OpenCache: A Content Delivery Platform for the Modern Internet

Matthew Broadbent

This dissertation is submitted for the degree of Doctor of Philosophy

Abstract

Since its inception, the World Wide Web has revolutionised the way we share information, keep in touch with each other and consume content. In the latter case, it is now used by thousands of simultaneous users to consume video, surpassing physical media as the primary means of distribution. With the rise of on-demand services and more recently, high-definition media, this popularity has not waned. To support this consumption, the underlying infrastructure has been forced to evolve at a rapid pace. This includes the technology and mechanisms to facilitate the transmission of video, which are now offered at varying levels of quality and resolution.

Content delivery networks are often deployed in order to scale the distribution provision. These vary in nature and design; from third-party providers running entirely as a service to others, to in-house solutions owned by the content service providers themselves. However, recent innovations in networking and virtualisation, namely Software Defined Networking and Network Function Virtualisation, have paved the way for new content delivery infrastructure designs. In this thesis, we discuss the motivation behind OpenCache, a next-generation content delivery platform. We examine how we can leverage these emerging technologies to provide a more flexible and scalable solution to content delivery. This includes analysing the feasibility of novel redirection techniques, and how these compare to existing means. We also investigate the creation of a unified interface from which a platform can be precisely controlled, allowing new applications to be created that operate in harmony with the infrastructure provision. Developments in distributed virtualisation platforms also enables functionality to be spread throughout a network, influencing the design of OpenCache. Through a prototype implementation, we evaluate each of these facets in a number of different scenarios, made possible through deployment on large-scale testbeds.

Declaration

I declare that the work in this thesis has not been submitted for a degree at any other university, and that the work is entirely my own.

Matthew Broadbent
December, 2015

Acknowledgements

First and foremost, I would like to express gratitude to my PhD supervisor, Dr. Nicholas Race. Throughout my postgraduate study, he has offered me the support and guidance necessary for me to succeed as both a researcher and a scientist. My thanks also go to those that I have worked with during this period, including Panagiotis Georgopoulos, Mu Mu and Arsham Farshad.

I would like to thank those that have kept me sane over the past few years, including my good friends, Oliver Bates and Richard Withnell, and my parents, Stephen and Jane Broadbent. The encouragement and motivation offered by these people has been unfaltering.

Contributing Publications

M. Broadbent, P. Georgopoulos, V. Kotronis, B. Plattner, and N. Race. OpenCache: Leveraging SDN to demonstrate a customisable and configurable cache. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pages 151–152. IEEE, 2014.

M. Broadbent, D. King, S. Baildon, N. Georgalas, and N. Race. OpenCache: A software-defined content caching platform. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5. IEEE, 2015.

M. Broadbent and N. Race. OpenCache: exploring efficient and transparent content delivery mechanisms for video-on-demand. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 15–16. ACM, 2012.

P. Georgopoulos, M. Broadbent, A. Farshad, B. Plattner, and N. Race. Using Software Defined Networking to enhance the delivery of Video-on-Demand. *Computer Communications*, 69:79–87, 2015.

P. Georgopoulos, M. Broadbent, B. Plattner, and N. Race. Cache as a service: leveraging SDN to efficiently and transparently support Video-on-Demand on the last mile. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–9. IEEE, 2014.

Other Publications

O. Bates and M. Broadbent. HomeFlow: inferring device usage with network traces. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 815–820. ACM, 2013.

A. Farshad, P. Georgopoulos, M. Broadbent, M. Mu, and N. Race. Leveraging SDN to provide an in-network QoE measurement framework. In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, pages 239–244, April 2015.

T. Fratczak, M. Broadbent, P. Georgopoulos, and N. Race. Homevisor: Adapting home network environments. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 32–37. IEEE, 2013.

P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race. Towards network-wide QoE fairness using openflow-assisted adaptive video streaming. In *Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking*, pages 15–20. ACM, 2013.

S. Nazir, Z. Hossain, R. Secchi, M. Broadbent, A. Petlund, and G. Fairhurst. Performance evaluation of congestion window validation for DASH transport. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, page 67. ACM, 2014.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has become an integral part of many people's lives. From its humble beginnings as a research network, it is now the primary means by which many people keep in contact with each other, stay abreast of current events, and consume their favourite TV shows and films. In the latter case, it has now surpassed physical media as the preferred way in which to deliver video.

This change in consumption is in part due to improvements in the availability of Internet access. As this has grown to encompass much of the general population, many of the aforementioned services have become accessible to a large number of people. Matched with a simultaneous improvement in last-mile connectivity, this has created an environment in which the distribution of high-quality video to the masses is both technically possible and financially viable.

To support the widespread consumption of video over the Internet, different technologies and methods have been employed over recent years. This includes the deployment of dedicated platforms to enable the necessary scale and availability to be realised, such as content delivery networks. The techniques used to deliver video to client devices have also seen recent advancements, especially towards adaptive streaming technologies.

In parallel, there has been a significant move towards open and programmable infrastructures. These range from the highly-configurable networks, to the ability to dynamically control compute and storage platforms to enable the flexible provision of resources.

1.1 Content Delivery in the Modern Internet

The Internet is now the primary conduit through which the general population consumes video. This is apparent in the overall traffic profile of the Internet, which is now dominated by video [46].

The scale and support necessary to deliver this video is considerable, and relies heavily on investment in both network and service infrastructure. This has been developed over numerous years, and has often resulted in the need to evolve in the many areas contained within the content delivery ecosystem.

Furthering Network Capacity

The underlying computer network is a core part of this; without sufficient capacity, congestion and latency manifest themselves as visual defects exhibited during playback. By expanding the amount of traffic that a network can carry, more users and services can be simultaneously consumed. Network expansion also delivers increased throughput, which can be used to deliver higher quality video. Considering the ongoing shift from standard definition to high definition media, and the not-so-distant move to ultra high definition, capacity needs to continually increase to match consumer demands and expectations.

However, provisioning networks, especially those that involve the physical installation of equipment and cabling, is a time and cost intensive process. A gradual approach is often taken, with long-lived planning cycles allowing replacement and upgrade to occur every few years. There is therefore an element of prediction involved in this process, as operators need to provision for expected future demand.

Matching Consumer Demand

Naturally, demand can often outstrip supply in these cases. This is especially true when rapid, unexpected growth occurs. The unprecedented popularity of video delivery over the Internet would be an example of such. To supplement the expansion of network capacity, content providers often rely upon supporting infrastructure to solve some of the scalability and performance requirements of delivering content to a great number of users.

These deployments can take many forms, but share commonality in their ability to replicate content over a number of locations. By doing so, they provide assurance that content is always available, regardless of the number of requests. They also aim to minimise the amount of network hops necessary to fetch the content, which should reduce the chance of encountering congestion.

Further tenants in this environment include the underlying delivery technologies used to get content to a requesting user. With an increase in the quality of available video, it became viable to offer multiple tiers to customers. Yet, clients cannot always receive the highest tier of quality; network impairments are sometimes unavoidable, regardless of the infrastructures in place. For example, a client transitioning in a mobile context cannot guarantee a constant level of throughput. A solution that adapts to network and device conditions adaptive solution is therefore necessary.

Managing Competing Interests

Operating within this environment are a number of different parties, each with different goals and motivations. Commercial content service providers are driven by either paid subscription or advertising revenue models. Regardless of their funding, these services commonly offer vast content libraries that can be accessed any time of day, and on a multitude of devices. Yet, as the range and depth of these services expands, customer expectations also increase. Users have now come to expect the availability of vast content libraries, as well as higher quality video delivered with minimal initial delay. Users also have a much lower tolerance to impairments during playback, with even a minor disturbance having a measurable impact on the quality of their experience. In the case of commercial service providers, consumers will often take their custom elsewhere if they are unhappy with the service they are receiving.

The burden of delivering content does not rest solely with the content service provider. Last-mile access networks are typically the final hop before a user's device, and as such, they play an important role in determining the perceived user experience. However, these networks receive no direct remuneration from the content service provider for delivering their traffic, and offer a best-effort service as a result. When it is considered that this video traffic has now become the predominate type carried by these networks, the need to expand network

capacity is essentially driven by the proliferation of these services; yet they receive no return on the profits generated by such.

To ensure that content is located in the best possible location given this situation, content service providers often employ the services of content distribution networks. These entities place content in strategic locations, typically at the edge of these access networks, to ensure that the impact of network conditions and impairments is reduced to those within the network itself. They achieve this by peering with the access networks and replicating content objects within their own infrastructure. Requests for content originating from the access network are then redirected towards these copies. In addition to this, content delivery networks also guarantee the availability of content without sole reliance on content service provider provision. This scale is necessary to meet the increasing demand generated by the popularity of content, particularly in cases where this demand is sudden and unexpected.

1.2 The Move Towards Programmability

For many years, there has been a trend towards creating programmability in previously fixed appliances and infrastructure. This has seen significant research in networking and virtualisation. Driven by the need to flexibly control the behaviour of these elements, great effort has been expended towards achieving such a goal. However, many attempts to address this have been met by concerns of scalability, performance and security.

More recently, *software defined networking* has been coined as a term to represent networks which are capable of being controlled through distinct standalone applications. In particular, it defines the process of decoupling control and forwarding functionality. By doing this, control is granted to a controller application, which then has the ability to define the behaviour of a forwarding element in response to incoming packets. This controller can also accept responsibility for multiple such entities, facilitating the control of an entire network from a centralised and unified location.

These concepts were embodied in a number of technologies, each of which aiming to successfully realise this functionality. Unlike previous attempts, industry followed academic trends and began manufacturing equipment compliant with

these specifications. This fast-tracked research in the area, and allows novel functionalities to be built quickly and evaluated rapidly. It also provides a migration path between lab-based experimentation and real-world deployment, something that was previously hard to realise, and which resulted in only a select few innovations seeing widespread usage.

A similar process has taken place in the field of virtualisation, and more specifically, in the virtualisation of computing machinery. This field also has a long history, but recent progress in platform design has enabled users to dynamically build, modify and destroy virtual machine instances as desired. Furthermore, these platforms also provide the supporting infrastructure for these new instances, including storage, networking and monitoring capabilities. This is tied with the ability to programmatically control these platforms; that is, perform these actions from within an application using a well-defined interface.

Together, these advancements have enabled a new generation of infrastructure-assisted applications, capable of modifying the behaviour of various elements of their own infrastructure where necessary. This allows new forms of flexibility, and paves the way for the efficient usage, sharing and allocation of resources. Importantly, the actions necessary to accomplish this can be achieved in almost real-time, enabling reactive behaviours in response to service load, availability of resources and the cost of operating.

1.3 Motivation

The growth in demand for content of all forms, and in particularly that of video, shows no sign of stopping. In order to satiate this demand, content service providers use various approaches to ensure that video is not only available, but delivered in a way which satisfies the increasingly stringent demands of consumers.

Content delivery networks are a core part of this solution. They ensure not only that scalability requirements are met, but also that customers receive the best possible experience. The success of these platforms is well documented, but this is not to say that developments in this area can cease; in fact constant innovation is required. As demand increases at levels beyond what was envisaged only a few years ago, so does the scale and measure of the delivery networks necessary to support it.

Scaling such a platform naturally introduces complexities concerning the effective management, provision and control of the service. As a result, there is a necessity to simplify this process to ensure that resources are appropriately utilised, to avoid misconfiguration and to negate the associated overhead of deploying additional equipment. This needs to be done in an open and programmable way to not only guarantee interoperability, but also to encourage the reuse of function and behaviour.

Given recent developments concerning the flexible provisioning of resources and networks, services can now take advantage of the ability to dynamically scale according to a number of factors, including anticipated load, associated cost and energy utilisation.

This also provides a number of potential cost benefits for operators, including a reduction in the cost of delivering content; both in terms of network utilisation and infrastructure deployment. Exploiting these should not only result in the extension in the lifespan of equipment, but better utilisation of the resources already in a network.

These resources can also be effectively freed to enable other services to utilise them, furthering the efficiency of existing fixed hardware resources. Content delivery networks rely on functionality offered by dedicated appliances or third-party services; this can now be realised in-network using commodity switching hardware.

Delivery technologies have also moved to become more adaptive in the face of changing network conditions. This introduces complexity, but provides the potential for additional efficiency gains. Content delivery platforms need to consider these developments, and determine the best way to work in harmony with them.

Given the current state of content distribution technologies, and the availability of flexible and configurable infrastructures, the focus for this thesis is to propose a next-generation content delivery platform for the modern Internet.

1.4 Thesis Aims and Contributions

This thesis aims to investigate the future of content delivery platforms. To be achieved through design, implementation and evaluation, the main aims and con-

tributions of this thesis are summarised below:

1. **OpenCache, an infrastructure-assisted content delivery platform design:** In this thesis, we aim to identify a set of key requirements for future content delivery infrastructures. This is achieved through the analysis of existing systems and solutions, but also with consideration for current trends and emerging technologies. In particular, this thesis will consider developments in software defined infrastructures and content delivery technologies. As these will undoubtedly shape the future of platform development, we will provide a comprehensive design. Entitled *OpenCache*, this will aim to contextualise and encompass these elements.

2. **A proof-of-concept implementation of the OpenCache content delivery platform:** To evaluate and examine the effectiveness of this design, a prototype implementation will be built. This follows the specification of the aforementioned design, and forms the basis to evaluate the application of software defined infrastructures in this scenario. It will also be used to consider the implications of using new delivery technologies, and the impact that this may have on both existing and future content delivery platforms.

3. **An evaluation of OpenCache's feasibility, performance and user experience impact in large-scale testbeds:** Through the use of the aforementioned prototype, this thesis will contribute an initial measure of feasibility in using these new technologies. This includes utilising the flexibility in virtualisation platforms to dynamically scale resource allocation dependent on availability, demand, or any other factor, as well as understanding the impact of using software defined networking to forward requests for content. To measure potential detriments to performance, this evaluation must be coupled with deployment into genuine network topologies, using production switching equipment and handling realistic traffic. This evaluation will also consider user experience in these scenarios; an important metric in commercial settings.

4. **The specification, implementation and evaluation of the OpenCache API for programmable cache control:** This thesis will propose an open API for the control of content delivery networks, leveraging newfound flexibility in infrastructure and network provision. Noting a lack of

such a specification, this thesis will provide a preliminary outline for the realisation of a programmable cache infrastructure. This allows logic and applications to be developed regardless of underlying capabilities, including across alternative providers, and should foster innovation and openness in the otherwise closed environment of content delivery networks.

1.5 Thesis Structure

This thesis is structured into six individual chapters. Following this introduction, we describe the importance of the Internet and its modern-day role in delivering content in Chapter 2. The chapter also examines the trend towards the softwarisation of networks and the services within them. This chapter also includes a detailed description of the evolution of video delivery, including the protocols and infrastructures that are employed to enable this. Finally, this chapter examines the fledgling field of infrastructure-assisted applications, highlighting work across a number of domains.

The following chapter, Chapter 3, presents the design of an evolutionary content delivery platform that seeks to utilise emerging technologies to improve the process of delivering video to large volumes of people. This includes a detailed motivation, which considers influences from a multitude of sources, and results in a multi-layered architecture capable of meeting the rigorous demands imposed by modern providers and their customers.

Chapter 4 provides details of a prototype implementation of the aforementioned design. This includes the node used to serve content in response to a user request, as well as the controller used to coordinate and manipulate multiple nodes at once. It also details the realisation of the API, and an exemplary application which utilises such to determine cache operation. Finally, this chapter describes the implementation of a number of complementary tools used to aid both usage and evaluation of the platform.

In Chapter 5, we present a detailed evaluation of the prototype in a number of different scenarios. Each of these evaluations takes place on a different pan-European experimental facility. In the first instance, we demonstrate the feasibility of using software defined networking technology to redirect requests for content towards a local cache. We then evaluate the prototype in respect

to a number of recognised quality-of-experience metrics. Finally, we show the flexibility and power afforded to applications using the API. More specifically, we show how functions typically provided by dedicated hardware appliances can be replicated within the network itself.

Finally, in Chapter 6, we present the contributions and impacts of this work, in addition to outlining future avenues of research created as a result.

Chapter 2

Background and Related Work

In this chapter, we examine the history of the Internet and how it has developed to meet the changing needs of users. This progression has led to a number of significant challenges, both past and present, which are outlined in Section 2.1.

One method of addressing these is through the use of programmable networks; designed to offer increased flexibility to network operators by allowing them to modify and control the network in an agile fashion. In Section 2.2, we examine how previous work has influenced current thinking in programmable network design, and highlight the current trends in technology. This includes the extension of these principles to encapsulate the underlying hardware and software that support the myriad of services that run within modern networks.

Many of the innovations in network design have been driven by a need for greater efficiency in the face of growing demand. A significant portion of the traffic in today's Internet is generated by the consumption of video content. In Section 2.3 we discuss the continued importance of the Internet in delivering video to thousands, if not millions, of users. This includes the parallel evolution of protocol and infrastructure design necessary to ensure that this continuous demand can be satisfied.

There is also a growing body of work concerned with utilising the flexibility in emerging infrastructures to aid application development and deployment. Outlined in Section 2.4, we discuss a number of areas in which this approach has been taken, including content delivery.

2.1 The Growth of the Internet

The Internet originally came into being with the development of personal electronic computers. This occurred in 1950s through to the 1970s. To begin with, computers were connected together using rudimentary packet networks. As the cost of owning and running such devices was high, these deployments were largely limited to various computer science research laboratories, located in different countries throughout the world.

As these facilities developed and grew, focus began to shift towards *internetworking*; connecting together these disparate networks into a network of networks. Many of these standalone networks ran their own proprietary protocols, and were otherwise incompatible with each other. In 1982, the TCP/IP suite was introduced, which standardised the way that one end host could communicate with another. Evidently, the networking equipment in between also needed to support this communication. Soon after, commercial Internet Service Providers (ISPs) began to appear. These offered network connectivity, primarily to commercial entities, which enabled them to communicate with other users using the service.

In the 1980s, Tim Berners-Lee theorised that documents could be interlinked together to form an information system. This consisted of a number of elements: Unique Resource Locators (URLs) used to globally identify a resource, HyperText Markup Language used to publish information and the Hypertext Transfer Protocol (HTTP) used to transfer this information from one client to another. Together, these elements formed the basis for the *World Wide Web*.

As networks developed, and capacity increased, the price of connectivity dropped. Coupled with the increased affordability of personal computers, this permitted many ordinary households access to what had become the *Internet*. Users now had unparalleled access to information, typically in the form of web pages. At this time however, there was not sufficient bandwidth to transfer media at any usable rate, particularly on a low-capacity residential connection.

Over time, new technologies provided households with even greater connectivity. During the 2000s, many households moved from a Dial-up (or ISDN) based service to higher-capacity ADSL connection. With this, home users now had the capability to access a far richer set of services, including multi-player gaming, video calling, and both live and on-demand video [126].

In more recent years, these connections have continued to increase in capacity.

A selection of technologies, focusing on pushing the high-capacity optical fibre part of the connection closer to the user, have seen widespread deployment in a number of countries. These range from FTTN (Fibre To The Neighbourhood) to FTTD (Fibre To The Desktop), varying in the distance from the fibre termination to the user. As well as enabling novel services, these connections also provide the ability for a household to consume a multitude of services in parallel.

Yet despite the development of these technologies, network provision continues to be challenged. Increasing last-mile capabilities simply transfers the bottleneck to other parts of the network, or even to the infrastructure that underpins the services in the first instance. Subsequently, this too must be upgraded to match the new level of demand. This perpetuates a continuous cycle of development and expansion, which can be a costly and time-consuming approach to all parties involved.

Given this, there is a clear case for developing more efficient uses for existing infrastructure, without the need to continue capital expenditure. These approaches offer an opportunity to break from the cycle of provisioning by allowing equipment and services to remain in place for longer.

In recent years, these efforts have focused on improving the behaviour and functionality offered by the underlying networks; responsible for delivering the services received by consumers. By innovating here, operators are able to benefit from increased flexibility and programmability, which allows them to better utilise the resources already at their disposal. Many of the innovations discussed in Section 2.2 move towards *softwarisation*; that is, enabling a transition from closed, hardware-based appliances, towards open, software-based implementations that achieve the same purpose. This trend extends from the switching fabric itself, right through to the services that support the continued operation of a network.

To compliment this drive for efficiency, there has also been significant innovation in the technologies and methods used to deliver the content. In this thesis, we focus on video as the primary example of media distribution. This is due to its significance in the overall traffic profile observed on the current Internet [46]. In Section 2.3, we examine the history behind this important medium, and examine how protocols have evolved over time to take advantage of reduced latency and increased throughput in residential Internet connections.

Moreover, we examine the state-of-the-art technologies used to deliver video in different ways, both by evolving existing well-understood techniques, or through

the development of radically different approaches that depart from traditional protocols and mechanisms. We also explore the infrastructures that are deployed in today's Internet to enable content distribution on a massive scale. This includes highlighting the various design considerations that must be taken into account when deploying a system in production.

2.2 Network Softwarisation

In the previous section, we highlighted the need to increase efficiency in existing networks so that costly upgrades can be avoided. One method of doing this is to empower network operators with the ability to dynamically change and adapt their network dependent on their needs, thus escaping the need to change the current hardware deployment.

In the case of *network softwarisation*, select components in a network are taken from their hardware-based counterparts and moved into software. This process varies amongst the constituent elements of a network, and ranges from the decoupling of the control plane in network switches and routers, to the virtualisation of hardware appliances. In most cases, software solutions have the distinct advantage that they can be deployed as required and thus scaled appropriately. This scaling, especially with the correct supporting platforms, can take a matter of minutes rather than days. This, coupled with the ability to modify functionality of software in-place, has led to significant interest from both industry and academia.

Yet, at least in the case of programmable networks, the concept is not a new one. In Section 2.2.1, we discuss previous work on the subject. Much of this work has provided influence and insight to current work in the area, which is described in Section 2.2.2. Included as part of this section is a focus on a technology which has seen widespread usage in many fields, including deployment in production networks.

With the rise of commodity servers offering cheap, affordable, storage and compute resources to network providers, operators and academics alike sought to exploit these infrastructures to afford the same programmability to the functions that are relied upon within a network. Network Function Virtualisation, as described in Section 2.2.3, is an attempt to do just that. By understanding

the potentially periodic nature of resource utilisation, it is possible to avoid over-provision by directly matching the allocation of resources to meet the current demand. This not only allows resources to be increased in the face of unexpected amounts of load, but also grants the ability to relinquish those resources during periods of low utilisation (at which point other functions can use the resources instead).

2.2.1 Programmable Networks

The desire to rapidly develop, deploy and subsequently manage networks is not a new one. Many of the issues surrounding the difficulty of deploying new protocols and standards were as important then as they are now. Early work included the *SOFTNET* project [173], which proposed that each network element acted as an interpreter on receipt of a packet. If this packet contained a pre-defined command string, it would be immediately executed by the device, allowing code to be run remotely.

This visionary work was conducted in the early 1980s, yet the concepts would not be revisited until the mid-1990s: *Active Networking* [164] introduced the concept of user-programmable switches and *capsules*; snippets of code carried in packets that could be run directly on the switch. However, there were some concerns around the safety and security of this approach, especially when it came to managing and enforcing resource utilisation [134].

Later, the *4D Project* [66] advocated a multi-plane approach to networking, with a clear separation between *decisions*, *data*, *dissemination* and *discovery* functionality. The *NETCONF* protocol [89] can also be viewed as a management protocol for modifying the behaviour of networking devices, particularly when combined with *SNMP* [72]. However, there were some shortcomings evident with this pairing, including a lack of data/control plane separation and vendor neutrality. It is also unsuitable for reactive control, as it did not provide any real-time functionality.

2.2.2 Software Defined Networking

Despite the previous work in programmable networks, current Internet infrastructures still lack the flexibility pioneered by these early projects; it is typical

to find the control logic of a network appliance co-located with the forwarding plane. More specifically, decisions on *where* are packet should be forwarded are made on the same devices that actually *forwards* the packet. As much of this infrastructure, at least in access and carrier networks, is physical hardware, these two processes are tightly coupled. If changes to the operation of a device are required, manual configuration is often needed. This time-consuming and intricate process is confounded when large networks, containing hundreds of forwarding devices, are considered.

This situation is made worse as there is no consistent method of interacting with these devices; manufacturers have their own configuration formats and syntax. In recent years, efforts have been made to standardise at least elements of this configuration [89], and facilitate a level of interoperability between vendors. However, even with a common configuration platform, it was previously impossible to reactively manipulate the control logic. Once defined, the logic would remain in place until modified by configuration and cannot dynamically react to individual packets or flows, adapt to changes in the wider network or facilitate the layering of network-aware applications on top of the infrastructure itself.

Software Defined Networking is a continuation of the programmable networks paradigm, designed to change the status-quo by decoupling these two layers. The forwarding plane remains on the switch, to enable the hardware accelerated forwarding necessary to satiate modern networking demands. However, the control plane is detached from the hardware and placed entirely into software. In the case of Software Defined Networking (SDN), the control plane is migrated to a software *controller*. This controller then takes on responsibility for the behaviour of all of the connected devices.

This behaviour is defined using two distinct methods. The controller can define the behaviour of a device by waiting until an alert is received on the initial establishment of a flow. Once this occurs, the controller receives a message, containing either the content, or part thereof, of the first packet. The controller can then parse this packet, and processes it as it wishes. In most cases, this will result in the packet being forwarded on an appropriate port of the device that produced the alert. The controller may also install a set of rules on the device (as a result of the initiation) to handle subsequent packets belonging to the same flow. By doing so, these packets will be processed in the fast-path of the switch, increasing performance and avoiding the associated latency with passing a packet

from the device to the controller, parsing and processing it.

Alternatively, the controller may install rules proactively, to ensure that alerts received by the controller are minimised. Evidently, this requires knowledge of the type of packets that are likely to traverse the switch, and also the destination that should be forwarded on. In reality, both techniques are used together to ensure performance close to that of existing infrastructure is achieved.

Given the nature of the control plane as a pure software implementation, it is significantly easier for operators and developers to add new functionality to the network. It no longer requires the costly refresh of hardware devices; by building new functionality in software, they can test and deploy the software into production networks much quicker than before. It also allows novel functionality to be built that otherwise would require in-depth collaboration in hardware and software device design.

Despite this new found flexibility, functionality can only be built within the constraints of the underlying protocol mechanisms used to control the hardware switches. In the remainder of this section, we discuss two such protocols, and examine in detail how one of these has changed over time to enable significantly more functionality to be programmed in the network. In the scope of this work, we discuss in later chapters how this technology can be used as a novel redirection technique for content delivery networks.

2.2.2.1 OpenFlow

OpenFlow is a realisation of Software Defined Networking concepts. It is a protocol specification designed to enable the control of network devices. Since its inception [71], OpenFlow has become the most widely recognised and deployed technologies related to the SDN paradigm. OpenFlow itself was initially envisioned as a tool to enable research and experimentation to take place on campus LAN [130].

Prior to OpenFlow, there had been a lack of work that had made its way from the research world to production. One of the fundamental issues identified with this process was a lack of scalability and realism in the development and evaluation of this work. OpenFlow aimed to combat this by allowing both research and production traffic to coexist on the same network. By doing so, network operators could continue to provide the connectivity and service to their users,

whilst the researcher would be afforded access to a specific portion or *slice* of this traffic, on which they could do experimentation.

This compromise between production and research requirements was typically realised through the use of tools such as Flowvisor [154], which enabled the separation of traffic through establishment of pre-defined network slices. Each slice represented a subset of traffic, defined using a number of tuples, such as source and destination IP addresses, port numbers or VLAN tags. Using this tool, an experimenter could precisely define exactly the traffic they were interested in, without disrupting the other traffic flowing through the network. Importantly, this could also be done for multiple experimenters simultaneously, affording each a separate slice that offered isolation between experiments.

Key to OpenFlow, and arguably a core reason for its success, is that it was vendor agnostic; that is, it was not developed with a particular vendor, and as a result was released openly for anybody to use. This created a situation whereby a vendor could support the OpenFlow standard by implementing the minimum necessary elements of the OpenFlow specification. This openness had a significant advantage, in that a software controller supporting OpenFlow could communicate and control a variety of devices, regardless of their vendor. The initial industry support for OpenFlow came with version 1.0, which is discussed in the following section.

Version 1.0

Version 1.0 [25] of the OpenFlow specification was released in 2009 and adopted by network vendors soon after. This version describes a single OpenFlow *flow table*, consisting of a number of *flow entries*. When a packet is received on the device, the packet header is compared against these entries. Each entry consists of a number of fields with specific values. If a field is omitted, it is assumed that any value can be matched (wildcarded). Once a packet has matched on an entry in this table, a set of actions is applied to the packet. These actions include forwarding the packet on a specific physical port, forwarding to a virtual port (such as a port aggregation or VLAN), or even flooding the packet to all ports.

Importantly, these matches also enable the packet headers to be modified, such as changing the source or destination address, or incrementing a TTL value. If the packet does not match on any of the rules found in the flow table, then

the device passes the packet to the control so that it may make a decision on its destination. Each of these flow entries also includes a set of counters, which are updated when a packet matches. These include both packet and byte counts, and can be retrieved by the controller to use in defining the behaviour of the network.

The set of actions included in the OpenFlow specification are defined in two sets; *Required Actions* must be implemented by the switch in order for it to be OpenFlow compatible, and include the functionalities mentioned previously (forwarding to all ports, forwarding to the controller, etc.). A number of *Optional Actions* are also defined. These can be omitted if necessary, particularly in cases where the underlying device does not support the action, but may assist the developer in implementing additional functionality.

One such action is *NORMAL*, which passes the packet to the traditional forwarding path supported by the switch. By chaining a number of these actions together, it is possible to manipulate the packet, or duplicate the packet, and still pass it through the regular forwarding stack of the switch. Inevitably, these optional features are not supported on every switch: typically they are found on devices where this OpenFlow has been developed on top of existing functionality, as is the case with many hardware switches. However, in the case of software switches, such as Open vSwitch [22], a traditional forwarding path is somewhat of a misnomer; they do not process packets without explicit instruction or configuration.

In order to support OpenFlow-capable switches, a number of compatible software controllers became available. Engineered in a variety of programming languages, they provided a framework in which basic network functionality could modified and new functionality could be built. Controllers such as Beacon [90] also have external APIs, which enable third-party applications to interact with the controller and modify the forwarding plane by installing matches and actions manually, as well as retrieving counter values.

Later, version 1.1 [26] of OpenFlow was released. This version provided support for multiple flow tables (rather than the single table enabled in version 1.0). It also introduced support for more field matches, and introduced multipath functions, where a flow can be sent over one of several paths. Further support was also added for MPLS and VLAN Q-in-Q encapsulation.

Version 1.3

In 2011, the Open Networking Foundation [21] took over the responsibility for developing the OpenFlow specification, with the aim to eventually standardise it. The first standard to be published under their custodianship was version 1.2 [27]. This version included features such as an extensible match support, allowing experimenters to define their own match fields. This was accompanied by an extensible header rewriting functionality, as well as support for IPv6. It also clarified the controller role mechanism, simplifying the process for which controllers were migrated between.

However, this saw little adoption from vendors, and in 2013, the 1.3 [28] specification was released. This incorporated many, if not all, of the changes made in previous versions. It also included new functionality in the form of capability negotiation, which allowed controllers to better discover the capabilities of a connected device. Further to this, improved support for common IPv6 extensions was added, as well as finer-grained reporting in the form of per flow meters. This improved reporting was supplemented with the addition of a duration field for statistics, allowing packet and byte rate to be determined by a controller.

With the development and release of switches supporting OpenFlow 1.3, new controllers began to appear. Examples include the Ryu [38] controller, which offers full OpenFlow version 1.3 support, and is designed to allow researchers and developers to create functionality easily. Ryu is also used as a tool to ensure compatibility and compliance with the 1.3 specification, which enables testing across a variety of vendor hardware.

Other controllers have also been developed to meet some of the scalability concerns evident in the SDN paradigm. ONOS [61] aims to do this by operating in a distributed fashion, whilst ensuring state is synchronized between controller nodes. In order to address concerns over commercial applicability, the OpenDaylight consortium [23] is also attempting to build a controller capable of running in a production environment, and support the vast array of protocols and functionality necessary to do so.

Version 1.4 and Beyond

OpenFlow continues to be developed, with subsequent versions (1.4 [29] and 1.5 [30]) being released recently. In version 1.4, new features include more exten-

sible wire protocols for defining custom match fields, and extensions encompassing optical port properties. This also includes bundles, which allows a group of OpenFlow messages to be applied as a single operation, and synchronised tables, allowing lookups to occur on multiple tables simultaneously.

In version 1.5, egress tables were introduced, enabling processing to be done in the context of an output port (rather than input port, as previously). This version also introduces packet aware pipelines, allowing packets other than Ethernet packets to be processed. There is also the ability to trigger an alert when a statistics exceeds a given threshold (in opposition to regularly polling for a metric, which has a processing overhead). However, despite the extra functionality provided in these versions, as of yet, no organisation has released a version capable of supporting either of these versions.

However, continuously iterating the OpenFlow specification to include new header fields is not necessarily a scalable nor sustainable solution; every version has included new fields, with the trend continuing in the latest releases. The expansion of fields is in part due to a growing range of mediums that OpenFlow is targeting, including optical and wireless technologies. In order to address this, P4 [64] proposed as a way to describe the data-plane connectivity of a network using a domain-specific language. Programs written in P4 specify how a switch processes packets, and works in harmony with OpenFlow to provide even greater flexibility. Importantly, P4 is both protocol and target independent, allowing new protocols to be developed without concern for the underlying hardware. It is also field reconfigurable, allowing hardware behaviour to be changed even after it is deployed.

2.2.2.2 ForCES

Forwarding and Control Element Separation (ForCES) [86] is similar to OpenFlow in that it seeks to detach the forwarding and control planes within a devices. However, it differs in that the control plane is still co-located with the devices rather than being located remotely. This difference means that a ForCES switch is still viewed as a single entity, which is intended to increase performance. Rather than the flow table approach used in OpenFlow, ForCES utilises well-defined Logical Function Blocks (LFBs), although they can ultimately be used to achieve the same effect [170]. Despite the earlier standardisation of ForCES through the

IETF, OpenFlow has seen far greater adoption, driven by a combined effort from industry and academia.

2.2.3 Network Functions Virtualisation

Building upon the enablers of Software Defined Networking, and in some cases, by exploiting related technologies, Network Functions Virtualisation (NFV) aims to extend the same transparency and flexibility to the many appliances and services that support the operation of large-scale networks. As these systems continue to grow in both scale and complexity, managing and operating them has become a challenge. Examples include firewalls, intrusion detection systems and broadband access routers. Functions provided by third-parties, such as content delivery networks, are also viewed as vital to the smooth operation of many of these networks, particularly when they provide access to consumers.

As with the underlying network hardware, network functions often perceived as *black boxes*; operators have little or no knowledge of their internal workings. This can create issues, particularly when engineering the network and planning for expected load. Furthermore, the appliance may not necessarily operate in the most efficient way, with no consideration for the operation of other functions or the availability of resources. In most cases, the operator has minimal control of their behaviour, with some appliances offering limited fixed configuration and others being completely managed by external entities.

To break away from this paradigm, there has been a substantial effort to create a new generation of network functions based at least partially in software. The primary reason for bringing these functions into software is so that they can be virtualised. This process brings about a number of unique challenges, which once addressed, present the potential for additional benefits. These requirements are discussed in the following section, and based upon documents published by the ETSI Network Functions Virtualisation Industry Specification Group [48], as well other sources [100].

Chief amongst the requirements for a virtualised function is portability; that is, the ability to create, migrate and destroy functions regardless of the underlying technology platform. This allows the functions (or elements thereof) to be optimised based upon location and availability of resources. As the optimal arrangement may change over time and as the availability of resources changes, this

portability allows flexibility not afforded to current fixed appliances. In this case software has to be decoupled from the underlying hardware. This gives operators the ability to use cheap, commodity servers without vendor lock-in, reducing their capital expenditure.

Virtualisation also presents new requirements for scaling. Presuming suitably implemented and managed software functions, scaling can be achieved by allowing a function to be matched with resources consummate to the task it needs to complete. Rather than a static one-time provision (where provisioning new hardware may take days or even weeks), this allows resources to be consumed more efficiently and without significant wastage. With a hardware-based solution, resources are allocated based upon a worse-case scenario. However, a virtualised function can dynamically adapt to load to both release and reserve resources as necessary. This also has consequences for energy efficiency as resource allocation can be tuned dependent on the availability and cost of electric tariffs. This flexibility also has the potential to power down servers or even entire racks.

The ability to adapt appropriately and in a timely manner also introduces a new requirement: these functions must report performance and usage. This information can then be used to adapt appropriately by either increasing allocation and creating new instances, or inversely, by reducing allocation and removing instances. This instance management is only possible if at least part of the function can be parallelised in some way. This allows instances to be located over multiple, potentially co-located, physical servers. This offers an alternative approach over increasing the allocation of a single, monolithic entity (which may have upwards bounds imposed by the underlying hardware).

