS plug-in to potentially index the content of all the other plug-ins. As such, sometimes there is no benefit in querying all the plug-ins as this simply creates an undue overhead. It is therefore important to manage this process; this is done within the DiscoveryPolicyMaker component, as detailed in Section 4.5.4. This component is contacted by the Discovery Framework whenever a request is received from the application. It is the responsibility of the DiscoveryPolicyMaker to return a description of which plug-ins to query and over what timeframe. This allows queries to be treated differently; for instance, a query for a small picture file might only utilise the quickest plug-ins alongside a short timeout (< 500 ms) to ensure that there isn't a significant delay during the query.

Like any other component, the DiscoveryPolicyMaker component is pluggable and can be re-configured during runtime. As such, a specialist JCDS DiscoveryPolicyMaker component is attached to the Discovery Framework whenever the JCDS is used. This allows the Discovery Framework's behaviour to be shaped by the knowledge of the JCDS. The JCDS DiscoveryPolicyMaker component is implemented using a simple probabilistic algorithm that creates a trade-off between

the overhead and accuracy of results. Its purpose is to minimise overhead whilst maintaining an acceptable degree of accuracy. This trade-off must be made on a per-node level to take into account individual preferences and resources.

When the JCDS DiscoveryPolicyMaker is first attached to the Discovery Framework, all available plug-ins are utilised (this includes the JCDS plug-in). This means every plug-in is queried for every request. This process has an obviously high overhead but allows (*i*) optimal results to be gained, and (*ii*) statistics to be generated based on each plug-in. After every lookup, the JCDS plug-in compares its own results against the results of the other plug-ins (this is possible because the Discovery Framework always passes the information in using the registerSource method). The results of this comparison are then passed to the JCDS's DiscoveryPolicyMaker component. In fact, these are actually implemented within the same software component, i.e. a single component offers both the IDiscoveryPlugin and IDiscoveryPolicyMaker interface.

This comparison consists of comparing the percentage of total sources that are discovered by the JCDS. For instance, if the plug-ins collectively find 10 sources of the content and the JCDS plug-in discovers 8 of those, this means that the the JCDS indexes 80% of available sources. Therefore, a node that consistently accesses well indexed content is likely to continue accessing well indexed content in the future, thereby negating the need for the other plug-ins. Importantly, this comparison is performed against each plug-in individually. For example, for two plug-ins, P^1 and P^2, it is possible that the JCDS indexes all sources provided by P^1 but none of the sources indexed by P^2. In such a situation, P^1 can be detached without any loss of performance whilst the opposite exists for P^2.

To enable this, the JCDS DiscoveryPolicyMaker maintains a moving average *hit rate* for each plug-in. A *hit* occurs when a plug-in finds a source that is also found by the JCDS, whilst a *miss* occurs when a plug-in finds a source that is not indexed by the JCDS. This is calculated for plug-in i using the equation,

$$\frac{hit^i}{hit^i + miss^i} \tag{5.1}$$

By calculating this hit rate over time, the JCDS DiscoveryPolicyMaker can then make appropriate decisions as to which plug-ins to remove and which to keep. To enable this decision process it is necessary to provide three important parameters,

- *Detachment Threshold:* The average percentage of sources that must be discovered by the JCDS before a plug-in can be detached
- *Minimum Queries:* The minimum number of queries before comparing the running average to the threshold
- *Probe Probability:* The probability of re-initiating old plug-ins to re-check their performance

Using these parameters, once the minimum number of queries have been generated, the DiscoveryPolicyMaker begins to compare the moving averages for each plug-in against the threshold. Any plug-in that offers an average that exceeds the threshold is detached. Using the previous example, this means that P^1 would be detached whilst P^2 would remain attached. Last, it is also important to allow plug-ins to 'redeem' previous bad performance. This is particularly beneficial for peer-to-peer discovery systems that can have highly variable performance. Therefore, every minute each plug-in has the probabilistic opportunity to be re-attached for the minimum number of queries again. This allows a plug-in to re-enter use to verify whether or not the JCDS still indexes the sources that it offers. This process is formalised in the following algorithm,

```
1: Source[] jcdsSources = all sources discovered by JCDS
2: Source[] pluginSources = all source discovered by other plug-ins
3: int[] hitCounter = # of sources for each plug-in also found by JCDS
4: int[] missCounter = # of sources for each plug-in not found by JCDS
5:
6: for i = 0 to pluginSources.length do
7:    if pluginSources[i] ∈ jcdsSources then
8:       ++ hit counter for plug-in i
9:    else
10:      ++ miss counter for plug-in i
11:   end if
12: end for
13:
14: //Iterates through each plug-in i
15: for i = 0 to hitCounter.length do
16:   if hitCounter[i]/(hitCounter[i] + missCounter[i]) > threshold then
17:      Stop using plug-in i and replace it solely with the JCDS
18:   end if
19: end for
```

5.4 Delivery-Centric Meta-Data Generation

5.4.1 Overview

Section 5.2 has provided an overview of the deployment challenges facing Juno's design. The previous section has shown how passive indexing can be used to discover content sources in existing systems without explicit support from providers. However, once these providers have been found to offer an item of content, it is necessary to generate accurate meta-data that represents their abilities to satisfy a set of requirements. This is a simple process when dealing with static meta-

data that does not vary at runtime. A good example of this is whether or not a provider supports data encryption; this is because the meta-data is a static facet of the delivery protocol being used. For instance, BitTorrent would implicitly not offer encryption whilst HTTPS would. Consequently, all such meta-data can be defined at design-time within each delivery plug-in.

In contrast, dynamic meta-data is based on varying runtime characteristics that cannot be predicted at design-time. The primary example used up until now is *performance*, which is based on a number of runtime parameters that are important to individual systems. Chapter 3 has provided an in-depth analysis of three large-scale delivery systems to show how their performance varies dynamically. Consequently, from this it has been concluded that a collection of application instances would only be able to optimise their performance if it were possible for each to be able to dynamically configure between the use of different providers, based on its specific environment. However, to enable this, it is necessary for a node to be able to ascertain the potential throughput that it will receive from each of the providers in advanced. This can be generalised to say that, any meta-data based on dynamic characteristics must be generated accurately at runtime to represent the predicted values that will exist if the application chooses to use a particular plug-in. If this is not possible, or if it is not accurate enough, it will not be possible to correctly configure between the use of different providers.

This section explores the deployment challenges related to this issue. First, a provider-based solution is detailed in which a provider is required to expose a reflective interface that offers runtime information about its current characteristics (e.g. loading). However, evidently, this requires provider-side cooperation; subsequently, it is likely that only a subset of providers will offer this support. To address this, it is then detailed how the passive techniques detailed in Chapter 3 can be introduced into Juno's software architecture to generate runtime meta-data on behalf of the delivery plug-ins. This is done by embodying the various prediction algorithms within components that can be attached at runtime based on the characteristics and requirements of the individual node.

5.4.2 Reflective Provider Interface

The first approach that can be taken represents the ideal scenario in which providers are prepared to expose information about the capabilities and resources of their services. This, for instance, is already implicitly done by many content distribution networks (e.g. Akamai [1]) by performing redirection between edge servers based on performance metrics. This is done by monitoring the status of the available resources and then internally publishing the information to the redirection service [146]. Subsequently, this could easily be extended to allow external access to such information as well.

As detailed in Chapter 4, reflection is the ability to gain introspection into

Method	Returns	Description
lookupMetaData (String magnetLink)	Vector <MetaData>	Provides meta-data containing a description of a provider's behaviour and performance when serving a particular item of content
lookupMetaData (String magnetLink, Hashtable<String, Object>)	Vector <MetaData>	Provides meta-data containing a description of a provider's behaviour and performance when serving a particular item of content based on certain characteristics provided by the consumer (e.g. its location, user group)

Table 5.4: Overview of Delivery Reflective Interface

the behaviour of a particular entity. In essence, it turns a black-box system into a white-box one. A possible way that this could be implemented is to place additional meta-data into the RemoteContent objects that are retrieved through the JCDS and the discovery plug-ins. This meta-data could describe the behaviour, capabilities and performance of any sources that it lists. However, this would often be an ineffective approach as such information is highly runtime dependent and, as such, would need frequent updates, creating a significant overhead for the discovery systems.

As an alternative to this, this thesis promotes the provider-side exposure of this information through remotely accessible interfaces (i.e. a web service). Subsequently, every provider would voluntarily offer this interface to allow (potential) consumers to first query it as to its current operating conditions, alongside any other meta-data of interest. Table 5.4 details the proposed interface, which would be exposed using a remote procedure call mechanism such as SOAP. This allows a consumer to query a provider as to any item of meta-data relating to its provision of a given item of content. In a controlled environment such as Akamai this would be offered through a single logical web service interface. In contrast, a decentralised system such as Limewire would require each potential source to be contacted and queried individually before the data is aggregated.

The actual information returned by the interface would be an extensible set of meta-data that could vary between different systems. Using such information, the consumer could then make an informed choice about which provider to select. Examples of important meta-data include such things as available upload bandwidth, network coordinates, monetary cost, as well as reliability and security issues.

5.4.3 Techniques for Passively Generating Meta-Data

The previous section has detailed one approach for allowing consumers to differentiate between the runtime characteristics of different providers. However, for this to work it is necessary for (ideally) all providers to support the standardised interface. This, unfortunately, makes its deployment unlikely as uptake would probably be very slow. Further, some providers would likely refuse its usage due to privacy and security concerns; for example, a server-based provider would be hesitant to expose information regarding its loading for fear of attackers taking advantage.

To address this, it is also important for Juno to be able to passively generate any pertinent meta-data for each potential provider, without explicit support. Consequently, Juno also supports the use of consumer-side meta-data generation. This involves passively collecting information relating to a provider so that a model can be built of its current operating conditions. Therefore, in this sense, passive refers to the ability for the consumer to collect the information without the explicit support of the provider. This means that such techniques can be used with any providers, regardless of their knowledge of Juno.

As previously discussed, the key item of dynamic meta-data considered important is *performance*. A number of approaches can be taken to generating meta-data in a passive manner. Generally, three approaches exist for predicting the performance that will be received from a provider,

- *Active Probing:* This involves generating probe traffic and sending it to the provider to gauge the performance that can be achieved
- *History-Based Predictions:* This uses past interactions to extrapolate the performance that will be achieved in future interactions
- *Monitoring Services:* This uses third party services that monitor a system to collect information that can be shared amongst many consumers

These techniques can be used to directly acquire throughput predictions or, alternatively, to acquire relevant parameters that can be transformed into predictions based on a model. Chapter 3 explored the dynamics of three important delivery protocols to understand their runtime behaviour. Alongside this, methods for generating runtime performance predictions were detailed (and validated) for each delivery protocol. These techniques can therefore be immediately used to accurately generate throughput predictions without the need for any prior deployment. The three techniques can be summarised as follows,

- *HTTP:* Throughput is calculated using the TCP model detailed in [110]; the necessary parameters are obtained using the iPlane monitoring services [102]

- *BitTorrent:* Throughput is calculated using the model detailed in [115]; the necessary parameters are obtained using public upload capacity measurements [79]
- *Limewire:* Throughput is calculated using history-based predictions; the throughput of past downloads is logged to generate a moving average

The above procedures have been validated in Chapter 3, to discover that such passive approaches can accurately generate performance-oriented meta-data with a controlled degree of overhead. Alongside these mechanisms, a currently utilised delivery system can also provide accurate meta-data about itself simply by measuring its own download rate. This allows bad decisions to be quickly rectified by repeating the meta-data generation process for each potential source and comparing it against the meta-data of the active delivery system.

5.4.4 Meta-Data Generation Plug-ins

The previous section has provided a brief overview of the passive techniques detailed in Chapter 3 for generating predictive performance meta-data. These, however, only offer an example set of possible approaches that are conducive with the general requirements of most applications. There are, in fact, many different approaches that can be taken that vary greatly. Consequently, a number of trade-offs can be observed when generating meta-data. For instance, an active probe in which a small download is performed will generally result in superior accuracy than using a model based on parameters acquired from monitoring services. This, however, will clearly result in a far greater delay and overhead, making it unsuitable for small items of content or low capacity devices.

These trade-offs mean that it is undesirable to build delivery plug-ins with predefined generation mechanisms. To do so would force many devices and applications to use unsuitable prediction techniques or, alternatively, create the need for multiple delivery plug-ins to be implemented for use in different environments. To address this, meta-data generation in Juno is performed using pluggable components that offer the ability to generate meta-data on behalf of the delivery plug-ins. Table 5.5 provides an overview of the component's interface, IMeta-DataGenerator. This interface accepts a request for a given item of meta-data for a particular item of content from a remote point. Such plug-ins can be used extensively to generate meta-data on behalf of the delivery plug-ins without the need to handle the burden themselves.

Each technique detailed in Chapter 3 is therefore embodied in an independent pluggable component that can be accessed by any delivery plug-in. An important question is therefore how nodes can select between the different possible meta-data generator components. Generally, this is managed by the application that operates above Juno by simply stipulating the preferred meta-data generators

Method	Returns	Description
generateMetaData(String attribute, RemoteContent content)	Object	Generates a particular type of meta-data and returns it to the caller
generateMetaData(Vector <String> attributes, RemoteContent content)	Vector	Generates a given set of meta-data and returns it to the caller

Table 5.5: Overview of IMetaDataGenerator Interface

within a configuration file. This file contains entries such as,

juno.delivery.http.HTTPDelivery=juno.generators.http.HTTPMetaDataGenerator

which indicate that a particular plug-in should have its meta-data generated by another component (both identified using their canonical class names). Multiple entries can also be given to allow a delivery plug-in to have meta-data generated by more than one component.

One limitation of this approach, however, is that often an application would not wish to handle this level of detail. To address this, if no configuration file is offered, Juno manages the selection process itself. This is done using the same mechanism used for the selection of any other component. Therefore, when the Discovery Framework issues a buildService request to the Configuration Engine for a new delivery plug-in, it also attaches further selection predicates. These selection predicates describe certain requirements that the node has for its meta-data generators. Consequently, the Configuration Engine will select a Configuration that contains both a delivery plug-in and a meta-data generator that matches these requirements.

Meta-Data	Type	Description
Accuracy	double	The average correlation coefficient of its accuracy
Delay	int	The time required to generate the prediction (in ms)
Download Overhead	long	The download bandwidth required to generate the prediction
Upload Overhead	long	The upload bandwidth required to generate the prediction

Table 5.6: Meta-Data Exposed by IMetaDataGenerator Components

Table 5.6 details the default meta-data that must be exposed by each of the generator components. These are performance and overhead metrics that can be

used as a trade-off. Evidently, the values of this meta-data might change depending on the node that it is operating on; for instance, the generation delay would be greater for a high capacity client when using active probing because it would need a greater length of time to reach TCP steady state. An IMetaDataGenerator component must therefore take this into account by retrieving the necessary information from the Context Repository.

Meta-Data	Comparator	Value
Accuracy	HIGHEST	N/A
Delay	<	0.05 * contentSize / ContextRepository.get("availableDown")
Download Overhead	<	ContextRepository.get("availableDown")
Upload Overhead	<	ContextRepository.get("availableUp")

Table 5.7: Selection Predicates for IMetaDataGenerator Components

Currently, the selection predicates utilised by the Discovery Framework are relatively simply. They are calculated dynamically on every request using a combination of the content size and available upload bandwidth. Table 5.7 provides the selection predicates, which request the generator with the highest accuracy whilst operating within strict delay and overhead requirements (based on the content size and the available upload bandwidth). These selection predicates are therefore bundled with the selection predicates provided by the application to construct a component configuration that consists of a delivery plug-in that is connected to the optimal generator component.

5.5 Conclusions

This chapter has explored the deployment challenges facing Juno's design. Regarding content discovery, it was identified that the primary challenge is reducing the overhead of using multiple simultaneous discovery plug-ins, whilst also improving the performance of slower ones. Regarding content delivery, the primary challenges was identified as generating accurate predictive meta-data to reflect current (and future) operating conditions. To assist in the discovery process, a cooperative lookup system called the Juno Content Discovery Service was detailed, whilst the deployment challenges of delivery were addressed using a combination of the passive modelling techniques detailed in Chapter 3, as well as architectural adaptation. To summarise, the following conclusions can be drawn from this chapter,

- Two key deployment challenges exist with Juno's design philosophy

  - The need to reduce the overhead and delay of using multiple heterogeneous plug-ins
  - The need to generate dynamic meta-data passively to ensure backwards compatibility with providers

- An indexing system, the Juno Content Discovery Service (JCDS), can be used to address the above discovery challenge
  - It allows providers to actively upload references to themselves for discovery
  - It allows consumers to collectively upload both identifier and source information about third party providers

- The integration of meta-data generation components into the Juno framework can address the above delivery challenge
  - It is possible for the Discovery Framework to dynamically select the generation technique used at request-time
  - Different approaches can be utilised in different environments to optimise the process

- An open reflective interface can be used to allow providers to offer meta-data relating to their own operating conditions

This chapter has dealt with the key deployment challenges facing Juno. Therefore, alongside Chapter 4, a full system specification has been offered to address the first two goals of this thesis. It is now necessary, however, to evaluate how effectively these goals have been achieved. The next chapter evaluates Juno's design as presented.

Chapter 6

Analysis and Evaluation of the Juno Middleware

6.1 Introduction

The previous chapters have explored the notion of content-centric networking to create a new abstraction that extends content-centricity to also include the concept of delivery-centricity. The key research goals of this thesis are realised through the design and implementation of the Juno middleware, alongside key techniques to enable its deployment. This implementation therefore constitutes the manifestation of the principles developed during this thesis. Consequently, it is necessary to investigate this implementation to evaluate this thesis.

This chapter analyses and evaluates this realisation of delivery-centricity by inspecting Juno's design. Specifically, it evaluates how effectively the key decisions made during Juno's design offer support for a content-centric and delivery-centric abstraction. This involves evaluating the Discovery and Delivery frameworks to understand how effectively Juno can offer the required support.

First, the approach taken by the evaluation is detailed, looking at how the techniques used map to the core research goals of the thesis. Following this is the Discovery Framework evaluation, which inspects the performance and overhead aspects of using Juno's discovery techniques. Next, the Delivery Framework is evaluated through a number of relevant case-studies that show how Juno achieves delivery-centricity, with a focus on improving delivery performance. Last, a critical evaluative summary is provided based on the design requirements detailed in Chapter 4.

6.2 Overview of Applied Evaluation Methodology

This section provides an overview of the evaluation performed within this chapter. First, it is necessary to revisit the original research goals to understand how the work in previous chapters, as well as the evaluative work in this chapter, contribute towards the overall thesis. Following this, details are provided relating to the methodology taken for evaluating the thesis.

6.2.1 Research Goals Revisited

Before entering the evaluation, it is important to gain an understanding of the overall perspectives and methods taken. To achieve this, the research goals presented in Chapter 1 are revisited,

1. To define an extended notion of a content-centric abstraction encompassing both discovery and delivery, capturing the requirements of (existing and future) heterogeneous content systems

2. To design and implement an end-to-end infrastructure that realises this (new) abstraction in a flexible manner

3. To show that this concept is feasible and can be deployed alongside existing systems in an interoperable way

These research goals have been investigated within the previous chapters of this thesis; the first and second goals have been fulfilled in Chapters 4 and 5 by defining a new delivery-centric abstraction, as well as a middleware framework to implement it. The third goal has further been fulfilled by using a highly deployable approach that supports backwards compatibility with existing content systems. The primary outcome of this thesis is therefore the realisation of these concepts through the development of the Juno middleware. Consequently, the key question to be asked is *how effective is Juno at fulfilling these goals?*. To achieve this, a range of techniques are used to evaluate the fulfilment of the above three goals, before performing a critical evaluative summary focussing on the five design requirements detailed in Chapter 4.

Juno can be decomposed into two primary elements. First, the Discovery Framework is responsible for offering traditional content-centric functionality through its ability to map content identifiers to location identifiers. Then, second, the Delivery Framework, which is responsible for supporting delivery-centricity through transparent component (re-)configuration. Collectively, these two frameworks offer an end-to-end middleware infrastructure for supporting the key tenets of this thesis. A number of differences, however, can be observed between these two frameworks, thereby making a single system-wide evaluation inappropriate.

The rest of this section investigates the possible evaluation techniques that can be used to address this problem.

6.2.2 Evaluation Methodology and Techniques

The previous section has revisited the research goals to ascertain the necessary aspects of Juno that must be analysed and evaluated. When performing an evaluation, it is important to understand the techniques available. In a distributed system such as Juno, three key approaches can be identified,

- *Simulation* involves "the imitation of the operation of a real-world process or system over time" [37]. Specifically, it allows a software model of a system to be executed (with certain environmental and parametric inputs) to create a set of evaluation outputs. The advantages and disadvantages are,

  + It allows large-scale, controlled experiments to be performed to ascertain results based on a range of potential situations.

  − It is possible, however, for incorrect assumptions in the simulation model to invalidate results.

- *Mathematical Modelling* involves building a formal mathematical representation of the system, in a similar manner to simulation. Specifically, it allows a set of equations to be devised that can utilise certain environmental and parametric inputs to create a set of evaluation outputs. The advantages and disadvantages are,

  + It allows large-scale, controlled experiments to be performed, generally in a short period of time.

  − It is highly limited in its ability to capture real-world complexity, making a sophisticated systems-based evaluation difficult.

- *Emulation* involves building a real-world implementation of a system and deploying it on a set of nodes connected via an emulated network. An emulated network possesses the ability to "subject traffic to the end-to-end bandwidth constraints, latency, and loss rate of a user-specified target topology" [142], thereby placing the system in a controlled environment. The advantages and disadvantages are,

  + It allows a prototype implementation to be tested, alongside real-world protocols, software and networks, thereby mitigating possible problems with the results and improving accuracy.

  - − The scalability of such tests, however, is reduced due to the increased resource requirements of using real-world implementations. Consequently, large-scale experiments often cannot be performed.

- *Real-world testing* involves deploying a system implementation for use in the real-world, potentially by third party testers. The advantages and disadvantages are,

  + It allows a prototype implementation to be tested on a large-scale, alongside real-world protocols, software and networks, thereby removing the potential to make incorrect assumptions in a system model.
  - − It is rarely possible to achieve such tests as it necessary to provide incentives for testers to use the system. Also, all experiments must take place outside of controlled conditions making the results non-deterministic and less tractable.

As outlined, each approach has different advantages and disadvantages, making their suitability variable for different tasks. Consequently, all three approaches are utilised within this thesis.

