Liberating Content: Adaptive Middleware for Providing Content-Centric Networking Support

Gareth Tyson[1], Andreas Mauthe[1], Sebastian Kaune[2], Colin Parsons[1], Thomas Plagemann[3], Paul Grace[1]

[1] InfoLab21, Lancaster University, UK
[2] Multimedia KOM, Technical University of Darmstadt, Germany
[3] Department of Infomatics, University of Oslo, Norway
[1]{g.tyson, andreas, p.grace, c.parsons}@comp.lancs.ac.uk, [2]kaune@kom.tu-darmstadt.de, [3]plagemann@ifi.uio.no

Abstract. This paper investigates the role of middleware in deploying content-centric networks. A middleware called *Juno* is described, exploiting recent observations that applications rarely have an interest in how or where their content (e.g. videos, software updates) is obtained from as long as it is delivered within certain requirement bounds. Juno provides a high level abstraction through which applications can request content agnostic to its underlying delivery. This abstraction subsequently allows dynamic (re)configuration, through software reflection, based on flexible and extensible runtime meta-data. The entire process involves the location of content sources alongside the selection of the most optimal available delivery scheme. The middleware then reconfigures to interact with this choice, providing transparent interoperation for the application. Juno has been evaluated by deploying a number of case-studies on the Emulab testbed. It is shown that traditional applications, deployed with static, design-time selected delivery schemes cannot compete with Juno's approach of dynamic selection. Further, through distributed adaptation it is shown how Juno can adapt to environmental variations.

Keywords: Content-centric, content delivery, configurable middleware

1 Introduction

A number of recent studies have highlighted the importance of content delivery in the Internet, showing that a predominant amount of traffic is attributable to content distribution [23]. It is envisaged that in the future a fully integrated content-centric infrastructure will replace various proprietary and heterogeneous content delivery systems [29]. This will consist of a large-scale decentralised network of hosts capable of exploiting various services and resources to optimise delivery. Currently, however, this is not available; instead, a large number of independent content delivery schemes exist, ranging from client-server HTTP [18] to peer-to-peer models such as BitTorrent

[5] and CoolStreaming [38]. These systems have been built to address particular requirements that arise when operating with divergent workloads and environments. These requirements can be high-level (e.g. the need for stored or streamed delivery) or quite low-level (e.g. the need for particular bit rates). Hence, it is the responsibility of the developer to select, statically at design time, the most appropriate delivery mechanism for the application's needs. This, however, is difficult to do efficiently due to the complexity of predicting future runtime behaviour.

As an example, a video streaming application might select the use of an RTP client-server model due to its convenient deployment. However, as its popularity increases, server resources will become saturated, degrading the service quality. In this situation, the choice of RTP becomes suboptimal compared to more scalable peer-to-peer equivalents (e.g. [38]). Client-side applications also suffer, as a large range of environmental (e.g. client bandwidth) and workload variations (e.g. user selections) affect performance. As these factors are only evident at runtime, it is difficult to take them into account during the design. This is exacerbated further if they vary between different nodes as each node will require separate configuration.

The emerging *content-centric networking* paradigm recognises that applications rarely have a vested interest in how or where their content is obtained from as long as it is delivered within certain requirement bounds [19]. So far, content-centric networking has focussed on the discovery and validation of content, providing an abstraction that allows applications to interact with the network using a content request/reply model [24]. It has not, however, focussed on the content-centric *delivery* of data. Content-centric delivery should not restrict an application to utilising statically fixed delivery schemes; instead, it should dynamically ensure that content is delivered optimally considering the content characteristics and delivery infrastructure.

This paper introduces a scheme for content-centric networking *and* content-centric delivery supported by a single middleware framework, called *Juno*. Juno exploits reflective software principles [4] to offer support for applications requiring content delivery. Central to Juno's contribution is the decomposition of content systems into three well-defined architectural entities: *content*, *discovery* and *delivery*. Through reflective, component-based (re)configuration, these elements are managed and adapted in a fine-grained way to locate and interoperate with the optimal available delivery scheme on behalf of the application. Subsequently, from the application's perspective, the middleware builds a seemingly fully integrated content infrastructure that handles requests through a high level abstraction. Therefore, unlike approaches such as [11][32], the application interacts with Juno rather than the network.

The rest of the paper is structured as follows; in Section 2 a background to content centric-networks is given. Section 3 provides a detailed overview of Juno's internal software architecture. Section 4 then evaluates Juno looking at the performance and overhead factors. Finally, in Section 5 the paper is concluded, outlining future work.

2 Background and Related Work

Content-centric networking is a paradigm-shift that is part of current research attempts at re-architecting the Internet (e.g. SPSwitch [11]). It involves the detachment of content from location, allowing applications to forward requests into the network as opposed to host-addressed packets. For example, DONA [24] provides a simple primate (FIND) through which nodes can access content. In essence, this is an anycast that locates and returns the closest instance of the requested object. Content-centric systems have become almost synonymous with such content-based routing schemes [6][20], however, we believe it is vital to place equal importance on the actual delivery of content. With this in mind, we view truly content-centric networks as possessing three discrete elements: *content*, *discovery* and *delivery*. This section outlines existing work alongside a brief overview of relevant middleware.

The first element, *content*, refers to the data itself and therefore constitutes the most important aspect of a content-centric system. This is usually manifested through some form of database such a Content Management System (CMS) [27].

The second element, *discovery*, is the process by which consumers locate sources of the content. A number of systems exist for this, including client-server search engines [15], peer-to-peer overlays [14][30][31] and content-based routing [6][20]. Most relevant to Juno is the emergence of content-centric networks such as DONA [24], AGN [19] and RTFM [32]. These are the primary manifestations of content-centricity and generally focus on the location and validation of content. These schemes, however, need large-scale deployment to become effective in the same way other global routing schemes do e.g. IP. This also extends to requiring providers to globally cooperate with the scheme. This is ineffective, considering the wide distribution of content over various systems (e.g. peer-to-peer, CDNs, etc.). A further criticism is that these schemes do not offer configurable delivery of content after discovery. For example, a content-centric routing scheme would not be able to join a BitTorrent swarm through its tracker. Instead, it would be necessary to request each BitTorrent chunk as a separate content object, which would be delivered using the content-centric network's (statically) chosen transport scheme.

The third element is *delivery*. A number of delivery schemes are available including peer-to-peer [5][14][38], server-oriented [12][18] and infrastructures such as Content Distribution Networks (CDNs) [1]. These have been shown to be suitable for a variety of (disjoint) situations [22][33]. It is the aim of this paper not to extend these, but to exploit their individual properties within a middleware framework to best serve application requirements and operating environments.

