Juno: An Adaptive Delivery-Centric Middleware

Gareth Tyson[1], Andreas Mauthe[2], Sebastian Kaune[3], Paul Grace[2], Thomas Plagemann[4]
[1]Department of Informatics, King's College London, UK
[2]Computing Department, Lancaster University, UK
[3]KOM, Technischen Universität Darmstadt, Germany
[4]Department of Informatics, University of Oslo, Norway
gareth.tyson@kcl.ac.uk, andreas@comp.lancs.ac.uk, kaune@kom.tu-darmstadt.de,
p.grace@lancs.ac.uk, plageman@ifi.uio.no

***Abstract*—This paper proposes a new *delivery-centric* abstraction. 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, resource consumption and monetary cost. A delivery-centric system therefore constantly adapts to fulfil these requirements, given the constraints of the environment. This abstraction has been realised through a delivery-centric middleware called *Juno*, which uses a reconfigurable 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.**

I. Introduction

A number of recent studies have highlighted the importance of content delivery, showing that a predominant amount of traffic is attributable to content distribution [1]. To exploit this, emergent content-centric designs such as CCNx [2] incorporate support for performing content publication and consumption at the network-level. Currently, however, such infrastructure is not available; instead, a large number of independent (application-level) content delivery schemes exist, ranging from client-server HTTP [3] to peer-to-peer models such as BitTorrent [4]. These systems have been built to address particular requirements that arise when operating with divergent workloads and environments. Hence, it is the responsibility of the developer to select how to best access content at design-time.

This process first involves the developer selecting one or more providers to publish their content through. Generally, the developer will utilise the application's core requirements to decide which provider is best suited at design-time. Once this has taken place, the content must be published before integrating the necessary software support into the application (e.g. using an FTP API to access a file server). In this paper, however, we argue that this 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*.

This observation is exploited to propose a new *delivery-centric* abstraction, which extends the traditional notion of content-centricity [5]. Current content-centric systems place a significant focus on detaching content from location. However, in practice, applications require greater (high-level) control over their content access in regards to such issue as performance, security, resource consumption and anonymity. A delivery-centric abstraction therefore further allows applications to dynamically stipulate complex delivery requirements that place constraints on how the content is accessed, thereby fueling underlying system adaptation.

This paper first formalises the delivery-centric abstraction before detailing the *Juno middleware*, which implements it. Therefore, unlike previous solutions, we do not take a network infrastructure approach; through this, we gain the flexibility to interpret high-level requirements and perform the necessary reconfiguration to select and utilise the optimal content source(s) based on per-node runtime conditions. This is achieved by dynamically computing the abilities of any discovered providers to fulfil the application's requirements, before transparently adapting to interoperate with the source(s) that best address the application's needs. Consequently, Juno offers a mechanism to immediately deploy a content-centric and delivery-centric interface in an interoperable and backwards compatible manner.

The rest of this paper is structured as follows. First the background to the issue is covered. Next, a new delivery-centric API is presented, before detailing Juno. Following this, Juno is evaluated using a set of Emulab experiments. Last, the paper is concluded.

II. Background and Related Work

A content-centric network is one that treats content as a first-class routable entity. It allows hosts to generate requests for uniquely identified content, which are routed towards the nearest source. We consider two aspects of research relevant, (i) content-centric APIs, and (ii) content-centric systems.

The first to propose a standard content-centric API were Demmer et. al. [5]. The API allows applications to *get* and *put* content using a unique key. This, however, has little support for applications that wish to stipulate diverse provision and consumption requirements, e.g. performance, security, overheads. In the existing development paradigm, application-level technologies allow these different requirements to be satisfied by developers intelligently selecting protocols, services and technologies (e.g. using FreeNet [6] for anonymous access).

This, however, is not possible using content-centric APIs as they do not expose the ability to represent complex preferences and requirements beyond simple attributes, e.g. NetAPI [7].

A small set of systems have been developed, which implement variations of Demmer's API. Prominent example are the Data Oriented Network Architecture (DONA) [8], CCNx [2] and LIPSIN [9]. These generally build network infrastructure, which allows low-level provision and consumption packets to be routed through. This creates significant deployment challenges and makes complex delivery-centric adaptation difficult. A variation on these is the Data Oriented Transfer service (DOT) [10], 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. We believe that these principles should be combined to build a deployable system, which allows applications' needs and requirements to be satisfied in a flexible, content-centric manner.

