A New Communications API

Ganesh Ananthanarayanan
Kurtis Heimerl
Matei Zaharia
Michael Demmer
Teemu Koponen
Arsalan Tavakoli
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-84
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-84.html

May 25, 2009

A New Communications API

Ganesh Ananthanarayanan, Kurtis Heimerl, Matei Zaharia,
Michael Demmer, Teemu Koponen, Arsalan Tavakoli, Scott Shenker, and Ion Stoica

ABSTRACT

We present NetAPI, a flexible communications interface. Although the ubiquitous Sockets API lets applications select among a number of mechanisms to accomplish networking tasks, it binds them tightly to their chosen mechanisms. Consequently, the network stack has little freedom in selecting the best protocols and mechanisms for each application, and innovating below the API is extremely difficult. NetAPI allows applications to specify their communication intents against an abstract interface that hides implementation mechanisms, encouraging innovation below the API. Application intents are combined with user policies and environmental conditions to let the network meet application goals in varied ways. We describe the design of NetAPI, comparing it to other system APIs that have supported evolution. We have also implemented a prototype of NetAPI called PANTS for the iPhone platform. We show that PANTS can provide innovative mobile networking features, such as disconnection tolerance, content quality adjustment and power-saving policies, without application modifications.

1 Introduction

Virtually all networked applications today access the network through the *Sockets API* [29], which was developed for BSD UNIX over two decades ago. While the Sockets API lets clients select between a number of network technologies, it binds them tightly to their chosen mechanism (e.g. specifying a destination using an IPv4 address). Trivial architectural changes, such as moving from IPv4 to IPv6, require all applications to be modified. This inflexibility has become problematic as the Internet has evolved. While the original Internet was used largely for file transfer between static hosts, today's Internet consists primarily of mobile hosts accessing a variety of content. The challenges to naming, addressing and content delivery introduced by this shift have led to numerous research projects [7, 25, 27, 28, 31, 33], but these advances have been difficult to deploy.

To understand the limitations of the Sockets API, consider another API used to access system resources: the filesystem API for accessing storage. Alhough it is nearly as simple as Sockets, the filesystem API has fostered far more innovation. Today, applications using the filesystem API can take advantage not just of new types of storage media and placement algorithms, but of filesystems cached in memory (buffer cache), served over the network (NFS), stored redundantly across multiple disks (RAID), partitioned and replicated on a cluster of commodity machines (GFS [39]), or employing advanced deduplication techniques to conserve space (NetApp). The same `ls`, `cat` and `vi` applications written for BSD UNIX will work on this wide range of storage systems without even needing to be recompiled. It is also possible to extend the filesystem in user space through FUSE [37], which has led to interesting and useful projects such as GmailFS [1] and SSHFS [4].

What features of the storage API have made it so much more capable of supporting innovation than the network API? First, the Sockets API requires applications to invoke specific network mechanisms (e.g. specify whether to use TCP or UDP, provide an IPv4 address for a destination host), whereas the filesystem API *hides* storage technology details (e.g. locations are given using paths instead of block numbers). Second, the Sockets API exposes no communication semantics to the network stack beyond a byte stream, whereas the filesystem API captures a minimal amount of information about *application intent* that aids the implementation (e.g. files are opened in read or write mode, which determines what caching may be performed). Stretching the filesystem analogy a little, the Sockets API resembles what might have happened if the filesystem interface included primitives for accessing blocks and inodes rather than named files: performance improvements would be possible, but there would be little room for broad architectural innovation. Our goal in this work is to define a communication API that provides enough abstraction to enable future innovation and meets the requirements of today's Internet applications.

To demonstrate the real-world need for a richer API and provide another point of comparison, we note that while the research community has been exploring new architectures, practitioners have not been idle. Today's practical solution to many of the shortcomings of Sockets is HTTP. HTTP has well-defined request semantics (e.g. GET versus POST) that allow middleboxes to understand how to cache responses, aiding content delivery. DNS load balancing and redirection provide some flexibility in naming. HTTP pro-

vides limited disconnection tolerance by allowing partial-content requests to resume a file transfer in the middle of the file. Finally, cookies let applications establish sessions, which can be used for tracking users across disconnections and for level-7 load balancing of stateful applications. As a result of these features and of the widespread deployment of HTTP-aware middleboxes, many applications that used to have their own protocols now run over HTTP, including file transfer, RPC (SOAP), syndication (RSS), instant messaging (XMPP), and streaming video (YouTube).[1] Nonetheless, HTTP is ultimately limited in flexibility because it is an application-layer protocol. For example, HTTP does not facilitate deployment of new naming mechanisms, or of transport protocols other than TCP. Furthermore, the caching and session features of HTTP are implicit conventions that were tacked onto the protocol over time. Their implementation in middleboxes is architecturally clumsy, and much of the network stack is unaware of these features.