To the best of our knowledge, no existing middleware focuses on the configurable content-centric delivery of content. The closest area of work is, as mentioned previously, existing content-centric routing schemes. Content publish-subscribe models have also become a popular paradigm (e.g. [2]); this abstraction, however, is fundamentally different (although often content-based routing is also employed e.g. [20]). A number of more traditional multimedia delivery middlewares have been developed such as TOAST [13] GOPI [8] and DaCaPo++ [35]. These, however, are not content-centric and depend entirely on location based delivery. Their focus is on providing real-time, configurable QoS support, making their contributions orthogonal to ours. Also, their operation is based on point-to-point streams of content. This is not appropriate for real-world use considering the heterogeneity of current schemes (c.f. [23]), comprising of both client-server and peer-to-peer mechanisms. In the field of peer-to-peer middleware a number of systems exist including GridKIT [16] and ODIN-S [7]. These, however, are not aimed at content delivery and therefore do not offer the correct abstractions or underlying functionality to efficiently provide content-centric support. Considering the lack of prior work we therefore believe it is important to investigate the advantages and limitations of providing content-centric delivery as well as discovery in the middleware layer.

3 Juno Middleware Design

This section details the Juno middleware. First, the architectural principles employed by Juno are described alongside its constituent elements. Following this, it is shown how Juno builds and adapts itself using these constituent elements.

3.1 Juno Architecture and Principles

In order to support content-centric applications, Juno exploits architectural software principles [4][16] to achieve optimisation through interoperation and adaptation.

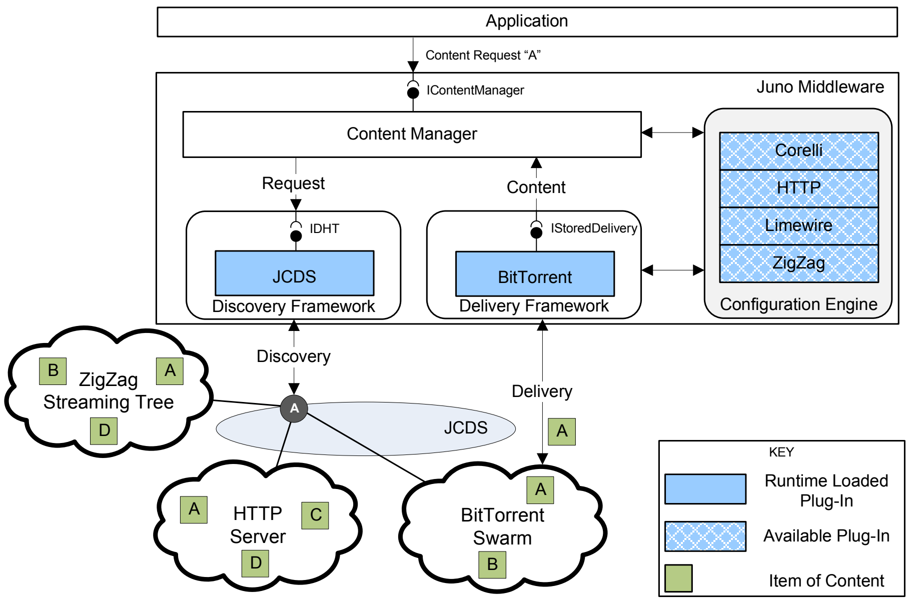

Fig. 1. Overview of Juno Architecture and Operations

Juno's design consists of three architectural elements: *content management*, *discovery* and *delivery*. These three elements are represented through frameworks capable of independent manipulation, as shown in Figure 1. Specifically, they host *plug-ins* (potentially provided by third parties) capable of discovering, delivering or providing content. Plug-ins are required to implement *i)* service specific interfaces and *ii)* framework specific interfaces for attaching, initiating, resuming and detaching plug-ins. There is no fixed component pattern required for building a plug-in; this is left to the developer. This flexible approach even allows plug-ins to consist of multiple plug-ins operating in a composite manner. A Configuration Engine maintains an extensible set of these plug-ins which can be attached based on application requirements and environmental conditions. This section describes the primary components of Juno.

3.1.1 Content Manager

Juno abstracts the content management away from any individual delivery schemes, thus allowing them to share a common content library. The Content Manager is an individual component within Juno that provides this service; it offers methods to lookup, store and manipulate local content. All delivery schemes utilise the Content Manager, negating the need to transfer content between delivery plug-ins if they are interchanged. Further, this provides an elegant mechanism by which multiple plug-ins can be coordinated without over-writing each others' data. In order to allow convenient usage, the Content Manager also offers multiple interfaces to enable both chunk-based and range-based access to content. Therefore different schemes can view data in different ways making implementation and development easier.

Due to the content-centricity of Juno, the Content Manager is considered a primary entity in its operation. As such, it is the first tier of interaction between the application and the middleware. Content requests from the application are routed through the Content Manager for resolution, as shown in Figure 1. Content requests are formatted as unique content identifiers; these are a 128 bit hash of the data to allow later content validation (as in [24]). Identifiers can be located through a variety of means; for instance, an IPTV system might locate its required identifiers through a Programme Manager whilst an application requiring software updates would retrieve a set of required updates from an Update Server. If the content is available locally it is returned to the application, however, if it is not, the Content Manager begins the process of obtaining it from remote sources. Once the Content Manager gains a handle on the content, the data is returned to the application using its preferred abstraction (i.e. stored or streamed). The Content Manager therefore remains as a constant within the middleware even during the adaptation of other elements.

3.1.2 Discovery Framework

When a local instance of the desired content is not available in the Content Manager it is necessary to locate sources through which it can be accessed. Current content-based routing approaches require that their systems are propagated with information by providers. This is largely unrealistic as often different content networks utilise their own discovery service e.g. eMule [10] uses KAD [21], BitTorrent utilises trackers [5], Limewire [26] utilises Gnutella [14], etc. Unlike current approaches, Juno does not rely on an individual content location mechanism through which lookups can be performed. The Discovery Framework instead hosts one or more *discovery plug-ins* that can be used to interoperate with existing already deployed discovery systems.

Discovery plug-ins are required to offer the IDHT interface which provides a hash table abstraction. The IDHT interface allows the Content Manager to issue requests using a flat content identifier such as used in [24]. Each plug-in must therefore adapt this identifier to their required data format; this can be done locally or through remote services (e.g. remote resolution using a DNS-like system). This need, however, is becoming increasingly mitigated as most discovery infrastructures are moving towards the use of such unique identifiers, often to enable multi-source downloading (e.g. [10][24][26]). For older systems limited to keyword searches it is also possible for applications to issue requests based on search terms. It is, however, necessary for the application to select from the subsequent results. Each plug-in must return these results using the standard RemoteContent object. The plug-ins that are attached to the framework are decided by the application based on its requirements and where it wishes or anticipates its content to be stored. When multiple schemes are attached, requests are forwarded through every plug-in and the results then aggregated. It is also possible for new discovery schemes to be plugged in dynamically, allowing new plug-ins to be attached if an item of content cannot be found.