III. The Delivery-Centric Paradigm

Current content-centric interfaces place a significant focus on detaching content from location with little support for adapting and specialising the delivery process. However, we believe that a content-centric API should not compromise the fine grained control of content deliveries in an attempt to homogenise access and protocols. To this end, this section defines a variation of existing content-centric APIs [5][7]. We term this API *delivery-centric*, allowing an application to (i) generate content requests using unique identifiers that do not pre-define the access mechanism or source, (ii) issue abstract computable requirements that define how the content should be accessed, and (iii) receive content in a way that is agnostic to how it has been acquired.

A. Modelling Delivery Requirements

To enable the content delivery process to be specialised, it is necessary for applications to represent their requirements computationally. The interfaces defined within the section both utilise such requirements. Requirements are presented to the API in the form of selection predicates, which we term *rules*. A rule is defined by the tuple,

$$rule =< attribute, comparator, value >$$

The attribute value must adhere to an 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. $R = \{rule^1, rule^2 ... rule^n\}$.

B. 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.

1) IProvider: The delivery-centric IProvider interface is presented to publishers that wish to distribute their content, and consists of two methods,

- **put**(InputStream input, Set<Rule> rules) → ContentID
- **remove**(ContentID) → HashMap<String, Boolean>

The first method, **put**, allows an application to publish an item of content. It accepts two parameters: (i) input is the data stream, and (ii) rules is a set of requirements for how the content should be distributed. It returns a ContentID handle to the caller.

The second method, **remove**, then allows an application to remove a published item of content. As the content may be published through multiple means, the method returns a HashMap containing the success of the operation for all the providers through which the content has been published.

2) IConsumer: The IConsumer interface is presented to client applications. 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 (high-level) abstract requirements with such requests. The interface consists of three methods,

- **get**(ContentID, Type type, Set<Rule> rules) → Content
- **stop**(ContentID) → Boolean
- **update**(ContentID, Set<Rule> rules) → Boolean

The first method, **get**, allows an application to request a given item of content. It accepts three parameters: (i) ContentID is a globally unique identifier, (ii) type is a reference to the desired method of access (e.g. file reference, live stream, etc.), and (iii) rules is a set of requirements. Calling this method returns a Content object, which offers an object handle on the content. This is an abstract class that is extended by four subclasses: FileStoredContent, MemoryStoredContent, RangeStoredContent and StreamedContent. Each one allows the application to view the content in a different way; the choice of which is stipulated by the type parameter.

The second method, **stop**, allows an application to cancel a delivery. The third method, **update**, allows an application to update its previously issued requirements regarding a given delivery. Following this method call, the underlying implementation should adapt the delivery to satisfy these new requirements (the success of this is returned via a Boolean).

IV. Juno Middleware Design

This section details Juno, which implements the above interfaces. In contrast to existing content-centric systems, Juno is built as a middleware rather than a network infrastructure, providing it with the flexibility to scalably interpret and address high-level requirements. It does this by utilising an extensible set of protocol plug-ins stored within a Configuration Engine to select and interoperate with the source(s) that best fulfil each content request's requirements.

A. Content Manager

Within Juno, a Content Manager handles the local storage and indexing of content, alongside managing content naming.

1) Content Storage: Juno abstracts the content management away from any individual plug-ins, thus allowing them to share a common content library. Whenever a content request is received by Juno, the Content Manager is first queried as to whether a local copy is available. If not, it is acquired and stored in the Content Manager.

2) Content Naming: Content identifiers in Juno are created by generating one or more hash values from the content's data. Consequently, when content is published, it is first passed through a set of hashing algorithms. This approach allows self-certifying identifiers that can be used to validate content on arrival. 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 [11] addressing standard, 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 peer-to-peer traffic supports Magnet Link identification [1]. Examples of delivery systems that support Magnet Links include Gnutella, ED2K, BitTorrent, Kazzaa and Direct Connect. The use of this standard thereby simplifies interaction with a range of different content protocols, as well as often allowing access to open third party sources.

