Juno: Reconfigurable Middleware for Heterogeneous Content Networking

Gareth Tyson[1], Andreas Mauthe[1], Thomas Plagemann[2] and Yehia El-khatib[1]

[1]Computing Department, InfoLab21
Lancaster University, UK.
[2]University of Oslo, Norway.
{g.tyson, andreas, yehia} @comp.lancs.ac.uk
plageman@ifi.uio.no

Abstract. Multimedia Content distribution is playing an increasingly prominent role in the Internet today, with a proliferation of diverse services and delivery mechanisms. Due to this increasing heterogeneity the management of next generation *content networks* is becoming increasingly complex. This paper presents Juno, a configurable component-based middleware designed to address the divergent nature of modern content networking. In Juno, functionality is separated into pluggable components that can be dynamically attached, detached and deployed, allowing the middleware to be specialised and adapted for different applications and environments. To demonstrate how functionality from (existing) content distribution networks can be realised through the middleware, an application operating over BitTorrent and Pastry has been developed using Juno. Through this, Juno is evaluated by looking at functional, non-functional and performance aspects of the framework.

Keywords: Content networking, middleware, content distribution, peer-to-peer, component-based engineering, content services.

1 Introduction

Multimedia content distribution is playing an increasingly prominent role in the Internet with a huge array of distribution mechanisms available for a diverse range of application areas. Early content distribution infrastructures [2] mainly focused on delivering stored content. However, as network and end system capabilities have increased there has also been an increasing demand for more diverse access mechanisms. Originally this focused on media streaming, but since then, systems have progressively begun to deliver more sophisticated applications such as video conferencing [9], video on demand (VoD) [19] and Internet television (IPTV) [27]. This propensity has seen an explosion in services and applications available under the umbrella term of *content networking* [22]. In contrast to traditional content

This work is supported by the Network of Excellence CONTENT (FP6-IST-038423)

distribution networks (CDN), content networks view the content itself as the focal point of the network.

This next generation of content distribution, however, creates a number of issues when both developing and deploying systems. These issues are primarily related to the heterogeneity observed in content networks. This heterogeneity can be separated into four areas:

a) *Delivery Heterogeneity* – characterised by a range of different delivery mechanisms employed, e.g. stored, live streamed, interactive, etc.
b) *Service Heterogeneity*- determined by the range of services available to improve the quality of experience, e.g. transcoding, content adaptation, replication, etc.
c) *Device Heterogeneity*– originating from the range of devices used to access the content, e.g. PCs, mobile phones, PDAs, etc.
d) *Network Heterogeneity* – reflected in the range of network capabilities available to different devices, e.g. ADSL, Ethernet, Bluetooth, WiFi.

When developing a distributed content-centric system it is therefore necessary to address these issues in order to provide the content network with the configurability required by real-world deployment. At present this is mainly dealt with by the application. However, we believe significant benefits can be gained by utilising middleware designed to handle these concerns. Traditional middleware lacks the required flexibility to manage this diversity as it is often restricted to dealing with limited aspects of functionality. A number of configurable middleware platforms [11][13][17][25] have been proposed but they do not address the specific issues relating to next generation content networking.

This paper introduces Juno, a configurable content networking middleware that addresses the heterogeneity of next generation content distribution. To achieve this, Juno promotes high levels of (re)configurability, allowing the middleware (and therefore the application) to be specialised and adapted to a variety of environments and constraints. In order to successfully provide a holistic architecture for content networking we believe it has to be: *configurable*, *adaptable*, *functionally scalable,* and *development oriented*. These properties are integral to providing effective support for content networking and therefore form the core principles of Juno.

Juno is designed in a component-based manner and has been implemented using the OpenCOM v1.4 [12] component model in Java. In order to demonstrate the feasibility of the approach we show how BitTorrent functionality can be implemented in Juno. Using an application developed over the middleware, the capabilities of Juno are then investigated; specifically *i)* by analysing how its (re)configurable approach deals with heterogeneity, *ii)* by examining the resource overhead associated with exploiting such (re)configurability, and *iii)* by assessing how its architectural design patterns can assist in the development and deployment of new applications.

The rest of the paper is organised as follows; section 2 offers an overview of related work in the field. Section 3 provides an overview of the Juno framework, using BitTorrent to highlight the development process. Section 4 subsequently provides an evaluation, using a prototype application developed in Juno. Lastly, section 5 concludes the paper outlining future work that is intended to be carried out.

