Towards a Modern Communications API

Michael Demmer[*†] Kevin Fall[†] Teemu Koponen[‡§] Scott Shenker[‡*]

Abstract

We contend that a new networking API could better serve the needs of data- and service-oriented applications, and could more easily map to heterogeneous environments, than the pervasive Sockets API does. In this paper, we present an initial design of a networking API based on the publish/subscribe paradigm, along with an exploration of its security implications, examples to demonstrate several common use cases, and a discussion of how the implementation of such an API could leverage a wide range of networking technologies. We propose this model not as a final design but as the first step towards a wider community discussion of the need for a modern communications API.

1 Introduction

In modern operating systems, applications interact with the network through a communications API. This interface defines the conceptual network model for applications, and determines which tasks belong to the protocol stack and which belong to applications. Given its central role in the overall networking framework, one might expect the API to be a carefully architected interface. Unfortunately, this is not the case.

The dominant interface today is the Sockets interface, first developed for BSD Unix roughly 20 years ago. In contrast to the profound architectural discussions about internetworking protocols (*e.g.*, IP and TCP), the Sockets API was designed "bottom up" to solve a very particular problem; once Unix supported TCP (RFC 793), it needed an API for it. To this day, the Sockets API clearly shows its TCP roots. More general interfaces were developed later (*e.g.*, XTI and TLI), but by that time the Sockets API had firmly taken hold.

Enterprise middleware systems (*e.g.*, CORBA, DCOM, J2EE) offer object-oriented method invocation, RPC abstractions and messaging services, but are limited in reach and scalability. Furthermore, they are typically implemented using Sockets. We thus contend that the Sockets interface represents the only generic API used widely on the Internet.

The Sockets API binds to a protocol and an endpoint, requiring the application to know both. This was appropriate for early Internet usage, which mostly involved host-to-host connectivity via a small selection of transport protocols (UDP/TCP). However, in recent years Internet usage has changed significantly, in two important ways.

First, modern Internet usage is now predominantly data- and service-oriented; that is, the application is designed to retrieve some data (or access some service), but it often does not matter from which host, nor via which protocol, the data arrives. Although the application might have latency and reliability requirements, it does not intrinsically depend on the details of the transfer. Thus, there is no reason for these data/service-oriented applications to be dealing with host names, addresses and byte-streams; instead, they should be dealing with directly named data/services and application data units (ADUs) [2].

Second, the Internet has spread quite widely, leading to increased heterogeneity. The Internet is now used in a variety of settings that have wildly divergent performance characteristics; they range, for example, from high-performance computing (HPC) infrastructures to intermittently connected environments. There has been significant work to develop protocols and systems to operate at these extremes, but none have addressed the need for a more general network API that can apply across all settings.

This paper is devoted to a discussion of such an interface. Our goal is to seek feedback on this very preliminary design, and to start a broader discussion of the topic within the community. We understand that the prospects for near-term adoption are nil, but we nonetheless believe that this endeavor is of great long-term importance. Defining a new interface isn't just a question of slightly better engineered software, but of defining new network abstractions that change the division of labor between applications and the protocol stack.

Defining a new "spanning layer" (to use the language of Clark [3]), would facilitate innovation below the API (*e.g.*, DTN [5], name-based routing [9], new HPC designs) and portability above the interface (applications won't have to be rewritten for different communication contexts, such as ad hoc, wired, disconnected, etc.).

Layering shields protocols from the implementation details of the adjacent layers, thereby enabling independent evolution. We wish to extend this decoupling

[*]UC Berkeley, Computer Science Division
[†]Intel Research Berkeley
[‡]International Computer Science Institute (ICSI)
[§]Helsinki Institute for Information Technology (HIIT)

principle beyond implementation issues. In particular, implicit in the Sockets interface is a tight temporal and spatial (location) coupling between the application and the underlying network. That is, the application must know the destination of the transfer (spatial coupling) and be involved during the time of the transfer (temporal coupling). One of the main design goals for this new API is to break these spatial and temporal bonds, thereby giving the protocol stack freedom to carry out its task in a manner appropriate to the current context, and relieving the burden on the application-writer to deal with a variety of communication modalities.