In this paper, we propose a simple communication API called NetAPI that *captures application intent* and *hides implementation mechanisms*, enabling innovation below the API. NetAPI identifies services using Uniform Resource Identifiers (URIs [30]), providing flexibility and extensibility in addressing and naming. This is analogous to how the filesystem API allows mounting multiple filesystems inside a common namespace. In addition, NetAPI operates on application-level messages (data plus properties), letting it capture semantics about content similarly to filesystems and HTTP.

Although our primary goal is to facilitate innovation in the network, NetAPI provides other benefits as well. First, NetAPI enables *adaptability* to environmental conditions in the network stack. For example, a mobile phone can switch a file download started on a cellular connection to a WiFi connection when it enters a WiFi hotspot, because it knows how to resume the transfer. Second, NetAPI enables *centralized policies* to control communications. For example, a phone might present a setting for "best performance" versus "best battery life". Selecting the latter could lower content quality in a video application and delay file downloads until WiFi is available. This feature would be not be possible if the network stack did not understand application semantics.

To evaluate NetAPI, we have built a NetAPI prototype for the iPhone platform called PANTS (protocol aware network technology selector), aimed at mobile networking challenges like disconnection tolerance and use of multiple interfaces. PANTS runs on the client only and interacts with legacy servers. We built two sample applications using PANTS, a file downloader and a news reader, and took advantage of NetAPI to add disconnection tolerance, power-saving policies and content quality adjustment to these applications without modifying them. We also implemented the "best battery life" versus "best performance" setting explained above.

[1] Some of this is also due to corporate firewalls blocking non-HTTP ports. However, many applications, such as video, also use HTTP to take advantage of web caches and commodity load balancers.

We start by describing NetAPI in Section 2. In Section 3, we show how NetAPI supports several popular Internet applications. We explain how NetAPI can be implemented in Section 4. We describe our NetAPI prototype, PANTS, in Section 5, and evaluate it in Section 6. In Section 7, we discuss the implications of NetAPI adoption on the network and applications. We survey related work in Section 8 and conclude in Section 9.

2 NetAPI Design

NetAPI provides four basic operations:

- **open(***scheme://resource*, *options***)** ⇒ *handle*
- **put(***handle*, *message*, *options***)**
- **get(***handle*, *options***)** ⇒ *message*
- **control(***handle*, *options***)** ⇒ *result*.

The user starts a connection through the **open()** call, which returns a connection handle. Instead of asking for an address or DNS name when opening a connection, NetAPI takes a Uniform Resource Identifier (URI) [30] of the form *scheme://scheme-specific-part*. The scheme portion of the URI selects one of a number of *communication schemes*. Each scheme represents a class of network service, such as *web://*, *video://*, or *voice://*. The scheme defines what types of messages and options can be used in API operations and how they are interpreted. The scheme also determines how the scheme-specific part of the URI is resolved, enabling novel naming mechanisms. We explain schemes in detail in Section 2.1.

The **put()** and **get()** operations send and receive *messages* over the connection. NetAPI messages are application-defined data units (ADUs) [17], such as frames in a video scheme. They consist of data plus a list of key-value *properties*. This lets the scheme implementation distinguish between messages types and understand the semantics of each message.

The **control()** function is used for scheme-specific control operations, such as seeking in a streaming media scheme. It takes an *options* argument, which is a list of key-value pairs interpreted by the scheme. It may return a *result* object, which is also a collection of key-value attributes, for schemes that wish to provide control operations with return values. The other NetAPI operations also support scheme-specific options, and may also return result objects (not shown in the API definition above for simplicity) for schemes that wish to return status information.

In addition to these four basic operations, NetAPI provides a **close()** operation for closing a handle, as well as **listen()** and **accept()** operations for creating a server, which function like those in Sockets. Other content publishing mechanisms, such as publish-subscribe, can also be supported, as described in this Section 2.3.

The rest of this section describes and motivates the main design elements of NetAPI: schemes (Section 2.1) and messages (Section 2.2). We also discuss how various server mechanisms can be supported in NetAPI in Section 2.3.

2.1 Schemes

A general-purpose communication API must support applications that require vastly different services from the network: media applications desire low jitter, real-time applications desire low latencies, content retrieval applications may accept a copy of a document from multiple providers, some applications tolerate dropping messages, some applications tolerate disconnections, and so forth. The solution to this problem in the Sockets API was to require applications to manage communication mechanisms themselves, but this inhibits evolution. The solution in NetAPI is schemes. Schemes are one of the main features of NetAPI, so we will explain and motivate them in detail.