2 Related Work

There has been a large body of work carried out in the field of content distribution, recently with a particular focus on P2P systems. Popular distribution paradigms include high bandwidth stored content delivery [4][5], live streaming [27], on-demand multicast [7][10], and video on-demand streaming (VoD) [19]. These systems are specialized to offer well-tuned services for the particular requirements endemic to those applications. Such systems, however, lack configurability as they are specifically designed to address issues endemic to those areas. This often makes them infeasible for deployment in diverse environments. Further their limited scope makes it impossible to adapt to variations in requirements and constraints.

Over recent years, content networking has also come to involve services such as content adaptation [21], transcoding [8] and replication [16]. These services augment the delivery in order to improve such things as performance and user experience. Traditionally these services have been operated in a client-server manner however research has also looked into hosting these in a P2P manner [14][18]. We believe this to be an important progression as the recent success of modular distributed systems (e.g. Web services) represents a significant trend in distributed computing.

There are a number of middlewares (e.g. JXTA[15]) that have been designed to offer convenient P2P abstractions for these systems alongside a number of development tools for implementing overlays [3][20]. These middlewares offer a platform over which P2P applications can be developed without the complexity of dealing with lower level issues. However, systems such as JXTA are built as a black box which limits configurability. This makes it hard to specialize or adapt a system for individual applications. Further, the low-level nature of these middlewares and toolkits mean that the construction of high-level systems such as content networks can become laborious.

To remedy the problems with existing approaches, a number of *configurable* and *reflective* middlewares have been developed [6][13][17]. These middlewares exploit architectural software patterns to provide a framework in which independent pluggable software components [12] can be attached. These middlewares, through reflection, can then inspect the operation of these components to select optimal architectural configurations. This allows a middleware to be specialized by attaching appropriate components, creating a bespoke platform for the application to operate over. Such middleware can then be dynamically reconstructed during runtime to adapt to changes in the environment. This occurs without direct intervention of the application. Instead, the application provides details of its requirements allowing the middleware to interpret and implement them. This removes a significant amount of complexity from the application without compromising such things as adaptability.

Research areas such as Grid computation [17], distributed objects [6] and multimedia QoS [11][25] have featured highly in configurable middleware development. However, little has been performed in the area of next generation content networking [22]. Unlike existing work, however, content networking embodies much higher level principles (e.g. the importance of user experience) alongside traditional low level aspects (e.g. QoS). This means that middleware for such an environment must be *cross-cutting.* We therefore believe considerable benefits can be gained from utilising configurable models. Through this approach, we believe it possible to address the complexities of developing, deploying and specialising different applications for their individual requirements and constraints.

3 The Juno Framework

The Juno Framework is a configurable middleware designed to address the divergent nature of next generation content networks. To achieve this it is therefore necessary to provide a configurable and extensible framework in which a diverse range of content related services and delivery mechanisms can be supported. The middleware consists of two aspects: pluggable functional components and utility support. The former constitutes the functionality of the middleware whilst the latter offers convenient support for these components (e.g. state management).

Juno has been implemented using the OpenCOM v1.4 [12] component model in Java. In order to illustrate how Juno realises content delivery functionality and demonstrate the feasibility of the underlying concept, it is shown how BitTorrent [5] can be implemented and extended in the Juno Framework.

3.1 Juno Overview

When an application is developed over Juno it provides Juno with the details of its requirements. Using this information Juno will then construct itself from the optimal components. Subsequently, it will also reconfigure itself dynamically to use different components as requirements and environmental factors change. This approach allows an application to operate over a bespoke middleware without the complexity of dealing with such issues as adaptation itself.

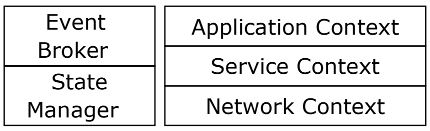

Fig. 1. Overview of Juno Framework

Juno is constructed from three layered *contexts*, shown in Fig. 1; each of these deals with a different type of heterogeneity: *network*, *service* and a*pplication*. These contexts are each built from components. This approach separates concerns and creates a well structured management framework.