Our ideas are borrowed from and inspired by the existing literature, and here we give a small sampling of the most influential sources. In the networking literature, the DOT proposal [12, 16] provides an interface that can invoke a wide variety of underlying transport methods. The literature on DTN [5] emphasizes temporal decoupling, and DONA [9] emphasizes spatial decoupling by providing a name-based anycast abstraction. Haggle [15] provides a set of mechanisms suitable for ad hoc communications, with the attendant need for spatial and temporal decoupling, but the focus is not on a general API.

There have also been important contributions from sources outside of networking. For instance, Linda [7] provided a generic communications API, but the strong semantics in the interface (such as atomic operations) makes it unsuitable for wide-area deployment.

However, we were most influenced by the literature on publish/subscribe systems [4], and the many voices (including Jon Crowcroft, Van Jacobson, and Pekka Nikander) who have been calling for the use of a pub/sub paradigm. Pub/sub is a very clean embodiment of spatial and temporal decoupling; it decouples the act of consuming content from the act of providing content. Publishers do not necessarily know who the subscribers are (and vice versa), and publication can occur before or after subscription.

Some specific features of our API proposal, discussed in more detail below, were heavily influenced by additional sources. The work on scalable data naming [13] defines grouping of and relationships between data items that are similar to our notions of a publication and inter-publication references. Distributed revision control systems [1, 8, 11] provide a pub/sub like multi-party access interface to data with temporal (and to some degree spatial) decoupling. However, neither scalable data naming nor distributed revision control systems are proposed as a generic communications API.

We obviously make no claim of originality in turning to the pub/sub paradigm. Our goal, however, is to go beyond promoting the paradigm in general to proposing a detailed API that could serve as a generic communication interface. We present preliminary results of this endeavor as follows: we define our API proposal in §2, describe its use in §3, discuss its implementation in §4, and conclude with a general discussion in §5.

Initialization	**open**(P:L, flags, attributes) → handle
Access	**get**(handle) → message(properties) **put**(handle, message(properties))
Cleanup	**close**(handle)

Table 1: API primitives. For open(), flags include *create*, *publish*, and/or *subscribe*, and indicate how the publication will be used, while attributes define the data flow model and publication security properties.

2 API Design

Publications: Publications are first-class objects in this API. That is to say, publications have a globally unique name, and the API primitives revolve around publications.

The API is agnostic to the naming of publications; the only requirement is that publications be uniquely identified. For the purposes of this discussion, we adopt the naming conventions introduced by DONA [9]: The granularity of naming is determined by application requirements; a *principal* may correspond to a user, a particular server, some content from a web site, or any other abstract entity. Each principal is associated with a public-private key pair and each publication is associated with a principal. Publication names are of the form P:L where P is the cryptographic hash of the principal's public key and L is a distinct label assigned by the principal.

Table 1 defines the core API. Publications contain one or more messages (defined in detail below) and applications open a publication with flags to indicate their intention to: *create* a new publication, *subscribe* to get messages, and/or *publish* by putting messages into the publication.

Publication names can be embedded in application data (like URLs in web content). In addition, messages within a publication may contain explicit *references*, *i.e.*, names of other publications, as shown in Figure 1. Applications use these references to convey structure of published data, *e.g.*, to indicate that two publications are related to each other, or to define a hierarchical arrangement of publications. As discussed in §4, these references may also be used by the protocol stack for certain optimizations.