This same flexibility can also be applied to scenarios that require increased resiliency, as functions can be recreated in the case of failure. This includes problems with the virtualised instance, as well as downtime in the hardware itself. Through the use of the above mentioned metrics, each virtualised function can be monitored for health and uptime. Once issues are observed, remedies can be sought quickly and automatically, even without user intervention. This reduces reliance on sometimes slow and manual processes, and helps to maintain availability guarantees in Service Level Agreements (SLAs).

A software based solution should also be significantly easier to update and modify in-place when compared to a hardware based alternative. This allows faster development cycles without the need to replace costly hardware appli-

ances. It also affords the ability to have a smooth transition path from hardware to software; devices can be trialled and replaced gradually without the need to completely remove existing provision. This is particularly important in cases where performance is a concern and software must be developed to strict performance and latency specifications, which may be a time consuming process.

Clearly there are many advantages to be gained by moving to a virtualised solution. However, getting to a stage where these functions can be used in production deployments that support millions of users is far from reality currently. Software implementations required extensive testing and need to prove that they can perform equivalent to their hardware counterparts. There has already been issues raised over current limitations in the off-the-shelf hardware that would ideally support virtualised functions [108]; commodity hardware does not necessarily support processing packets at line-rate, a requirement for high bandwidth network functions. To overcome this, the authors have proposed using a combination of technologies and techniques to bypass inefficiencies in the underlying operating systems and program design.

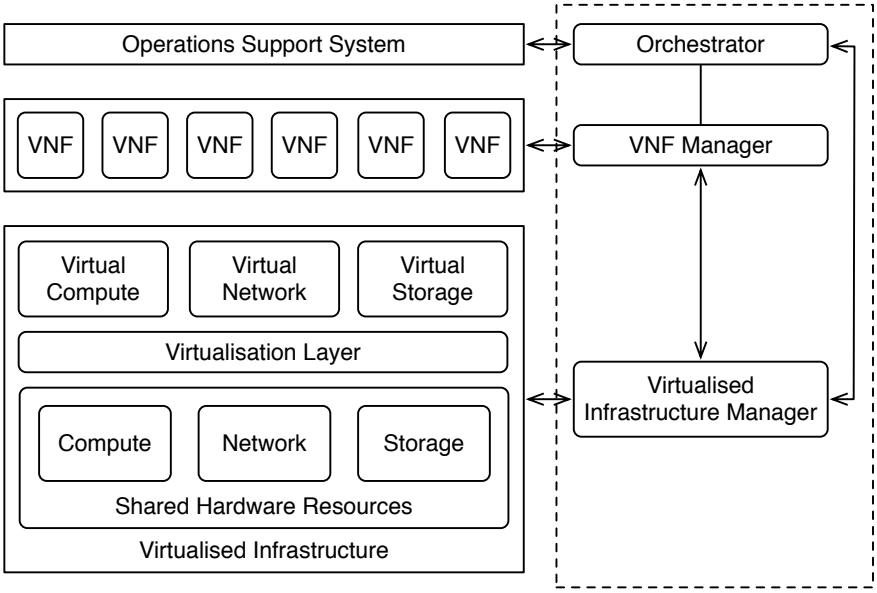

Figure 2.1: NFV Architectural Framework

It has become clear that coordination and cooperation is necessary in order to address the monumental challenge of architecting and building a platform capable of realising truly virtual functions. Recent efforts have focused on specifying a platform design [9], illustrated in Figure 2.1. This describes the various layers on

which the Virtual Network Functions (VNFs) sit. A VNF provides a specific network function, with the possibility for multiple instances to be chained together to provide a broader range of services.

This design also outlines various managers and an overall orchestrator needed to ensure the smooth operation of the platform. However, there are significant steps that need to be taken in order to realise this platform today. As such, current efforts [32] have focused on a subset of this architecture, namely the lower *Virtualised Infrastructure* layer, and the elements contained within it. These are the initial steps necessary to bring NFV into production environments, and utilise existing production-ready cloud platforms, such as OpenStack [31], to provide the necessary functionality.

Alternatives to this approach include the usage of tools designed specifically to meet the stringent latency and throughput requirements required in modern network appliances [128]. This work includes extensive evaluation which demonstrates the diminutive size of the memory footprint, as well as the rapid startup times possible. A number of proof-of-concept implementations are also included, such as a firewall, carrier-grade network address translation and a broadband remote access server.

Other notable areas requiring work include the placement and scheduling of virtual appliances. As this process can both introduce deficiencies, and prove valuable in addressing latency requirements, it is a key area of study. Existing early work includes an evaluation of numerous algorithms that can be used to map and schedule functions [132], as well exploring the efficient placement of such given a set of constraints [133]. Further work has examined the placement problem in the context of a mobile core, and combines function placement with topology optimisation [132].

In carrier networks, it is common to find multiple functions linked together in a service chain. The order through which traffic flows from one function to the next is important in these cases. For example, if traffic flows through a Intrusion Detection System (IDS) and then on to a Wide Area Network (WAN) optimiser, it is critical that the packet is inspected at the IDS before it is encrypted by the WAN optimiser. This chaining can impose additional requirements on the location and composition of functions for latency or security reasons. Specifying and placing these chains of network functions requires the ability to not only formally describe the composition [131], but also ensure that they are arranged

optimally given the requirements of each function [122]. Realising this chaining with existing software platforms has again shown the deficiencies of current implementations [67], with a clear need to develop further existing platforms.

2.3 Video as an Emerging Application

It is not just networks that have seen rapid development in recent years; the services and technologies that utilise these resources have also seen similar advancements. The Internet is used increasingly to deliver media and content to thousands of simultaneous users. In fact, this is now a core part of the Internet's current function: the delivery of video now constitutes a large percentage of overall Internet traffic with recent estimations predicting a 80% share of all traffic on the Internet by 2019 [46]. Clearly the techniques used to distribute this video are as critical as the infrastructure necessary to support it. In the following section, we examine the evolution of these techniques into the technologies we use today.

As bandwidth increased in the Internet, particularly in the access networks used by consumers, new services became viable. Over time, and as the throughput continued to increase, the services began to not only offer an increased amount of content, but also content available in higher resolutions and qualities. Evidently, this requires a higher sustained throughput to deliver in a satisfactory manner. Coupled with an increase in production quality and user expectations, this trend continues even now, with consumers now demanding high-definition content by default, and ultra high-definition on the near horizon. By 2019, it is predicted that 70% of content will be high definition [46]. These requirements place huge demand not only on the networking infrastructure, but also on the technologies and techniques used to deliver the video from the source to many users.

The importance of ensuring that video is delivered in a satisfactory manner has now become a commercial matter, especially in the case of paid or ad-supported services. In these cases, it is critical that the user receives the highest quality of experience, without any degradation. A suboptimal service can lead to user disengagement, with even with a small delay in start-up time causing users to switch video [88]. As traditional disc-based technologies which have increased vastly in capacity in recent years, the Internet is increasingly becoming a viable alternative from which to distribute content (in all forms) to the masses.

2.3.1 Protocols for Video Delivery

Without an adequate protocol for delivering video to end clients, advances in infrastructure design cannot otherwise be fully realised. Since the inception of the Internet, the protocols used to deliver video have changed. Different standards and techniques are also appropriate dependent on the circumstances, and have been deployed to match changing user behaviours and habits. In the following section, we explore a number of these, explaining their significance in the evolution of video delivery over the Internet.

2.3.1.1 Real Time Transfer Protocol

The Real Time Transfer Protocol (RTP) is a technique long used to deliver video. Defined by the IETF in 2003 [150], it defines a transport protocol which can be used for real-time transfers. The underlying technology used to achieve this is typically[1] the User Datagram Protocol. UDP is a stateless mechanism through which information can be sent from a server to a client without the client having to specifically acknowledge the receipt of each packet. Although it is considered a lightweight mechanism, it provides an unreliable service where packets can arrive out of order, or even be lost altogether.

The usage of UDP is particularly appropriate in circumstances where live real-time video needs to be delivered as soon as possible, regardless of impairments in the network. As a result, UDP is used for delivering video during conference calls and during live broadcasts, where timeliness of delivery is of paramount importance. However, the best-effort service offered by UDP transport streams is also a significant downside: the unpredictable nature of the Internet, where congestion and queuing can prevent reliable transmission, means that the quality of experience provided to the user can be variable at best, and suboptimal at worst.

Semantically above RTP is the Real Time Control Protocol (RTCP), which operates at the session layer. Its primary function is to provide feedback to the server in regards to the quality of the data layer, allowing the server to adjust the rate at which it sends to the client. To compliment the capabilities of RTP, the Real Time Streaming Protocol (RTSP) [151] is used to control streaming at a presentation-level. Connected clients can send commands to the server streaming

[1] Reliable TCP can theoretically be used, although this is rarely the case.

the video to them. These included *play*, *pause* and *record*, and emulated the interaction with physical media players of the day, providing a level of interactivity that users were familiar with [78].

This technique carries a large management overhead, as each session needs to be maintained on a per-client basis. This is a complex and resource intensive process. As the popularity of Internet video services increased, scaling became a challenge as infrastructure requirements reached an infeasible level.

2.3.1.2 Real Time Messaging Protocol

The Real Time Messaging Protocol (RTMP) is an important protocol in the history of Internet video. Proprietary by design, it became the de-facto standard for delivering video in the 2000s. This was brought about largely by the ubiquity of Flash Player and its associated plug-in. It usually used TCP as the underlying transport mechanism, although it too was flexible enough to also be compatible with UDP if needed.

Importantly, RTMP is actually format agnostic; any video can be delivered using the technology. Extensions to RTMP include support for tunnelling through HTTP (RTMPT) to avoid issues with firewalls blocking the port normally used. Various versions also offer differing levels of encryption on the stream. Recent years have seen a drastic demise in the usage of RTMP, however. This can likely be attributed to the growing usage of handheld devices, which lack support for the protocol.

2.3.1.3 Multicast

Unlike RTMP or the RTP family of protocols, multicast is one-to-many distribution method, rather than one-to-one. This means that information can be disseminated from a single sender to many receivers. The advantage of this in an IP context is that the sender does not require knowledge of the receivers before transmitting the information: the network maintains the list of receivers. These clients then selectively join the multicast *group*, leaving and joining at any point. By doing so, they subscribe to receive messages from the sender. As the network handles the replication on a packet level, the sender need only send a single message towards the multicast group, greatly increasing efficiency.

Multicast is well suited to distributing live content where many clients require

the same content delivered in parallel and at scale. However, this method of dissemination is not suited to content delivered on-demand: requests for content are instead distributed over time. The on-demand model is also a pull-based principle; the client requests the content when it requires it, whereas multicast is a push-based technique; content is sent from the server, regardless of whether or not a client is subscribed to receive the content. As a result of these limitations, multicast is often deployed using UDP as a transport protocol. This avoids having to maintain state, but does mean that transmission is unreliable.

In order to apply a multicast strategy for on-demand content delivery, a number of alternative multicast video distribution techniques have been categorised [76]. *Broadcasting* [104] dictates that the sender periodically broadcasts the video, irrespective of demand. A user can then receive the video by waiting for the next broadcast cycle. Evidently, although this increases efficiency by ensuring that the content only has to traverse the network once in a given time frame, it is however not a real-time on-demand solution, as the user will likely be delayed until they can consume the content.

An alternative approach is *batching* [81], in which the server waits for a number of requests for the same content, and servers it using multicast once a threshold has been met. As with broadcasting, early users will have to wait until this threshold has been reached. This variable must be flexible, otherwise the early users will simply cancel the video playback before it has begun. Similarly, batching does not provide a true on-demand service.

Patching [105] on the other hand aims to achieve real-time playback by offering the initial requester a multicast stream, whilst subsequent requesters are served using unicast flows. These late arrivals continue to be served in this way until they have buffered sufficient content to join the multicast stream. Finally, the *stream merging* [87] technique is similar to patching with the modification that later streams are delivered over multicast too, with users batched appropriately.

Later in this chapter, we examine the move towards adaptive streaming, where video is offered at multiple quality levels and resolutions. A client can use this selection of content to adapt its own playback to best match its capabilities and present network conditions. Multicast does not have the ability to manipulate the quality of video without effecting the entire multicast group, which would evidently impact all the connected clients. An alternative approach would be to offer many multicast groups, one for each version of the content. However, this

would likely carry a significant management and provisioning overhead, and in a worst-case scenario, simply resemble a unicast distribution infrastructure.

2.3.1.4 Peer-to-Peer

Peer-to-Peer (P2P) networking is a paradigm where end-hosts communicate directly with each other, in opposition to the client-server model. In the latter case, multiple clients connect to the same server, whereas in a P2P network each client can connect to any another participating client in order to share a resource. As such, a P2P network is self-organising; clients can leave and join as they wish, and the topology will change to match. This behaviour is ideal for the delivery of content as distribution scales naturally (as more participating nodes join) without the need to also increase server provision.

In a content delivery scenario, a client requests content from other participating nodes, assuming that they hold a copy of this content. In order to increase the fidelity of such a request, content is often separated into smaller chunks; a client may not necessarily hold all of the chunks that represent a file, but they can still participate in the sharing of the parts of the content that they do hold.

There are a number of variations found in P2P networks [57], including the level of centralisation and the structure of the network. For example, a P2P network can be fully decentralised, which is where each node has an equal role in the network. Another approach is to partially centralise the P2P network, and have a number of nodes act as *supernodes*. These supernodes may preform additional functions, such as storing a hash-table containing the nearby location of various chunks of content.

A hybrid approach can also be taken, whereby a centralised server is responsible for facilitating the connections between clients, storing meta-data collected from each so that it can correctly direct a request. The formation of the network can also vary: unstructured networks are formed without consideration for the location of content. In order to find the location of a required item of content, a node must search amongst all the nodes until their request is satisfied. As this is somewhat of a random search, the time taken to do so is indeterminate. Clearly, in cases where many hundreds or thousands of clients are searching for content simultaneously, this can produce a significant overhead, especially in cases where the content is scarce and the chances of finding it narrow [123].

A solution to this is to use a structured network, in which an overlay topology is used to direct requests to a specific node which either holds the content, or knows where a copy is located. Technologies such as Chord [161], a distributed lookup protocol, enable a relationship to be made between a key and a node. If a key is associated with an item of content, this can be used to locate the item in the network. This addresses the responsiveness and efficiency of searching, although maintaining the required information can be difficult in networks with a high turnover of clients. Other previous work [84] has highlighted P2P networks as a strong candidate for fault-tolerant delivery platforms, allowing clients to recover from failures quickly.

P2P technology can also be used in a live streaming context, and has been deployed as viable alternative to unicast IPTV services [102]. However, in the case of video playback, there is an additional constraint in that the content needs to arrive in a particular order, in opposition to a bulk file transfer (such as in a file sharing network) where the order is irrelevant. This stands true for both live and on-demand video; live because the user will not receive a true real-time experience if the sequencing does not occur, and on-demand because the user will typically watch the video from the beginning (and thus those initial segments should be retrieved first).

The necessity for ordered chunk delivery in an on-demand context is highlighted in [58], to which they propose a system of network coding, segment scheduling, and topology management in order to address some of these challenges; failure to correctly determine the connectivity and content availability in a P2P network can lead to issues during playback. In [124], these challenges are overcome by accurately deriving peer connectivity and devising a random mesh to ensure content is delivered efficiently.

For a device to also serve content to other participants, it must also be stored for a period of time. This is the case regardless of whether the content is to be consumed live or on-demand. A P2P distribution network relies on this storage to operate, which offloads some of the cost in infrastructure necessary to operate in a client/server model [75, 106]. The storage requirements, and indirectly the length of time necessary to store content, differ depending on the context: on-demand usually requires a larger capacity coupled with greater persistency, whereas live P2P streaming can be achieved using a much smaller storage allocation, given that content needs be retained for a much shorter period of time.

There are a number of approaches to P2P streaming, which are broadly divided into two main categories [125]. The first approach is to use a *mesh-based* approach, which is similar to that used for conventional file sharing (such as BitTorrent) and video-on-demand content. There is an alternative approach in *tree-based* streaming: this organises peers into multiple trees using an overlay mechanism. Each of the trees is designed to be balanced amongst peers, stable (as to avoid peer churn) and short. A peer may belong to multiple trees, with content disseminated from the top of the tree structure. If a peer does not have the required bandwidth to forward the content to the underlying peers, then they do not receive the content.

Regardless of the playback method, the inherent resource requirements can be problematic on a resource-constrained device such as a set-top box or mobile phone, where storage is always at a premium. A P2P network also requires the user to also share their bandwidth capacity so that others may retrieve the content from them. It can be difficult to incentivise users to do this, particularly when it is a relatively scarce resource (as in the case of ADSL networks) [120]. P2P traffic can also be difficult to manage and predict due to its opportunistic behaviour and is often seen as unfriendly to ISPs [113]. It has also proven to be incompatible with many firewalls and network address translators [80], further lessening its potential usefulness.

2.3.1.5 HTTP Progressive Downloads

The Hypertext Transfer Protocol (HTTP) can also be used to transfer video. As the current de-facto standard for delivering files of any size over the Internet, HTTP is a reliable protocol used every day to transfer millions of web pages and images. Although alternative technologies exist, HTTP is easy to use in comparison and well understood amongst administrators and developers. HTTP servers have become somewhat of a commodity, and building them at scale is a well understood process. Most, if not all, networks are configured and engineered to handle HTTP traffic, including compatibility with the supporting services that often run aside modern networks. It is a ubiquitous part of the modern Internet.

It is natural then that HTTP came to be used to deliver on-demand video content. An important part of enabling this is the use of progressive downloads. This process allows a browser or media player to download parts of a large file

in smaller ranges. This enables users to watch the beginning of the video whilst the remainder of the video continues to be downloaded, significantly reducing the start up delay compared to required the entire video to be downloaded first. This process also forms the basis for the evolution in video delivered over HTTP, described in the following section.

2.3.1.6 HTTP Adaptive Streaming

Despite the relative success of HTTP as a technology, there were other challenges that simply could not be overcome by HTTP alone. As networks expanded, and bandwidth become more available, a disparity appeared between the connections of different users. Many Internet Service Providers offer tiered services, with different levels of throughput guaranteed at different price points. Similarly, some of these tariffs enforce a limit or cap on the amount of traffic a household can consume over a billing period. Geographically, not all countries moved at the same technological pace either, making it difficult for services to cater for a wide range of connection types and speeds. The explosion of data-based cellular networks only exacerbated the situation, leading to an even greater spectrum of network capabilities.

Despite the ubiquity of the Internet, the capacity of the networks which underpin it are not an infinite resource: with an increase in popularity came a need to further increase the provision of resources. Unfortunately, this is a relatively time and capital intensive process, especially when considered in the scope of a nationwide network. These national networks are often-times connected together, with connections sometimes spanning continents. Clearly this requires vast amounts of coordination, planning and foresight to expand and upgrade. Yet despite this ever increasing technology provision, it is inevitable that during busy periods, congestion can occur in the path between a client and the service they are requesting. This congestion often results in a loss of observed throughput, and in the context of video, a potential reduction in the quality of video a stream can carry.

As alluded to previously, mobile networks have also seen a huge explosion in popularity. In particular, data services capable of carrying video have become an affordable reality for many. A mobile context carries its own challenges though, as the physical movement of a client can lead to a fluctuation in service strength,

which consequently impacts usable bandwidth. Despite this, with the evolution of the transmission technologies, there is a notable trend in the increased capacity that these cellular networks nonetheless afford to clients. This trend is not set to change with the development of future networks [167], which should enable more users to access even higher quality videos whilst on the move.

The suitability of mobile networks has also led to an increase in the specification of the mobile devices themselves. As these handsets advanced to a level capable of video playback, new video codecs were developed in order to allow a relatively resource constrained device to play back video. Content providers now needed to not only encode a video in multiple quality levels, but also with multiple codecs. However, the capability of at least some of these devices has now progressed so far that they can decode videos that were previously limited to personal computers. Regardless of this innovation, these devices are often still resolution constrained due to their very nature as hand-held devices.

It is clear that content creators need to offer various levels of quality, resolutions and encodes in order to match the diversity in both networks and devices. This situation is not fixed either; despite the increase in available bandwidth, the impairments described previously result in fluctuations in bandwidth for an end-host. This can vary from day-to-day, and even from minute-to-minute in the case of a moving client in a mobile network. Clearly a blanket approach to delivering a single quality level is no longer appropriate. Adaptive video streaming aims to combat the unpredictable nature of modern networks by enabling clients to dynamically adjust the quality of video by requesting a representation that best matches its own available bandwidth. The premise behind this is that user experience is maximised: a client will always request the maximum video quality possible given the resources it has at its disposal.

With a huge amount of content variations to be provided, storage requirements are vastly increased for providers. Rather than storing a single copy of the content, many versions need to be available to handle all possible requests. Given that HTTP is used to delivery thousands of files every day, it became a natural choice for serving the many variants found in an adaptive representation, and thus a new set of HTTP Adaptive Streaming (HAS) based technologies came into being. These can be broadly categorised by the fact that they rely on HTTP for their transport mechanism. Proprietary commercial solutions (such as Apple HLS [3] and Microsoft Smooth Streaming [16]) are complimented with open and

standardised techniques (such as MPEG-DASH [159]). In the case of the latter, content of different qualities and encodings are *chunked* into smaller, often fixed-length (in terms of playback), segments. These segments are then grouped together to form a single representation (an entire video, from start to finish). Alternate representations are collected together in the same manifest, which is used by the player to enable playback. If the player wishes to change representation during playback, this process is as simple as requesting an alternate representation from the manifest.

The manifest also includes annotation and metadata for each of the representations. This includes information necessary for the client to determine the most appropriate representation given its own capabilities (decoding capability and resolution) and those of the connected networks (required throughput). This ability gives the playback client the adaptability required to maximise the experience of a user in a constantly shifting environment.

Importantly, content can be organised and described in two main ways. Firstly, content can be segmented on a file-system level, with a number of different smaller files, each of which is directly equal to an individual segment. Together, these segments represent an entire video. In this context, each chunk is individually playable, affording the player the flexibility to freely swap between representations without the need to download header information. In the second method, segments can also be represented as a byte-range of a much larger file (much the same as a HTTP progressive download technique described previously in Section 2.3.1.5). By downloading a particular byte-range, the client can reassemble the file necessary for playback. This method also requires an initial set of headers to be downloaded, often before playback starts. Without the headers, the playback element will not be able to correctly recognise and process the content, as the header contains the information and structure necessary to do so.

2.3.1.7 Information-centric Networking

There is a growing number of researchers and engineers that believe that current method of connection-orientated delivery is inherently inefficient and unscalable. A number of alternatives have been proposed, typically described as *clean-slate* approaches, which require a fundamental restructuring of the technology and hardware deployed in the Internet. This is achieved by completely replacing

the existing IP provision that is used today and replacing it with alternative technologies. One such example is Information-centric Networking (ICN) [56]: networks specifically built for the purpose of disseminating information.

The ICN approach enables clients to request information by making said information addressable through a naming scheme. In order to retrieve information, a client disseminates a request using one of these names, otherwise known as an *interest*. The connected network devices are then responsible for locating, and subsequently delivering, the requested information to the client. This data is otherwise separated from its location, requesting application and method of transport. This enables the devices within the network to act as caches, and allows data to be replicated in the network. A specific example of an ICN approach is Content Centric Networking [109], a realisation of the ICN architecture has been specifically designed for the purpose of delivering content.

This approach avoids the need to deploy various competing technologies connecting to a vast array of differing services, as is the situation in today's Internet. Instead, there is a unified method of requesting data, which can be fulfilled at many levels within the network hierarchy, allowing scalability. The content caches (akin to network devices in this case) can be populated on-demand, or pre-populated, similar to modern content delivery techniques. There are also cache replacement strategies built explicitly for these networks [74], which incorporate the distance between nearby cache nodes holding the desired content. CCN also has a proof-of-concept software implementation in CCNx [35], which is designed to run alongside existing IP networks.

Despite the potential advantages that such an approach could provide, fully implementing these approaches in a real-world network would require a complete replacement of equipment and software. This is a prohibitively expensive process, that would likely obsolete all of the supporting equipment and knowledge that are used in modern computer networking. Furthermore, ICN has existing challenges that must be solved before a widespread evaluation could take place. This includes evolving the technology within the Internet to support CCN at scale [139].

More recently, there have been attempts to partner ICN and CCN techniques with complementary technologies, such as the HTTP adaptive streaming [118] described in Section 2.3.1.6. In this work, they directly integrate a MPEG-DASH compliant player with a CCN network. Through doing so, they observe that CCN

incurs a significant messaging overhead when compared to HTTP. However, they do note that CCN allows additional functionality, not otherwise possible with HTTP, such as fastest route selection and a natural network resiliency.

ICN has also been matched to the Software Defined Networking (SDN) discussed in Section 2.2.2 [148]. As ICN approaches require a fundamental change in network behaviour, SDN technology enables a compromise to be made the two: clean-slate approaches can be realised with existing infrastructure provision. Despite this, they are in fact limited by the capability of OpenFlow, their chosen SDN technology, which enforces some restrictions on the ability to forward by name.

2.3.2 Infrastructures for Video Delivery

Despite the move towards adaptive protocols and advanced networking techniques, the Internet is still best-effort; congestion, outages and latency can all have a negative impact on a users experience. A complimentary method used to avoid these issues is to use infrastructure located topographically close to the user. The aim of this infrastructure is to avoid the necessity for delivery to take place using external links, over which the provider has little or no control. In the following section, we discuss these and investigate how they have evolved alongside the changing Internet.

2.3.2.1 Web Caches

As the consumption of content increased, it became clear that a significant portion of the requests are duplicated. This results in content being delivered repeatedly, often to a group of individual users, over a period of minutes, hours or days. By storing this content nearby, an appliance could deliver the content for subsequent requests without having to continuously fetch the content from the origin server. Given this situation, tools were created to remedy the inefficiency. Software such as the Harvest web cache [73] (which later evolved into the Squid web cache [171]) allowed network operators, both small and large, to deploy caches in their network. Prior to the wide-spread deployment of dedicated content delivery networks (described later), these content caches were an effective means for operators to cache content. They were primarily deployed to reduce external network

traffic, which often had to traverse metered transit links. By deploying a cache, an operator could effective reduce their operational expenditure.

Web caches typically operated as a proxy; traffic was directed to them by various means and then inspected before a policy was applied. This policy determined the type of traffic to be cached. If the request matched content that was destined to be cached, the appliance would first check if it already had a content object which could satisfy the request. If possible, the content would be delivered from the cache (a *cache-hit*). However, if the cache did not have a copy of the content, the cache would request the content from the remote server (a *cache-miss*). This content would then be delivered to the requester, as well as being stored on the cache to serve future requests.

For the most part, cache implementations such as Harvest and Squid did not require the participation of the remote server in the process; they were a transparent appliance from their perspective. However, depending on how the traffic was redirected towards the cache, manual configuration would often be necessary on the end-user's device. For example, if the cache was operating as a gateway proxy, the user would be required to configure their application or browser to point at this gateway for external connectivity. With the advent of the *Bring Your Own Device* concept, in which users bring their own devices to connect to either a wired or wireless network, disseminating the required configuration became increasingly complex. Importantly, these cache implementations were also tailored primarily for caching static HTTP traffic. This limits the utilisation of protocols such as RTSP and RTMP, with much of the video traffic been excluded from the caching policy.

More recently, Varnish [43] has become a popular tool to be used as a HTTP *accelerator*. It is designed exclusively to work with the HTTP protocol, and acts as a reverse proxy sitting in front of a HTTP origin server. It caches the response to requests entirely in virtual memory, leaving the operating system to decide what is stored in memory and what is stored on disk. Interestingly, Varnish also contains its own configuration language that can be used to define how specific requests are handled, facilitating a level of customisation for the owner.

Content Replacement

An important consideration in the design and implementation of web caches is the content replacement policy. In most cases, storage is a constrained resource, and cannot be easily altered and expanded, particularly on the fly. To combat this, a suitable cache replacement strategy needs to be implemented. This is especially important as consumers requests and demands change over time. In order to optimise the availability of content given this inherent variability, it is necessary to evict items of content from the cache and remove them from the storage medium. There is a wide variety of content replacement strategies, many of which are described and categorised in [140].

Recency-based strategies are based upon the locality of reference given a set of requests. This locality can be either spatial (where requests imply future requests to other related objects) and temporal (where an object is requested repeatedly in a short period of time). Least Recently Used (LRU) is the most popular example of this strategy, but other variants exist.

Frequency-based strategies are often based on the Least Frequently Used (LFU) technique, and track the popularity of different pieces of content over time. This information can then be used to aid future decisions on what content should be retained.

Furthermore, there also combinations of the two previously described techniques, such as the Segmented LRU [59], which partitions the cache into protected and unprotected segments, each of which is a pre-determined size. Popular content is maintained in the protected area until evicted using a LRU policy. At this point it is moved into the unprotected segment as the most recently used object. If the content is subject to eviction in this segment (also using LRU) it will be removed permanently.

Function-based strategies use general functions to calculate the value of keeping an object. These may also take outside input into consideration, as is the case with server-assisted cache replacement [77]. In this case, the servers provide information to the cache as to the inter-request distribution currently being observed. Finally, there is also randomised strategies, which simply removes a random object from the cache. This can be combined with LRU [160] to influence the probability of said eviction.

In the next section, we discuss Content Delivery Networks: large-scale infras-

tructures used to aid delivery on a massive scale. By introducing them, limitations in storage and capacity become less relevant due to the strong over-provisioning often employed by the operators. As such most of these networks do not employ replacement strategies, instead hosting a large set of content that is rotated according to customer requirements.

2.3.2.2 Content Delivery Networks

Over time, bandwidth increased in the environments where traditional web caches were typically found: businesses, campuses and schools. Driven by the demand for content, over-provisioning became the norm. Operators no longer needed to operate their own cache, which removed some of the burden on them to deploy and maintain such an appliance. However, there was still advantage to be gained through caching, albeit on an entirely grander scale.

Content Delivery Networks (CDNs) took advantage of this by hosting this content in a topologically centralised location [116]. In the first instance, they were used to serve the increasingly popular medium of video, only being employed for serving HTTP requests at a later time. With connectivity to a number of different access networks, a huge number of users could be served from a relative modest deployment, important for large-scale delivery requirements. This enabled comparatively similar network efficiency gains (although each request still had to be delivered repeatedly over the access network), whilst moving the onus of operation to a third party.

This shift in responsibility coincided with a change in the relationship between network operators and content providers. Network operators carry the traffic generated by user requests for items stored by content providers. These operators are serving the subscribers who pay for their access to the Internet, and so are obliged to carry this traffic. As many of the content creators are driven commercial enterprises, whose main revenue stream was derived from either the subscription of users to their service or through advertising placed alongside their content, it became increasingly important for them to ensure that the user received the best possible quality of experience. It has been shown that users are very sensitive to even small delays in the loading of content [97]; CDNs are often employed by the content creators to guarantee that a user is not dissuaded from watching a video due to availability or poor connectivity.

CDNs ensure that content can be served to the user from a nearby location. In this case, locality is related to the number of network hops required to fetch the content. CDNs achieve this by simultaneously deploying in a number of Points of Presence (PoPs) [158]: centralised points in a nations network infrastructure where most access networks connect to the wider Internet. This ensures that users received a consistent standard of service, regardless of where they are located. CDNs also actively manage the load to ensure that they have sufficient provision to meet the anticipated demand. This scaling ensures that users always have access to their service, especially important when they are paying to do so. This availability also helps to combat unexpected events, such as flash-crowds; where content becomes very popular, very quickly. By localising the demand, this can be handled without impacting other users or the origin servers. Similarly, the same techniques can be used to prevent malicious Distributed Denial of Service (DDoS) from having a widespread impact on the overall user experience.

Without replicating the content close to the edge of networks, it would be necessary for content providers to host all of their own content. In this case, not only would each request have to travel across a wider number of network hops, but it would also require the provider to scale up their own infrastructure significantly. In the early days of the Internet, hosting content on a single server may have sufficed for the relatively small amount of demand generated. As demand grew, it may have also been satisfactory to provision multiple servers, and load-balance between them.

In the aforementioned cases, the service providers themselves would typically operate the infrastructure; as this grew, it would become a resource intensive task. In most cases, it now makes more financial sense for these content providers to employ the services of a third-party CDN [162, 2, 14], and offload all of their content distribution demands on to them. Not only do these operators have experience running networks used for the sole purpose of delivering content, they also do so for a number of different clients, enabling economies of scale and allowing them to pass the savings on to their customers, further strengthening the business case for usage by a content service provider.

Despite the possible advantages to a content provider, there is irrefutably a cost attached to using their services. As the demand for content exploded in recent years, this cost has, at least in some cases, outweighed the potential benefits of using a third-party CDNs services. Recently, a number of content

providers have made a decision to operate their own CDNs [51, 19, 166], designed and deployed in a similar way, but tailored to their own content delivery platform.

Usually, placing the content topologically closer to the user results in a more stable and reliable connection between the client and the server (in this case, the CDN surrogate). This is the case because the connection has to traverse fewer networks, and is not at the mercy of latency and bottlenecks that could otherwise interfere with the flow. This reliability typically affords the user with more throughput and thus allows them to request higher quality media. When we consider that user expectations of media quality are consistently rising, this enables content providers, via the collaboration with CDNs, to meet this demand.

The composition of a CDN can vary greatly from provider to provider [137], and often change over time. However, these deployments can often be generalised into a number of key components:

- *Surrogate or Replica Servers:* These servers are the core infrastructure of a CDN; they replicate the content precisely in order to serve it once requested by a client.

- *Origin Servers:* The origin servers are where the master copy of the content is located. This may be a server operated by the content provider, or a designated server within the CDN operators infrastructure. Typically, changes in the content made on this origin server will be replicated across the surrogate servers, as the origin server is deemed to be authoritative in this respect.

- *Clients:* The clients are the end-user devices requesting the content. They may be located anywhere on the Internet, connected through a multitude of networks and technologies.

- *Redirection Infrastructure:* The redirection infrastructure ensures that a client requests a cached piece of content located on one of the surrogate servers. Importantly, the redirection mechanism must also ensure that the redirection is destined for a *nearby* surrogate server. This ensures that the client experiences the most consistent network conditions, which should result in a higher quality of service.

- *Distribution Infrastructure:* The distribution infrastructure is responsible for delivering the content, stored on the origin server, to a number of surrogate servers. Distribution can be achieved in a number of ways, although a composite is often used dependent on the scenario in which the surrogates are deployed:

  - *Proactively:* A set of content is duplicated on the surrogate server *before* it is ever requested from a client. This ensures a client request can be handled in the most timely fashion, as the surrogate server does not have to retrieve the content from the origin server before delivering it to the client.

  - *Reactively:* No content is stored on the surrogate server by default. Instead, content is only retrieved from the origin server when it is initially requested by a client. This means that the initial request will be typically slower, but subsequent requests will benefit from the surrogate server having already stored the content. That content will remain in the cache until is evicted through a cache replacement policy. Despite the small performance disadvantage, this method of content distribution is useful in cases where the cache has a very limited storage resource, and cannot store a large catalogue of content, as would typically be the case with proactive content caching.

- *Accounting and Monitoring Infrastructure:* The accounting infrastructure within a CDN enables the provider to accurately charge for the usage of the CDN to their clients. In conjunction with monitoring, it also enables them to monitor the health of the other elements of the CDN, ensuring that content remains available and surrogate servers are reachable at all times. It also allows the CDN provider to make informed decisions about what content should be situated on which surrogate servers; by analysing previous request data, there is the potential to locate content in such a way as to maximise the cost savings, efficiency gains or quality of experience. In reality, a balance of each of these metrics is probably required.

Content Placement

Content Delivery Networks (CDNs) are distributed in their very nature; content needs to be available to customers regardless of their location. A core part of meeting this requirement is to ensure that content is located in as many appropriate locations as possible. In the case of commercial CDNs, the location of the content (and the nodes that serve it) are a closely guarded commercial secret. It is in fact part of their business model to place these nodes efficiently and conveniently, something that their competitors could easily replicate. Despite this secrecy, a large number of works have attempted to benchmark, observe and measure the behaviour of commercial content delivery platforms and their traffic patterns [162, 50, 51, 166, 54, 52]. These provide an interesting insight into the patterns for content placement, and demonstrate the diversity and variance in such strategies.

As one of the core aims of a CDN is to improve user Quality-of-Experience, moving content even closer to the user is an important capability. This should, under the same circumstances, allow the CDN to deliver content at a higher quality level and with less variance. Previous work has observed the relationship between CDNs and Internet Service Providers (ISPs) [99, 96]. In some cases, it is not possible to deploy equipment into the access network (for cost or access reasons), yet there are clear advantages to cooperation on the matter [55]. A flexible and coordinated approach is clearly required in such cases in order to achieve maximum efficiency and benefit.

To combat this disconnect between ISPs and CDNs, PaDIS [141] is a tool which enables the transfer of meta-information without revealing internal topologies and/or deployments. In particular, it allows ISPs to identify, and subsequently use, both server and path diversity towards CDNs. This can provide benefits to users who should receive improved throughput capability. From an ISPs point of view, it also allows greater control of traffic which can be directed at will and thus accurately engineered.