Due to the interoperable nature of nodes running Juno, they are in a unique position to bridge the incompatibility between many diverse discovery schemes and a single unified system. Juno exploits this position by building a shared lookup overlay called the Juno Content Discovery Server (JCDS) which is built over Pastry [31] (shown in Figure 1). This distributed hash table is propagated with information uploaded by providers wishing to offer content. As well as this, however, consumers that are aware of content and sources (obtained through alternate discovery schemes) also upload information. For instance, a node that downloads a file from a HTTP server will upload a reference to this server on the JCDS. This allows a substantial content directory to be built up progressively through contributions by all peers. Data in the JCDS is hashed using the content identifier; lookups return a reference to the content meta-data (name, encoding rate, owner, etc.) and a list of sources with their related characteristics (supported delivery schemes, upload bandwidth, network coordinates [25], etc.). Obviously, the availability of such information varies between different discovery and delivery systems and therefore the node simply uploads as much information as possible. Information uploaded by multiple nodes is also aggregated together to best reflect the current environment. A lookup in the JCDS therefore provides the requester with all the necessary information to find out what sources exist, their characteristics and what delivery protocols they support. Nodes can therefore solely utilise the JCDS until its desired content cannot be found, resulting in the attachment of alternate plug-ins.

3.1.3 Delivery Framework

Once the Discovery Framework has located a set of potential sources, the Configuration Engine is used to select the optimal source(s) and delivery plug-in (c.f. Section 3.2) in order to allow interoperation with the chosen content infrastructure. The selected delivery plug-in is then attached to the framework; the Content Manager can interact with it using one of the delivery interfaces from Table 1. Data received by the plug-in is returned to the Content Manager so that it can either send it to the application or the file system. Importantly, through this indirection, the Delivery Framework can also adapt and reconfigure the underlying delivery infrastructure without application awareness. For instance, if during a HTTP delivery, the service breaches the application’s requirements (e.g. bit rate drops below a threshold) then the Delivery Framework can remove HTTP and attach an alternative plug-in that offers the same content e.g. BitTorrent [5]. This process can further be performed in a distributed sense by requesting remote side reconfiguration (c.f. Section 3.2.2).

The Delivery Framework also hosts plug-ins capable of providing content to other users. This is intuitive as many plug-ins that download content will also upload it (i.e. peer-to-peer). Further, plug-ins can also solely offer provision support. Providers can therefore actively place their content on multiple delivery systems or, alternatively, publish it through Juno which can be dynamically (re)configured to support multiple schemes (Juno client applications can interact with either approach).

Interface	Description

IStoredDelivery	`- startDownload(RemoteContent content)`
	`- pauseDownload(String contentID)`
	`- stopDownload(String contentID)`

IStreamedDelivery	`- startStream(RemoteContent content)`
	`- pauseStream(String contentID)`
	`- stopStream(String contentID)`

IContentProvider	`- provide() //everything in content manager`
	`- provide(Collection<LocalContent> content)`
	`- stopProviding(String contentID)`

Table 1. Overview of standard delivery abstractions

3.2 Configuration Engine

The Configuration Engine selects optimal plug-ins. By performing (re)configuration at the middleware layer, it allows applications to operate transparently over optimised content support without having to deal with the complexities themselves. This section gives an overview of Juno's local and distributed (re)configuration.

3.2.1 Local Configuration and Reconfiguration

In contrast to traditional configurable systems (e.g. distributed objects [4], media streaming [8] and configurable network protocol stacks [35]), there is often no point-to-point requirements in a content-centric system. This is because, by their nature, content-centric applications do not require access to a particular node; instead, they require access to a particular item of content which will often exist at a number of different locations. This means that it is much more effective for the middleware to reconfigure locally and simply find *compatible* remote sources rather than attempting to perform remote adaptation of currently *incompatible* sources.

Delivery plug-ins are associated with meta-data that describes their performance and overhead; this information is structured in *attribute-value* pairs. The application represents its requirements in terms of selection predicates that dictate the preferred values of the meta-data exported by the plug-in e.g. mem_load <500KB. A collection of predicates is termed a *meta-description*. Predicates are also generated by the Configuration Engine based on policy rules (rules are compared against monitored context information such as bandwidth, processor load, etc., in the same way as [16]). If a rule is trigged it creates, modifies or removes one or more selection predicate(s).

Unlike the meta-data utilised by many reflective systems, information pertaining to content delivery is highly runtime dependent. For instance, each delivery plug-in must state the availability (existing number of replicas) available to it. This information can only be provided dynamically. To achieve this, it is necessary to *seed* each plug-in with runtime information acquired through the discovery process. When the Discovery Framework performs a lookup on the JCDS for an item of content the information returned is the content's meta-data, a list of sources and meta-data describing the sources' most recent characteristics (e.g. available upload bandwidth). This information is then passed to the relevant plug-ins (e.g. tracker information is passed into the BitTorrent plug-in). Using this runtime information, each delivery plug-in generates a representative set of meta-data for inspection. Details of this meta-data and how it is used are shown in Section 4. If such information is not available from the Discovery Framework, it is necessary to utilise heuristic default values that are associated with each plug-in; these, however, are obviously less accurate.

This decision process is executed every time the application requests an item of content that isn't satisfied by the local Content Manager. It is also executed whenever a plug-in's meta-data changes, a new source of desired content is found or the meta-description changes. To locate new sources and to modify plug-in meta-data the discovery plug-ins are periodically used to re-request information. If this results in a different plug-in being considered optimal, the current one is unplugged and replaced. All plug-in state is externalised and all data exists within the Content Manager, allowing the reconfiguration to take place without inter plug-in cooperation.

3.2.2 Distributed Configuration and Reconfiguration

Whilst it is usually possible to exploit local (re)configuration due to the non point-to-point relationship between consumers and content, it is also useful to offer coordination between multiple nodes. For instance, if a node has been performing a chunk based download, it becomes difficult to reconfigure to use a superior source that does not support the notion of chunks (e.g. HTTP [18], FTP [12], etc.). It is therefore beneficial for the consumer to request that the other node offers its content in an alternative way. This can be done by both providers and consumers.

Distributed Configuration is handled by the DistributedConfigurationCoordinator. This component receives configuration requests from plug-ins alongside relevant nodes to contact. It then contacts the necessary nodes, on behalf of the plug-in, to initiate the configuration. This service is pluggable allowing different coordinators to be used in different situations. For instance, a point-to-point coordinator will be used for client-server reconfigurations whilst a gossip-based coordinator can be used for larger scale decentralised reconfigurations; currently, Juno only offers the former.

It is possible for nodes to either accept or reject reconfiguration requests; this is a decision made by the application. This is achieved by applications implementing the IConfigurationAcceptor interface. This allows Juno to pass a reconfiguration request, alongside a reference to the requesting node, so that the application can decide whether or not to accept it. For instance, a server application operating over Juno would usually accept reconfiguration requests (or maybe just for premium clients) whilst an application operating on standard peers might only ever accept reconfiguration requests from nodes fulfilling some form of incentive criteria (e.g. friends or nodes providing resources). If a node accepts a reconfiguration request then it will instantiate the desired configuration and reply with any required bootstrapping information.