When an application has a publication opened in *subscribe* mode, the communication stack arranges to deliver all currently-relevant messages published to that publication. Although there is no mechanism to subscribe to a portion of a publication, we expect that the communications stack will contain state to record previously-

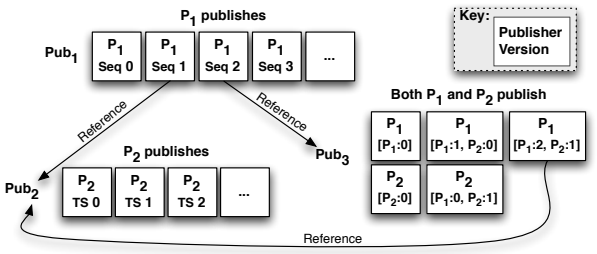

Figure 1: Publications are sets of messages, partially ordered by their versions (defined by publishers using various methods), and containing optional references to other publications.

received messages to avoid duplicate transmissions.

Finally, publications are created with attributes that describe their usage model. These attributes define the lifetime of the publication, the method used for versioning the messages, and optionally the method used to obsolete messages.

Messages: Messages are collections of properties, including meta-data as well as data content. Some properties are well-defined and are interpreted by every communications stack implementing the API, while others may be relevant only in some contexts; in this way they are similar to HTTP headers. The key properties are as follows (we demonstrate their usefulness in §3):

Publisher: The message publisher's identity is always included to disambiguate between multiple publishers and for authenticity verification.

Reference: A message may contain one or more references to other publications by name.

Version: Messages are identified within a publication using application-specified versions. Some version schemes may be well-defined and understood by the communications stack, others may be opaque for the stack and application-specific; examples include: (i) sequence numbers as in TCP, (ii) opaque identifier strings, (iii) absolute or relative time stamps as in RTP, and (iv) logical version vectors, *i.e.*, sequences of [*publisher*, *per-publisher-version*] tuples.

Obsoletes: A message may define a version or range of versions that it renders obsolete, and the stack is expected to expunge obsolete messages upon publication of a newer one.

Lifetime: A lifetime is included with messages to help the stack proactively expire message state. Messages with infinite lifetimes remain in their corresponding publications until they are explicitly made obsolete, or until the publication itself expires.

Data: Of course, messages may also contain application content.

Security: We adopt a "data-oriented" approach to security, so the API focuses on securing the content and not the delivery channel. Indeed, from a security perspective, it makes little difference to applications what the communication media is – it could be a secure filesystem on a USB stick, a public network, or anything between.

Security is achieved using cryptographic techniques and key management. For example, publishers would attach a signed digest to each message, allowing subscribers to ensure the message was not tampered with in transit or in storage. Similarly, message attributes would be encrypted such that only their intended recipient(s), assumed to be holding appropriate key material, can access their values.

The API provides appropriate interfaces for applications to express their required security transforms for message publication/reception. The communications stack is therefore aware of applications' security requirements, so it is not obligated to use redundant or poorly-performing mechanisms. For instance, using self-certifying names implicitly provides security properties such as publisher authenticity; the use of an alternate naming scheme might require an implementation to explicitly include additional authentication credentials.

The data-oriented approach has significant implications on the network infrastructure. In particular, maintenance of access control rights relies on a sophisticated key distribution (and revocation) infrastructure. Also, the ability to selectively apply security transforms to certain sets of message attributes requires a flexible encoding scheme for messages. Neither of these requirements should be taken lightly; however, given the decoupled nature of a pub/sub communications model, we believe the data-oriented security approach to be most appropriate.

Finally, the API does not preclude applications from implementing their own security mechanisms outside the structures defined here, similar to SSL/TLS over insecure sockets. However, this decision would limit the effectiveness of the protocol stack, as it would not have access to the unencrypted data content, nor its structures, that may be useful for optimizing performance, as well as being less convenient for application developers.

3 Using the API

In this section we show how our proposed API is powerful enough to warrant consideration as a new spanning layer, by demonstrating how it can be used for applications in common use today.