The Discovery Framework relies on the cooperative indexing of content by a wide range of nodes, applications and users. Consequently, it is necessary to utilise an evaluation method that can model Juno's discovery aspects on a large-scale. The techniques that fulfil this need are simulation and real-world testing. Obviously, a real-world deployment would be beneficial in that it would allow real user behaviour to be investigated. However, it would be impossible to perform this on a sufficiently large-scale to gain the necessary results. Further, as this would be performed in a non-controlled environment, it would be difficult to build a results set that could assist developers in understanding how their applications might perform with Juno. Consequently, to address this, the Discovery Framework is evaluated using large-scale simulations with models taken from existing measurement studies. This allows the discovery algorithms to be tested in a wide-range of environments to gain an understanding of how various applications would operate over Juno.

In contrast to the above, the Delivery Framework operates on a far smaller-scale as each instance makes individual per-node choices that do not involve the cooperation of other nodes. However, unlike the discovery evaluation, it involves a number of third party protocols that are already deployed in the Internet. The evaluation of the Delivery Framework must therefore validate the ability of Juno to perform (re-)configuration and interoperation to achieve delivery-centricity. Clearly, this cannot be achieved using modelling or simulations as it is necessary to evaluate Juno's underlying architectural design and implementation. The techniques that fulfil these needs are therefore emulation and real-world experiments. Several real-world experiments have already been performed in Chapter 3

to quantify the potential benefits that could be achieved using dynamic delivery (re-)configuration. Therefore, the next step must be to validate that these benefits can actually be gained by Juno. This cannot be performed using real-world testing as it is impossible to control the characteristics of third party content systems. Thus, making it difficult to gain an evaluative understanding of how Juno responds to different situations. Consequently, the Delivery Framework is evaluated (in conjunction with the measurement results from Chapter 3) by performing a number of controlled emulations in which a variety of case-study environments are created to evaluate how Juno performs. Through this, a number of providers can be setup (and configured between) using real-world implementations whilst also maintaining a strict level of control over the test environment.

The rest of this chapter is structured into three main sections. First, the Discovery Framework is evaluated, looking at the performance and overhead issues of building content-centric lookups in an interoperable and deployable manner. Following this, the Delivery Framework is also evaluated to validate whether or not the design approach is feasible, as well as to highlight the strengths and limitations of offering a delivery-centric abstraction at the middleware layer. Finally, a critical evaluative summary is presented in which the design requirements are revisited to validate how effective Juno has been at fulfilling them.

6.3 Content-Centric Discovery in Juno

The first aspect of Juno's operation is the discovery of content. This is generally considered the paramount operation in a content-centric network. The process involves mapping a unique content identifier to one or more potential sources. In Juno, content-centric discovery is handled by the Discovery Framework using a number of configurable discovery plug-ins, in conjunction with a cooperative indexing service called the Juno Content Discovery Service (JCDS). Collectively, this allows a Juno node to interact with existing content providers to ascertain their ability to serve an item of content.

This section evaluates the Discovery Framework based on the research goals and design requirements of this thesis. First, the methodology is outlined, detailing the techniques used. Following this, a performance analysis is performed using simulations; this studies the ability of the Discovery Framework to extract information from the plug-ins and cooperatively index it on the JCDS. Alongside this, the overhead is detailed to show the costs associated with performing discovery in this fashion. Last, two case-studies are explored in which existing measurement studies are used to simulate real-world systems to provide a detailed overview of how the Discovery Framework would operate if it were actually deployed in current Internet environments.

6.3.1 Methodology

This section provides a detailed overview of the methodology used to evaluate Juno's content discovery.

Evaluation Overview. Juno's Discovery Framework offers support for interoperable content-centric discovery in third party systems. This is achieved using plug-ins that offer the necessary protocol support to interact with external lookup systems. Consequently, it has been designed to support a content-centric abstraction that is instantly deployable alongside existing systems (in-line with research goals 2 and 3).

The primary evaluation question that must be asked regarding the Discovery Framework is therefore, *how effectively can Juno discover content in third party systems?* Evidently, this is required to validate the claim that Juno's design, indeed, satisfies the interoperable and deployable needs of the research goals. This question can be decomposed into both performance and overhead aspects. First, it is necessary to validate that using Juno's approach can offer sufficient performance to allow content sources to be effectively discovery. The measure of this is generally application-specific (e.g. query delay); however, a common factor is the need for as many sources as possible to be discovered. This is necessary to ensure optimal delivery-centric decisions can be made; consequently, this is considered as the primary metric of performance. Following this, it is also necessary to ensure that this takes places with controlled levels of overhead.

To perform the evaluation, a large-scale simulation study is undertaken, which allows a range of controlled environments to be investigated in detail. The justification for this choice has been explored in Section 6.2.2. Through the use of well-known models, it becomes possible to highlight how Juno's discovery mechanisms would operate in the real Internet. The following sections provide an overview of the simulator and models used before detailing the evaluation metrics chosen.

Simulations. To understand the behaviour of the JCDS, a simulator has been built that models its operation based on a number of important parameters. The simulator operates from the perspective of a single application utilising Juno without the possibility for any other applications to access the same content. The results presented in this section therefore provide details of the *worst-case* scenario. An alternative to this is, of course, the existence of a set of applications that share many content interests. In such a situation, far higher performance would be achieved because the population of users accessing the content would be greater therefore ensuring more up-to-date information is contributed.

The simulator utilises three primary parameters: popularity (p), request rate (λ) and number of objects (n). These parameters collectively dictate which items

of content are indexed on the JCDS; p dictates which items are requested by each node; λ dictates the time it takes for these requests to be issued; whilst n has a significant effect on both these parameters. The probability of object i being available at time t on the JCDS can therefore be modelled by $\lambda \cdot t \cdot p(i)$, where $p(i)$ is the per-request probability of item i being requested.

The simulator is a discrete time simulator that operates in rounds. Every round, on average, λ requests are generated using a Poisson distribution [119]; this involves first checking the JCDS and then the discovery plug-ins. If the content is not indexed on the JCDS yet, the content is downloaded (and verified) before uploading a reference to the JCDS. The download time is defined by two factors: (i) file size and (ii) consumer download speed. The file sizes are based on the traces taken from the macroscopic BitTorrent measurement study detailed in Section 3.3; Figure 6.1 shows the cumulative distribution of file sizes for all file types. The consumer download rates are taken from [56]; this data set includes the download bandwidth of 1894 American DSL and Cable users. Using these data sets, the download time can be generated using the equation, $\frac{filesize}{bw}$. An obvious limitation of this, however, is that it is not always possible for a client to saturate its download link; to take this into account, a random saturation level is generated between 0.1 and 1. This therefore results in the download time being, $\frac{filesize}{bw \cdot saturation}$. Once a download has completed, the Magnet Link and source information are verified and uploaded to the JCDS.

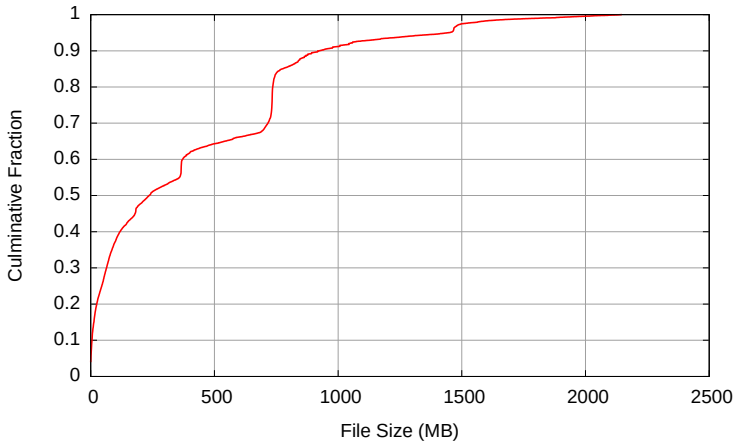


Figure 6.1: Cumulative Distribution of File Sizes observed in BitTorrent Traces

By default, four discovery systems are utilised in the simulation, as detailed in Table 6.1. The plug-in success rate refers to the probability that a plug-in can provide information regarding a given content item; evidently, at least one plug-in will always return information. This represents a likely situation in which there is some relationship between the deployed discovery systems and the provider of the content identifiers. The providers returned from each discovery system have

Plug-in	Type	Plug-in Success Rate	Churn
Server Lookup	Web Service	1	No
Gnutella	Flooding Overlay	0.4	Yes
eD2K	Client-Server	0.6	Yes
KAD	Iterative DHT	0.8	Yes

Table 6.1: Overview of Inspected Discovery Protocols

a uniform probability of being selected for delivery. Alongside this, each plug-in is associated with a boolean to indicate whether or not if suffers from churn. To model churn accurately in affected discovery systems (i.e. peer-to-peer systems), the Gnutella churn trace is taken from [132]. This model provides a mapping between the age of an entry on the JCDS and the probability of it being stale.

Finally, to study overhead, each plug-in is also associated with a background and per-request overhead. To configure the simulator with representative information, a number of overhead measurements have been taken from real-world systems. Table 6.2 details the overheads of these various dominant discovery protocols. These were gained by performing local measurements using Wireshark and SocketSniff, which allow the inspection of the network interactions of different applications. The server lookup is a centralised indexing system (measurements taken from the Bitzi web service [4]); Gnutella [10] is a fully decentralised query overlay; and KAD [106] is a popular DHT. Note that the JCDS uses the same overhead values as KAD; this is because KAD is the most likely DHT to be used for deployment due to its robustness and proven performance in the real-world [61].

Table 6.2 also provides an overview of the different applications used to measure the protocols (all support Magnet Link addressing). To gain the request overhead, 10 requests were generated for each protocol and then the averages taken. To gain the background overhead, each application was monitored for a period of 15 minutes without activity and then the average taken. For both values, this includes both upload and download overhead. It is important to note that these are exemplary figures that are used to provide a representative evaluation; they do not, however, offer exact details as many more complicated aspects have been abstracted away from. For instance, a Gnutella client will witness various levels of background overhead based on the behaviour of its neighbours; the model does not take this into account, instead a single standard background overhead is used.

Evaluation Metrics. To study the performance of the JCDS, it is important to compile a set of evaluation metrics. These inspect the ability of the JCDS to provide a given piece of information after a certain period of operation. To

Plug-in	Per-Request Overhead (bytes)	Background Overhead (bytes per min)	Application Used
Server Lookup	7342	0	IE Explorer
Gnutella	20,617	6020	Limewire
eD2K	16,899	570	eMule
KAD	1168	9753	eMule
JCDS (KAD)	1168	9753	N/A

Table 6.2: Overhead of Discovery Plug-ins (in Bytes)

measure the performance, three metrics are used; the first is hit rate,

$$hitrate = \frac{hits}{\lambda \cdot t} \tag{6.1}$$

where *hits* is the number of successful requests that are issued to the JCDS. A request is considered successful if a Magnet Link has been registered on the JCDS under the requested content identifier. This therefore only requires that one previous consumer has accessed the file and uploaded the Magnet Link. This metric therefore measures the frequency at which an application instance with only a single identifier can gain access to a full set of identifiers (to allow subsequent access to any discovery system). The limitation of this, however, is that a consumer would have to generate further queries in each of the plug-ins using the acquired Magnet Link to locate a set of available sources. To measure this, a further metric is therefore defined,

$$sourcerate = \frac{sourcehits}{\lambda \cdot t} \tag{6.2}$$

where *sourcehits* is the number of requests to the JCDS that successfully yield one or more available sources. This models the percentage of requests that are provided with at least one source without the need to query further plug-ins. This therefore captures the frequency at which an application can solely use the (low latency and low overhead) JCDS to gain access to the content. A final metric is also defined to ascertain the number of requests to the JCDS that return a complete list of all potential sources,

$$fullsourcerate = \frac{fullsourcehits}{\lambda \cdot t} \tag{6.3}$$

where $fullsourcehits$ is the number of requests to the JCDS that yield the full set of available sources without the need to query any further plug-ins. Consequently, this metric measures the percentage of times a node can solely use the (low latency and low overhead) JCDS to select between all available sources without needing to utilise any plug-ins to gain extra information. Collectively, these three metrics represent the effectiveness of the JCDS at achieving its primary aims.

6.3.2 Performance

The performance of the Discovery Framework is highly dependent on the request characteristics of the application operating above it. For instance, an application that accesses content uniformly from a huge content set (e.g. $> 10^7$) is likely to get inferior performance when compared to an application with only a small content set. This is because the Discovery Framework depends on the JCDS, which is a cooperative discovery system that depends on contributions made by users. It is therefore impossible to perform a thorough evaluation based on a single application. Instead, this section explores performance by identifying and inspecting a number of key parameters that collectively define an application's *content profile*. This therefore makes the evaluation extensible to a large range of applications and environments.

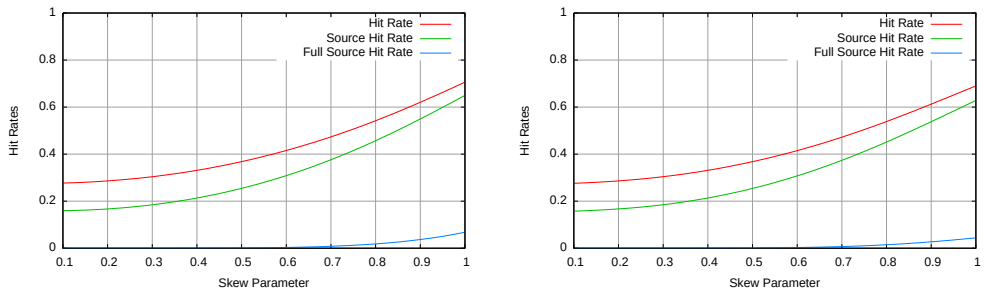
Throughout the evaluation, it is assumed that all providers remain ignorant of the JCDS and refuse to upload any information about themselves. As such, the evaluation solely deals with the consumer-side indexing process, thereby focussing on the deployability aspects of the service (i.e. allowing the system to be deployed without explicit cooperation from providers). This is because when providers choose to actively upload their information to the JCDS, the various hit rates increase to 100%; instead, it is far more interesting to investigate the worse and average case scenarios. Regardless of this, however, it is always possible for a node to query plug-ins to discover content if the JCDS fails.

To provide a representative starting point, the simulator is configured with values taken from a Kazaa measurement study [72], as detailed (later) in Table 6.5. Collectively, these values represent an application's content profile. As highlighted in [37], an advantageous property of simulation is the ability to gain insight into a system through the modification of various inputs whilst monitoring the resulting outputs. In-line with this, each parameter is selected and modified from this set to inspect how it affects the performance of the system. First, the content popularity distribution is looked at (p), then the number of objects (n), followed by the request rate (λ). This approach therefore allows any third party developer to gain an understanding of how an application with a given content profile would perform when utilising Juno's Discovery Framework.

Popularity. The first parameter investigated is the popularity of files (p); this is modelled using a Zipf [151] probability function where $p(i) > p(i+1)$. Variations of this function also exist such as the fetch-at-most-once model proposed by Gummadi et. al. [72]. This follows a Zipf trend but limits each client to requesting a given item of content only once, thereby modelling the presence of persistent caching on each peer (i.e. by the Content Manager).

To investigate the importance of p, Figure 6.2a shows the various hit rates whilst varying the α parameter of the Zipf distribution. Figure 6.2b also shows the

hit rates whilst varying the α parameter of the fetch-at-most-once distribution. The α parameter dictates the skew of the content popularity with a larger value indicating that the users probabilistically select from a smaller range of items.



(a) Zipf (b) Fetch-at-most-once

Figure 6.2: Performance of JCDS with Different Popularity Distributions and α Parameters

For both popularity distributions, it is immediately evident that the performance of the JCDS is unacceptably low when operating with non-skewed content demand (e.g. $\alpha < 0.4$). However, this increases dramatically as the skew increases. A higher skew results in fewer items of content being probabilistically selected from. This effectively decreases the number of items in the content set, making it easier and faster to index them. Both popularity distributions get a fairly similar hit rate, with a skew of 1 reaching 70% for Zipf, and 69% for fetch-at-most-once (hit rate).

Different applications can have vastly different popularity distributions. Simple software update systems are likely to have highly skewed demand as recent updates will be downloaded many times. In contrast, large-scale content archive services may have far less skew. Fortunately, most content systems have been found to follow a Zipf-like distribution meaning that most applications utilising the JCDS will similarly have quite skewed demand. Studies of VoD services, file sharing applications, video archives and web access all show a Zipf distribution. Table 6.3 provides an overview of the Zipf parameters observed within a number of important content applications. Interestingly, recent work has also proposed alternative probability models for looking at content popularity such as stretched exponential [74] and log-normal [41] distributions. All studies, however agree that content systems show a highly skewed demand. Consequently, the Discovery Framework would offer a high performance for most applications.

Number of Objects. The second parameter investigated is the number of objects (n); this is closely related to p as they both impact the perceived popu-

Application	Zipf α
Video on Demand	0.801 [147], 0.56 [67]
Streaming	0.47 [45]
File Sharing	1 [72], 0.62 [127]
Web Access	0.98 [52], 0.719, 0.609 [143]

Table 6.3: Overview of Skew in Zipf-like Content Applications

larity. For instance, even a highly unpopular item from a set of 10 is likely to be accessed at least once (and therefore indexed on the JCDS). To study the effect of different content set sizes, Figure 6.3 shows the performance of the JCDS when operating with various n values. Note that the x-axis is in a log scale.

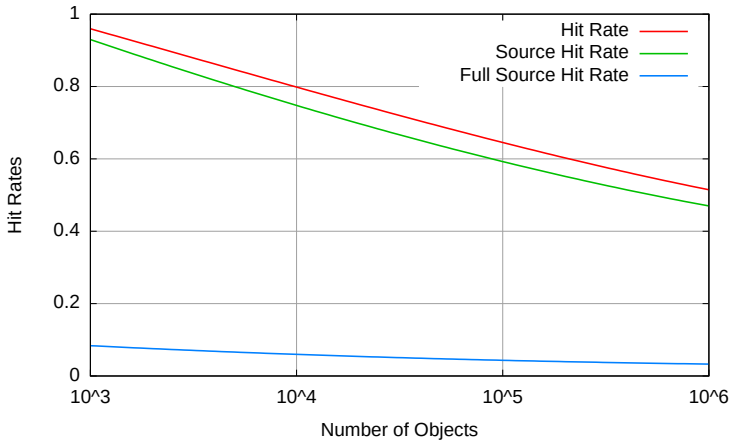


Figure 6.3: Performance of JCDS with different n Values

It can be observed that as the number of objects increase, the various hit rates decrease. This occurs because the popularity distribution (defined by p) becomes spread over a far greater number of objects. This makes it less likely that two nodes will request the same item of content. Intuitively, it can also be seen that a larger content set will take longer to index as even in the best case scenario it will take at least n requests before every item can be accessed and indexed. Unfortunately, however, this best-case is rarely achieveable as less popular items must wait far longer before a request for them is generated.

Different applications can have vastly different content sets; archive systems such as YouTube [32] possess huge content sets whilst applications utilising Juno for software updates may have sets below 10. Applications with small sets of content (i.e. < 1000) will get an extremely high performance from the JCDS, especially if their demand trends are highly skewed. Unfortunately, however, content sizes of a similar level to YouTube (i.e. $> 10^8$) are likely to gain low

performance levels, even with high levels of skew. In such circumstances, it is therefore necessary for providers to take a more proactive role in indexing their content by uploading references themselves. Despite this, even with one million items, a hit rate of 51% is achieved indicating that a noteable proportion of content can still be indexed.

Request Rate. The third parameter is the request rate (λ); this similarly has a significant impact on performance because applications that generate more requests are likely to access more non-indexed items and subsequently assist in the indexing process. This will occur if either (*i*) each application instance generates many requests, or (*ii*) there are many deployed instances of the application. It has previously been stated that a content set that is fully indexed must have received at least n requests. Consequently, the lower bound of this time period is defined by $\frac{n}{\lambda}$ making the request rate a highly important parameter.

To study this, Figure 6.4 shows the effect that per-node request rates have on the JCDS. It can be seen that as the number of requests increase, so does the hit rate. Depending on the application's content popularity distribution, this can result in different outcomes. If the popularity is highly skewed (as in Figure 6.4), the performance will be largely unchanged as the same items (that are already indexed) will be frequently re-requested. However, if the popularity distribution is more uniform, then significant gains will be made as there is a higher probability that a non-indexed item will be accessed. Regardless of this, a high request rate is often required to maintain high source hit rates in systems that suffer from churn (i.e. peer-to-peer providers). This is because out-of-date sources need to be refreshed periodically; in contrast, more persistent sources (e.g. HTTP servers) can remain indexed because they do not suffer from churn.

It can be seen that the performance gains are most significant as the request rate increases from 0 to 20 requests per minute. Following this, the gradient shallows, as the consumers simply re-request the same items of content. Evidently, applications with low request rates will benefit dramatically from small increases whilst applications with already high request rates will gain little.

Different applications will have vastly different request patterns depending on the type of content being distributed. An application playing music might generate requests every few minutes, whilst a software distribution service might have request intervals in the order of weeks. Arguably, however, the more significant contributing factor is the number of deployed instances of the application. For instance, even an application with a low request rate, such as 1/month, will have a huge request frequency with 10 million application instances (≈231/min). Such request rates can easily be handled by the Juno middleware, as shown in Appendix A.

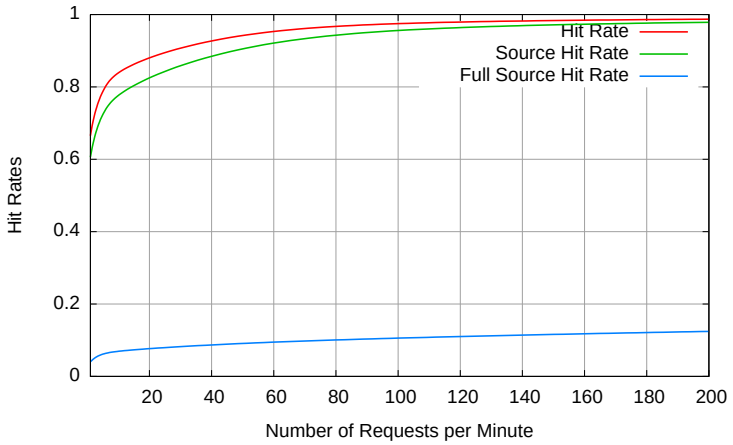


Figure 6.4: Performance of JCDS with Different λ Values

6.3.3 Overhead

The previous section has explored the performance of the Juno Content Discovery Service (JCDS). Alongside this, it is also important to understand the overhead of using the Discovery Framework's design philosophy. In the worst-case, the Discovery Framework uses a full set of plug-ins for every request. To mitigate this, however, the JCDS also allows certain plug-ins to be detached once they have been sufficiently indexed.

This section explores the overhead of utilising multiple plug-ins and the subsequent benefits that can be gained by dynamically detaching them. It does not, however, look at the overhead of running the JCDS as, in essence, this is simply a lookup system (e.g. a DHT). As such, this is redirected towards evaluations of various prominent lookup systems [61][106]. Details about the memory and processing overheads related to the Discovery Framework can also be found in Appendix A.