4 Validation and Evaluation

Juno has been implemented in Java using the OpenCOM component model [9]. It has been enabled with a number of discovery and delivery plug-ins alongside corresponding meta-data; an overview of plug-ins is shown in Table 2. To validate and evaluate Juno's ability to optimise delivery through (re)configuration, a number of case-study experiments are employed. These case-studies have been designed to reflect the most relevant features and common use cases. They have been implemented and tested using the Emulab testbed [36] to provide quantitative performance and overhead details. Emulab contains a number of dedicated hosts connected via an emulated network. Each node can be configured to possess specific network characteristics (e.g. bandwidth) allowing tests to be performed in a realistic setting that is subject to all appropriate limitations including bandwidth variations, bottlenecks, packet-loss, latency and real-world network protocol implementations.

Framework	Available Plug-ins
Discovery	Pastry [31], Gnutella[14], ASAP [28], Query Server
Delivery	BitTorrent [5], HTTP [18], Corelli [37], Limewire [26], ST [34]

Table 2. Overview of Supported Juno Plug-ins

To allow a thorough evaluation, this section focuses on the stored delivery of content rather than streamed delivery or its prior discovery. The first case-study highlights the benefits that can be gained through providing transparent reconfiguration of delivery plug-ins. The second case-study shows how distributed reconfiguration can adapt consumers to new operating conditions whilst the third case-study highlights how providers can also be adapted to best optimise delivery.

4.1 Delivery Optimisation through Transparent Interoperation

The first evaluative case-study investigates Juno's ability to achieve delivery optimisation by transparently selecting and interoperating with the best available sources and delivery schemes, leaving the application agnostic to the process.

4.1.1 Case-Study Overview

In this case-study two vital parameters are looked at: *client bandwidth* and *content size*. To study this, two experiments are carried out using two nodes of different capacities. In the first experiment a low capacity node, *Node LC*, is operating over an asynchronous, non-contended ADSL connection capable of 1.5Mbps download capacity alongside 784Kbps upload capacity. In the second experiment, another node, *Node HC*, operates over a much faster 100Mbps synchronous connection. A Juno client application is operating on both nodes, firstly requesting the download of a 72MB cartoon followed by a 4.2MB music file. After the discovery process, the content is found to be available on three delivery mechanisms by Node LC, whilst four are found by Node HC; these are listed with their characteristics in Table 3.

Available for	Delivery Scheme
Node LC Node HC	*HTTP* [18]: A HTTP server. There is 2Mbps capacity for the download to take place. The server is 10ms away from the client.
Node LC Node HC	*BitTorrent* [5]: A swarm sharing the desired file. The swarm consists of 24 nodes (9 seeds, 15 leeches). The upload/download bandwidth available at each node is distributed using a real world measurements that are already used in BitTorrent studies [3].
Node LC Node HC	*Limewire* [26]: A set of nodes possessing entire copies of the content. Four nodes possessing 1Mbps upload connections are available and considered the optimal sources.
Node HC	*Replication Server*: A private replication server hosting an instance of the content on Node HC's local area network. The server has 100Mbps connectivity to its LAN and is located under 1ms away. The server provides data through HTTP to 200 clients.

Table 3. Overview of Available Delivery Schemes

4.1.2 Performance Evaluation

The above case-study has been setup in Emulab; Juno has then been deployed on the nodes and measured. Figure 2 shows measurements taken from both Node LC and HC. It shows the application layer throughput for the large and small file downloads when utilising each delivery scheme. It also shows the throughput of Juno which is capable of selecting any plug-in (the decision process is shown in Section 4.1.3).

It is first observable that the results between Node LC and HC are disjoint, indicating that in this situation an application utilising statically assigned delivery mechanisms would not be capable of reaching optimality. In fact, in the case of the 72MB delivery, HTTP is the highest performing plug-in for Node LC but the lowest performing plug-in for Node HC. The deployment of a homogenous application would therefore result in significant performance limitations. For instance, if the application was designed to optimise Node LC, then the performance of Node HC would be lowered by ~50%. Similarly, if the application optimised performance for Node HC, then Node LC would have its performance reduced by ~33%. In contrast, Juno selects the best plug-in on a per-node basis, overcoming this heterogeneity.

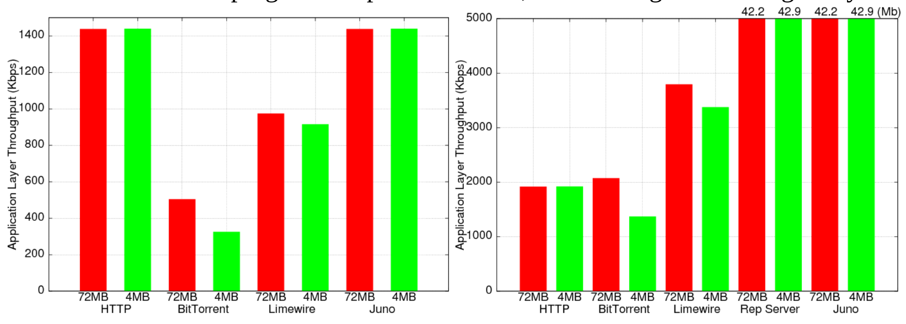

Fig 2. Average Throughput of Deliveries for *(a)* ADSL and *(b)* 100Mbps Connections

The second observation is that the delivery mechanism considered optimal for the 72MB download is not always the optimal choice for the 4.2MB download. For instance, when operating on Node HC, BitTorrent is faster than HTTP for the 72MB file but slower than HTTP for the 4.2MB file. In fact, the 4.2MB delivery achieves only 66% of the throughput measured by the 72MB delivery using BitTorrent. This is due to the complexity and length of time associated with joining a peer-to-peer swarm. Existing applications requiring content do not differentiate between delivery schemes on such a fine-grain basis as it creates a high coding overhead. Applications utilising BitTorrent will therefore observe a noticeable performance decrease (34%) when downloading the second file. In contrast, Juno selects delivery schemes on a per-content basis therefore reconfiguring between different schemes for the first and second downloads. This removes such performance limitations elegantly without a coding overhead. These two considerations result in Juno selecting HTTP for Node LC and the replication server of Node HC, as can be seen in Figure 2.

	App Worst Case		**App Second Best Case**		**App Best Case**	
	4.2MB	**72MB**	**4.2MB**	**72MB**	**4.2MB**	**72MB**
DSL	+343%	+185%	+57%	+48%	+/- 0%	+/- 0%
100	+2979%	+2141%	+1013%	+1174%	+/- 0%	+/- 0%

Table 4. Performance Improvement of Juno compared to Static Application Selection

Table 4 provides the percentage increase in throughput when using Juno during these experiments. The worst case scenario compares Juno against an application that has made the *worst* possible design-time decision (using the above figures). The best case is when the application has made the *best* decision (obviously resulting in the same performance as Juno in this situation). Node HC gets such high throughput improvements due to the availability of the local replication server.