3.1 Pre-Published Content

Content distribution protocols, like HTTP and BitTorrent, generally have a publisher that registers content with a distribution or name resolution system, followed by

Publisher	*Subscriber 1*	*Subscriber 2*
open(P_{vid}:L_{ch}, *(create, publish)*, *attrs*) → h	open(P_{vid}:L_{ch}, *subscribe*) → h	
put(h, data:*frame*$_0$, ver:t_0, obsoletes:∅))	get(h) → *frame*$_0$, ver:t_0, obsoletes:∅	
put(h, data:*frame*$_1$, ver:t_1, obsoletes:≤ t_0))	get(h) → *frame*$_1$, ver:t_1, obsoletes:≤ t_0	open(P_{vid}:L_{ch}, *subscribe*) → h
put(h, data:*frame*$_2$, ver:t_2, obsoletes:≤ t_1))	get(h) → *frame*$_2$, ver:t_2, obsoletes:≤ t_1	get(h) → *frame*$_2$, ver:t_2, obsoletes:≤ t_1

Figure 3: Using the API for streaming video to two (or more) subscribers. Each frame is versioned with an increasing timestamp, and the *obsoletes* property expires old frames so late subscribers start in the middle.

Publisher: open(P_{www}:L_{htm}, *(create, publish)*, *attrs*) → h_{htm}
P: open(P_{www}:L_{img}, *(create, publish)*, *attrs*) → h_{img}
P: put(h_{htm}, data:*index.html*, ref:P_{www}:L_{img})
P: put(h_{img}, data:*logo.jpg*)
Subscriber: open(P_{www}:L_{htm}, *subscribe*) → h_{htm}
S: get(h_{htm}) → *index.html*, P_{www}:L_{img}
S: open(P_{www}:L_{img}, *subscribe*)) → h_{img}
S: get(h_{img}) → *logo.jpg*

Figure 2: Using the API to fetch a static web page.

clients that issue requests to obtain content. Such applications are designed around distributing data ahead of time, awaiting later consumption of that data; for them, the API can offer the benefits of both temporal and spatial decoupling between content producers and consumers.

Static Content: Figure 2 shows a simple example of publishing and fetching a static web page that contains an embedded image. Each web object (HTML page, image, etc.) exists in its own publication, and thus has a name. This enables the protocol stack to benefit from caching on the granularity of a single object (similar to existing caches that index on the URL), and allows objects like the logo image to be re-used among multiple pages.

The publishing server opens the publications in the combined create and publish mode, and would publish messages with a signed digest to ensure authenticity, but with no encryption, so anyone can subscribe to the publication and access the content. The client opens the publications in subscribe-only mode and verifies that the received content is properly signed by the publisher.

Because the web page also contains an image, the publisher includes *references* to the image publication when publishing the HTML object. This way, an implementation stack wishing to minimize round trips could prefetch the other publications or pipeline them into the same transport connection (see §4 for a more in-depth discussion of this mechanism). Moreover, because the clients request the content by name, the protocol stack can easily leverage anycast or caching services offered by the networking infrastructure.

Streaming: In situations such as live audio/video streaming, syndicated content such as RSS, or continuous stock quotes, the publishing application does not necessarily have all of the content *a priori*, but instead wants to continuously post new content as it becomes available. For these applications, ordering, versioning, and expiring messages become essential.

As discussed previously, different applications may use different mechanisms for versioning. In Figure 3, we demonstrate an example of streaming live video to two subscribers. The publisher opens the publication in create/publish mode, then continuously puts a sequence of video frames, each with a *version* indicating its relative timestamp to the beginning of the video. Each frame also includes the *obsoletes* property which indicates that prior frames are no longer relevant. Clients can subscribe at any time during the broadcast and begin receiving data. The versioning ensures that clients (and the implementation stack) know which frames are still relevant, and how they should be paced during playback.

3.2 On-Demand Published Content

Instead of pulling content prepared beforehand, some applications may generate content on-demand. Clients of these applications must transfer a request to a server which prepares a response. HTTP POST and RPC protocols are common examples of this kind of interaction.

Dynamic Content: Figure 4 shows a case in which the publisher offers a web search service and dynamically generates web pages in response to requests.