Karsalis et. al [114] consider the case of content replication in light of extensible, distributed cloud resources and a converged wireless-optical-datacentre virtual environment. They conclude that distributed approaches to solving replication offer significant benefits when compared to centralised off-line alternatives. This works also highlights the additional challenges that increasingly distributed

and available infrastructures bring to effectively placing content given a multitude of constraints. In this case, these include the cost of object retrieval, the size of the object in storage, the amount of requests generated over a fixed interval and the probability of an object being requested. Papagianni et. al [136] again consider the replica placement in the context of cloud resource availability. This is part of a larger, overarching process designed for use in a multi-provider networked cloud environment, which offers benefits in terms of operational cost and computational complexity.

2.3.2.3 Redirection Techniques

As CDNs became popular, the redirection techniques utilised in historic web caches no longer became scalable across many thousands of users. For example, using the proxy gateway approach would be infeasible given the traffic load. It would also be difficult to manage connections to multiple CDNs simultaneously. Clearly, the redirection techniques need to be scalable and support the simultaneous use of a number of different CDNs. In the modern Internet, a number of methods are used to achieve this redirection; these are described in the following section.

HTTP

In the context of a request made over HTTP, the HTTP 1.0 specification [63] defines a number of response codes that can be used to redirect a client to an alternative location. For example the *301* code is defined as *Moved Permanently*. This indicates that content is no longer available at the current location, and all future requests should be directed to a given URI. To supplement this, the *302* code was originally described as *Moved Temporarily*. Despite this specification, most browser implementations would typically modify the subsequent request method to a GET, regardless of the previous method used.

In order to rectify the ambiguity in the use of the *301* and *302*, three further codes were added in the HTTP/1.1 specification [92]. Specifically, the *303: See Other* and *307: Temporary Redirect* codes were added to explicitly define this behaviour, with *303* mandating a GET request, and *307* requiring the preservation of the original request type. Similarly, *308* was added to signify a *Permanent Redirect*, and just as *307*, does not allow the request type to change.

There is a clear limitation for using these HTTP codes to modify requests for content: it requires the participation of the end servers, and mandates that they specifically have to have knowledge of where the content is now located. In a distributed system, where the content can be located in multiple locations, dynamically rewriting the target of a redirect on a per-client basis is not a particularly scalable solution. Furthermore, if the content is located in multiple alternative locations, as can be the case with a video stream, successive redirects will have to be issued to the client. This requires inter-server collaboration, and would likely require large amounts of messaging overhead and coordination. It would also be infeasible if the servers belong to different organisations or delivery networks.

DNS

DNS resolution is a core part of any connection establishment process initiated over the Internet. Before a client can request content from a remote server, it will seek to resolve the given URL to an IP address using a lookup to a DNS server. In the case of redirection, this process can also be used to direct requests to a topographically closer cache [53]. To achieve this, the DNS server will inspect the source of a request, and associate this with a topographical region. A resolver will then return a response to the client, the contents of which will instruct the client to request content from a *nearby* edge cache.

This method of redirection is by far the most commonly used technique in use in the Internet today. However, when used in conjunction with a traditional web cache, it can result in inefficiencies. As these caches rely on the URL as the unique identifier for a piece of stored content, each object will be treated individually. However, as the DNS resolution may not be the same between clients, the same piece of content can be delivered to different clients, and thus stored under different identifiers within a cache. This leads to cache duplication and wasted disk utilisation.

Under normal circumstances, a DNS resolver should always resolve clients located within the same network to the same surrogate server. However, many of these DNS resolvers also act as a type of load balancer, ensuring requests are distributed between a number of surrogate servers, each of which evidently has its own address. DNS redirection can also be problematic in cases where a user

utilises a third party DNS resolver, rather than one provided by the ISP; this may result in ignorance of content located within an ISPs network [53], leading to the inefficient delivery of content.

The usefulness of DNS-based redirection is also diminished when the client itself caches the DNS response [152]. This caching can result in a slower response to failures and changes in demand, with the client not aware of changes in the availability or location of content. To address this, content providers use low time to live (TTL) values in their DNS entries. This in turn results in frequent DNS cache misses, adding additional latency to the request process.

Transport Layer

Another approach to request routing is to do so at the transport layer. Typically, this requires the introduction of a request routing middlebox or appliance, to be used as an initial gateway. This intermediary will then select a surrogate from a connected group, and facilitate a connection between it and the client. Once this connection is made, the surrogate will typically deliver the content to the user without traversing the middlebox again. This enables the maximum possible throughput to be used, without the performance penalty of traversing the middleware on the throughput intensive return flow of traffic.

This approach is often used in conjunction with another approach to ensure that requests arrive at the appliance. It can therefore be seen as a complimentary technique: it offers fine-grained control and redirection, but only once a requested is routed appropriately to it. Using an intermediary appliance also requires the need to purchase, maintain and house said equipment. Changing the behaviour of a device can also be a time-consuming process, especially as there is no standard technique of interacting with devices across vendors.

Anycast

This method utilises the behaviour of IP packet routing to select the nearest possible surrogate server. This is typically achieved by using routing protocols (such as the Border Gateway Protocol) to announce the same IP address from many different places within the Internet. When a request for content is sent from a client, the nearest router will automatically forward the packet to the nearest surrogate server, which should theoretically provide the best service to the client.

This technique requires a deterministic approach; identical requests can be handled in different ways if the routing table differs in any way. For a connection-orientated protocol, such as HTTP, this approach can lead to clients attempting to connect to a different surrogate during a long-lived connection, such as that found in video playback. As these surrogates do not share the same connection state, reconnects will occur, which can disrupt availability and thus playback.

The same deterministic nature of Anycast also has implications for the content catalogues stored on a surrogate server, as not all surrogates will replicate the same set of content. This can result in inefficient behaviour, and lead to increased and/or variable content delivery times.

2.4 Infrastructure-assisted Applications

Section 2.2 presented current trends towards the softwarisation of networks and the services that support them. This process facilitates a new generation of network applications that have full control over the underlying infrastructure; whether that be network, additional compute resources or storage. Through this tighter integration, applications can use these resources to more flexibly assist them in their operation.

A wide range of actions is available to this software, from modifying the forwarding plane in some way, to creating new instances of a function in response to demand, anticipated or otherwise. In the remaining sections we examine some of these applications. They are categorised broadly into different groups dependent on their purpose or goal. We also highlight a number of technologies designed to aid developing with this new found flexibility, including specific languages and tools used to determine the correctness of a technique or application. These are found in Section 2.4.5.

2.4.1 Switching and Routing

Computer networks provide switching and routing as fundamental functions. However, the softwarisation of networks has re-established the potential for innovation in this long-established technology domain. For example, Routeflow [147] proposes the use of centralised engines to handle routing behaviour. The computed routes are then realised in the network through the use of OpenFlow, which

pushes flow rules into switching hardware. This enables novel behaviour that was near-impossible to achieve in existing networks, such as secure inter-domain routing and optimal best path reflection.

OSHI [149] eases the move towards full Software Defined Networks (SDN) by providing a migration path from traditional IP equivalents. This is done in the context of enabling a hybrid SDN/IP backbone, which allows existing networking behaviour to co-exist with SDN-enabled services such as VPNs, Virtual Leased Lines and Traffic Engineering. To this end, they created an OSHI node, which contains both IP and SDN functionality on a single switching device.

In [98], the authors outline the role and function of a Software Defined Internet Exchange (SDX): a modern replacement for the ubiquitous Internet Exchange Point (IXP). This is designed to overcome some of the shortcomings of existing BGP routing through the use of SDN, which enables fine-grained control and matching based upon multiple header fields. To demonstrate this, they create an application where two networks peer only select traffic, such as streaming video. Through experimentation, they show that their solution allows hundreds of participants to advertise full routing tables whilst maintaining sub-second convergence.

The applicability and suitability of SDN in production networks is compounded in Google's decision to utilise the technology in a private WAN connecting their worldwide data centres [110]. This move aims to maximise capacity by maintaining a constant 100% link utilisation. It also enables end-to-end programmability, which has been demonstrated through the centralisation of their traffic engineering logic; this allocates bandwidth amongst competing services based upon application priority, communication patterns and network availability.

2.4.2 Security

Security is an ongoing threat to the uptime of any infrastructure. The continued safety of such has been the focus of much research, and recently, academics have looked towards the increased flexibility in infrastructure to assist in securing these networks.

FRESCO [155] is a framework designed to facilitate the development of OpenFlow-based detection and mitigation applications. Implemented as an OpenFlow appli-

cation itself, FRESCO aims to enable researchers to implement and share different security modules, including a scan deflector service and a P2P malware detection service, which are implemented as examples. These applications introduce minimal overhead, as well as providing the necessary functionality in significantly fewer lines of code.

A core part of modern security measures is the ability to sample traffic. FleXam [157] describes the need for additions to the OpenFlow specification in order to sample effectively, noting deficiencies in existing techniques. To remedy this situation, they propose the design of an extension to enable applications to define not only which packets are sampled, but also which part of the packet should be sampled and where it should be sent.

CLOUDWATCHER [156] is a tool that can be deployed in cloud environments where conditions are constantly changing and topologies are often large. CLOUDWATCHER addresses this environment by redirecting packets to be inspected by existing network security devices. This behaviour can be configured through the use of simple policies determined by the network administrator. This work provides a number of algorithms for detouring packets, and evaluate each in terms of the time necessary to generate the necessary flow rules to be installed in the network.

A facet of infrastructure-assisted applications is that they are granted unparalleled control over networks and their resources. However, this does not prevent malicious actors from threatening the security of the infrastructure itself. Similarly, it does not prevent multiple applications from requesting potentially conflicting resources, particularly in the network later. FortNOX [142] aims to counter this by providing role-based authorisation and security constraining enforcement as an extension to an existing OpenFlow controller. FortNOX operates by checking for flow rule contradictions in real-time, and attempting to remediate any problems. It also adds additional flow rules so avoid circumvention of rules imposed by the application. This work is demonstrated through a prototype which offers minimal overhead compared to the standard controller when determining if a flow rule can be inserted.

2.4.3 Resiliency

Network infrastructure has also been deployed to aid resiliency. For example, in [153], the authors utilised OpenFlow to provide a unique fast-failover mechanism which allows networks to rapidly recover from hardware failures. They compare the performance of this mechanism with existing MAC-based re-convergence and client-initiated recovery using ARP. In all cases, it exhibits a faster switch-over time with lower packet loss.

There is also a significant body of work concerning the use of Software Defined Networking (SDN) to offer load-balancing. An initial demonstration of this [101] shows the advantages of taking into account both the congestion of the network and the load on the servers to better adapt to prevailing conditions. In response to this load, OpenFlow is used to reconfigure routes accordingly.

In [169], the authors note that the functionality provided by dedicated hardware-based load-balancers can be provided at a lower cost using commodity network switches compatible with OpenFlow. This work proposes the exploitation of the *wildcard* functionality within OpenFlow to reduce the amount of necessary rules, and thus load on the controller. They present a number of algorithms which automatically adjust to changes in policy without disrupting existing client connections.

Furthering work this in this area, [115] addresses load-balancing in the face of multiple concurrent services. Using Flowvisor [154], the network is logically partitioned into a number of slices, each controlled by a separate controller, with each slice belonging to a single service. With this, each controller implements its own specific logic dependent on the service it is carrying. For example, web server load balancing may require a different procedures to an e-mail server. Through a prototype implementation, they discover that although this approach is viable, the existing generation of OpenFlow-capable forwarding equipment did not offer sufficient performance to make this approach viable in production networks.

Taking the ability to control the network further, [121] proposes a method of load-balancing that offers increased performance when compared to two competing load-balancing techniques. LABERIO again relies on OpenFlow as a tool to modify the network, which allows it handle load imbalance even during the life of an existing network flow, something that other existing approaches fail to consider.

2.4.4 Data Centre

The data centre environment has also garnered a significant amount of attention from the academic community. CrossRoads [127] looks to ease the migration of Virtual Machines (VMs) between these facilities by providing a layer agnostic network fabric that enables seamless live and offline mobility of VMs. A prototype was realised with OpenFlow and showed a negligible performance overhead compared to a regular network, even outperforming it in some cases.

Similarly, [65] proposes the use of OpenFlow as a tool to aid inter data centre connectivity. This work claims that there is significant complexity in the topology and configuration of their networks, and through the abstraction of internal configuration, the process of interconnecting them can be made significantly easier. They demonstrate their solution using inter data centre VM migration, and show that their solution offers reasonable delay values in doing so.

Migration also forms a core part of the XenFlow [129]. XenFlow aims to provide flexible virtual networks in coordination with the Xen hypervisor [45]. This includes the ability to isolate networks, as well as provisioning networks with QoS guarantees. During migration, their prototype offers performance better than the existing Xen hypervisor mechanism, without the need to create tunnels or losing packets.

Duet [95] proposes a combined hardware and software approach to data centre load balancing. Their prototype offers all the benefits of a software-based load balancer, including low latency and high availability, at no additional cost. This is achieved through the utilisation of the switches themselves, in which they embed the majority of the load balancing functionality. When combined with a small deployment of software-based load balancers, a hybrid solution is created. This solution offers a 10-fold increase in capacity, whilst reducing latency by a factor of 10. It also able to adapt to changing network conditions, including failures. Interestingly, the authors intentionally avoid the use of OpenFlow, instead relying on equal-cost multi-path routing [165] and IP-in-IP encapsulation to tunnel traffic accordingly, noting that the APIs to control these features have only just become available in vendor's switching hardware.

ElasticTree [103] looks to improve the energy efficiency of data centre networks. It tackles this using a power manager, which dynamically adjusts not only the switching fabric of the network, but also the power state of each networking

element. This is done in the face of changing load in the data centre. Through analysis of multiple strategies, and by utilising an OpenFlow-based testbed, the authors show how trade-offs between energy efficiency, performance and robustness can be tuned to offer desired performance, cost and robustness. Results indicate that ElasticTree can save up to 50% of the energy consumed by data centre networks whilst maintaining the capability to deal with peak demand.

2.4.5 Application Development

To aid the development of infrastructure-assisted applications, a number of tools and programming languages have been created. In [91], the authors coin the term *participatory networking*, proposing a common API for application to control SDNs. This is designed to allow applications to work more effectively with the network, including providing predictability, performance and security. To achieve this, they address the safety concerns surrounding presenting a network topology and its details, as well as the resolution of conflicts across different user's requests. To demonstrate the applicability of this API, they port four well-known applications to used the API, including Hadoop [83] and ZooKeeper [107].

Atlas [143] proposes an agent-based approach to enable fine-grained traffic classification with SDNs. As existing technologies are limited to information gathered the lower layers of the networking stack, agents located on client devices provide ground truth data. When combined with data derived from the SDN, a machine learning approach can be used to provide accurate traffic classification. Through a prototype implementation using Android devices, they show a 94% success rate.

A number of programming languages have been designed specifically for developing applications for software-defined infrastructures. For example, Frenetic [94, 93] is a high-level language that can be used to program a number of distributed network switches. Taking a declarative approach, Frenetic allows network traffic can be classified and aggregated, as well enabling high-level packet-forwarding policies to be determined. The ability to reuse code and modules is highlighted in a comparison of program size between Frenetic and an existing OpenFlow controller. To further highlight the usefulness of this approach, a number of applications were implemented, including a fault-tolerant routing, a load balancer and a Dynamic Host Configuration (DHCP) server. A Python-based

version of Frenetic is also available [144]. Nettle [168] also looks to address the issue of programming networks, albeit this time taking an event-based functional approach.

There has also been a small body of work concerning the correctness of these applications. In particular, the automated testing of OpenFlow-based applications is addressed in [68, 138]. They note that given increased programmability, the likelihood of a single bug disrupting the entire network become apparent. Challenges faced by applications include the multitude of possible of inputs, the complexity of a distributed topology, and a dependency on external events. Working towards ensuring correctness, applications should be subjected to a number of packet sequences and network events. These should be derived from both simple traffic models and from the nature of the environment in which the application will be deployed. Further work includes applying this validation to a number of real applications [69], notably a load-balancer, a layer-2 learning switch and an energy-efficient traffic engineering mechanism. In doing so, the authors found a number of previously undiscovered bugs.

2.4.6 Content Delivery

Particularly important in the context of this thesis, there has been a number of related works conducted in the area of infrastructure-assisted content delivery.

In [111], the authors utilise the flexibility enabled in the data plane by Software-defined Networking (SDN) to realise path selection for specific applications, in this case YouTube video streaming. In effect, this work combines application-state information with the control element of SDN. In their experiments, it allows specific flows (all those related to YouTube) to traverse links that have favourable characteristics. Their evaluation shows benefits that far outweigh the usage of standard Quality-of-Service (QoS) flags, used to denote traffic with particular requirements. This is largely due to the time-dynamic requirements of a user, which can be easily signalled to an application layer via SDN, yet can be hard to convey using QoS alone. Similar work [82] also demonstrates the advantage of application-aware aggregation and traffic engineering, albeit in a converged packet and optical network. In particular, they show how circuits can be initialised dynamically with specific flow properties, such as guaranteed bandwidth, low latency and low jitter. These allow differential treatment of different packet

flows, in this case denominated more generally as voice, video or web.

Kaleidoscope [174] presents a prototype designed to aid real-time content delivery using both SDN and cloud computing technologies. This offers a flexible resource allocation that changes in response to demand. By modifying their usage, expenditure can be reduced whilst maintaining service quality. The prototype uses a mixture of SDN-based broadcasting, network virtualisation and the ability to dynamically provision cloud resources when needed. Live media distribution has a number of unique challenges, such as the need for real-time processing and simultaneous distribution to multiple clients.

Software-defined networking has also been used to aid delivery of content over wireless networks [135]. Addressing a deficiency in the cooperation between network operators and content providers, they suggest a unified control plane that is realised end-to-end. This includes both the wireless access network and the mobile core network. This can then be used to dynamically control traffic flows in response to changing network conditions, maximising the users quality of experience and avoiding traffic traversing links which may negatively impact such. Similar work [79] aims to optimise content caches in LTE mobile networks by exploiting the inherent flexibility in software-defined networks. Not only do they reduced expenditure on equipment (which can now be generic rather than specific to mobile networks), it also enables caches to be relocated dynamically and without the necessity to consider physical topologies in the process.

Liu et al. [119] take a different approach to addressing the inherent uncertainty in today's networks and delivery infrastructures. They make the case for a coordinated video control plane that uses measurements derived from the client to dynamically adapt playback to maximise performance. These adaptations are based around the selection of CDNs in conjunction with the bitrate requested. Through collaboration between network operators and content delivery networks, a network-wide view can be established. This ensures that various policy goals and constraints can be met, whilst simultaneously improving end-user experience. This collaboration is also realised in [172], where the authors instead utilise software-defined networking to enable the sharing of information between Internet Service Provider (ISP) and Content Delivery Network (CDN). This provides the basis for optimisation to take place, as the accuracy of redirection can be improved (avoiding costly transit costs to the ISP) whilst maximising the experience provided to the user (achieving the CDNs primary purpose). This is possible due

to the control logic of the ISPs network being in software, allowing the CDN to build an application to interact with said logic.

2.4.7 Moving Forward

In this section, we have outlined existing work in the area of infrastructure-assisted applications. Despite the relative infancy of this field, we examined work in the area of content distribution in Section 2.4.6. The innovations in this area have been matched by a clear movement within the video delivery space to produce similarly open and adaptable technologies that make the best use of the resources at their disposal. As described in Section 2.3, these fundamentally change the way that video is delivered over the Internet.

As a result of developments in the networking, infrastructure and delivery technology fields, a new approach to content delivery is required. This is driven not only by the availability of these new technologies and the subsequent benefits that they may bring, but also out of the necessity to innovate in order to satisfy the continuous growth in the demand for content. Bringing all of these elements together presents the opportunity to create more adaptive and responsive methods of delivering both content and services.

In this section, we have shown a precedent for increased network and service collaboration, with this early work demonstrating some of the potential benefits possible. Yet there is a clear need to prove that the use of Software Defined Networking is the correct and appropriate tool for the situation. This can be quantified in a number of ways, but the benefits should at least be tangible to the end user. This can only be demonstrated through implementation and extensive evaluation.

The extension of programmability towards the process of content delivery has also not yet been explored in literature. This presents some interesting prospects in terms of specifying the behaviour of distributed content delivery platforms, yet continues a notable trend in this area. In the next chapter we outline the motivations behind a next-generation content delivery platform. Taking into account existing infrastructure architectures, this thesis proposes a design which also encompasses a number of novel features. This includes an emphasis on utilising the aforementioned new technologies, and deploying them as tools to aid the process. The design must also consider evolution in delivery technologies,

particularly those that are adaptive. These may introduce additional implications to the process, something that existing work has not yet considered.

2.5 Summary

This chapter has presented background material and related work straddling many different research topics. To begin with, we described the continued importance of the Internet in today's world. We then identified a particular trend in the field of networking, which moves towards softwarisation: transitioning away from fixed hardware-based functionality towards more flexible and agile software alternatives. Importantly, this is done without compromising current performance and scalability requirements. This movement has its roots in programmable networking, the concept of which has seen somewhat of a resurgence with the recent popularity of Software Defined Networking (SDN). When these same features and desires are applied to the myriad of functions that underpin modern networks (under the title of Network Functions Virtualisation (NFV)), further benefits become evident.

A further topic described in this chapter is video, and particularly the distribution of it over Internet architectures. This too is a rapidly evolving field, with a distinct trend towards the use of commodity hardware and adaptive delivery techniques. To support the ever-increasing demand for this content, there has been the need to deploy additional infrastructure and apply novel techniques to direct requests towards local duplicates. As noted in this chapter, this enables scalability as well as efficiency; fewer requests are handled at the origin server.

Combining such infrastructure with the application that it supports has led to the concept of Infrastructure-assisted Applications. Although it is to be noted that this is a relatively new, and thus underdeveloped, area of research, it has nonetheless been viewed from a number of different perspectives, including security, resiliency and content delivery. The core commonality across all of these related items of work is the ability for an application to somehow influence the form and structure of the underlying infrastructure in a way that aids the application in some way.

It is through this process that we have identified some shortcomings in the area, particularly in the scope of content delivery. These are follows:

- There are clear benefits to content delivery by using an infrastructure-assisted application, but these have yet to be realised.
- The application of SDN and NFV concepts in this domain has the potential to impact a number of the underlying functionalities, such as content delivery, infrastructure provision and request redirection. Understanding these is of paramount importance if provision is to adequately match increased demand.
- There is also a distinct lack of investigation into how these infrastructures are impacted with the use of emerging standards for delivering video, which may have further, as of yet unconsidered, ramifications.

Chapter 3

Design

In this chapter, we discuss the motivation behind the need to develop universal content delivery platforms which are both open and programmable. We discuss how emerging technologies are applicable in this context, and how we can use them to create a platform for the next-generation of software-based virtualised functions. We present a novel architecture for achieving this, illustrated through a comprehensive design. In Section 3.1, we discuss the motivation behind such a design, and define the aims that such a design should achieve. In Section 3.2, we discuss how these aims are met in a comprehensive design analysis.

3.1 Motivation and Aims

Given the background and related work described in Chapter 2, it is clear that there is significant effort centred around the delivery of content and the infrastructures that support such. Over time, these have developed into a large-scale ecosystem which ensures that content is available to the maximum number of users and delivered in the timeliest fashion. However, existing platforms do not make use of advances in technology, nor consider changes in thinking concerning the design of network services. In this chapter, we outline a number of core requirements central to both present and future content delivery platforms. Further to this, we consider the motivation behind a universal approach which is both open and programmable. This includes the use of emerging technologies and service paradigms. Ultimately, this enables both new research opportunities as well as disruptive business models.

3.1.1 Content Delivery Fundamentals

Any content delivery platform must satisfy a number of core functions. The most important of these is that it must ensure that it can always satisfy a users request. This can be achieved in a number of ways, including pre-loading content ready to serve to a client, retrieving content in response to a user request, or serving content already stored within the cache. A cache in this case is a collection of objects or files that have been previously retrieved from an origin server, and now retained to enable requests to be handled internally. Regardless of the method, this must be completed in a timely fashion, ensuring that the user experiences no degradation in service. A comparison in this case can be made between fetching content directly from the origin server (where the content is located by default) or fetching the content from a cache.

The baseline scenario for a cache is that content is delivered at approximately the same latency and throughput as though it was requested from the origin server. However, in ideal circumstances, the cache should be able to handle a request in a much smaller time frame and with a higher observed throughput. Evidently, achieving this will at least partially rely on the placement of the cache, and the resources that this affords. Nonetheless, the design of a cache should ensure that no impairments are introduced in this process.

Given that caches are often deployed to ensure availability of content, a design should be capable of handling a large number of requests simultaneously. The amount of incoming requests will largely fluctuate given the time of day and the placement of the cache (and thus the number of potential clients). A design must therefore be able to handle the maximum possible number of requests expected in a peak period. During these times, it is again important not to degrade the service received by the user. In particular, it is imperative to avoid content becoming unavailable due to service interruption, especially as a cache is typically deployed to avoid such a situation.

Where a high volume of requests may overwhelm a single server, such as the origin server, the aim of a cache is to ensure that this load is more evenly balanced. As before, this can be partly accomplished through the intelligent placement of content replicas. Similarly, scaling the hardware deployment can also aid in ensuring that there is sufficient resources to meet the level of requests; through provisioning more capability, more simultaneous requests can be handled.

However, it is important that the underlying design makes the most efficient use of resources, regardless of what is provisioned. This prolongs the need to expand resource allocation, especially important when this is a long-term process that is both time and capital intensive.

Another fundamental aim of any cache is to provide network efficiency. In addition to taking load off the server, a direct effect of the caching process is a distinct reduction in the network traffic flowing towards the origin server. This includes any request having to traverse the network between the client and the aforementioned server. In some cases, this may even be a metered transit connection. The efficiencies gained by serving the content locally, and thus reducing the unicast flows, ensures that only the initial flow has to be handled in the external network; that is, there is a single flow that retrieves the initial object from the server. Instead, the unicast flows are now internalised between the client and the cache. This cache would be based either within the same network, or at least at the edge, to attain maximum efficiency.

Another core design feature, present in all modern content delivery networks, is that they are capable of being distributed. By hosting content in multiple locations, providers can ensure a minimal round-trip time when replying to requests for content. It also allows the tailoring of caches dependent on the connected customers, hosting content specific to that region. To realise this distribution, a level of coordination and cooperation is necessary. This is achieved through the passing of messages between constituent entities. A design should therefore aim to define this process in a rigid way, allowing entities to join and leave seamlessly and as required. This may be facilitated by the specification of a discovery mechanism. Furthermore, there should be a level of abstraction which provides operators with the ability to address a group of elements as one. This is particularly important in an environment where similar elements may co-exist closely together, and presents an opportunity to simplify the control and command of these elements.

3.1.2 Programmable Control

In Section 2.2.2, we described Software Defined Networking (SDN), a new paradigm for computer networking. It allows for greater management over the underlying forwarding with in a network. Traditional control decisions are separated from

hardware-based appliances and placed into software. This enables programmability in the network, as applications can define custom network behaviour and manipulate this on-the-fly.

Thus far, these concepts have not been applied to the field of content delivery. A design should aim to evaluate the effectiveness of this technology to forward requests for content to a nearby cache. A programmable approach to this process may have other advantages, such as the ability to modify existing redirects almost instantaneously. There has been numerous comparisons made between the effectiveness of different redirection techniques [112], including a focus on the predominant DNS-based technique used today [152]. However, due to newness of SDN, it has not yet been considered in this comparison.

The design should also aim extend this programmability to the cache deployment itself. A well-defined interface can be used to achieve this by allowing critical operations to be determined and modified during runtime. To aid programmability, this should also aim to provide feedback and statistics to the user. This allows the development of applications that are reactive in nature, ensuring that the decisions made in the underlying cache are both timely and appropriate.

Programmability should also be extended to the underlying infrastructure. In fact, the ability to dynamically allocate resources according to the needs of a program is a core concept within Network Function Virtualisation (NFV). As described in Section 2.2.3, NFV is a concerted moved towards more flexible and virtualised functions for use in the network. Given that content delivery networks are one of the recognised use cases [47], it is clearly a pertinent example.

In existing content delivery networks, CDN nodes (the entities that serve the content to the users), are physical devices, dedicated to the purpose. This approach can be viewed as inflexible, as the hardware is provisioned in order to serve content during peak demand. During periods of unexpected demand, such as flash-crowds, this resource provision can be challenged. However, upgrading the hardware requires foresight and preparation, as well as increased capital expenditure. Hosting dedicate nodes per CDN can also increase operation complexity, especially if they are managed devices. As mentioned previously, content delivery is a constantly changing field, and the ability to create new software rather than hardware is a distinct advantage in order to remain agile.

In order to address these challenges, a design should ensure that the constituent components of a CDN be realised entirely in software. These elements

can then be run on generic virtualised infrastructure, shared with other components and functions. This allows the CDN to release resources when they are not in use. Given the periodic nature of user requests [163], a solution should be able to dynamically resize the provision when demand is not present.

3.1.3 Open Processes and Interfaces

In today's world, there is an ever present trade-off between closed and open innovation; each has their own merits. In terms of content delivery platforms, many of the existing solutions are proprietary and a closely guarded commercial secret. When a customer approaches a content delivery network, they do not necessarily need to understand the details of how and where their content is delivered from; they are simply paying for a service to enable their content to be distributed widely.

However, an open solution has a distinct set of advantages, which includes transparency and an understanding of the underlying mechanisms. Given an open solution, it is possible to effectively modify the fundamental behaviour of a deployment to suit the necessary requirements. When combined with a modular design (as discussed in Section 3.1.4), it presents the opportunity to create a community in which modules can be shared and refined between users. This allows developers to build modules and distribute them to others without the need for duplicated effort, and with the benefit of combined wisdom and knowledge. In cases where content providers do not need to understand the complexity of a deployment, any implementation must be easy to deploy and maintain so that operators are not burdened with the complexity of managing such a solution.

In Section 2.4, we discussed a continued trend towards increased collaboration between content distribution networks and Internet service providers. Through the sharing of information, they hope to further optimise the benefits of delivering content from a cache. An open and accessible method of enabling this exchange of information to continue, and eventually become commonplace, is to the benefit of all parties involved.

It is believed that this information sharing may also be extended beyond the aforementioned organisations, and opened up to other potential interested parties. Examples include those that do not currently have any influence over the caching process, such as content creators. As they may have detailed and

specific knowledge of their own libraries and catalogues, this can be exploited to realise further efficiency gains.

3.1.4 Flexible Deployment

Any design should also be capable of operating in a myriad of possible deployment scenarios. For example, this could be a small-scale cache node deployed in a users home, constrained by a small amount of storage, available bandwidth and limited processing capability. Similarly, it should be possible to deploy the same solution in a data-centre scenario, where resources are abundant and it is relatively trivial to scale resources dependent on demand and availability. The same could be said for any deployment in-between these two extremes, such as in a Internet Point-of-Presence or even within an operators network. In order to achieve this, it is necessary to build a solution under a universal code-base. This is, the software can be run on a multitude of platforms. The software should also scale to the resources it has available, with the operational constraints being either automatically discovered or manually defined.

As the methods for content delivery have changed over time, the infrastructure to support such has changed with it. However, this technological innovation is a constantly ongoing process, with new techniques persistently revolutionising how we meet the demand for content. Evidently, any design needs to be sufficiently flexible to adapt these changes. In Section 2.3.1.6, we observed how delivery technologies are now moving towards the use of adaptive streaming techniques matched with commodity HTTP servers. Support for this technology should be critical in any design. In Sections 2.3.1.3, 2.3.1.4 and 2.3.1.7, we noted a number of alternative technologies, all equally capable of delivering content, but with vastly different methods of doing so. Ensuring that any design supports both current and future technologies is an important design aim. A modular design enables this; existing functionality can be replaced, as well as new functionality trialled and added, when the need arises.

Similarly, the process of redirecting requests should also be independent of protocol or request: it should also be transparent in use, and without the requirement that clients have to configure their devices or download additional software to take advantage of content delivered from the cache. It should also offer compatibility with existing services found in the network, such as firewalls.

Importantly, the realisation of the design should also work in harmony with existing content delivery platforms, regardless of their location in the Internet. This provides a viable integration path, and does not preclude services from using a multitude of means to ensure that their content is available to the greatest amount of users.

3.1.5 Summary

In this section, we have identified a number of features to be used in the design and implementation of a next-generation content delivery platform. In Table 3.1, we collect and provide a summary of each of these desired design attributes.

Core Functionality	Incur no additional impairment to the content delivery process
	Handle a large number of simultaneous requests
	Efficiently utilise given resources
	Exploit network efficiency by satisfying requests closer to the user
	Operate in a distributed fashion
Programmable Control	Evaluate the effectiveness of SDN in a content delivery context
	Explore new ways of controlling CDNs
	Extend programmability to the infrastructure
	Bring content delivery functionality into software
Open Processes and Interfaces	Provide a transparent and open alternative to existing CDNs
	Foster a community to develop useful functionality
	Define well-documented and open interfaces
	Increase cooperation between CDNs and other interested parties

Table 3.1: Feature Summary

3.2 Architecture and Design

In this section, we discuss how the aforementioned aims will be met with an overall design. This design will also become the basis for a prototype implementation, discussed later in Chapter 4. The OpenCache architecture contains a number of layers. The components in each of these layers work in harmony to achieve the goals set out in the previous section. In the scope of OpenCache, a number of existing technologies and techniques will be used, combined with novel functionality implemented in a number of new components; namely the OpenCache node, controller and the optional proxy. It also includes a well-defined interface, which should afford an unprecedented level of control and configuration to operators

over their deployment and resources. These components are described in the following section.

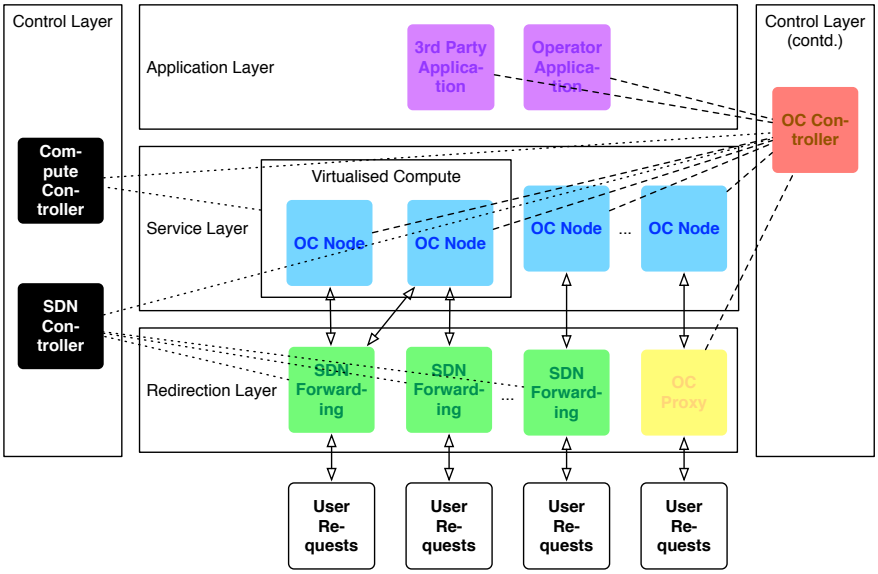

Figure 3.1: Layers of the OpenCache Architecture

The layers (and constituent components) are illustrated in Figure 3.1. At the top of this architecture is the application layer, responsible for defining the behaviour of the caches. This is where third-party developers and operators will interact with the whole deployment to create the functionality that they require in their particular deployment scenario. It is described in further detail in Section 3.2.4.

Below the application layer is the control layer (described in more detail in Section 3.2.2), and is responsible for orchestrating the remaining layers, notably the service and redirection layers. This is shown as two split vertical layers in the figure: one containing the external components and one containing the integral OpenCache controller. In reality, these would be present in the same layer; they're illustrated as such only for clarity. The OpenCache controller interacts with these existing control entities, which includes Software Defined Networking (SDN) controllers and computer controllers, in order to provide flexible resource allocation and network forwarding.

The service layer, described in Section 3.2.1, contains a number of distributed caches in the form of OpenCache nodes, whom directly serve content in response to user requests. The final and lowest layer is the redirection layer, described in

Section 3.2.3. This layer redirects user requests towards the caches using a variety of means, including direct modification of the forwarding layer and through the use of an OpenCache proxy, designed to further backwards compatibility.

It is important to note that in Figure 3.1, communication between the various elements is signified by a connecting line. If the line is dashed, this is communication using an internal protocol, designed to enable OpenCache nodes and proxies to be controlled from the OpenCache controller. Similarly, the applications interact with the controller using a protocol developed especially for this architecture. In the case of dotted lines, this signifies communication using an existing third-party protocol, designed in each case to enable the elements to be controlled or communicated with (but not built specifically for OpenCache).

