A Juno: A Middleware Platform for Supporting Delivery-Centric Applications

Article *in* ACM Transactions on Internet Technology · December 2012

DOI: 10.1145/2390209.2390210

6 authors, including:

Gareth Tyson
Queen Mary, University of London
60 PUBLICATIONS **412** CITATIONS
SEE PROFILE

Andreas Mauthe
Lancaster University
162 PUBLICATIONS **1,038** CITATIONS
SEE PROFILE

Adel Taweel
Birzeit University
89 PUBLICATIONS **429** CITATIONS
SEE PROFILE

Thomas Plagemann
University of Oslo
126 PUBLICATIONS **852** CITATIONS
SEE PROFILE

A

Juno: A Middleware Platform for Supporting Delivery-Centric Applications

Gareth Tyson, King's College London

Andreas Mauthe, Lancaster University

Sebastian Kaune, Technical University of Darmstadt

Paul Grace, Lancaster University

Adel Taweel, King's College London

Thomas Plagemann, University of Oslo

This paper proposes a new delivery-centric abstraction, which extends the existing content-centric networking API. A delivery-centric abstraction allows applications to generate content requests agnostic to location or protocol, with the additional ability to stipulate high-level requirements regarding such things as performance, security and resource consumption. Fulfilling these requirements, however, is complex as often the ability of a provider to satisfy requirements will vary between different consumers and over time. Therefore, we argue that it is vital to manage this variance to ensure an application fulfils its needs. To this end, we present the *Juno* middleware, which implements delivery-centric support using a re-configurable software architecture to (*i*) discover multiple sources of an item of content; (*ii*) model each source's ability to provide the content; then (*iii*) adapt to interact with the source(s) that can best fulfil the application's requirements. Juno therefore utilises existing providers in a backwards compatible way, supporting immediate deployment. This paper evaluates Juno using Emulab to validate its ability to adapt to its environment.

Categories and Subject Descriptors: C2.4 [**Computer- Communication Networks**]: Distributed Systems—*Distributed Applications*

General Terms: Design, Experimentation, Management, Performance, Standardization

Additional Key Words and Phrases: Middleware, content delivery, content-centric

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 [Schulze and Mochalski 2009]. It is envisaged that in the future a fully integrated infrastructure will replace various proprietary and heterogeneous content delivery systems [Plagemann et al. 2006]. Currently, however, this is not available; instead, a large number of

independent content providers and protocols exist. These have been built to address particular requirements and often offer content in fundamentally different manners. For example, some offer stored delivery [Cohen 2003][Fielding et al. 1999] whilst others offer streamed delivery [Zhang et al. 2005]. Similarly, non-functional aspects such as performance, reliability, scalability and resource consumption also vary heavily. Due to this, the suitability of a given provider will vary heavily between different applications. For example, a security critical application could not use an unencrypted protocol such as HTTP, whilst a streaming application could not use a non-linear delivery protocol such as BitTorrent. Hence, it is the responsibility of the developer to statically select (during the design period) how to best access content based on how well they consider a given delivery protocol and provider satisfies their own requirements. Unfortunately, however, these requirements are often complex and diverse, with dynamic elements that cannot be properly analysed at design-time (e.g. the performance of a provider will generally vary over time). We therefore argue that this static design-time selection of delivery options is an inefficient approach considering the developer's true needs — instead of wishing to utilise a particular delivery protocol to connect to a given provider, the developer, in fact, wishes to simply gain access to a unique item of content within certain requirement constraints. As such, we posit that developers should be liberated from statically managing these requirements, allowing runtime decisions to be made based on operating conditions.

To this end, we propose the *Juno* middleware, which implements a new *delivery-centric* API (extending the traditional content-centric interface [Demmer et al. 2007]). Our delivery-centric API allows applications to issue requests for uniquely identified items of content, alongside diverse delivery requirements that place constraints on how the content is provided. Through this, developers are liberated from statically managing these requirements, empowering Juno to select optimal delivery mechanisms at runtime. To achieve this delivery-centricity, Juno exploits the previously discussed diversity of providers and protocols in the Internet. Specifically, for each item of requested content, Juno attempts to discover multiple content sources that each possess divergent characteristics, e.g. different protocols, qualities of service etc. Juno then exploits this diversity to dynamically select and (re-)configure between the sources that best fulfil the application's needs. Importantly, by performing this function on a per-node basis, Juno can specialise each node's delivery to handle any vari-

ance that can be observed both over time and between different consumers. Consequently, applications can use Juno to delay their content access decisions until the point of request, thereby removing the need to statically make decisions that may later become suboptimal. Thus, unlike previous content delivery work (e.g. [Su and Kuzmanovic 2008][Zhang et al. 2005][Cohen 2003]), we attempt to exploit the diversity of existing infrastructure and protocols rather than building a one-size-fits-all approach.

In this paper, we build on our previous work on Juno. In [Tyson et al. 2008] we provided a preliminary architectural design of the Juno middleware, which this paper adapts significantly. Alongside this, in [Tyson et al. 2012], we presented a brief overview of some of the main components in Juno's design; this work is now extended by providing a detailed design description of all components involved, alongside an investigation of variance and a system evaluation. Specifically, the contributions of this work are as follows:

— The identification and validation of content delivery variance, alongside its effect on content delivery performance.
— The formalisation of a new *delivery-centric* API, which extends existing content-centric APIs to allow applications to associate delivery requirements with each content request.
— The design, implementation and evaluation of a middleware system, *Juno*, which realises the delivery-centric API to allow per-request adaptation to satisfy delivery requirements.

The rest of the paper is structured as follows. In Section 2 we provide a background to the problem space, before performing an analysis of content system variance in Section 3. Section 4 then defines the delivery-centric API, whilst Section 5 details the Juno middleware. Following this, Section 6 evaluates the approach. Last, Sections 7 and 8 describe the related work in the field and conclude the paper.

2. BACKGROUND

This section provides a background to the issue of content distribution. First, we investigate the existing content distribution paradigm before exploring its key limitations and inspecting emerging trends in the field.

2.1. Current Content Distribution Paradigm

Modern application development increasingly involves the use of content. This can range from streaming videos to the distribution of software updates. Generally, most applications utilise statically-selected generic toolkits to offer this necessary support. A simple example is the use of a web server to publish software updates. To achieve this, an organisation will acquire the necessary resources to host a web server (or use third party servers) then integrate a HTTP toolkit into their client software.

Within this paper, we term a content source as a *provider*; this could range from an individual HTTP server to a BitTorrent swarm. Alongside these, a variety of other alternatives are possible with current mainstream delivery schemes including cloud services [Palankar et al. 2008], peer-to-peer networks [Bharambe et al. 2006][Zhang et al. 2005] and various third party content hosts [Antoniades et al. 2009]. All of these, however, follow the same process of (*i*) *publication:* making the content available; (*ii*) *consumer discovery:* allowing consumers to discover sources of the content; and (*iii*) *consumer delivery:* allowing consumers to gain access to the content. Vitally, the bespoke and non-standardised nature of these systems mean that selection and integration must be performed statically at design-time with little support for the future adaptation of any decisions made.

2.2. Limitations of the Existing Content Distribution Paradigm

We focus in this paper on one key limitation of the existing content distribution paradigm: the lack of *per-node (re-)configurability*. This occurs because most applications are currently developed using a fixed statically selected content distribution mechanism. This means that it is impossible for such an application to be configured or re-configured to adapt to future runtime changes regarding this choice. The need to adapt might arise for a number of reasons that generally occur due to some sort of environmental change. For instance, a key requirement of many content distribution strategies is high performance; this could be impacted by a number of (runtime) events, for example:

— It is possible for *protocol characteristics* to introduce unpredictable behaviour that can only be resolved post-deployment. For instance, an application that chooses to utilise BitTorrent will only gain high performance if its host has sufficient upload capacity to

compete in the swarm [Bharambe et al. 2006]. If it does not, BitTorrent will become a highly sub-optimal choice. A statically configured application could therefore not react to this, as it can only be measured post-deployment (based on a comparison of each individual host's upload capacity against the swarm).

— It is possible for *infrastructural characteristics* to introduce unpredictable behaviour that can only be resolved post-deployment. For instance, a HTTP server might be re-located or suffer from resource changes (e.g. an upgrade/downgrade). Similarly, the way this impacts different consumers will vary; for example, if a server is moved nearer to a set of consumers, performance will increase due to the likely lower packet loss and delay. This also means that the same protocol (or similar ones such as HTTP and FTP) can display totally different behaviour based on the infrastructure it is running over.

— It is possible for *new providers* to become available after an application has been deployed, as well as old providers to become unavailable. A statically configured application could therefore not handle this. For instance, a mobile host might witness providers come and go frequently; alternatively, more practical aspects such as route failures or firewalls might create similar effects. This is particularly difficult to manage if the new providers use protocols that the application does not already have support for.

The above situations are examples of *variance*; we define variance as any environmental change that might alter a given provider's ability to satisfy the requirements of a consumer (e.g. performance, security, reliability etc.). Variance can be separated into a variety of sub-categories based on a range of different factors. However, within this paper, we group types of variance into two logical classifications:

— *Consumer Variance:* This is the observation that the ability of a given provider to satisfy certain requirements will often vary from the perspectives of different consumers. For example, a HTTP consumer that is more distant from a source will generally get inferior performance to a consumer which is nearer [He et al. 2007]. Consequently, it is important that consumers independently select the optimal means by which they access content.

— *Temporal Variance:* This is the observation that the ability of a given provider to satisfy certain requirements will often vary over time, even from the perspective of a single consumer. For example, a BitTorrent swarm's performance will generally degrade over time

due to population decay [Kaune et al. 2010]. Consequently, it is important that individual consumers can adapt previous choices to reflect new operating conditions.

Both of these observations mean that static decisions regarding how to distribute content can often later become suboptimal. This is further exacerbated by the possibility for application requirements to change over time, thereby potentially invalidating previous choices. The current ad hoc way in which content support is integrated means that there are no mechanisms to easily allow these changes to be addressed without extensive effort and recoding. Further, the fine grained per-node basis at which these changes can occur (due to consumer variance) means that system-wide software modification will also likely result in further sub-optimality. Building such support into applications, however, has not yet been investigated due to the high complexity. This is exacerbated by the fact that most application developers do not possess a vested interest in content distribution; instead, they simply wish to utilise simple mechanisms that allow them to focus on their core goals.