The server first opens a publication in *subscribe* mode to receive search requests. Before posting its request, the client first creates and subscribes to a new unique publication for the purpose of retrieving the results. When the server receives the request, it also receives a reference to the unique publication; it then publishes the search response to this publication with references to publications for the top search results.

The client can benefit from anycast-like functionality in this situation. The search engine can include an authorization property when setting up the search publication (P_{www}:L_{srch}), enabling a server nearby to the client to handle the request. Unlike the streaming video case described above, in which video messages were delivered to *all* subscribers, in this case the search engine principal would have set up its publication so messages need only be delivered to one of its subscribing servers.

Remote Execution: The previous example also illustrates how remote command execution would work in a data-oriented manner, unlike the stream-oriented nature of RSH/SSH. As in the search example above, a server

Server (S): open(P_{www}:L_{srch}, *(create, subscribe)*) → h_{srch}

Client (C): open(P_{www}:L_{srch}, *publish*) → h_{srch}
C: open(P_{cli}:L_{uniq}, *(create, subscribe)*) → h_{uniq}
C: put(h_{srch}, data:*query="pub/sub api"*, ref:P_{cli}:L_{uniq})

S: get(h_{srch}) → *query="pub/sub api"*, P_{cli}:L_{uniq})
S: open(P_{cli}:L_{uniq}, *publish*) → h_{uniq}
S: put(h_{uniq}, data:*result.html*, ref:P_1:L_{doc1}, ref:P_2:L_{doc2}, ...)

C: get(P_{cli}:L_{uniq}) → result.html, P_1:L_{doc1}, P_2:L_{doc2}, ...

Figure 4: Using the API for dynamic web content.

would create and subscribe to a publication where it would receive remote execution requests. A client application would publish new requests to this publication, including the command to be executed and relevant credentials to authorize execution, and would encrypt the message so only the appropriate destination server(s) can access the requests. The server, upon receiving the request, would perform the operation and publish the execution result onto the referred-to unique publication, encrypted such that only the client can access it.

3.3 Multiple Publishers

The lack of widespread deployment of IP multicast forces multi-party communication systems to use centralized servers or overlay networks, but nevertheless such applications are widespread. These applications generally involve multiple entities simultaneously transmitting to the rest of the group. For such applications, the API provides implementation flexibility as the protocol stack is free to use any multicast-like transport it has, and applications do not need to be rewritten for different multicast environments or as technologies evolve.

Shared-Group Multicast: A participant in a multi-party session would first create a publication for the group communication and notify others of this publication identifier. All interested parties would then open that publication both for publishing and for subscribing. Messages could be ordered based on version vectors, ensuring the correct logical sequencing regardless of reception order, or perhaps by timestamps, at the discretion of the application. Messages could expire after some lifetime (*e.g.*, in a group chat), or may be subsumed by more recent messages (*e.g.*, in voice conferencing).

Converge-Cast: In many applications, multiple parties send messages to a single destination. This communication pattern is known as converge-cast and is well known in sensor networks and distributed systems. It also exists in e-mail, as a single e-mail server receives messages from a large set of senders. In this case, the recipient creates a long-lived publication and subscribes to it. The senders open the publication in publish mode, and submit messages for the recipient, appropriately signed and encrypted to provide authenticity and privacy.

3.4 None of the Above

Applications not falling into any of the above categories may still benefit from the API, as the protocol stack can provide structures beyond single byte-streams or datagrams (as in Structured Streams [6]). For example, the remote login (or its secure version, SSH) application is tightly integrated with the model of a transport-layer connection to a particular host, with strong time requirements on the exchange to support interactive use. Our pub/sub interface accommodates these applications, as they simply would create a unique publication (or many, to have logical channels) for each inter-node connection, then trade put/get operations to effect message transfers back and forth.

4 Implementing the API

The implementation challenge is to map the described pub/sub paradigm to a heterogeneous set of network technologies, and to do so without causing significant performance disadvantages. To support the publish and subscribe primitives, we require the network infrastructure to provide the following capabilities:

Name resolution: Subscribers and publishers must be able to resolve publication names to find their content. Understanding the semantics of anycast in the name resolution mechanism is beneficial, but not required.