3.2.1 Service Layer

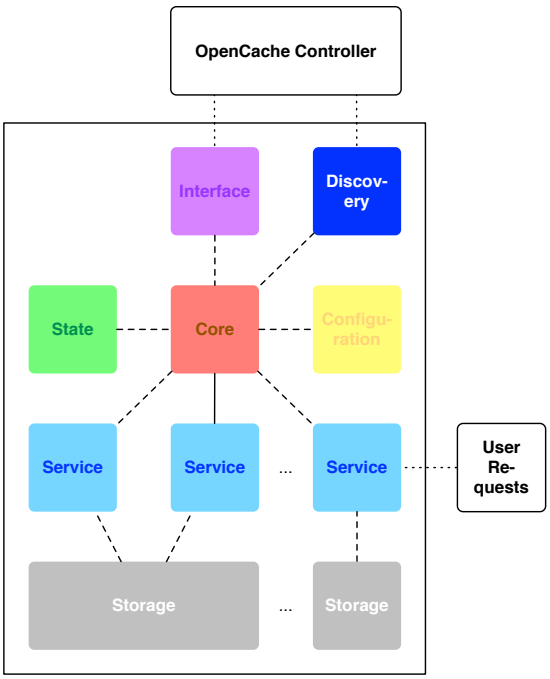

Figure 3.2: OpenCache Node Design

The service layer is at the core of the OpenCache design. The sole component of this layer is the OpenCache node, illustrated in Figure 3.2. This node consists of a number of modules, and is responsible for delivering the content to the user. As such, it is the actual realisation of a cache and responsible for storing the

content. It will subsequently deliver this when requested by a client.

At the heart of the OpenCache Node is the *core* module. This orchestrates the other remaining modules, including communicating changes in state, creating and destroying services and responding to commands made through the interface. In this case, communication between modules is established through the use of a connecting dashed line[1]. This inter-module communication is preformed through method calls and returns between each module.

Attached to the *core* module is a number of other modules. This includes the *state* module, responsible for maintaining the state of the OpenCache node as a whole, and also of the individual services running alongside it. This also allows the *core* module to respond to interface calls for statistics, as well as providing an important role for the underlying services. In this case, the *state* module maintains a representation of the content objects that are currently stored by the cache, and the location of them on any of the various storage mediums. It also maintains the state of each service, enabling the *core* module to report any issues or defects to the control layer.

When an OpenCache node begins operation, it will typically need some initial configuration, mainly so that it can connect to the wider OpenCache deployment; this is handled by the *configuration* module. As time progresses and the node begins operation, additional configuration may be disseminated to the node. In this case, the configuration will be modified and the node will operate in a modified manner.

This configuration, and in fact any communication within the scope of OpenCache, will be achieved through interaction with two modules: namely the *interface* and *discovery* modules. The *discovery* module will be used in the first instance to broadcast the presence of a new OpenCache node to the overarching control framework. Once this has been established, communication will migrate across to the *interface* module, which has a much broader and richer capability. It is through the *interface* module that the OpenCache node will be controlled and configured during its operation. This is signified through the use of a dotted line to represent external communication between itself an external OpenCache controller[2]. This external communication is discussed further in Section 3.2.4.

[1]The use of a dashed line to represent internal module communication is consistent amongst the remaining figures of this chapter.

[2]The use of a dotted line to represent external communication is consistent amongst the

The main functionality afforded by this interface is the ability to start and stop *service* modules. In the scope of OpenCache, a service represents a specific set of content to be served. This is defined through the use of an OpenCache expression: a pattern passed to the *service* module on startup that defines the content that a service should handle. This can be intentionally specific, or utilise wild-carding to enable more generic sets of content to be served. An expression can also define content from multiple locations, such as a set of geographically distributed servers. A *service* is always started, and subsequently controller, through the *interface* module: the interface provides operations such as stopping and pausing a service. An OpenCache node will typically consist of multiple services, running simultaneously. Under normal conditions, each of these services will be serving a unique set of content.

When a service starts, it will not hold any content. Without interference from the control layer, this will remain true until an initial user request arrives. Once this occurs, the service will check (via the *core* module) with the *state* module to discover if this content is already stored. As the service has just started, this is unlikely to be the case, and thus, a *cache-miss* event is created. At this point, the *service* module will request the content from the origin server. Once this has been retrieved, it will be delivered to the client, satisfying their request. Importantly, the node will also store this content.

This storage of content is achieved through utilisation of a *storage* module. Each of these storage modules is unique to a type and/or location of storage. The decision of which *storage* module to use is determined through the *core* module. Once content is retrieved and a request has been handled, the *service* module will then store this content on a chosen *storage* module. This enables the content to be served in response to future requests. Importantly, an OpenCache node should be able to support numerous underlying storage technologies and locations for such storage. Multiple services can also share the same underlying storage.

When a subsequent request for content arrives at the same *service* module, the process is repeated. However, in this case, the query to the *state* module will yield a positive result. This is a *cache-hit* event, as the content is already located on the node. As such, the result will contain the location of such content on the underlying *storage* module. The *service* can then retrieve this, and use it to

---

remaining figures of this chapter.

serve the content to the user. Importantly, the content does not have to retrieved from the origin server, reduce the load on the network. The response to this request will also be completed in a more timely manner, as there is significantly less latency involved when retrieving a file from a *storage* module as apposed to retrieving it across a network.

3.2.2 Control Layer

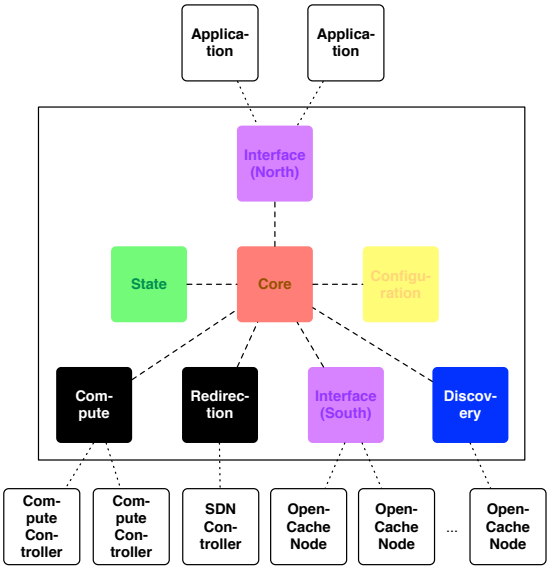

Figure 3.3: OpenCache Controller Design

As mentioned previously, the OpenCache design also includes a control layer. This layer will typically consist of one or more OpenCache controllers, and may also be accompanied with third-party controllers for software-defined networking and compute resources. The design of this OpenCache controller is illustrated in Figure 3.3. This controller is responsible for the behaviour of all of its connected nodes, with each node having a connection to a single controller. Typically, a node will stay connected to a controller until terminated or migrated to another controller.

Much like the node design presented in the previous section, a *core* module is at the heart of the controller design. This module ensures that the other modules are coordinated. As with the node, this coordination is achieved through the calling of functions located in the other connected modules. The controller will

also use the return values to determine if the calls have been successful or to gather additional information.

The *configuration* module configures the controller on start-up, much like the node. However, in the case of the controller, this configuration is rather static, and will likely remain for the duration of the controllers lifetime. As much of the application logic is located in the nodes themselves, it is unlikely that the behaviour of a controller will change.

However, the behaviour of the underlying services can be modified using one of the *interface* modules found in the OpenCache controller. It is important to note that the OpenCache controller has two of these modules. One of these module is north facing; that is, it allows external applications to interact with OpenCache controller in order to modify the behaviour of underlying OpenCache nodes. The south-facing *interface* module is responsible for direct communication with these nodes.

Often, many of the calls to the OpenCache controller on the north-bound interface directly translate to a number of calls on the south-bound interface. That is, an external application will make a call to the controller, which will then determine the destination of these calls, and send them directly to the relevant nodes. Ensuring that the correct nodes and/or services receive the messages is the responsibility of the OpenCache controller. This accuracy can be achieved because the controller has an overall awareness of the state of each of the connected OpenCache nodes.

The aggregation of calls in the north-bound *interface* module also allows the design of the interface to remain rather generic, whilst allowing the controller to handle the complexity of knowing where and how to communicate with the connected nodes. It also provides a form of aggregation to the application layer, allowing a single interface through which a number of OpenCache nodes (and the services running upon them) can be controlled in synchrony. This functionality should also allow a form of selection, in that both individual or groups of specific nodes can be addressed in a call. This is a key feature of the design when used in flexible, service-based approach.

The availability of a single north-bound interface also provides a focal point of authority when working with an OpenCache deployment, and allows security, verification and authorisation to be readily enforced. Similarly, the controller-node architecture proposed in the OpenCache design is also an important way to

secure the channel between the two layers. In this case, there is only one possible way for a third-party to interact with the OpenCache deployment, and this is through the north-bound *interface* module: it is prohibited to communicate with a node directly. This too ensures consistency in the controller *state* module, but also ensures that the responses received from the use of the interface are guaranteed accurate.

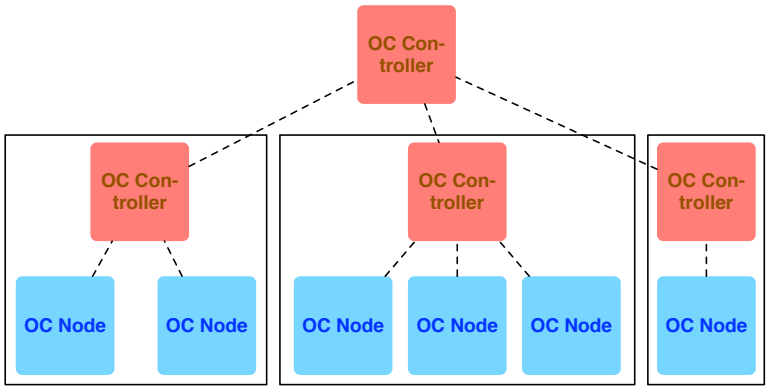

Figure 3.4: OpenCache Controller Hierarchy

The north-bound *interface* module also presents an interesting design possibility for the OpenCache architecture. Rather than relying on a single control element to command all of the available OpenCache nodes, this same interface can also facilitate a form of hierarchical control. This is possible because of similarities in the operation of the north-bound and south-bound *interface* modules, which utilise a very similar command structure. As such, the overarching controller should have no knowledge as to whether it is communicating directly with a set of nodes or through another controller, illustrated in Figure 3.4. This should help to alleviate scalability and reliability concerns present if a single-controller approach was adopted in a deployment. This design feature also furthers the aggregation facility offered in the *interface* module, by allowing a controller to be co-located with a group of nodes. This is particularly appropriate in a virtualised scenario, where a group of nodes may be commanded to work as if they were a single node; useful in situations where load needs to be distributed between instances.

In addition to these state modifying calls, the OpenCache architecture also incorporates some internal mechanisms, not exposed through the north-bound *interface* module. These are used to monitor the availability and state of both the

OpenCache nodes and the services running upon them. This should be achieved through a heartbeat mechanism, which is a periodic communication between each of the entities. Piggybacked onto these messages are fresh sets of statistics and metrics, ensuring that the *state* module within the controller can be updated with the most recent version of information. These statistics include items such as the amount of objects stored, the size of those objects, and how many cache-hits and cache-misses have been observed. The controller keeps a local copy of this information to ensure calls to the north-bound *interface* module are served quickly, without having to specifically retrieve metrics from each node. This also reduces messaging overhead, although it can be overridden if necessary.

Importantly, the OpenCache controller also communicates with other control-layer elements, including the SDN controller and compute controllers. For instance, when communicating with an SDN controller, the *redirection* module allows OpenCache to modify the behaviour of the network's forwarding layer. In the scope of the OpenCache design, this is used as the primary means to redirect requests for content towards a specific cache node.

Evidently, when a new service is started on an OpenCache node, a matching forwarding rule needs to be included in the network. In these cases, ordering and consistency is paramount; if a forwarding rule is installed prior to the service becoming ready, forwarded requests will be rejected or timed-out by the service, resulting in the user being unable to retrieve the content, and thus a degradation in their overall experience. This situation must be avoided at all costs. As a result, the OpenCache controller is the sole entity in the OpenCache design that can introduce these forwarding rules. It will only send a request to modify the forwarding plane to the SDN controller when it can guarantee that a service is ready to accept user requests. Through the use of the south-bound *interface* module, the controller can update and modify the *state* module to ensure that it always has the current view of the state of the OpenCache deployment, ensuring this consistency between the network and the running services.

The *compute interface* module enables an OpenCache controller to communicate with a distributed, virtualised hardware resource. In this case, it will likely be a form of compute hypervisor. With this interface, the OpenCache controller should be able to create, destroy and modify instances of OpenCache nodes in a dynamic and flexible fashion. As with much of the functionality in OpenCache, this will typically be in response to a call made to the north-bound *interface*

module. In this case, example usage includes requests for new OpenCache nodes in response to flash-crowds, migration of resources to different locations due to potential cost savings, or suspension of resources due inactivity amongst the user base. In each of these cases, the OpenCache controller will communicate with the computer controller to ensure that this occurs. These changes should then be reflected in the underlying resource provision, using whatever mechanisms the compute controller supports.

As with the OpenCache node, the controller also features a *discovery* module. This facilitates the connection of new OpenCache nodes, and ensure that they are then migrated across to the south-bound *interface* module once a satisfactory state has been reached. This feature is particularly important in an environment where the *compute interface* module is used. In these scenarios, a number of nodes may be dynamically brought online to in response to some criteria being met (as illustrated in Figure 3.1). The *discovery* module can be used to ensure that the nodes connect quickly and efficiently, allowing the resources to be used as soon as possible. This mechanism also enables the nodes to start with minimum configuration, allowing the controller to determine how they should operate once available.

3.2.3 Redirection Layer

As mentioned in the previous section, OpenCache will typically modify the underlying network by communicating with an SDN controller. This communication will indicate, through whatever means necessary, the desire of the OpenCache controller to modify the forwarding plane. Although this communication occurs at the control layer, the changes are realised at the redirection layer. This layer consists of two main parts, the first of which is the network forwarding described previously.

To achieve the necessary redirection functionality, the forwarding layer is explicitly modified to ensure that requests for content are redirected towards a chosen cache. One of the design aims declared in Section 3.1 was that this process should be transparent; in other words, the client should not be aware of the process. The only requirement in this case is that the SDN controller has responsibility for at least one forwarding element in the path between the client and the origin server.

As OpenCache will operate the redirection in a reactive style, rather than a catch-all proxy-based approach, modifications will only be made if a request matches a specific set of criteria. This criteria is defined in the passed OpenCache expression: a flexible approach to both coarse and fine-grained content redirection. The redirection will occur by modifying a flow within the network. As mentioned in the previous section, the flow will only be modified if there is a service already started; it is a one-to-one mapping between a modification and a service in this case. When this occurs, packets will be redirected away from the origin server and towards the cache, where the request for content will be handled. Supposing the content is already stored on the cache, this prevents the content from being delivered over the link(s) between the client and the origin server.

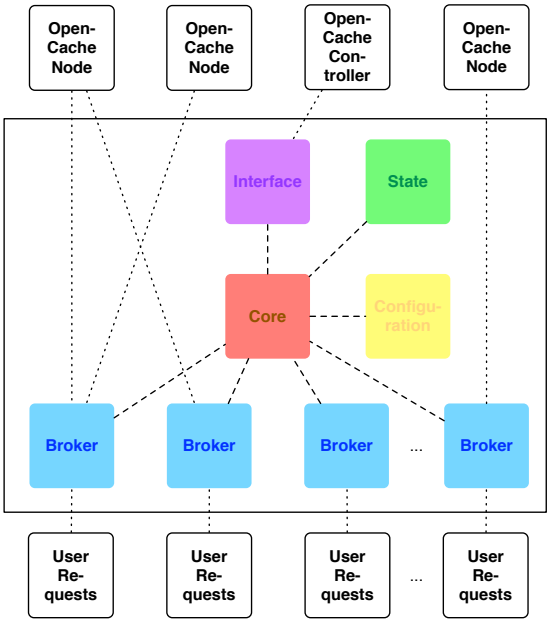

Figure 3.5: OpenCache Proxy Design

The modular design of OpenCache allows other potential redirection techniques to be used, including those outlined in Section 2.3.2.3. In order to support these, the OpenCache design also proposes the inclusion of a proxy element, illustrated in Figure 3.5. Importantly, this is an optional component, used in a select few scenarios, and is not required for the fundamental operation of the rest of the OpenCache architecture.

The OpenCache proxy is not a catch-all proxy, nor equivalent to a proxy cache.

Instead, the proxy operates by creating a number of *broker* instances. Each of these instances is responsible for receiving user requests, regardless of where they came from. Whereas an OpenCache node *service* module expects a specific set of requests, usually redirected towards the node by modifying the forwarding layer of a network, the *broker* module does not make such an assumption. Instead, it will handle each individual request, regardless of where it came from or is destined to.

Once a request is received at a *broker* module, it will lookup the potential destination for a request using *state* module. If a positive response is returned, the module will forward this request towards the correct OpenCache node. In this case, this will be a node with the appropriate service running, which will then take over responsibility for handling the request. This method allows requests to be redirected to a single point in the infrastructure; the proxy then handles the rest.

The proxy includes a single *interface* module, which is used for communication between it and the controlling OpenCache controller. It is not envisaged that OpenCache proxies will be created and destroyed on demand, and a such, a *discovery* module is not required. A proxy will typically start with a set number of *broker* instances, configured statically through the *configuration* module. The OpenCache controller will then periodically update the proxy to ensure that it is aware of the available service running on OpenCache nodes, as well as their location and current state. This ensures that the controller has full command over the proxy, and as with the *redirection* module in the controller, ensures that consistency is maintained and that requests are always handled most appropriately.

In most cases, the OpenCache proxy will not be necessary, as the forwarding provided by the underlying software defined network should provide the necessary functionality. However, it is important to consider different technologies and also the capability of existing software defined networking technologies in potential paths to migration. As such, the proxy can be used to overcome some of these challenges, and is considered critical in doing so.

Method	Input(s)	Function
start	expression, node, tag	Create a new service on a node
stop	expression, node, tag	Halt a running service on a node
pause	expression, node, tag	Halt a running service on a node, without removing the associated content objects
move	expression, from, to	Move a running service to a different node
fetch	expression, node, tag, target	Pre-emptively fetch content object for a service
seed	expression, node, tag	Define equivalent services
stat	expression, node, tag	Retrieve statistics related to a service or node
register	expression, node, tag, alert	Subscribe to a particular event occurring
describe	node	Retrieve the capabilities of a node
tag	node, tag	Define an arbitrary identifier for a node
create	expression	Create a new instance of a node
destroy	node	Destroy a node instance

Table 3.2: OpenCache Methods

3.2.4 Application Layer

The application layer is found at the top of the OpenCache architecture. It is at this layer where the actual behaviour and operation of the cache deployment is determined. This is achieved through interaction with the north-bound *interface* module found in the OpenCache controller. Through use of this interface a developer can perform numerous operations on the cache. To achieve the required programmability, a number of methods are outlined in Table 3.2. This includes a set of inputs which should be included in the method call, as well as a brief description of the expected functionality in each case.

This interface allows the developer to define the specific set of nodes that a call is being addressed towards. This can include a single node, a set of nodes or even all of the nodes under the control of the OpenCache controller. Similar granularity is offered on a service-level, with services being capable of being addressed by the OpenCache expression they were started with. With this functionality, the developer does not have to be concerned about the location or method by which these commands are disseminated; this is the responsibility of the controller and handled using its internal knowledge of the connected OpenCache nodes and the services running upon them.

To compliment this addressing, OpenCache will also support both the allocation and usage of an arbitrary description given to an OpenCache node. This can be used to define a custom subset of nodes. This is particularly useful in cases where nodes may be physically co-located (such as in a data-centre) or if a set of

nodes have a particular resource not common amongst the rest of the deployment (such as SSD-backed storage).

Core methods include the ability to start and stop individual *service* modules on specific OpenCache nodes, as well as temporarily pausing them. It also includes the ability to move existing services between nodes. These methods provide fine-grained control over individual services, providing this in full to any application.

Table 3.2 also outlines some of the advanced functionality of OpenCache. In Section 3.1, we identified some of the gaps present in current cache design, namely pre-fetching and de-duplication. As with the other functionality, these are controlled solely through the interface. In the case of *fetch*, this allows a user to define the content that they want to be fetched in advanced. This can be beneficial when the developers have access to additional information, such as popularity metrics, or algorithms designed to predict the frequency of requests. Similarly, the de-duplication functions are defined explicitly using *seed*. By taking this approach, a developer can avoid duplication through an understanding of alternative locations of their own data. This may be derived through collaboration with other content delivery platforms, or some geographical understanding of how requests are redirected (in the case of DNS usage).

As described in Section 3.1, the OpenCache architecture needs to offer rich reporting and statistics to application developers. This is realised through a *stat* method call, available to applications in this layer. This functionality enables applications to make decisions and operations given the greatest amount of information possible, including both application and resource information. Thanks to the heartbeat mechanism, a developer can ensure that any changes to the underlying OpenCache deployment are both timely and appropriate; this is, the information that they receive is current. These features are complimented with a call-back mechanism, which allows the application to react to changes in the conditions immediately, without having to constantly poll the interface to retrieve statistics.

The OpenCache architecture is specifically designed to be self-contained. This allows the resource allocation (and subsequent deployment of nodes and services) to be increased and decreased on-demand. In addition to the service-level manipulation described previously, additional methods allow nodes to be created and destroyed as required. This enables OpenCache to adapt to load on-the-fly, which

is achieved not only on an infrastructure and service level, but also in the network through an SDN-enabled dynamic redirection mechanism. The collaboration and synchrony of these elements is key to ensuring the availability and consistency of a service.

The applications that reside in this layer can be structured in any way, and programmed in any language as desired by the developer. The only requirement is that they have compatibility with the technology used to create the interface to the controller. As such, the application layer facilitates a wide range of programming types and styles, and should suit many potential applications of OpenCache.

3.3 Discussion

The architecture described in the previous section is designed to meet the aforementioned goals and aims. Much of the arrangement of the architecture is based on similar fundamental designs found in the distributed compute and software defined networking fields. In particular, the one-to-many relationship between the OpenCache nodes and the controller is inspired that used in the OpenFlow technology previously described in Section 2.2.2.1. This provides an air of familiarity to operators, whilst enabling scalability at the service layer. Importantly, it also allows the control layer to be hierarchical, which is illustrated in Figure 3.4.

The OpenCache architecture requires significant development in the cases of the OpenCache node, controller and proxy. Importantly though, it utilises existing technologies, such as the software defined networking and compute virtualisation, to lessen the necessary time spent. OpenCache is also designed to be universally deployable; that is, it should be able to run on the largest possible subset of both hardware and operating systems. This is realised by developing OpenCache in programming language that has significant compatibility, and also a wide range of support. As mentioned previously, the aim is to make OpenCache easy to deploy for user who are unconcerned with the internal details. As such, OpenCache should be packaged in a way that it can be run out-of-the-box, without the need for extensive configuration.

Chapter 4

Implementation

In the previous chapter, we described the components that make up the OpenCache architecture. We also discussed the utilisation of existing elements, such as third-party network and computer controllers, and software-defined networking forwarding elements. Once these are taken into consideration, there is a number of elements that must be implemented specifically for use in the OpenCache architecture.

The core elements (those that are necessary for the operation of OpenCache) are described in Section 4.1. The implementation of a number of additional tools are also described in this section. For example, a graphical user interface, implemented specifically to allow users to interact with an OpenCache deployment in a visual way, is described in Section 4.2. A number of example applications are also implemented in Section 4.3. These utilise the OpenCache API specified in Section 4.1.4 to demonstrate the flexibility and control offered by such. Finally in Section 4.4, we also present the implementation of a tool used in a later chapter to evaluate certain facets of OpenCache.

4.1 OpenCache Core

The items described in this section are the core components of OpenCache, specifically implemented for the purposes of realising the OpenCache architecture. These contributions satisfy the requirements established in Sections 3.1 and 3.2 by providing the fundamental capability of a distributed and controllable cache deployment. In this section, we discuss the implementation details of each.

As discussed in the previous design, OpenCache follows a modular implementation. This aids the community development of additional modules, which can simply be swapped in to alter the functionality of a given component. Each of these modules share some commonality in the way they are built.

In order to satisfy the flexible deployment requirements laid out in Section 3.1.4, all of these modules are built using the Python programming language [1]. In particular, version 2.7 of Python was used. This was chosen because it is the variant with the greatest amount of both support and libraries. Although Python has since evolved into the version 3.0 specification, version 2.x variants still see a significant amount of usage due to their familiarity and the range of libraries currently available. Python 3 changes the language significantly, and breaks background compatibility. As a result, the OpenCache core is developed entirely using version 2.7. However, there is no reason why the implementation could not be ported to run on this newer runtime in the future; at present there is no real justification to do so.

4.1.1 Shared Library

All of the modules in the OpenCache core share some basic functionality[1]. In order to reduce code duplication, these are provided in a standalone Python module. This enables this functionality to be compartmentalised and updated without effecting the implementation of the other modules. Through the use of Python's packaging system, installing this library is as simple as including it in the requirements specification for the relevant component.

This module includes support for logging, which enables an OpenCache component to supply in-depth information to the user during running. In this case, the library utilises the in-built *logging* module [15]. The configuration provided by module grants OpenCache with a flexible capability to output logs in different formats, including to *stdout* and to a file. In the case of writing to a log file, this module supports standard specifications for logging output, including time-stamping and verbosity. It also enables log rotation, ensuring that log files never consume excessive resources. This is particularly important if the module is running on a resource constrained device and the resources would be more effectively allocated to store the cache content (rather than a large log file). The

[1] `https://github.com/opencache-project/opencache-lib`

logging functionality provides important runtime information, which allows the user to determine if the cache is operating normally. Furthermore, the logging also enables debug output to be stored, important during both the implementation and deployment phases, as issues can be discovered and remedied quickly and effectively. The same logging module also enables simultaneous output; that is, logging output can be sent to multiple locations simultaneously.

The shared library also includes basic file system functionality used by each of the OpenCache modules to manage folders used for logging and debug output. Importantly, these methods are *not* used for storing the content objects used in the caching process. Instead, this is handled in the module itself, and will be discussed later.

Much of the communication desired from the interface described in Section 3.2.4 is realised using the JSON-RPC protocol, version 2.0. This was chosen as the candidate protocol not only for the internal OpenCache communication, but also for the external application interface. JSON-RPC was deemed as the most appropriate due to its ease of use, availability of supporting libraries and human readability (particularly as it is self-describing). The specification rigidly defines the format of a dictionary in JSON format, and consists a two main object types: the *request* and *response*.

When a request is made, the string must consist of the following members: *jsonrpc*, *method*, *params* and *id*. The *jsonrpc* field contains the version of JSON-RPC used, in this case 2.0. The *method* field will contain the name of the method to be invoked on the called module. *params* contains the passed variables for this call. If multiple are to be passed, this can be in the form of a nested dictionary, a list or one of the other numerous supported data types in JSON. Finally, the *id* field is used to uniquely identify the call. The object returned in response to this request must contain the same value within the *id* field so that the correct response can be accurately determined. If the *id* field is omitted during a request, the responder assumes that this is a notification, in which case no response is required.

The *response* object has a similar implementation, albeit without the inclusion of the *method* and *params* keys. In replacement of these, the *result* and *error* fields are now included. If the method has been a success, a response must contain the *result* field with an appropriate value. This may be the result of a calculation, or indication of a change in state. The *error* field has its own specification, and

within the appropriate value contains the following keys: *code*, *message* and *data*. The value of *code* corresponds to one of the codes outlined in the specification, including JSON parse errors, method not found and invalid parameters. The *message* field will contain a short description of the error, whilst the *data* field can include additional information, including further nested error objects.

The JSON-RPC specification also includes support for batching. This enables multiple calls to be sent at once, which is important in cases where the order of execution is paramount. Batched requests take the format of an ordered list. The corresponding results are also returned in the same corresponding order, with the exception that notifications will be omitted due to the fact that they do not require a response.

Given this specification, the shared library handles basic JSON-RPC functionality for each of the following modules, including handling the request and response objects, as well as creating error objects where necessary. In all cases, the library will marshal the given functions into a suitable call object, or inversely, parse the returned object into a Python object ready for the receiving module to utilise. JSON-RPC satisfies all the functional requirements for both internal and external interfaces, as we will demonstrate through evaluation in Section 5.3. Further details on the actual contents of the various interfaces is discussed in Section 4.1.4.

Finally, the shared library also provides basic functionality for the OpenCache expressions. As discussed in Section 3.2.3, these are used to define sets of content that should be cached within OpenCache. The library offers parsing support for these expressions, which are based loosely on Python's regular expression pattern syntax [37]. Importantly, the library does not dictate the functionality or behaviour given one of these expressions; only the parsing and compliance with the format is handled in the shared library.

4.1.2 Node

As discussed in Section 3.2.1, the OpenCache node[2] consists of a number of smaller components. This arrangement is illustrated in Figure 3.2. These components are realised as individual python modules, which together compose the module as a whole. For example, the *core* module is the main executable for

[2] `https://github.com/opencache-project/opencache-node`

this component, and contains the necessary code to initialise the other connected objects. Importantly, in order to retain a modular design, this module handles all communication between objects, and contains very little functionality within itself. However, it does provide monitoring capability for the accompanying modules, particularly the *service* modules. In this case, it monitors the availability and uptime of each of these to ensure that any of the statistics fed back to the controller are true and correct.

Importantly, the controller needs to be alerted to a change in the availability of these services to ensure that the redirections in place are appropriate. For example, if a service ceases to serve content for whatever reason, the redirects need to be removed immediately as not to effect the users request and prevent service being disrupted. The *core* module is also manages a central logging facility, used by both itself and the other modules that together constitute the OpenCache node. As mentioned previously, the *core* module also has responsibility to ensure that the other components contain the required methods before they are allowed to run. This ensures consistency whilst providing the flexibility for alternative implementations to be used.

The *state* module is responsible for storing information on the running *service* instances, including the various statistics outlined previously in the design. It is also used to store the location (upon the relevant file system) and details of the individual content objects stored by each of the services. Such storage and retrieval of information is well-suited to use of database, especially as persistent storage and search functionality is required. Given this, the implementation of the *state* module is rather a thin layer of compatibility for the underlying database. This is intentionally implemented as such to allow different database technologies and style to be used. For the purposes of this implementation, we use a MongoDB [18] database.

MongoDB is a NoSQL database: it an unstructured data store, whereby data is retrieved given a specific key. This is particularly appropriate in the case of object storage within OpenCache, as a request URL can be used to lookup the location of the object. Similarly, in case where de-duplication is applied to objects, multiple keys can point to the same location reference. One of the advantages offered by NoSQL is that horizontal scaling (the inclusion of additional databases) is much easier to achieve. If a cluster of OpenCache nodes were to share a single underlying database, scaling would be an important consideration.

This is especially the case when retrieval times are crucial to responding to a users request in a timely fashion.

The implementation of the *configuration* element is again realised through a Python module. OpenCache supports numerous methods of configuration, each of which is provided through a different implementation of this module. The most basic configuration is static in nature; a file is parsed from the local file system. In this case, the *conf* format is used, similar to that utilised in the Microsoft Windows INI specification. This is a simple way of describing configuration, and is supported natively in Python as a standard library [5]. Files contain numerous sections, each defined by a section header. Within each section is a number of key/value pairs, separated using a pre-defined delimiter.

A typical OpenCache node configuration will contain configuration specific to each of the loaded modules, as well as details of the capabilities and resources of the environment of which the OpenCache node is running. It will also define the various ports that the node will operate using as well as information required for initialisation, such as thresholds for storage limits and alerts. However, these will likely be modified by the OpenCache controller once the node is running. As discussed in Section 3.2.1, a node should also support a bare minimum configuration, with the remainder of the details supplied by the controller once connected. In this case, additional information will be passed through the OpenCache API (discussed later in Section 4.1.4).

The modular design of OpenCache also enables emerging technologies to be used to realise the same bootstrapping process, including utilities such as *etcd* [41]. The advantage with this, and other similar technologies, is that zero-configuration is required beforehand. Instead, a randomly generated token is used to bootstrap devices, who then form a cluster to enable configuration to be shared amongst peers.

Other elements crucial to the flexibility of the OpenCache node are the *interface* and *discovery* modules. The implementation of the *interface* module is relatively simple, in that it is a JSON-RPC module which listens for commands on a given port. As discussed in Section 4.1.1, much of the JSON handling functionality is realised in the shared module. Instead, the implementation starts the server to receive requests, also checks the validity of calls and their passed parameters. Once a request is deemed valid, it will be passed to the core module, which will then perform the required actions using the companion modules.

As discussed in the design, OpenCache also features a heartbeat mechanism so that the controller can accurately determine the availability and connectivity of attached nodes. This functionality remains entirely within the implementation of this *interface* module. To this end, the module will send out messages to the controller at pre-defined intervals. Piggybacked onto these messages is the latest version of statistics. The *interface* module will therefore periodically request the most recent version of these from the *core* module (who will subsequently retrieve these from the *state* module). The *interface* module will then martial these statistics into a JSON-RPC call, ready to send to the controller.

The *discovery* module is used in cases where the node is not configured to connect to an OpenCache controller. If this is the case, the *discovery* module will send out a specially formed packet into the network at regular intervals. This can only be utilised when an OpenCache controller is connected to a software-defined network. If this is true, the controller should have pre-emptively inserted a set of flow modifications in the underlying forwarding plane. These will ensure that the discovery packets sent by a node are forwarded immediately to the controller, which can then use this to determine a new node has come online. Once the controller has begun to receive some of these discovery packets, the controller will make contact with the node using the aforementioned *interface* module. Once this is successful, the *discovery* module has achieved its primary function and will cease to transmit packets.

4.1.2.1 Services

Arguably the most import element of the OpenCache node is the *service* module. Instances of this module are responsible for retrieving and delivering the content given a client request. Instances of this module will be spawned by the *core* in response to requests made through the *interface*. When this occurs, the request will be accompanied by an expression. As outlined in the design, this expression defines the set of content that a single *service* will serve, as defined by the given pattern. This will typically be unique to a node, as no two services should be storing and serving the same set of content. However, it would be normal for an OpenCache node to have tens or even hundreds of services running at one time, depending on the underlying resource allocation.

When a *service* starts, it will request a port number from the *core* module.

It will then start handling request on this port, and will continue to do so until instructed otherwise. This port number is allocated from a pool of configured ports, and will be relinquished back into the port when the *service* is stopped. When a port is allocated, the node must also inform the controller of this number; it must be correlated against the relevant modifications made in the forwarding plane so that requests can be redirected to the correct port (and thus service).

In the case of this implementation, a HTTP-centric module is provided, although there is obviously scope for modules supporting other delivery technologies to be developed. When a new instance of the *service* module is created, a modified HTTP server is brought online. This server used the *BaseHTTPServer* [39] library already present in Python. However, some important modification are made, including overwriting the logic when handling a HTTP *GET* request. When a *GET* request is received, a HTTP servers would typically either serve a file from disk or generate a page dynamically. As much of the content objects found on the Internet are static in nature, the latter does not concern the implementation of this *service* module.

In replacement of this behaviour, the module will consult the *state* module (via the *core* module), to determine if content is already stored on the cache for this particular request. This is achieved by preforming a lookup given the request URL. If a match is found (a *cache-hit*), a storage location will be returned to the requesting service. In this case, the *service* module will retrieve the object, and deliver it as normal to the client. This includes segmenting the object for delivery over smaller packets (if necessary). However, if the lookup does not yield a result, the content must be retrieved from the origin server (a *cache-miss*). In OpenCache, this is implemented within the *service* module itself, mainly due to the potential performance and latency impact handing this over to a different process may incur. As OpenCache retrieves this object, it will simultaneously deliver this to the client. Importantly, this is a parallel process; the node can deliver the content as itself is fetching it. This prevents the client from having to wait while the whole object is fetched.

Once this process is complete, the *service* will then store this content using one of the attached *storage* modules. Once this has been successfully archived, the *service* module will alert the *core* module that a new object has been stored along with the location of said object. This will then be stored in the *state* module in preparation for future requests. This compartmentalisation between

the fundamental cache functionality and the underlying storage enables different technologies to be used in delivering content, whilst still utilising the same underlying storage mechanisms.

The *BaseHTTPServer* used as the foundation for this module is also modified to enable multi-threading. This enables a *service* module to handle multiple simultaneous user requests without blocking. This is implemented by spawning a new thread on receipt of each client request. The *service* module is also responsible for maintaining a local copy of statistics, providing these to the *core* module when requested (typically when the node needs to send a heartbeat message to the controller). Storing these local to the module also enables the module to alert the *core* component if a particular threshold is reached for any one of the supported metrics. This facilitates the alerting process defined later in the API description.

4.1.2.2 Storage

The final module implemented in the OpenCache node is the *storage* module. This module is responsible for storing the content object, using whatever means necessary. For this implementation of OpenCache, a simple file system based approach is used. This will store an object directly to an attached storage device, such as a hard-disk or solid-state drive. In order to achieve this, the implementation uses Pythons in-built file handling functions to store files in a pre-configured directory in the case of *cache-misses*. Similarly, the same base functions are used to open and read files in the case of *cache-hits*.

As illustrated in Figure 3.2, any of the instantiated *service* modules can have its own attached *storage* module. This enables the storage used to be tailored to the given application. For example, if there is a mixture of storage devices and technologies available on the underlying hardware (such as mechanical disks, solid state drives or even large amounts of random access memory), the service can be configured to utilise the most appropriate technology. This decision may be based upon the likely popularity, the latency requirements or given specific cost constraints. The type of storage used when the service is started is defined in the relevant API call, discussed in Section 4.1.4.