The overhead of the Discovery Framework is based on two factors. First, the plug-ins that are attached and, second, the content profile of the application. This is because collectively these define the Discovery Framework's ability to detach the plug-ins (to be replaced by the JCDS). Therefore, when using a fixed set of plug-ins, the overhead will be constant; however, when also using the JCDS it will vary over time. This variance is based on a number of key parameters, detailed in Table 6.4.

To evaluate the overhead costs of the Discovery Framework, these parameters are altered in a similar way to the previous section. The simulator is again configured with the Kaz parameter set and the results presented are those from the perspective of a single monitor node.

Parameter	Default	Description
Detachment Threshold	0.75	The minimum fraction of results from the JCDS that must match plug-in i before it is detached
Request Rate	2/hour	The average interval between a given node generating queries through the Discovery Framework
Churn	[132]	The probability that a reference on the JCDS will be stale
Sim Length	14 days	The length of time cumulative results are accumulated over

Table 6.4: Overview of Primary Parameters Relating to Discovery Framework Overhead

Detachment Threshold. The first parameter is the detachment threshold; this dictates the required level of similarity between the JCDS's and a plug-in's results before it can be detached. A value of 1 indicates that the JCDS and a given plug-in must generate identical results before it can be detached. Clearly, this parameter must be low enough to ensure that plug-ins are removed to reduce overhead, yet high enough to ensure that plug-ins are not removed before their information is indexed on the JCDS.

Figure 6.5 shows the Discovery Framework's cumulative request overhead after 14 days based on a range of thresholds (note that this does not include background overhead). It can be seen that lower thresholds result in lower overhead. This is because a low threshold results in plug-ins being detached faster. For instance, a threshold of 0.5 indicates that only 50% of results generated by a plug-in must also be generated by the JCDS. As the threshold increases, it becomes less likely that the JCDS will provide results of a sufficient quality to allow the plug-in to be detached. In such circumstances, the plug-ins remain attached to the Discovery Framework and overhead continues to be high.

Although lower thresholds result in lower overhead, it is evident that an unwise decision regarding detachment can lead to inferior performance. If a plug-in is detached without its information being indexed on the JCDS, it is likely that the node will remain ignorant of many sources that could have been discovered. This can be measured by comparing the JCDS results against the results of each plug-in after they have been detached. The optimal case would be that the two result sets are identical for the rest of the node's lifetime; this can be calculated by,

$$missrate = \frac{misses}{numrequests} \tag{6.4}$$

where $misses$ is the number of requests that could have returned results from a detached plug-in, which are not indexed on the JCDS. It is therefore the fraction

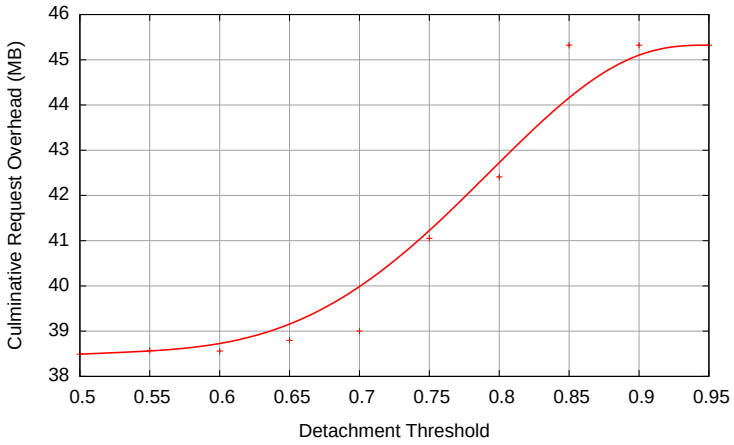


Figure 6.5: Overhead of Discovery Framework with Different Detachment Thresholds

of requests that did not get access to all sources due to the removal of one or more plug-ins.

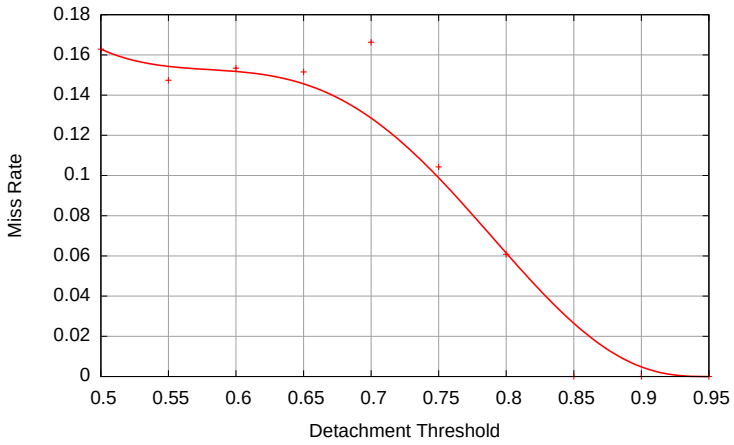


Figure 6.6: Miss Rate of JCDS Based on Detachment Threshold

Figure 6.6 shows the miss rate for various threshold values. Evidently, the miss rate decreases with higher values set for the detachment threshold. This is because with low thresholds, it becomes impossible to make an informed decision about the removal of a plug-in. Figures 6.5 and 6.6 can therefore be contrasted to see that there is an obvious trade-off: detaching plug-ins reduces overhead but also reduces the accuracy of results. The stipulation of the detachment threshold is therefore an application/user specific decision based on the needs of the system, as well as any resource constraints. In environments with sufficient resources (e.g. a PC with a DSL connection) it is recommendable to set the detachment thresh-

old to > 0.9 to ensure that only those plug-ins that have been extremely well indexed are detached. However, in low capacity environments this threshold can be progressively lowered (automatically) until an acceptable overhead is reached.

Request Rate. The next important parameter is the local request rate. This can be modelled by looking at the average interval between requests generated by a given client. If all plug-ins are attached, approximately 46 KB of traffic is generated per request; assuming the 2 request per/day model of [72] this would only result in 92 KB of traffic a day. This is, of course, acceptable although this will increase linearly with more intensive request patterns. Generally, of more concern is the background traffic as this remains regardless of an application's request pattern. Certain plug-ins generate no background overhead, however, others generate large amounts of overhead even when not being queried (i.e. peer-to-peer systems). The attachment of all plug-ins results in approximately 730 KB per minute (12 KB per second). On a typical 5.5 Mbps connection this only constitutes 1.7% of capacity; however, it is still undesirable.

As well as directly impacting overhead in this manner, the request rate is also important because it largely defines the length of time it takes a node to profile a plug-in. This is required so that a node can decide whether or not a plug-in should be removed. A higher local request rate subsequently allows this to be calculated sooner. For instance, when generating one request every 30 minutes, it takes 76 hours to remove the first plug-in; in contrast, it only takes 25 hours when generating one request a minute. A very important aspect to note is that this time also includes the period that it takes the JCDS to passively index the content in that plug-in. Therefore, even if a node generates 100 requests per minute, it will not be detached unless the JCDS has sufficiently indexed that plug-in's information. An increased local request rate therefore only allows a node to make a decision faster; it does not affect the outcome of that decision. As can be seen from the Section 6.3.2, this is, instead, defined by the three key global parameters: p, λ and n.

Figure 6.7 shows the overhead incurred based on various request patterns over 14 days. Note that the results are significantly higher than those shown in Figure 6.5 due to the inclusion of background traffic. However, it is clear that nodes with higher request rates get lower overhead. At first this seems counter-intuitive, however, this occurs because such nodes are capable of profiling their plug-ins faster. This subsequently allows such nodes to detach their plug-ins sooner, thereby reducing overhead. This is particularly beneficial when utilising plug-ins with a high background overhead. This is because in such situations, overhead would increase linerally with time rather than with the number of requests. Consequently, the overhead of generating a single request becomes proprotionally lower when using such plug-ins. A good example of this is Gnutella,

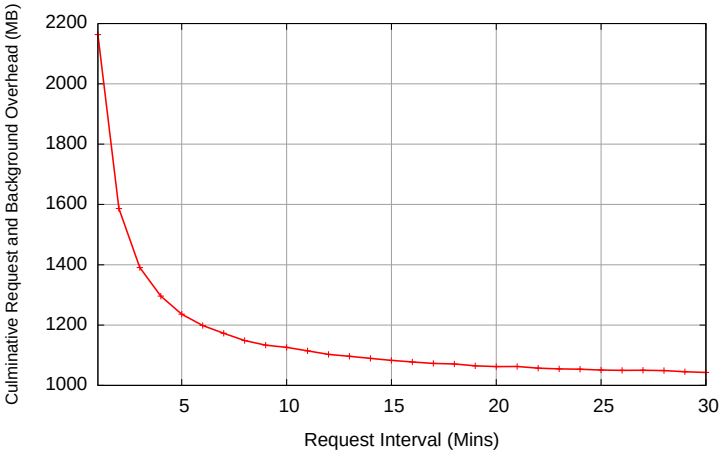


Figure 6.7: Cumulative Background and Request Overhead Based on Local Request Interval

which generates 6 KB of background traffic per minute and 20 KB of traffic per request. An hour of idleness therefore still creates the equivalent of 18 requests. This can be contrasted with the server lookup plug-in which has no background overhead; proportionately a further request therefore has a greater impact.

Churn. The third vital parameter affecting bandwidth overhead is that of churn. This is not because of maintenance traffic but, instead, due to the effect that it has on the JCDS's ability to maintain up-to-date information. Plug-ins can offer real-time search information for their respective discovery systems. The JCDS, however, is restricted to results that have been uploaded previously. If the intervals between requests (in the global system) are longer than the intervals between providers going offline, then content indexed on the JCDS will be consistently stale. This does not create an issue in a low churn system (e.g. a web server) but dramatically increases overhead in a high churn system because it becomes impossible to detach the plug-ins without creating high miss rates.

Figure 6.8 shows the overhead for two nodes over 14 days using different plug-ins: one with all churn-susceptible plug-ins and one without. It is evident that the overhead is higher when operating with plug-ins that are susceptible. This is because a node that is utilising plug-ins without any churn can generally detach them all and exclusively use the JCDS after a relatively short period of time. In contrast, any plug-in that suffers from churn can only be detached if the global request rate (for a given item of content) is higher than the churn rate. This, however, is only feasible for extremely popular content. As such, nodes using many churn-susceptible plug-ins will have a relatively constant overhead.

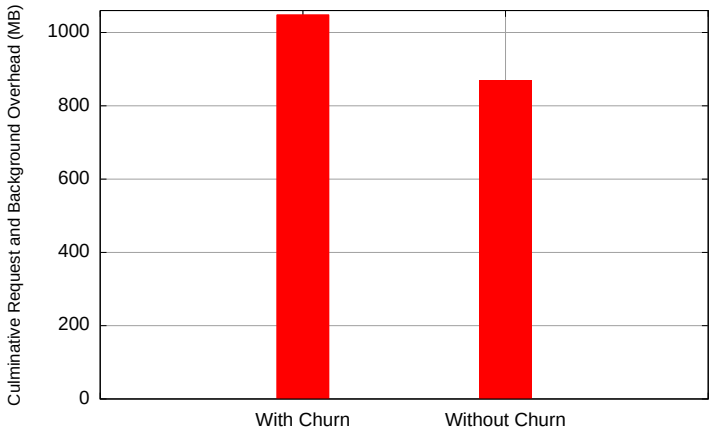


Figure 6.8: Overhead Comparison of Discovery Framework using Plug-ins with and without Churn ($\lambda = 10$)

6.3.4 Case Studies

The previous sections have investigated how the different key parameters identified affect the performance and overhead of the the Discovery Framework. These parameters collectively build up a *content profile* that defines how an application utilises content. These parameters can therefore be varied by developers to predict the performance of their application when using Juno. To further evaluate the JCDS and to concretise the discussion, two example case studies are now inspected based on measurements of previous content applications: a Video on Demand (VoD) system and a file sharing system. This shows how the previous experiments can be extended and specialised to benchmark a given application, as well as highlighting Juno's performance in typical environments. Consequently this helps validates Juno's ability to aid deployment through its ability to interoperate and index third party discovery systems.

The first parameter set, shown in Table 6.5, is taken from [72] based on measurements taken from the Kazaa file sharing application (Kaz set). The second set, shown in Table 6.6, is taken from [147] based on measurements of the China Telecom VoD service (VoD set). These parameter sets offer the ability to gain insight into how the Discovery Framework would behave under realistic request trends. To also simulate representative file sizes, the macroscopic BitTorrent measurement study detailed in Section 3.3 is used; VoD uses the film file size traces whilst Kaz uses the traces for all files.

As previously identified, there are three core parameters that define an application's content profile: p, λ, and n. These are therefore the only parameters taken from these studies, as to extract system parameters (e.g. churn) would taint the results with details of the individual protocols and implementations

Parameter	Default Value
Popularity (p)	Fetch-at-most-once ($\alpha = 1$)
Request rate (λ)	Poisson ($\lambda = 1.39/min$)
# objects (n)	40,000
# Users	1,000
Churn ($churn$)	Stutzbach et. al. [132]
Bandwidth (bw)	Dischinger et. al. [56]
Saturation ($saturation$)	Uniform
File Size ($filesize$)	Trace-Based (All files)

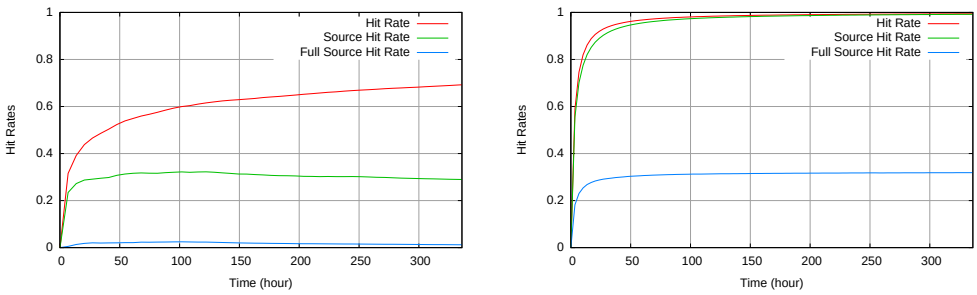
Table 6.5: Overview of Kaz Parameters [72]

Parameter	Default Value
Popularity (p)	Zipf ($\alpha = 0.801$)
Request rate (λ)	Poisson ($\lambda = 108/min$)
# objects (n)	7036
Churn ($churn$)	Stutzbach et. al. [132]
Bandwidth (bw)	Dischinger et. al. [56]
Saturation ($saturation$)	Uniform
File Size ($filesize$)	Trace-Based (Video Files)

Table 6.6: Overview of VoD Parameters [147]

that the measurement studies look at. The rest of this section now explores these parameter sets to see how they would operate in the real-world.

Control Benchmark. Before investigating the importance of the different parameters, it is first necessary to inspect the standard performance achieved by the two case studies using their default parameters. Figure 6.9a shows the three performance metrics for the Kaz parameter set, whilst Figure 6.9b shows the three performance metrics for the VoD results. For the Kaz set, after 14 days, the hit rate is 69%, with 64% of requests gaining at least one valid source from the JCDS. In contrast, the VoD set shows a far higher performance level for all metrics. After 14 days, both the *hitrate* and the *sourcerate* is at 99%. The $fullsourcerate$ is also different between the two systems; Kaz has a very low value (3%) whilst the VoD values is over 20%. Clearly, the characteristics of these two applications mean that they achieve vastly different performance levels, however, whether or not these metrics are acceptable is a decision left to the application developer. The reason for these deviations can be found in the values of the important parameters. The rest of this section now investigates and contrasts their results.



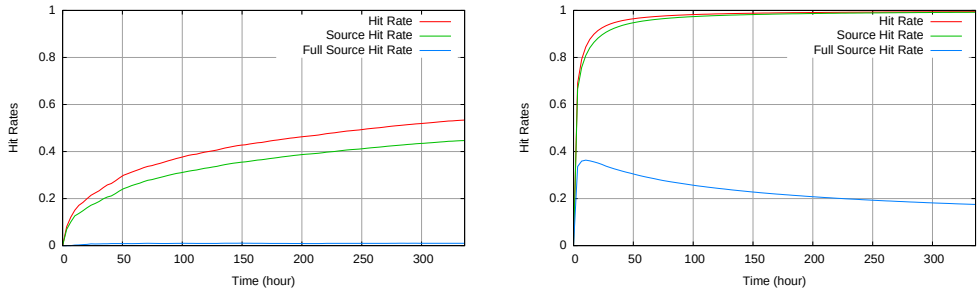
(a) Kaz Parameter Set (Default) (b) VoD Parameter Set (Default)

Figure 6.9: Performance of Case Studies with Default Parameters

Popularity. The first important parameter is p. Interestingly, the VoD parameter set has a less skewed popularity than Kaz, despite its higher performance. Importantly, however, Kaz uses a fetch-at-most-once popularity function, which means that the popularity of a given item of content is limited by the number of peers in the system. In contrast, VoD uses a pure Zipf function, which means that a given client can request an item of content multiple times, thereby improving hit rates. A prominent example of this is the Google logo which will likely be re-requested many times in a single day by each user.

To look at the impact of popularity, Figure 6.10 shows the performance results again whilst swapping the popularity parameters of the two sets, i.e. VoD's p becomes fetch-at-most-once ($\alpha = 1$) and Kaz's p becomes Zipf ($\alpha = 0.801$). It can be observed that the hit rate of VoD stays constant whilst the hit rate of Kaz decreases by 15% to 54%. From this it can be concluded that p is not a dominant factor in the VoD parameter set but is highly important in the Kaz set. In both cases, the most noticeable change is that of the $fullsourcerate$, which drops significantly. For Kaz, this involves achieving an extremely low rate; this is because, the Zipf α parameter is lowered from 1 to 0.801, which has a significant effect when dealing with such large content sets. In contrast, the VoD set achieves an initially higher $fullsourcerate$ due to the more skewed nature of its demand ($\alpha = 1$); however, this tails off due to the fetch-at-most-once distribution. This means items cannot be re-requested by individual users making it easy for churn to make the source entries stale.

Number of Objects. The next parameter is n, which refers to the number of items of content used by the application. Kaz has a large content set (40,000) whilst VoD has a much smaller number of items (7036). Depending on the nature of the application, these values will vary dramatically. Once again, to inspect the importance of these parameters, their values are swapped between

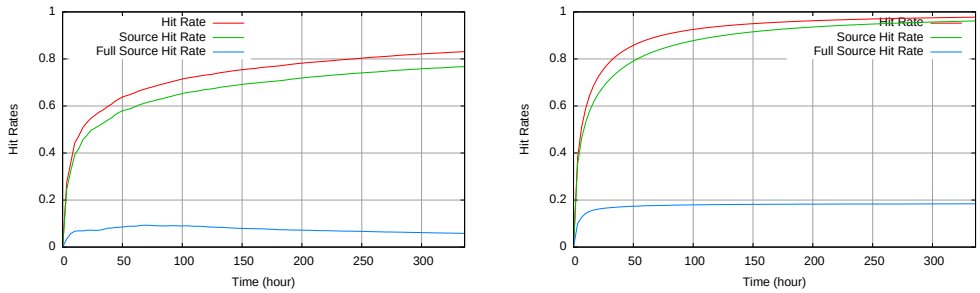


(a) Kaz Parameter Set p = Zipf ($\alpha = 0.801$ (b) VoD Parameter Set p = fetch-at-most-once ($\alpha = 1$)

Figure 6.10: Performance of Case Studies with Changed p Values

the two parameter sets so that Kaz's n is 7036 and VoD's n is 40,000.

Figure 6.11 shows the results. It can be seen that the swap has a positive effect on the Kaz set for all performance metrics, with a *hitrate* increase of 14%. This is obviously because the popularity distribution is over a smaller content set. This reduces the number of popular content items, thereby increasing the probability of a hit. In contrast, unsurprisingly, the VoD results have a hit rate that increases with a far more shallow gradient than previously. This is because the size of the content set takes longer to index. Despite this, after 14 days, the hit rate has reached 97% - only 2% less than when operating with an n of 7036. As such, it is evident that an increase in the number of objects can be managed as long as the request rate is high enough to ensure that they can be indexed.



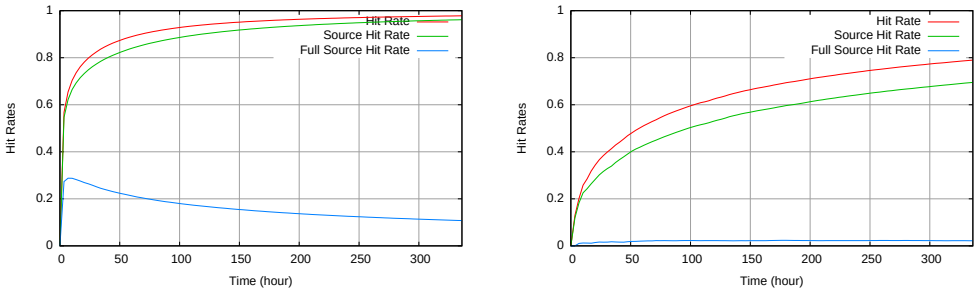
(a) Kaz Parameter Set $n = 7036$ (b) VoD Parameter Set $n = 40{,}000$

Figure 6.11: Performance of Case Studies with Changed n Values

Request Rate. The last parameter is λ, which dictates the frequency of requests. The VoD parameter set has a far higher request rate than Kaz. The

file sharing application only has, on average, 1.39 requests per minute (2 requests from each user a day); in contrast, the video on demand system receives an average of 108 requests per minute.

Figure 6.12 shows the results of swapping the λ parameter between the two parameter sets. It can be seen that the variance has a significant effect on both systems. Kaz now gains an extremely high hit rate (98%) with a similarly high source rate (96%); this can be compared to the default results of 69% and 64%, respectively. The VoD results, in contrast, achieve far lower performance than in the default configuration. The hit rate drops by 20% to 79% whilst the source rate drops to 70%.



(a) Kaz Parameter Set $\lambda = 108$ per min (b) VoD Parameter Set $\lambda = 1.39$ per min

Figure 6.12: Performance of Case Studies with Changed λ Values

It is evident from these results that the request rate plays a significant role in defining the effectiveness of the JCDS, specifically in these case-studies. When swapping the λ value, the two case studies exchange their performance levels almost exactly. This means that in this particular circumstance, the request rate is the dominant factor. Closer inspection shows that this is a logical finding as this value varies most dramatically out of the two parameter sets; the request rate of Kaz is only 1.2% of the request rate of VoD. In contrast, the equivalent percentages for α and n are 124% and 17% (note that Kaz's α is greater than VoD's).