4.1.3 Overview of Juno's Behaviour

To validate how Juno achieves these performance benefits, the exact process by which it operates is now outlined. The following details pertain to Node LC.

1) The application requests Cartoon.avi from the Juno Content Manager alongside a set of requirements and its preferred access mechanism (i.e. IStoredDelivery). For simplicity, we use the content identifier 'Cartoon.avi'.
2) The Content Manager checks its local database and finds it is not available.
3) The Content Manager passes the request to the Discovery Framework which creates a lookup on the JCDS for 'Cartoon.avi'. The lookup returns a list of all the available sources and delivery schemes. The results are encapsulated in a RemoteContent object; this contains the meta-data for the content alongside a list of all the sources and their characteristics e.g. supported delivery schemes.
4) The Discovery Framework returns the information to the Content Manager which *i)* loads the source characteristics into the respective delivery plug-ins and *ii)* invokes a `buildService` request on the Configuration Engine. This invocation includes the required interface (i.e. IStoredDelivery), the list of available plug-ins and the predicates originally provided by the application.
5) The Configuration Engine compares the selection predicates against the currently offered meta-data by each delivery plug-in. Details of this for Node LC are shown in Table 5, resulting in the selection of HTTP.

6) The HTTP delivery plug-in's `execute` method is invoked to construct it. This initiates the plug-in and attaches it to the Delivery Framework. The Configuration Engine then returns a reference to the Content Manager.
7) The Content Manager invokes the plug-in's `startDownload` method, passing in the RemoteContent object returned by the Discovery Framework.
8) The HTTP plug-in selects the best source from the RemoteContent parameter and initiates the download. The received data is passed through the Content Manager which writes it to disk. Once the delivery is completed, the application is notified through a `DeliveryComplete` event.

Meta-Tag	**Selection Predicate**	**Delivery Plug-in Meta-Data**			**Result**
		HTTP	**BitTorrent**	**Limewire**	
Estimated_ Download_Rate	> 988Kbps	1400Kps	500Kbps	1000Kbps	HTTP, Limewire
Esimated_ Upload_Rate	< 392Kbps	0Kbps	500Kbps	400Kbps	HTTP
Min_File_Size	<=72MB	0MB	20MB	4MB	All
Max_File_Size	>=72MB	∞ MB	∞ MB	5GB	All

Table 5. Predicates and Meta-Data for Delivering a 72MB File to Node LC

Table 5 shows an example set of selection predicates used by Node LC when requesting the cartoon. The first predicate is the estimated download rate; the application requires the file within 10 minutes and therefore it simply issues the required download rate to achieve this. The node, however, has limited resources (only 784Kbps upload capacity) and therefore the Configuration Engine also introduces the <392Kbps upload predicate. This is simply a parametric default defined by a policy rule indicating that the maximum upload must be 50% of the node's capacity. The next predicates define the file size; these indicate the minimum and maximum file sizes that the plug-in is suitable for. A number of other predicates can also be used such as reliability, latency, resilience and memory overhead. It is hoped that this ontology will be extended to also incorporate many other aspects.

4.1.4 Overhead Study and Discussion

Table 6 shows the dynamic memory footprint of the possible Juno configurations detailed in this section (it does not include the application or the JVM). It also gives configuration time, including application-middleware interaction, the selection process and plug-in initiation. It does not include any delivery system bootstrapping as this varies greatly between plug-ins. The measurements have been taken on an Intel Core 2 Duo 2.1GHz PC with 4GB RAM; running Ubuntu 8.04; OpenCOM v1.4; and the OpenJDK JRE 1.6. These figures constitute the average results over 10 tests.

The above figures highlight that there is only a limited quantity of overhead related to running an application over Juno. The memory overhead of maintaining Juno's framework (e.g. Configuration Engine) is only 472KB and the average time taken for Juno to select and attach these plug-ins is only 366ms. We consider these values to be small when compared to the performance benefits previously detailed. This data is also made available through the meta-data of each plug-in, allowing low capacity devices to take resource consumption into account.

Configuration	Empty	HTTP	BitTorrent	Limewire
Memory Footprint	472 KB	514 KB	522 KB	573 KB
Configuration Time	329ms	357ms	374ms	369ms

Table 6. Runtime Memory Footprints and Configuration Times of Juno

In summary, this experiment has shown that *i)* there is a significant performance benefit by using non-statically assigned delivery schemes and *ii)* Juno effectively provides these benefits. Importantly, by placing content-centricity in the middleware layer, these benefits can be gained by individual clients without prior deployment. Juno is always capable of interacting with the optimal system and therefore the only potential loss is the memory overhead and configuration time.

4.2 Adaptive Delivery Reconfiguration

The second evaluative case-study investigates the mid-delivery reconfiguration of delivery plug-ins to react to variations in the environment. The specific scenario is a server providing content through HTTP which becomes overloaded. Juno detects this and reconfigures the provision scheme to BitTorrent allowing more effective delivery.

4.2.1 Case-Study Overview

Within this case-study there are a number of clients downloading a 698MB video from a single HTTP server. The server allocates 10Mbps to the provision of this content. Initially three clients begin to download the content, however, after 20 minutes a number of other nodes also request the content. These consist of 22 nodes requesting the content sequentially with 20 second intervals. This results in a considerable degradation of delivery quality as the server's resources become saturated. The distribution of bandwidth between the different clients is generated using real-world measurement data [3]. The average seeding time is taken from [17] with a 20% chance of nodes departing immediately after their delivery completes.

4.2.2 Performance Evaluation

Figure 3 shows the average gain, in terms of download time, of reconfiguration for each node when compared to non-reconfiguration. Nodes are ordered by their download/upload capacity with the slowest nodes at the left. Figure 3 (a) shows the circumstance in which the server mandates that all peers reconfigure so it can lower its own utilisation. It can be seen that lower capacity nodes actually suffer from the reconfiguration; this is because swarms form around the higher capacity nodes leaving lower capacity nodes struggling to achieve downlink saturation. Juno, however, resolves this by allowing fine grained per-node reconfiguration. This allows each node to individually select the optimal plug-in for itself. Figure 3 (b) shows the situation in which the server also provides the content simultaneously through HTTP allowing each node to select its own plug-in. A policy rule is therefore added to each node that rejects the reconfiguration request if the node's downlink capacity is below 2Mbps. On average, through this, peers complete their download 65 minutes sooner. This is an average saving of 30% with the highest saving being 51%. This is because high capacity peers are allowed to exploit each others' resources whilst low capacity peers can utilise the extra available server bandwidth. This cannot be ordinarily achieved with a traditional deployment as this type of fine-grained reconfiguration is not usually possible. Instead, it is necessary for all nodes to follow the same policy due to the complexities of coding such facilities in the application.