2.3. Emergent Content Distribution Trends

Since the inception of multiple divergent content distribution schemes, primarily fuelled by peer-to-peer and cloud technologies, many organisations have begun to decentralise the way in which they deliver content, using a range of different mechanisms. This is particularly prevalent in web environments, which often see users presented with a number of different delivery options, e.g. a range of 'mirrors' using different protocols. For instance, Linux ISOs can be accessed through a wide-range of mediums including HTTP, FTP and BitTorrent, to name a few. Various studies have investigated this phenomenon; for example, Ager et al. [Ager et al. 2011] found that content is typically replicated across many ASes. Similarly, Antoniades et al. [Antoniades et al. 2009] also found that various content provided through RapidShare (HTTP/HTTPS) is also offered via BitTorrent. Publishing content through multiple means is a way of addressing the previously discussed forms of variance. For instance, providing geographically distributed mirrors allows forms of consumer variance to be addressed by selecting nearby sources (mitigating the different TCP delays of users); whilst, offering scalable peer-to-peer alternatives allows forms of temporal variance to be addressed by scalably handling peak demands. However, as previously mentioned,

currently, the complexity of this must be handled either by the application or the user. This is particularly difficult in the face of the above types of variance.

This paper posits that it is undesirable to force applications to make design-time decisions regarding which providers and protocols it uses to access content. Instead, this emerging diversity of providers (in terms of both infrastructure and protocols) should be exploited based on whatever runtime conditions a consumer observes. Consumers should therefore be given the necessary knowledge and understanding to dynamically select the best provider based on their protocol and infrastructural characteristics. In this paper, to achieve this, a new development paradigm (which extends the concept of content-centricity [Demmer et al. 2007]) is proposed, alongside a middleware that allows this variance to be effectively addressed without complex development on the part of the designers.

3. UNDERSTANDING VARIANCE

Before inspecting the solution space, it is vital to explore the existence of consumer and temporal variance in real-world content distribution. This section first provides a discussion of variance before detailing its existence in some of the key protocols in use today.

3.1. Modelling Variance

Variance can be understood as the runtime variation of certain parameters that impact the ability of a given provider to satisfy the requirements of a consumer. These *variance parameters* could relate to any component involved in the content distribution process, including the protocol, the consumer, the provider, the network or the content. To exploit variance it is therefore first necessary to define a function $x(r, p, c, o)$, which allows a consumer to calculate provider p's ability to serve a content request for object o from consumer c in a way that satisfies requirement r. To compute this function, it is necessary to collect runtime measurements of these variance parameters, as defined by the delivery protocol(s) supported by the provider. These measurements could be locally observed, predicted or acquired from remote information sources (e.g. through a web service). Each $\langle r, p, c, o\rangle$ tuple will therefore be dependent on one or more variance parameters, which can then be dynamically collected to compute the fulfilment of r. For instance, if $\langle r, p, c, o\rangle$ relates to the performance of a consumer accessing an object using HTTP, it would be necessary to collect measurements

on link packet loss and delay (at least), which could then be used to calculate predicted throughput [Padhye et al. 1998]. Lastly, in-line with previous discussion and for convenience, we categorised these parameters into two inter-related[1] groups: consumer and temporal.

3.2. Exploring Variance in Content Distribution

This section briefly applies the above principles to three key content distribution technologies currently in use: HTTP, BitTorrent and Content Distribution Networks (CDNs). The purpose of this is to highlight the forms of real-world variance that any solution will need to handle. To achieve this, we focus on the most popular requirement: performance ($r = perf$).

3.2.1. HTTP. HTTP is the predominant web distribution protocol, as well as having the second heaviest traffic profile after peer-to-peer [Schulze and Mochalski 2009]. Like many client-server protocols, it is built over TCP, thereby taking on many of its characteristics.

Consumer Variance in HTTP is highly prominent; two variance parameters that are of particular importance are *delay* and *packet loss*. Delay will vary significantly between different $\langle p, c \rangle$ pairs due to the potentially geographically distributed nature of consumers, as well as the significant differences in different region's network infrastructure (e.g. Africa has much larger delays than Europe [Kaune et al. 2009]). Generally, this will result in significantly different performance levels for these various consumers, particularly when using delay-based congestion algorithms (e.g. Reno [Afanasyev et al. 2010]) or performing small transfers [Krishnan et al. 2009]. Similarly, different packet loss rates between these various $\langle p, c \rangle$ pairs will also result in large performance variations due to TCP's interpretation of packet loss as congestion [Padhye et al. 1998]. Importantly, these variations are the norm, particularly when comparing different access technologies, e.g. 802.11, DSL, UMTS etc.

Temporal Variance in HTTP is also very typical; interestingly, the above parameters will similarly vary with time as well as between different consumers. A specific temporal variable, however, is *provider load*, which helps define a provider's available resources. This is highlighted well by [Antoniades et al. 2009] through RapidShare measurements: using a single measurement site, they found that the download rates ranged from as little as 1 Mbps to over 30 Mbps due to loading, with a 50:50 split between those achieving under 8

[1]Many variance parameters (e.g. packet loss) cause both consumer and temporal variance.

Mbps and those achieving more. Interestingly, these often follow patterns showing that users accessing content between 12-2PM or 6-9PM, for instance, will suffer higher competition for provider resources [Yu et al. 2006].

Put simply, the presence of these variance parameters will mean that the runtime performance of a HTTP provider will vary heavily over time and between different consumers. For example, imagine a provider p and a set of consumers $C = \{c^1, c^2 ... c^n\}$. Clearly, the performance of $\langle p, c^1 \rangle$ will differ from $\langle p, c^2 \rangle$ if the network characteristics of $c^1 \rightarrow P$ and $c^2 \rightarrow P$ also differ. Beyond this, as the loading of p changes, the performance will change on a temporal dimension. Consequently, when faced with multiple HTTP providers, each consumer must therefore compute the optimal based on these individual characteristics.

3.2.2. BitTorrent. BitTorrent is by far the most popular peer-to-peer distribution protocol in use, constituting up to 80% of peer-to-peer traffic [Schulze and Mochalski 2009].

Consumer Variance in BitTorrent is a typical observation; the most dominant consumer variance parameter is the host's upload resources [Piatek et al. 2007]. This will vary heavily between different consumers, with studies showing upload capacities in BitTorrent ranging from ≈300 Kbps to in excess of 30 Mbps [Idal et al. 2007]. Thus, due to tit-for-tat [Cohen 2003], the individual performance a consumer receives from a BitTorrent swarm will vary heavily based on this, i.e. hosts exceeding the swarm average will see notable benefits, whilst those falling below will achieve the opposite [Rasti and Rejaie 2007].

Temporal Variance in BitTorrent is also very usual; the most obvious temporal variance parameter is the seeder:leecher (S:L) ratio, which varies hugely over time. Typically, young torrents will have high S:L ratios that degrade over time; in fact, this degradation even leads to 64% suffering from intermittent content unavailability (i.e. when no seeders are present) [Kaune et al. 2010]. The S:L is very important because it largely dictates the level of resource competition in a swarm. Specifically, a torrent, t, can be identified as having a particular service capacity (up^t), which is the aggregate of available upload capacities from all members (seeder and leechers). This can therefore be compared against the service requirement ($down^t$) to calculate the competition over resources: $C = min\left(\frac{up_T^x}{down_T^x}, 1\right)$. Theoretically, if $C = 1$, all consumers will be able to saturate their connections, however, if, as often is the case, $C < 1$, some saturation percentage will be achieved; for example, on average, a S:L ratio of 0.78 achieves 61% download saturation [Tyson 2010].

Put simply, the presence of these variance parameters will mean that the runtime performance of a BitTorrent swarm will vary heavily over time and between different consumers. For example, two peers with different upload capacities will get vastly different performance, even when operating in the same swarm. Whilst, peers that join at different stages in a swarm's lifecycle will encounter vastly different levels of resource availability based on the current S:L ratio (e.g. older torrents will likely have fewer seeders). Consequently, when faced with multiple BitTorrent providers, it becomes necessary to be able to dynamically switch between the optimal based on the observed characteristics of each.

3.2.3. Content Distribution Networks. Content Distribution Networks (CDNs) are used to deliver content on a large-scale. Akamai, for instance, is a widely deployed CDN that maintains a significant market share (64% [Huang et al. 2008]), claiming to have over 56,000 edge servers, distributed in over 70 countries. It acts as an augmentation to existing (web) content hosts by placing their content on its distributed edge servers. When a provider that uses Akamai receives a request, it can optionally redirect it into the Akamai network, which will then attempt to redirect the consumer to the 'optimal' edge server. The main protocol used by Akamai is HTTP and therefore, when considering an edge server as a provider, the same parameters discussed in Section 3.2.1 apply. However, it varies significantly from the previous examples because Akamai also attempts to address variance by intelligently redirecting consumers between different edge servers. Thus, in contrast to the above, a key question is what are the limitations of a CDN's approach to handling variance?

In essence, CDNs attempt to deal with HTTP variance by minimising delay and improving available bandwidth. It is undeniable that performance is improved over a single HTTP server, however, it is evident that variance is only mitigated, it is not fully addressed. For example, when looking at CDN delays, Krishnan et al. [Krishnan et al. 2009] found that over 20% of clients witness, on average, 50 ms greater delay than other clients operating in the same geographical location. Further, it was also found that 40% of clients suffer from over 200 ms delays even when accessing content provided through CDNs such as Google. Thus, whilst a CDN does improve performance over traditional HTTP, it clearly does not resolve the phenomenon of variance. This is primarily because CDNs like Akamai mitigate such variance in a backwards compatible, provider-driven manner (i.e. DNS redirection). Therefore, the business model moves towards empowering providers rather than consumers

(e.g. a provider chooses when to allow a consumer to utilise Akamai). Instead, we believe this functionality should be embedded in the consumer, which is in a better position to make such decisions. Further, by decentralising the responsibility, far more complicated sets of requirements can be handled. Beyond this, CDNs limit consumers to only handling variance that has been specifically chosen by them; for instance, the Limelight CDN hosts content only at a small number of sites, consequently, not mitigating connection delays for geographically distributed consumers. Perhaps more importantly, it can also be observed that only a small minority of HTTP providers actually use CDNs, meaning that the majority of providers will not benefit. These providers are also often the worst resourced; for instance, the University of Washington probed a set of 13,656 web servers to discover that more than 90% had under 10 Mbps upload capacity [Padmanabhan and Sripanidkulchai 2002].