6.3.5 Summary

This section has evaluated the Discovery Framework with the intent of answering the following question, *how effectively can Juno discover content in third party systems?*. The ability to access third party systems has been implicitly designed into Juno through the use of plug-ins. Therefore, to answer this question, performance and overhead aspects of the Discovery Framework have been investigated. Specifically, the performance of the Juno Content Discovery Service (JCDS) has

been investigated using detailed simulation studies. Alongside this, the overhead of the JCDS has been inspected, as well as the costs of using multiple plug-ins in parallel. It has been shown that the JCDS can effectively index content in third party systems for retrieval by Juno nodes. Further, the ability of the JCDS to mitigate the use of multiple plug-ins has also been validated. Collectively, this has shown the Discovery Framework to effectively fulfil its function as part of Juno.

6.4 Delivery-Centric Delivery in Juno

The second aspect of Juno to be evaluated is the delivery of content. This process involves selecting and accessing the optimal providers to meet certain delivery-centric requirements issued by the application. In Juno this is managed by the Delivery Framework using a number of (re-)configurable plug-ins, which are exploited to provide access to a range of sources.

This section evaluates the Delivery Framework based on the research goals and design requirements of this thesis. First, the methodology is outlined, detailing the techniques used. Following this, Juno's capabilities are evaluated using a number of key case-studies that represent the most common use-cases. These are exploited to show how the benefits detailed in Chapter 3 can be realised by Juno. After this, a number of further case-studies are also provided to show how Juno could operate under a range of diverse delivery requirements.

6.4.1 Methodology

This section provides an overview of the methodology taken for evaluating Juno's approach to building delivery-centricity in the middleware layer.

Evaluation Overview. The Delivery Framework is the key building block for supporting delivery-centricity in Juno. This is achieved by performing intelligent consumer-side (re-)configuration to enable interoperation with various third parties. Consequently, the Delivery Framework has been explicitly designed to support a delivery-centric abstraction that is instantly deployable alongside existing systems (in-line with research goals 2 and 3).

Chapter 3 has provided a quantitative analysis of both the need for runtime (re-)configuration, as well as the benefits that can be gained through the process. This is due to the heterogeneity that has been discovered based on both consumer and temporal variance (i.e. the propensity for a provider's ability to satisfy requirements to vary over time and between different consumers). Consequently, the design of the Delivery Framework has been shaped by these findings to offer explicit support for delivery (re-)configuration. This means the primary evaluation question that must now be asked regarding the Delivery Framework is,

can Juno's design feasibly exploit this heterogeneity to enable the measured benefits? Evidently, this must be validated to ensure that Juno's design does, indeed, fulfil the research goals outlined in Chapter 1 by supporting a delivery-centric abstraction.

The main challenge is therefore defining an evaluation that can validate this thoroughly. This question cannot be satisfied using quantitative means, as the ability to offer delivery-centricity in the way defined throughout this thesis is simply a binary metric (yes or no). Consequently, the evaluation presented must operate in conjunction with the results presented in Chapter 3. One possible option would be to deploy a real-world implementation that users could actually test. This, however, is unfeasible as it would be necessary for a large range of application to be developed and adopted. Further, by operating outside of a controlled environment, it would become difficult to monitor behaviour. An alternative approach would be to use simulations; this, however, has already been accomplished in Chapter 3. Further, this would not allow the prototype of Juno to be investigated, thereby preventing a practical validation. To address these limitation, a number of key case-study experiments are designed and deployed using the Emulab network testbed instead. The justification for this choice has been explored in detail in Section 6.2.2. Emulab [9] is a testbed consisting of a number of hosts that can be configured to possess various network characteristics. This allows the Juno prototype to be deployed in a controlled and realistic setting, alongside real-world protocol implementations. This approach can therefore be used to validate that Juno can, indeed, be (re-)configured to interoperate with third party systems based on various delivery requirements.

Case-Studies. A case-study is a well-defined environmental context that hosts a particular system attempting to a perform a given function. The principle purpose of this is to explore the behaviour of a use-case to understand how it would operate in similar real-world situations. The key requirements of a case-study are therefore that it is representative and extensible. This means that it must explore a realistic scenario and that its results must be applicable to a range of situations that follow that given use-case.

These requirements have been closely followed when designing the case-studies used to evaluate Juno's delivery-centric support. They capture the primary examples of its operation, namely configuration, re-configuration and distributed re-configuration. These three use-cases represent the key modes of Juno's operation, starting with the most dominant; Table 6.7 provides a brief summary.

Within the following sections, each case-study is first defined, alongside the relevant motivation. After this, an analysis of the case-study is given, including both performance and overhead metrics. Finally, the findings of the case-study are summarised to show the outcomes as well as how they can be applied to

Case-Study	Summary
Case-Study 1: Configuration	This explores the process of configuring a delivery based on certain requirements. It focusses on optimising the delivery when faced with a range of consumer variance.
Case-Study 2: Re-Configuration	This explores the process of re-configuring a delivery to react to variations in the environment. It focusses on addressing temporal variance from the perspective of a single consumer.
Case-Study 3: Distributed Re-Configuration	This explores the process of re-configuring multiple nodes cooperatively. It focusses on handling both consumer and temporal variance in a way that ensures each node can individually address changes in its environment.

Table 6.7: Summary of Delivery-Centric Case-Studies

alternate scenarios.

Emulab Testbed. The evaluative case-studies detailed within this section have been deployed and measured on the Emulab testbed [144]. Emulab consists of almost 400 interconnected PCs that can be configured to possess a range of network characteristics (delay, bandwidth etc.). Any number of these nodes can be reserved for dedicated usage, allowing an emulated network (or internetwork) to be constructed with the chosen software running. In the case studies, collections of node are used to act as content providers (e.g. BitTorrent swarms, web servers etc.), whilst one or more nodes are used to act as consumers. Consumers host the Juno middleware with a simple content application operating above; the middleware then utilises (re-)configuration to achieve delivery-centricity based on the available providers. Finally, a single node also operates as a JCDS server to resolve content queries on behalf of the consumers.

6.4.2 Case-Study 1: Consumer-Side Configuration

The first evaluative case-study investigates Juno's ability to handle consumer variance. As identified in Chapter 3, this refers to the variance in a provider's ability to fulfil requirements when observed from the perspective of different consumers. The purpose of this case-study is therefore to validate the feasibility of addressing this need through Juno's approach to dynamic delivery configuration.

Case-Study Overview

The primary use-case of Juno is the situation in which multiple delivery systems are discovered to offer a desired item of content. In such a circumstance, Juno executes selection algorithms to dynamically choose the best one before dynamically configuring itself to interoperate with it. To address consumer vari-

ance, this is performed individually by each node. The process can be broken down into two steps: (*i*) selection and (*ii*) configuration. The selection process is performed by comparing selection predicates to meta-data associated with each possible provider/protocol. In its simplest form, this involves stipulating some static boolean property (e.g. ENCRYPTED==true), however, this can also be extended to involve more complicated co-dependent selection predicates. Following a decision, the configuration process is performed by dynamically loading the necessary functionality (within a component configuration), interconnecting all the components and then initiating the download.

The first case-study explores this key situation, in which multiple sources are found by multiple nodes that observe consumer variance. In-line with previous examples in the thesis, it is assumed that the application's primary requirement is to satisfy some performance need, whilst operating within certain constraints. Within this case-study, an application (operating on multiple nodes) wishes to consecutively download two items of content: a 4.2 MB music file and a 72 MB video file. To explore this, the case-study has been implemented in Emulab using two nodes of different capacities. In the first experiment a low capacity node, *Node LC*, is operating over a typical asynchronous DSL connection with 1.5 Mbps download capacity alongside 784 Kbps upload capacity. In the second experiment another node, *Node HC*, operates over a much faster 100 Mbps synchronous connection. A Juno client application operates on both nodes, first requesting the download of the 72 MB video followed by the 4.2 MB music file. This therefore introduces two variable factors: content size and consumer capacity.

A number of content providers are also set up within the testbed. After the discovery process, the content is found to be available on three providers by Node LC, whilst four are found by Node HC, as listed in Table 6.8. The three common delivery providers are a web server, a BitTorrent swarm and a set of Limewire peers. Node HC further discovers a replication server offered as a service on its local network. Both nodes therefore generate their necessary meta-data for each of these sources using the techniques detailed in Chapter 5; as previously discussed, these are low overhead mechanisms that can be completed in under a second.

Clearly, when the application generates the content requests, it associates them with a set of selection predicates. Table 6.9 details the selection predicates issued by Nodes LC and HC for the 72 MB delivery*. The first predicate is the estimated download rate; in this case, the application simply requires the content as quickly as possible. Node LC also has limited resources (only 784 Kbps upload capacity) and therefore the Content-Centric Framework introduces the upload predicate, which stipulates the delivery scheme should not consume upload resources. In contrast, Node HC has no such limitations and consequently

*Similar predicates are used for the 4.2 MB delivery

Available for	Delivery Scheme
Node LC, Node HC	*HTTP:* There is 2 Mbps available capacity for the download to take place. The server is 10ms away from the client.
Node LC, Node HC	*BitTorrent:* A swarm sharing the desired file. The swarm consists of 24 nodes (9 seeds, 15 leeches). The upload/download bandwidth available at each node is distributed using a real world measurements taken from existing BitTorrent studies [3].
Node LC, Node HC	*Limewire:* A set of nodes possessing entire copies of the content. Four nodes possessing 1 Mbps upload connections are available.
Node HC	*Replication Server:* A private replication server hosting an instance of the content on Node HC's local area network. The server has 100 Mbps connectivity to its LAN and is located under 1ms away. The server provides data through HTTP to 200 clients.

Table 6.8: Overview of Available Delivery Schemes

Meta-Tag	Selection Predicates	
	Node LC	Node HC
DOWNLOAD_RATE	HIGHEST	HIGHEST
REQUIRES_UPLOAD	FALSE	N/A
MIN_FILE_SIZE	<=72 MB	<=72 MB
MAX_FILE_SIZE	>=72 MB	>=72 MB

Table 6.9: Selections Predicates and Meta-Data for Delivering a 72 MB File

Meta-Tag	HTTP	BitTorrent	Limewire
DOWNLOAD_RATE	1.4 Mbps	480 Kbps	970 Kbps
REQUIRES_UPLOAD	FALSE	TRUE	TRUE
MIN_FILE_SIZE	0 MB	8 MB	0 MB
MAX_FILE_SIZE	∞ MB	∞ MB	∞ MB

Table 6.10: Delivery Scheme Meta-Data for Node LC

does not use this selection predicate. The final two predicates define the file size; these indicate the minimum and maximum file sizes that the plug-in should be suitable for. Tables 6.10 and 6.11 provide the values of the relevant meta-data exported by the providers for each node.

The rest of this section now explores this case-study to understand how Juno fulfils these requirements, alongside the benefits gained in this particular setting. Clearly, the results presented are extensible to any environment in which multiple sources are discovered, as quantified in Chapter 3.

Meta-Tag	HTTP	BitTorrent	Limewire	Rep Server
DOWNLOAD_RATE	1.9 Mbps	2 Mbps	3.6 Mbps	41 Mbps
MIN_FILE_SIZE	0 MB	8 MB	0 MB	0 MB
MAX_FILE_SIZE	∞ MB	∞ MB	∞ MB	∞ MB

Table 6.11: Delivery Scheme Meta-Data for Node HC

Analysis of Case-Study

The above case-study has been setup in Emulab; Figures 6.13 and 6.14 show measurements taken from both Nodes LC and HC as they were downloading the two files. It shows the average application layer throughput for the large and small file downloads when utilising each provider. It also shows the throughput of Juno, which is capable of selecting any plug-in.

It is first noticeable that the results between Node LC and HC are disjoint, i.e. the results for Node LC are in direct opposition to Node HC. This means that an application optimised for Node LC would be suboptimal for Node HC, and vice-versa. Consequently, a statically configured application would not be able to fulfil the delivery requirements for both nodes simultaneously. This therefore confirms the presence of consumer variance. Thus, an application would need to implement control logic to follow different optimisation paths depending on the host. In contrast, Juno manages this process on behalf of the application.

The reasons for these disjoint results between the two nodes can be attributed to three key sub-categories of consumer variance. First, the two nodes have access to different providers; second, the consumers possess different characteristics; and third, the two items of content requested have different properties (i.e. size). Consequently, different combinations of the above factors can drastically alter a node's ability to satisfy given performance requirements. Each of these forms of consumer variance are now addressed in turn.

The first and most obvious consumer variance is that of *provider availability*. This refers to the differences in content availability when observed from the perspective of different consumers. For instance, in the case-study, Node HC operates in a network that offers a replication service with very strong connectivity. In contrast, Node LC does not have any such service available because it is limited to members of a particular network (or often paid members). Variations of this can happen in a range of different situations; Gnutella, for example, will allow different sources to be discovered based on a node's location in the topology. Any content-centric system should therefore be able to exploit this consumer variance to ensure that the delivery requirements are best fulfilled and Node HC gains the content from its local replication server. Evidently, Juno supports this through allowing each node to individually select the access mechanism that best fulfils its requirements.

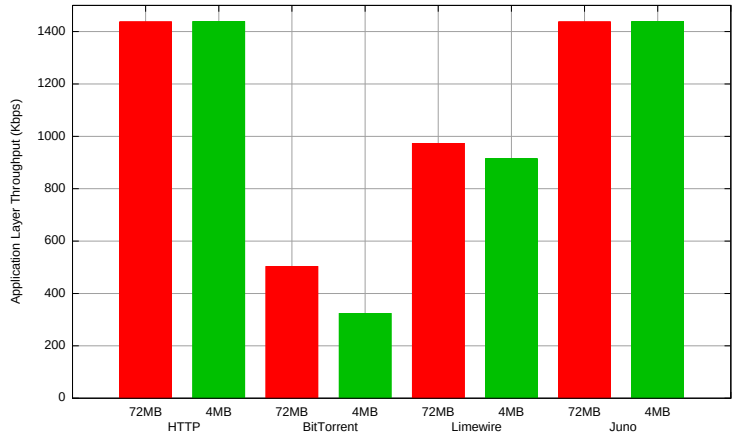


Figure 6.13: Average Throughput of Deliveries for ADSL Connection (Node LC)

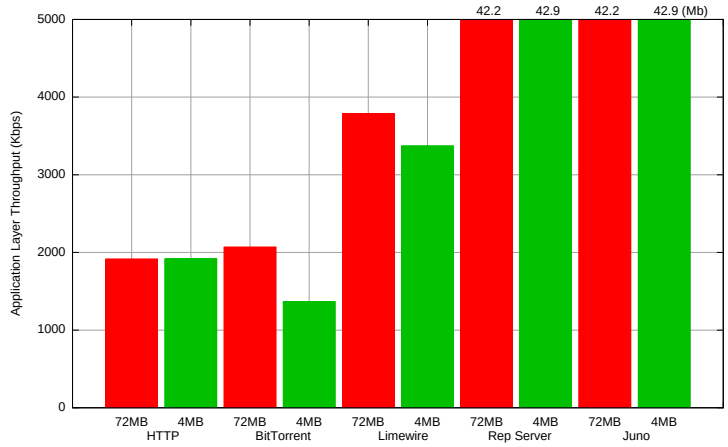


Figure 6.14: Average Throughput of Deliveries for 100 Mbps Connection (Node HC)

The second type of divergence is caused by differences in *consumer characteristics*. This variance is best exemplified by the observation that, for the 72 MB delivery, HTTP is the optimal plug-in for Node LC but the most suboptimal plug-in for Node HC. This is because Node HC can better exploit the resources of the peer-to-peer alternatives (i.e. BitTorrent or Limewire), whilst Node LC fails to adequately compete. In essence, Node LC is best suited to utilising the least complicated method of delivery because the more complicated approaches simply increase overhead without the ability to contribute real performance gains. Once again, this form of consumer variance is effectively addressed by Juno, which configures itself to satisfy requirements on a per-node basis.

The final type of divergence is caused by differences in the *content* being accessed. The previous two paragraphs have shown that in this case-study it

is impossible to fulfil the delivery requirements for divergent consumers without performing per-node configuration. However, a further important observation can also be made: the delivery mechanism considered optimal for one item of content is not always the best choice for a different item of content. This is best exemplified by the observation that the optimal delivery system for accessing the 72 MB file is not necessarily the best for accessing the 4.2 MB file. For instance, when operating on Node HC, BitTorrent is faster than HTTP for the 72 MB file but slower than HTTP for the 4.2 MB file. In fact, the 4.2MB delivery achieves only 66% of the throughput measured by the 72MB delivery using BitTorrent. This is due to the complexity and length of time associated with joining a peer-to-peer swarm. Consequently, optimality does not only vary between different nodes but also between different individual content downloads. An application using BitTorrent that cannot re-configure its delivery protocol would therefore observe significant performance degradation between the two downloads. Consequently delivery system selection must not only occur on a per-node basis but also on a per-request basis. Juno addresses this by seamlessly re-configuring between the different optimal delivery plug-ins, thereby effectively addressing this problem whilst removing the development burden from the application. This divergence therefore highlights the fine-grained complexity that can be observed when handling real-world content distribution. This complexity makes it difficult for an application to address all possible needs and therefore provides strong motivation for pushing this functionality down into the middleware layer.

The above analysis can now be used to study the behaviour of Juno during this case-study. Table 6.12 details the decision process undertaken, showing which plug-ins are selected for each content request. In this situation, for both items of content, Node LC selects HTTP (due to the high download rate and no need for upload resources), whilst Node HC selects the replication server (due to the high download rate). In terms of fulfilling performance requirements, this therefore allow a quantification of the supoptimality of not using Juno's philosophy of delivery (re-)configuration. Table 6.13 provides the percentage increase in throughput when using Juno during these experiments. The worst case scenario compares Juno against an application that has made the worst possible design-time decision (using the above figures). The best case is when the application has made the best decision (obviously resulting in the same performance as Juno in this situation). Evidently, these results highlight Juno's ability to effectively improve performance based on delivery requirements provided by the application.

Summary of Findings

In summary, this experiment has (*i*) confirmed that the consumer variance identified in Chapter 3 is prevalent, and (*ii*) shown it can be effectively handled using Juno's approach to delivery configuration. Importantly, by placing content-

Meta-Tag	Predicates		Valid Configurations	
	Node LC	Node HC	Node LC	Node HC
72 MB Content Request				
DOWNLOAD_RATE	HIGHEST	HIGHEST	**HTTP**	**Rep Server**
REQUIRES_UPLOAD	FALSE	N/A	**HTTP**	N/A
MIN_FILE_SIZE	<=72 MB	<=72 MB	Any	Any
MAX_FILE_SIZE	>=72 MB	>=72 MB	Any	Any
4.2 MB Content Request				
DOWNLOAD_RATE	HIGHEST	HIGHEST	**HTTP**	**Rep Server**
REQUIRES_UPLOAD	FALSE	N/A	**HTTP**	N/A
MIN_FILE_SIZE	<=4.2 MB	<=4.2 MB	**HTTP**, Limewire	HTTP, Limewire, **Rep Server**
MAX_FILE_SIZE	>=4.2 MB	>=4.2 MB	**HTTP**, Limewire	HTTP, Limewire, **Rep Server**

Table 6.12: Predicates for Delivering a 72MB File (Top) and 4.2 MB (Bottom); the right lists the plug-ins that are compatible with the selection predicates (emboldened shows the selected plug-in)

	App Worst Case		App Second Best Case		App Best Case	
	4.2 MB	72 MB	4.2 MB	72 MB	4.2 MB	72 MB
DSL	+343%	+185%	+57%	+48%	+/- 0%	+/- 0%
100	+2979%	+ 2141%	+1013%	+1174%	+/- 0%	+/- 0%

Table 6.13: Predicates and Meta-Data for Delivering a 72MB File to Node LC

centricity in the middleware layer, these benefits can be gained by individual clients without prior network deployment or a complicated coding overhead for applications. The key findings from this section can be summarised as,

- When accessing content there can often be a divergence in the available providers
  - Different consumers will locate different sources and can subsequently only be optimised by enabling the exploitation of their particular 'view' of the network
- Diversity between consumers can often result in an inability for a single

static delivery system to satisfy given requirements for all

  - The varying properties of different consumers will often result in different levels of satisfication (for a given set of requirements) when accessing content. This means it is only possible to achieve optimisation by taking these variances into account and allowing per-node configuration to be performed

- Diversity between content can result in an inability for a single delivery system to satisfy given requirements for multiple items of content

  - Diversity between content will affect the ability for a given provider/ protocol to best serve the content; consequently, decisions must be made on not only a per-node but also a per-request basis

6.4.3 Case-Study 2: Consumer-Side Re-Configuration

The second evaluative case-study looks at how Juno can handle temporal variance. As defined in Chapter 3, this refers to the variance of a provider's ability to fulfil a set of requirements over time. The purposes of this case-study is therefore to validate that Juno's approach to delivery re-configuration can adapt an application to varying environmental conditions.

Case-Study Overview

The second case-study focusses on addressing temporal variance, in which a previously made decision becomes suboptimal. Generally, delivery protocols do not explicitly support sophisticated adaptation to address such changes. Juno therefore extracts the required adaptive functionality and manages it on behalf of delivery protocols. As such, plug-ins remain relatively simple entities that are externally managed. Performing consumer-side re-configuration involves three steps, (*i*) detection of an environmental change, (*ii*) re-selection, (*iii*) re-configuration. The first step is performed by the active plug-ins by dynamically modifying their meta-data during the delivery (i.e. to reflect current conditions), whilst the latter two steps are performed in a similar way to described in the first case-study.

In-line with the previous case-study, it is assumed that the key requirement is performance-oriented. Within this case-study, a high capacity node is downloading a 698 MB video using FTP at ≈2.5 Mbps. This node is resident on a campus network with a 100 Mbps connection to the Internet. The node's user has an account with a content distribution network (CDN) that uses a number of strategically placed replication servers. When the application initiates the download, the CDN does not possess a copy of the desired content. However, 20 minutes after the download starts, the CDN acquires a replicated copy. This is discovered by periodically repeating previous queries to ensure that up-to-date

Meta-Data Item	Selection Predicate
DOWNLOAD_RATE	HIGHEST
MIN_FILE_SIZE	<=698MB
MAX_FILE_SIZE	>=698MB

Table 6.14: Selection Predicates for Accessing File

provider information is available. At this point in time, the replication server has ≈6.5 Mbps of available upload capacity, thereby making it vastly superior to the currently selected provider. The server, however, only offers the content through HTTP; to access it, the node must therefore re-configure its delivery to interoperate with the new source.

Table 6.14 shows the selection predicates for the delivery request. The rest of this section now explores the behaviour of Juno when operating with this case-study. Importantly, this case-study is extensible to any two sets of delivery protocol and therefore this just shows an exemplary usage of the functionality.

Analysis of Case-Study

The above case-study has been implemented and deployed on the Emulab testbed. Figure 6.15 shows the instant download rate of the client when utilising Juno. It can be observed that the consumer initially receives a steady rate of ≈2.5 Mbps from the FTP server it is utilising. However, after 20 minutes, the replication server is discovered to also possess the content. This information is acquired through the periodic re-execution of queries by the Discovery Framework to ensure that up-to-date source information is available. This is therefore followed by generating meta-data for the new HTTP source; as detailed in Chapter 5, this is done using the iPlane service with only a small delay (<1 second).