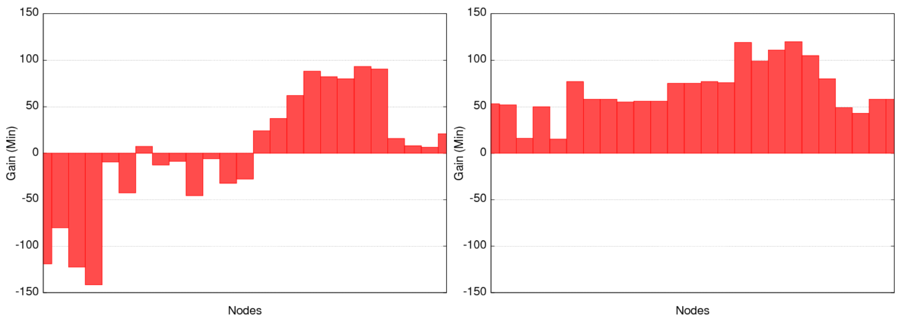

Fig 3. Benefit of Juno for *(a)* system-wide *(b)* per-node reconfiguration

Figure 4 (a) shows the measured download bandwidth of a representative Juno node possessing a 3Mbps/1Mbps down/upload capacity; the graph shows the bandwidth with reconfiguration enabled and disabled. This peer joins before the flash crowd and receives a steady bit rate of ~600Kbps early in its lifetime. However, after 20 minutes the flash crowd occurs, resulting in the degradation of its bit rate to ~350Kbps. Without adaptation of the delivery scheme, this situation remains until the flash crowd has alleviated (seen slightly after ~150 minutes). Juno, however, reconfigures 9 minutes after the beginning of the flash crowd. At first there is a significant drop in the bit rate to ~200Kbps; this occurs due to the bootstrapping period of BitTorrent. Importantly, however, the bit rate quickly returns to that of before the flash crowd (~700Kbps) with the usual end game period at the end [3].

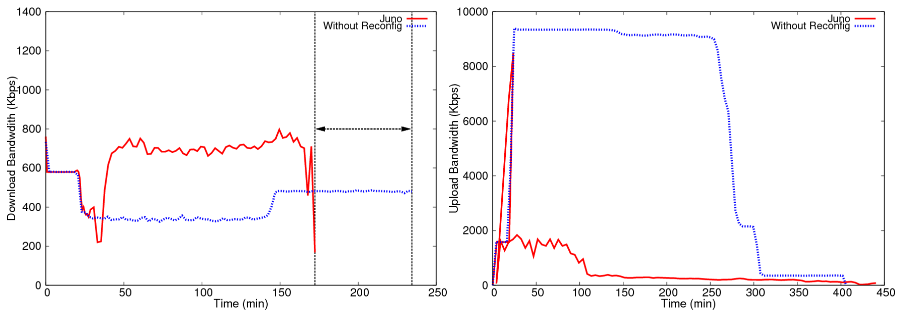

Fig 4. *(a)* Download rate of Client *(b)* Upload rate of Server

Figure 4 (b) shows the circumstance in which the server enforces system-wide reconfiguration to reduce its own utilisation. It shows the upload rate of the server with both reconfiguration enabled and disabled. It can be seen that the loading on the server increases rapidly in proportion to the number of nodes requesting the content. Without reconfiguration, this results in a high level of server utilisation. However, with reconfiguration enabled, the server reacts and adapts appropriately to its new operating environment. This can be seen by the rapid fall in utilisation. Over a short period of time the server remains seeding chunks to the BitTorrent swarm. However, after 105 minutes this process largely ceases with over 95% of bandwidth being freed.

4.2.3 Overview of Juno's Behaviour

From Juno's perspective (at the server), the increase in clients represents a change in its environment. Juno therefore addresses this by reconfiguring the delivery to utilise a more appropriate plug-in for the current requirements. For clarity, this process is now outlined from the perspective of a provider trying to reduce its utilisation:

1) The provider's bandwidth becomes saturated; this constitutes an environmental change. Information from the upload bandwidth monitor is compared against reconfiguration policy rules by the Configuration Engine.
2) One particular rule is trigged when the provider's upload bandwidth is saturated by a certain threshold (94% of maximum network throughput) for longer than 9 minutes. When triggered, this rule modifies the meta-description. The new meta-description stipulates that the delivery plug-in should export the meta-data, "upload_scalability>4". This refers to the plug-in's ability to scale up with greater clients; it is a heuristic metric from 1-5 (1 is low, 5 is high).
3) The Configuration Engine compares the new meta-description to the meta-data offered by the available plug-ins. HTTP's meta-data for upload_scalability is only 2 making it unsuitable. The only plug-in with an upload_loading meta-data tag of 5 is BitTorrent due to its ability to handle flash crowds [22].
4) The StoredDeliveryState object is retrieved from the HTTP plug-in and passed into the BitTorrent plug-in. This is a generic state object containing information about the current consumers being served.
5) The Configuration Engine observes that the HTTP service has consumers. A request is created for each connected client listed in the StoredDeliveryState object and passed to the DistributedConfigurationCoordinator. This request details the original plug-in used by the provider; the new plug-in that should be used by the consumer; and information for bootstrapping of the consumers.
6) Each client receives the configuration request; the new required plug-in (BitTorrent) is then compared against the current meta-description to ensure that there isn't a conflict with the application/environmental requirements.
    a. If there is not a conflict, the HTTP plug-in is removed and the BitTorrent plug-in is attached to the Delivery Framework.
    b. If there is a conflict the request is rejected and an alternate source must be utilised (usually located through the Discovery Framework).
7) The server's HTTP plug-in is removed using its `detach` method. The BitTorrent plug-in is initiated and connected using its `execute` method.
8) Each consumer's Delivery Framework resumes the delivery by calling the new delivery plug-in's `resume` method. BitTorrent then contacts the newly created tracker operating on the provider and the delivery continues.

4.2.4 Overhead Study and Discussion

Table 7 outlines the overhead of the case-study. The reconfiguration time is the length of time between the removal of the HTTP plug-in and the generation of the first BitTorrent chunk request. Alternatively, the bootstrapping time is the subsequent time it takes to receive the first chunk of data. Unlike the previous overhead details, these measurements are taken from a number of nodes operating in Emulab during the experiment, each possessing a large variety of capabilities; for this reason the average, maximum and minimum are shown. The Reconfiguration time is much longer than shown in Section 4.1.4 because these details also include distributed interactions and the generation of temporary data files to write chunks to.

	Average	**Maximum**	**Minimum**
Reconfiguration	29 sec	42 sec	13 sec
Bootstrapping	6 sec	18 sec	3 sec

Table 7. Reconfiguration and Bootstrapping Time for Clients

At the server side, the reconfiguration time is much shorter (12 seconds) as it executes on a higher specification node (c.f. Section 4.1.4) and does not require any distributed interactions. It only initiates itself and waits for requests. It takes much longer than the time shown in Section 4.1.4 (302ms) because this also includes the calculation of hash values for each chunk.