Message transfer: After name resolution enables the rendezvous of subscribers and publishers, some mechanism is needed to transfer the content.

Subscription state management: Subscribers need to maintain state about publications (*i.e.*, messages already seen) to avoid duplicate deliveries.

Finding available communication resources: Since the API does not define the particular mode of communication, the protocol stack can make opportunistic use of any available means for communicating, and therefore needs to have some intelligence to determine the appropriate resources to use.

Fixed Internet: There are a wide range of options for name resolution, *e.g.*, DNS, Akamai's use of DNS, DONA [9], a BitTorrent swarm. From the API's point of view, it doesn't matter which is used, but some methods will produce better application performance than others. A similar argument applies to the range of transport protocols used to transfer messages.

We envision pre-published content publications would be registered to a resolution infrastructure; subscribing to a publication would thus result in resolving the name, guiding subscribers to a publisher, and using some transport protocol(s) to transfer the publication data. The protocol stack is therefore ultimately responsible for transferring published content to subscribed clients, without the involvement of the publishing application.

On-demand published content requires two transfers per interchange: the request and the response. To accept requests, the server would use a well-known registration in the naming system, which guides the client stacks in routing requests. When the client transfers a request message using some transport connection, the server in its implementation stack can notice the reference to the publication where the client is expecting a response. Therefore, when the server application posts the response, the same transport connection may be reused. Alternatively, the request for content and the content itself could be transferred using a different media (as in DOT [16]).

Shared multicast group applications would use the resolution infrastructure to discover the publication representing a group. Whether it's a network-layer multicast group, an overlay multicast infrastructure, or a central server hosting the group doesn't make any semantic difference to the applications (though, of course, their performance may vary). The message transfer mechanism would depend on the particulars of the underlying multicast mechanism used, if available, but the applications would remain unaware of its details as well.

Delay Tolerant Networks: Given the wide range of environments described as DTNs, there is likely no single implementation approach appropriate for all. Still, there are some key aspects of this API that are amenable to challenged environments like DTNs.

The API is fundamentally asynchronous, so it is suitable for environments with long round trip times. The protocol stack has plenty of knowledge to meet the requirements of such environments. Publications are self-contained (quite similar to DTN bundles [14]) and not packets or streams, which gives information to the routing layer when faced with storage or bandwidth constraints. Moreover, detailed versioning and security information is passed to the stack in the messages, so applications do not need to depend on arrival order to determine inter-message or inter-publication relationships.

The most challenging part of implementing this API in a DTN context is the name resolution. However, given that many DTNs are small in scale, we could proactively flood local name bindings throughout the network to avoid needing a potentially long round trip to obtain the location of where to find a publication, and unknown name lookups would be routed to a well-known gateway node.

Mobile Ad-Hoc Networks: We expect much of the complexity in ad-hoc networks to stem from the name resolution and opportunistic (and intelligent) use of communication resources (as in Haggle [15]). The transfer of messages would be straightforward once the stack has figured whom to contact to subscribe/publish a publication and which communication method to use. However, much as in the DTN case, the challenges are not due to the API itself.

HPC: Esoteric environments, like HPC, have optimized protocols for both name resolution and content transfer, which are not built using TCP/IP nor the Sockets API. However, once protocols are interfaced with the API, the applications would benefit from having a unified interface for both HPC and non-HPC resources. Within HPC environments, applications could benefit from the highly optimized protocols for data transfer, while connectivity to services and hosts beyond the environment could use more common protocols.

In addition to carrying messages on subscriptions, the API must also implement a mechanism for applications to manage their publications. Unsurprisingly, our proposed communications API is similar to filesystem APIs. Both interfaces have temporal and spatial decoupling, and both are file/publication-oriented. Thus, we envision filesystem like abstractions (hierarchical structures and human readable names) may be especially useful for applications managing their publications.