B. Publishing Content in Juno

The first mode of operation supported by Juno is that of a provider. This is exposed by the *Provider Framework*, which handles any publication requests (through IProvider). When an item is published, a set of hash-based identifiers are first generated by passing the data through a set of hashing algorithms (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) or offering it to a peer-to-peer network.

Once this process has completed, the Provider Framework uploads tuples (one tuple for each content identifier) consisting of $< contentID, sources >$ 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 algorithms 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 locate any sources indexed on the JCDS, as well as in any third party providers supporting Magnet Links.

C. Consuming Content in Juno

The second mode of operation is that of a consumer. This is exposed by the Content-Centric Framework, which is composed of the Discovery and Delivery Frameworks, as shown in Figure 1.

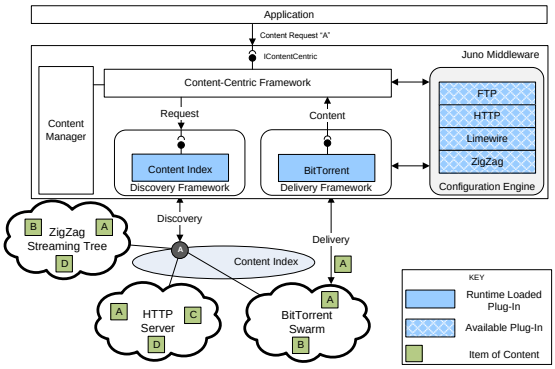

Fig. 1: Overview of Juno's Consumer-Side Operation

1) Discovery Framework: The Discovery Framework is responsible for performing the mapping between content identifier and location; it is therefore used to discover any potential sources of content. Evidently, however, within Juno's design, content can be provided from a range of different providers. 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). 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. By default, the Discovery Framework always utilises the JCDS plug-in, which is used by the Provider Framework to upload references to any known sources of the content. However, 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 examples of this are peer-to-peer sources. 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.

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 for the Delivery Framework to select the optimal one(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. The interface that the plug-ins expose differs between different types of delivery plug-ins based on how the consumer wishes to 'view' the content. Currently, there are interfaces and plug-ins defined for four content types: FileStoredContent (stored as a file), MemoryStoredContent (stored in memory), RangeStoredContent (stored in memory with the ability to request different data ranges) and StreamedContent (stored in memory and allows stream access). This 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.

3) Delivery Plug-in Selection: To achieve delivery-centricity, the Discovery Framework attempts to acquire as many sources as possible, whilst the Delivery Framework dynamically selects the one(s) that best fulfil the application's needs. This is done on a per-node basis, as optimality can vary significantly between consumers; for instance, two nodes will get different throughputs from a server, based on their TCP connection delays [12].

To allow Juno to map delivery requirements to the optimal underlying delivery mechanism, each plug-in is required to expose meta-data about itself, structured as $< attribute, value >$ pairs. This meta-data consists of both static (e.g. supports encryption, anonymity) and dynamic (e.g. performance, reliability, startup delay) attributes. Clearly, static meta-data is trivial to maintain, however, dynamic meta-data must be generated at runtime. To enable this, each delivery plug-in exposes the following method,

- **generate**(Attribute, RemoteContent) $\rightarrow$ Object

This method requests a plug-in to return a particular item of meta-data (attribute) regarding the access of a specific item of content. The ontology of this meta-data is shared with the ontology used by applications to stipulate their delivery requirements (e.g. avg_bit_rate). Consequently, the selection process simply involves comparing the requirements against the meta-data of all the available plug-ins to find the one(s) that are compatible. Currently, when multiple compatible plug-ins are found, a random one is simply selected. Alternatively, if no compatible plug-ins are found, an exception is thrown. This is unlikely to happen in the usual circumstance, in which the provider and consumer applications are deployed by the same organisation. However, if the consumer is solely using third party providers, the exception can be used to re-design the requirements or, alternatively, to discard the request (e.g. if security requirements are compromised).