This experiment has shown that it is possible for runtime variations in a delivery's environment to be detected and responded to. Juno's approach of distributed reconfiguration allows nodes to not only alert each other to changes but also to coordinate their responses. An area of future work is looking at more sophisticated negotiation techniques in which multiple parties can decide upon actions. Lastly, the experiment has also verified the previous results indicating that optimisation can often only be gained by fine grained per-node decisions (which Juno fully supports).

4.3 Proactive Delivery Reconfiguration

The third case-study looks at how Juno exploits changes in the environment to proactively improve performance. Specifically, a superior source is located by the Discovery Framework during a delivery. However, this source is incompatible with the already downloaded BitTorrent data as it only offering the content using HTTP (which does not support the concept of chunks). Juno therefore requests the new source to reconfigure to also offer the content using BitTorrent.

4.3.1 Case-Study Overview

In this case-study a high capacity node is downloading a 698MB video using BitTorrent at ~2.5Mbps. This node is resident on a campus network with a 100Mbps connection to the Internet. The node's user has an account with a content distribution network (CDN) that provides content using a number of strategically placed replication servers. When the application initiates the download, the CDN does not possess a copy of the desired content. However, 20 minutes after the download starts, the CDN acquires a replicated copy of the desired content. At this point in time, this replication server has ~6.5Mbps of available upload capacity. The server is only offering HTTP access at this point in time and therefore requires reconfiguration.

4.3.2 Performance Evaluation

Figure 5 (a) shows the download bandwidth of the client with Juno. Initially the client is limited to utilising a BitTorrent swarm; this provides a stable rate of ~2.5Mbps. However, after 20 minutes, the new source is discovered offering ~6.5Mbps of upload capacity. After reconfiguration, the download rate increase to ~6.5Mbps. This allows the delivery to complete 18 minutes earlier, achieving a 36% faster delivery

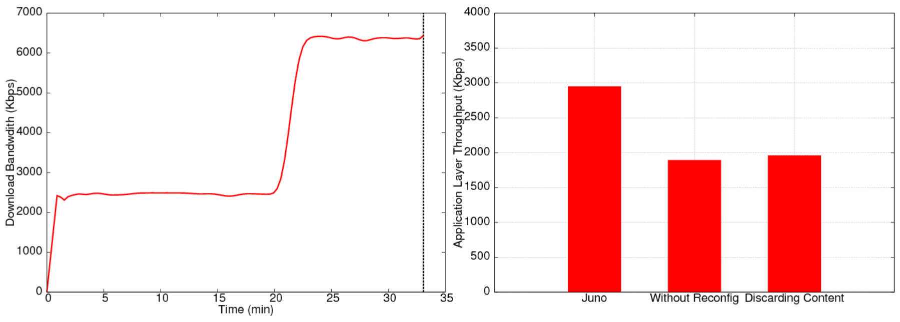

Fig 5. *(a)* Download Bandwidth of Juno Node *(b)* Application Level Throughput

In Figure 5 (b) the application layer throughput is also shown; this is measured from the point at which the application requests the content to the point at which it is fully received. It can be seen that Juno's throughput (with reconfiguration) is significantly higher than without reconfiguration. This is due to the higher download rate achieved after the first 20 minutes. These two results can also be compared to the alternative of simply discarding the original data retrieved through BitTorrent and starting a new HTTP delivery from the replication server. By discarding the first 20 minutes of chunk-based delivery (286MB), this approach's throughput drops to 66% of Juno's, resulting in a 17 minute longer delivery. Other undesirable side effects also occur such as increased client bandwidth utilisation. Further, in today's systems this would not be automated, instead the user would be required to cancel and delete one download before initiating a new client and starting the download again.

4.3.3 Overview of Juno's Behaviour

This process is similar to the previously described case-studies and therefore only a brief overview is given. After 20 minutes the Discovery Framework returns the availability of a replica in the CDN. This is done using a web service plug-in that can query the CDN. Traditional applications often have to build extra functionality (such as proxy support) to allow access to such infrastructure whilst, conversely, CDNs have to build increasingly complicated ways in which users can transparently access their data. Juno remedies these concerns by providing an abstracted framework in which CDNs can easily deploy plug-ins to interact with their infrastructure. After this discovery, the Configuration Engine re-executes the selection process. This process yields the replication server; however, this is not feasible to use as the server currently only offers HTTP. This invalidates the reconfiguration policy dictating that chunk based deliveries cannot be reconfigured into range based deliveries (in the worst case scenario, to do so would result in over 1500 HTTP range queries to emulate chunk requests). Ordinarily this would mean that the client would have to either begin a fresh download, discarding the previously downloaded data, or alternatively simply proceed with the original scheme, ignoring the superiority of the new source. In contrast, Juno handles this problem through distributed reconfiguration. To achieve this, the client sends a request to the replication server, asking it to offer the content through BitTorrent. As the server is a paid service it accepts the request and instantiates BitTorrent to offer the item of content using the process outlined previously. Following this, the client can immediately begin a client-server delivery from the replication server using the BitTorrent protocol.

4.3.4 Overhead Study and Discussion

The reconfiguration time at the server side has been measured using the same node specification as described in Section 4.1.4. It takes the server 11 seconds to reconfigure; this includes the attachment of the BitTorrent plug-in and the time taken to calculate the hash values for each chunk in the file. In contrast, the clients see no reconfiguration time as they are already configured with BitTorrent and can therefore continue utilising the original swarm until the server's reconfiguration has completed.

This case-study has shown that Juno can exploit both positive as well as negative changes in its operating environment. Importantly, is has also highlighted how distributed adaptation can be exploited to elegantly resolve fundamental issues with delivery reconfiguration such as data storage formats. Lastly, it has also been shown how Juno can ease the deployment of new systems by providing a pluggable framework through which discovery and delivery schemes can transparently interoperate with applications.

5 Conclusion and Future Work

It has been shown that the current approach of utilising fixed, design-time selected delivery schemes is suboptimal considering the modern heterogeneity of content-oriented systems, as well as communication environments. Recent content-centric observations have indicated that this is an unnecessary burden to place on the developer as most applications do not have a vested interest in how or where their content is obtained from [19]. In order to provide content-centric discovery and delivery, a reflective middleware, *Juno*, has been designed, built and evaluated. Juno provides content management and delivery, on behalf of the application through a well-defined middleware abstraction. This allows dynamic delivery scheme selection to take place using architectural (re)configuration and flexible, extensible requirement descriptions. In order to validate Juno and evaluate the approach taken, three case-studies, investigating various properties of the middleware have been used. The case studies cover the relevant aspects of providing content-centric delivery at the middleware layer. Significant performance benefits have been found by allowing dynamic delivery selection and reconfiguration at runtime. It is also evident that as heterogeneity and complexity increases in content-based applications, the importance of this will grow. Unlike infrastructural or network level solutions, Juno is easily deployable and interoperable with existing diverse content systems. Importantly, Juno also recognises the importance of content delivery, offering specialised support for optimising this process. This is achieved by leveraging its ability to reconfigure in order to exploit the various capabilities of divergent delivery systems.