The lowest layer is the *Network Context* which deals with the overlay aspects of the content network. This allows different overlays to be dynamically installed, adapted and managed to support more sophisticated functionality. Above the Network Context sits the *Service Context*. This context contains components that provide a variety of content distribution services operating over the lower overlays. These can range from delivery aspects such as chunk selectors, to services such as caching and transcoding. Lastly, above the Service Context, is the *Application Context* which offers a convenient interface for the application to interact with. Further to this, it also deals with combining multiple services for the ease of the application.

To support the three contexts, Juno also provides state and event management. Juno components do not maintain persistent state so to facilitate the easy reconfiguration of the middleware. This assists in open component introspection as well as allowing component to be easily removed without data loss.

3.2 Network Context

The Network Context forms a platform for more sophisticated distributed services to operate over. It consists of a set of components that interact through interfaces to manage and operate an overlay. These overlays are attached to Juno to provide the necessary distributed support for running higher level services and distribution paradigms. It consists of four primary components shown in Fig. 2. *Construction* deals with initiating, joining and leaving an overlay. *Maintenance* deals with monitoring and repairing the overlay. *Forward* deals with the routing of data in the overlay. Finally, *Transport* deals with transporting data between nodes.

Each component implements a defined interface that provides access to its capabilities. These interfaces can also be extended to be specialized for individual overlays. Furthermore, Juno's open and extensible nature allows different component architectures to be used. The details of finer grained alternate architectures can be found in [26].

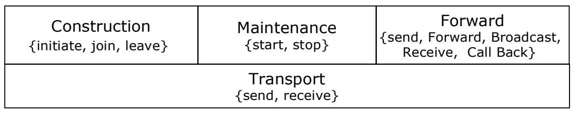

Fig. 2. Overview of Network Context

The Network Context is built by installing a set of compatible components that provide the necessary overlay level functionality required by the Service Context. Multiple overlays can be instantiated in the Network context either in a layered manner (e.g. SplitStream [7] over Pastry [24]) or side-by-side to offer multiple capabilities (e.g. DHT lookup [24] and streaming [27]).

Table 1. Overview of Generic Mesh Functionality

Component	**Interface Operations**
Generic Mesh *Construction*	*Join, Leave, Add Link, Remove Link.*
Lazy Mesh *Maintenance*	*Initiate, End.*
Generic Mesh *Forward*	*Send, Forward, Broadcast, Receive, Call Back*
Object *Transport*	*Send, Receive*

In the BitTorrent example, the Network Context is built using a *Generic Mesh* overlay; this is a highly reusable unstructured overlay that abstracts the topology to simple links between peers; its interfaces are provided in Table 1. It can be seen that the construction component is extended to also support add and remove operations in order to allow links to be manipulated. BitTorrent also uses a Lazy Mesh Maintenance component; this does not perform active probing and simply updates state information on the detection of a fault. Lastly, an Object Transport component is attached; this uses Java ObjectStream objects to transport chunks and protocol message.

3.3 Service Context

The Service Context consists of a number of content services and delivery mechanisms embodied in a set of cooperating components. These components use the Network Context as a platform over which they perform distributed interactions.

There are three primary types of components in the Service Context: *Managerial*, *Functional* and *Policy*. There is one *Managerial* component per service; this component will deal with managing multiple components to work in conjunction. For example, it will deal with the reconfiguration of cooperating components to react to environmental events. A *Functional* component embodies aspects of functionality to perform a particular service in the system. It is defined by its ability to actively initiate procedures itself. Alternatively, *Policy* components make decisions passively on behalf of the other components; an example of this is a source selector component which decides on the optimal source to use in a distribution scenario.

The Service Context is where the majority of BitTorrent's functionality resides; this functionality deals with a number of aspects operating over the Network Context:

a) *Bootstrapping* – It is necessary to obtain a list of potential sources.
b) *Request Generation* – It is necessary for requests to be issued to remote nodes.
c) *Request Handling* – It is necessary for chunk requests to be handled.
d) *Chunk Selection* – It is necessary to select which chunks to request first.
e) *Source Selection* – It is necessary to select which sources to utilise.
f) *Incentive Management* – Incentive mechanisms must encourage contribution.

Fig. 3 shows the Service Context of BitTorrent; *bootstrapping*, *request generation* and *request handling* are all embodied in functional components. These are components that perform active functions and can therefore initiate their own procedures. They are attached above the Network Context and use its *Forward* interface to perform distributed interactions.