3.3. Summary

The above sections have sought to highlight the real-world existence of both consumer and temporal variance. Clearly, it has been shown that a variety of prominent protocols and systems suffer from both forms of variance, even those such as Akamai that attempt to mitigate it. Specifically, we have shown that:

— Two consumers may witness different performance levels from a given provider based on key variance parameters, e.g. the packet loss rate between a HTTP client and server (*consumer variance*).
— The performance a consumer receives from a provider will vary over time as these variance parameters change, e.g. the S:L ratio of a BitTorrent swarm (*temporal variance*).

A further observation is that both of these concerns can *only* be observed (and handled) dynamically. Consequently, it would be extremely difficult to handle such issues through the static design-time selection of providers and delivery strategies. Thus, to address this, we argue that it is necessary to liberate applications from such choices and allow request-time decisions to be made on a per-node basis, instead. Each node should therefore individually resolve its own requirements at request-time and then make an appropriate choice regarding how to access the content based on the available sources.

4. THE DELIVERY-CENTRIC PARADIGM

So far, it has been identified that it is difficult to optimise content access using static design-time decisions due to variance. We therefore consider it integral to make content access (in terms of sources and protocols) an explicit runtime decision that can be dynamically re-configured based on operating conditions. To liberate applications from such responsibilities, we therefore propose a new API that can provide a simple interface for requesting content, whilst offering the above support. Such an interface should allows applications to (*i*) generate content requests using unique identifiers that do not pre-define the access mechanism or source (e.g. unlike a URL); (*ii*) issue computable requirements that abstractly define how the content should be accessed; and (*iii*) receive content in a way that is agnostic to how it has been acquired. We term such an interface *delivery-centric*, extending the previously defined content-centric interface [Demmer et al. 2007], which does not support the stipulation of requirements. First, we describe how requirements can be stipulated before providing an overview of the interfaces for providing and consuming content.

4.1. Modelling Delivery Requirements

To enable the content delivery process to be (re-)configured at runtime, it is necessary for the application to be able to represent its requirements computationally. These requirements can vary from performance issues to far more diverse aspects relating to things such as security, monetary cost, overheads and resilience. Requirements are presented to the interface in the form of selection predicates, which we term *rules*. A rule is defined by an $\langle attribute, comparator, value \rangle$ tuple. The attribute value must adhere to an extensible *requirements ontology* exported by the underlying API implementation, whilst the comparator can be $=$, $>$, $<$, *min* or *max* (it is also possible to plug new functions in). For instance, a rule 'avg_bit_rate >= 500 Kbps' indicates that the underlying method of delivery must achieve a download rate of at least 500 Kbps. Subsequently, the requirements are stipulated through a set of these rules bound by a logical AND, i.e. $R = \{rule^1, rule^2 ... rule^n\}$.

4.2. Interface Definitions

There are two aspects of a delivery-centric system: provision and consumption. Within this paper, these are represented by two interfaces, IProvider and IConsumer. This section provides a summary of them; a formal specification can be found in [Tyson et al. 2012].

4.2.1. IProvider. The delivery-centric IProvider interface is presented to publishers that wish to distribute their content and consists of two methods, as detailed in Table I. The first method, **put**, allows an application to publish an item of content. It accepts a reference to the data alongside a set of rules (as defined above). A unique content identifier is generated and then returned (the rest of this paper is based on the use of hash-based identifiers that are generated from the content's data). At a future point, the application can also call the **remove** method, which will un-publish an item.

Table I. IProvider Definition

Method	Description
put	This method allows an application to publish an item of content. It accepts a reference to the data, alongside a set of rules.
remove	This method allows an application to withdraw an item of content from publication.

The current Juno implementation supports the following requirements ontology for IProvider: 'encrypted:boolean', 'type:String', 'local_upload:long', 'local_hosting:bool'. The 'type' refers to the method of provision, either streamed or stored (i.e. file download). The 'local_upload' refers to the acceptable number of bits that can be uploaded from the local host per second, whilst 'local_hosting' refers to whether or not the content can be hosted from the local node (e.g. by instantiating a HTTP server).

4.2.2. IConsumer. The IConsumer interface is presented to applications that require access to content; Table II provides an overview. The defining properties of the consumer delivery-centric interface are two-fold: (i) it receives content requests formatted as unique content identifiers without any reference to location or the method of access; and (ii) it allows the association of abstract requirements with such requests. The first method, **get**, allows applications to request an item of content using a unique global identifier, which can also be associated with a set of rules. It is similarly necessary to state how the application wishes to 'view' the content, e.g. an in-memory live stream, a file reference etc. Each of these 'views'

is represented by an object, which extends the abstract Content object. Currently these subclasses are: FileStoredContent, MemoryStoredContent, RangeStoredContent and StreamedContent. Depending on the application's choice, one of these objects is therefore returned as a reference to the data, providing the appropriate methods to access it.

An active content request can also be cancelled using the **stop** method. Whilst, finally, the **update** method can be used to modify previously issued requirements (e.g. to increase the required performance for a request).

Table II. IConsumer Definition

Method	Description
get	This accepts a unique content identifier, a set of rules and a type of access (e.g. stored file, stream). The underlying system must then retrieve the content item in a way that is conducive with the requirement rules and compatible with the type of access.
stop	This cancels a previous **get** request for a given item of content.
update	This updates a previously issued set of rules for a given content request. The underlying system should then adapt to reflect these new requirements.

The current Juno implementation supports the following requirements ontology for IConsumer: 'avg_bit_rate:int', 'upload_resources_required:bool', 'anonymous:bool', 'encrypted:bool' and 'encryption_strength:int'. Beyond this, further dynamic requirements support is being developed including resilience information and monetary cost.

5. JUNO MIDDLEWARE DESIGN

This section details *Juno*, a middleware which implements the delivery-centric API described in Section 4. First, a general overview is given; following this, each of the main frameworks within Juno are detailed, showing how the required underlying functionality is built.

5.1. Juno Overview

Juno is a component-based middleware that utilises dynamically (re-)configurable plug-ins to adapt the way it provides/consumes content based on its environment and any higher-level requirements. Figure 1 provides a general overview of how Juno consumes content. Each framework uses protocol *plug-ins* that allow the framework to interoperate with a given external system (e.g. BitTorrent, HTTP servers, a cloud service). These are simply pluggable software components that implement various protocols behind standard shared interfaces. The fundamental principle behind Juno is that it can exploit these plug-ins to

dynamically select the best mechanism to provide or consume content with at request-time. Consequently, if a range of potential sources were available and BitTorrent were considered the optimal, a BitTorrent plug-in could seamlessly be attached to access the content from. This therefore liberates an application from making static design-time decisions regarding content distribution, instead allowing them to dynamically interact with the interfaces defined in Section 4.

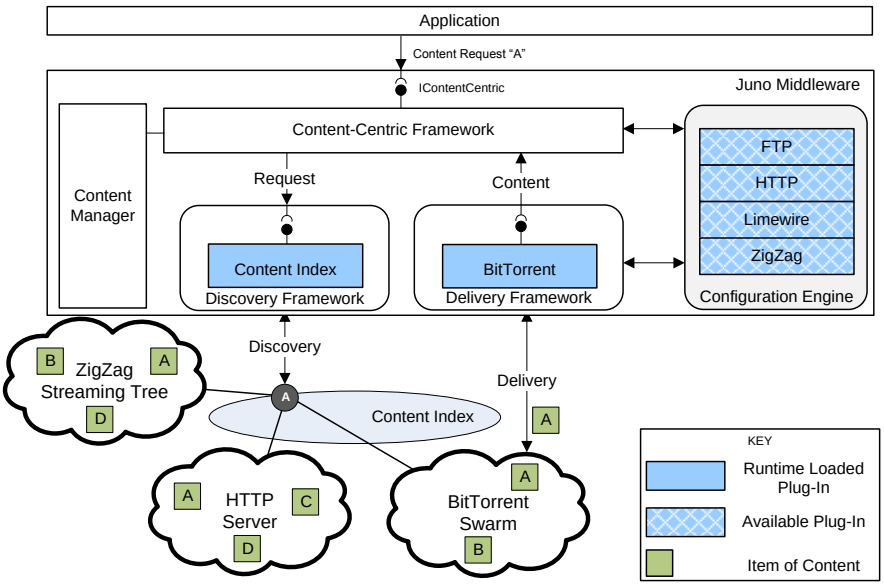


Fig. 1. Overview of Juno's Operation

There are a number of key components in Juno that work in cooperation to offer this functionality. These are as follows:

— *Configuration Engine:* Maintains a repository of available plug-ins and selects the optimal ones at runtime based on the stipulated requirements (on behalf of the other frameworks).
— *Content Manager:* Provides a unified method of indexing and accessing local content. All frameworks (and plug-ins) utilise this to write/read local content, thereby simplifying development and ensuring that data cannot be lost during re-configuration.
— *Provider Framework:* Deals with publishing and providing content to consumers.
— *Content-Centric Framework:* Deals with the access of content for consumers. This framework consists of two sub frameworks:

— *Discovery Framework:* Deals with mapping content identifiers to available sources.
— *Delivery Framework:* Deals with accessing the content through the preferred abstraction (e.g. downloaded to file, streamed to memory etc.).

When an application wishes to *consume* an item of content, it requests the IConsumer interface from Juno, which is implemented by the Content-Centric Framework. First, the framework contacts the Content Manager to find out if the content is already locally available (e.g. has been previously downloaded). If not, the Discovery Framework is queried to locate a set of potential sources for the content; these could include such things as HTTP servers and BitTorrent swarms. These sources are then passed to the Delivery Framework, which instantiates a plug-in for each delivery protocol available. These plug-ins are then provided with their appropriate sources so that they can generate any required dynamic meta-data describing the characteristics of each source.[2] Using this meta-data, the Configuration Engine then selects the optimal plug-in by comparing them against the requirements issued by the application (e.g. avg_bit_rate = max). This dynamic meta-data is periodically re-computed and compared against the requirements so that any environmental changes can be reacted to by replacing the previously selected plug-in. Importantly, by utilising a shared Content Manager, this can be done without loss of data.