Furthering this capability, the modular design of OpenCache facilitates compatibility with distributed file stores, such as GlusterFS [13] or Ceph [4]. When

used in combination with a shared database, multiple OpenCache nodes can share the same file system (and thus stored objects), regardless of whether or not they are located on the same physical or virtual machine. The distribution and retrieval of objects is largely handled by the third-party software; the *state* component would be a simple compatibility layer.

4.1.3 Controller

The OpenCache controller[3] follows a very similar design to the node, with many of the functions fulfilling a similar purpose. For example, the implementation of the *core* module includes the required logging functionality, and is also responsible for calling various methods in the companion modules. This is an important operation from the controller, as the main method that a user will interact with an OpenCache deployment is through the controller. The *core* module also contains the logic to interpret the contents of the OpenCache *expressions*. As mentioned previously, these will be parsed and checked for validity using a shared library. However, once this is complete, the controller is then responsible for directing these messages and disseminating them to the required OpenCache nodes.

This capability is possible with the implementation of the *state* module. One of the functions of this module is to maintain the state and location of the connected OpenCache nodes, including the services running on them and their capabilities. This allows the controller to send messages to the correct location, as well as facilitating the selection capability engrained in the *expressions*. For example, if a call is destined towards all of the connected OpenCache nodes, then the controller needs to be aware of all of these. Similarly, if the *expression* is directed towards nodes with a particular attribute, such as contained within a cluster of OpenCache nodes or with access to a given specialist resource, the *state* module enables the command to be accurately sent, without the need to first check each of the nodes.

The *state* module also includes up to date metrics derived from each of the connected nodes. As noted in the design section, this is implemented to intentionally accelerate the response to requests for statistics made on the north-bound *interface* module, which is discussed later. As with the *state* module found in the OpenCache node, this module facilitates the storage of this information using a

[3] `https://github.com/opencache-project/opencache-controller`

connected database; the module itself only acts as a compatibility layer to ensure that different databases and the location of such, can be used to best effect given the scenario that OpenCache is deployed in.

Rather than the approach to configuration found in the OpenCache node, the implementation does not require the same level of flexibility in the controller. As a result, the *configuration* module in the controller only supports loading a configuration file from a locally attached disk. This is because the behaviour of the controller is not likely to change during operation. The format of this configuration file is much the same as the basic outline discussed in the previous section, following the *conf* format also described. The content of this configuration file will likely contain much the same information, such as the ports on which it will operate. However, the configuration will differ in that it will also contain information as to the location of the various additional controllers it will communicate with. It will also define the hard time-out used to determine when a node has moved from a connected to a disconnected state.

The controller contains a number of *interface* modules; one facing towards the user, and another used for communication with the connected nodes. The north-bound interface, used by developers and users to interact with the OpenCache deployment, offers a number of possibilities for controlling and modifying connected OpenCache nodes. As with the sole *interface* module found in the nodes, calls made to this interface are parsed and checked for correctness using the shared library described in Section 4.1.1. Once this has been deemed valid, a call is made from the *interface* module to the *core* module. The *core* will then perform the required functions in order to execute this command, including consulting the *state* module to retrieve statistics, creating a message sent to a specific set of connected nodes, modifying the forwarding plane via the software-defined networking controller or creating a new instance of a node using the compute controller.

Once the *core* module has determined the correct set of nodes to which commands will be sent to, they will be delivered using the south-bound *interface* module. This module again uses the shared library for parsing and validating JSON-RPC calls, as well as forming them. Importantly, this module will also receive as well as transmit; heartbeat requests will be sent from each of the connected nodes and received on this interface. This allows the controller to maintain the most up-to-date list of available nodes and thus resources.

The *discovery* module preforms an important function by receiving the messages broadcast from new nodes. If this behaviour is enabled, the will request that a modifications are made the forwarding plane to enable broadcast messages to be received at this interface. Once this is complete, the *discovery* module will listen on a given port, ready to receive messages from these new nodes. Once a message is received, this module will then alert the *core* module that a new node has been discovered. The *core* module will then move further communication over to the south-bound *interface* module, which will attempt to contact the node. There is an important difference in behaviour in this case, as this module is omitted, then the controller will rely on the node making initial contact with the controller. However, if a node is discovered instead, the controller will initiate this communication.

The other two modules that are part of the controller implementation are the *redirection* and *compute* modules. These are interfaces used to communicate with third-party controllers used to modify network and computer resources, respectively. In both of these cases, there is no standardised way to communicate with them. As such, the modular design of OpenCache enables developers to create modules suitable to whichever controllers, and subsequently use the interfaces that they offer. Although there are initiatives to move towards standardised interfaces, especially in the software-defined networking field [49], as of yet none has come to the fore nor seen implementation. Concepts such as network *intents* show promise, in that OpenCache would simply notify the controller the functionality that is desired from the network (the redirection and discovery mechanisms), without specifically defining the actions that need to take place. In this implementation of OpenCache, this is exactly what we have to do in both cases.

4.1.3.1 Redirection

As mentioned in the design section, the modularity of OpenCache is designed to enable different redirection techniques to be used. To demonstrate this, the implementation includes two modules for use with different software-defined networking controllers. Due to the lack of availability of alternative technologies, these controllers are designed to work with the OpenFlow technology discussed in Section 2.2.2.1. For v1.0 compatibility, the Floodlight [11] controller was used whereas for v1.3 compatibility, the Ryu [38] controller was used instead. This

change was necessary due to lack of support for the updated protocol in Floodlight.

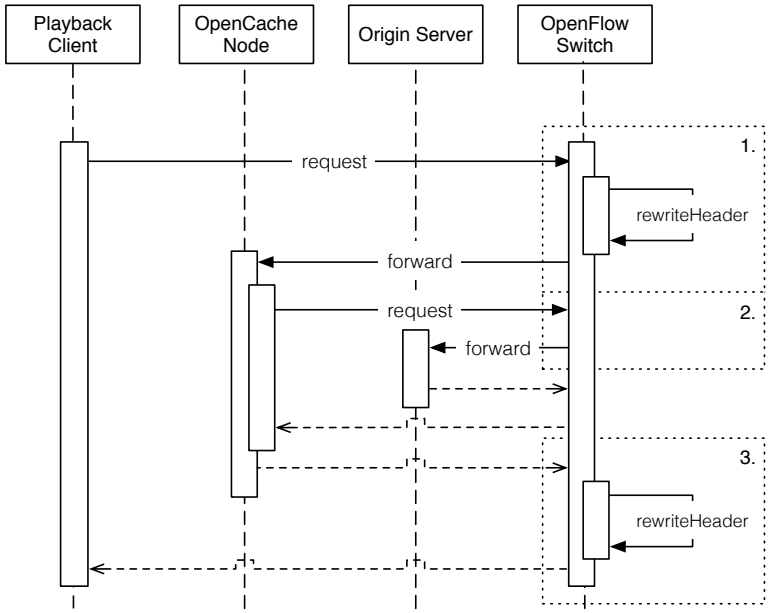

Figure 4.1: Request Redirection Process

When an application instructs the OpenCache controller to start a new service on a node, the controller will send a command to the node to do just that. Matching that, a modification is made to the underlying network. More specifically, a number of OpenFlow rules are added to the network. In the case of Floodlight, this is achieved using the Static Flow Pusher API [40]. This API allows flows to be installed directly into the network, and is essentially a thin compatibility layer offering direct access to many of the functions offered by OpenFlow itself. More specifically, OpenCache will install three OpenFlow rules for every service that is started. These rules are demonstrated in Figure 4.1, which shows how a client request is handled when a redirect is in affect.

Rule 1 redirects requests towards the cache by changing the destination IP address and port to that of the node and service, respectively. This is achieved by rewriting the packet header using the OpenFlow *action* command. It is then forwarded throughout the rest of the network as normal.

Rule 3 does the opposite when a response traverses the switch: it changes the port and IP address back to that of the origin server. Again, this uses OpenFlow's header rewriting functionality to modify the packet accordingly.

Rule 2 is used to allow the service to request content from the origin server. Depending on the location of the user, node and forwarding element on which the rule is placed, this may or may not be an issue. However, in the case where they are located nearby, and the node's request has to traverse the same forwarding device, rule 2 is necessary to ensure that the node's outgoing request does not match on rule 1 (which will incorrectly forward the request back to the node). To combat this, rule 2 provides an explicit match with a higher priority than the other rules. When a packet matches this, it will be forwarded normally, ensuring that the node can retrieve content as intended.

Together, these rules provide the base redirection facility required by OpenCache. This is achieved transparently; the client has no knowledge that a cached item has been delivered to them, as the process has been completed at a network layer.

A similar process is also applied when using the Ryu controller to ensure v1.3 OpenFlow capability. In this case, no external API was available to use, so a simple module was implemented in Ryu that provided the necessary functionality, achieving the same as provided by Floodlight.

4.1.3.2 Virtualised Compute

The OpenCache controller will also communicate with a compute controller via the *compute* module. This will be used to create and destroy OpenCache nodes in a virtualised environment. Again, this will use a third-party controller to do so. In the case of this implementation, experimental functionality was built around OpenStack; an open-source cloud platform. This contains a number of APIs for interacting with various elements that make up an OpenStack deployment. OpenCache uses these to complete a number of functions, the first of which is to install a templates image in OpenStack. This is a pre-built operating system that is based upon the CoreOS [6]. This is an operating systems created specifically for hosting *containers*.

Containers provide a way to logically separate items of function into smaller manageable instances. Multiple containers will then reside on the same CoreOS instance, which also feature some orchestration features. The features are used to bring up a number of containers when an instance is created. To do this, we used the popular Docker [8] format. This allows containers to be defined by a simple

text configuration file (a *dockerfile*), which states the process of how a container should be built, including the software that needs to be installed, the order of such, where configuration should be fetched from also any companion container dependencies.

In the case of an OpenCache node, an instance will contain a number of Docker containers, one of which contains the running OpenCache node. Other containers will contain an instance of MongoDB, the NoSQL database to support the *state* module. It may optionally contain docker containers to support other services, such as alternative storage methods supported by new *storage* implementations. Importantly, this method also gives fine-grained control over the services running on the operating system; for example, multiple OpenCache nodes could be run in the same instance.

Other features of the OpenStack API includes the ability to define the internal network of instances, so they can be connected together in a particular arrangement. In the initial implementation, these instance are simply connected to an external network, providing them connectivity with the outside world. OpenStack is also developing support for software switches for this internal networking, using tools such as OVS [22] which also support OpenFlow. This provides some potential for further granularity on the forwarding plane in the scope of the redirection used in OpenCache, although this is not explored in this thesis due to the immaturity of the implementation at present.

4.1.4 API

The OpenCache application program interface (API) is one of the most important elements of the OpenCache design. It not only provides the internal mechanisms for control, monitoring and reporting, but also enables the flexible configuration and instruction of a cache deployment by an application or user. As discussed in the design section, both the OpenCache controller and node have *interface* modules, which are responsible for both receiving and sending JSON-RPC messages.

In the case of the controller, multiple modules are used to provide communication in two distinct directions. The basic format of these messages was outlined at the beginning of this chapter; this section highlights the specific *method* and *params* that are compatible and valid in the context of the OpenCache API. The OpenCache API is split into an external and an internal API. Although these

are described individually, they are nonetheless similar. It is this commonality that enables the hierarchical control of OpenCache that is proposed in the design chapter.

Following the JSON-RPC specification, a call made to the OpenCache API will contain a *method* field and a set of *params*. The *method* field is a string containing the name of the function to be called, whilst *params* is the values passed to that call. The content of the *params* follows the JSON object format, defined through encapsulation braces. Within this object, a number of *pairs* will be passed. These key/value pairs are indicated through the use of a colon separator. For the majority of calls, these will be selection criteria; a method for the caller to define explicitly the nodes and services on which they want an operation to be run. For example, passing a single value in the *expr* field associates your call with the given expression (analogous to a single instantiated service), potentially running on a variety of OpenCache nodes.

Declaring a set of *node-id* values can also be used to indicate that an operation should be preformed on a number of nodes simultaneously. In this case, multiple IDs can be given through the use of a JSON array; a comma separated list defined within a set of brackets. This nomenclature is used in both the description of the external API in Table 4.1.4.1 and the internal API in Table 4.2.

In addition to the standard JSON notation, the use of parentheses in these tables denotes an optional value. These are almost always exclusively used in cases of selection, and the omission of such indicates that the field should be wild-carded. In some cases, at least one of these fields should be provided (if the *params* contains multiple option fields); this is defined in the detailed description of the methods below. When wild-carding functionality is used, any value is acceptable in this field, providing flexibility to the programmer to address a vast array of potential nodes without specifically defining them.

Finally, chevrons are also used throughout the two tables to indicate the location of values that will be substituted in during actual calls; these are simply place-holders for the sake of readability, although they hint towards the type or format of the variable. Tables 4.1 and 4.2 also outline the expected return values once a call is made. This value will be part of the JSON-RPC response object, and will include this value in the *response* field. If a function fails, the response will contain an error message, also in the JSON-RPC format, indicating the relevant error code along with a description of the issue.

Method	Params	Result	Direction
start	{ "expr" : [<expr>], ("node-id" : [<node>]), ("tag" : [<tag>]), ("storage" : <storage>) }	<boolean>	A → C
stop	{ ("expr" : [<expr>]), ("node-id" : [<node>]), ("tag" : [<tag>]) }	<boolean>	A → C
pause	{ ("expr" : [<expr>]), ("node-id" : [<node>]), ("tag" : [<tag>]) }	<boolean>	A → C
move	{ "expr" : <expr>, "from" : <node>, "to" : <node>, } ("storage" : <storage>) }	<boolean>	A → C
fetch	{("expr" : [<expr>]), ("node-id" : [<node>]), "target" : [<url>], ("tag" : [<tag>]) }	<boolean>	A → C
seed	{ "expr" : [<expr>], ("node-id" : [<node>]), ("tag" : [<tag>]) }	<boolean>	A → C
refresh	{ ("expr" : [<expr>]), ("node-id" : [<node>]), ("tag" : [<tag>]) }	<boolean>	A → C
stat	{ ("expr" : [<expr>]), ("node-id" : [<node>]), ("tag" : [<tag>]) }	{ "cache-hit" : <cache-hit>, "cache-miss" : <cache-miss>... }	A → C
register	{ ("expr" : [<expr>]), ("node-id" : [<node>]), ("tag" : [<tag>]), "metric" : <metric>, "threshold" : <threshold>, "ip" : <ip>, "port" : <port> }	<boolean>	A → C
alert	{ "expr" : <expr>, "node-id" : <node>, "metric" : <metric>, "value" : <threshold> }	<boolean>	C → A
describe	{ ("node-id" : <node>)}	{ "storage" : [{ "type": <type>, "capacity": <capacity> }] ... }	A → C
tag	{ "node-id" : [<node>], "tag" : [<tag>] }	<boolean>	A → C
create	{ "expr" : [<expr>] }	{ "node-id" : <node> }	A → C
destroy	{ "node-id" : <node> }	<boolean>	A → C

Table 4.1: External API Specification

4.1.4.1 External

The external API is implemented in the north-bound interface module within the OpenCache controller. It is through this interface that users and applications will interact with OpenCache. In the following section, we describe in more detail the methods and parameters identified in Table 4.1.

The *start* method is used to instantiate a *service* module on a particular node or set of nodes. This service will handle requests matching the expression (*expr*) given during its creation. Under normal circumstances, a new service will start with no content stored in the cache; each initial request would likely produce a *cache-miss*. When using a software-defined networking controller to provide the required redirection, there should be no need to check if a request is valid for a given expression. This is because the matching is done at a lower layer (the network) and will only be forwarded if a request successfully matches. From the perspective of the OpenCache controller, once a *start* command is issued, the internal API will be used to start the service on the required set nodes. The *expr* parameter is not optional in this method, as it is required to start a service. However, this is not the case in the accompanying service control methods, where it can be either included and omitted.

Importantly for OpenCache, during the execution of the *start* operation, the controller will also communicate with the network control element to ensure that the forwarding layer is modified appropriately. This will ensure synchrony between the redirection of requests and the availability of running services ready to

handle them. This function also contains an optional *storage* parameter. This allows the application to determine the underlying storage module used for this service. The availability and choice of storage will be determined with the *describe* command, outlined later. If this parameter is omitted, then the default storage module will be used. This method will return a boolean value (*true* or *false*) depending on the success of the operation. If the method has failed, the JSON-RPC error object will be included with details of the failure. As part of the external API, this method will always be called in the direction of application to controller.

The *stop* method achieves the opposite effect to the *start* method: it will stop the services (selected through their respective expressions) on the nodes specified. In this case, the service is stopped and will no longer respond to user requests. Clearly, the redirects need to be removed to match this change in state, as otherwise requests may go unanswered. The content that was once associated with the stopped service is removed and permanently deleted. If the service was to be started again at a later date, it would do so afresh without any content stored. Similarly, the compute, thread and network resources that were once associated with this services will be relinquished for use by other services. As with *start*, the controller will use the internal API to enact the changes on the specified nodes and services. The *stop* method also has a similar response format to *start*, and follows the same call direction.

The *pause* method is very similar to the *stop* method. However, the main difference between the two methods is that in the case of *pause*, the content objects used to serve user requests still remain on the underlying storage, ready to be served once more. When a service is paused, the compute, thread and network resources will be freed, but the storage usage remains. In this case, the OpenCache controller will also remove the request forwarding rules from the network, as the service will not respond to requests whilst paused. A service can be brought out of a paused state by issuing a *start* command with matching parameters. When the service restarts, it will have access to all of the previous fetched content, reducing the likelihood of *cache-miss* events that may occur when a new service is started. The *pause* method shares the same response and direction as both *start* and *stop* methods.

The *move* method is a composite function with an interesting mix of functionality; it combines specific usage of *start* and *stop* commands as well as a

unique modification of the redirection rules used in the forwarding plane. It is used primarily to migrate services between nodes. It achieves this by starting a new service on the destination node (the *to* field), modifying the redirection to now send requests towards this new service, and then removing the old service which is no longer required (the *from* field). In this case, the *move* command is intentionally explicit; only one service is moved at a time. This is because the order and speed of the required operations is important to ensure that a consistent service is provided when handling user requests. For instance, the redirection should not be modified until the service is running, ready to handle requests. This method enables the user or application to interact with the service offered without requiring knowledge of the specifics of doing so.

The *move* command also optionally defines a type of storage to be used when the service is migrated; if this is not given, the default storage method on the destination node will be used. The response from a call utilising this method will only be returned once the whole process has been completed. This ensures that additional actions are not taken without knowledge that the method has been successfully completed (or otherwise). As with the previous methods, this is indicated with a boolean value. Naturally, this request is made from the application towards the controller.

The *fetch* method is another interesting piece of behaviour possible through the API. It allows a service to pro-actively fetch content and store it in the cache, ready to be served in response to a client request. Importantly, this is done without it ever being requested by a client. This prevents a *cache-miss* and ensures that even the initial request results in a *cache-hit*.

This functionality can be used in situations where advanced knowledge is used to identify objects that are requested frequently. By placing the object in the cache beforehand, a node can serve the content immediately without having to request the content from the origin server. This is particularly important for large objects that will take a considerable amount of time to fetch, thus reducing the potential latency observed by a client. This method will also produce a response indicating the success of the operation. However, it will not wait until the content has been downloaded before determining the result, as this could a significant delay in the case of large objects. As before, this is a call made from the application to the controller.

The *seed* method is used primarily in cases where identical content is located at

multiple locations. By defining a list of equivalent expressions in the method call parameters, requests for content matching any of the expressions will result in the same content being served. The seed function can significantly reduce instances of duplication in the cache object store, thus maximising the storage efficiency. This method too is called from an application to the OpenCache controller, with a suitable response provided once the information has been disseminated to the nodes and services defined in the method parameters.

Statistical reporting is a core part of OpenCache. Two calls are concerned with this: *stat* and *refresh*. In most cases, *stat* will be used to retrieve statistics of the nodes and services defined in the selection criteria. Due to the fine-grained nature of this, statistics can be retrieved in a number of different scopes, ranging from the entire deployment to an individual service. Such a method will be answered using the most recent copy of statistics stored in the *state* module of the controller. This can include such information as the amount of cache-hits and misses, the size and count of the object in the cache, as well as the usage of disk resources and uptime of the node and/or service. Calls made using the *stat* function will be sent from the application towards the controller, with the response invariably containing the full set of statistics for the selected set of nodes and services.

To accompany this functionality, a helper method is used in the form of *refresh*. This simply requests that the selected nodes respond with the latest set of statistics, which will instantly update the local copy stored in the controller. This bypasses the piggybacking of statistics in the heartbeat messages, and forces the node to send a response. This is useful if the resolution and period of the heartbeat message is set to a large time-interval, and for whatever reason, the application requires the most recent statistics possible. Evidently, this call will originate at the application and be sent towards the controller. The response will be a boolean value, indicating that the message has been sent to the chosen OpenCache nodes.

In some cases, a poll mechanism is not sufficient nor efficient for the application. As a result, the OpenCache API contains an alert subsystem. An application can subscribe to events using the *register* command, indicating the metric and threshold for which they want to be alerted to. As with the majority of the other OpenCache commands, they can select the nodes and services from which these alerts should be received from. When one of these thresholds is exceeded,

the service will alert the controller, which in turn will send an *alert* message to the application. This will only occur if the application has registered for an alert previously, and will be sent to the IP address and port number as given in the *register* command.

Applications may wish to discover the underlying composition and capabilities of a given OpenCache deployment. This can be achieved using the *describe* command. This will return the configured resources of a given OpenCache node, including the availability of storage, the type of this storage, connectivity, hypervisor etc. In the present implementation, this information is manually described in the configuration of a node, and will not change during the lifetime of the node. Evidently, it would be more appropriate if the node discovered its own capabilities, and monitored them for changes. This command will also relay information about the possible *storage* modules that a node has access to, and that can be given in the *start* and *move* commands.

An application may also want to refer to a particular subset of nodes and/or services without referring to them each individually. To do this, the *tag* command can be used to annotate individual nodes and services with arbitrary labels. These labels can then be used in the aforementioned methods as part of the selection criteria. These tags are stored and maintained on the controller; the individual nodes and services have no knowledge of what values they have been tagged with. Potential applications include clustering nodes together based upon location or specific hardware resource. The information used to annotate a node may be derived from external information not accessible to OpenCache itself.

The *create* function can only be used when an OpenCache controller has connectivity to a compute controller. This method will instantiate a single virtual machine, running as an OpenCache node, and start each of the services optionally defined in the method parameters. Similarly, the `destroy` method will remove this virtual machine instance, and all of the services running upon it. Evidently, *destroy* can only be called on nodes that are instantiated through the OpenCache controller and API. Note that the *create* module does not take a *storage* value as a parameter; this is because it is not currently possible in the API for the application to discover the potential capabilities of a new node when using a compute controller. As a result, services started this way will use the default storage module loaded when OpenCache starts (in most cases, using the underlying filesystem).

Method	Params	Result	Direction
start	{ "expr" : [<expr>] }	<port>	C → N
stop	{ ("expr" : [<expr>]) }	<boolean>	C → N
pause	{ ("expr" : [<expr>]) }	<boolean>	C → N
fetch	{ "url" : [<url>] }	<boolean>	C → N
seed	{ "expr" : [<expr>] }	<boolean>	C → N
refresh	{ ("expr" : [<expr>]) }	{ "cache-hit" : <cache-hit>, "cache-miss" : <cache-miss>... }	C → N
register	{ ("expr" : [<expr>]), "metric" : <metric>, "threshold" : <threshold> }	<boolean>	C → N
alert	{ "expr" : [<expr>], "node-id" : <node>, "metric" : <metric>, "value" : <threshold> }	<boolean>	N → C
describe	{ "node-id" : [<node>] }	{ "storage" : [{ "type": <type>, "capacity": <capacity> }] ... }	A → C
broadcast	{ "host" : <host>, "port" : <port> }	<boolean>	N → C
hello	{ "host" : <host>, "port" : <port>, "node-id" : <node> }	<boolean>	C → N
hello	{ "host" : <host>, "port" : <port> }	"node-id" : <node>	N → C
goodbye	{ "node-id" : <node-id> }	<boolean>	N → C
keep-alive	{ "expr" : [<expr>], "node-id" : <node>, "stat" : { "cache-miss": <cache-miss>... } }	<boolean>	N → C

Table 4.2: Internal API Specification

4.1.4.2 Internal

To compliment the external API offered to applications, an internal API is used for controller to node communications. Many of the methods contained within the internal API, outlined in Table 4.2, are shared between the external interface. This is because in most cases, when a call is made by an application towards an OpenCache controller, the controller will simply disseminate this method to required nodes using the internal API; the nodes does not need to know the other nodes to which the command has been sent. This is due to centralised design of OpenCache, whereby nodes communicate solely with the controller doing their entire lifecycle.

The controller is therefore responsible for maintaining the location and availability of the nodes, as well as maintaining any arbitrary labels applied to them. This enables the controller to send the calls to the correct nodes given the selection criteria present in the method parameters. As a result of this, the calls made from the controller to the node are more specific; the node will only receive commands destined for itself. The methods that are completely omitted from the internal API are those that either rely on communication with an external controller (and are thus the OpenCache controllers responsibility) or depend upon state stored in the controller (such as the tagging functionality). However, the internal API calls do contain selection criteria for sending specific messages to individual services running on the node. In this case, they are addressed using

the *expr* that they were started with. If this field is optional and omitted in a call, all running services will have the method applied to them.

In the previous section, we described the effect that each of these methods will have when executed. For example, the *start* method will start a set of *service* instances on the node, matching the *expressions* given. Similarly, the controller can also pause and completely stop these services using the *pause* and *stop* commands, respectively. Importantly, the returns from the commands will indicate the success of starting such a service. If an error occurs, the controller will understand that an operation has failed on that node. This information can then be relayed back to the application, which will receive an error response indicating which nodes are failed along with some debug information on the matter. In the case of *start*, the node will relay back the port number that the service has been started on. This allows the controller to add the relevant rules to forwarding layer to offer the required redirection. As these port numbers are taken from (and subsequently replaced back into) a pool, the port chosen can differ between nodes, depending on what has been run prior to the application of this process.

The same process applies to the *fetch* and *seed* functions, as these are entirely implemented in the node. In fact, the pre-fetching offered by the *fetch* method is realised in the *service* itself, as it will go ahead and fetch the content from the given URL without the need for it first to be requested by a client. The *seed* function will contain a set of *expressions* that are mutually equivalent. As a result, the value of the *expr* parameter will also be in the format of a JSON list, of which each element defines a set of content that should be served as one. This will overwrite the URL checking element in the *service* implementation by allowing multiple values to be correct, and thus, objects to be served from each of the locations defined.

The *refresh* method in the internal API forces the node to send a fresh set of statistics to the controller. In this case, this information is sent back in the response field. This temporarily overrides the usual process of reporting statistics, which is to piggyback them onto the *keep-alive* message. The *keep-alive* messages primary use is to ensure that communication can still be made between the controller and the node. Once the controller receives such a message, it will refresh the timeout back to pre-defined maximum. However, if the node fails to send this message for whatever reason, the controller will eventually disconnect the node, and prevent requests from being sent to it.

Before a node moves to sending periodic *keep-alive* messages, it must make an initial connection with the controller. As mentioned previously, this can be initiated in two separate ways. If the broadcast functionality is being used in an OpenCache node, it will send out regular *broadcast* messages on a fixed port. This port number is always the same in an OpenCache deployment, and is not configurable in the OpenCache controller. The *broadcast* message contains sufficient details for the controller to move onto the alternative communication channel, handled by the south-bound *interface* module, rather than the *broadcast* module. Once a *broadcast* message has been received by the controller, a controller-initiated *hello* message will be sent to the node. This will contain details of how the node can communicate with the controller, as well as an allocated identification number. Communication will then continue as normal once this process is complete, including the regular transmission of *keep-alive* messages.

Alternatively, a node may be configured with the controller's details before it starts. If this is the case, the node will initiate a connection to the controller, rather than the opposite process in the above-mentioned case. The node will therefore send a *hello* message, detailing the means by which the node should be contacted. Once the controller has received this message, it will respond with an identification number, used throughout the lifecycle of that node. As before, the node will then migrate to sending regular *keep-alive* messages to ensure the controller is both aware of its presence, and to update the local statistics stored on such. To compliment this process, the *goodbye* method can be used to gracefully leave the OpenCache deployment. Rather than a node timing out due to a lack of *keep-alive* messages, the *goodbye* message can be used to gracefully disconnect from the network. Example uses include when a device or hypervisor is going offline for scheduled maintenance.

The internal API also contains the methods to support the alert subsystem that is an important part of the OpenCache implementation. As with many of the other methods, calls made from the application to the controller are then translated into messages sent to the nodes themselves. The nodes then handle the monitoring, as defined in the *register* method. Once a service or node exceeds one of these thresholds, the node will send a warning to the controller using the *alert* method, containing details of the threshold and its current value. Similarly the *describe* method, when called by the controller, will return detailed information about the OpenCache node back to the controller for use by the application.

As mentioned previously, in the current implementation, this is achieved by simply returning the details stored in the configuration file, rather than the node discovering them itself.

4.1.5 Development and Deployment Aids

OpenCache is implemented in a modular way. This intentionally allows developers to build alternative modules designed to meet their own differing requirements. To facilitate this, we provide a set of design guidelines, along with extensive documentation, which demonstrates how a module should be implemented. These outline a minimum requirement for functions that should be implemented, and the values that they should return once complete. This rigidity is necessary so that the *core* module in both the OpenCache controller and node can interoperate with these alternative implementations. The *core* module enforces these guidelines by checking whether or not the object has the required methods before it is allowed to continue. Failure to implement these will result in the node or controller refusing to run.

To encourage development, we have made available an OpenCache Vagrant-based [42] virtual-machine image[4]. This is designed to provide developers with a convenient development environment that can be used to build and test new OpenCache modules as required. Within this environment, a full install of OpenCache is included. To compliment this, Mininet [17] is also bundled in the image: Mininet allows virtual networks to be created and modified at will. This allows developers to emulate networks and attach OpenCache nodes, controllers, hosts and servers to them. Together, these can be used to create realistic networks and generate genuine traffic, ideal for ensuring modules are operating as expected. The same tool-chain was also used extensively during the development and implementation as OpenCache. Providing this environment to other developers ensures a level of parity amongst potential developers.

To aid the deployment of OpenCache in different scenarios, we take advantage of Python's native packaging tools. If the user is building from source, *pip* [34] can be used to install the required libraries automatically. Each component is also available on *pypi* [36], a global repository of open packages. Installing OpenCache is therefore as simple as running *pip* with the chosen package (*opencache-*

[4] `https://github.com/opencache-project/opencache-vagrant`

controller or *opencache-node*). This install also includes an executable shell helper script, which starts the given component from command line. From installation to execution, the processes are made intentionally simple.

4.2 OpenCache Console

The external API presented in Section 4.1.4.1 facilitate numerous different interaction techniques with an existing OpenCache deployment. Evidently, these must all be based upon the usage of JSON-RPC, as required by the specification. However, given this restriction, it is still possible to use the interface in a flexible manner, including from a command line, an application or, as presented in this section, a graphical user-interface[5]. Designed specifically for use with OpenCache, the console is a web-based interface from which users can control and modify the behaviour of OpenCache nodes and the services running upon them. However, the most important feature of the console is the ability to visualise statistics over time, enabling a user to better understand and rationalise the results that they are seeing.

As with the rest of the OpenCache implementation, the console is implemented using mainly Python. In particular, the Flask [145] framework was used. Flask is a lightweight framework designed to enable web applications to be built quickly whilst offering production-quality capabilities. It also includes templating functionality, used extensively in the OpenCache console. This functionality, enabled by the Jinja2 [146] library, allows the console to replace values in a template when it is rendered. It also features control logic, allowing different information to be shown depending on the value of a variable.

The console features two main views, the first of which is the *management* pane, shown in Figure 4.2. This allows the user to directly call any of the OpenCache commands present in the external API, and also define the parameters that accompany the specific call. There is no conversion or manipulation of these calls on the behalf of the console; it simply makes the request using a JSON-RPC library to a connected OpenCache controller. The location and port of this controller is defined in a simple configuration file, and loaded when the console is initialised. Once a call is made, the console will report back the result to the

[5] `https://github.com/opencache-project/opencache-console`

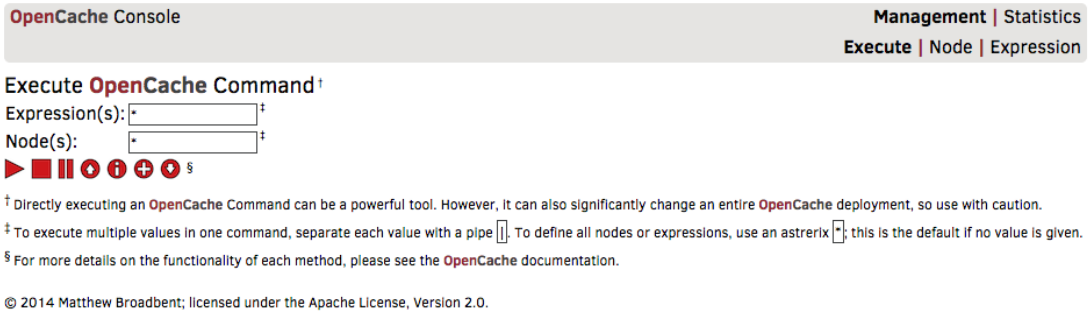

Figure 4.2: OpenCache Console Management Pane

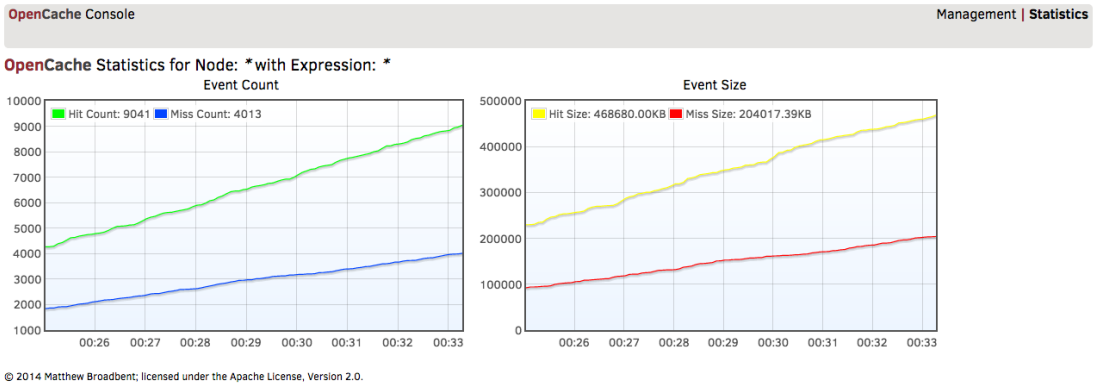

Figure 4.3: OpenCache Console Statistics Pane

user.

The second view is the *statistics* pane, shown in Figure 4.3. As with the *management* pane, this allows the user to define the selection criteria for the statistics that they are interested in. Once this has been determined, the console will fetch an initial set from the controller. During this phase, the user can also set an *interval* value. This is the time between each poll request to the controller. This is necessary because the *statistics* view will render near-real-time data retrieved from the controller. This will then be visualised on a line-graph. As these are retrieved regularly, the graph uses asynchronous calls to constantly update the values shown. This allows a user to monitor the OpenCache deployment, and determine its behaviour over time. It provides a simple overview to show that the deployment is functioning correctly, and the load it is under.

4.3 OpenCache Applications

More advanced applications can also be built semantically on-top of OpenCache. These contain logic and processes defined by the programmer, and enable unique functionality to be created. This includes tailoring the behaviour to a certain scenario or set of circumstances, as well as using information not available to OpenCache to influence decisions. Determining the operation of OpenCache could also be realised through implementing a module with the controller itself; the design is created specifically to do this type of modification. Yet implementing such, especially when the module is specific to only a single use-case, reduces the general applicability of OpenCache as a system. Therefore the external API is deemed as the most appropriate way to achieve the same goals, especially in its current form whereby it exposes all of the internal decisions made within OpenCache.