Following this, the consumer re-executes the selection process to choose between the FTP server and the now available replication server (HTTP). The new server offers the content at ≈6.5 Mbps and is obviously selected based on the HIGHEST selection predicate detailed in Table 6.14. To re-configure the delivery, the FTP download is terminated before detaching the plug-in. Following this, the HTTP plug-in is attached and the content requested again; this time, however, the HTTP plug-in generates a range request to avoid downloading the previous data again. Clearly, the frequency of this re-configuration process is limited by a configurable value (default 1 minute) to ensure that oscillation does not occur and to give each plug-in an opportunity to reach its optimal performance.

In this case-study, the re-configuration only takes approximately 0.5 seconds, making its effect on the delivery negligible. Importantly, once the re-configuration has taken place, the download rate can be seen to increase significantly. This

allows the delivery to complete 18 minutes earlier, achieving a 36% faster delivery. Consequently, it is evident that delivery-centricity can only be achieved if the per-request configuration is extended to allow re-configuration at any point during the delivery due to *temporal* variance.

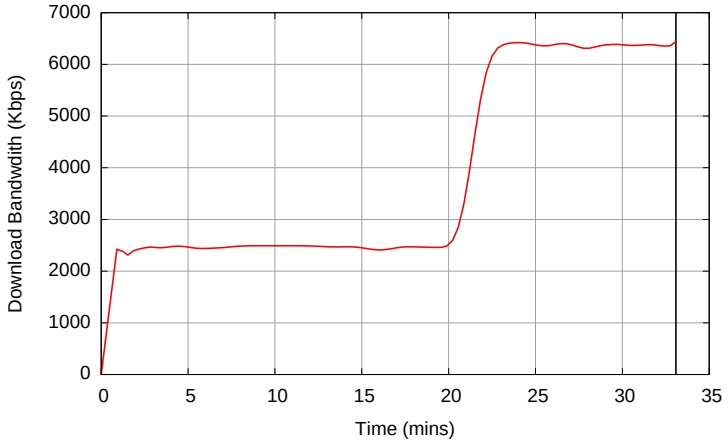


Figure 6.15: Download Rate of Juno Node with Re-Configuration

Figure 6.16 also shows the application layer throughput of the different download options; this is measured from the point at which the application requests the content to the point at which it is fully received. It can be seen that Juno's throughput (with re-configuration) is significantly higher than without re-configuration. This is due to the higher download rate achieved after the first 20 minutes. These two results can also be compared to the alternative of simply discarding the original data retrieved through FTP and starting a new HTTP delivery from the replication server (this would be a necessary step if using most applications such as Microsoft Internet Explorer). By discarding the first 20 minutes of delivery (286 MB), the throughput drops to 66% of Juno's, resulting in a 17 minute longer delivery. Other undesirable side effects also occur such as increased client bandwidth utilisation.

This case-study therefore shows that it is possible for Juno to adapt to the temporal variance identified in Chapter 3. On the one hand, this could be used to exploit a positive change to the environment (e.g. the discovery of a superior provider) or, alternatively, to address a negative change (e.g. network congestion at the current provider). The conclusions from the previous case-study can therefore be extended to state that it is necessary to not only allow per-node and per-request decisions to be made, but also to allow such decisions to be re-made at any point during a delivery. Juno's approach of abstracting the delivery process from the application, and allowing it to be dynamically re-configured therefore offers an effective solution to these issues. Importantly, by constantly monitoring the current choices and comparing them against alternatives, an application

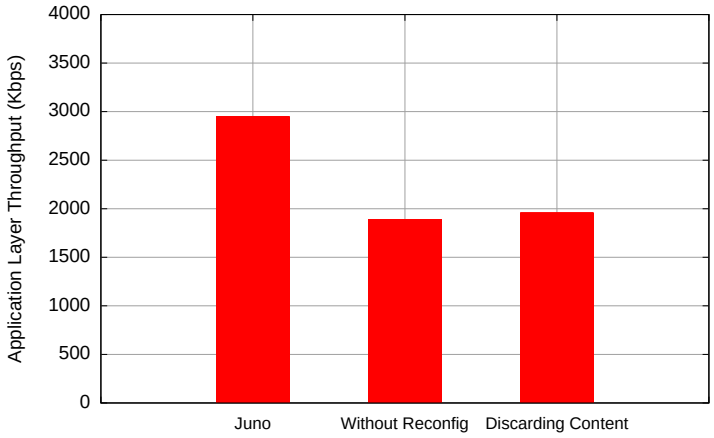


Figure 6.16: Average Application Level Throughput of Deliveries

operating over Juno can confidently ensure that its delivery requirements will always be met (within the constraints of the environment).

Summary of Findings

In summary, this experiment has (*i*) highlighted the presence of temporal variance, as identified in Chapter 3, and (*ii*) shown it can be effectively handled using Juno's approach of delivery re-configuration. With small files (e.g. 1 MB), it is often not critical to consider decisions beyond the initial request time. However, with large files, the likelyhood of decisions becoming suboptimal increases due to longer download times. Juno's approach to delivery-centricity, which involves constantly monitoring the environment in regards to the application's requirements, therefore offers an effective solution to this case-study. This is because it can modify previous decisions at any point in time by re-configuring the underlying way it accesses content. The key findings that can be extracted are,

- It is possible for a runtime decision to become incorrect later on due to environmental changes (i.e. temporal variance)
  - Changes to the provider(s), consumer(s) or network can mean that a previous decision becomes suboptimal compared to alternative sources
  - These can be both positive changes (e.g. discovery of a new source) or negative changes (e.g. increased network congestion)
- Request-time decisions can be as equally restrictive as design-time decisions. To perform a delivery without support for re-configuration can therefore result in an inability to guarantee that certain requirements can be satisfied for the entirety of a delivery

Node	Download	Upload
A	784 Kbps	128 Kbps
B	1.5 Mbps	384 Kbps
C	3 Mbps	1 Mbps

Table 6.15: Bandwidth Capacities of Initial Nodes

- Juno's re-configurable approach effectively offers a feasible solution to address these problems

6.4.4 Case-Study 3: Distribution Re-Configuration

The third evaluative case-study investigates Juno's ability to address both consumer and temporal variance at the same time, as well as the potential of distributed re-configuration. The previous sections have inspected these concepts individually, however, it is also necessary to ensure Juno can handle this complexity holistically. The purpose of this case-study is therefore to validate that Juno's distributed re-configuration support can be exploited to address the variance observed in Chapter 3.

Case-Study Overview

The previous two case-studies have involved the uni-sided configuration and re-configuration of behaviour to best fulfil certain performance requirements. These are obviously the most common use-cases due to the ease at which Juno can be deployed in a uni-sided manner. However, an interesting variation of this is the situation in which all nodes utilise Juno (including the provider). This would obviously be the case when a new application has been fully developed using Juno. Such a scenario subsequently allows adaptation messages to be exchanged between nodes to enable distributed re-configuration.

The third use-case explores this space; within this case-study there are a number of clients downloading a 698 MB file from a single server hosting Juno with HTTP configured. The server has 10 Mbps of upload bandwidth to distribute this content. In contrast to the previous experiments, the client application issues no delivery requirements. This is because the content is only available from a single (application controlled) server. Instead, the server application issues provision requirements to Juno stating that the UPLOAD_RATE meta-data should remain below 9 Mbps to ensure it that is not overloaded.

Initially three clients begin to download the content, thereby entitling them to 3.33 Mbps each. Each client possesses a different bandwidth capacity as shown in Table 6.15, thereby creating consumer variance within the experiment. After 20 minutes the demand for the content increases and 22 further nodes issue requests

to the HTTP server. This obviously constitutes a form of temporal variance, in which the operating environment changes. These nodes each request the content sequentially with 20 second intervals. The bandwidth capacities of the new nodes are distributed using the data from [40].

At the server, Juno detects this drain on its resources by monitoring the upload rate for each item of content; when it exceeds the given threshold (9 Mbps) it initiates a distributed re-configuration. This re-configuration involves changing its chosen delivery scheme from HTTP (limited scalability) to BitTorrent (high scalability). This constitutes probably the most common distributed re-configuration use-case, as is evident from the wealth of previous literature that looks at exploiting peer resources during flash crowds (e.g. [55][112]). This is therefore one example of a re-configuration response, although any other strategy can also be introduced, as detailed in Section 4.4.4. Unlike previous examples of peer-to-peer flash crowd alleviation, Juno therefore supports the easy addition of new strategies that are external to the protocol implementations.

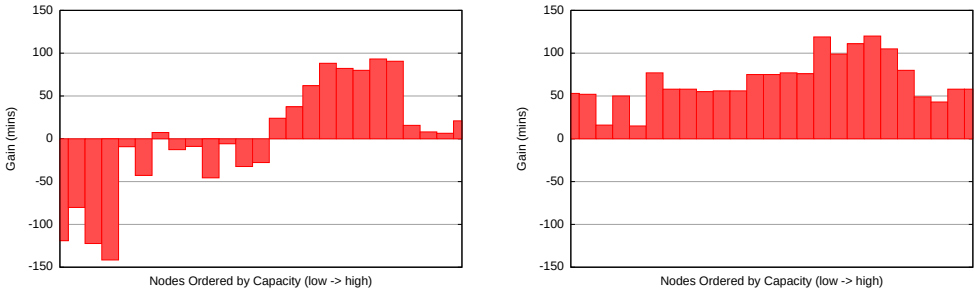
Analysis of Case-Study

The above case-study has been setup in Emulab over a number of nodes; Juno has then been deployed on each node (including the server) and measurements taken. Once again, as the case-study operates with performance-oriented requirements, this section now explores the performance benefits of utilising Juno's distributed re-configuration.

Figure 6.17 shows the gain, in terms of download time, of re-configuration for each node when compared to non-re-configuration. Nodes are ordered by their download capacity with the slowest nodes at the left. First, Figure 6.17a shows the circumstance in which the server mandates that all peers re-configure so it can lower its own utilisation. It can be seen that lower capacity nodes actually suffer from the re-configuration; 12 out of the 25 nodes take longer to complete their downloads. This occurs because during the initial bootstrapping period of the swarm, nodes tend to cluster into neighourhoods of similarly resourced peers [62]. In the case of the lower capacity peers, this leaves them interacting with fellow peers possessing as little as 128 Kbps of upload capacity. Due to their similarly limited upload resources, it is impossible for these peers to acquire download slots from the higher capacity peers. This leaves the lower capacity peers with low download rates.

The above observation emerges as a significant problem because a given re-configuration strategy is likely to benefit certain peers but disadvantage others. This clearly is an example of consumer variance and raises the question of how to incentivise peers that would potential become disadvantaged if they conform to the collective re-configuration. Juno takes steps towards addressing this by supporting fine-grained per-node re-configuration, i.e. allowing different nodes to

select their own re-configuration strategy. This allows each peer to accept or reject a re-configuration based on its own requirements, thereby improving fairness within the process. However, ultimately, the decision is made by the provider to ensure consistency within the system. This is because the provider has a position of dominance (and often trust), allowing it to perform a re-configuration that best matches it requirements. These requirements, however, are not necessary selfish ones; instead, it is likely that many providers will wish to improve the common good and offer the optimal configuration(s) to achieve this. In the case-study, each node is therefore allowed to select whether or not they implement the strategy, leaving the provider with both BitTorrent and HTTP support. Figure 6.17b shows the situation in which the server also provides the content simultaneously through HTTP, allowing each node to select its own plug-in. It can be observed that every peer improves its performance when utilising this strategy. It allows high capacity peers to exploit each others' resources whilst freeing up the server's upload bandwidth for use by the lower capacity peers. On average, through this mechanism, peers complete their download 65 minutes sooner. This is an average saving of 30% with the highest saving being 51%.



(a) System-Wide Re-Configuration (b) Per-Node Re-Configuration

Figure 6.17: Benefit of Juno using Different Re-Configuration Strategies

Figure 6.18 also shows the measured download bandwidth of a representative Juno node[†] possessing a 3 Mbps/1 Mbps down/upload capacity. The graph shows the bandwidth with re-configuration both enabled and disabled. This peer joins before the flash crowd and receives a steady bit rate of ≈600 Kbps early in its lifetime. However, after 20 minutes the flash crowd occurs, resulting in the degradation of its bit rate to ≈350 Kbps (-41%). Without adaptation of the delivery scheme, this situation remains until the flash crowd has alleviated (seen slightly after ≈150 minutes). Juno, however, re-configures 6 minutes after the beginning of the flash crowd. This occurs after the server has observed resource saturation for a given item of content. At first there is a significant drop in the

[†]Similar behaviour can be observed in the other nodes

bit rate to ≈200 Kbps; this occurs due to the bootstrapping period of BitTorrent, which involves peers exchanging chunk maps and bartering for each others' upload resources. Importantly, however, the bit rate quickly returns to that before the flash crowd (≈700 Kbps) with the usual end game period at the end [40].

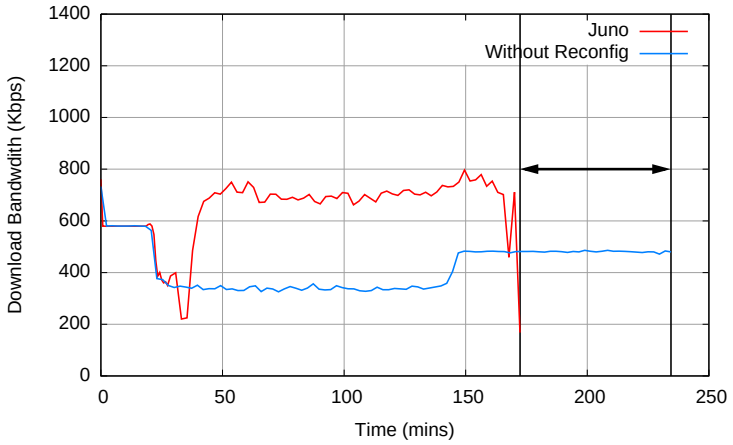


Figure 6.18: Download Rate of Consumer (i) with Re-Configuration (ii) without Re-Configuration

So far, two re-configuration strategies have been explored: per-node and system-wide. The decision of which to use is left to the provider as it is in the most important bartering position (i.e. it possesses the content). A per-node strategy is clearly the best approach if the provider is intent on improving performance for consumers; this is a likely situation if the provider is operating in a fluid commercial setting. In contrast, a system-wide strategy is likely to be preferred if re-configuration is initiated with the intent of reducing the resource overhead for the provider. This can be seen in Figure 6.19, which shows the upload rate of the provider when it both enforces system-wide re-configuration as well as allowing HTTP access to continue (per-node). It can be seen that the loading on the server increases rapidly in proportion to the number of nodes requesting the content. Without system-wide re-configuration, this results in a high level of server utilisation (upload saturation). However, with re-configuration enabled, the server reacts and adapts appropriately to its new operating environment. This can be seen by the rapid fall in utilisation. Over a short period of time the server remains seeding chunks to the BitTorrent swarm. However, after 105 minutes this process largely ceases with over 95% of bandwidth being freed.

These results can also be contrasted with the overhead of performing distributed re-configuration. The bandwidth costs of adaptation messages are trivial; of more interest is the re-configuration delays. Due to the fact that a number of different types of node operated in the experiment, Table 6.16 details the average, maximum and minimum re-configuration and bootstrapping times (on

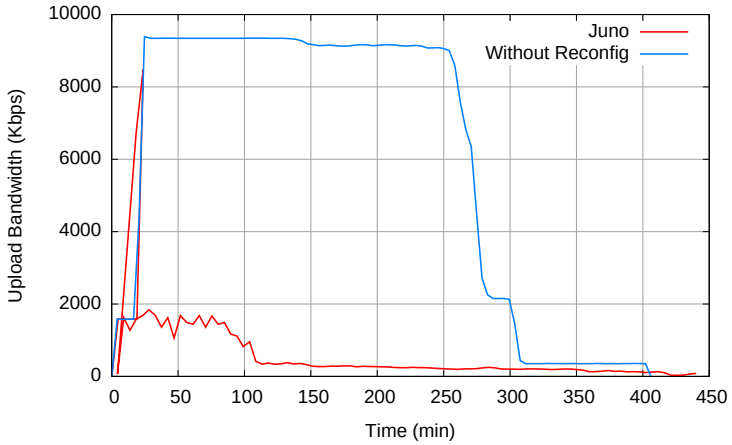


Figure 6.19: Upload Rate of Provider

	Average	Maximum	Minimum
Re-Configuration	29 Sec	42 Sec	13 Sec
Bootstrapping	6 Sec	18 Sec	3 Sec

Table 6.16: Re-Configuration and Bootstrapping Times for Clients

a per-node basis). The re-configuration time is the length of time required to re-configure the node for use with BitTorrent, whilst the bootstrapping time is the subsequent time it takes for a node to receive its first chunk. Consequently, these represent the period over which no download is being performed. Assuming the average download rate taken from the microscopic BitTorrent measurement study detailed in Section 3.3 (1613 Kbps), this would result in between 2.5 and 8 MB of bandwidth being lost by each node during the re-configuration period. This, however, can be considered as justified due to the increase in overall benefits; further, this can be alleviated by simply continuing the download with the previous system (HTTP) until the next system is ready (BitTorrent). At the server-side, the re-configuration time is much shorter (12 seconds) as it executes on a higher specification node and does not require any distributed interactions. It only initiates itself and waits for requests.

Summary of Findings

In summary, this experiment has (*i*) highlighted the holistic effects of consumer and temporal variance, and (*ii*) shown it can be handled effectively using Juno's support for distributed re-configuration. Importantly, by encapsulating protocols within plug-ins, this can be managed externally by Juno without either application or protocol developers being involved. This case-study has shown a relatively

straight-forward adaptation, however, far more significant alternatives can also be envisaged that involve various negotiations and the use of different plug-ins and services (e.g. transcoding). In summary, the main findings from this use-case are,

- Some changes in the environment can only be addressed through the cooperative adaptation of multiple nodes
  - This is most prevalent when there are only a limited number of (heterogeneous) available providers, thereby preventing consumer-side reconfiguration
- Advantages can include both performance and overhead improvements
  - Generally, the preference as to which is managed by the provider as it is in a position of dominance
- Such functionality is best placed in the lower layers to reduce the burden on application developers
- Such functionality should be externalised from individual protocols to improve extensibility

6.4.5 Further Case-Studies

The previous three sections have provided detailed case-studies of the three primary (re-)configurable operations performed by Juno. These are consumer configuration, consumer re-configuration and distributed re-configuration. Through these case-studies, it has been shown that Juno's design is both (*i*) feasible, and (*ii*) effective at fulfilling performance-oriented requirements. It has also shown that Juno's approach of supporting both per-node and per-request configuration is best suited to achieving this. Performance has been used as a primary requirement use-case because of its dynamic nature as well as the predominance of its usage. However, there are also a large number of alternative requirements that might be considered important to an application, e.g. overhead, security etc.

The purpose of this section is to briefly explore a more extended set of requirements that applications may choose to issue to Juno. Each section explores why an application might choose to generate a particular requirement as well as the configuration response that Juno will perform. Table 6.17 provides a brief overview of the extended meta-data investigated alongside the values set for a variety of protocols.

Protocol	Encryption	Anonymity	Upload Resources	Cost
HTTP	false	partial	false	various
HTTPS	true	partial	false	various
BitTorrent	false	none	true	0
Limewire	false	none	true	0
ed2K	true	none	true	0
RTP	false	partial	false	various
FreeNet	true	full	true	0

Table 6.17: Overview of Extended Meta-Data for Various Protocols

Security

Security is a term that covers many aspects of content system design. In a broad sense, it refers to the provision of mechanisms to prevent the unauthorised access to or modification of system resources. These resources can range from the content itself (at the data-level) to the infrastructure that serves it. In the context of a consumer accessing content, security generally refers to three functional aspects that can be offered by a delivery system,

- *Network Encryption:* This prevents intermediate parties observing or modifying the interactions between a consumer and its provider(s)
- *Data Validation:* This allows a consumer to validate (post-delivery) that an item of content is, indeed, the one requested
- *Anonymity:* This prevents certain parties from discovering that a consumer has accessed a particular item of content

Alongside these needs, providers will generally also have more extended requirements. The most obvious ones being access control and digital rights management. Both of these issues, however, are generally managed within the content-layer; for instance, using an encrypted shell. In-line with the rest of the thesis, this section focusses on consumer-oriented requirements. Security needs, from the perspective of different consumers, are likely to vary dramatically. Many applications will simply want their content's integrity to be validated, however, other applications will have far more stringent requirements. Due this observation, the different aspects of security are decomposed into individual items of meta-data that can be requested by the application. These are now discussed in turn.

Encryption. The first security-related item of meta-data is encryption. Encryption refers to a delivery system's ability to encode a data stream so that any intermediate parties cannot ascertain what is being accessed or alter the data being exchanged. This is a static item of meta-data that is exposed by each

delivery plug-in. It will primarily be requested by applications that view the content accessed by a consumer to be sensitive. This might occur, for instance, if the content is personal to the consumer or if access to the content is provided through monetary exchange. Examples of delivery protocols that support encryption include HTTPS, FreeNet and FTPS. Juno can easily be configured to utilise an encrypted delivery by adding the following requirement,

```
new MetaDataRule("ENCRYPTION", MetaDataRule.EQUALS, true);
```

This creates a new selection predicate (to be passed to Juno) that dictates that the chosen delivery protocol must offer encryption. Subsequently, the chosen delivery protocol will be restricted to ones that expose the correct meta-data.

Data Integrity. The next, highly important, aspect of content security is that of data integrity. This refers to an application's ability to validate that a received item of content is, in fact, the one that was requested. Within Juno, this is an implicit function that is offered through the use of hash-based content identification. Whenever a request is generated, the content is addressed using a hash-based identifier that uniquely identifies the content through the use of hashing algorithms such as MD5 and SHA-1. These are created by executing the hashing functions on the content's data. Consequently, once an item of content has been received, its integrity can be checked by re-executing the hashing process and comparing the output to the content's identifier. If they match, the content can be considered valid.

Anonymity. The third security aspect considered is anonymity; this refers to a delivery system's ability to hide a consumer's identity (e.g. IP address). Unlike the previous item of meta-data, this is not always static. On the one hand, some protocols such as FreeNet do implicitly support anonymity, thereby making it an entirely static piece of meta-data. However, other protocols such as HTTP do not implicitly offer the support; instead, it is a policy decision left to the provider. In contrast, there are also other protocols such as BitTorrent, which simply do not offer any anonymity.