The feasibility and benefits of the Juno approach have been shown, however, a number of areas of future work exist. This centres of the distributed and reliable acquisition of source characteristics to improve the decision making process. Currently, there are a relatively limited number of standard meta-data tags and therefore it is important to expand these to allow much richer decisions to be made. Also, security, incentive and trust concerns must be investigated to look at how the bespoke mechanisms employed by each delivery scheme can be unified to allow true interoperability and re-configurability. Lastly, to deploy Juno in a diverse real-world setting will help to further validate the feasibility of the Juno approach.

References

[1] Akamai Content Distribution Network. http://www.akamai.com/

[2] Baldoni, R., Beraldi, R., Querzoni, L., and Virgillito, A. Efficient Publish/Subscribe through a Self-Organizing Broker Overlay and its Application to SIENA. The Computer Journal, vol. 50, 4 (2007)

[3] Bharambe, A. R., Herley, C., Padmanabhan, V. N. Analyzing and Improving a BitTorrent Networks Performance Mechanisms. In Proc. 25th Infocom, Barcelona, Spain (2006)

[4] 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 2. IEEE Distributed Systems Online 2, 6 (2001)

[5] BitTorrent Specification. *http://www.bittorrent.org/beps/bep_0003.html*

[6] Carzaniga, A; Rutherford, M.J., Wolf, A.L. A Routing Scheme for Content-Based Networking. In Proc. Infocom, Hong Kong, China (2004)

[7] Cooper, F., B. Trading Off Resources between overlapping Overlays. In Proc. 7th ACM/IFIP/USENIX Middleware Conference, Melbourne, Australia (2006)

[8] Coulson, G., Blair, G. S., Clarke, M., and Parlavantzas, N. The Design of a Configurable and Reconfigurable Middleware Platform. In Distrib. Comput. 15, 2 (2002)

[9] Coulson, G., Blair, G. Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan,T. A Generic Component Model for Building Systems Software. ACM Transactions on Computing Systems. 26, 1 (2008)

[10] eMule Protocol Specification. http://jmule.org/files/emule.pdf

[11] Esteve, C., Verdi, F. and Magalhaes, M.F. Towards a new generation of information-oriented internetworking architectures. In Proc. 1st Workshop on Re-Architecting the Internet, Madrid, Spain (2008)

[12] File Transfer Protocol Specification. http://www.ietf.org/rfc/rfc959.txt

[13] Fitzpatrick, T., Gallop, J. J., Blair, G. S., Cooper, C., Coulson, G., Duce, D. A., and Johnson, I. J. Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware Platform. In Proc. Workshop on Distributed Multimedia Systems (2001)

[14] The Gnutella Protocol v0.4. www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[15] Google Search Engine. http://www.google.co.uk

[16] Grace, P., Coulson, G., Blair, G.S., Porter, B. Deep Middleware for the Divergent Grid. In Proc. 6th ACM/IFIP/USENIX Middleware, Grenoble, France (2005)

[17] Guo, L., Chen, S., Xiao, Z., Tan, E., Ding, X., and Zhang, X. Measurements, analysis, and modelling of BitTorrent-like systems. In Proc. 5th ACM Conference on Internet Measurement, Berkeley, CA (2005)
[18] HTTP Protocol Specification. http://www.ietf.org/rfc/rfc2616.txt
[19] Jacobson, V., Mosko, M., Smetters, D., and Garcia-Luna-Aceves, J.J. Content-centric networking. Whitepaper, PARC (2007)
[20] Jerzak, Z. and Fetzer, C. Bloom Filter Based Routing for Content-Based Publish/Subscribe In Proc. 2nd Intl. Conference on Distributed Event-Based Systems, Rome, Italy (2008)
[21] Kaune, S., Lauinger, T., Kovacevic, A., and Pussep, K. Embracing the Peer Next Door: Proximity in Kademlia. In Proc. 8th Intl. Conference on Peer-To-Peer Computing (2008)
[22] Kaune, S.; Stolzenburg, J.; Kovacevic, A.; Steinmetz, R.; Cuevas, R. Understanding BitTorrent's Suitability in Various Applications and Environments. In Proc. 1st Intl. Workshop on Computational P2P Networks: Theory & Practice, Athens, Greece (2008)
[23] Kim, M, Won, Y.J, Hong, J.W. Application-Level Traffic Monitoring and an Analysis on IP Networks**.** In ETRI Journal. Vol. 27, no. 1 (2005)
[24] Koponen, T., Chawla, M., Chun, B., Ermolinskiy, A., Kim, K. H., Shenker, S., and Stoica, I. A Data-Oriented (and beyond) Network Architecture. In Proc. ACM SIGCOMM (2007)
[25] Ledlie, J. Gardner, P., and Seltzer, M. Network Coordinates in the Wild. In Proc. USENIX Networked Systems Design and Implementation, Cambridge, MA (2007)
[26] Limewire Technical Documentation. http://wiki.limewire.org/index.php?title=Techdocs
[27] Mauthe, A. and Thomas, P. Professional Content Management Systems - Handling Digital Media Assets. John Wiley & Sons (2004)
[28] Peng, G., Wang, J., Cai, H. ASAP: An Advertisement-based Search Algorithm for Unstructured Peer-to-peer Systems. In Proc. Intl. Conf. on. Parallel Processing (2007)
[29] 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 (2006)
[30] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S. A Scalable Content-Addressable Network. In Proc. SIGCOMM, San Diego, CA (2001)
[31] Rowstron, A., Druschel, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In Proc. Middleware, Heidelberg, Germany (2001)
[32] Särela, M., Rinta-aho, T., and Tarkoma, T. RTFM: Publish/Subscribe Internetworking Architecture. ICT Mobile Summit, Stockholm, Sweden (2008)
[33] Saroiu, S., Gummadi, K. P., Dunn, R. J., Gribble, S. D., and Levy, H. M. An Analysis of Internet Content Delivery Systems. In Proc. USENIX OSDI, San Francisco, CA (2002)
[34] Scott, D. Improving Channel Zapping. B.Sc Dissertation, Lancaster University (2009)
[35] 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)
[36] White, B., Lepreau, J., Stoller, L., Ricci, R. Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. In Proc. Operating Systems Design and Implementation, MA (2002)
[37] Tyson, G.**,** Mauthe, A., Kaune, S., Mu, M., Plagemann, T. Corelli: A Dynamic Replication Service for Supporting Latency-Dependent Content in Community Networks. In Proc. 16th Multimedia Computing and Networking, San Jose, CA (2009)
[38] Zhang,.X, Liu, J., Li, B., and Yum, T.S.P. CoolStreaming/DONet: A Data-driven Overlay Network for Live Media Streaming. In Proc. Infocom, Miami, FL (2005)