Dynamic meta-data can be generated in any manner; for instance, Juno-aware providers offer an 'oracle' interface, which exposes supportive meta-data. In contrast, unaware providers must be modelled without support. Currently, Juno supports a single dynamic item of meta-data: 'avg_bit_rate', which refers to the throughput that can be expected from a particular source. Transparent generation techniques have been defined for the following plug-ins,

- *HTTP:* The iPlane service [13] is used in conjunction with the model detailed in [12] to calculate predicted download performance.
- *BitTorrent:* The model from [14] is used to calculate predicted download performance. The necessary runtime parameters are obtained using the publicly available dataset detailed in [15].
- *Limewire:* History-based predictions are used to predict download performance [16]. HTTP predictions between each individual source can also be utilised to augment this information, as Limewire utilises multi-source HTTP to perform downloads.

Dynamic meta-data is re-generated periodically (default 2 minutes) to ensure that the optimal plug-in is continually used. If a superior one is found, the current one is replaced. This is greatly simplified by the use of the shared Content Manager. Currently, the Delivery Framework also supports the following static meta-data: 'upload_resources_ required: bool', 'anonymous: bool', 'encrypted: bool' and 'encryption_strength: int'. Importantly, these are protocol-specific static items and are extremely efficient to compare.

V. Evaluation

A Juno prototype has been developed with support for a number of protocols, including BitTorrent, HTTP, RTP and Limewire. Within this section, we focus on evaluating Juno's ability to re-configure itself to address performance requirements. Due to space constraints, we only investigate this single scenario; a more detailed system evaluation can be found at [17]. To realise this scenario, we have built a simple Juno application that has been deployed on the Emulab testbed [18]. This consists of a media service, which publishes content alongside a consumer application, which accesses it. Within the scenario, the consumer application first requests a 4.2 MB music file, followed by a 72 MB video file, associated with the following delivery-centric requirement: 'avg_bit_rate=max'. To study Juno's ability to perform adaptation on a per-node basis, two instances of the consumer application were deployed: a low capacity consumer, Node **L**ow **C**apacity, which operates over a typical asynchronous DSL connection (1.5 Mbps down/784 Kbps up); and a high capacity consumer, Node **H**igh **C**apacity, which operates over a 100 Mbps synchronous connection. Three shared providers were also created: a HTTP server with 2 Mbps capacity, a BitTorrent swarm (with 9 seeds and 15 leeches, possessing bandwidth taken from [4]), and four Limewire peers possessing 1 Mbps capacity each. Beyond this, a network replication service containing a replica was also available to Node HC.

Figure 2 details the performance of Juno, alongside the results that would have been obtained by selecting each of the providers for both nodes. The results show that the optimal provider for Node LC is different to Node HC, validating that *a traditional, statically configured application would not be able to fulfil the requirements for both nodes*. In the best case, the application could therefore only optimise the delivery for one of the nodes unless it integrated complex control logic like Juno's. This is attributable to three variations that are frequently observed between different consumers,

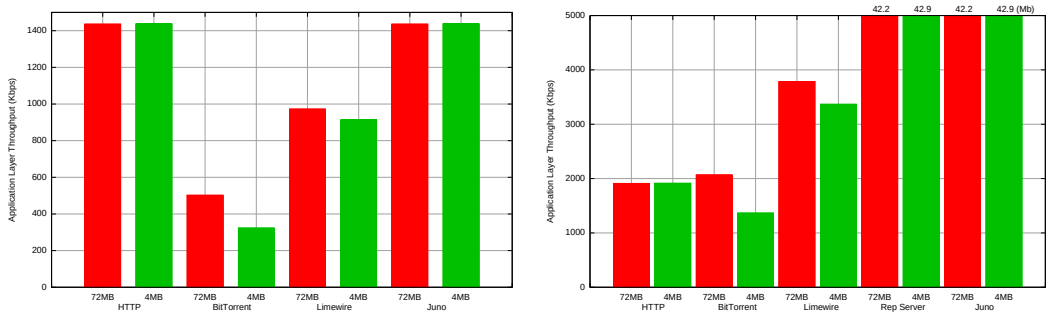

Fig. 2: Average Throughput of Deliveries for (i) Node LC and (ii) Node HC

- Different consumers can locate different sources, e.g. Node HC has access to a replication service; alternatively, peer-to-peer systems will often return different sources for different queries.
- Different consumers have different characteristics, e.g. 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 gain better performance in peer-to-peer alternatives through tit-for-tat [4].
- The ability for a provider to fulfil a requirement can vary between different content requests, e.g. BitTorrent is faster than HTTP for the 72 MB file but slower than HTTP for the 4.2 MB file (on Node HC). This is because BitTorrent performs far better with larger files [19].