Similarly, when an application wishes to *publish* an item of content, it requests the IProvider interface from Juno. This provides access to the Provider Framework, which hosts multiple *provider plug-ins*, thereby allowing the framework to multiplex publication requests into any attachable plug-in. A plug-in implementation could range from a locally hosted web server to a remote cloud storage service to upload the content to. The rest of this section now details each of the previously mentioned frameworks in turn.

5.2. Configuration Engine

Juno's Configuration Engine maintains a repository of all plug-ins available and is responsible for selecting optimal plug-ins at runtime. It therefore receives plug-in requests (from the other frameworks) associated with sets of requirements; if multiple plug-in implementations

[2]This meta-data uses the same ontology as that used to present the requirements (e.g. avg_bit_rate).

are available, it then returns the one that best fulfils the requirements. This section details the principles of (re-)configuration and how the Configuration Engine achieves it.

5.2.1. Principles of (Re-)Configuration. Each framework in Juno offers the functionality (and interface) to provide a given service, such as accessing an item of content. However, clearly, each service can be achieved in a number of different ways; specifically, in the context of Juno, content can be consumed and provided using a variety of different protocols and infrastructure. The aim of Juno is to achieve these two functions in the optimal manner by dynamically switching between the use of different protocols and infrastructure. To enable this, different protocol implementations are embodied in dynamically attachable software components called *plug-ins*. These plug-ins can then be used by the frameworks to interoperate with the most suitable provider infrastructure, based on runtime conditions.

Plug-ins are implemented as software components that each support one or more *plug-in interfaces*. These pre-defined interfaces are known by the frameworks and allow plug-ins to receive requests for given tasks (e.g. request a content item). Importantly, plug-ins also have explicit life cycle management, i.e. the ability to be initiated and shutdown during runtime. Through this, plug-ins can be dynamically attached (and detached) to Juno's core frameworks and utilised to interact with a given external system. We define the act of *configuration* as the process of selecting the optimal plug-in dynamically at request-time, whilst we define *re-configuration* as the process of later replacing it with another plug-in to reflect some environmental change. Replacing a plug-in can be performed (*i*) *sequentially* by removing the first and then attaching the second; or (*ii*) in *parallel* by bootstrapping the second before removing the first. The former (which is the default) has the lowest memory/processing overheads but introduced a re-configuration delay, whilst the latter can reduce delays but increase the overheads. In both cases, Juno manages all memory referencing to protect applications from complexity

5.2.2. Selection of Plug-ins. When a framework wishes to perform a task (e.g. to access a content item), it issues a request to the Configuration Engine for a plug-in that can offer the service, alongside a set of requirements structured as $\langle attribute, comparator, value \rangle$ tuples; generally these requirements are acquired directly from the application through the IConsumer and IProvider interfaces. Each plug-in is required to expose corresponding meta-

data about itself, structured as $\langle attribute, value \rangle$ pairs. For each type of plug-in interface,[3] the Configuration Engine maintains a set, P, of compatible plug-ins, whilst the function $get(p, x)$ retrieves the attribute x from plug-in p. Therefore, for example, if a request for that service is received with the requirement $x = 5$, the set P is filtered as, $compatible = \{p | p \in P \wedge get(p, x) = 5\}$. If $|P| > 1$, a random plug-in is simply selected, whilst, alternatively, if $|P| = 0$, an exception is thrown to alert the application.

As discussed in Section 4.1, meta-data can deal with any characteristic that might be of importance to an application. This can be both static and dynamic. Static items are those that do not change during runtime, whilst dynamic items are those that must be dynamically generated to reflect current operating conditions. Clearly, it is dynamic meta-data that must be used to address consumer and temporal variance. To enable the generation of dynamic meta-data, each plug-in must be provided with details of the available sources that are compatible with that plug-in. The plug-in is then responsible for computing predicted dynamic meta-data values for those sources. Specifics of this are delayed to the following sections. However, these principles (embodied within the Configuration Engine) are utilised by both the Content-Centric Framework and the Provider Framework to fuel adaptation.

5.3. Content Manager

Within Juno, a Content Manager handles the local storage and indexing of content, alongside managing content naming. This section details the operation of this framework.

5.3.1. Content Storage. Juno abstracts the content storage away from any individual plug-ins, thus allowing them to share a common content library. All plug-ins read from and write to the Content Manager without storing any data within themselves. This has two key benefits; first, it eases plug-in development complexity by allowing convenient content read/write methods; and, second, it allows plug-ins to be detached and replaced without losing content. Whenever a content request is received by Juno, the Content Manager is first queried as to whether a local copy is available. If not, a plug-in is instantiated to remotely access the content. Whenever a plug-in is instantiated, it uses the Content Manager to ascertain what parts (if any) are already locally available; via this mechanism, plug-ins can

[3] A number of plug-in interfaces exist for handling content discovery, publication and various types of content access (e.g. streamed access).

be attached and detached seamlessly without loss of data or the complexities of transferring data between old and new plug-ins.

5.3.2. Content Naming. Content identifiers in Juno are created by generating one or more hash values from the content's data (each one constitutes a valid name). Consequently, when content is published, it is first passed through a set of hashing algorithms to create the necessary name. The use of this approach has two benefits, (*i*) it allows self-certifying identifiers that can be used to validate content on arrival; and (*ii*) it allows globally unique identifiers to be generated in a distributed manner without the use of a centralised identification authority. More important, however, is the observation that a large number of existing discovery systems already support the use of such hash-based identifiers. Thus, allowing interoperable and open access to previously published content that is unaware of Juno, as well as more convenient interaction with existing content protocols. To further enable this, Juno utilises the Magnet Link addressing standard,[4] which provides a format for passing hash-based content requests into a variety of different content distribution systems. This allows consumers to request uniquely identified content from a range of different systems; according to one study, ≈99% of Internet peer-to-peer traffic supports Magnet Link identification [Schulze and Mochalski 2009]. Examples of delivery systems that support Magnet Links include Gnutella, Gnutella2, ED2K, BitTorrent, Kazzaa and Direct Connect. The use of this standard thereby simplifies Juno's interaction with a range of different content protocols, as well as often allowing backwards compatible access to third party sources.

5.4. Provider Framework

This section details the Provider Framework, which is responsible for publishing content items when needed by an application. First, the framework is described before discussing how it can be (re-)configured to address an application's individual needs.

5.4.1. Provider Framework Design. The first mode of operation supported by Juno is that of a provider. When this is requested, Juno returns the IProvider interface detailed in Section 4.2.1. This is exposed by the Provider Framework, which handles any publication requests. When it receives a publication request, a set of hash-based identifiers are first generated by

[4]http://magnet-uri.sourceforge.net/

passing the data through a set of hashing algorithms (by default SHA-1, MD5, and MD4). The values returned from these algorithms become the content's identifiers.

Once this has taken place, the framework utilises one or more *provider plug-ins* to publish the content. A provider plug-in has the ability to expose an item of content through one or more delivery schemes. This could perhaps be by instantiating a locally hosted web server, uploading the content to a cloud service (e.g. S3 [Palankar et al. 2008]) or offering it to a peer-to-peer network. All provider plug-ins are required to expose **put** and **remove** methods to enable the Provider Framework to interact with them. When a plug-in publishes an item of content, it also returns a RemoteContent object which contains details of exactly how the content has been made available (e.g. protocols, source information, meta-data etc.). Importantly, a RemoteContent object can contain information about an arbitrary number of sources, each with their own protocols and characteristics.

Once this process has completed, the Provider Framework combines all sources into a single RemoteContent object and then uploads tuples (one tuple for each content identifier) consisting of $\langle ContentID, RemoteContent \rangle$ to a bespoke indexing service called the *Juno Content Discovery Service* (JCDS). This is a simple lookup service, which allows consumers to map unique content identifiers to any potential sources known by Juno. Currently, there are two versions of this: a client-server implementation and a distributed hash table implementation. Importantly, by also utilising common hashing algorithm such as SHA1, it becomes possible to perform the same mapping in existing search protocols such as Gnutella and eMule, which already support the use of Magnet Link addressing. Consequently, any consumers possessing the unique hash identifier(s) can use them to locate any sources indexed on the JCDS, as well as in any third party providers[5] supporting Magnet Links.

5.4.2. (Re-)Configuring the Provider Framework. Provider (re-)configuration refers to the dynamic selection of providers at publication-time, based on the requirements stipulated by the application. Currently, the Provider Framework supports all the requirements detailed in Section 4.2.1. By default, all provider plug-ins that satisfy these requirements will be attached and utilised. Therefore, generally, the Provider Framework will simply multiplex publication requests into multiple plug-ins. However, if a plug-in later invalidates any selec-

[5]This refers to providers that are not managed by the organisation which developed the application.

tion predicates (e.g. if 'local_hosting' exceeds its limit), it will be detached. This is periodically checked every measurement cycle, which therefore limits (re-)configurations to once every cycle (by default every 2 minutes).

5.5. Content-Centric Framework

The second mode of operation is that of a consumer; when this is requested, Juno returns the IConsumer interface detailed in Section 4.2.2. This is offered by the Content-Centric Framework, which encompasses two other frameworks that collectively offer the desired functionality: the Discovery Framework and the Delivery Framework. Figure 2 shows these frameworks and how abstract content requests are mapped into concrete provider requests.

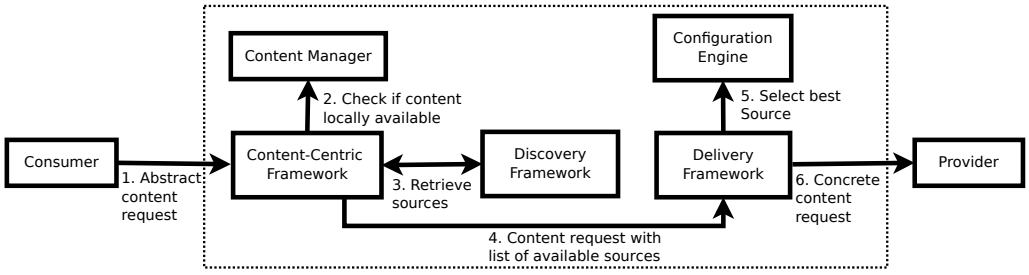


Fig. 2. Flow Chart of Content Request Consumption Process (Dotted lines indicate Juno)

5.5.1. Discovery Framework. The Discovery Framework is responsible for performing the mapping between content identifier and content location; it is therefore used to discover any potential sources of the content when it is not available from the local content manager. Evidently, however, within Juno's design, content can be provided from a range of different providers/protocols. This could be due to the use of multiple plug-ins by the Provider Framework or, alternatively, because the application is accessing open content that is widely distributed by third parties (for instance, Linux ISOs are openly available through various HTTP, FTP and BitTorrent sources, to name a few). Consequently, it is necessary for the Discovery Framework to enable interoperation with this wide range of providers.