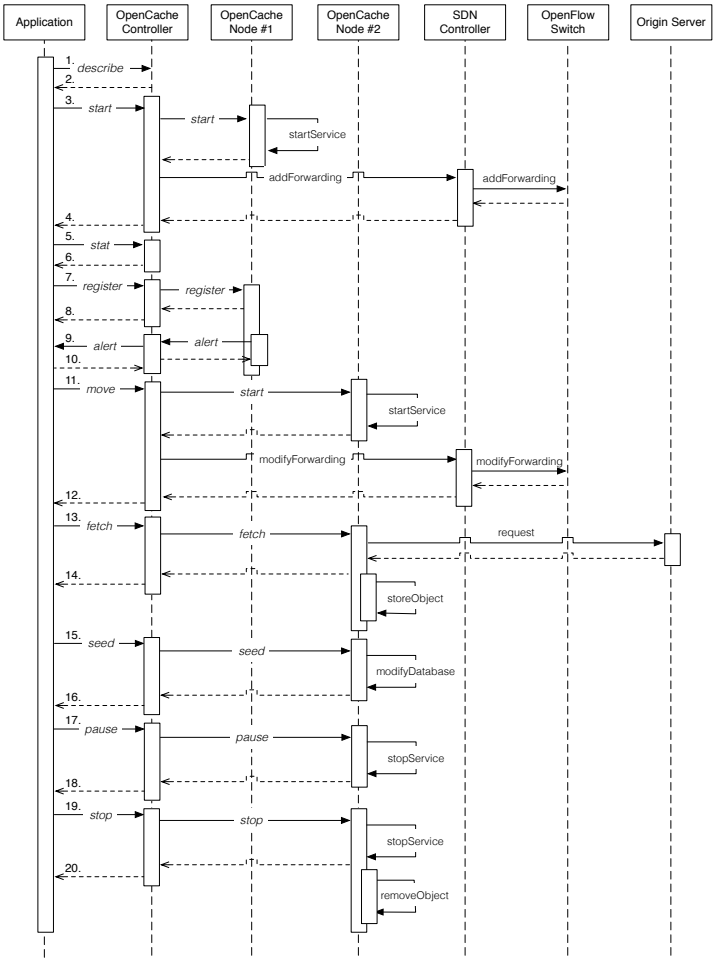

Figure 4.4: Example Application Message Flow

No.	JSON-RPC Call
1.	{"jsonrpc": "2.0", "method": "describe", "params": {"node-id": 1}, "id": 1}
2.	{"jsonrpc": "2.0", "result": [{"storage": "ssd, "capacity": "100GB"}] "id": 1}
3.	{"jsonrpc": "2.0", "method": "start", "params": {"node-id": 1, "expr": "64.15.119.99"}, "id": 2}
4.	{"jsonrpc": "2.0", "result": True, "id": 2}
5.	{"jsonrpc": "2.0", "method": "stat", "params": {"node-id": 1, "expr": "64.15.119.99"}, "id": 3}
6.	{"jsonrpc": "2.0", "result": {"cache-hit" : 9263, "cache-miss": 806}, "id": 3}
7.	{"jsonrpc": "2.0", "method": "register", "params": {"node-id": 1, "expr": "64.15.119.99", "metric": "cache-hit", "threshold": 10000, "ip": "10.32.110.134", "port": 50001}, "id": 4}
8.	{"jsonrpc": "2.0", "result":True, "id": 4}
9.	{"jsonrpc": "2.0", "method": "alert", "params": {"node-id": 1, "expr": "64.15.119.99", "metric": "cache-hit", "value": 11083}, "id": 5}
10.	{"jsonrpc": "2.0", "result": True, "id": 5}
11.	{"jsonrpc": "2.0", "method": "move", "params": {"expr": "64.15.119.99", "from": 1, "to": 2}, "id": 6}
12.	{"jsonrpc": "2.0", "result": True, "id": 6}
13.	{"jsonrpc": "2.0", "method": "fetch", "params": {"node-id": 2, "expr": "64.15.119.99", "target": "64.15.119.99/content/bigbuckbunny.mp4"}, "id": 7}
14.	{"jsonrpc": "2.0", "result": True, "id": 7}
15.	{"jsonrpc": "2.0", "method": "seed", "params": {"node-id": 2, "expr": ["64.15.119.99", "64.15.119.114", "64.15.119.98"]}, "id": 8}
16.	{"jsonrpc": "2.0", "result": True, "id": 8}
17.	{"jsonrpc": "2.0", "method": "pause", "params": {"node-id": 2, "expr": "64.15.119.99"}, "id": 9}
18.	{"jsonrpc": "2.0", "result": True, "id": 9}
19.	{"jsonrpc": "2.0", "method": "stop", "params": {"node-id": 2, "expr": "64.15.119.99"}, "id": 10}
20.	{"jsonrpc": "2.0", "result": True, "id": 10}

Table 4.3: Example Application JSON-RPC Calls

This section will describe a typical workflow for an application that wishes to use the OpenCache interface, presented in the form of a hypothetical use-case. This example is illustrated in Figure 4.4. This describes the message flow between the different entities within this scenario. In this diagram, methods shown in italics indicate calls made using the OpenCache API. The numbers allocated to these calls also correspond to rows in Table 4.3. Each of these entries represents a single call, and demonstrates the format and parameters passed in each case.

Most applications will start by discovering the capabilities and size of an OpenCache deployment. This is achieved with a call to the *describe* method, leaving the selection criteria empty. This will return a list of all of the nodes and their inherent capabilities. An application may then also label these using its own semantics. The application will then use whatever internal logic that it contains to start services on any number of these nodes. This will be achieved using the *start* command alongside the appropriate selection criteria. At this point, OpenCache will start serving the content defined in the *expression*.

After a while, the application may wish to check the state of this service, and will send a *stat* request to retrieve the operational metrics for the newly started service. If the application notices that one of the variables is moving towards the extremities of what is permissible, it can use the *register* command to mon-

itor this, without having to continuously poll the controller. In this example, an alert is triggered sometime after because one of these metrics has been exceeded, such as the amount of cache-hits or the number of objects stored. This may be approaching the limit for the capabilities of this node. As a result, the application decides to migrate the service to a different, previously unused node using the *move* command. The changes to the necessary redirections are made automatically, and the new service begins serving content, taking the load off of the original service.

Over time, the application begins to analyse the patterns of the user requests, and realises that there are potential efficiency gains by pre-caching this content. As the node has a high-capacity link and is well within its capacity limits, the application uses the *fetch* command to pre-emptively instruct the cache to fetch content that it predicts will be requested at a later date. It also receives information from its upstream content delivery network that the data is now located in a number of different locations due to the popularity of user requests. To avoid object duplication, the application uses the *seed* command to define the location of these copies, furthering the performance of the cache.

Eventually, the number of requests for this content abates, and the application decides that it needs to scale back its resource allocation to allow them to be used by other functions running on the same hardware. On some nodes, it uses the *pause* function to relinquish the compute and network resources, whereas on some nodes it uses the *stop* function to remove all of the resources once consumed by the running services. In the cases where the services were paused, they can be immediately restarted if the amount of requests returns to the previous level.

To better demonstrate the effectiveness and flexibility offered by the OpenCache API, an number of example applications were implemented[6]. These took the form of a service load-balancer and failover monitor, which leverage both the application and network layer control offered by OpenCache. These are discussed further in the following evaluation chapter, specifically in Sections 5.3.1 and 5.3.2, including detailed information on message flows and application logic.

[6] `https://github.com/opencache-project/opencache-applications`

4.4 Scootplayer

In order to evaluate certain aspects of OpenCache, realistic user traffic had to be generated. As the initial implementation of OpenCache centres around the delivery of HTTP content, and in particular the use of adaptive streaming technologies, a suitable client was required. Through an initial exploration of clients at the time, none was deemed suitable for our purposes. Requirements included the availability of fine-grained output, as well as control over the behaviour and algorithms used during the request process.

It was found that the behaviour of existing players often changed between revisions, presumably due to the relatively new nature of the chosen streaming standard, MPEG-DASH. Some of the candidates also featured proprietary code, which could not be modified to our needs. Given the situation, the decision was taken to develop a new MPEG-DASH compliant player, designed specifically for our needs. This player could also be used at various stages of our evaluation, without the risk of the underlying behaviour changing due to a code-revision. This enables a fair comparison, and makes the evaluation results directly comparable between our own experiments and others. This is furthered by the release of Scootplayer as a free and open-source tool, which can be used by other experimenters in their work.

Scootplayer was implemented not as a player, but as a request engine. This difference is due to the nature of our evaluation, which is concerned with functionality and performance of a cache, and subsequently its impact on the network. As a result, Scootplayer was designed without the need to render video or display this to a user. However, metrics derived from the simulated playback, such as the quality requested and latency received, are still important to the overall result. The player therefore replicates the behaviour of the alternative clients, including the logic behind quality selection, startup procedures and the actual downloading of content. Importantly, all of this information is logged and documented in Scootplayer, allowing the user to directly compare different evaluation runs and observe the effect of changes in both the network and the delivery of content from whichever source is used.

As with OpenCache, Scootplayer is built using the Python programming language. This allows the client to run on a number of different architectures and operating systems, without the need to recompile the code: important in an eval-

uation scenario. It uses a modular style, which promotes separation between the different entities that make up a player. This includes the ability to change how the download and playback queues work, as well as the procedure at start-up. The states of each of these queues, as well as general information, are all output to log files during the operation of Scootplayer. These are presented in a comma-separated format, making them easy to parse and graph by a vast array of applications. Basic system information is also bundled alongside this output, to ensure that comparisons are as fair as possible.

Scootplayer also contains a remote control element, allowing numerous Scootplayer instances to be started, paused and stopped at will. This ensures basic consistency in experimentation where necessary. The player is primarily a command line tool, as it does not render nor present video. However, to compliment this, a graphical user interface was developed for Scootplayer. Much like the OpenCache console, this is built using the Python-based Flask framework. However, this interface does not provide control of a player, it merely presents the current state of various playback elements in near real-time. This allows a user to check that playback is occurring, as well as providing a visual tool for demonstration purposes.

Finally, Scootplayer is also released open source, and available for others to freely use and modify. In addition to providing a stable implementation that is compliant with the MPEG-DASH specification, it also enables the scientific scrutiny of a tool used in evaluation. Furthermore, it aids experimental repeatability and transparency.

4.5 Summary

In this section, we presented the implementation of OpenCache. In particular, we explored how the core components of the prior design are realised in working code. Together, these components facilitate the distributed content caching and control necessary for OpenCache to meet its aims. We also describe a number of tools used to evaluate OpenCache, including an example of an application that utilises the OpenCache API to control and configure a content delivery network. In the following section, we take this implementation and evaluate it in a number of scenarios, each of which exercises a particular functionality or design requirement.

Chapter 5

Evaluation

In this chapter, we aim to comprehensively evaluate the OpenCache design and implementation. In the first instance, we evaluate the suitability of SDN technology to provide the necessary redirection functionality required for OpenCache. This is presented in Section 5.1. Further to this, we examine the benefits of deploying OpenCache, with a particular focus on evaluating a number of recognised Quality-of-Experience metrics. These results reflect the direct benefit to end users and are described in Section 5.2. Finally, we demonstrate the OpenCache API by building and evaluating two sample applications. These exploit advances in infrastructure capability to dynamically adapt to changes in service availability and load. The outcome of this evaluation is explored in Section 5.3.

5.1 Redirection

The main goal of this evaluation is to investigate the feasibility of using Software-defined Networking (SDN) as a redirection technique. To this end, the OFELIA [24] facility was used for experimentation. OFELIA contained resources located in a number of physically-diverse universities and research institutions, spread throughout Europe. The scale of the facility is illustrated in Figure 5.1. Each of these locations consisted of a set of network switches and virtual machine hypervisors. Within each of these, resources were connected together to enable virtual machine instances created on different hypervisors to communicate with each other. Importantly, this network was OpenFlow-capable, facilitating unprecedented experimentation in network behaviour and providing the necessary

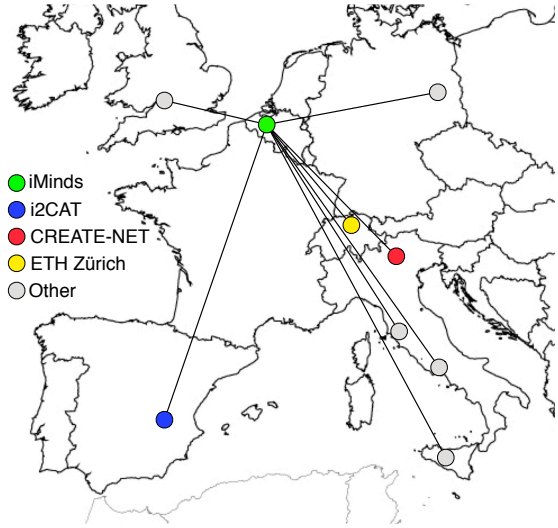

Figure 5.1: OFELIA Experimental Facility

functionality for our own experimentation. As the first OpenFlow-based testbed in Europe, OFELIA is based around the OpenFlow v1.0 specification [25].

Each of these locations (known as *islands*) could be connected together. Enabled through network connectivity to a central location, resources could be provisioned in a number of these islands, and then used and connected together in a single cohesive experiment. This process enabled the rapid set up of experiments without the need to host, run and maintain our own equipment. It granted a level of realism and scale not possible with simulation nor lab-based experimentation, important when we are considering the viability of OpenCache as a distributed Content Distribution Network (CDN).

To compliment this connectivity, OFELIA also utilised a common method of reserving resources, regardless of location or owner. This was achieved using a tool called OFELIA Control Framework [20] (OCF). A derivative of work developed for the GENI testbed, the OCF allowed experimenters to define their own experimental *slice*. This slice would contain a set of resources that belonged to the experimenter for the duration of their experiment. This includes virtual machine hosts and network switches.

An experimenter designs an experiment by defining a set of resources that they wish to include. These can be located in any of the encompassing OFELIA facilities. Once they are satisfied by this design, resources can be reserved and provisioned, thus creating their own slice for the duration of their experimen-

tation. These slices are also used to realise experimental isolation; a key facet of any scientific research. This allows multiple experiments to run concurrently, without the state of one impacting the result of another.

This isolation is realised in different ways dependent on which layer is concerned (compute or network). In the case of compute, this is achieved through the use of virtualisation technologies such as Xen [45]. Once an experiment is executed, a set of virtual machines will be created on the chosen hypervisors. These are under the full control of the experimenter, and can be used to run whatever application or tool that is desired. In OFELIA, each of these virtual machines runs a base Debian 6.0 Linux operating system [7].

Network isolation is facilitated through the same technology that underpins the experimentation on the testbed. By making use of software-defined networking, tools such as Flowvisor [154] can be used to ensure that experiments are granted their own unique flowspace which prevents traffic from crossing from one network domain into another. This can be realised in the testbed a number of different ways, but in the case of OFELIA, each experiment is defined by a unique VLAN tag. This allows modification or matching to be completed on any of the other IP headers, but means that VLAN-based experimentation is not possible. Flowvisor acts as an intermediary controller, rewriting requests to match the allocation defined in the experiment's design.

The experimental topology consists of 6 virtual machines, each with a unique purpose. Firstly, a video client was located in the ETH Zürich island, and consisted of a base operating system with the VLC [44] media-player client installed. VLC's MPEG-DASH HTTP adaptive streaming features were used in this experimentation. This client was then provided with one of two manifest files. These files describe the location, format and importantly, the quality levels available for playback. The two variants of this manifest file are different only in the location at which the content is stored. This can be at one of two islands: either CREATE-NET or i2CAT. The content used in this case is consistent throughout experimentation, and is reference material derived from *Big Buck Bunny*, a public domain motion picture [33]. The duration of the playback offered by this material is 9 minutes and 56 seconds.

At the beginning of each experiment, the client will start playback with one of these manifest files, and is allowed to run until playback is complete. Once this has occurred, the experimental run is finished. During this run, two important metrics

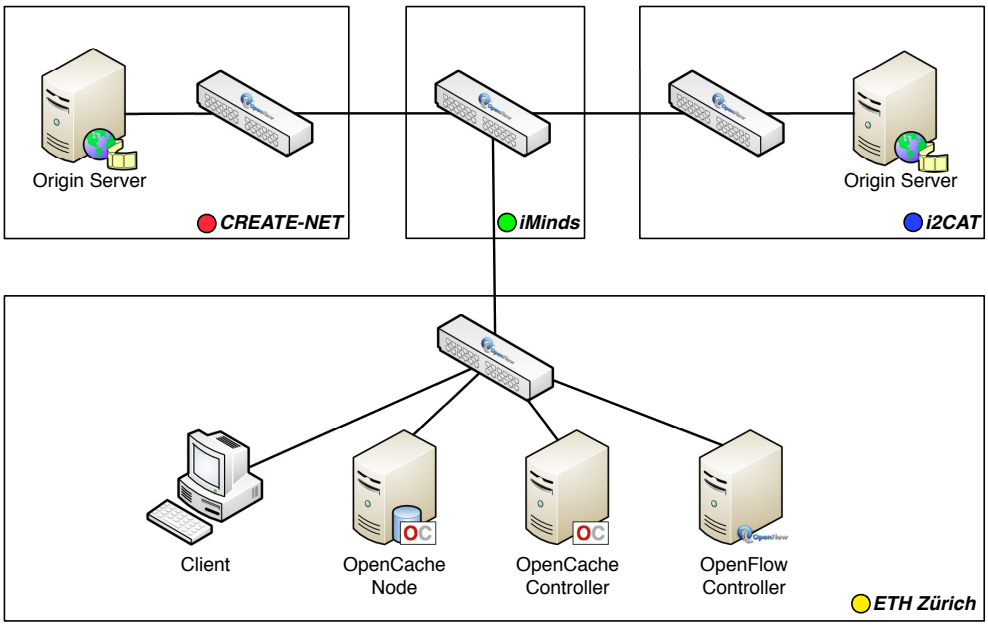

Figure 5.2: OFELIA Evaluation Topology

are recorded. Firstly, the startup delay is measured. This indicates the amount of time necessary for a client to start playback once initialised. Secondly, the external link utilisation is also observed. This is the amount of bytes transmitted over the link connecting the client's island with the other facilities. Together, these metrics indicate the impact of caching, and in particular, the use of software-defined networking to achieve such.

The experiment also contains two virtual machines that are responsible for hosting the content in the first instance. These origin servers utilise the SimpleHTTPServer module, part of Python's core libraries [39], to serve content requested by the client over HTTP. Each of these holds the full set of content, including all possible quality levels available. In each of the experimental runs, we use one of these two instances; that is, content will be retrieve from one of the two servers, each of which is located in a different geographical island.

Adding to this topology, we also create an OpenCache node, which is also located within the ETH Zürich island. This will be used in experimental runs where a cache is required. To control this cache, we also deploy an OpenCache controller in the same island. This will have responsibility for aforementioned node for the duration of the experiment. Through this controller, three different scenarios are realised (these are discussed later in this section).

Finally, we also locate an OpenFlow controller within the same island. During the experiment definition, we indicate that this virtual machine is responsible for the behaviour of the network. The OpenFlow controller is responsible for the behaviour of all of the switches in the topology, regardless of where they are located. To ensure reliability, this communication is made over a separate management channel. This ensures connectivity between the switches and controller, regardless of experimental conditions, granting the stability required in the experimentation. The OpenFlow controller used for this experiment was Floodlight v0.90 [11]. This controller supports v1.0 of OpenFlow, in line with the capability of the switches in the OFELIA testbed.

Also included in the topology are 4 OpenFlow-capable switches: one located in each island, plus a central switch which connects each facility together. In order to supply basic forwarding functionality in the network, regardless of the presence of experiments, a simple layer-2 forwarding module was loaded within Floodlight. This enables basic end-to-end connectivity to be established, and relies upon a simple MAC learning method.

Together, these elements combine together to make the evaluation environment. The overall topology is visualised in Figure 5.2, with each island represented by a coloured circle. This setup enables evaluation across three distinct scenarios. By duplicating resources and content across two different islands, we are able to provide six unique sets of results. This allows us to determine the effect of the different link capabilities between each of these destination islands and the central interconnect, whilst isolating the impact of caching.

The first of the three scenarios acts as a baseline for the remaining experiments: a run consists of a client requesting content from an origin server. This is shown in the *no-cache* case in Figure 5.3. No caching is involved in this interaction, and the full set of content is retrieved from the server without interference from any modifications made in the network. The second scenario introduces the cache into the experimentation. In this case, modifications are made to network flows to redirect them to the cache. As these will be initial requests, at least from the perspective of the cache, the node will not hold any of the content. As a result, it will have to retrieve this content from the origin server, and deliver it to the client. This is the *cache-miss* case in Figure 5.3. The third scenario is similar to the first, with the modification that the node already has the content cached. That is, it will not have to fetch the content from the origin server, and

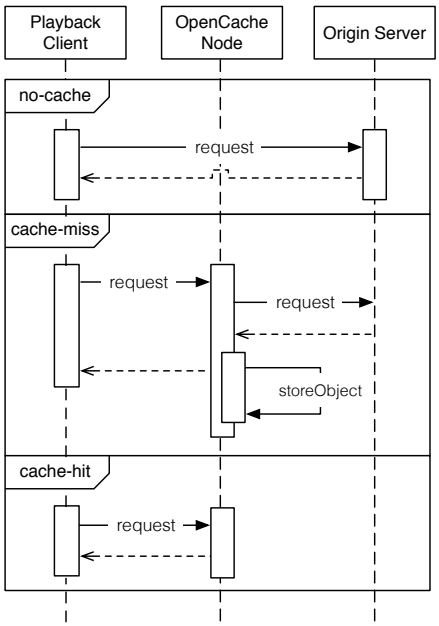

Figure 5.3: Request Message Flow

can deliver it straight from its own copy of the content. This is illustrated in the *cache-hit* case in Figure 5.3.

These three scenarios cover all of the iterations facilitated by the *start*, *stop* and *fetch* OpenCache commands. These are issued to the OpenCache controller present within the experiment. When the latter experiments were initialised, the *start* command was issued to the single node present in the OpenCache deployment. As mentioned before, this was located in one of the two islands, depending on the experimental run. In the scenario where no content exists on the cache, this would be all the required setup. Once the experiment had completed, the service could be cleaned-up using the *stop* command, which would remove any content that had been stored in the process of the evaluation.

The same process was repeated for the final experiment, where content is served from the cache. To ensure that content is always delivered from the cache without the need to fetch it from the origin server, the *seed* command is used. This will pre-emptively force the cache to retrieve and store the content, and is run before the client commences playback. As the client is requesting content based upon their current observed bandwidth, combined with the realistic nature of the links used (which vary in latency and throughput over time), the full dataset is retrieved by the node. This ensures the necessary cache behaviour.

	No Cache	Cache-miss	Cache-hit
Average Startup Delay (s)	2.484	2.088	1.639
Improvement over Baseline (%)		16.02	34.02
Standard Deviation (σ)	0.208	0.225	0.226
External Link Usage (Bytes)	105,734,144	105,827,872	0

(a) CREATE-NET

	No Cache	Cache-miss	Cache-hit
Average Startup Delay (s)	2.212	1.982	1.441
Improvement over Baseline (%)		10.40	34.85
Standard Deviation ()	0.145	0.138	0.226
External Link Usage (Bytes)	105,734,144	105,827,872	0

(b) i2CAT

Table 5.1: OFELIA Evaluation Results

5.1.1 Results

As mentioned previously, the most import aspect of this initial evaluation was to demonstrate that OpenFlow could be used to achieve the required redirection techniques. This required significant development, testing and subsequently evaluation. Through the latter, we determined that it was possible to achieve the desired behaviour using the network itself.

The experimental results are presented in Table 5.1, and are averaged over 20 experimental runs. Table 5.1a represents results gathered when using CREATE-NET as a destination island, whilst Table 5.1b represents observations made when using i2CAT to locate our origin server. They show that using OpenFlow for redirection is a feasible and viable method of doing so. There is no noticeable impairments introduced in this process, with users still receiving a service, even with the addition of OpenCache into the delivery chain.

Startup delay is often used a key differentiator in Quality-of-Experience measurements. In the initial experimentation OpenCache reduces the startup delay by minimising the amount of time a user has to wait before the content can start playing. Compared to the baseline, handling the request from an OpenCache node offers a 34% improvement. This is a best-case scenario, where the cache already holds the content and can deliver it without having to fetch it first. This result is consistent across the two destination islands, and also exhibits a low amount of standard deviation.

The evaluation also demonstrated the potential network efficiency gains that can be made by serving the content locally. As the origin servers in this case are located close to the OpenCache nodes, the traffic did not have to traverse any metered connections. However, in circumstances where this would be the case, the network provider would have been charged for each request that traversed that link.

To observe the amount of traffic in all scenarios, we monitored and recorded the size and count of packets travelling towards the origin server. As expected, in cases where content had to be fetched from the origin server, the amount of traffic observed was equivalent to the size of the video. This is approximately the same regardless of whether it is the client or the OpenCache node (in the case of cache-miss) requesting the video. In scenarios where the content is delivered entirely from the cache, the traffic heading towards this origin server is reduced to nil. This shows the reduction in load on both the server and the network possible through the use of localised caching.

It is important to note that neither the origin server nor the client have not been modified to achieve this evaluation. This satisfied our requirement that OpenCache be transparent in operation, and work seamlessly with existing delivery technologies.

5.1.2 Discussion

During the evaluation, it was observed that the modifications made in the network in order to provide this redirection have an impact on performance. Although the client received sufficient throughput to download and consume the video, this maximised the processing resource on the switch providing the modification. As OpenFlow was a relatively new technology at the time of the evaluation, these early implementations could not realise some of the more advanced functionality offered by the protocol (particularly in the fast-path of switch hardware). This meant that operations were performed in software instead, which is considerably slower as it contains no hardware-acceleration. The header rewriting operations necessary for the redirection used in OpenCache is included in the functionality. Nonetheless, a single-client evaluation was still possible, and this limitation did not hamper or degrade the rest of our experimentation.

OpenCache also manages to reduces startup delay in cases where it has to

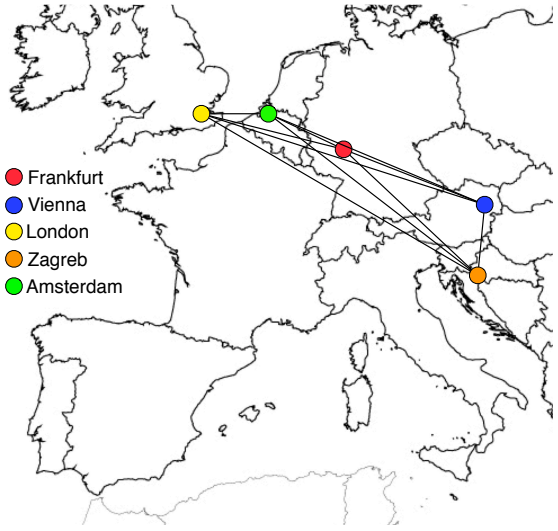

Figure 5.4: GOFF Experimental Facility

fetch the content from the origin server first. In this case, the improvement shows more variance over the two islands, likely due to the differences in the underlying resources found in each location. Although this behaviour seems unexpected, it is still statistically significant, and can be attributed to the aggressive nature exhibited by the service in OpenCache when fetching content. As OpenCache is multi-threaded, the process for retrieving content will spawn a number of threads, each of which will begin to download a specific part of the requested file. This parallel download process is much faster than that used in VLC, hence the reduction in startup delay, even when the content has to be fetched. Obviously, this behaviour may not be appropriate in all circumstances, as it places a larger amount of load on the origin server. However, the initial fetch of content can often be a bottleneck in the caching process, particularly if it has not been prefetched. This helps to resolve this, at the cost of more instantaneous network utilisation and resource consumption on the origin server.

5.2 Quality-of-Experience

The second evaluation focuses on evaluating the impact that an OpenCache deployment may have on client Quality-of-Experience (QoE). For this purpose, we used Scootplayer to provide evaluation beyond the startup time observed in the previous experimentation. As described in Section 4.4, Scootplayer is a fully in-

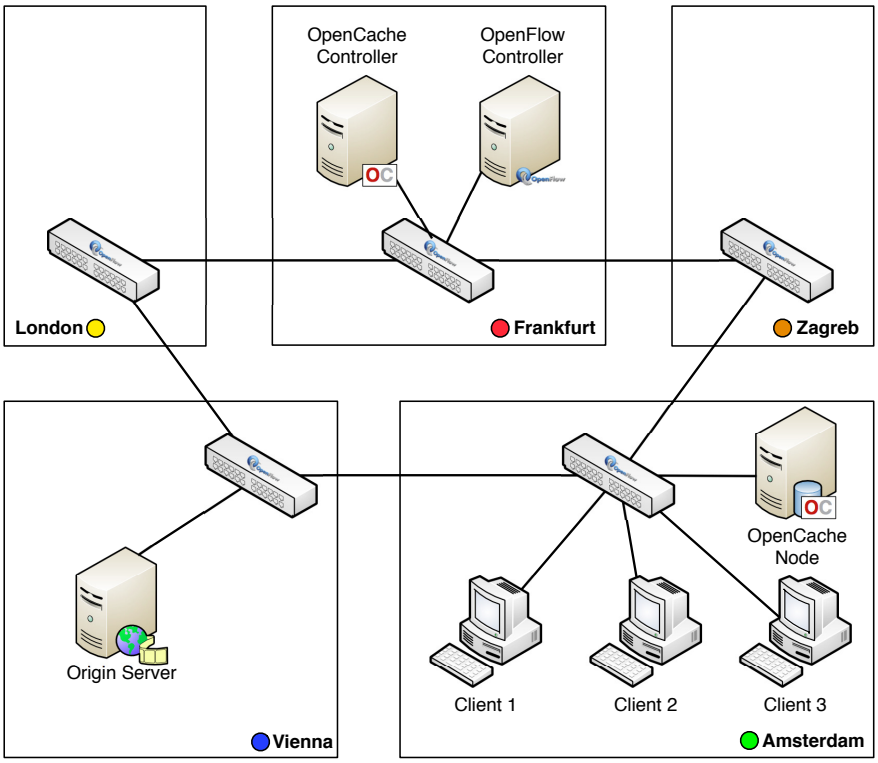

Figure 5.5: GOFF Evaluation Topology

strumented, MPEG-DASH-compliant player designed specifically to monitor and record a number of recognised QoE metrics. This evaluation again takes places in a real-world environment and considers a multi-user scenario. This consists of a number clients requesting content at the same time. This realism was furthered with the inclusion of link emulation, used to modify the characteristics to more closely resemble a congested network.

Figure 5.5 illustrates the topology used during our evaluation. We use the full resources of a large-scale pan-European testbed: the GÉANT OpenFlow Facility (GOFF) [12]. The location of each facility is outlined in Figure 5.4. The composition and tools used to realise this evaluation are similar to those used in the OFELIA experimentation described in Section 5.1. Resources are located across a number of separate physical locations, or *islands*. These are located in various cities throughout Europe and are connected together using GÉANT's network. This ensures high capacity and throughput between sites.

As with OFELIA, sites are based around virtual machine resources under a Xen hypervisor [45]. These can be dynamically provisioned using the control

framework, which is again a derivative of that used in OFELIA. Using this, we provision a number of virtual machines to act as network controllers, OpenCache controllers and OpenCache nodes. Similarly, clients are also provisioned in this way, each running an instance of Scootplayer. These request content from an origin server also provisioned using the same interface. These origin servers serve the same Big Buck Bunny dataset as used in the previous experimentation. The virtual machines also run using the same Debian image used in the OFELIA testbed, with the same SimpleHTTPServer used to serve content. The specific locations of each of these elements is as follows: the OpenFlow controller and OpenCache controller were located in the Frankfurt facility, whilst the origin server is located in Vienna. The remaining elements (multiple video clients and the OpenCache node) were located in Amsterdam. The GOFF testbed provides connectivity in a full-mesh configuration, with each location connected to every other.

To interconnect these clients, OpenFlow switches are used in each location. In comparison to the OFELIA testbed, which consisted solely of hardware-based switches, GOFF contains software-based switches. These too support OpenFlow, and are based around Open vSwitch [22]. These switches run using commodity servers co-located with virtual machined resources. Much like OFELIA, experimental separation is realised using slicing. This grants each experiment a logical topology within the network and prevents experiments from interacting with other. Similarly, each switch connects to the OpenFlow controller using a separate management channel. As with the previous experimentation, the OpenFlow controller used was the same version of Floodlight. This controller also acted as a basic layer-2 forwarding switch, ensuring packets could be switched between the constituent locations included in the topology.

This experimentation uses the same basic subset of OpenCache commands, namely: *start*, *stop* and *fetch*. These are necessary to ensure that the evaluation environment is initialised correctly, as well as cleaned up appropriately. As before, the *fetch* command is used to pre-fetch content and ensure that is always available when a cache-hit is desired.

This evaluation contains a more diverse set of experimentation, designed to evaluate OpenCache under different circumstances. Firstly, we introduce variation in the links between each of the locations. The first scenario in this case is the baseline: we do not modify the links in any way. The characteristics expe-

rienced by the client are those inherent in the link; the average latency between two of the islands was 30ms, with no packet loss.

To create a more realistic scenario, in which clients experience packet loss and latency similar to that of the Internet, impairments were emulated using the *dummynet* [70] tool. The characteristics of the link are set so that each packet can fall into one the following three probabilistic categories at each point in time: 45% probability to encounter default link characteristics, 45% probability that an additional 50ms of round-trip time and 0.1% packet loss is experienced, and 10% probability that an additional 150ms of round-trip time and 0.1% packet loss is observed. This creates random packet drop and can simulate the impact of congestion over multiple paths leading to out-of-order packet delivery.

To further the situation, we consider an evaluation scenario whereby multiple users are accessing the same set of content over a small period of time. The baseline in this scenario is a single-user environment, where only one client is accessing the content. In the multi-user scenario, we scale the amount of users up to three. These users will start accessing the content over a set interval: one client will start at the beginning of the experiment, the next at 30 seconds from the start, and the last at 90 seconds. This scenario should demonstrate the possible advantages, especially when multiple clients are accessing the same content. This is the likely situation when a cache is deployed to serve many clients at once.

In total, we have 4 individual experiments, with all possible link and user combinations explored. Each of these experiments contains three experimental runs. In the case of single-user experiments, we examine the three cases discussed in the previous evaluation (Section 5.1). In the multi-user experiments, we compare the results between the three different clients, each beginning playback at their allocated times (client 1 at the start of the experiment, client 2 at 30 seconds from the start and client 3 at 90 seconds from the start).

Building on the previous measurements of startup delay, we expand the evaluation with three more metrics, namely: a count of video bitrate changes, a weighted average video bitrate and the minimum video bitrate. These are metrics derived from contemporary literature [117, 85, 119] and recorded with the use of Scootplayer; instrumented specifically for this purpose.

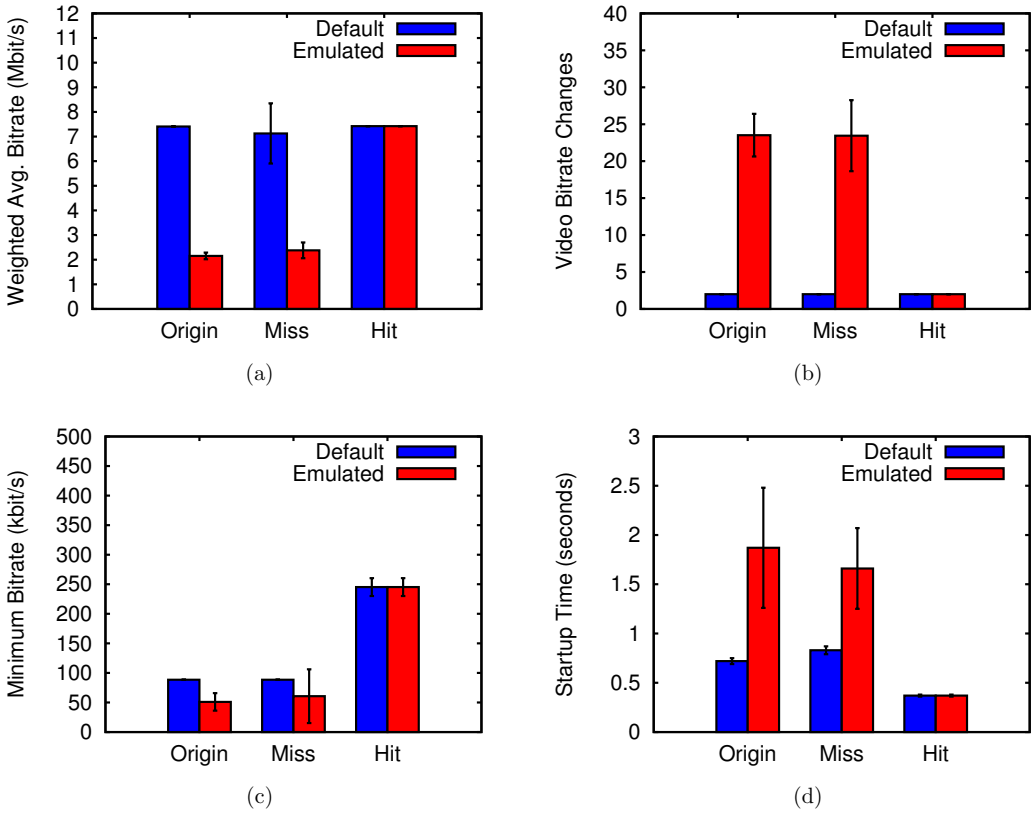

Figure 5.6: GOFF Single-user Results

5.2.1 Results

Through our evaluation, we demonstrate the benefits of OpenCache in multiple environments. These are illustrated through the use of recognised Quality-of-Experience metrics, which show a notable improvement across every measurement. More specifically, we examine the single-user results in Section 5.2.1.1, and the multi-user results in Section 5.2.1.2.