Importantly, the results show that for both consumers, Juno adapts to access content in a way that best fulfils the application's requirement on a per-node basis. This therefore, in turn, improves the application's performance without requiring significant development or resource overhead (Juno's memory footprint is only 472 KB). It therefore highlights that (i) Juno can dynamically adapt to address delivery requirements, (ii) Juno can interoperate with a range of existing protocols making it backwards compatible (each provider was setup using an existing third party implementation), and (iii) Juno's abstraction successfully allows content requests to be generated and resolved at runtime.

VI. Conclusion

This paper has investigated the extension of the content-centric paradigm. A middleware, *Juno*, has been detailed, which implements the newly defined *delivery-centric* abstraction, which allows application to stipulate content requirements on a per-node and per-request basis. Through this, Juno adapts the underlying mechanism through which content is accessed by modelling the ability of each source to fulfil the requirements. This is achieved through an extensible set of plug-ins, which allow interoperation with third party systems. Future work includes further investigation into dynamic requirements, as well as dynamic meta-data generation; the integration of new plugs; and an open source deployment for developers.

ACKNOWLEDGMENT

We acknowledge the EPSRC Project: A Framework for Innovation and Research in MediaCityUK (FIRM).

References

[1] H. Schulze and K.Mochalski, "Ipoque internet study," tech. rep., Ipoque GmbH, 2007.
[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *CoNEXT '09: Proc. 5th Intl Conference on Emerging networking experiments and technologies*, 2009.
[3] "Hypertext transfer protocol – http/1.1." http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[4] A. Bharambe, C. Herley, and V. Padmanabhan, "Analyzing and improving a bittorrent network's performance mechanisms," in *INFOCOM '06: Proc. 25th Intl. Conference on Computer Communications*, 2006.
[5] M. Demmer, K. Fall, T. Koponen, and S. Shenker, "Towards a modern communications api," in *HotNets'07: Proc. 6th Workshop on Hot Topics in Networks*, 2007.
[6] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: a distributed anonymous information storage and retrieval system," in *Intl. workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*, 2001.
[7] G. Ananthanarayanan, K. Heimerl, M. Zaharia, M. Demmer, T. Koponen, A. Tavakoli, S. Shenker, and I. Stoica, "Enabling innovation below the communication api," Tech. Rep. EECS-2009-141, UC Berkeley, 2009.
[8] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 181–192, 2007.
[9] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "Lipsin: line speed publish/subscribe inter-networking," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 195–206, 2009.
[10] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil, "An architecture for internet data transfer," in *NSDI'06: Proc. 3rd Conference on Networked Systems Design & Implementation*, 2006.
[11] "Magnet uri project." http://magnet-uri.sourceforge.net.
[12] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling tcp throughput: A simple model and its empirical validation," tech. rep., Amherst, MA, USA, 1998.
[13] H. V. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane nano: path prediction for peer-to-peer applications," in *NSDI'09: Proc. 6th Symposium on Networked systems design and implementation*, 2009.
[14] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do incentives build robustness in bittorrent," in *NSDI'07: Proc. 4th Symposium on Networked Systems Design & Implementation*, 2007.
[15] T. Idal, M. Piatek, A. Krishnamurthy, and T. Anderson, "Leveraging bittorrent for end host measurements," in *PAM: Proc. 8th Intl. Conference on Passive and Active Measurements*, 2007.
[16] Q. He, C. Dovrolis, and M. Ammar, "On the predictability of large transfer tcp throughput," *Comput. Netw.*, vol. 51, no. 14, pp. 3959–3977, 2007.
[17] G. Tyson, *A Middleware Approach to Building Content-Centric Applications*. PhD thesis, Lancaster University, 2010.
[18] "Emulab network testbed." http://Emulab.net.
[19] S. Kaune, J. Stolzenburg, A. Kovacevic, and R. Steinmetz, "Understanding bittorrent's suitability in various applications and environments," in *ICCGI '08: Proc. 3rd Intl. Multi-Conference on Computing in the Global Information Technology*, 2008.