Chunk selection, *source selection* and *incentive management* are all embodied in policy components. This is because they are passively used to make decisions based on the current state of the node. For example, a chunk selector will make its decisions based on the current chunks that are required.

A *BitTorrent Management* component is also attached to the system. It is responsible for coordinating the behaviour of the other components. For instance, it will coordinate interactions between the Bootstrapper component and the Request Generator. This also allows it to act as an adapter between incompatible components.

To enable these components to cooperate it is necessary to provide them with an interaction mechanism. In contrast to the strictly defined nature of the Network Context, the divergent nature of the Service Context lends itself well to event based interaction. This allows components to offer functionality in a very fine grained, event based manner. Therefore, Juno can support the use of subsets of component functionality allowing operations to be spread over a set of multiple components. Unlike the Network Context, the use of this event based architecture therefore does not fix the Service Context to use components in a particular architecture. Instead, components exist in an event orientated container. This means that components can simply be added to augment or modify functionality by automatically manipulating events and shared state.

3.4 Application Context

The Application Context resides above the Service Context and provides a layer of abstraction between the application and the middleware. The Application Context consists of a minimum of one component that provides an interface to the application. The type of interface is not strictly defined therefore allowing a variety of interaction approaches to be utilised. For instance, remote invocations can be utilised by installing a remote procedure call interface.

Generally, reusable, generic components are installed in the Application Context to offer abstractions to the application. However, as well as this, it is also possible for developers to implement their own components to offer more specialised access to the lower layers. For instance, a developer can extend the generic stored delivery interface to allow more detailed access to state information. An application utilising BitTorrent would therefore install a *Generic Stored Delivery* component. This component offers a simple abstraction, allowing downloads to be initiated or cancelled.

4 Evaluation

This section investigates a number of properties of the Juno middleware. The primary concern of this paper is how the heterogeneity encountered in content networking can be dealt with. This is achieved through a components based architecture that allows (re)configuration. Thus, the Juno (re)configurable architecture supports the utilisation of varieties of functionality within a single framework to cope with the heterogeneity of delivery, service, devices and networks. Hence, the (re)configurable properties of Juno, in the context of heterogeneity, is investigated first. This is then qualified against the resource overhead of utilising a (re)configurable approach. Lastly, an investigation into Juno's developmental benefits is performed in order to inspect the advantages of utilising the middleware to develop content based applications.

To aid in the evaluation a simple file-sharing application has been developed over Juno which utilises a Pastry [24] lookup facility alongside a BitTorrent [5] distribution mechanism. An overview is shown in Fig. 3. Arrows represent interaction between components. Further to this, the BitTorrent Management component also can interact with all components. This application will therefore be used to highlight a variety of features of Juno's operation.

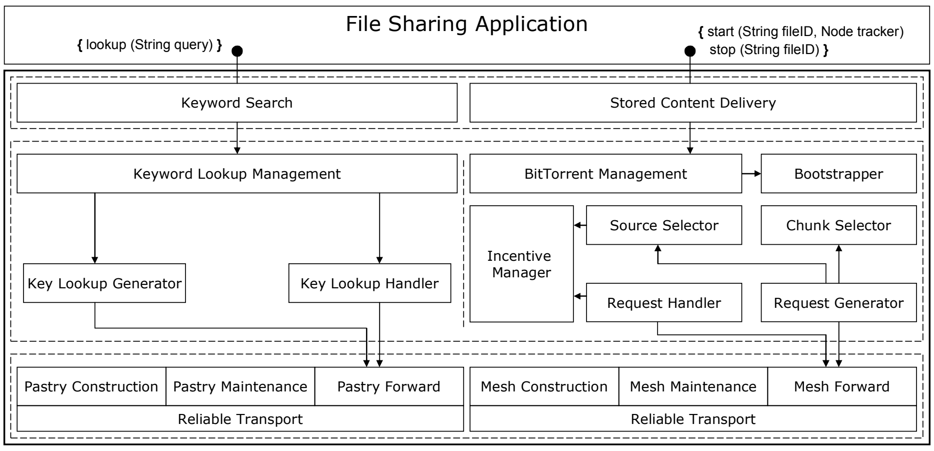

Fig. 3. Overview of File Share Architecture

4.1 (Re)Configurability

Section 1 introduced four types of heterogeneity (summarised in Table 2). These highlight the diversity in requirements and constraints involved in developing and deploying a content network. Juno addresses heterogeneity through the (re)configuration of individual nodes in the content network in order to embody optimised qualities. Configurability refers to the specialisation of the middleware for a particular set of requirements and constraints whilst re-configurability refers to the process of changing this configuration during runtime. This section therefore looks at how well Juno deals with the different types of heterogeneity (shown in Table 2) through its (re)configurable architecture.