To address this, anonymity meta-data is constructed on a sliding scale from zero to two. Zero indicates the provider does not offer anonymity; one indicates the provider offers partial anonymity; and two indicates the provider offers full anonymity. Subsequently, using the previous examples, BitTorrent is set to zero, HTTP is set to one, and FreeNet is set to two. This is because any node can connect to a BitTorrent swarm and discover the members; whilst, alternatively, in HTTP only the provider can discover a consumer's identity. Last, a FreeNet node offers full anonymity because no third party can discover what a consumer

is accessing. In the future, this can also be extended to include provider-specific information so that different providers can expose their personal anonymity policies.

One other mechanism by which a node can gain anonymity is through the use of services such as Tor [30], which offer an anonymous VPN-like network connection. Consequently, any higher level services become completely anonymous when connected to the Tor network. Clearly, this should be taken into account by Juno when exposing the meta-data because all plug-ins would subsequently become anonymous. To address this, the Context Repository maintains a global ANONYMOUS_UNDERLAY item of boolean meta-data. When exposing meta-data, each plug-in can subsequently check this to ascertain whether or not their configuration is anonymous by default.

Resilience

In the context of content delivery, resilience refers to the certainty with which an application can successfully retrieve an item of content. Juno's design implicitly offers a greater degree of resilience when compared to previous approaches. This is because Juno is not limited to utilising an individual provider/protocol. Instead, it can exploit any available sources, thereby providing a far greater degree of redundancy in the face of failures. Juno therefore can re-configure to use alternate sources when a previous one becomes unavailable. Further, because Juno utilises a shared content manager this can be done without the loss of prior downloaded data.

Beyond this, certain applications are also likely to issue resilience requirements that specifically refer to the confidence with which a provider can serve an item of content. Juno supports this by allowing plug-ins to expose resilience meta-data. For instance, a BitTorrent swarm can serve an item of content with varying degrees of certainty based on the replication level of the rarest chunk [88]. If, for example, there is only one copy of the rarest chunk, then it is highly likely that at some point the content will become unavailable due to the departure of this chunk from the swarm. This resilience level can therefore be exposed to applications. A measure of resilience from a consumer's perspective can be modelled by a probability indicating the certainty with which an item of content will be delivered successfully. Evidently, a value of 1 would indicate that there is no possible manner in which the delivery can fail (this, however, is clearly impossible). This meta-data can therefore be generated dynamically based on any pertinent runtime properties and exposed to applications in a similar way to performance meta-data.

Upload Bandwidth Resources

Upload resources refer to the upload bandwidth consumed by a delivery protocol. This is a static consideration for certain protocols (e.g. HTTP) whilst being a dynamic aspect for others (e.g. BitTorrent).

Generally, client-server systems such as HTTP and FTP have no upload bandwidth consumption, excluding the obvious TCP ACKs. In contrast, however, peer-to-peer systems such as BitTorrent and Limewire also offer resources for uploading content to other peers. In fact, protocols such as BitTorrent rely on this to ensure high reciprocal download performance. Currently, to represent the required upload resources, Juno simply uses a boolean item of meta-data that can be expressed with the following code,

```
new MetaDataRule("REQUIRES_UPLOAD", MetaDataRule.EQUALS, true);
```

This is therefore very coarse grained because it only allows a consumer to avoid using delivery systems that require upload resources. However, it is still extremely useful in certain circumstances. For instance, it allows a low capacity device or a node utilising a rate-based connection to avoid the usage of such systems.

Monetary Cost

The monetary cost is an extremely important aspect of most real-world systems. It refers to the monetary charge issued to any consumers that wish to access a specific item of content from a provider. This therefore applies to any providers that request money on a per-view basis.

Generally, protocols that handle this are separate from the underlying delivery system. For instance, a user might be required to securely login to a website to purchase an item of content before being granted access via HTTP. This, for instance, is the approach of the current iTunes implementation, which has access to the iTunes Store provider. In the Juno approach, however, multiple providers are accessible, allowing the application to select the cheapest one that fulfils its criteria. In terms of music providers, the list is extensive including iTunes, Amazon, HMV and Napster, to name a few.

Juno applications can easily introduce the principles of monetary cost; for instance, if an application were only willing to pay less than 50 pence, this could be stipulated using the following rule,

```
new MetaDataRule("MONETARY_COST", MetaDataRule.LESS_THAN, 50);
```

When issued with this meta-data, Juno would therefore only utilise providers that were sufficiently cheap. Obviously, the plug-in that accesses this provider

would have to be configured with the user's account details (which would be provided through its parameters). Alternatively, other applications may simply wish to access the content in the cheapest manner, regardless of other concerns such as performance; this can similarly be performed using the following rule,

```
new MetaDataRule("MONETARY_COST", MetaDataRule.LOWEST);
```

6.5 Critical Evaluation Summary

The previous two sections have provided a quantitative evaluation of both the discovery and delivery aspects of Juno's design. This section revisits the core design requirements of Juno to critically evaluate how well they have been fulfilled. These were detailed in Section 4.2 in the form of five key requirements for a content-centric solution. From these requirements, four questions can be derived; specifically, these are,

- *Open Abstraction:* Does the middleware offer a content-centric and delivery-centric abstraction?
- *Delivery-Centric:* Can the middleware successfully fulfil delivery requirements issued by the application?
- *Interoperable:* Can the middleware successfully interoperate with multiple diverse third party content systems?
- *Deployable:* Is it possible to deploy the middleware instantly without significant cost?
- *Extensible:* Can the middleware be extended to utilise new protocols and support future delivery requirements?

The first requirement has already been implicitly shown through the design section. Therefore, this section now briefly explores how the four other requirements have been holistically addressed by Juno's design.

6.5.1 Delivery-Centric

Delivery-centricity refers to a content-based system's ability to accept and fulfil potentially diverse delivery requirements. Section 6.4 has explored delivery-centricity using a number of case-studies. Alongside the results from Chapter 3, this has shown there to be a number of benefits associated with deploying delivery-centricity. To achieve this, Juno relies on three stages of operation: discovery, selection and delivery. The effectiveness of each of these processes is now explored in turn.

Discovery is the process by which potential sources are acquired. This is vitally important as delivery-centricity in Juno is based on the ability to re-configure between different providers. Due to the skewed nature of content popularity and replication [74][147], this is a highly effective approach when accessing popular objects. This is because there will generally be many providers offering the content in many different ways. It has been shown that, when available, this diversity can be effectively exploited to satisfy application requirements. In contrast, unpopular content is generally not so well sourced, making it more difficult for Juno to re-configure between different providers. To address this, however, Juno also supports distributed protocol re-configuration, allowing consumers to adapt providers based on personal requirements.

Once a set of potential sources have been located, it is necessary to *select* the one(s) that best fulfil the delivery requirements. Selection predicates are used to intuitively describe requirements. The selection process is therefore simply a matter of applying the selection predicates to the meta-data of the available sources. The challenge is generating accurate and representative meta-data for each of the available providers; this has been investigated in Chapters 3 and 5. It has been shown that effective mechanisms exist for performing this task in a low overhead way. Further, Juno's component-based implementation also allows these mechanisms to be replaced and adapted at any point to reflect the operating conditions of the node. Clearly, it is hoped that such components will be developed and deployed by third parties in the future.

Delivery is the final stage in the process; this occurs once an optimal provider has been chosen. Juno has shown itself to achieve this task effectively as its interoperable nature allows it to effectively re-configure between the use of different delivery protocols. Further, it has been shown that it does this whilst maintaining a standard abstraction between itself and the application. To ease development, this abstraction can even be dynamically selected by the application to match its preferred access means (e.g. file reference, live stream etc.). So far, a number of protocols have successfully been implemented in this way (including HTTP, BitTorrent, Limewire and RTP) and therefore it can be confidently inferred that this will extend to a range of other protocols as well.

Juno has been shown to effectively address these three functional steps collectively. Unlike previous content-centric designs, it does not restrict itself to individual statically selected delivery protocols. Instead, it can adapt to reflect application needs and operating conditions to best access the content. Consequently, through this flexibility, it can be inferred that Juno will continue to offer strong delivery-centric support for a variety of future applications.

Plug-in	LoC	Concrete Invocations
HTTP (java.net)	45	2
BitTorrent (HBPTC)	51	1
BitTorrent (Snark)	65	1
RTP	42	5

Table 6.18: Coding Complexity of the Initiate Delivery Methods of various Stored and Streamed Plug-ins

6.5.2 Interoperable

Interoperability refers to a system's ability to interact with third party systems. Many of the principles investigated in this thesis, such as deployability and delivery-centricity, are founded on Juno's ability to achieve interoperation. It can be identified that interoperation with content-based systems has two key requirements, (*i*) protocol interoperation and (*ii*) address interoperation. Protocol interoperation is the ability of a system to interact on the protocol-level with another system; whilst, address interoperation is the ability to uniquely identify content within multiple systems.

Within Juno, protocol interoperation is achieved using configurable plug-ins. The purpose of these plug-ins is to translate abstract method calls to concrete protocol interactions. To achieve this, a suitably high level abstraction has been constructed to hide the underlying discovery and delivery functionality. To study this approach, Table 6.18 shows the number of lines of code required to perform the abstract-to-concrete mappings of the currently implemented delivery plug-ins. This details the number of required lines within the delivery initiation method, alongside the number of concrete invocations to the underlying toolkit implementation (these plug-ins were implemented with re-used APIs). Evidently, the complexity is very low, indicating that the generic abstraction does a suitably good job of abstracting their functionality. From this it can further be derived that the current abstraction is sufficiently generic to map easily onto a number of different protocols; it can therefore be inferred that this trend will continue with alternative protocols.

The second form of interoperation is address interoperability, i.e. the ability to interact with any arbitrary addressing scheme. Natively, Juno does not support this, as it is impossible to translate one-way hashing functions (as different systems use different hashing algorithms). Instead, Juno allows multiple hash-based identifiers to be embodied within a single identifier using the Magnet Link standard [19]. This is further supported by the JCDS, which allows individual addresses (e.g. a SHA1 address) to be mapped to other types of hash-based addressing (e.g. Tiger Hash, MD5 etc.). Consequently, Juno has a high degree of address interoperability, with the ability to uniquely identify content in a wide

range of systems. Its major limitation, however, is when interoperating with systems that do not support unique addressing. For instance, keyword searching is a popular method for users to interact with indexing systems (e.g. Google); such mechanisms, however, cannot be used by Juno because Juno requires an underlying method of unique content addressing, i.e. some form of hash-based identifier. To index such systems it is therefore necessary to manually perform the mapping using the JCDS. It can therefore be stated that Juno can interact with any third party discovery system as long as it supports the underlying ability to uniquely identify content.

6.5.3 Deployable

Deployability refers to a system's ability to be practically deployed. This is an important research challenge in any distributed system as it has previously proven prohibitory for a number of technologies including IPv6 and multicast [34]. Two deployment issues exist when building new networking paradigms, (*i*) hardware deployment, and (*ii*) software deployment.

Hardware deployment is by far the most difficult deployment challenge: it is slow, expensive and often unsupported by third parties. Juno effectively addresses this challenge by entirely removing the need for new infrastructure. Instead, Juno exploits its interoperable capabilities to interact with existing infrastructure, thereby removing the complexity and infrastructural needs of building new network protocols. Further, Juno's approach similarly means that providers and ISPs do not need to introduce any new hardware.

Software deployment is generally an easier problem to address, as it offers better potential for progressive deployment. Generally, this takes place in either the operating system and/or the application, at both the provider's and consumer's side. Juno does not require modifications at the provider-side, which therefore results in far greater deployability. This is because existing providers can simply be interoperated with, regardless of their knowledge of the content-centric system. Similarly, no network-level software deployment need be performed (e.g. router updates). It is clearly necessary, however, for software to be updated at the consumer-side because Juno is a consumer-side middleware. Consequently, the middleware must be installed on every node that wishes to utilise the content-centric functionality; this is, however, more convenient than performing operating system modifications, considering that it can simply be bundled with the application (its Java implementation also makes it highly portable).

6.5.4 Extensible

Extensibility refers to a system's ability to be extended to include new functionality and support. This is a vital property for most real-world systems that are

subject to frequent advances and changes in their environment. Content networking is an example of a domain that has witnessed a huge variety of technological advances during the last decade.

The development process has shown Juno to be a highly extensible framework due to its component-based design. Its functionality is separated into individual building blocks that maintain explicit dependencies whilst offering predefined services. Consequently, Juno can modify these interconnections to introduce new components into its architecture. The primary use-case for this is the introduction of new content protocols (i.e. discovery and delivery plug-ins). However, another key aspect of Juno's extensibility is the ability to introduce new meta-data generator components to allow a delivery plug-in to have new meta-data associated with it (c.f. Section 5.4.4). In both these use-cases, the process of expanding functionality simply involves building the new plug-in and then adding a reference to it in the Configuration Repository. By implementing the standardised interfaces required for this, Juno simply detects the new functionality and integrates it into the decision making process.

Obviously, this process has been validated multiple times as new plug-ins have been developed. This has also been validated by third party developers; Skjegstad et. al. [129] detail the development of two discovery protocols using Juno. Specifically, this involved the development and integration of a number of peer-to-peer components into Juno's architecture. At the time of writing, Juno is still being used in the project. Clearly, this provides strong evidence for the ease at which Juno can be extended.

Beyond this, Juno also supports the extension of its core framework functionality through a similar process. As previously mentioned, all components are connected using explicit bindings, which can be subjected to manipulation. The core aspects of Juno can therefore be similarly modified at runtime in the same manner. This is further assisted by Juno's design philosophy of building components with life cycle awareness, as well as the use of explicit parameters and state that can be exchanged between new and old components.

6.6 Conclusions

This chapter has evaluated the Juno middleware in terms of its content-centric and delivery-centric services. First, the Discovery Framework was investigated, looking at the performance and overhead of using plug-ins alongside the JCDS. Following this, the Delivery Framework was evaluated using a number of case-studies to show how the heterogeneity quantified in Chapter 3 could be achieved using Juno. Following this, was a critical evaluative summary, which returned to the design requirements described in Section 4.2 to ascertain how effectively they had been fulfilled. The following conclusions can therefore be drawn from this

chapter,

- Juno effectively realises a content-centric and delivery-centric abstraction
  - Through this abstraction, applications can access content in a way that is best conducive with their needs (e.g. streamed with >500 Kbps)
- Juno effectively achieves unified discovery over a range of third party content systems
  - Any system using hash-based unique content identification can be seamlessly interoperated with using plug-ins
  - Cooperative indexing offers an attractive way through which these third party systems can be further integrated
    - The performance, however, is based heavily on the request profile of the application
    - An ideal request profile is: a high request rate, and a small number of objects with highly skewed content popularity
    - However, various combinations of these still achieve a high performance
  - The overhead of Juno's plug-in approach is of a manageable size and can be further lowered by using the JCDS
- Delivery-centricity can be achieved through intelligent source selection and consumer-side (re-)configuration
  - This can only be achieved using Juno's approach of per-consumer and per-request (re-)configuration
  - Juno's approach, however, is restricted to utilising existing providers and can therefore only achieve delivery-centricity if appropriate providers are available
  - Juno, however, can also perform distributed re-configuration amongst multiple Juno-aware nodes
  - Further diverse delivery requirements can also be stipulated and satisfied using Juno, e.g. anonymity, monetary cost etc.
- Content support in applications can be dynamically extended through Juno without code modification

Chapter 7

Conclusion

7.1 Introduction

The previous chapters have investigated the challenges of building and deploying a content-centric and delivery-centric paradigm in the Internet. A new abstraction has been defined that allows applications to generate location-agnostic content requests associated with specific delivery requirements. This has been realised through the design and implementation of a configurable middleware framework called Juno. It utilises reflective software (re-)configuration to allow seamless interoperation with existing content providers (using heterogeneous discovery and delivery techniques). This ability puts Juno in the unique position of being able to exploit the availability of content and resources provided by third parties. In its simplest form, this allows immediate deployment to take place without the need for network operators or providers to modify their behaviour. However, by supporting access to many different providers, Juno is further able to achieve delivery-centricity by dynamically selecting between them based on various requirements issued by higher level applications.

This section concludes the thesis; first, a brief overview of the thesis is given, looking at each of the chapters in turn. Next, the major contributions of the thesis are detailed, followed by other significant results. Then, several avenues of future work are described before, finally, providing the concluding remarks.

7.2 Overview of Thesis

Chapter 1 introduced the broad topic of research undertaken within this thesis. It presented the key research goals as well as placing the work within the field of existing work in the domain.

Chapter 2 provided a detailed background to content networking. It first

detailed the key content discovery and delivery technologies available today. Following this, it explored the primary techniques used for achieving interoperation in network systems. Alongside this, it also performed a detailed analysis of existing work in the domain of content-centric networking, looking at previous support for deployment, interoperation and delivery-centricity.

Chapter 3 measured and modelled the dynamics of current popular delivery systems. It identified key parameters in each system and explored how they varied both temporally and between consumers. The primary finding was that it would generally be impossible for static delivery provider/protocol selection to achieve global optimisation, therefore requiring the use of some form of delivery adaptation.

Chapter 4 proposed a new content-centric and delivery-centric abstraction that was realised through the design of the Juno middleware. This abstraction allowed applications to associate location-agnostic content requests with delivery requirements that must be fulfilled by the underlying system. The design promoted the use of software (re-)configuration to enable interoperation with existing content systems, thereby improving deployment and enabling the exploitation of this wealth of content and resources. Through this, it was shown how Juno can achieve delivery-centricity by dynamically switching between the use of different providers and protocols.

Chapter 5 investigated deployment challenges regarding the middleware design, as well as appropriate solutions. Specifically, a shared lookup system called the Juno Content Discovery Service (JCDS) was designed to address the limitations of contacting multiple discovery systems simultaneously. Alongside this, the challenge of deciding which provider/protocol to use to satisfy given delivery requirements was investigated. The use of meta-data generation components was proposed to allow consumers to calculate predictions regarding a particular provider/protocol's ability to satisfy requirements.

Chapter 6 evaluated the Juno middleware based on the research goals detailed in Chapter 1. First, it inspected the performance and overhead issues of utilising the Discovery Framework's interoperable approach to discovering content in third party systems. Following this, the Delivery Framework was evaluated using a set of key use-cases to highlight the strengths and weaknesses of the design. It was shown that Juno could effectively discover and access content in third party systems, as well as feasibly gaining the performance advantages detailed in Chapter 3.

7.3 Major Contributions

The Introduction of a new Delivery-Centric Abstraction

A key contribution of this thesis is the introduction of delivery-centricity into the existing content-centric abstraction. A delivery-centric abstraction is one that allows applications to associate content requests with sets of delivery requirements. These requirements collectively stipulate the quality of service required by an application, and can relate to such issues as performance, resilience and security.

This thesis has explored how requirements can be based on either static or dynamic properties. The stipulation of static requirements allows an application to ease the development burden by abstracting away the delivery process and its associated complexities. Despite this, it has been shown that an application could, instead, make these decisions at design-time and statically implement support within the software; this makes the benefits solely developmental. In contrast, however, requirements based on dynamic properties *must* be resolved during runtime. This is because the ability of a particular provider or protocol to satisfy such requirements can only be decided on a per-node basis. Consequently, this makes design-time decisions suboptimal, creating a strong need for the runtime support offered by Juno.

Chapter 3 investigated this hypothesis in detail to show that three of the major delivery protocols suffer from high degrees of runtime variance. This chapter focussed on the provision of performance-oriented requirements in these delivery protocols, through the use of measurement studies, emulated experiments and simulations. The findings showed that often two nodes accessing content from the same provider are likely to get different qualities of service if either (*i*) they have different runtime properties (e.g. location, upload bandwidth etc.) (*consumer variance*), or (*ii*) they access the content at different times (*temporal variance*). This problem was found to be exacerbated further if those consumers choose to access different items of content. Consequently, abstracting applications away from this complexity allows sophisticated 'behind-the-scenes' adaptation to be performed to ensure that content is accessed from the source(s) that offers the best match for the requirements at any given time.

The Juno Middleware Design and Implementation

The next significant result from this thesis is the design and implementation of the Juno middleware, which offers a solution to the existing failings identified in content-centric network designs; namely, no support for delivery-centricity, a lack of interoperability and poor deployability. It was identified that these problems emerged from the lack of flexibility that is associated with operating at the network-layer. Therefore, to address this, a middleware approach was taken.

Juno therefore realised the delivery-centric abstraction, alongside providing the underlying support to implement it.

A component-based architecture was designed that consisted of three abstract bodies of functionality, (*i*) a Content Manager to handle all interactions with local content, (*ii*) a Discovery Framework to manage the discovery of content residing in multiple third party content systems, and (*iii*) a Delivery Framework to dynamically select the optimal provider(s) to access the content. All this is managed by a middleware framework that offers a standardised delivery-centric abstraction to the application. This abstraction allows applications to issue requests for content whilst dynamically stipulating their preferred access mechanism (e.g. stored, streamed etc.), alongside any delivery requirements they have. Importantly, behind this abstraction, Juno dynamically adapts to ensure that these requirements are met. The evaluation showed that the two key stages in Juno's operation (discovery and delivery) are performed effectively with acceptable levels of overhead.

Content-Centric Interoperation with Existing Providers

The next significant result from this thesis is the introduction of technologies and support for the content-centric interoperation of applications with third party providers. Importantly, by introducing this within a unified middleware infrastructure, this can be achieved without complicated application involvement. Further, new protocols can be dynamically introduced at the middleware layer, thereby transparently expanding the capabilities of the application. Through this, applications are provided with access to all content previously available through a simple delivery-centric abstraction, without the need to develop support themselves. Similarly, providers are also not required to modify their behaviour, indicating that applications utilising Juno would gain access to a far greater pool of content and resources than alternate design. Juno therefore offers a way in which applications can immediately improve their development practices without high complexity or deployment challenges.

7.4 Other Contributions

Detailed Analysis of Delivery System Dynamics

A notable contribution of this thesis is the detailed analysis of a variety of popular content distribution protocols. Through simulation, emulation and real-world studies, it was shown that HTTP, BitTorrent and Limewire all suffer from significant runtime heterogeneity. This heterogeneity was categorised as either consumer or temporal variance. This referenced how the ability of a given provider to satisfy requirements varies both over time and between different consumers. This

detailed study therefore offers new evidence for the explicit need for runtime (re-)configuration between providers and protocols. Specifically, it was shown that this (re-)configuration must be performed on a per-request and per-node basis to ensure that optimality can be achieved.