To achieve this, the Discovery Framework hosts one or more *discovery plug-ins*, which each contain the functionality to discover content in one or more indexing services. All discovery plug-ins are required to expose a **locateSources** method, which performs a mapping from a content identifier to a set of sources. Discovered content is represented using

a RemoteContent object, which corresponds to that generated by the Provider Framework used to publish the content.

By default, the Discovery Framework always utilises the Juno Content Discovery Service (JCDS) plug-in, which is used by the Provider Framework to upload references to any known sources of the content. Alongside this, a range of other discovery plug-ins can simultaneously be queried to discover sources that are not within the remit of Juno's control. This allows third party sources to be exploited, thereby improving performance. The most prevalent example of this is peer-to-peer sources, which often can be found to offer third party content. Through Juno's use of Magnet Links it becomes possible to discover such sources and pass them to the Delivery Framework, alongside any sources available via the JCDS. References to any third party sources located are also uploaded to the JCDS so other nodes can better discover them. Importantly, by dynamically mapping content identifiers to sources through the Discovery Framework, it becomes possible for applications to discover new sources post-deployment. It therefore does not restrict developers to statically configuring an application with provider information (e.g. a URL); thus, new providers can be added at any time.

5.5.2. Delivery Framework. The Delivery Framework is responsible for accessing an item of content once the Discovery Framework has provided a set of available sources. Evidently, the discovery process will potentially return multiple sources utilising different protocols. It is therefore necessary to be able to (re-)configure the framework to select the optimal source(s) based on the application's requirements.

To achieve this, the Delivery Framework hosts one or more *delivery plug-ins*, which each contain the functionality to access content using a given protocol (or set of protocols). This, for instance, could consist of a generic BitTorrent client implementation or, alternatively, a provider-specific implementation that only operates with a single service.

Evidently, different delivery plug-ins will offer differing types of content access based on the underlying protocols they implement. Some plug-ins (e.g. BitTorrent) cannot be used for media streaming as they do not perform in-order deliveries, whereas others (e.g. HTTP) could be used for both stored content downloads and streaming. To address this, applications must stipulate the type of content access they require (e.g. live stream, file reference); this is done through IConsumer's **get** method (c.f. Section 4.2.2). This preference is then used to inform the selection of the delivery plug-in as only compatible ones are included in the se-

lection process. To formalise this diversity, a set of different plug-in interfaces exist for each of the Content subclasses detailed in Section 4.2.2: FileStoredContent, MemoryStoredContent, RangeStoredContent and StreamedContent. For example, the FileStoredContent plug-in interface requires a file reference at initiation so that the Content Manager can be told where to store the file. This diversity allows applications to operate with content using the abstraction that is most convenient for their needs. For instance, a file sharing application would request a FileStoredContent plug-in, whilst a video streaming application would request a StreamedContent plug-in. Importantly, a plug-in implementation can support multiple interfaces, thereby allowing a single plug-in to be used differently by each application (e.g. the HTTP plug-in supports all of the above interfaces).

5.5.3. (Re-)Configuring the Content-Centric Framework. The Delivery and Discovery Frameworks provide the basis for configuring and re-configuring Juno to access content in the optimal way. The Discovery Framework is responsible for locating as many sources as possible, whilst the Delivery Framework is responsible for selecting the optimal one to utilise. This form of (re-)configuration is therefore more sophisticated than in the Provider Framework, which generally multiplexes all publication requests into all compatible plug-ins. This is due to the one-to-one nature of consumption in comparison to the one-to-many nature of provision (i.e. a provider needs to satisfy many consumers whilst a consumer only needs to satisfy its own requirements).

In essence, (re-)configuration of the Content-Centric Framework involves four stages: (*i*) stipulation of requirements by the application; (*ii*) discovery of available sources and their characteristics; (*iii*) comparison of application requirements against source characteristics; and (*iv*) selection and attachment of protocol plug-in to interact with optimal source. Source characteristics (represented by their meta-data) can be broken down into two groups. The first are *static* characteristics; these are generally based on the protocol that the source supports. For instance, if a source uses HTTP, it would not be possible to access it over an encrypted connection; this attribute will therefore never change (i.e. it is static). In contrast, source characteristics can also be dynamic, i.e. they can change at runtime between different consumers (e.g. performance). Only dynamic characteristics generate consumer and temporal variance. Currently, Juno supports a single item of dynamic meta-data: 'avg_bit_rate:int'.

This refers to the throughput that can be expected from a particular plug-in when accessing an item of content. Techniques to generate this have been defined for the following protocols:

— *HTTP:* The iPlane service [Madhyastha et al. 2006] is used in conjunction with the model detailed in [Padhye et al. 1998] to calculate predicted download performance.
— *BitTorrent:* The model from [Piatek et al. 2007] is used to calculate predicted download performance. The necessary runtime parameters are obtained using a publicly available dataset [Idal et al. 2007].
— *Limewire:* History-based predictions are used to predict download performance [He et al. 2007]. HTTP predictions between each individual source can also be utilised to augment this information, as Limewire utilises multi-source HTTP to perform downloads.

Consequently, if, for instance, the content is accessible via a HTTP server, meta-data for that source is generated using iPlane. This meta-data is then used (alongside static meta-data) to select the optimal plug-in in same way as detailed in Section 5.2.1. Currently, the Delivery Framework also supports the other meta-data detailed in Section 4.2.2. Importantly, these are all static items of meta-data that are extremely efficient to compare whilst the above dynamic techniques complete accurately in under a second.

To ensure that suitable re-configurations are performed, all meta-data predictions must also include any overheads involved in bootstrapping the plug-in. These are generated by the individual plug-ins and integrated into the meta-data predictions so that they are automatically taken into account by the Configuration Engine. For instance, a bit rate prediction by a BitTorrent plug-in must also include the cost of bootstrapping itself in the swarm (usually in the order of seconds). Consequently, any re-configuration overheads are always taken into account during decision making; this, for example, prevents re-configuration taking place when only a small amount of data is left to be downloaded. This is assisted by the use of the shared Content Manager, which allows all plug-ins generating meta-data to inspect the current progress of the delivery (allowing them to find out exactly which parts of the content still remain to be downloaded). By default, all dynamic meta-data is re-generated every 2 minutes and compared against the requirements, thereby preventing frequent oscillation between plug-ins. Importantly, by hiding the application from such changes, it can simply continue to interact with the returned Content object.

6. EVALUATION

The aim of this section is to evaluate Juno's ability to perform per-node (re-)configuration in order to best distribute content.

6.1. Evaluation Methodology

In this evaluation, we aim to validate Juno's ability to (re-)configure itself in reaction to consumer and temporal variance. To enable this, we use a typical middleware evaluation methodology and utilise a set of case-studies. These intend to extensibly generalise the core environments and workloads Juno will operate with; importantly, these should also highlight how Juno reacts in different situations and scenarios. There are four key consumer usage scenarios that could be used to design case-studies: (*i*) a consumer discovers multiple non-Juno providers offering the desired content using different protocols; (*ii*) a consumer discovers a single Juno provider offering multi-protocol support; (*iii*) a consumer discovers a single non-Juno provider offering only a single protocol; (*iv*) a consumer discovers multiple Juno providers, each offering multi-protocol support

Within this evaluation we focus on the first two scenarios. Clearly, in Scenario (*iii*) there is no potential for (re-)configuration as there is only a single provider; thus, it is important to state that Juno offers no immediate advantages beyond the development benefits (e.g. the abstraction of content distribution behind a reusable API). Further, in practice, the setting in Scenario (*iv*) is identical to that of Scenario (*i*) in which multiple providers using multiple protocols are discovered (Juno implicitly offers multi-protocol support). Consequently, we view the first two scenarios to be of primary importance.

Alongside these scenarios, it is also necessary to consider typical requirement sets that will be generated by consumers. We believe that most requirement sets will involve performance-oriented selection predicates and therefore this is used as the key requirement in the case-studies. This is also a requirement that suffers from both consumer and temporal variance; this therefore is an appropriate choice for highlighting Juno's capabilities. Further, it is also important to include static meta-data (e.g. 'encrypted:bool') to highlight how different protocol properties can be exploited. Vitally, because these requirement sets include both dynamic and static aspects, they are extensible to represent any other requirement set (the Configuration Engine applies selection predicates to all meta-data identically).

To realise these case-studies, we have built two simple applications over Juno and deployed them on the Emulab testbed [White et al. 2002]. 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, packet-loss, latency and real-world network protocol implementations. Through this, we create a bespoke environment to study the behaviour of Juno. We therefore use this to compare Juno against the alternative of using statically configured applications, which cannot adapt to address variance. Following these case-studies, overhead measurements are then presented to contrast Juno's benefits with.

6.2. Case-Study 1: Addressing Consumer Variance

The primary use-case of Juno is the situation in which multiple delivery systems are discovered to offer a desired item of content, and Juno must re-configure to access it. This occurs when the Provider Framework publishes content through multiple schemes or, alternatively, when the content is also openly available through multiple third-parties (Scenario (i) and (iv)). This case-study analyses this situation to highlight how Juno addresses consumer variance to dynamically select the provider most suitable for the individual host.

6.2.1. Case-Study Design. We have developed a test application over Juno, with the purpose of requesting content; first, a small 4.2 MB file, followed by a larger 72 MB file. These two sizes have been selected to represent generic music and video files. This consumer application has then been deployed on two different Emulab nodes. The first consumer runs on a low capacity node, *Node **L**ow **C**apacity*, which operates over a typical asynchronous DSL connection with 1.5 Mbps download capacity alongside 784 Kbps upload capacity. The second consumer, *Node **H**igh **C**apacity*, operates over a much faster 100 Mbps synchronous connection. This experiment therefore introduces two variable factors: content size and consumer capacity.