Table 2. Summary of how Juno Addresses Heterogeneity

	Lightweight Configuration	Pluggable Mechanisms	Fine Grained Adaptability	Orthogonal Instantiation	Stacked Instantiation
Delivery		X	X	X	
Service		X	X	X	
Device	X		X		
Network	X		X	X	X

Delivery heterogeneity is of particular interest in content networks. This represents the diversity in which users access content. Many content networks offer a number of different access mechanisms such as stored and live streaming, stored content delivery and interactive content delivery. An example of this is 4oD [1] which offers both stored and streamed access to its content. Juno allows such diversity to be managed through its (re)configurability. For example, when a file sharing application is developed over Juno, streaming can be easily introduced to it. In the file sharing implementation this is done by installing streaming components orthogonal to the existing BitTorrent mechanism. Juno also allows a variety of different streaming mechanisms to be installed without mandating individual approaches. For instance, the use of tree based streaming [10] can be utilised in reliable environments whilst the use of mesh-based streaming [27] can be used in more transient environments. Importantly, Juno's support for installing multiple delivery paradigms also allows diverse delivery systems to be supported within one framework. This allows applications developed over Juno to adapt their delivery capabilities to interact with a range of systems. For example, if a user attempts to access a piece of content hosted in a Julia [5] network, Juno can attach the Julia components to provide compatibility. The use of (re)configuration therefore allows both coarse grained and fine grained architectural modifications to be made to ensure that delivery mechanisms coincide with user preference and application requirements.

Service heterogeneity is another significant concern that must be addressed by content networks. Thus, there are considerable benefits associated with the easy deployment, instantiation and interaction of services. It is therefore important to offer a generic framework to facilitate this. Juno deals with these issues by allowing services to be dynamically installed through using its (re)configuration capabilities. For instance, traditionally BitTorrent does not offer a search service; instead it focuses on the actual distribution of the file. Juno, in contrast, allows the addition of a file lookup service as a separate component. This is performed by plugging the Pastry overlay components into the Network Context whilst attaching the *Key Lookup* component in the Service Context. This component receives user queries from the Application Context and uses the Pastry overlay to route the query to the necessary node responsible for the specific hash space. Further, the utilisation of alternate search mechanisms can also be made without modification to the application. This is possible through the level of abstraction provided by the Application Context, meaning that it is only necessary for the application to know how to interact with the higher levels of the middleware. Therefore, through Juno, BitTorrent can incorporate new services with limited effort on the part of developers. Importantly, it is possible to introduce services without predefined support. Instead, the necessary functionality can be attached through components to ensure correct operations. Further, Juno's use of event-based interactions allows services to augment existing functionality through the monitoring, interception and modification of events.

Device heterogeneity is an increasingly prominent aspect of distributed systems. This refers to the range of devices connected to the content network. To ensure high performance and acceptable user experience, content networks must make consideration for this heterogeneity. By allowing fine grained (re)configuration, Juno can ensure that each device in the content network utilises optimised components. Therefore, a low capacity device will utilise a light-weight configuration in which only essential components are installed. This has two effects: *i)* it limits the memory and processing consumption on the device and *ii)* it allows specialised components that reduce resource utilisation to be installed. For instance, in the file sharing application, low capacity nodes utilise specialised Pastry components in the Network Context. These ensure that transient nodes play no part in routing. Instead these nodes use reliable peers as proxies. This offers improved performance due to the adverse effect churn has on routing. The configuration is performed by replacing the Pastry components in the Network Context with a single *Hidden Pastry Forward* component. This component is initiated with the location of one of the Pastry peers (*N*), which it will use to forward messages through. This is the only reconfiguration required; no modifications in the Service Context are made. Therefore, when the standard *Key Lookup* component sends a message through the Network Context the *Hidden Pastry Forward* component will always redirect it through node *N*. This highlights Juno's ability to modify functionality by reconfiguring small aspects. This allows the same core functionality to be performed in the system whilst exploiting the natural variations in end host capabilities.