Techniques for the Dynamic Resolution of Delivery Requirements

An important result from this thesis is an architectural approach to generating and comparing content system capabilities with dynamic requirements. This involves decomposing delivery functionality and meta-data generation into independent pluggable components that can be (re-)configured at runtime. This allows meta-data generators to be attached based on the operating environment and the provider support. Therefore, non-aware providers can have meta-data generated passively on their behalf by one of the approaches proposed in Chapter 3, whilst other Juno-aware providers can explicitly offer information through reflective interfaces. This therefore offers a new way in which *multiple* delivery systems can be compared at runtime, as opposed to previous attempts that solely deal with comparing individual systems (e.g. server selection). Further, this process can continue after a delivery has begun to ensure that poor decisions can be remediated through later re-configuration (i.e. addressing temporal variance). Last, it also offers development support for extending requirements to include many diverse aspects, far beyond the remit of previous approaches. Clearly, this vital contribution is therefore paramount for enabling the deployment of a new delivery-centric abstraction.

Cooperative Content-Centric Indexing

The promotion of interoperability as a mechanism for improving content-centric networking has been an important contribution. However, the use of consumer-side (re-)configuration to achieve this means that there is an overhead involved. A further contribution is therefore a cooperative indexing scheme for aggregating information discovered in individual third party discovery systems. The Juno Content Discovery Service (JCDS) performs this operation, whilst allowing the Discovery Framework to intelligently detach plug-ins that (mostly) index content already available in the JCDS. It therefore extends previous work on interoperability middleware [68] and multi-protocol applications [27] to improve both performance and overhead issues. This also extends this work into a new domain, thereby addressing the unique problems of content-centric interoperation (e.g. unique addressing). It has been found that this offers a highly effective means to overlay content-centric lookups over many different heterogeneous systems.

A Development Environment for Content System Functionality

Currently, no standardised development frameworks are available for building content systems and protocols. Therefore, an important contribution of this thesis is the design of a supportive development environment in which such systems can be built. First, this thesis has promoted a number of design principles; namely, component-based decomposition, the use of standardised APIs and the dynamic (re-)configuration of services. However, to further support this, Juno also offers a number of utility classes that can assist in the development process; this allows such things as interface definition, service instantiation, content management and software deployment to be easily handled. Further, by building new content protocols within Juno, they become immediately compatible and accessible to any applications built over Juno. This contribution has been validated by the third party usage of Juno; Skjegstad et. al. [129] detail the construction of two peer-to-peer discovery protocols in Juno, highlighting the degree of (re-)configurability offered by the middleware.

7.5 Juno in the Wider Context: Challenges and Limitations

It has been concluded that Juno's approach to building content-centric applications is highly effective at fulfilling the research goals discussed within this thesis, namely improving interoperability, deployability and delivery configurability. However, it is important to understand these contributions in the wider context of current content distribution systems. Specifically, this refers to the future challenges and limitations of utilising Juno in the context of the real-world. These come about due to the use of third party infrastructure, alongside the need for applications to subsequently modify their own behaviour. This section briefly summarises the challenges and limitations when looking at Juno from the wider perspective.

Performance and Overhead

To enable the use of third party resources, it is necessary for Juno to interact with providers that are outside of Juno's immediate control. This means that, unlike clean-slate solutions, it is impossible to design new bespoke discovery and delivery optimisations. Instead, it is necessary to operate within the confines of the infrastructure interoperated with. This means that Juno is restricted to the performance and overheads of any such third party systems. For instance, if only low performance content providers are available, it is impossible to gain a higher performance than these providers allow. One way this problem could be addressed is through the successful deployment of Juno at the provider-side, thereby allow-

ing distributed (re-)configuration to take place. Despite this, initially Juno must operate within the confines of available content providers. However, it is also important to understand that any future (superior) content protocols can easily be integrated into Juno's architecture, thereby improving any prior performance and overhead limitations.

Consumer Uptake

This thesis has focussed on supporting consumer access to content; therefore, clearly, an evaluative mark of success is the future uptake of Juno by application developers. This, however, is often the most difficult challenge, as simply providing the technological support for deployment does not necessarily result in wide-spread adoption.

There are two stages involved in the uptake of Juno by developers at the consumer-side. First, it is necessary to persuade developers to incorporate Juno's abstraction into their applications; and, second, it is necessary to actually implement the support using Juno's prototype. The introduction of Juno into new applications is therefore largely a marketing challenge, in which developers must be persuaded of the benefits discussed in this thesis. Considering the simplicity of Juno's interface, it is unlikely that alternative approaches (e.g. using a HTTP API) would be necessarily more attractive from a development perspective. Therefore, detailed and simplistic documentation is necessary to assist developers that are previously used to existing location-oriented toolkits. In terms of more sophisticated software projects, it is likely that many applications will also have greater concern with potential overhead and complexity issues (e.g. memory footprints, dependency management). Within the wider context, it is therefore important to ensure the complexity of Juno remains low. Further, it must be capable of build-time optimisation to ensure that only the minimum support is provided for a given application. Last, this clearly must be accompanied by increasingly sophisticated requirement support to provide sufficient functionality for various important real-world aspects such as security, resource management and monetary exchange. Importantly, however, through Juno's deployable and interoperable technologies, it becomes possible to progressively introduce this support, thereby easing these uptake challenges.

Provider Uptake

The second form of uptake, is the introduction of Juno into content provider infrastructure. One observation made during Chapter 6 is that the cooperation of providers can dramatically improve both performance and overheads. Consequently, this is a vitally important challenge.

As with consumer uptake, it is necessary to persuade providers that the benefits of utilising Juno outweight any associated costs with re-engineering software.

Clearly, this is an easier task when dealing with new providers that have not invested significant funds into previous technologies (e.g. Akamai's redirection infrastructure). Therefore, because there are fewer providers than consumers, it is likely that this task will be more difficult. This is because there are a small set of well established providers that already control a significant market share (e.g. YouTube, RapidShare, Akamai etc.). Therefore, it would be necessary for these companies to utilise Juno to ensure its widespread provider-side uptake. It is because of this that backwards compatibility with providers has been a focus of Juno's consumer-side design. The major challenge regarding provider-side uptake is therefore incentivising providers to modify their behaviour. This could be done by promoting the benefits detailed in this thesis (e.g. supporting distributed re-configuration for scalability reasons). However, this also requires consumer-side uptake to allow such benefits to be gained. Consequently, it is likely that an intermediate stage would be necessary in which third party providers cooperate with Juno without actually utilising it (e.g. uploading information to the JCDS). Beyond this, progressive provider-side deployment is obviously fully support in Juno and therefore this could take place over an extended period of time.

7.6 Future Work

The previous section has detailed a set of challenges regarding this thesis. The section now details a prominent set of specific future work that can be undertaken to extend this research.

Extended Requirement Support

This thesis has focussed on the provision of support for performance-oriented requirements. However, alongside this, it is also important to explore the variety of other requirements that applications may generate. These have been briefly discussed in Section 6.4.5 (e.g. encryption, anonymity, overheads, monetary cost etc.). A key aspect of future work is therefore to study these in far greater detail. This will involve both developing the necessary meta-data generators to model the requirements, as well as performing a study to understand how applications might generate such needs. Clearly, it would similarly be necessary to evaluate Juno's ability to satisfy such requirements through its architectural approach to requirement resolution.

Further Evaluative Testing

So far, Juno has only been evaluated in a controlled research environment (i.e. through simulations, emulations and measurement studies). However, it is also important to gain an understanding of how Juno would perform in a real-world

setting. This includes performance aspects regarding its fulfilment of requirements, as well as various uptake issues. An important area of future work is therefore to perform a more comprehensive evaluation of Juno when operating in the real-world (with real applications and users). This would allow a true understanding to be gained of the potential of content-centric networking, as well as Juno's design philosophy.

Expanding Content Abstractions

So far, the main focus has been on accessing stored content. This is suitable for many applications, however, it is also important to better explore the use of Juno's streamed media abstraction, alongside the requirements that are important to applications in this domain [140] (e.g. startup delay, continuity etc.). Beyond this, it would also be interesting to investigate the use of other content abstractions with Juno such as interactive and 3D content. Clearly, this must take place in conjunction with the development of further standard interfaces. The exploration of this space would therefore allow the extensibility of Juno to be exploited to its full potential.

Expanding Distributed Re-Configuration Approaches

Juno currently supports only a limited form of distributed re-configuration, which involves one (or more) nodes issuing re-configuration requests to other nodes. This obviously can be expanded to include a far wider range of aspects. Most important is the integration of more sophisticated dissemination techniques for propagating the requests. This also includes introducing a more sophisticated learning process by which nodes can improve their decisions (early work regarding this is detailed in [139]). Further, in conjunction with this, it should be made possible for negotiation to take place between multiple conflicting nodes. Last, incentive mechanisms should be introduced that encourage peers to cooperate in distributed re-configuration; this is a vital prerequisitive in the real-world, as some nodes might not benefit themselves from performing re-configuration.

Introduction of Greater Provider Support

Currently, Juno is primarily a consumer-oriented middleware that focusses on improving content access. An important area of future work is therefore to introduce greater support for applications that also wish to provide content. Juno already exposes the IProvider interface to allow this, however, this functionality has not been focussed on, or evaluated thoroughly. Any future work should therefore look at allowing applications to generate provision requirements that can be satisfied by Juno; this would be particularly interesting when using these to fuel negotiation between providers and consumers.

7.7 Research Goals Revisited

A number of research goals were detailed in Chapter 1; this section now revisits these to ascertain how effectively they have been fulfilled.

To define an extended notion of a content-centric abstraction encompassing both discovery and delivery, capturing the requirements of (existing and future) heterogeneous content systems. The need for this goal was explored in Chapter 3 through the quantitative analysis of three key delivery systems using simulations, emulations and real-world measurement studies. It was found that due to runtime delivery system dynamics, it would be impossible to achieve this goal without an adaptive approach to content delivery. This is because the ability of a provider to serve an item of content varies dynamically both over time and between different consumers. This goal was therefore addressed in Chapter 4 through the design of a new delivery-centric abstraction that allows applications to stipulate delivery requirements when requesting content. The underlying process by which it is accessed, however, is abstracted away from the application, thereby allowing the type of adaptation identified as being required in Chapter 3. This approach was evaluated within Chapter 6, looking at how the abstraction could be utilised by applications. It was shown that through this abstraction, it is possible to perform a range of optimisations to ensure that requirements are met. Importantly, it was shown that the necessary adaptations discovered in Chapter 3 could take place transparently to higher level applications.

To design and implement an end-to-end infrastructure that realises this (new) abstraction in a flexible manner. This goal was addressed within Chapters 4 and 5 through the design and implementation of the Juno middleware. This realised the above delivery-centric abstraction by exploiting the findings of Chapter 3 to allow the dynamic (re-)configuration between different content providers and protocols. This design was evaluated in Chapter 6, looking at how effectively it could discover content in third party systems before re-configuring itself to interact with the optimal sources. It was shown that Juno's approach could offer the performance advantages identified in Chapter 3 effectively, whilst maintaining controlled levels of overhead.

To show that this concept is feasible and can be deployed alongside existing systems in an interoperable way. This goal was addressed within Chapter 6 by thoroughly evaluating the key aspects of Juno's design. Using a prototype implementation, a number of emulated case-studies were explored to show how the advantages discovered in Chapter 3 could be realised be Juno. This involved validating Juno's ability to interoperate in a backwards compatible way with various unaware providers. It was further shown that through this ability, Juno

could dynamically be adapted to achieve delivery-centric requirements. Alongside this, simulations were performed to take a large-scale view of the content discovery process to show its performance and overhead. Collectively, these showed Juno's design to be an effective approach to addressing limitations of existing content-centric networks. Beyond these quantitative gains, it was also shown that the approach offers an effective platform for development, as well as the potential to assist greatly in the extension of content system support. These claims were further validated by the third party development of discovery protocols in Juno, which highlight how Juno's protocol interoperability can easily be extended [129].

7.8 Concluding Remarks

This thesis has identified the need for more dynamic and flexible content delivery support within the Internet. A number of delivery systems are currently used by applications, however, their ability to fulfil delivery requirements generally fluctuates both between consumers and over time. This makes optimising a global set of applications impossible without explicit runtime awareness. This thesis has proposed the abstraction of applications away from such complexities using a standardised content-centric and delivery-centric interface. This allows applications to simply generate requests for uniquely identified content, alongside any pertinent delivery requirements. The Juno middleware has been designed and implemented to offer such an interface by utilising software (re-)configuration to dynamically adapt between the use of different providers and protocols (based on their ability to fulfil given requirements). By doing this in a transparent and backwards compatible way, the abstraction can be instantly deployed for use with existing content providers.

As content distribution becomes more and more prevalent in the Internet, it is likely that the need for such technologies will increase even further. Importantly, the use of Juno offers an effective and simple way of achieving much-needed content delivery support without placing an unnecessary burden on the application developer.

Bibliography

[1] Akamai content distribution network. http://akamai.com. [cited at p. 10, 22, 32, 44, 45, 49, 61, 157]

[2] Akamai peering policy. http://www.akamai.com/peering/. [cited at p. 48]

[3] Bitcollider magnet link generator. http://sourceforge.net/projects/bitcollider/files/Bitcollider/. [cited at p. 150]

[4] Bitzi. http://bitzi.com. [cited at p. 172]

[5] Cachefly. http://www.cachefly.com. [cited at p. 48]

[6] Comet project: Content mediator architecture for content-aware networks. http://www.comet-project.org/. [cited at p. 5]

[7] The component object model (com). http://www.microsoft.com/com/. [cited at p. 101, 103, 114]

[8] The corba working group. http://www.corba.org/. [cited at p. 29]

[9] Emulab network testbed. http://Emulab.net. [cited at p. 189]

[10] Gnutella protocol 0.4. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf. [cited at p. 14, 172]

[11] Google search engine. http://www.google.co.uk. [cited at p. 55]

[12] Grub distributed search engine. http://www.gurb.org. [cited at p. 56]

[13] Harvest: A distributed search system. http://harvest.sourceforget.net. [cited at p. 56]

[14] Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html. [cited at p. 10, 19, 25]

[15] iplane datasets. http://iplane.cs.washington.edu/data.html. [cited at p. 79]

[16] Level 3 communications. http://www.level3.com. [cited at p. 48]

[17] Limelight networks. http://www.limelightnetworks.com. [cited at p. 47, 48, 62]

[18] Limewire peer-to-peer system. http://www.limewire.com. [cited at p. 15, 20]

[19] Magnet uri project. http://magnet-uri.sourceforge.net/. [cited at p. 126, 213]

[20] Microsoft .net. http://www.microsoft.com/net/. [cited at p. 103]

[21] Mininova. http://mininova.org. [cited at p. 68]

[22] Mirror image. http://www.mirror-image.com. [cited at p. 22, 48]

[23] Napster inc. http://napster.com. [cited at p. 14]

[24] Pando networks. http://www.pando.com. [cited at p. 20]

[25] Rapidshare one click hosting. http://www.rapidshare.com. [cited at p. 13, 56, 59]

[26] Rtp: A transport protocol for real-time applications. http://www.ietf.org/rfc/rfc3550.txt. [cited at p. 19, 25, 53]

[27] Shareaza peer-to-peer system. http://shareaza.sourceforge.net. [cited at p. 15, 30, 221]

[28] The soap specification. http://www.w3.org/TR/soap/. [cited at p. 30]

[29] Soap2corba. http://soap2corba.sourceforget.net. [cited at p. 27]

[30] Tor: Anonymity online. http://www.torproject.org. [cited at p. 142, 209]

[31] Trustyfiles. http://www.trustyfiles.com/. [cited at p. 30]

[32] Youtube. http://www.youtube.com. [cited at p. 17, 22, 176]

[33] Bengt Ahlgren, Matteo D'Ambrosio, Marco Marchisio, Ian Marsh, Christian Dannewitz, Börje Ohlman, Kostas Pentikousis, Ove Strandberg, René Rembarz, and Vinicio Vercellone. Design considerations for a network of information. In *CoNEXT '08: Proceedings of the 4th International Conference on Emerging Networking Experiments and Technologies*, 2008. [cited at p. 4, 12]

[34] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the internet impasse through virtualization. *Computer*, 38(4):34–41, 2005. [cited at p. 4, 26, 94, 214]

[35] Demetris Antoniades, Evangelos P. Markatos, and Constantine Dovrolis. One-click hosting services: a file-sharing hideout. In *IMC '09: Proceedings of the 9th ACM Internet Measurement Conference*, 2009. [cited at p. 19, 21, 59, 62]

[36] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 21st conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002. [cited at p. 20]

[37] Jerry Banks, John S. Carson, and Barry L. Nelson. *Discrete event system simulation*. Upper Saddle River, N. J. : Prentice Hall, 1996. [cited at p. 167, 174]

[38] Chadi Barakat, Chadi Barakat, Eitan Altman, Eitan Altman, Walid Dabbous, Walid Dabbous, and Projet Mistral. On tcp performance in an heterogeneous network: A survey. *IEEE Communications Magazine*, 38:40–46, 1999. [cited at p. 20, 59]

[39] Naimul Basher, Aniket Mahanti, Anirban Mahanti, Carey Williamson, and Martin Arlitt. A comparative analysis of web and peer-to-peer traffic. In *WWW '08: Proceeding of the 17th International Conference on World Wide Web*, 2008. [cited at p. 1]

[40] Ashwin Bharambe, Cormac Herley, and Venkat Padmanabhan. Analyzing and improving a bittorrent networks performance mechanisms. In *INFOCOM '06: Proceedings of the 25th IEEE International Conference on Computer Communications*, 2006. [cited at p. 20, 68, 69, 74, 202, 204]

[41] Andrew Brampton, Andrew MacQuire, Idris Rai, Nicholas J. P. Race, Laurent Mathy, and Michael Fry. Characterising user interactivity for sports video-on-demand. In *NOSSDAV '07: Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. [cited at p. 175]

[42] James Broberg, Rajkumar Buyya, and Zahir Tari. Metacdn: Harnessing 'storage clouds' for high performance content delivery. *Networked Computer Applications*, 32(5):1012–1022, 2009. [cited at p. 23]

[43] Brad Cain, Oliver Spatscheck, Kobus van de Merwe, Lisa Amini, Aabbie Barbir, Martin May, and Delphine Kaplan. Content network advertisement protocol (cnap). IETF Draft, 2002. [cited at p. 26]

[44] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 22nd Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003. [cited at p. 17]

[45] Maureen Chesire, Alec Wolman, Geoffrey M. Voelker, and Henry M. Levy. Measurement and analysis of a streaming-media workload. In *USITS '01: Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, 2001. [cited at p. 176]

[46] David R. Choffnes and Fabián E. Bustamante. Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems. *SIGCOMM Computer Communications Review*, 38(4):363–374, 2008. [cited at p. 21]

[47] Jaeyoung Choi, Jinyoung Han, Eunsang Cho, Hyunchul Kim, Taekyoung Kwon, and Yanghee Choi. Performance comparison of content-oriented networking alternatives: A tree versus a distributed hash table. In *LCN '09: Proceedings of the 34th IEEE Conference on Local Computer Networks*, 2009. [cited at p. 34, 35]

[48] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies*, 2001. [cited at p. 5]

[49] Bram Cohen. Incentives build robustness in bittorrent. In *EP2P '03: Proceedings of the Workshop on the Economics of Peer-to-Peer Systems*, 2003. [cited at p. 5, 20, 25, 66]

[50] George F. Coulouris and Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. [cited at p. 29]

[51] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42, 2008. [cited at p. 29, 103, 104]

[52] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of www client-based traces. Technical report, Boston University, 1995. [cited at p. 176]

[53] Mark Day, Ned Freed, and Patrik Faltstrom. Content distribution internetworking. RFC 3570, 2002. [cited at p. 26]

[54] Michael Demmer, Bowei Du, Andrey Ermolinsky, Kevin Fall, Igor Ganichev, Teemu Koponen, Junda Liu, Scott Shenker, and Arsalan Tavakoli. A publish/subscribe communications api for data-oriented applications. In *NSDI '09: Proceedings of the 5th Symposium on Networked Systems Design and Implementation*, 2009. [cited at p. 2, 4, 10, 11, 94, 124, 252]

[55] Mayur Deshpande, Abhishek Amit, Mason Chang, Nalini Venkatasubramanian, and Sharad Mehrotra. Flashback: A peer-to-peerweb server for flash crowds. In *ICDCS '07: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems*, 2007. [cited at p. 202]

[56] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *IMC '07: Proceedings of the 7th ACM Internet Measurement Conference*, 2007. [cited at p. 68, 70, 75, 79, 171, 184]

[57] Miklos Telek Dragana Damjanovic, Michael Welzl and Werner Heiss. Extending the tcp steady-state throughput equation for parallel tcp flows. Technical report, 2008. [cited at p. 66]

[58] Martin Dunmore. An ipv6 deployment guide. Technical report, 6Net Consortium, 2005. [cited at p. 26, 27]

[59] Yehia El-khatib, Christopher Edwards, Michael Mackay, and Gareth Tyson. Providing grid schedulers with passive network measurements. In *ICCCN '09: Proceedings of the 18th IEEE International Conference on Computer Communications and Networks*, 2009. [cited at p. 64]

[60] Jeffrey Erman, Anirban Mahanti, and Martin Arlitt. Internet traffic identification using machine learning. In *GLOBECOM '06: Proceedings of the 7th IEEE Global Communications Conference*, 2006. [cited at p. 1]

[61] Jarret Falkner, Michael Piatek, John P. John, Arvind Krishnamurthy, and Thomas Anderson. Profiling a million user dht. In *IMC '07: Proceedings of the 7th ACM Internet Measurement Conference*, 2007. [cited at p. 16, 172, 178]

[62] Bin Fan, Dah-ming Chiu, and John Lui. The delicate tradeoffs in bittorrent-like file sharing protocol design. In *ICNP '06: Proceedings of the 14th IEEE International Conference on Network Protocols*, 2006. [cited at p. 20, 74, 202]

[63] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI '04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, 2004. [cited at p. 10]

[64] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, 2003. [cited at p. 103]

[65] Panos Georgatsos, Jason Spencer, David Griffin, Takis Damilatis, Hamid Asgari, Jonas Griem, George Pavlou, and Pierrick Morand. Provider-level service agreements for inter-domain qos delivery. In *ICQT '04: Proceedings of the 4th International Workshop on Advanced Internet Charging and QoS Technologies*, 2004. [cited at p. 48]

[66] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. The flattening internet topology: Natural evolution, unsightly barnacles or contrived collapse? In *PAM '08: Proceedings of the 9th Passive and Active Measurements Workshop*. [cited at p. 23]

[67] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *IMC '07: Proceedings of the 7th ACM Internet Measurement Conference*, 2007. [cited at p. 176]

[68] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mobile Computer Communicaions Review*, 9(1):2–14, 2005. [cited at p. 29, 221]

[69] Paul Grace, Danny Hughes, Barry Porter, Gordon S. Blair, Geoff Coulson, and Francois Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008. [cited at p. 29]

[70] Object Management Group. Dcom/corba interworking specification part a and b, 1997. [cited at p. 27]

[71] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, 2005. [cited at p. 35]

[72] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating Systems Principles*, 2003. [cited at p. 174, 176, 181, 183, 184, 253]