A number of content providers are also set up within the testbed. The content is available from three providers for Node LC, whilst four are discovered by Node HC, as listed in Table III. The three common delivery providers are a HTTP server, a BitTorrent swarm and a set of Limewire peers. These have been selected as they constitute three of the most prominent content protocols currently in use [Schulze and Mochalski 2009]. Node HC further discovers

a private replication server offered on its local network. Clearly, this is only a snapshot of the many possible providers (and environments) that could be discovered, however, we consider these to represent a typical situation. For instance, a number of alternate TCP-based providers (e.g. FTP, HTTPS etc.) could also be included, each with different infrastructural characteristics. This would be automatically handled by Juno during its plug-in selection process. We therefore consider this setup extensible to any situation in which multiple potential providers are discovered.

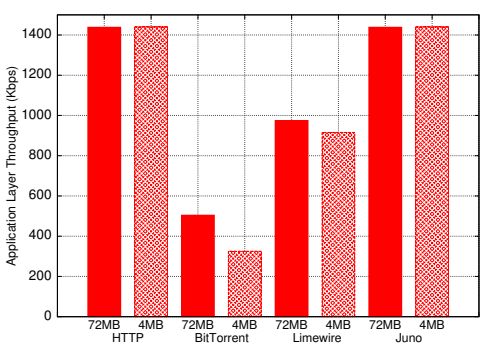
Table III. Overview of Available Delivery Schemes

Available for	Delivery Scheme
Nodes LC and HC	*HTTP:* A server offering the file. There is 2 Mbps available capacity for the download to take place. The server is 10ms away from the clients.
Nodes LC and HC	*BitTorrent:* A swarm sharing the desired file. The swarm consists of 24 nodes (9 seeds, 15 leechers). The upload/download bandwidth available at each node is distributed using a real world measurements taken from existing BitTorrent studies [Bharambe et al. 2006].
Nodes LC and HC	*Limewire:* A set of nodes possessing entire copies of the content. Four nodes possessing 1 Mbps upload connections are available.
Node HC	*Replication Server:* A private replication server hosting an instance of the content on Node HC's local area network. The server has 100 Mbps connectivity to its LAN and is located ≈1 ms away. The server provides data through HTTP to 200 clients.

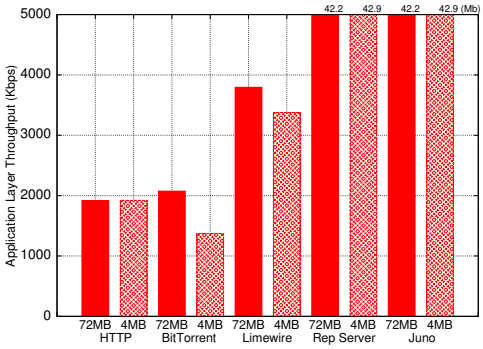
When the application generates the content requests, it associates them with a set of requirements. To study performance aspects, the only requirement generated is '$avg\_bit\_rate = max$'. However, due to Node LC's low upload capacity (784 Kbps), it also stipulates '$upload\_resources = false$', to ensure that its limited resources are not consumed (it should be noted that Juno also supports the automatic introduction of such requirements). This is a static item of meta-data, which is pre-defined in each plug-in; it is therefore representative of any other similar static item of meta-data supported by Juno such as 'encrypted:bool'. In addition to this, the application also provides details of the content sizes to assist in the selection process, e.g. '$min\_file\_size <= 72MB$' and '$max\_file\_size >= 72MB$' (once again, Juno introduces these requirements automatically if the discovery process returns such information). Of course, a variety of other rules could also be added (e.g. encryption support, monetary cost, anonymity), however, as these are less complicated to resolve, we focus on performance issues.

6.2.2. Analysis of Case-Study. The above case-study has been set up in Emulab; Figures 3(a) and 3(b) show measurements taken from both Nodes LC and HC as they were downloading

the two files. It shows the application layer throughput for the 72 MB and 4.2 MB file downloads when utilising each provider. It also shows the throughput of Juno, which selects the optimal plug-in based on meta-data generator predictions.



(a) Average Throughput of Deliveries for ADSL Connection (Node LC)



(b) Average Throughput of Deliveries for 100 Mbps Connection (Node HC)

Fig. 3. Average Throughputs for Nodes in Case-Study

It is first noticeable that the results for Node LC and HC are disjoint, i.e. the optimal providers for Node HC are not the optimal providers for Node LC. This means that an application optimised for Node LC would be suboptimal for Node HC and vice-versa. Consequently, a statically configured application would not be able to fulfil the delivery requirements for both nodes simultaneously. This therefore confirms the presence of consumer variance. Thus, without Juno, an application would need to implement control logic to follow different optimisation paths depending on the host.

The reasons for these disjoint results between the two nodes can be attributed to three key factors that generate variance. First, the two nodes have access to different providers; second, the consumers possess different characteristics; and third, the two items of content requested have different properties (size). Consequently, different combinations of the above factors can drastically alter a provider's ability to satisfy performance requirements. To increase the extensibility of the case-study, each form of variance is now analysed.

The first and most obvious cause of variance is *provider availability*. This refers to the differences in content availability when observed from the perspective of different consumers. For instance, in the case-study, Node HC operates in a network that offers a replication service with very strong connectivity. In contrast, Node LC does not have any such service

available because it is limited to members of a particular network (or often paid members). Variations of this can happen in a range of different situations; Gnutella, for example, will allow different sources to be discovered based on a node's location in the topology. Any delivery-centric system should therefore be able to exploit this consumer variance. Juno supports this by allowing each node to select the source(s) and access mechanism that best fulfils its requirements. Clearly, this also improves interoperability and extensibility by allowing new providers to be introduced without re-coding of applications.

The second type of divergence is caused by differences in *consumer characteristics*. This variance is best exemplified by the observation that, for the 72 MB delivery, HTTP is the optimal plug-in for Node LC but the most suboptimal plug-in for Node HC. This is because Node HC can better exploit the resources of the peer-to-peer alternatives (i.e. BitTorrent or Limewire), whilst Node LC fails to adequately compete (due to its poor upload capacity). In essence, Node LC is best suited to utilising the least complicated method of delivery because the more complicated approaches simply increase overhead without the ability to contribute real performance gains. Once again, this form of consumer variance is effectively addressed by Juno, which configures itself to satisfy requirements on a per-node basis.

The final type of divergence is caused by differences in the *content* being accessed. The previous two paragraphs have shown that in this case-study it is impossible to fulfil the delivery requirements for divergent consumers without performing per-node configuration. However, a further important observation can also be made: the delivery mechanism considered optimal for one item of content is not always the best choice for a different item of content. This is best exemplified by the observation that the optimal delivery system for accessing the 72 MB file is not always the best for the 4.2 MB file. For instance, when operating on Node HC, BitTorrent is faster than HTTP for the 72 MB file but slower than HTTP for the 4.2 MB file (by 34%). This is due to the length of time associated with joining a peer-to-peer swarm. Consequently, optimality not only varies between different nodes but also between different content requests. An application using BitTorrent that cannot re-configure its delivery protocol would therefore observe significant performance degradation between the two downloads. Consequently, delivery system selection must not only occur on a per-node basis but also on a per-request basis. Juno addresses this by seamlessly re-configuring between the different optimal plug-ins, thereby effectively addressing

Table IV. Performance Improvements of Juno over Static Configurations

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

this problem whilst removing the burden from the application. This divergence highlights the fine-grained complexity that can be observed when handling content distribution. This complexity makes it difficult for an application to address all possible needs and therefore provides strong motivation for pushing this functionality into the middleware layer.

The above analysis can now be used to study the behaviour of Juno during this case-study. As shown in the above graphs, for both items of content, Node LC selects HTTP (due to the high download rate and the '$upload\_resource = false$' requirement), whilst Node HC selects the replication server (due to the high download rate). In terms of fulfilling performance requirements, this therefore allows a quantification of the sub-optimality of not using Juno's philosophy of delivery (re-)configuration. Table IV 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). These results highlight Juno's ability to effectively improve performance based on delivery requirements provided by the application. However, this is also extensible to applications that wish to have their deliveries configured based on other requirements, e.g. security, resilience etc.

6.3. Case-Study 2: Addressing Consumer and Temporal Variance

The previous section has investigated consumer variance, showing that applications using Juno can dynamically configure themselves to interoperate with the provider that best fulfils their requirements. It is now necessary to extend this to validate that Juno can similarly address temporal variance by re-configuring to reflect any temporal changes in the environment. The most prominent example of a requirement that suffers from temporal variance is performance. Therefore, this case-study highlights Juno's approach to addressing consumer and temporal variance when trying to fulfil performance-oriented requirements. To further extend the previous case-study we also discuss the provider-side behaviour.

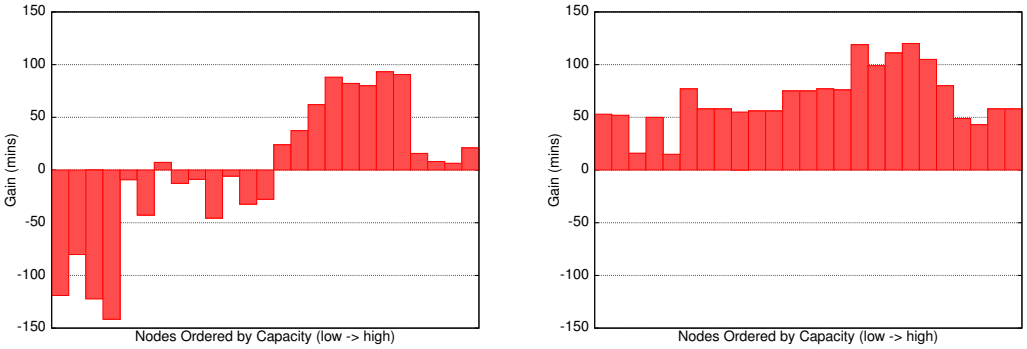
6.3.1. Case-Study Design. We have developed a second test application using Juno, which consists of a provider that is distributing a 698 MB video file to a set of consumers (this is a typical MPEG-4 movie file size). This application has been deployed onto a range of nodes in the Emulab testbed. Importantly, unlike the previous case-study, the content is solely provided by a single publisher, rather than being available through many different sources (i.e. Scenario (*ii*)). The provider operates on a single node with 10 Mbps upload capacity; this is a typical server capacity as shown by [Antoniades et al. 2009], which found that ≈60% of users gained at least 10 Mbps from the Rapidshare Premium service. 25 consumers are instantiated on nodes configured with bandwidth data taken from [Bharambe et al. 2006]. Initially, only three nodes are present in the experiment; however, after 20 minutes, the other 22 nodes begin to arrive sequentially in 20 second intervals. This is done for evaluative purposes to better follow performance changes (as opposed to using a more realistic Poisson arrival rate). The experiment therefore extends the previous case-study to include both consumer variance (the bandwidth characteristics of the different consumers) and temporal variance (the changes in server loading as the new nodes arrive).