Network heterogeneity refers to the diversity in which devices are connected to each other. Some can possess high bandwidth, reliable connectivity (e.g. Ethernet) whilst others can be considerably more constrained (e.g. Bluetooth). Juno's (re)configurability addresses this heterogeneity through utilising fine grained component configurations to ensure devices observing different network conditions behave differently to reflect this. For instance, the file share application could be placed in a number of environments (e.g. a reliable wired campus network or a mobile ad-hoc network). These differences can similarly be reflected in a number of different configurations. For instance, in the reliable environment Juno attaches lazy maintenance components in the Pastry overlay. These uses periodic keep-alive messages to maintain the leaf set. Conversely, in the unreliable environment, leaf set broadcasts are used to address the number of node failures. Juno can also perform this process dynamically in response to changes in network conditions (e.g. moving from a reliable connection to Bluetooth) without the need to modify the application. This process therefore allows overlays to have fine grained runtime modifications made to them to ensure resilience against different network environments.

This section has investigated Juno's approach of using (re)configuration to address the heterogeneity observed in content networks. Importantly, it can be seen that the process of encapsulating functionality in dynamically interchangeable components provides an effective mechanism for dealing with heterogeneity. This is achieved by abstracting services and requirements from their implementations, allowing different components to perform the same procedures in different environments. Further, the ability to easily extend the middleware through (re)configuration means that applications can easily incorporate new capabilities to address changes in heterogeneity. Importantly, the application is agnostic to these changes since Juno autonomously (re)configures itself allowing the application to simply interact with abstracted interfaces provided in the Application Context.

4.2 Resource Overhead

This section examines the performance overheads associated with implementing a content network using Juno. All tests were performed on a 3.4GHz Intel Pentium D processor; 2 GB RAM; Sun JVM 1.6.0.5.

The operational throughput of BitTorrent's *new source found* notification was measured over a 5 second period; this operation requires two parameters: a file identifier and a node reference. This operation was implemented in Juno using both event passing and receptacle calls. As a benchmark it was also implemented using native Java method calls. The results are shown in Table 3; it can be seen that when compared to native calls, there is a noticeable reduction in performance.

Table 3. Invocation Throughput

Type	**Throughput (Invocations/Second)**
Java Method Call	15.863570 10^6 (16 million)
OpenCOM Receptacle Call	3.222367 x 10^6 (3 million)
Juno Event Passing	1.510376 x10^6 (1.5 million)

Juno's use of receptacles and event passing therefore creates a clear overhead in the system. Receptacles and event passing, however, reduce coupling and allow reconfiguration; this therefore creates a trade-off between performance and (re)configurability.

The memory overhead of Juno has been assessed by implementing six modules as both components and Java objects. These modules have been implemented with an increasingly large number of interfaces and receptacles. The experiments show that implementing the system in Juno adds approximately 370 bytes of overhead per component, compared to the equivalent Java object. This value increases by approximately 20 bytes for every additional interface. This can be compared to 300 bytes for each extra OpenCOM receptacle. Therefore, development in Juno will lead to a small increment in memory overhead. However, the ability to use lightweight configurations (installing the minimal components), allows limited capacity devices to actually reduce the overall memory footprint.

4.3 Development Capabilities

Clearly, a significant evaluative metric is how well Juno supports the development of new applications. This is assessed through three approaches; firstly, looking at the potential for component reuse in the system; secondly, looking at how applications can utilise new functionality through adding new components to Juno; and thirdly, through the coding overhead of implementation in Juno. Development can take place in any of the contexts, or alternatively, above Juno. This section focuses on the former as it deals more specifically with Juno rather than applications built over it.

Reusability levels in Juno are significant; most noticeably these are in the Network Context due to its role as a platform. This therefore allows a number of Service Context components to operate over reused/shared overlay components. For instance, the mesh components used by BitTorrent can be further used with overlays such as Julia [4], Narada [9] and Gnutella [23]. This offers significant development opportunities as it can dramatically reduce coding time.