5.2.1.1 Single-user

Figure 5.2.1.1 shows the results for a single-user environment. Each subfigure represents one of the 4 QoE experience metrics used in our evaluation. These are derived over 20 experimental runs. The three scenarios in each subfigure are those previously described: *Origin* is content fetched straight from the remote origin server, *Miss* is content delivered from the OpenCache node that first has to

be fetched from the remote origin server, whilst *Hit* is content delivered straight from the OpenCache node with no need to fetch it first. The scenarios are all orchestrated by issuing the appropriate commands to the OpenCache API which will clear the cache content, as well as pre-fetching it as required. Each scenario is then shown in two different circumstances: the blue histogram uses the default link characteristics, whilst the red histogram represents link emulation with the aforementioned defects.

The first metric analysed is the weighted average video bitrate, shown in Figure 5.6a. In cases where the default link characteristics are used (the blue histogram), there are little to no difference in the average bitrate between each of the three cases. This can be attributed to the high capacity and low latency link used in the *Default* case. It is likely that the player quickly rose from the initial bitrate towards the maximum bitrate available in the representations. Once there, playback did not deviate from the maximum because of the stability and quality of the links traversed. With the use of emulated links (shown in the red histograms), a significant reduction in the average bitrate is observed in the cases where requests have to traverse the effected links (*Origin* and *Miss*). As the capacity and stability of these links is now reduced, the clients receive an overall worse quality level throughout playback. In this case, we observe over three times higher weighted average bitrate when the content is delivered from the cache.

The second metric considered in this evaluation is a count of video bitrate changes, shown in Figure 5.6b. As with Figure 5.6a, the default link characteristics provide consistent results regardless of where the content is fetched from. This is shown in the small number of bit rate changes, which is again accounted to the fact that the player simply moves from the initial bitrate to the maximum bitrate, and stays there for the duration of playback. In the case of the emulated link, we see similarities between the *Origin* and *Miss* cases, where the amount of changes is high (23 in total). This is due to the latency and packet-loss introduced in the path. Since both cases have to traverse the external link, they share a similar level of impairment, which naturally impacts QoE significantly. By delivering the content from the OpenCache node, we offer the same amount of low variability observed when the default link characteristics are used, which results in a more stable playback for the user.

The third metric recorded by Scootplayer is the minimum bitrate, illustrated in Figure 5.6c. This metric records the smallest playback bitrate observed during

playback; typically at the beginning of playback. It would also capture circumstances where link capacity dropped drastically, but this was not observed during our experimentation. As a result, the minimum bitrate in our experimentation is a good indicator of the estimated throughput calculated by the Scootplayer in the first instance. This is demonstrated in the *Default* link results, as both the *Origin* and *Miss* case have to fetch content from a remote location. The links necessary to do so do not have the same throughput as two devices connected together using the same physical switch. This is why the *Hit* result offers a higher minimum bitrate. These results are mirrored across the emulated link experiments, albeit with the intentional impairments causing an even lower bitrate to be experienced in the cases of *Origin* and *Miss*. The *Hit* results remains the same regardless of the link, as would be expected. This provides a minimum bitrate four times higher than having to fetch the content remotely.

The final metric captured in our experimentation is the startup time or delay. This is shown in Figure 5.6d. In the single-user results, this is a repeat of the experimentation conducted earlier on the OFELIA testbed. In the case of default link characteristics, we observe similar results, with the *Origin* and *Miss* cases proving slower at startup compared to when the content is delivered straight from the cache. This situation worsens when the emulated links are introduced, which causes the *Origin* and *Miss* cases to produce even slower startup times, whilst the *Hit* cases remains consistently smaller. In this scenario, delivering content from OpenCache provides a 4-fold improvement when compared to traversing the external link.

Across each of these metrics, we demonstrate the QoE benefits of using OpenCache. These benefits come from the ability to redirect requests for content close to the user. By delivering content nearby, we avoid the necessity to traverse external links which are susceptible to various network impairments which negatively impact the quality received by end clients. These improvements are particularly evident when the link characteristics are changed to better reflect conditions found in access networks.

5.2.1.2 Multi-user

The same four metrics are used in our second set of evaluations, which involve multiple clients rather than a single client. The results from this evaluation are

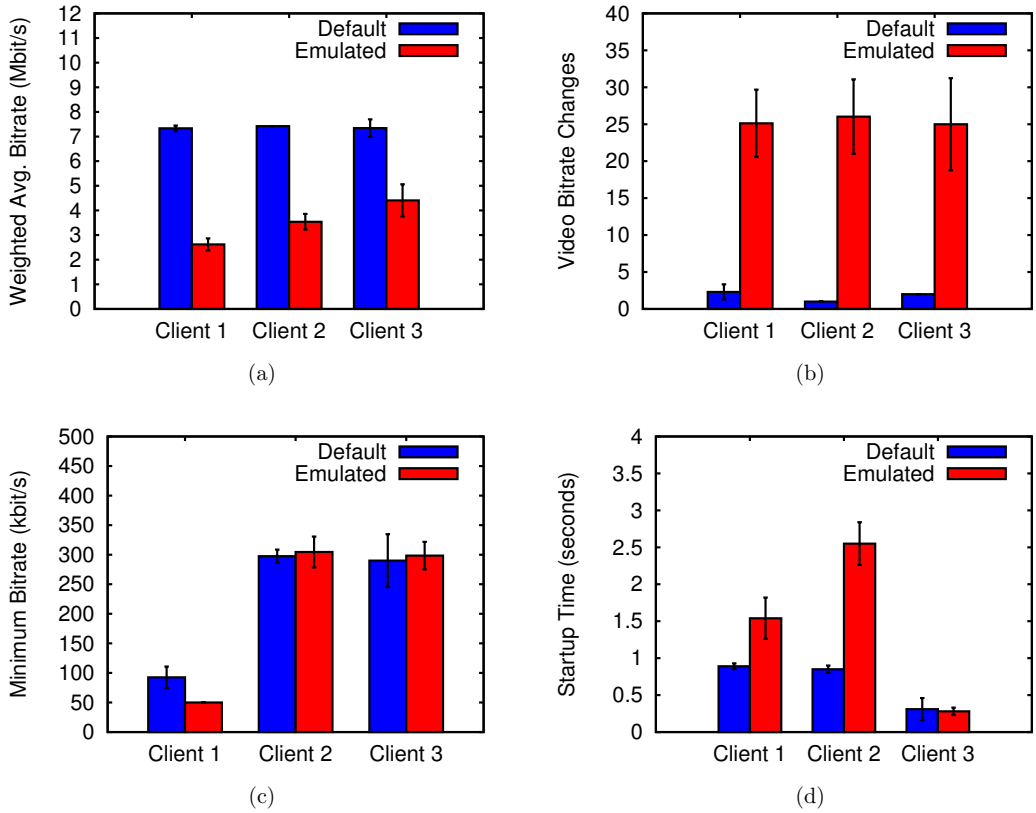

Figure 5.7: GOFF Multi-user Results

shown in Figure 5.7. As before, each of the subfigures represent the results from one of the Quality-of-Experience metrics identified as important. Experiments were repeated 20 times for each case. In each case, no content was pre-cached on the OpenCache node, and cache-hits and cache-misses were allowed to happened naturally. The three clients are setup in an identical fashion with each running the same software and connected to the same switch. The same link characteristics are employed as before, with *Default* shown in the blue histogram (indicating an unmodified link) and *Emulated* illustrated in the red histogram (indicating a link with intentional impediments). The clients start at predefined times: Client 1 starts at the beginning of the experiment, Client 2 starts 30 seconds afterwards, whereas Client 3 starts at 90 seconds from the beginning.

As with the single-user experimentation, we first look at the weighted average bitrate, which is shown in Figure 5.7a. In the case of the *Default* link setup, the average bitrate is consistently high, regardless of the state of the cache or

when the client starts; the client will move to the maximum bitrate present in the representations very quickly and sit there for the remainder of the playback, resulting in a high average bitrate. However, when the emulated link is introduced, the average bitrate is no longer consistent amongst the clients. The Client 1 is impacted the most, as it starts first and will request content from the cache when it is empty. As a result, all of its requests will be cache-misses. When Client 2 starts 30 seconds later, a number of the initial requests will be cache-hits, as Client 1 will have already requested the content. However, the speed in which these requests are delivered will alter the throughput observed by the client. As Scootplayer believes it has increased resources, it will eventually adjust its estimation to request higher bitrate video. This will result in more cache-misses, as the cache does not hold these higher quality representations; time equivalent segments are available, but only in lower quality variations. These cache-misses mean that the client reverts back to receiving a lower average bitrate, albeit higher than Client 1. The same situation is repeated for Client 3, although it benefits from an extended period of cache-hits and thus receives a higher average bitrate. In all cases, the average bitrate is lower than that received under the *Default* link conditions.

With the count of bitrate changes (shown in Figure 5.7b), the standard link characteristics again ensure that a client experiences minimal changes. These are attributed to a rapid switch to the maximum bitrate, which it then remains at for the duration of playback. When the emulated link is used, the client experiences a large amount of changes, as it is now subjected to the latency and random packet-loss found on the link. This situation is not bettered when subsequent clients begin playback, as even with a limited amount of content stored in the cache, the prevalence of cache-misses ensures that requests are still subject to the link impairments and variability.

The minimum bitrate shows a significant improvement after the initial client has already requested content in both *Default* and *Emulated* cases. This is illustrated in Figure 5.7c. As described in the previous section, the minimum bitrate received by the client is often observed at the beginning, unless the link degrades significantly at some point during playback. The initial estimation of bandwidth is therefore determined during initialisation. As the files requested during this phase will be cached regardless of quality levels, the estimation will be consistently higher for subsequent requests as the content is delivered from the cache.

This is reflected in the fact that Client 2 and 3 both have similar minimum bitrates, having both benefited from the delivery of cached files after Client 1's initial startup.

Finally, we analyse the startup times of each client in Figure 5.7d. These results are closely related to the minimum bitrates observed in Figure 5.7c. Client 1 requests the content as normal, including the initialisation files necessary. All of these fetches result in cache-misses at the OpenCache node. When Client 2 starts playback, it receives these initialisations from the cache. As mentioned previously, this results in an increased throughput estimation by the player. As a result of this estimation, the client starts to request higher bitrate chunks (hence the increased minimum bitrate). However, retrieving these chunks takes more time, especially as they will produce cache-misses. This results in an increased startup delay in Client 2. Client 3 follows the same principle, estimating higher throughput from the outset. Yet, in this case, when it goes to retrieve the initial higher quality chunks, these will already be stored on the cache. Thus, Client 3 has the smallest startup delay whilst also receiving a higher minimum bitrate.

These results show how the potential QoE benefits increase as more users request the content. As this number grows, so do the potential QoE and efficiency gains. These gains are intertwined with the availability of content on the cache: more clients increases the chance of a cache hit for subsequent client requests. However, in cases where adaptive streaming is used, the interaction between cache availability and the throughput estimation can lead to unpredictable behaviour. This is observed in our experimentation and discussed in further detail in the following discussion section.

5.2.2 Discussion

This evaluation revealed the efficiency of OpenCache in the case of a cache-miss. For example, in Section 5.2.1.1, OpenCache only presents a 15% overhead in startup times when compared to the client directly fetching the content from the origin server. This is despite the fact that an additional hop is introduced in the network topology. Yet the case of 100% cache-hit ratio is also unrealistic. In a real-world, a cache would likely hold only parts of the entire set of content, especially if the content available is only that has previously been requested by a client.

In Section 5.2.1.2, we see a magnified version of this problem, as clients are requesting the same set of content, but at different quality levels. This causes inefficiencies in the cache storage, as multiple copies of the same content exist. Although these are at different quality levels, they still represent the same segment of playback time-wise. To avoid this mismatch, which can ultimately lead to unpredictable user experience, it is necessary to in some way manipulate the cache.

This problem is somewhat unique to adaptive streaming techniques, but as these are becoming the norm, it requires careful and due consideration. All of the evaluation presented in this section uses the same player, developed for the purpose of revealing key QoE metrics. However, different players will likely have slightly different bandwidth estimation techniques, different sizes of queues, etc. This makes predicting which content that a client will request an extremely computationally expensive process, especially without cooperation between the cache and player.

The situation is exacerbated in cases where there are many different quality levels: the higher the count, the more likely that this mismatch will occur. Intelligently pre-caching content or manipulating the objects served (regardless of request) are two methods that can avoid this situation. If executed correctly, they can still provide similar performance to a fully populated cache, whilst reducing storage requirements. The success of this would largely be dependent on the applied technique and logic, but as mentioned previously, made difficult by variation in client behaviour.

However interfacing with the cache is a simple process when using the API. Realising this behaviour would be possible with the use of commands such as *seed* and *fetch*, primitives in OpenCache. As the required logic would likely be based in software too, developers can experiment and adapt to find a beneficial solution that works best with the content, deployment and clients. In the following section, we evaluate this API more closely, including exercising some of the more advanced features present in the specification.

5.3 Application Programming Interface

This third evaluation aims to demonstrate the applicability of the OpenCache API, described previously in Section 4.1.4. As discussed in Section 4.3, this API can be used to implemented applications which are placed semantically *on-top* of the cache infrastructure. For this evaluation, we use this to implement novel cache behaviours through the development of two distinct applications. These highlights the benefits of control and service plane integration, showing how the dynamic nature of both can be used to achieve rapid response to changing conditions.

The applications chosen for this evaluation are based around resiliency in a cache deployment. As well as evaluating the suitability of the API, they also require multiple OpenCache nodes to be orchestrated together to provide a consistent service. In the first application, we implement a load balancing application, described in Section 5.3.1. This is concerned with pre-emptive avoidance of failure, achieved through effectively managing the load on different OpenCache nodes. The second application is a failover monitor, described in Section 5.3.2. The behaviour differs in that it is designed to detect unscheduled and unpredictable downtime in a deployment.

Both applications have their own distinct logic, which operates independently to OpenCache itself, and relies on the API for interaction. It is important to note that load balancing and failover behaviours would often be individual hardware appliances in their own right. As these applications are built entirely in software, there is the possibility for detailed customisation and optimisation, dependent on the deployment requirements. This could include utilising input from other sources, such as Network Management Systems (NMSs) or Quality of Experience (QoE) measurement frameworks. This evaluation demonstrates only one example of how to implement each; alternatives would be equally feasible.

This evaluation utilised a subset of the resources within the Fed4FIRE [10] testbed. This testbed relies on a different set of supporting tools when compared to the two previously used. Although similar in function, the testbed enables more fine-grained resource allocation, defined through a specification format rather than a graphical user interface.

This evaluation involves facilities located in multiple physical locations, interconnected together to create a large-scale pan-European topology. The arrange-

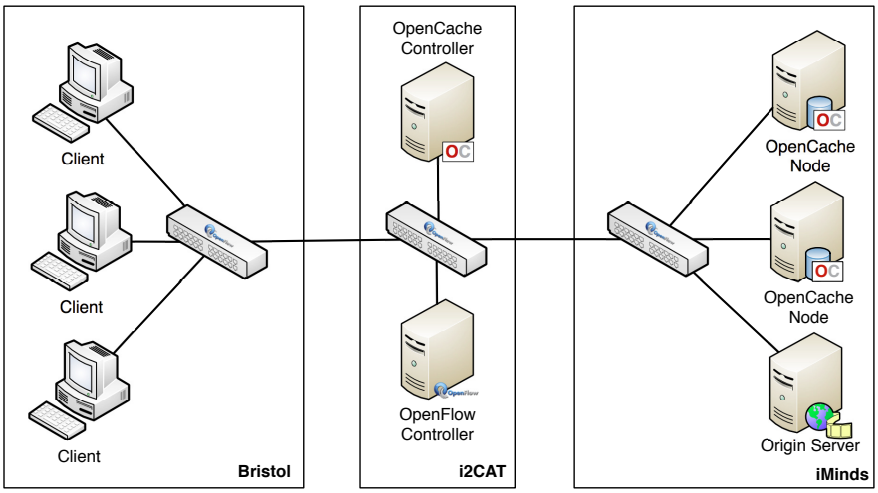

Figure 5.8: Fed4FIRE Evaluation Topology

ment of resources is shown in Figure 5.8. The Bristol facility hosted the clients; each running an instance of Scootplayer. These requested the same Big Buck Bunny content as used previously.

Serving these clients was an origin server located in the iMinds testbed. This was again running Python's SimpleHTTPServer module to handle HTTP requests. For requests to reach this server, they traverse a connecting link over the i2CAT testbed. The OpenCache nodes are also deployed alongside the server. For this experiment, we deployed two individual nodes, each running on a separate virtual machine. As before, the testbed provided the necessary SDN capabilities required for OpenCache to operate.

Different to the previous experiments is the prevalence of OpenFlow v1.3-compatible hardware and software switches in the topology. The previously used Floodlight controller did not have compatibility with this new OpenFlow specification. As a result, this evaluation uses the Ryu controller [38]. Previously, the OpenCache controller utilised an API in Floodlight to push specific flow rules into the network. However, Ryu does not contain such an interface by default. As such, we developed an interoperable API as a Ryu module and exposed this this to OpenCache. This requires no changes to OpenCache. This process also confirms the need for a consistent north-bound API; rather than developing a separate module for each controller, a standard interface and API would insure that OpenCache could communicate with any controller, regardless of the underlying technology and forwarding used in the network.

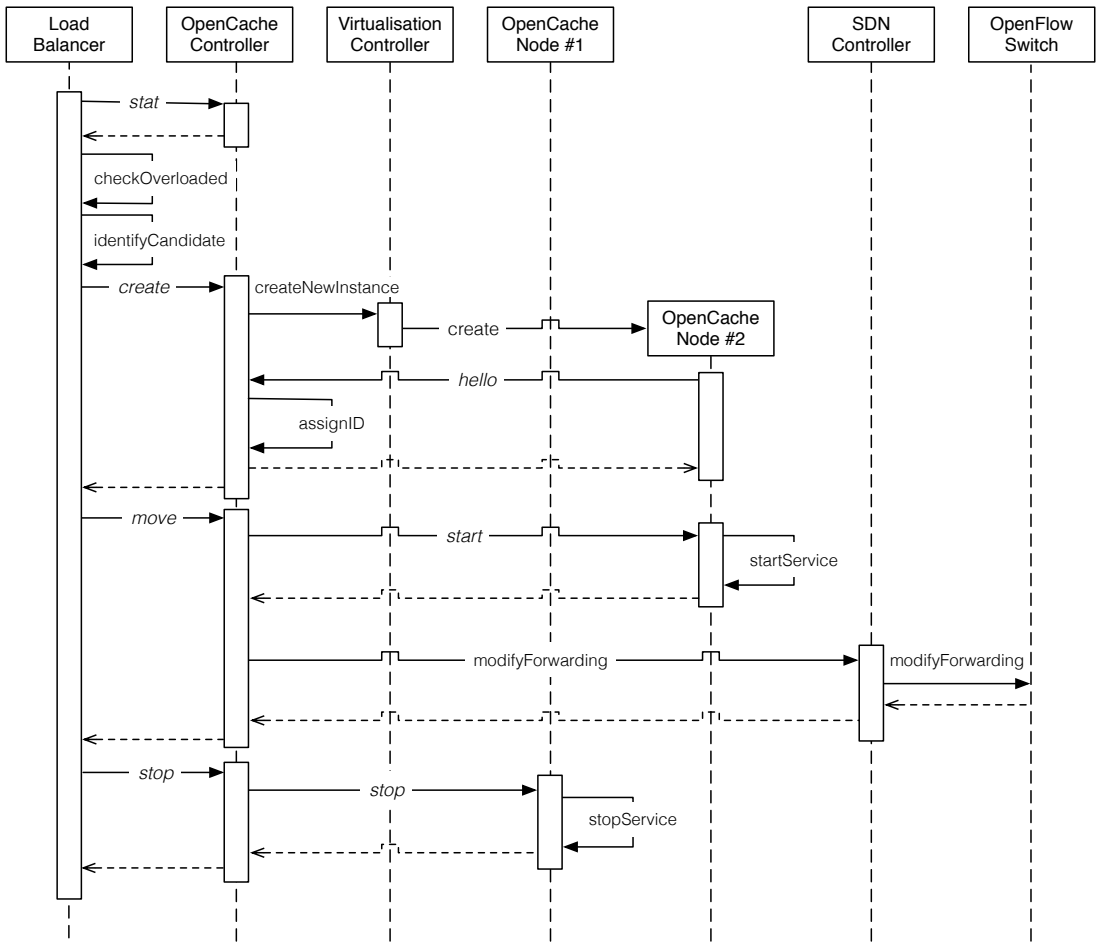

Figure 5.9: Load Balancing Message Flow

5.3.1 Load Balancer

This application effectively load balances requests for content between different OpenCache nodes. We will use the information ascertained from the statistics provided through the OpenCache API to determine when a node or individual service is deemed to be overloaded. This is only one potential method to determine the load on a service, and could also be supplemented with metrics ascertained through other means, such as information from the virtualisation or network controllers.

The process and message flow followed by our load balancer is described in Figure 5.9. The method calls shown in italics are part of the OpenCache API, whereas the other calls are outside of the scope of this evaluation (and thus differ dependent on the controller or application used). In the first instance, the load

balancer will request information about specific nodes it is monitoring (using the `stat` command). These statistics are returned back to the application, which will then analyse them to determine if any nodes are overloaded.

If any nodes are deemed to be currently consuming resources over the configured threshold, the application will seek to find a suitable candidate to move the load to. In this scenario, Node 1 is designated as overloaded. As this is the only node present in the OpenCache deployment, there will be no suitable candidate found as the target for migration. As a result of this, the application will create a new node on which the service will be migrate to. This will be achieved by the application sending a `create` command to the OpenCache controller, which will negotiate with the virtual infrastructure manager to bring a new OpenCache node online.

Once this process is complete, the load balancer will use the *move* command to start migration of the service. In the first instance, this will start an identical service running on the new node, Node 2. Once the service is started and ready to handle requests, the OpenCache controller will modify the forwarding plane and change the destination of the redirected requests for content.

At the completion of this process, Node 2 will handle all the requests that were previously destined for the Node 1. Consequently, the application will stop the existing service on Node 1, and thus free up the resources previously consumed.

5.3.2 Failover Monitor

The unavailability of content will undoubtedly negatively impact the Quality of Experience for a user requesting from the cache. When it is considered that content delivery networks are often provided as a paid service, an interruption will typically constitute a breach of a Service Level Agreement (SLA). As such, we demonstrate the ability of an application using the OpenCache API, to not only detect failure, but also react and remedy the situation quickly and effectively.

The message flow for this process is similar to that of load balancing. However, the application logic is slightly altered, as illustrated in Figure 5.10. Instead of detecting capacity (and consumption thereof), we are rather detecting the availability and uptime of a node. This is achieved by periodically polling the service in order to elicit a response. If the service does not respond, the service is deemed failed and offline.

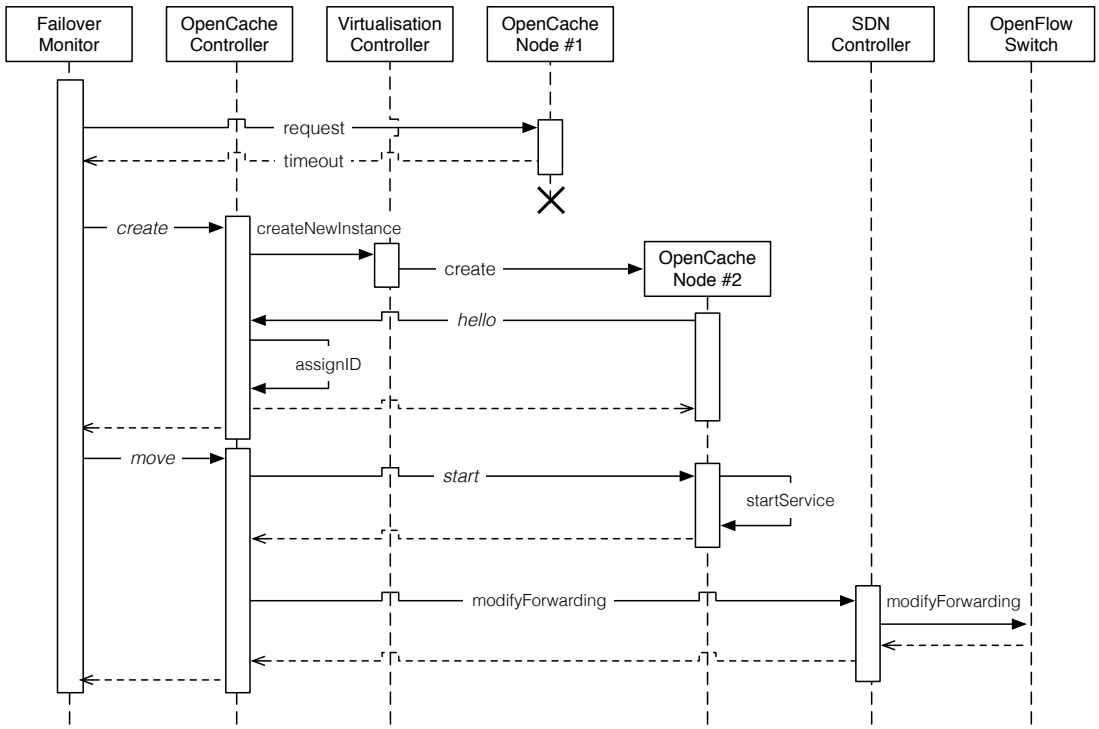

Figure 5.10: Failover Monitor Message Flow

Once failure is detected, the same flow (as with the load balancer) continues: the failover monitor will seek to migrate this offline service to an alternative node. If no existing nodes are available, a new virtualised node will be created. In the same way as the load balancer, the forwarding layer will be modified to match the current location and availability of services.

5.3.3 Results

The experimentation examined the impact of the two applications on Quality-of-Experience from the perspective of the client. In the case of the load balancer, we wanted to ensure that the load balancing process, and thus the migration of a service between two nodes, had minimal impact on the client. Five experimental runs where performed, each using a Scootplayer client to request content as required.

Scootplayer monitored the same set of QoE metrics as defined in previous experimentation. However, they showed no deviation during a load balancing operation, and are thus are not illustrated. However, an important metric was

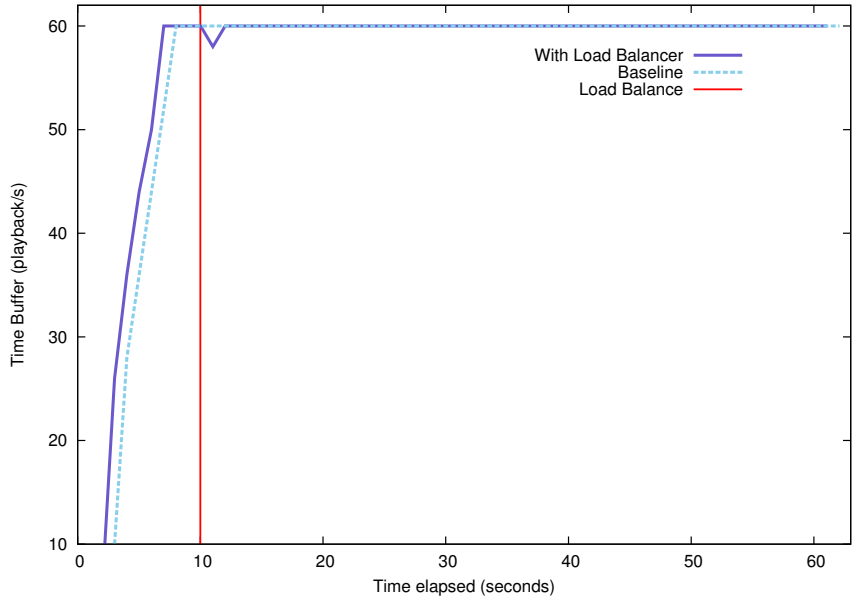

Figure 5.11: Client Buffer During Load Balancing

added for the sake of this evaluation: buffer occupancy. This clearly demonstrates the impact of load balancing, as shown in Figure 5.11, where the vertical line denotes the load balancing operation at approximately 13 seconds into playback.

At this point in time, a reduction in the amount of content buffered on the client is observed. The buffer, which holds a maximum of 60 seconds worth of playback, temporarily reduces in size to hold 58 seconds of playback. As we used 2 second segment lengths in our playback, this is equivalent to one chunk in the buffer.

This reduction, and the subsequent recovery, can attributed to the modification of the forwarding plane during a request. More specifically, the modification to the forwarding plane (necessary to implement the load balancing) will break the existing connection between the client and the cache. This will cause an application-layer request retry in Scootplayer. As the player is still consuming content (playing back), the fill of the buffer will be reduced momentarily. However, once the client re-establishes the connection, it will download the necessary content and refill the buffer back to the maximum 60 seconds.

As the load balancing has already taken place, it will now be downloading the content from the new cache node, rather than the overloaded one. It is important to note that although we only show buffer occupancy in this figure, other metrics

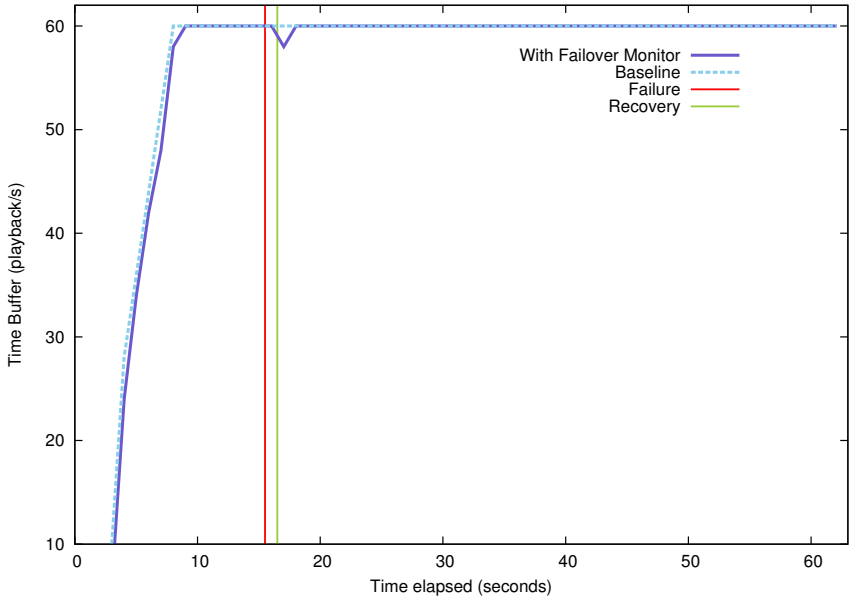

Figure 5.12: Client Buffer During Failover with 1s Resolution

were otherwise unaffected. We ascertained this fact from a baseline experiment (also shown), without the load balancing application present.

In the case of the failover monitor, we wanted to observe the impact of the time taken to respond to a failure, and how this may effect the client in a similar way to the load balancer. It became clear through our experimentation that the time taken to respond to a node failure is dependent on the resolution of detection. In the case of our example application, failure is identified through periodically polling the service to detect reachability. If the service does not respond, it can be assumed that the service is offline.

In our experimentation, we examine a number of different polling frequencies at 1, 5 and 10 second intervals. These are shown in Figures 5.12, 5.13 and 5.14 respectively. As before, 5 experimental runs were performed. This was repeated for each of the three polling frequencies. Similarly, a baseline experiment was undertaken for each resolution to ensure that the application had no impact on either the buffer occupancy or other QoE metrics.

In these figures, the first vertical line dictates when the initial node fails. The second vertical line indicates when the application detects the failure and remedies the situation by moving the requests across to a functioning cache node. It is evident that once failure occurs, the client continues to consume content from

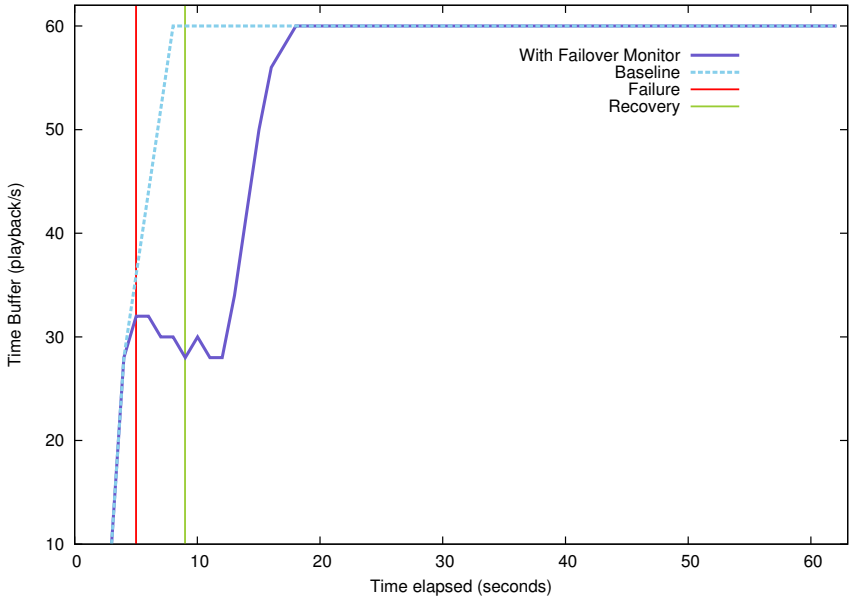

Figure 5.13: Client Buffer During Failover with 5s Resolution

the buffer, reducing its size. However, as new content cannot be retrieved, the buffer becomes significantly depleted.

The greater this depletion becomes, the longer it will take for the client to recover back to a fully-buffered state, as evidenced in Figure 5.14. During this period of time, the cache node itself will be under heavier load (more requests per second) and the client more susceptible to further interruptions. This buffer depletion continues to occur until detection takes places and appropriate actions are taken by the fail-over monitor.

5.3.4 Discussion

The amount of buffer depletion a client encounters is strongly linked to the detection resolution; a larger polling interval will result in the service remaining in a failed state for longer. The impact on the client is that it cannot retrieve new content, and moves closer to the buffer becoming empty. At this point playback will stop. An operator must therefore consider client requirements and resources before establishing a suitable value for polling.

This interval will be driven by the amount of buffer their customer's playback clients can accommodate. Although this thesis focuses on the delivery of video, many applications running over the Internet do not necessarily have a buffer at all.

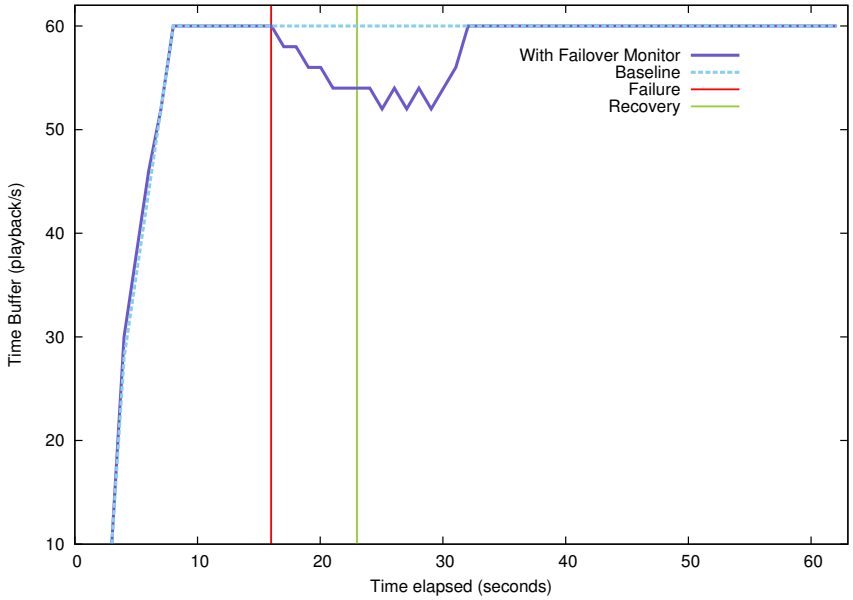

Figure 5.14: Client Buffer During Failover with 10s Resolution

As a result, such applications have no inherent ability to deal with unavailability of a service. As latency and failure can be potentially crippling to a service [88], it would be recommended that in these cases, a resolution interval should be set at the highest possible frequency without incurring a significant messaging overhead that would further impede the delivery of the service.

This evaluation considers the use of both software-defined networking and service-layer programmability to counter potential failures. These failures are constrained to the compute-side of the infrastructure, whilst relying upon the availability of the network to modify request handling. However, in some cases, a simultaneous failure in the network may also occur, especially if the root cause is shared (a catastrophic power failure for example). In these worst-case scenarios, the techniques demonstrated in this evaluation can be combined with network-based restoration techniques [60] to ensure that service is resumed as quickly as possible.

5.4 Summary

In this chapter, we evaluated OpenCache in a number of different ways. Each of these evaluations is designed to examine a specific aspect of OpenCache, particu-

larly in respect to the originally envisaged design goals (summarised in Table 3.1). The success and extent to which these have been evaluated is discussed in the following section.

For OpenCache to be a viable alternative to current content delivery techniques, it must first meet the core requirements sought from any such content delivery platform. In the first evaluation (outlined in Section 5.1), we examined the impact that OpenCache has on the content delivery process, and noted that it had negligible impact on the user's experience. In fact, OpenCache improved the situation in all circumstances, particularly when the content was available from the cache. When this occurred, content could be delivered directly to the user and network efficiency increased significantly.