5 Discussion

This paper outlines a preliminary design for a modern communications API. Our intent is to decouple, to the extent possible, the application from the underlying communications mechanisms, and to do so we adopt the publish/subscribe design paradigm. One can view this effort as moving away from a procedural interface, as represented by the Sockets API, to a more declarative [10] one. This frees the communication infrastructure to act opportunistically, using whatever mechanisms are best suited to the current context. It also frees the application from having to cope with different communication mechanisms explicitly.

Designing such an interface is an exercise in tradeoffs – between generality and usability and between powerful semantics and feasibility. There are some capabilities we have deliberately left out of this interface, such as content-based routing, because we don't know how to scalably support such functionality.

There are some tasks that this API may make more difficult than they are today. For instance, counting page hits in the web application is a challenge because the publisher is not explicitly notified when subscriptions are fulfilled (by design). This can be rectified by labeling certain content as "must count" (just as today some content is labeled uncacheable), and requiring that all network caches serving such data must notify the owner (listed in the metadata) of the number of accesses. Of course, this design requires caches to obey this injunction, but this is no different than relying on caches to not cache uncacheable data.

In addition, we expect there are some applications with tight temporal and/or spatial requirements, and which may circumvent this API and access the communication mechanisms directly, with the attendant loss of portability across a range of environments. However, we expect these to represent a small subset of Internet-connected applications.

More generally, adoption of the API increases portability, but does not ensure interoperability. We still need standards for the underlying communication mechanisms, as wide use of the Sockets API doesn't preclude the need for a TCP standard. The key point is that the *applications* need not know about new communication methods, but instead can focus on how to best arrange and convey their data.

The proposal described here is very preliminary, and we plan to implement this interface and test it with a variety of applications. More importantly, since our examples and use cases are limited by our own experience, we hope to get feedback from the broader community. This paper is the first step in soliciting such feedback.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful and helpful feedback on this paper. Also, this material is based upon work supported by the National Science Foundation under Grant Numbers 0722033, 0716342, 0520241, 0225660, and 0205519.

6 References

[1] BitKeeper. The Scalable Distributed Software Configuration Management System. `http://www.bitkeeper.com/`.

[2] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. of ACM SIGCOMM '90*, Philadelphia, USA, 1990.

[3] D. D. Clark. Interoperation, Open Interfaces, and Protocol Architecture. *The Unpredictable Certainty, Information Infrastructure Through 2000: White Papers*, 1997.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[5] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proc. of ACM SIGCOMM '03*, Karlsruhe, Germany, Aug. 2003.

[6] B. Ford. Structured Streams: a New Transport Abstraction. In *Proc. of ACM SIGCOMM '07*, Kyoto, Japan, Aug. 2007.

[7] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of ACM*, 35(2):97–107, 1992.

[8] Git. Fast Version Control System. `http://git.or.cz/`.

[9] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of ACM SIGCOMM '07*, Kyoto, Japan, Aug. 2007.

[10] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of ACM SIGCOMM '05*, Philadelphia, PA, 2005.

[11] Mercurial. `http://www.selenic.com/mercurial`.

[12] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting Similarity for Multi-Source Downloads Using File Handprints. In *Proc. of NSDI '07*, Cambridge, MA, Apr. 2007.

[13] S. Raman and S. McCanne. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proc. of the Sixth ACM International Conference on Multimedia*, Bristol, England, Sept. 1998.

[14] K. Scott and S. Burleigh. Bundle Protocol Specification. Internet Draft. draft-irtf-dtnrg-bundle-spec-10.txt, July 2007. Work in Progress.

[15] J. Su, J. Scott, P. Hui, E. Upton, M. H. Lim, C. Diot, J. Crowcroft, A. Goel, and E. de Lara. Haggle: Clean-slate Networking for Mobile Devices. Technical Report UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, Jan. 2007.

[16] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An Architecture for Internet Data Transfer. In *Proc. of NSDI '06*, San Jose, CA, May 2006.