For simplicity, the consumer applications just use the '$avg\_bit\_rate = max$' requirement. Also, when the provider application generates the publication request, it associates it with a single requirement: '$local\_upload < 9Mbps$'. This indicates that the upload rate of the host should not exceed this upper capacity for longer than the given measurement cycle (default 2 minutes). As the provider is on a single host, initially, only a HTTP plug-in is therefore instantiated.

6.3.2. Analysis of Case-Study. The case-study has been set up in Emulab over a number of nodes. Initially, the first three consumers select HTTP because this is the only source. After 20 minutes, however, the demand for the content increases and the 22 further nodes begin to issue requests to the HTTP server. This temporal change results in a performance degradation for all the consumers, as the server's resources become saturated. At the server, this temporal change also results in the provider's requirements being invalidated (as shown through the HTTP plug-in's meta-data). Consequently, the rules are re-executed on the available provider plug-ins. Similarly, the consumers also re-execute their local rules.

We consider two outcomes of this process, (*i*) the removal of the provider HTTP plug-in and its replacement by the BitTorrent plug-in; or (*ii*) the addition of the BitTorrent plug-in

to operate alongside HTTP. This is a policy decision made by each provider application. In the former case, all consumers must then re-configure to use BitTorrent as it becomes the only available source; whilst, in the latter case, each consumer is left to select its preferred access mechanism (BitTorrent or HTTP). Importantly, Juno handles both situations without application awareness. For completeness, Figure 4 shows the gains, in terms of download time, when utilising both policies; nodes are ordered by their download capacity with the slowest nodes at the left (results include re-configuration times).



(a) System-Wide Re-Configuration (b) Per-Node Re-Configuration

Fig. 4. Benefit of Juno using Different Re-Configuration Strategies

First, Figure 4(a) shows what happens when a system-wide re-configuration is employed, i.e. the provider detaches its HTTP plug-in and forces all nodes to use BitTorrent. This therefore involves all nodes replacing their HTTP plug-ins with the BitTorrent one. Clearly, it can be seen that the lower capacity nodes suffer from the system-wide re-configuration; 12 out of the 25 nodes take longer to complete their downloads. This occurs because in BitTorrent nodes are required to compete for download capacity [Piatek et al. 2007]; in the case of lower capacity peers, often this is difficult, resulting in low performance. This therefore highlights the unattractive nature of ignoring consumer variance through system-wide re-configuration strategies (unlike Juno's per-node approach).

In contrast to this, Figure 4(b) shows Juno's performance when utilising per-node re-configuration, allowing each node to individually select its own plug-in, i.e. the provider serves the content simultaneously through both HTTP and BitTorrent. It can be observed that every peer improves its performance when utilising this strategy. It allows high capacity

peers to exploit each others' resources through BitTorrent whilst freeing up the server's HTTP upload bandwidth for use by the lower capacity peers. On average, through this mechanism, peers complete their download 65 minutes sooner. This is an average saving of 30% with the highest saving being 51%. Consequently, the case-study further validates the importance of supporting per-node configuration, as discussed in Section 3 and the previous case-study. Importantly, Juno has also been shown to effectively handle temporal variance by periodically re-executing meta-data generation to ensure optimality is maintained.

6.4. Overheads

This section presents an evaluation of Juno's overheads; specifically, its re-configuration delays, memory/processing costs and development burden.

6.4.1. Re-Configuration Delay. Re-configuration is a vital part of Juno's operation; however, it can also introduce an overhead in terms of the delay between detaching and replacing plug-ins. Re-configuration is performed using one of two concurrency models (c.f. Section 5.2.1); the first is *sequential*, which involves removing one plug-in and replacing it with another, whilst the second is *parallel*, which involves leaving the first plug-in attached whilst bootstrapping the second one. The latter option creates no noticeable re-configuration delay but results in far greater resource utilisation. In contrast, sequential re-configuration has a low overhead but results in a delay during which no plug-in operates.[6]

Table V presents the delays for sequential re-configuration, as recorded in Case-Study 2. It can be seen that, on average, re-configuration takes 6 seconds; this delay is caused by BitTorrent's high bootstrap complexity (e.g. contacting peers, calculating hash values), as it only takes 126 ms to locally instantiate the BitTorrent component. It can also be contrasted with re-configuring to use the simpler plug-ins (e.g. HTTP), which take only ≈500 ms. Generally, streaming applications are most sensitive to these delays; however, it is important to note that such a re-configuration would only take place if sufficient data had been buffered to ensure continuous playback. This is in a similar vein to rejecting re-configuration when a delivery has nearly completed (c.f. Section 5.5.3).

[6]Sequential is the default method due to its simplicity.

Table V. Re-Configuration Times for Clients

	Plug-in Re-Configuration			Local Plug-in
	Average	Maximum	Minimum	Instantiation
Consumers	6 Sec	18 Sec	3 Sec	126 ms
Provider	11 Sec	11 Sec	11 Sec	349 ms

6.4.2. Memory and Processing Overhead. Table VI details the memory and processing overheads of various plug-ins. The measurements are taken when a number of different plug-ins are attached individually. Clearly, it shows that there is only a limited overhead involved in utilising Juno, which we consider acceptable for most applications.

Table VI. Runtime Memory Footprint of Configurations (inc. JVM) and Instantation Times

Plug-in Attached	Footprint	Plug-in Instantiation	Juno Instantiation
None	472 KB	N/A	329 ms
HTTP	512 KB	35 ms	357 ms
BitTorrent	522 KB	95 ms	374 ms
Limewire	573 KB	42 ms	369 ms

6.4.3. Development Overhead. The core development overhead is that of *code complexity*. To quantify this, Table VII shows the lines of code required for performing various operations with Juno compared against various alternate delivery toolkits: HTTP (java.net), BitTorrent (HBPTC and trackerBT APIs) and NetAPI [Ananthanarayanan et al. 2009]. These are based on the provision and consumption of a single static object. Clearly, there is relatively constant coding effort required amongst the different APIs, indicating that Juno does not create a noticeable increase in overhead. However, importantly, both the Juno and NetAPI interfaces provide significant gains over HTTP and BitTorrent through their content-centric nature. Further, Juno's ability to achieve delivery-centricity comes at only a small increase in coding overhead (i.e. the need to define the necessary rules).

A further interesting issue is the complexity of Juno's abstract-to-concrete mappings. This is necessary to map the calls made to the delivery-centric API into concrete interactions with the underlying protocol implementations. Ideally, for most plug-ins, there should be a one-to-one mapping to indicate that (*i*) the mapping is a low overhead process; and (*ii*) the mapping is likely extensible to other protocol implementations. This can be studied by looking at the get method of IConsumer. This method only required a maximum of five concrete invocations (for RTP) to interact with the underlying protocol implementations,

Table VII. Coding Complexity for IConsumer and IProvider Interfaces

Interface	Operation	Juno	HTTP	BitTorrent	CCN
IConsumer	get	4	3	11	2
IConsumer	stop	1	2	1	1
IConsumer	update	4	N/A	N/A	N/A
IProvider	put	3	1	13	2
IProvider	remove	1	1	1	1

indicating that the complexity is relatively low. In fact, many plug-ins only required one or two invocations, indicating that integrating new plug-ins is relatively straight forward.

6.5. Discussion and Limitations

This evaluation has shown that Juno can, indeed, dynamically (re-)configure to address the needs of higher-level applications. Further, the overheads of this process have been shown to be low. As such, even when the environment prevents Juno from re-configuring (i.e. only one provider is available), the software engineering benefits can be gained without high costs. However, it is also important to note that the case-studies are not exhaustive and therefore have limitations. Specifically, the choice to use emulated case-studies means that certain real-world considerations have been abstracted away from. On the one hand, this improves control and determinism, however, it also potentially reduces the applicability of the results for some scenarios. For instance, route dynamics were not introduced into the emulations because it has been long understood that most routes are relatively static [Zhang et al. 2000]. However, this does not preclude the existence of network path variations in a real-world deployment. Unfortunately, it is difficult to perform such real-world experiments until Juno has received more uptake and, as such, the use of case-studies means that a smaller number of application-level concerns have been explored (e.g. content types, requirements etc.). Our longer-term evaluative aims therefore include (*i*) building more diverse applications over Juno, ideally involving third parties; (*ii*) exploring larger and more complex requirement sets for these applications, including real-time aspects; and (*iii*) deploying and monitoring these applications over the Internet for long-term periods to understand how real-world variance can actually be handled consistently. Despite this, we consider the case-studies to have been configured realistically using various measurements to offer a number of evaluative insights, which we consider highly promising for Juno's approach.

7. RELATED WORK

Demmer et al. [Demmer et al. 2007] were the first to propose a standardised content-centric API. It is similar to Juno's delivery-centric abstraction, however, it does not allow complex delivery requirements to be represented. Instead, limiting requirements to be expressed through simple properties when performing the **open** operation on content (although further details of how this would work are not provided). Further, the abstraction does not allow such requirements to be adapted after the content has been requested. Other attempts at standardisation include the NetAPI [Ananthanarayanan et al. 2009]. However, currently, none support the stipulation of requirements like Juno does. The defining property of these content-centric APIs is therefore the ability to request an item of uniquely identified content without stipulating any particular source.