This first evaluation was also vital in ascertaining the effectiveness of Software Define Networking in the context of content delivery, and specifically, as a tool to redirect requests for content. By completing this evaluation, we observed some performance limitations in the current generations of switch hardware supporting OpenFlow. This is likely to be overcome as technology evolves and support for these technologies becomes the norm.

The second evaluation (presented in Section 5.2) was used to further examine OpenCache's behaviour. To highlight any potential impacts, we identified the effect that OpenCache had across an extensive set of recognised Quality-of-Experience metrics. Through this experimentation, we showed again that OpenCache actually improves the overall experience. To further the results of the initial evaluation we scaled our experimentation in a way that encompassed multiple simultaneous clients and their respective requests. This evaluation was also used to demonstrate the distributed nature of OpenCache, with constituent elements placed in a number of different locations.

This evaluation also highlighted complexity introduced in the process of caching when adaptive streaming technology is used. Without sufficient knowledge of quality levels and equivalences in content, cache object duplication can occur. Although this has no impact over the worst-case scenario, it can lead to wastage in storage capacity. Furthermore, it can prevent content from being stored and delivered from the cache, as would otherwise be the case in a non-adaptive scenario.

In our final evaluation (described Section 5.3), we examine the suitability of the OpenCache API. In particular, we looked to demonstrate the flexibility and

power offered by the interface. To do this we built two applications that utilise the API with their own custom logic to provide different resiliency features: one pro-active and one reactive.

This experimentation demonstrated that the API provided sufficient levels of control to achieve advanced levels of functionality. This includes unprecedented levels of configurability and control, highlighted through tight coordination between multiple OpenCache elements and the platform's interaction with the underlying infrastructure provision.

In this case, application-layer runtime information was combined with direct control over network behaviour, and used to remedy failure and excessive load. These changes can be applied near instantaneously with minimal impact on the client's experience. This evaluation highlights the potential benefits as we move towards the full softwarisation of both the network and its constituent services.

Together, these evaluations demonstrate the feasibility of moving content delivery functionality entirely into software. Not only does this meet core functionality requirements, it has also highlighted additional benefits. Chief amongst these is the flexibility and agility of infrastructure-assisted applications; the capability to not only efficiently use resources, but adapt their provision as such. This is an important step towards understanding the future of content delivery networks.

Chapter 6

Conclusions

In this thesis, we acknowledge the importance of the Internet, and its pervasiveness in many peoples lives. As usage patterns have changed over time, the Internet has become the predominant method used to deliver video to a world-wide user-base. To meet this change in demand, the ways in which content is delivered have also evolved.

This thesis examines how the latest wave of technologies may influence the future of this process. These advances can be found in the network, where new paradigms and products are allowing unprecedented control over networking equipment. This work investigates the usage of novel technologies, not just as a replacement for conventional functionality, but also as a tool to create behaviour not often realised *within* the network.

Developments have also been made in the field of virtualisation: the availability of platforms capable of effectively scaling resource allocation in a dynamic manner have become widely and freely available. This has enabled a new generation of flexible software functions to be created. These replace existing network functions and offer a realistic alternative to the hardware variants that already exist in today's networks.

Content delivery can be considered as one of these functions, and is an important example of such. The process of softwarisation creates the possibility of replacing large-scale content delivery networks with flexible alternatives that can adapt to both resource availability and consumer demand. They can also share their infrastructure with other functions, releasing or increasing their own reservation when necessary.

Advances in content delivery techniques have also made video playback adaptable; clients now automatically adjust dependent on both their own capabilities and those of the connected network. As mobile devices become more prevalent, the ability to modify quality dependent on throughput has never been more important.

6.1 Thesis Contributions

This thesis lays down a specific set of motivations and aims that can be used to influence the design and implementation of next-generation content delivery platforms. These are based on an understanding of existing designs, as well as a forward-facing look towards the potential benefits that the utilisation of new technologies can bring to this area.

Taking these considerations forward, we provide a comprehensive design of such a platform. This is segmented into a number of layers, each of which is responsible for achieving a specific set of functions or behaviours. Importantly, it offers a logical separation of the caching functionality from the control plane, with each existing in a different layer of the architecture. This allows the behaviour of a number of content caches to be determined from a central location, thus providing the desired cache programmability.

Building on this design, we implement a prototype content delivery platform, built specifically to explore and evaluate a subset of the design requirements laid out in the earlier sections. This is released free and open-source[1] for others to examine and use.

The prototype is used in the first instance to understand and examine the feasibility of utilising software defined networking in the process of content delivery. Through this work, we have determined that this technology can be used to effectively redirect requests for content towards a particular content cache.

Although this proof of concept showed that this functionality was possible with existing technologies, we encountered a number of performance degradations that would ultimately prevent its use in existing production environments. However, these have been attributed to the relative immaturity of current implementations, and future iterations will likely not impose any such limitations.

[1] `https://github.com/opencache-project`

This evaluation, as with the others presented in this thesis, utilised a large-scale experimental testbed spanning a number of countries within Europe. From this topology, we created a realistic environment in which the design (through use of the resultant prototype) could be evaluated using realistic traffic traversing real networks.

We continued this evaluation by scaling up the experimental environment and using this to measure the impact of caching on a number of recognised quality of experience metrics. This is an important process considering that content distribution networks are now deployed, at least partially, to ensure that these measures are maintained at a reasonable level.

Through this evaluation, a number of issues were identified that impact the efficiency of caching content in scenarios where adaptive content delivery techniques are used. This includes potential cache duplication, and also the increased probability of cache-misses. This phenomenon is even present when clients are connected using identical links and in the same network segment; the interaction of cache-hits in this process also complicates the matter further.

Clearly, due consideration needs to be taken to overcome these issues and ensure that experience can be maintained at a satisfactory level. The design of this platform provides the necessary tools to achieve this, without dictating the method and logic by which it is to be achieved. These choices are left to the operator or application to decide on which best fits their own content, customers and infrastructure.

Another contribution of this thesis is the realisation of a novel control layer, which allows a distributed content delivery platform to be controlled and maintained as-per user requirements. This functionality is achieved through the implementation of a well documented and fine-grained API, which gives the owner full control over the content delivery nodes connected in their deployment.

This functionality allows a new type of application to be built semantically *on top* of a content delivery platform, and facilitates unprecedented levels of flexibility and scalability. Through the use of this interface, an application can dynamically alter the behaviour, content and provision of a set of distributed caches using whatever logic that the operator requires. This might include information derived from internal systems or analytics, that would otherwise be difficult to integrate into existing content delivery decisions. This control includes the fetching of content from remote locations prior to it becoming popular, potentially alle-

viating the demand otherwise placed on the infrastructure when many hundreds or even thousands of users begin to request the content.

This API also explores novel functionalities in content delivery, particularly the use of external compute platforms to provision new content delivery nodes in response to current or even expected demand, enabling new cache nodes to be created in response to load, network conditions or cost constraints. Given the drive behind the virtualisation of network functions, it was imperative that we explored the use of cutting-edge technologies to ascertain their suitability to host and deploy the required functionality; in our case, a content delivery platform. Through this work, we have demonstrated both the advantages and potential pitfalls in doing so.

In the final evaluation, we demonstrated the suitability and flexibility of the API by implementing a number of behaviours. These would usually require the deployment of dedicated hardware middleboxes, but is instead realised using the commodity switching equipment already located within the network. This is only possible with the tight integration of compute and network resources, which by working together, can provide a seamless transition of services between two different cache nodes; whether this be due to excessive load or in response to hardware failure.

Together, these evaluations exercise a number of features, both fundamental and novel, and explore the form of future content delivery platforms. They also raise a number of limitations and drawbacks of using some of these cutting edge-technologies, which allows us to better understand the steps necessary for deployment into the real world.

6.1.1 Commercial and Research Impacts

This design also creates the potential for new business models to be realised. As well as utilising disruptive technologies in the network and computing space, a flexible and responsive software-based design also creates new opportunities to offer content distribution as a commodity service.

As the design relies on the use of off-the-shelf hardware rather than a proprietary platform, it removes a significant barrier to entry and allows competing companies to offer alternative services that may even be located on the same physical substrate. This should force business to innovate in the technical solu-

tion they offer, rather than a dependency on the infrastructure (and its location) that they currently rely upon.

The design also enables cache deployments in different locations, especially in places where cache are not traditionally present. It creates new opportunities for operators of all different sizes to host their own cache. They can then auction and/or sell the capacity contained within this deployment to interested parties. Clients could include existing content distribution networks looking to store content in new, previously inaccessible, locations, or content creators looking for a cost effective way to distribute their own content even closer to the user.

The monetisation of independently-hosted cache resources should help to subsidise the acquisition and running of the necessary equipment, as well as contributing towards costs incurred by both the network operator and client in order to deliver the content.

The thesis also contributes a platform that can be used by researchers and academics to develop novel and interesting techniques for delivering content. Rather than having to develop and build the underlying cache implementation, this work provides a common solution in which a researcher can build and modify the behaviour of a cache without the burden of extensive development. This enables new areas to be explored around content placement, replacement and delivery techniques in an age where content is being consumed in ever greater amounts.

6.1.2 Summary

In summary, this thesis has made the following contributions:

- Proposed a set of guidelines and desired features to be used in the creation of future content delivery platforms.

- Realised the aforementioned design goals by building a free and open-source prototype implementation. This encompasses many of these features, and serves as a platform for experimentation.

- Conducted extensive evaluation using this prototype. These realistic exercises were possible through the use of multiple international testbed facilities.

- Highlighted the need for a standardised and open way of interacting with content delivery networks. This includes an exploratory implementation which was demonstrated as part of the aforementioned evaluation.

Throughout this process, we have had to overcome a number of challenges. The lessons learnt include:

- Identification of deficiencies in the performance of current *software-defined* tools and technologies, which are still relatively immature.

- Unexpected interactions when adaptive streaming technologies are used; these are likely to have substantial impacts as the technology sees greater adoption.

The capability to not only work alongside these technologies, but exploit their inherent behaviour to the advantage of the content delivery network, is vital if content consumption continues to increase.

6.2 Future Work

In this thesis, we presented a comprehensive design for a future content delivery platform. We then realised this through a prototype implementation, which explored a number of the proposed features. Further development continues on OpenCache, which serves as a platform to identify, implement and evaluate new techniques for content delivery.

An important goal for OpenCache is the fostering of a community of developers and users. Progress in this respect is ongoing due to the recency of the work. In the future, this community will be vital in ensuring transparency and oversight in OpenCache. It will also serve as a platform to share additions and modifications that feedback directly into future iterations of the platform.

Future work also includes the proposition of the OpenCache API as a candidate for standardising the method through which applications can communicate with (and subsequently control) CDNs. This can be achieved without the need to understand the specific details of the underlying resources or how to address its components.

There is also the opportunity to expand this API, including the provision of access control and tailored views. This allows different users and applications to not only have different representations of an OpenCache deployment, but also access to a different set of functions consummate to their role in the organisation or purpose as an application. This is a key step towards achieving the collaborative approach to content distribution outlined in the OpenCache design.

An important facet of using software defined networking technology is the ability to go beyond its use as a redirection technique. We explored this in the evaluation by implementing load-balancing and traffic steering behaviour. However, this can be taken even further. For example, replication could also be introduced, with little or no requirement to change the underlying code-base to support this. Instead, this functionality can be realised using a combination of flow modifications, and replicating the requests amongst a collection of cache nodes.

In order to give users and applications even more control over the caches in their deployment, it would be interesting to facilitate finer granularity over what content is stored on the cache. Although out of scope for this initial implementation, possibilities include determining the cache replacement policy on a per-object level. As this requires almost instantaneous decisions, it may be appropriate to have a compilable decision list or set of logic, which is then pushed to the cache node. As this is performance sensitive, this would avoid the need for a node to request information from the controller before making a decision on whether or not a content object should remain in the cache or be evicted.

There is also a clear need to investigate how resources of all types are described, provisioned and shared. Although OpenFlow makes progress towards achieving this in the field of networking, it is clearly not designed, nor intended to, consider all of these aspects in detail. However, there are wider concerns in other areas of softwarisation. For example, the OpenCache prototype is built around a proprietary interface necessary to provision resources in a compute environment. Although there is a clear trend towards the format and style of these interfaces, harmonising these and ascertaining consensus amongst interested parties would clearly be beneficial to all interested parties.

Further research directions include the identification and realisation of additional services and platforms that can be brought into software. This has been attempted in some areas, but many aspects have gone unexplored as of yet. This

process involves not only replicating basic functionality in software, but also exploring how programmable infrastructure can by exploited to go beyond existing service provision and delivery.

Other directions include an investigation into how the commoditisation of compute and network resources impacts business models and user communities. This is particularly interesting in light of current trends towards the use of homogenised hardware and networking platforms, which remove some of the market leverage that many of the current organisations and business hold. This is likely to be furthered by the widespread availability of flexible and configurable hardware platforms. These will be capable of operating as either compute or networking elements, without performance limitations, and seamlessly transitioning between each function.

With the flexibility offered by programmable platforms, there is a necessity to consider how the applications can best use the dynamic environment. In this case, resources are fluid, and functions can be migrated and moved without user intervention. In such an environment, it is important to define the requirements of a function (or set of such). For example, in the case of content delivery networks, it is vital to specify the performance and latency requirements, which are key to the success of the platform. Currently, there is no standard way of achieving such.

These requirements may also need to be matched and resolved against grander goals, such as reduced operating cost or more sustainable provisioning. Considering that all of these metrics can change over time, there are unsolved challenges around the design and implementation of such an orchestration platform. This includes designing a common the method for gathering information from all the different potential inputs, and furthers the desire for greater coordination in such environments.

6.3 Concluding Remarks

In this thesis we have provided contributions towards the future of content delivery design. This includes a framework in which future deployments can be built. We have shown through extensive evaluation the benefits of some of these design decisions. There is a clear advantage to utilising next-generation technolo-

gies, and ensuring that they are integrated into platform designs will be key to realising increased efficiency, control and programmability.

Many of these functionalities rely somewhat on software defined networking, and in the case of this thesis, OpenFlow technology. However, there are no guarantees that this technology will see continued adoption, especially in production environments. Although early signs are encouraging, the future in this respect is unclear.

Nonetheless, there is no reason why the techniques demonstrated in this thesis cannot be replicated using other technologies labelled under the same software defined umbrella. Although the thesis does not consider these, the modular nature of the implementation makes their integration and evaluation a relatively simple process. Mirroring this, work in the network controller space [23] shows a concerted effort towards the simultaneous support of technologies other than OpenFlow.

Before this technology ever sees production use, there are still a number of challenges to be met. One of OpenFlow's major advantages is that the network controller can work with networking hardware from multiple vendors. Yet recent developments are threatening this situation, with a trend towards vendor-specific extensions. Currently, the ability to configure switches, especially to create tunnels and bridges, requires knowledge of the underlying hardware and the capabilities it supports. There is no common way to interact with them at the moment, and there is a real risk that this will segment the market and negate the interoperability that has so far been the key to success.

Although many of the virtualised compute platforms are now used in large-scale commercial deployments, the same cannot be said for their network counterparts. This is likely in part due to the scale and performance of the software controllers, which at the moment lack the robustness required in a production environment. There are currently efforts which are showing genuine progress towards solving this problem [62].

Bibliography

[1] About Python. http://www.python.org/about. Accessed: 16/10/2015.

[2] Akamai Technologies. http://www.akamai.com/. Accessed: 18/06/2015.

[3] Apple HTTP Live Streaming. https://developer.apple.com/streaming/. Accessed: 18/06/2015.

[4] Ceph: a distributed object store and file system designed to provide excellent performance, reliability and scalability. http://ceph.com/. Accessed: 16/10/2015.

[5] ConfigParser Configuration file parser. http://docs.python.org/2/library/configparser.html. Accessed: 16/10/2015.

[6] CoreOS is Linux for Massive Server Deployments. http://coreos.com/. Accessed: 21/10/2015.

[7] Debian: The Universal Operating System. http://www.debian.org/. Accessed: 15/10/2015.

[8] Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/. Accessed: 21/10/2015.

[9] ETSI Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. http://portal.etsi.org/NFV/NFV_White_Paper.pdf. Accessed: 29/09/2015.

[10] Fed4FIRE: Federation for Future Internet Research and Experimentation. http://www.fed4fire.eu/. Accessed: 16/10/2015.

[11] Floodlight OpenFlow Controller: Open Source Software for Building Software-defined Networks. http://www.projectfloodlight.org/floodlight/. Accessed: 21/10/2015.

[12] GÉANT Project GN3plus Open Call: Technical Annex B - GÉANT OpenFlow Facility. http://geant3.archive.geant.net/opencalls/Overview/Documents/Open%20Call%20Technical%20Annex%20B%20GEANT%20Openflow%20Testbed%20Facility%20FINAL.pdf. Accessed: 16/10/2015.

[13] Gluster: Storage for your Cloud. http://www.gluster.org/. Accessed: 16/10/2015.

[14] Limelight Networks. http://www.limelight.com/. Accessed: 18/06/2015.

[15] logging - Logging facility for Python. http://docs.python.org/2/library/logging.html. Accessed: 16/10/2015.

[16] Microsoft Smooth Streaming. http://www.iis.net/downloads/microsoft/smooth-streaming. Accessed: 18/06/2015.

[17] Mininet - An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/. Accessed: 21/10/2015.

[18] MongoDB: Launch your GIANT idea. http://www.mongodb.org/. Accessed: 16/10/2015.

[19] Netflix Open Connect. http://openconnect.netflix.com/. Accessed: 18/06/2015.

[20] OFELIA Control Framework (OCF): a set of software tools for testbed management. http://github.com/fp7-ofelia/ocf. Accessed: 15/10/2015.

[21] Open Networking Foundation. https://www.opennetworking.org/. Accessed: 19/06/2015.

[22] Open vSwitch: Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/. Accessed: 19/06/2015.

[23] OpenDaylight Consortium. http://www.opendaylight.org/. Accessed: 19/06/2015.

[24] OpenFlow in Europe: Linking Infrastructure and Applications. http://www.fp7-ofelia.eu/. Accessed: 15/10/2015.

[25] OpenFlow Switch Specification: Version 1.0.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf. Accessed: 19/06/2015.

[26] OpenFlow Switch Specification: Version 1.1.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf. Accessed: 19/06/2015.

[27] OpenFlow Switch Specification: Version 1.2. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf. Accessed: 19/06/2015.

[28] OpenFlow Switch Specification: Version 1.3.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf. Accessed: 19/06/2015.

[29] OpenFlow Switch Specification: Version 1.4.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed: 19/06/2015.

[30] OpenFlow Switch Specification: Version 1.5.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf. Accessed: 19/06/2015.

[31] OpenStack: Open source software for creating private and public clouds. http://www.openstack.org/. Accessed: 21/06/2015.

[32] OPNFV: An Open Platform for Accelerating NFV. http://www.opnfv.org/sites/opnfv/files/pages/files/opnfv_whitepaper_103014.pdf. Accessed: 29/09/2015.

[33] Peach Open Movie Project Big Buck Bunny. http://peach.blender.org/. Accessed: 16/10/2015.

[34] pip - The PyPA recommended tool for installing Python packages. https://pypi.python.org/pypi/pip. Accessed: 21/10/2015.

[35] Project CCNx: Content-Centric Networking CCNx Reference Implementation. http://github.com/ProjectCCNx/ccnx. Accessed: 10/06/2015.

[36] PyPI - the Python Package Index. https://pypi.python.org/pypi. Accessed: 21/10/2015.

[37] Regular Expression Syntax. http://docs.python.org/2/library/re.html#regular-expression-syntax. Accessed: 16/10/2015.

[38] Ryu: Component-based Software Defined Networking Framework. https://osrg.github.io/ryu/. Accessed: 19/06/2015.

[39] SimpleHTTPServer - Simple HTTP request handler. http://docs.python.org/2/library/simplehttpserver.html. Accessed: 16/10/2015.

[40] Static Flow Pusher API (New). http://floodlight.atlassian.net/wiki/display/floodlightcontroller/Static+Flow+Pusher+API+(New). Accessed: 21/10/2015.

[41] Using etcd with CoreOS. https://coreos.com/etcd/. Accessed: 21/10/2015.

[42] Vagrant - Development environments made easy. https://www.vagrantup.com/. Accessed: 21/10/2015.

[43] Varnish Cache. https://www.varnish-cache.org/. Accessed: 04/10/2015.

[44] VideoLAN VLC Media Player. http://www.videolan.org/vlc/. Accessed: 16/10/2015.

[45] Xen Project Hypervisor. http://www.xenproject.org/. Accessed: 15/10/2015.

[46] Cisco VNI: Global mobile data traffic forecast update, 2010-2015. *White Paper, February*, 2011.

[47] ETSI GS NFV 001 Network Functions Virtualization (NFV); Use Cases, 2013.

[48] ETSI GS NFV 004 Network Functions Virtualization (NFV); Virtualization Requirements, 2013.

[49] Open Networking Foundation North Bound Interface Working Group (NBI-WG) Charter, 2013.

[50] V. Adhikari, Y. Guo, F. Hao, V. Hilt, Z.-L. Zhang, M. Varvello, and M. Steiner. Measurement Study of Netflix, Hulu, and a Tale of Three CDNs. *Networking, IEEE/ACM Transactions on*, PP(99):1–1, 2014.

[51] V. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netflix: Understanding and improving multi-CDN movie delivery. In *INFOCOM, 2012 Proceedings IEEE*, pages 1620–1628, March 2012.

[52] V. Adhikari, S. Jain, Y. Chen, and Z.-L. Zhang. Vivisecting YouTube: An active measurement study. In *INFOCOM, 2012 Proceedings IEEE*, pages 2521–2525, March 2012.

[53] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Comparing DNS Resolvers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 15–21, New York, NY, USA, 2010. ACM.

[54] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Web Content Cartography. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 585–600, New York, NY, USA, 2011. ACM.

[55] V. Aggarwal, A. Feldmann, C. Scheideler, and M. Faloutsos. Can ISPs and P2P users cooperate for improved performance. *ACM SIGCOMM Computer Communication Review*, 37:29–40, 2007.

[56] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26–36, 2012.

[57] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-peer Content Distribution Technologies. *ACM Comput. Surv.*, 36(4):335–371, December 2004.

[58] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena, and P. R. Rodriguez. Is High-quality VoD Feasible Using P2P Swarming? In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 903–912, New York, NY, USA, 2007. ACM.

[59] M. Arlitt, R. Friedrich, and T. Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. *Perform. Eval.*, 39(1-4):149–164, February 2000.

[60] S. Astaneh and S. Heydari. Multi-failure restoration with minimal flow operations in software defined networks. In *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, pages 263–266, March 2015.

[61] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[62] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[63] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, RFC Editor, May 1996. `http://www.rfc-editor.org/rfc/rfc1945.txt`.

[64] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[65] B. Boughzala, R. Ben Ali, M. Lemay, Y. Lemieux, and O. Cherkaoui. OpenFlow supporting inter-domain virtual machine migration. In *Wireless and*

Optical Communications Networks (WOCN), 2011 Eighth International Conference on, pages 1–7. IEEE, 2011.

[66] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[67] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea. Implementing dynamic chaining of Virtual Network Functions in OpenStack platform. In *Transparent Optical Networks (ICTON), 2015 17th International Conference on*, pages 1–4, July 2015.

[68] M. Canini, D. Kostic, J. Rexford, and D. Venzano. Automating the Testing of OpenFlow Applications. In *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering (WRiPE)*, 2011.

[69] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, et al. A NICE Way to Test OpenFlow Applications. In *NSDI*, volume 12, pages 127–140, 2012.

[70] M. Carbone and L. Rizzo. Dummynet Revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, April 2010.

[71] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker. ETHANE: Taking Control of the Enterprise. In *SIGCOMM Computer Comm. Rev*, 2007.

[72] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin. Simple Network Management Protocol (SNMP). STD 15, RFC Editor, May 1990. `http://www.rfc-editor.org/rfc/rfc1157.txt`.

[73] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. Technical report, DTIC Document, 1995.

[74] X. Chen, Q. Fan, and H. Yin. Caching in Information-Centric Networking: From a content delivery path perspective. In *Innovations in Information*

Technology (IIT), 2013 9th International Conference on, pages 48–53, March 2013.

[75] B. Cheng, X. Liu, Z. Zhang, and H. Jin. A Measurement Study of a Peer-to-Peer Video-on-Demand System. In *IPTPS*, 2007.

[76] J. Choi, A. Reaz, and B. Mukherjee. A Survey of User Behavior in VoD Service and Bandwidth-Saving Multicast Streaming Schemes. *Communications Surveys Tutorials, IEEE*, 14(1):156–169, First 2012.

[77] E. Cohen, B. Krishnamurthy, and J. Rexford. Evaluating Server-Assisted Cache Replacement in the Web. In *Proceedings of the 6th European Symposium on Algorithms*, pages 307–319. Springer-Verlag, 1998.

[78] C. P. Costa, I. S. Cunha, A. Borges, C. V. Ramos, M. M. Rocha, J. M. Almeida, and B. Ribeiro-Neto. Analyzing Client Interactivity in Streaming Media. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 534–543, New York, NY, USA, 2004. ACM.

[79] J. Costa-Requena, M. Kimmerlin, J. Manner, and R. Kantola. SDN optimized caching in LTE mobile networks. In *Information and Communication Technology Convergence (ICTC), 2014 International Conference on*, pages 128–132, Oct 2014.

[80] L. D'Acunto, M. Meulpolder, R. Rahman, J. Pouwelse, and H. Sips. Modeling and analyzing the effects of firewalls and NATs in P2P swarming systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[81] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proc. of ACM Multimedia*, pages 15–23, 1994.

[82] S. Das, Y. Yiakoumis, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and P. Desai. Application-aware aggregation and traffic engineering in a converged packet-circuit network. In *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*, pages 1–3, March 2011.

[83] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[84] T. Do, K. Hua, and M. Tantaoui. P2VoD: providing fault tolerant video-on-demand streaming in peer-to-peer environment. In *Communications, 2004 IEEE International Conference on*, volume 3, pages 1467–1472 Vol.3, June 2004.

[85] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 362–373, New York, NY, USA, 2011. ACM.

[86] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810, RFC Editor, March 2010. `http://www.rfc-editor.org/rfc/rfc5810.txt`.

[87] D. Eager, M. Vernon, and J. Zahorjan. Optimal and Efficient Merging Schedules for Video-on-Demand Servers. In *Proc. ACM Multimedia*, pages 199–202, 1999.

[88] S. Egger, T. Hossfeld, R. Schatz, and M. Fiedler. Waiting times in quality of experience for web based services. In *Quality of Multimedia Experience (QoMEX), 2012 Fourth International Workshop on*, pages 86–96, July 2012.

[89] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). RFC 6241, RFC Editor, June 2011. `http://www.rfc-editor.org/rfc/rfc6241.txt`.

[90] D. Erickson. The Beacon Openflow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[91] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 327–338. ACM, 2013.

[92] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999. `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[93] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for Software-defined Networks. *Communications Magazine, IEEE*, 51(2):128–134, February 2013.

[94] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[95] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 27–38. ACM, 2014.

[96] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube Traffic Characterization: A View From the Edge, IMC. In *In: Proc. of IMC*, 2007.

[97] L. Guo, S. Chen, Z. Xiao, and X. Zhang. Analysis of Multimedia Workloads with Implications for Internet Streaming. In *Proc. of WWW*, 2005.

[98] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 551–562. ACM, 2014.

[99] G. Haílinger and F. Hartleb. Content Delivery and Caching from a Network Provider's Perspective. *Comput. Netw.*, 55(18):3991–4006, December 2011.

[100] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, Feb 2015.

[101] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow. *ACM SIGCOMM Demo*, 4(5):6, 2009.

[102] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross. A Measurement Study of a Large-Scale P2P IPTV System. *Multimedia, IEEE Transactions on*, 9(8):1672–1687, Dec 2007.

[103] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, volume 10, pages 249–264, 2010.

[104] A. Hu. Video-on-Demand Broadcasting Protocols: A Comprehensive Study. In *Proceedings of IEEE INFOCOM*, pages 508–517, 2001.

[105] K. A. Hua, Y. Cai, and S. Sheu. Patching: A Multicast Technique for True Video-on-demand Services. In *Proceedings of the Sixth ACM International Conference on Multimedia*, MULTIMEDIA '98, pages 191–200, New York, NY, USA, 1998. ACM.

[106] C. Huang, J. Li, and K. W. Ross. Peer-Assisted VoD: Making Internet Video Distribution Cheap. In *Peer-to-Peer Systems*, 2007.

[107] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[108] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association.

[109] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[110] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013.

[111] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia. SDN-Based Application-Aware Networking on the Example of YouTube Video Streaming. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 87–92, Oct 2013.

[112] J. Kangasharju, K. W. Ross, and J. W. Roberts. Performance Evaluation of Redirection Schemes in Content Distribution Networks. In *Computer Communications*, pages 207–214, 2000.

[113] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet Service Providers Fear Peer-assisted Content Distribution? In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[114] K. Katsalis, V. Sourlas, T. Korakis, and L. Tassiulas. A cloud-based content replication framework over multi-domain environments. In *Communications (ICC), 2014 IEEE International Conference on*, pages 2926–2931, June 2014.

[115] M. Koerner and O. Kao. Multiple service load-balancing with OpenFlow. In *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*, pages 210–214. IEEE, 2012.

[116] B. Krishnamurthy, C. Wills, and Y. Zhang. On the Use and Performance of Content Distribution Networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 169–182, New York, NY, USA, 2001. ACM.

[117] S. S. Krishnan and R. K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 211–224, New York, NY, USA, 2012. ACM.

[118] S. Lederer, C. Mueller, B. Rainer, C. Timmerer, and H. Hellwagner. An experimental analysis of Dynamic Adaptive Streaming over HTTP in Content Centric Networks. In *Multimedia and Expo (ICME), 2013 IEEE International Conference on*, pages 1–6, July 2013.

[119] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A Case for a Coordinated Internet Video Control Plane. In *Proceedings of*

the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, pages 359–370, New York, NY, USA, 2012. ACM.

[120] Z. Liu, Y. Shen, K. Ross, S. Panwar, and Y. Wang. LayerP2P: Using Layered Video Chunks in P2P Live Streaming. *Multimedia, IEEE Transactions on*, 11(7):1340–1352, Nov 2009.

[121] H. Long, Y. Shen, M. Guo, and F. Tang. LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 290–297. IEEE, 2013.

[122] M. Luizelli, L. Bays, L. Buriol, M. Barcellos, and L. Gaspary. Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 98–106, May 2015.

[123] Q. Lv, S. Ratnasamy, and S. Shenker. Can Heterogeneity Make Gnutella Scalable? In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, pages 94–103, 2002.

[124] N. Magharei and R. Rejaie. PRIME: Peer-to-Peer Receiver-drIven MEsh-Based Streaming. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1415–1423, May 2007.

[125] N. Magharei, R. Rejaie, and Y. Guo. Mesh or Multiple-Tree: A Comparative Study of Live P2P Streaming Approaches. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 1424–1432, May 2007.

[126] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 90–102, New York, NY, USA, 2009. ACM.

[127] V. Mann, A. Vishnoi, K. Kannan, and S. Kalyanaraman. CrossRoads: Seamless VM mobility across data centers through software defined network-

ing. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 88–96. IEEE, 2012.

[128] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, April 2014. USENIX Association.

[129] F. Mattos, D. Menezez, and O. C. Muniz Bandeira Duarte. XenFlow: Seamless migration primitive and quality of service for virtual networks. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 2326–2331. IEEE, 2014.

[130] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[131] S. Mehraghdam, M. Keller, and H. Karl. Specifying and placing chains of virtual network functions. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 7–13, Oct 2014.

[132] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy. Design and evaluation of algorithms for mapping and scheduling of virtual network functions. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–9, April 2015.

[133] H. Moens and F. De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 418–423, Nov 2014.

[134] J. T. Moore and S. M. Nettles. Towards Practical Programmable Packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001*, pages 41–50, 2001.

[135] H. Nam, D. Calin, and H. Schulzrinne. Intelligent content delivery over wireless via SDN. In *Wireless Communications and Networking Conference (WCNC), 2015 IEEE*, pages 2185–2190, March 2015.

[136] C. Papagianni, A. Leivadeas, and S. Papavassiliou. A Cloud-Oriented Content Delivery Network Paradigm: Modeling and Assessment. *Dependable and Secure Computing, IEEE Transactions on*, 10(5):287–300, Sept 2013.

[137] M. Pathan and R. Buyya. A Taxonomy of CDNs. In R. Buyya, M. Pathan, and A. Vakali, editors, *Content Delivery Networks*, volume 9 of *Lecture Notes Electrical Engineering*, pages 33–77. Springer Berlin Heidelberg, 2008.

[138] P. Perešíni and M. Canini. Is Your OpenFlow Application Correct? In *Proceedings of The ACM CoNEXT Student Workshop*, CoNEXT '11 Student, pages 18:1–18:2, New York, NY, USA, 2011. ACM.

[139] D. Perino and M. Varvello. A reality check for content centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pages 44–49. ACM, 2011.

[140] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[141] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving Content Delivery Using Provider-aided Distance Information. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 22–34, New York, NY, USA, 2010. ACM.

[142] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.

[143] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir. Application-awareness in SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 487–488, New York, NY, USA, 2013. ACM.

[144] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN Programming with Pyretic. *USENIX ;login*, 38(5):128–134, Oct. 2013.

[145] A. Ronacher. Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions. `http://flask.pocoo.org/`. Accessed: 21/10/2015.

[146] A. Ronacher. Jinja2 is a modern and designer-friendly templating language for Python, modelled after Djangos templates. `http://jinja.pocoo.org/docs/dev/`. Accessed: 21/10/2015.

[147] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting Routing Control Platforms with the Eyes and Muscles of Software-defined Networking. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 13–18, New York, NY, USA, 2012. ACM.

[148] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri. Information centric networking over SDN and OpenFlow: Architectural aspects and experiments on the OFELIA testbed. *Computer Networks*, 57(16):3207 – 3221, 2013. Information Centric Networking.

[149] S. Salsano, P. Ventre, L. Prete, G. Siracusano, M. Gerola, and E. Salvadori. OSHI - Open Source Hybrid IP/SDN Networking (and its Emulation on Mininet and on Distributed SDN Testbeds). In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 13–18, Sept 2014.

[150] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. STD 64, RFC Editor, July 2003. `http://www.rfc-editor.org/rfc/rfc3550.txt`.

[151] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, RFC Editor, April 1998. `http://www.rfc-editor.org/rfc/rfc2326.txt`.

[152] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1801–1810 vol.3, 2001.

[153] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester. Enabling fast failure recovery in OpenFlow networks. In *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the*, pages 164–171, Oct 2011.

[154] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[155] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. *Internet Society NDSS*, 2013.

[156] S. Shin and G. Gu. CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–6. IEEE, 2012.

[157] S. Shirali-Shahreza and Y. Ganjali. Flexam: Flexible sampling extension for monitoring and security applications in openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 167–168. ACM, 2013.

[158] S. Sivasubramanian, D. R. Richardson, C. L. Scofield, and B. E. Marshall. Request routing using network computing components, April 12 2011. US Patent 7,925,782.

[159] I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming over the Internet. *IEEE MultiMedia*, (4):62–67, 2011.

[160] D. Starobinski and D. Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(23):125 – 137, 2001. Advanced Performance Modeling.

[161] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM'01*, pages 149–160, 2001.

[162] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting Behind Akamai: Inferring Network Conditions Based on CDN Redirections. *Networking, IEEE/ACM Transactions on*, 17(6):1752–1765, Dec 2009.

[163] G. Szabo and B. A. Huberman. Predicting the Popularity of Online Content. *Commun. ACM*, 53(8):80–88, August 2010.

[164] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26:5–18, 1996.

[165] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, RFC Editor, November 2000. `http://www.rfc-editor.org/rfc/rfc2991.txt`.

[166] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. Munafo, and S. Rao. Dissecting Video Server Selection Strategies in the YouTube CDN. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 248–257, June 2011.

[167] C. VNI. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018. 2014.

[168] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages*, pages 235–249. Springer, 2011.

[169] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[170] Z. Wang, T. Tsou, J. Huang, X. Shi, and X. Yin. Analysis of Comparisons between OpenFlow and ForCES. Internet-Draft draft-wang-forces-compare-openflow-forces-01, IETF Secretariat, March 2012. `http://www.ietf.org/internet-drafts/draft-wang-forces-compare-openflow-forces-01.txt`.

[171] D. Wessels and K. Claffy. ICP and the Squid web cache. *Selected Areas in Communications, IEEE Journal on*, 16(3):345–357, Apr 1998.

[172] M. Wichtlhuber, R. Reinecke, and D. Hausheer. An SDN-Based CDN/ISP Collaboration Architecture for Managing High-Volume Flows. *Network and Service Management, IEEE Transactions on*, 12(1):48–60, March 2015.

[173] J. Zander and R. Forchheimer. The SOFTNET project: a retrospect. In *Electrotechnics, 1988. Conference Proceedings on Area Communication, EUROCON 88., 8th European Conference on*, pages 343–345, Jun 1988.

[174] Q. Zhang, S. Q. Zhang, J. Lin, H. Bannazadeh, and A. Leon-Garcia. Kaleidoscope: Real-time content delivery in software defined infrastructures. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 686–692, May 2015.