[73] Lei Guo, Songqing Chen, Zhen Xiao, Enhua Tan, Xiaoning Ding, and Xiaodong Zhang. Measurements, analysis, and modeling of bittorrent-like systems. In *IMC '05: Proceedings of the 5th ACM Internet Measurement Conference*, 2005. [cited at p. 22, 74]

[74] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *PODC '08: Proceedings of the 27th ACM Symposium on the Principles of Distributed Computing*, 2008. [cited at p. 175, 212]

[75] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: a scalable overlay network with practical locality properties. In *USITS '03: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003. [cited at p. 16]

[76] Qi He, Constantinos Dovrolis, and Mostafa Ammar. On the predictability of large transfer tcp throughput. *Computer Networks*, 51(14):3959–3977, 2007. [cited at p. 86]

[77] Cheng Huang, Angela Wang, Jin Li, and Keith W. Ross. Measuring and evaluating large-scale cdns. In *IMC '08: Proceedings of the 8th ACM Internet Measurement Conference*, 2008. [cited at p. 44, 45, 249]

[78] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6):1, 2005. [cited at p. 20]

[79] Thomas Idal, Michael Piatek, Arvind Krishnamurthy, and Thomas Anderson. Leveraging bittorrent for end host measurements. In *PAM '07: Proceedings of the 8th Passive and Active Measurements Workshop*, 2007. [cited at p. 76, 79, 160]

[80] Arun K. Iyengar, Mark S. Squillante, and Li Zhang. Analysis and characterization of large-scale web server access patterns and performance. *World Wide Web*, 2(1-2):85–100, 1999. [cited at p. 57, 59]

[81] Van Jacobson. A new way to look at networking. http://video.google.com/videoplay?docid=-6972678839686672840. [cited at p. 2, 44]

[82] Van Jacobson, Marc Mosko, Diana K. Smetters, and Jose Garcia-Luna-Aceves. Content-centric networking. *Whitepaper, Palo Alto Research Center*, 2007. [cited at p. 2, 4, 38]

[83] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *CoNEXT '09: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, 2009. [cited at p. 10, 11, 32, 38, 41, 42, 43, 96, 126, 146, 147]

[84] Hosagrahar V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB '05: Proceedings of the 31st International Conference on Very large databases*, 2005. [cited at p. 16]

[85] Petri Jokela, Andras Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: line speed publish/subscribe inter-networking. *SIGCOMM Computer Communications Review*, 39(4):195–206, 2009. [cited at p. 11, 17]

[86] Philippe Joubert, Robert B. King, Richard Neves, Mark Russinovich, and John M. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *USENIX '02: Proceedings of the 7th USENIX Annual Technical Conference*, 2002. [cited at p. 57]

[87] Sebastian Kaune, Konstantin Pussep, Gareth Tyson, Andreas Mauthe, and Ralf Steinmetz. Cooperation in p2p systems through sociological incentive patterns. In *IWSOS '08: Proceedings of the 3rd International Workshop on Self-Organizing Systems*, 2008. [cited at p. 20, 61]

[88] Sebastian Kaune, Ruben Cuevas Rumin, Gareth Tyson, Andreas Mauthe, Carmen Guerrero, and Ralf Steinmetz. Unraveling bittorrent's file unavailability: Measurements and analysis. In *P2P '10: Proceedings of the 10th IEEE International Conference on Peer-to-Peer Systems*, 2010. [cited at p. 22, 69, 73, 74, 209]

[89] Sebastian Kaune, Jan Stolzenburg, Aleksandra Kovacevic, and Ralf Steinmetz. Understanding bittorrent's suitability in various applications and environments. In *ComP2P '08: Proceedings of the 3rd International Multi-Conference on Computing in the Global Information Technology*, 2008. [cited at p. 21]

[90] Sebastian Kaune, Gareth Tyson, Konstantin Pussep, Andreas Mauthe, and Ralf Steinmetz. The seeder promotion problem: Measurements, analysis and solution space. In *ICCCN '10: Proceedings of 19th IEEE International Conference on Computer Communications and Networks*, 2010. [cited at p. 86]

[91] Tor Klingberg and Raphael Manfredi. Gnutella 0.6 protocol. Technical report, Network Working Group, 2002. [cited at p. 15, 20, 82]

[92] Simon Koo, Catherine Rosenberg, and Dongyan Xu. Analysis of parallel downloading for large file distribution. In *FTDCS '03: Proceedings of the The 9th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2003. [cited at p. 5]

[93] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. *SIGCOMM Computer Communications Review*, 37(4):181–192, 2007. [cited at p. 2, 4, 11, 17, 32, 35, 96, 126, 146, 149]

[94] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *IMC '09: Proceedings of the 9th ACM Internet Measurement Conference*, 2009. [cited at p. 17, 62]

[95] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *NSDI '07: Proceedings of the 4th USENIX symposium on Networked Systems Design and Implementation*, 2007. [cited at p. 61]

[96] Junsoo Lee, Joo P. Hespanha, and Stephan Bohacek. A study of tcp fairness in high-speed networks. Technical report, University of Southern California, 2005. [cited at p. 57]

[97] Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. Clustering and sharing incentives in bittorrent systems. In *SIGMETRICS '07: Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2007. [cited at p. 74]

[98] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the kazaa network. In *WIAPP '03: Proceedings of the The 3rd IEEE Workshop on Internet Applications*, 2003. [cited at p. 15]

[99] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. Bittorrent is an auction: analyzing and improving bittorrent's incentives. In *SIGCOMM '08: Proceedings of the 27th International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.*, 2008. [cited at p. 67]

[100] Chung-Sheng Li, Yueh-Min Huang, and Han-Chieh Chao. Upnp ipv4/ipv6 bridge for home networking environment. *IEEE Transactions on Consumer Electronics*, 54(4):1651–1655, November 2008. [cited at p. 27]

[101] Wei-Cherng Liao, Fragkiskos Papadopoulos, and Konstantinos Psounis. Performance analysis of bittorrent-like systems with heterogeneous users. *Performance Evaluation*, 64(9-12):876–891, 2007. [cited at p. 71, 74]

[102] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: an information plane for distributed services. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. [cited at p. 64, 65, 159, 249]

[103] Harsha V. Madhyastha, Ethan Katz-Bassett, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane nano: path prediction for peer-to-peer applications. In *NSDI '09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009. [cited at p. 65]

[104] Nazanin Magharei and Reza Rejaie. Prime: peer-to-peer receiver-driven mesh-based streaming. *IEEE/ACM Transactions on Networking*, 17(4):1052–1065, 2009. [cited at p. 21]

[105] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Computer Communications Review*, 27(3):67–82, 1997. [cited at p. 63]

[106] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, 2001. [cited at p. 16, 149, 172, 178]

[107] Nancy Miller and Peter Steenkiste. Collecting network status information for network-aware applications. In *INFOCOM '00. Proceedings of the 19th IEEE International Conference on Computer Communications*, 2000. [cited at p. 64]

[108] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. [cited at p. 40]

[109] Rob C. van Ommering. Koala, a component model for consumer electronics product software. In *ARES '98: Proceedings of the 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, 1998. [cited at p. 103]

[110] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp throughput: A simple model and its empirical validation. Technical report, University of Massachusetts, 1998. [cited at p. 63, 159]

[111] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, 2000. [cited at p. 63, 65]

[112] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *IPTPS '01: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, 2001. [cited at p. 59, 202]

[113] J. Parker. Recommendations for interoperable ip networks using intermediate system to intermediate system (is-is). RFC 3787, 2004. [cited at p. 40]

[114] Ryan S. Peterson and Emin Gün Sirer. Antfarm: efficient content distribution with managed swarms. In *NSDI '09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009. [cited at p. 71]

[115] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent? In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007. [cited at p. 75, 76, 77, 78, 79, 80, 81, 160, 250, 252]

[116] Thomas Plagemann, Vera Goebel, Andreas Mauthe, Laurent Mathy, Thierry Turletti, and Guillaume Urvoy-Keller. From content distribution networks to content networks - issues and challenges. *Computer Communications*, 29(5):551 – 562, 2006. [cited at p. 2, 9, 12]

[117] Lili Qiu, Yin Zhang, and Srinivasan Keshav. On individual and aggregate tcp performance. In *ICNP '99: Proceedings of the 7th International Conference on Network Protocols*, 1999. [cited at p. 64]

[118] Jarno Rajahalme, Mikko Särelä, Pekka Nikander, and Sasu Tarkoma. Incentive-compatible caching and peering in data-oriented networks. In *CoNEXT '08: Proceedings of the 4th International Conference on Emerging Networking Experiments and Technologies*, 2008. [cited at p. 12]

[119] Radhakrishna Rao and Herman Rubin. On a characterization of the poisson distribution. Technical report, Defense Technical Information Center OAI-PMH Repository, 1998. [cited at p. 171]

[120] Amir Rasti and Reza Rejaie. Understanding peer-level performance in bittorrent: A measurement study. In *ICCCN '07: Proceedings of 16th IEEE International Conference on Computer Communications and Networks*, 2007. [cited at p. 75]

[121] Pierre-Guillaume Raverdy, Oriana Riva, Agnes de La Chapelle, Rafik Chibout, and Valerie Issarny. Efficient context-aware service discovery in multi-protocol pervasive environments. In *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*, 2006. [cited at p. 27]

[122] Dan Rayburn. Pricing pressure on akamai and limelight overblown by analysts. http://seekingalpha.com/article/53938-pricing-pressure-on-akamai-and-limelight-overblown-by-analysts. [cited at p. 23]

[123] Yakov Rekhter and Tony Li. A border gateway protocol 4. RFC 1771, 1995. [cited at p. 25, 38, 40]

[124] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-osgi: distributed applications through software modularization. In *Middleware '07: Proceedings of the 8th ACM/IFIP/USENIX 2007 International Conference on Middleware*, 2007. [cited at p. 29]

[125] Jordan Ritter. Why gnutella can't scale, no really. [cited at p. 16, 82]

[126] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms*, 2001. [cited at p. 16, 35]

[127] Osama Saleh and Mohamed Hefeeda. Modeling and caching of peer-to-peer traffic. In *ICNP '06: Proceedings of the 14th IEEE International Conference on Network Protocols*, 2006. [cited at p. 176]

[128] Hendrik Schulze and Klaus Mochalski. Ipoque internet study. Technical report, ipoque GmbH, 2007. [cited at p. 1, 20, 53, 66, 127, 250]

[129] Magnus Skjegstad, Utz Roedig, and Frank T. Johnsen. Search+: A resource efficient peer-to-peer service discovery mechanism. In *MILCOM '09: Proceedings of the 28th IEEE Military Communications Conference*, 2009. [cited at p. 143, 215, 222, 227]

[130] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *SIGCOMM '02: Proceedings of the 21st Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002. [cited at p. 32, 125]

[131] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 20th Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001. [cited at p. 16]

[132] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM Internet Measurement Conference*, 2006. [cited at p. 172, 179, 184]

[133] Ao-Jan Su and Aleksandar Kuzmanovic. Thinning akamai. In *IMC '08: Proceedings of the 8th ACM Internet Measurement Conference*, 2008. [cited at p. 46]

[134] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. [cited at p. 102]

[135] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002. [cited at p. 29]

[136] Duc A. Tran, Kien A. Hua, and Tai Do. Zigzag: an efficient peer-to-peer scheme for media streaming. In *INFOCOM '03: Proceeding of the 22nd IEEE Conference of Computer and Communications*, 2003. [cited at p. 20]

[137] Elisa Turrini. *An Architecture for Content Distribution Internetworking*. PhD thesis, Univeristy of Bologna, Italy, 2004. [cited at p. 17, 26]

[138] Gareth Tyson, Paul Grace, Andreas Mauthe, and Gordon Blair. A reflective middleware to support peer-to-peer overlay adaptation. In *DAIS '09: Proceedings of the 9th IFIP International Conference on Distributed Applications and Interoperable Systems*, 2009. [cited at p. 101]

[139] Gareth Tyson, Paul Grace, Andreas Mauthe, and Sebastian Kaune. The survival of the fittest: an evolutionary approach to deploying adaptive functionality in peer-to-peer systems. In *ARM '08: Proceedings of the 7th Workshop on Reflective and Adaptive Middleware*, 2008. [cited at p. 225]

[140] Gareth Tyson, Andreas Mauthe, Sebastian Kaune, Mu Mu, and Thomas Plagemann. Corelli: A peer-to-peer dynamic replication service for supporting latency-dependent content in community networks. In *MMCN '09: Proceedings of the 16th Annual Multimedia Computing and Networking*, 2009. [cited at p. 12, 225]

[141] Gareth Tyson, Andreas Mauthe, Thomas Plagemann, and Yehia El-khatib. Juno: Reconfigurable middleware for heterogeneous content networking. In *NGNM '08: Proceedings of the 5th International Workshop on Next Generation Networking Middleware*, 2008. [cited at p. 93]

[142] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Operating Systems Review*, 36(SI), 2002. [cited at p. 167]

[143] Eveline Veloso, Virgílio Almeida, Wagner Meira, Azer Bestavros, and Shudong Jin. A hierarchical characterization of a live streaming media workload. In *IMW '02: Proceedings of the 2nd ACM Workshop on Internet Measurments*, 2002. [cited at p. 176]

[144] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proceedings of the 3rd USENIX symposium on Operating systems Design and Implementation*, 2002. [cited at p. 56, 190]

[145] Richard Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, 1997. [cited at p. 64, 65]

[146] Bo Yang. Evaluation of akamai's server allocation scheme. `http://gregorio.stanford.edu/~byang/pub/akamai/html/akamai.htm`. [cited at p. 157]

[147] Hongliang Yu, Dongdong Zheng, Ben Y. Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. *SIGOPS Operating Systems Review*, 40(4):333–344, 2006. [cited at p. 57, 58, 176, 183, 184, 212, 253]

[148] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. Yum. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM '05. Proceeding of the 24th IEEE Conference of Computer and Communications*, 2005. [cited at p. 20, 21]

[149] Yin Zhang, Paxson, Vern, and Scott Shenker. The stationarity of internet path properties: routing, loss, and throughput. Technical report, AT&T Center for Internet Research, 2000. [cited at p. 65]

[150] Yongguang Zhang, Elliot Yan, and Son K. Dao. A measurement of tcp over long-delay network. In *ICTS '98: Proceedings of the 6th International Conference on Telecommunication Systems*, 1998. [cited at p. 60]

[151] Shanyu Zhao, Daniel Stutzbach, and Reza Rejaie. Characterizing files in the modern gnutella network: A measurement study. In *MMCN '06: Proceedings of the 16th SPIE/ACM Multimedia Computing and Networking Conference*, 2006. [cited at p. 85, 174]

Appendices

Appendix A

System Overheads

This appendix provides a set of overhead measurements taken from the Java implementation of Juno. It focusses on the costs of running the system in a component-based manner, alongside issues such as configuration times. The experiments are all executed on an Intel Core 2 Duo 2.1 GHz PC with 4 GB RAM; running Ubuntu 8.04 and the OpenJDK JRE 1.6.

A.1 Component Overheads

This section measures the overheads introduced by using components in Juno. Specifically, this relates to extra memory and processing requirements placed on the system. This is incurred due to the need to manage the explicit bindings that exist between the components.

A.1.1 Construction Time

The first overhead measured is the construction time. The usage of components introduces certain extra processes that must be performed during the initiation (e.g. manipulating the system graph).

To test this, a dummy object, a dummy OpenCOM component and a dummy Juno component are created. They do not contain any functionality apart from that necessary to be instantiated. The time taken to construct each type is then measured 100 times; the first part of Table A.1 shows the results. It is immediately evident that there is a performance disadvantage from using OpenCOM components. This is even worse for constructing Juno components, which possess extra functionality. On average, it takes 805 μs to construct a Juno component, which can be compared to an average time of 5 μs for an equivalent Java object.

	Min (μs)	Average (μs)	Max (μs)
Empty Java Class	1	5	342
Empty OpenCOM Component	136	480	16,620
Empty Juno Component	216	805	34,536
HTTP Java Class	1	37	3,591
HTTP OpenCOM Component	213	884	30,911
HTTP Juno Component	471	1498	70,384

Table A.1: Comparison of Java object, OpenCOM Component and Juno Component Construction Times

The previous tests were performed using empty components without any functionality. This, however, ignores the costs associated with loading the necessary code. To investigate this, three forms of the HTTP delivery plug-in are created. The first places the functionality in solely a Java object; this does not possess any Juno/OpenCOM functionality such as dependency management or reflective interfaces. The second places the functionality in solely an OpenCOM component; this does not possess any of the Juno functionality such as open state management, event subscription or advanced life cycle control. The third is simply the original Juno component implementation. The second part of Table A.1 shows the results. It is clear that the difference between the components and objects lowers; with dummy functionality, a native Java object takes 0.2% of the time it takes to construct an OpenCOM component. Instead, it now takes 4% of the time; although this difference is limited, it is evident that the cost of loading the contents of the object has an impact.

A.1.2 Processing Throughput

The next overhead measured is processing throughput; this represents the extra cost incurred by components that interact with each other through dynamic bindings (i.e. receptacles). This is measured by benchmarking two identical classes that contain five dummy methods containing no code and varying numbers of parameters of type int. The first test performs invocations on the class using direct Java method calls, whilst the second test performs invocations using OpenCOM receptacles. To ascertain the maximum invocation throughput, both tests are performed using a while loop that cycles for one second whilst calling the method on every round.

Table A.2 shows the number of invocations per second for both direct Java method invocations and Juno component invocations. Clearly, there is a decrease in invocation throughput for all experiments when utilising receptacles. This can also be observed in Table A.3, which shows the average execution time for a single invocation. There is very little difference, however, in the processing throughput

between the various number of parameters passed in, with a steady decrease of ≈36%.

Parameters	Receptacle Invocations per Sec	Direct Java Invocations per Sec	Overhead
0	1210814	1266299	55485
1	788927	1252342	463415
2	806245	1260852	454607
3	801094	1253439	452345
4	800208	1259498	459290
5	800983	1263520	462537

Table A.2: Invocations per/sec for OpenCOM Components using Different Numbers of Parameters

Parameters	Receptacle Execution Time (ns)	Direct Java Execution Time (ns)	Overhead
0	825	790	35
1	1267	790	477
2	1240	793	447
3	1248	797	451
4	1249	793	456
5	1248	791	457

Table A.3: Execution Times for OpenCOM Components using Different Numbers of Parameters

A.1.3 Memory Costs

The next overhead is the memory costs of implementing components, as opposed to native Java objects. To create a benchmark, an individual OpenCOM runtime is initiated without any components; this consumes 5592 bytes of memory. Table A.4 subsequently shows the memory overhead associated with implementing functionality in an OpenCOM component when compared to a native Java class. This shows a range of empty components that are associated with different numbers of receptacles and interfaces.

These results constitute measurements of purely the OpenCOM component model. However, a number of extensions have been developed that turn an OpenCOM component into a Juno component. This involves such things as state management, parameter management and various interactions with the framework. To enable this, all components in Juno extend the JunoComponent abstract class (c.f. Section 4.4.2). To measure the added cost of this, a dummy Juno component

Module	OpenCOM Component	Java Class	Overhead
One (1 intf, 0 recps)	990	623	367
Two (2 intf, 0 recps)	1703	1307	396
Three (3 intf, 0 recps)	2123	1703	420
Four (1 intf, 1 recps)	2999	2051	941
Five (1 intf, 2 recps)	3299	2051	1248
Six (1 intf, 3 recps)	3555	2051	1504

Table A.4: Memory Overhead of OpenCOM Components (in Bytes)

is instantiated without any functionality. The tests show integrating the Juno code creates a 2384 byte overhead.

A.2 Configuration Overheads

This section measures the overhead of building Juno in a (re-)configurable manner. Specifically this relates to the processing delay associated with building component configurations.

A.2.1 Configuration Time

Table A.5 details the configuration time for creating various configurations. This process does not involve the generation of meta-data or any distributed interactions. Instead, it is simply the creation of a Configurator and the loading of the necessary components. These results can be contrasted with the results from Table A.1, which show that the construction of components takes place in nano seconds. As such, it is evident that Juno does introduce a degree of overhead when creating configurations.

Configuration	Instantiation Time (ms)
HTTP	35
BitTorrent (HBPTC)	126
BitTorrent (Snark)	95
Limewire	42
BitTorrent Tracker (HBPTC)	349
BitTorrent Tracker (Snark)	357

Table A.5: Average Instantiation Time for Configurations

A.2.2 Connection Delay

The previous section has detailed the overall configuration time. This consists of loading components and then interconnecting them. Section A.1.1 details the loading overhead. This section now measures the overhead of interconnecting components. To study this, Table A.6 details the minimum, maximum and average time taken to connect two components. Evidently, the delay is tiny compared to the time required to instantiate a single component.

Min (ns)	Average (ns)	Max (ns)
938	1,190	1,327

Table A.6: Connection Time for Interconnecting Two Components

A.3 Framework Overheads

The previous sections have inspected Juno's general approach to design, i.e. the use of runtime components. This section now details the Juno-specific overheads incurred by utilising the current Java implementation.

A.3.1 Bootstrapping Delay

The first overhead considered is the time required to initiate each framework. This consists of both the memory load time, as well as any further processing required. To measure this, each framework is bootstrapped and the delay recorded; Table A.7 details the results.

Framework	Bootstrap Time (ms)
Content Manager	10
Discovery Framework	27
Delivery Framework	17
Configuration Engine	163

Table A.7: Average Bootstrapping Times for Frameworks

A.3.2 Memory Overhead

Table A.8 shows the memory footprints of the different frameworks in Juno. These were measured without any plug-ins attached using the runtime stack memory; consequently, they do not include the overhead of the Java Virtual Machine (JVM). In contrast, Table A.9 also shows the full memory footprints (including JVM) of Juno with various plug-ins configured.

Framework	Memory Footprint
Content-Centric Framework	15 KB
Content Manager	3.2 KB
Discovery Framework	44.7 KB
Delivery Framework	44.6 KB

Table A.8: Runtime Memory Footprint of Frameworks (exc. JVM)

Configuration	Memory Footprint
Empty	472 KB
HTTP	512 KB
BitTorrent	522 KB
Limewire	573 KB

Table A.9: Runtime Memory Footprint of Configurations (inc. JVM)

A.4 Magnet Link Generation Overhead

The last overhead measured is that of content identifier generation. Due to the cooperative nature of the Juno Content Discovery Service, this is a process that would be performed by all consumers. Table A.10 provides an overview of the processing time required to build the necessary hash identifiers for a variety of popular delivery systems; specifically, for BitTorrent, Kazaa and eD2K.

	Music (4.2MB)	Music (9.6MB)	Cartoon (72MB)	TV Show (350MB)	Movie (720MB)
Time (sec)	5	5	9	20	26

Table A.10: Processing Time of Content Identifier Generation

List of Figures

List of Tables