Variations of these APIs have been realised by a small set of existing systems. Current content-centric solutions involve the deployment of network infrastructure to perform routing. These systems generally focus on the discovery of content sources, rather than its subsequent delivery. Prominent examples of these systems are DONA [Koponen et al. 2007] and CCNx [Jacobson et al. 2009] (from the Named Data Networking initiative). DONA is a content-based equivalent to the current domain name system (DNS). It builds a distributed tree overlay consisting of a number of Resolution Handlers (RHs), which are used to route REGISTER and FIND messages. Providers use REGISTER messages to publish content, whilst consumers use FIND messages to request content. These FIND messages are routed to the 'closest' source, which then initiates an out-of-band delivery to the requester (over IP). CCNx is an alternative solution, which uses network infrastructure to route content requests to sources. A content request is issued by sending an INTEREST packet, which is routed through the network to the 'closest' instance of the content. Unlike DONA, however, CCNx then returns the content in a DATA packet, which is passed through the content-centric infrastructure (as opposed to out-of-band). The key limitations of these proposed solutions are therefore as follows:

— *Poor configurability of deliveries:* Content-centric networks currently focus on discovery; they do not offer the necessary underlying functionality to adapt source and protocol selection based on complex application requirements. In contrast, Juno's interfaces allow the stipulation of requirements, which can then be adapted at runtime. Importantly, these

requirements can extend to a number of characteristics including both static protocol issues (e.g. supports encryption) and dynamic infrastructural concerns (e.g. performance).

— *A lack of backwards compatibility:* Content-centric networking's aim of re-architecting the Internet suffers from many of the deployment challenges encountered by technologies such as IPv6 and RSVP. In contrast, Juno places a far smaller burden on deployment. Applications (both consumers and providers) need only integrate Juno's interfaces into their software. Importantly, Juno also offers interoperable support with many of the existing prominent protocols; this, for instance, allows consumers to easily discover and interact with existing (third party) providers without modification to them (through the use of the Magnet Link standard and passive indexing on the JCDS).

— *High Deployment Costs:* Content-centric networks often require new routing infrastructure to be built, which mandates heavy investment. In contrast, by integrating content-centric functionality at the middleware layer, such costs can be avoided. Importantly, if a content-centric networking solution were later deployed, this could be integrated into Juno through a plug-in, providing an immediate basis for usage by any Juno applications.

An interesting variation on these is the Data Oriented Transfer service (DOT) [Tolia et al. 2006], which allows applications to abstract control over deliveries to a software toolkit. The DOT service then accesses the content on the application's behalf. However, it does not support the receipt of content-centric identifiers, instead requiring the application to perform the necessary negotiations with the chosen content source. Its key goal is therefore superior software engineering and component reuse (also a key aim of Juno). In contrast to the consumer-driven approaches of Juno and DOT, there are also content distribution networks (CDNs) [Fortino et al. 2009] such as Akamai [Su and Kuzmanovic 2008], which utilise DNS redirection to select optimal sources. These, however, do not allow individual consumers to adapt their deliveries — it is solely controlled by providers (with considerable monetary costs). Such things as protocol adaptation are therefore not supported, as this would require consumer involvement. Thus, Juno empowers individual consumers and applications in a way that is not possible using CDNs.

8. CONCLUSION

This paper has introduced the concept of delivery-centricity. This exploits the observation that many applications do not have a vested interest in how and where their content comes

from, as long as it verifiable and conducive with their requirements. To this end, delivery-centric interfaces have been developed, alongside a middleware solution that implements them. The middleware, *Juno*, utilises software (re-)configuration to adapt its behaviour to the requirements issued by the applications.

It has been shown that Juno can dynamically select and (re-)configure between different protocols and providers in a way that satisfies higher-level abstract requirements. Specifically, we have shown that it is possible to address performance-oriented requirements by exploiting runtime observations of available providers. Beyond this, we have also shown how static protocol-specific characteristics (e.g. upload requirements, encryption support) can be handled by Juno to conveniently address application needs. Importantly, Juno has been designed in a highly extensible way that allows new plug-ins and meta-data generators to be easily added; consequently, there are both immediate and future benefits in using Juno.

Based on the presented work, a number of further research directions are possible. First, it is important to evaluate Juno's usage in the real-world, alongside real systems and users. This could involve the development of both new plug-ins and new meta-data generation techniques. Clearly, these should be aimed towards motivating people to build their applications over Juno. Lastly, an important line of future work is to create more sophisticated decision making algorithms regarding the mapping of requirements onto configurations.

REFERENCES

Afanasyev, A., Tilley, N., Reiher, P., and Kleinrock, L. 2010. Host-to-host congestion control for tcp. *Communications Surveys Tutorials, IEEE 12,* 3.

Ager, B., Mühlbauer, W., Smaragdakis, G., and Uhlig, S. 2011. Web content cartography. In *Proc. ACM SIGCOMM conference on Internet measurement conference (IMC)*.

Ananthanarayanan, G., Heimerl, K., Zaharia, M., Demmer, M., Koponen, T., Tavakoli, A., Shenker, S., and Stoica, I. 2009. Enabling innovation below the communication API. Tech. Rep. EECS-2009-141, University of California at Berkeley. October.

Antoniades, D., Markatos, E. P., and Dovrolis, C. 2009. One-click hosting services: a file-sharing hideout. In *Proc. 9th ACM Conference on Internet Measurement Conference (IMC'09)*.

Bharambe, A., Herley, C., and Padmanabhan, V. 2006. Analyzing and improving a BitTorrent networks performance mechanisms. In *Proc. 25th IEEE INFOCOM*.

Cohen, B. 2003. Incentives build robustness in BitTorrent. In *Proc. 1st Workshop on Economics of Peer-to-Peer Systems*.

Demmer, M., Fall, K., Koponen, T., and Shenker, S. 2007. Towards a modern communications API. In *Proc. 6th Workshop on Hot Topics in Networks (HotNets'07)*.

Fielding, R., Frystyk, H., Berners-Lee, T., Gettys, J., and Mogul, J. C. 1999. Hypertext transfer protocol - HTTP/1.1. In *RFC2616*.

Fortino, G., Mastroianni, C., Pathan, M., and Vakali, A. 2009. Next generation content networks: trends and challenges. In *Proc. 4th UPGRADE-CN Workshop on the Use of P2P, GRID and agents for the development of content networks*.

He, Q., Dovrolis, C., and Ammar, M. 2007. On the predictability of large transfer TCP throughput. *Comput. Netw. 51,* 14, 3959–3977.

Huang, C., Wang, A., Li, J., and Ross, K. W. 2008. Measuring and evaluating large-scale CDNs. In *Proc. of ACM SIGCOMM Conference on Internet measurement*. ACM, New York, NY, USA, 15–29.

Idal, T., Piatek, M., Krishnamurthy, A., and Anderson, T. 2007. Leveraging BitTorrent for end host measurements. In *Proc. 8th Intl. Conference on Passive and Active Measurements (PAM'07)*.

Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. 2009. Networking named content. In *Proc. 5th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT'09)*.

Kaune, S., Pussep, K., Leng, C., Kovacevic, A., Tyson, G., and Steinmetz, R. 2009. Modelling the internet delay space based on geographical locations. In *Proc. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 301–310.

Kaune, S., Rumin, R. C., Tyson, G., Mauthe, A., Guerrero, C., and Steinmetz, R. 2010. Unraveling BitTorrent's file unavailability: Measurements and analysis. In *Proc. International IEEE Conference on Peer-to-Peer Computing (P2P'10)*.

Koponen, T., Chawla, M., Chun, B.-G., Ermolinskiy, A., Kim, K. H., Shenker, S., and Stoica, I. 2007. A data-oriented (and beyond) network architecture. *SIGCOMM Comput. Commun. Rev. 37,* 4.

Krishnan, R., Madhyastha, H. V., Srinivasan, S., Jain, S., Krishnamurthy, A., Anderson, T., and Gao, J. 2009. Moving beyond end-to-end path information to optimize cdn performance. In *IMC '09: Proceedings of the 9th ACM Internet Measurement Conferenc (IMC'09)*.

Madhyastha, H. V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2006. iPlane: An information plane for distributed services. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*.

Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. 1998. Modeling TCP throughput: A simple model and its empirical validation. Tech. rep., University of Massachusetts.

Padmanabhan, V. N. and Sripanidkulchai, K. 2002. The case for cooperative networking. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS'02)*.

Palankar, M. R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. 2008. Amazon S3 for science grids: a viable solution? In *Proc. International Workshop on Data-aware distributed Computing*.

Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2007. Do incentives build robustness in BitTorrent? In *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*.

Plagemann, T., Goebel, V., Mauthe, A., Mathy, L., Turletti, T., and Urvoy-Keller, G. 2006. From content distribution networks to content networks - issues and challenges. *Computer Communications 29,* 5, 551–562.

Rasti, A. and Rejaie, R. 2007. Understanding peer-level performance in BitTorrent: A measurement study. In *Proc. of 16th IEEE Intl. Conference on Computer Communications and Networks (ICCCN'07)*.

Schulze, H. and Mochalski, K. 2009. Ipoque internet study. Tech. rep., ipoque GmbH.

Su, A.-J. and Kuzmanovic, A. 2008. Thinning akamai. In *Proc. 8th ACM SIGCOMM Conference on Internet Measurement (IMC'08)*.

Tolia, N., Kaminsky, M., Andersen, D. G., and Patil, S. 2006. An architecture for internet data transfer. In *Proc. 3rd USENIX Conference on Networked Systems Design & Implementation (NSDI'06)*.

Tyson, G. 2010. A middleware approach to building content-centric applications. Ph.D. thesis, Lancaster University.

Tyson, G., Mauthe, A., Kaune, S., Grace, P., and Plagemann, T. 2012. Juno: An adptive delivery-centric middleware. In *In Proc. 4th Intl. Workshop on Future Media Networking (FMN'12)*.

Tyson, G., Mauthe, A., Plagemann, T., and El-khatib, Y. 2008. Juno: Reconfigurable middleware for heterogeneous content networking. In *Proc. 5th Intl. Workshop on Next Generation Networking Middleware (NGNM'08)*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. 2002. An integrated experimental environment for distributed systems and networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

Yu, H., Zheng, D., Zhao, B. Y., and Zheng, W. 2006. Understanding user behavior in large-scale video-on-demand systems. *SIGOPS Oper. Syst. Rev. 40,* 4, 333–344.

Zhang, X., Liu, J., Li, B., and Yum, Y. 2005. Coolstreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *Proc. 24th IEEE Conference of Computer and Communications (INFOCOM'05)*.

Zhang, Y., Paxson, V., and Shenker, S. 2000. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. *ACIRI Technical Report*.