The Service Context also offers high-levels of reusability; components such as the Keyword Lookup component can obviously be ported to a number of applications that require this functionality. Further, fine grained components such as the Incentive Manager, Request Generator and Request Handler can be reused in a variety of different systems. For example, BitTorrent can easily be configured to support streaming applications. To do this, temporally-aware chunk and source selectors are installed, leaving all other components the same.

Another developmental benefit of Juno is its support for functionally scaling applications. This is achieved through the introduction of new components that can dynamically manipulate events and component connections to augment functionality. This allows new components to be dynamically deployed between nodes to extend functionality ‘on-the-fly’. On a coarse level, entire sets of components can be installed. For instance, if a node wishes to download an item of content from another but they do not have compatible delivery mechanisms; this can be easily resolved through component exchange. More fine grained deployment can also be performed; for instance, a peer utilising a modified BitTorrent implementation to stream content can easily interact with other oblivious BitTorrent implementations. However, the traditional BitTorrent incentives scheme will not be effective, as chunks that are nearer to a node’s playback position are more valuable than distant ones. Therefore, new incentive mechanisms (e.g. a digital currency) can simply be deployed between peers to facilitate access to certain chunks.

Table 4. Transport Component Code Complexity Overview

	Lines of Code	**Difference**
Full Component	110	0
Without Event Capabilities	102	- 8
Without Component Capabilities	88	- 22

To provide an overview of the coding overhead related to developing systems in Juno the Generic Mesh Construction component is inspected, shown in Table 4. This component has one receptacle, *Transport*, which provides network level transport functionality. The full component has 110 lines of code; 22 lines are attributed to managing component receptacles and 8 lines are required to deal with the event based notification of messages received by the Transport component. There is therefore a small coding overhead in implementing the Juno components. However, this overhead is in the form of template-like coding; further the use of Juno’s well defined approach can assist in such things as code maintenance and project management.

5 Conclusion and Future Work

This paper provides an overview of the Juno content networking middleware. Juno is designed to address the complexities of next generation content distribution. The proliferation of multimedia content distribution over the Internet has led to an explosion in the ways in which users choose to view content, leading to a transition from content distribution networks to more integrated content networks [22]. This diversification has resulted in huge array of content, overlays, services and delivery mechanisms, creating significant complexities when developing and deploying content networks over the Internet.

Juno addresses these issues through its use of an open, (re)configurable component architecture allowing it to dynamically build and rebuild itself. This allows Juno to efficiently support a diverse range of applications by (re)configuring itself based on environmental constraints and application requirements. Juno has been evaluated through the development of a file sharing application using BitTorrent and Pastry. It is shown that significant levels of (re)configurability can be achieved to specialise and adapt content networking systems. This is evaluated by showing how Juno deals with the four primary heterogeneity factors (i.e. delivery, service, devices and network heterogeneity). More specifically, it is shown how the different requirements of these factors can be accommodated through using Juno's (re)configuration. Further, Juno's functional scalability and the ability to reuse components have been shown to offer considerable benefits to developers. These properties have also been placed in consideration of an overhead study, showing that there was a noticeable but manageable overhead, causing a trade-off between performance and configurability.

There is a considerable body of future work that can be carried out in this area. Middleware support for this new generation of content networking is in its infancy whilst Juno is still in the relatively early stages of development. The next step is to develop Juno further, introducing a wider range of services and delivery mechanisms. One area of significance is the security of Juno; currently the use of digitally signed components is presumed to offer security, however, more sophisticated support for the secure functional scalability of applications is necessary. This will involve both the development of more advanced component deployment alongside more sophisticated remote reconfiguration of nodes.

References

1. 4oD – Channel 4's TV and Film on Demand Service. *http://www.channel4.com/4od/*
2. Akamai. *http://www.akamai.com.*
3. Behnel, S. and Buchmann, A. Models and Languages for Overlay Networks. In Proc. Intl. Workshop on Databases, Information Systems and Peer-to-Peer Computing, Trondheim, Norway (2005).
4. Bickson, D. and Malkhi, D. The Julia Content Distribution Network. In Proc. Conference on Real, Large Distributed Systems, San Francisco, CA (2005).
5. BitTorrent Specification. *http://www.bittorrent.org/beps/bep_0003.html*.
6. Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. The Design and Implementation of Open ORB V2. In IEEE Distributed Systems Online, Special Issue on Reflective Middleware, vol. 2 (2001).
7. Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., and Singh, A. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In Proc. ACM Symposium on Operating Systems Principles Bolton Landing, NY (2003).
8. Chen, F., Repantis, T., and Kalogeraki, V. Coordinated Media Streaming and Transcoding in Peer-to-Peer Systems. In Proc. IEEE Intl. Parallel and Distributed Processing Symposium, Denver, CO (2005).
9. Chu, Y., Rao, S., Seshan, S., and Zhang, H. Enabling Conferencing Applications on the Internet Using an Overlay Muilticast Architecture. In SIGCOMM Computer Communications Review, vol. 31, issue 4 pp. 55-67. Oct (2001).
10. Chu, Y., Rao, S.G., and Zhang, H. A Case for End System Multicast. In Proc. ACM SIGMETRICS, Santa Clara, CA (2000).

11. Coulson, G. A Configurable Multimedia Middleware Platform, IEEE Multimedia Magazine, vol 6, issue 1, pp 62-76, IEEE Press, January-March (1999).
12. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J, and Sivaharan, T. A Generic Component Model for Building Systems Software. In ACM Transactions on Computer Systems, vol. 27, issue 1, pp. 1-42, February (2008).
13. Furmento, N., Lee, W., Mayer, A., Newhouse, S., and Darlington, J. ICENI: An Open Grid Service Architecture Implemented with Jini. In Proc. ACM/IEEE Conference on High Performance Networking and Computing, Baltimore, MA (2002).
14. Gerke, J., Hausheer, D., Mischke, J., and Stiller, B. An Architecture for a Service Oriented Peer-to-Peer System. In Praxis der Informationsverarbeitung und Kommunikation (PIK), No. 2, 2003.
15. Gong, L. JXTA: A Network Programming Environment. In IEEE Internet Computing, vol. 5, issue 3, pp.88-95, May/June (2001).
16. Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., and Keleher, P. Adaptive Replication in Peer-to-Peer Systems. In Proc. Intl. Conference on Distributed Computing Systems, Tokyo, Japan (2004).
17. Grace, P., Coulson, G., Blair, G.S., and Porter, B. Deep Middleware for the Divergent Grid. In Proc. IFIP/ACM/USENIX Middleware, Grenoble, France (2005).
18. Gu, X., Nahrstedt, K., and Yu, B. SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. In Proc. IEEE Intl. Symposium on High Performance Distributed Computing, Honolulu, HI (2004).
19. Hefeeda, M., Habib, A., Botev, B., Xu, D., and Bhargava, B. PROMISE: Peer-to-Peer Media Streaming using CollectCast. In Proc. ACM Intl. Conference on Multimedia, Berkeley, CA (2003).
20. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing Declarative Overlays. In SIGOPS Operating Systems Review Oct (2005).
21. Mohan, R., Simth, J.R., and Li, C.S. Adapting Multimedia Internet Content for Universal Access. In IEEE Transactions on Multimedia, vol. 1, issue 1, pp. 104–114, March (1999).
22. Plagemann, T. Goebel, V., Mauthe, A., Mathy, L., Turletti, T., and Urvoy-Keller, G., From Content Distribution to Content Networks – Issues and Challenges. Computer Communications, vol. 29, issue 5, pp. 551-562, (2006).
23. Ripeanu, M. Peer-to-peer Architecture Case Study: Gnutella Network. Technical Report, University of Chicago (2001).
24. Rowstron, A. and Druschel, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems, In Proc. IFIP Middleware, Heidelberg, Germany (2001).
25. Stiller, B., Bauer, D., Caronni, G., Class, C., Conrad, C., Plattner, B., Vogt, and M., Waldvogel, M. DaCaPo++ – Communication Support for Distributed Applications, ETH Zürich, Computer Engineering and Networks Laboratory TIK, Switzerland, TIK-Report issue 25 (1997).
26. Tyson, G. Component Based Overlay Development in Gridkit. Available at *http://www.comp.lancs.ac.uk/~tysong/MScThesis.pdf.* MSc Thesis, Lancaster University (2006).
27. Zhang,.X, Liu, J., Li, B., and Yum, T.S.P. CoolStreaming/DONet: A Data-driven Overlay Network for Live Media Streaming. In Proc. IEEE Infocom, Miami, FL (2005).