At a high level, a scheme is a protocol between the application and the network stack for accessing one class of communication service, such as web content retrieval, media streaming, or RPC. Schemes define the format and significance of names, messages and options used in NetAPI operations. They also define the meaning of the **get()**, **put()** and **control()** operations, and the protocol for calling these (e.g. does a communication contain multiple messages or just one). Finally, schemes capture application *requirements* (e.g. is the application latency-sensitive or disconnection-tolerant). Schemes let NetAPI treat different types of communication differently, and scheme implementations provide a place in the operating system to change how a type of communication is performed without modifying applications.[2]

Scheme implementations are responsible for resolving names, binding to addresses, selecting a transport protocol, encoding messages, and ensuring communication security. As shown in Figure 1, this represents a significant shift from the current responsibilities of the network stack. For example, an implementation of a generic file download scheme (*download://*) may choose to employ new naming mechanisms [26], new transport protocols [43], BitTorrent [14], delay-tolerant networking [34], or even a clean-slate architecture like DONA [31]. The implementation may also choose which network interface to use on a mobile device. Finally, the decisions made by the scheme implementation may be guided by *options* provided by the application in the **open()** call, such as an `authenticate` option requiring the file digest to be verified against a trusted database or a `max_delay` option conveying the level of delay tolerance of the application.

This new division of responsibilities between the application and the network stack lets applications specify their communication goals at a high level while allowing network technologies to evolve underneath them. Nonetheless, the ability of schemes to select communication mechanisms does not entail a loss of control in applications. For example, the system can provide a *datagram://* scheme analogous to UDP and a *stream://* scheme analogous to TCP for applications that desire fine-grained control over networking. These

[2]While we talk about scheme implementations being provided by the operating system, it is entirely possible to run NetAPI in user space. This is what PANTS does.

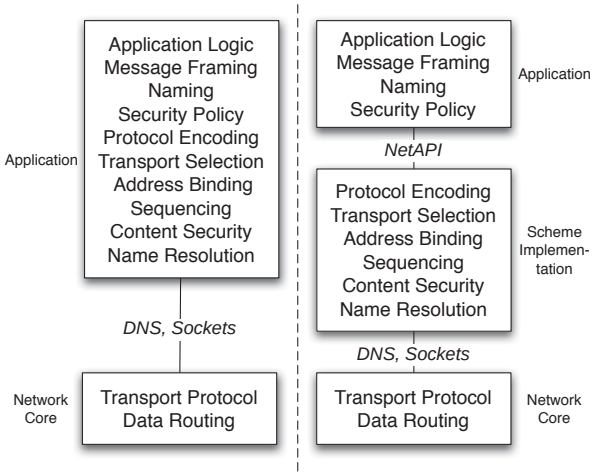

Figure 1: Block diagram showing how the division of responsibilities shifts from the current model (left) to a new functional model (right) with adoption of NetAPI.

schemes could accept raw IP address and port number pairs as names. Even in this case, NetAPI is beneficial because it allows new naming and addressing mechanisms to be added in the future through new formats for the **open()** URI.

In addition to defining the structure and format of messages, schemes also define their semantics. The use of semantic information can go far beyond encoding messages differently or caching them, and into policy decisions. For example, a *web://* scheme may give higher priority to HTML files than to media files like JPEG. This is useful when browsing the web over constrained network links. Likewise, an RSS scheme on a mobile phone might decide to fetch feeds over the cellular interface, but synchronize any attached MP3 podcast files only when in range of a Wi-Fi network. Schemes are also allowed to modify messages, so a multimedia scheme on a mobile phone might adjust content quality depending on the network interfaces available as in BARWAN [28]. Finally, scheme actions may be controlled by global settings in the operating system, like the "best battery life" versus "best performance" example explained in the Introduction. Because of the great flexibility available to scheme implementations, we expect scheme specifications to include a list of permissible implementation choices, similar to today's Internet RFCs, as well as options that can be passed to NetAPI operations to provide hints to the implementation.

We also chose to give schemes the responsibility for communication security, by having applications express security requirements for their content through options in the **open()** call. For example, a *web://* scheme may support an option called `secure` in the **open()** call, which requires it to use SSL, or a *download://* scheme may support an option called `authenticate`, which verifies the MD5 digest of the received object against a trusted database. In contrast, today's applications tend to use their own security mechanisms

(often via libraries like TLS), hiding the security semantics from potential in-network implementations (e.g. IPsec). In our model, the scheme defines ways for applications to express confidentiality, integrity and authentication policies, giving the implementation enough guidance to meet those needs. This allows implementations to eventually upgrade to newer security protocols. Of course, applications that want more control over security can implement it themselves.

The final responsibility of schemes is name resolution. Although the many existing identifier spaces and proposals for new naming systems [23, 24, 26, 31, 33] offer various advantages (and disadvantages), NetAPI mandates no specific naming scheme beyond the basic URI syntax. Using URIs, many different types of names can be expressed, including location-based identifiers, explicit addresses, globally-scoped flat names, domain-specific identifiers (e.g. private network addresses) or user-specific identifiers [20]. This general approach to naming has been adopted by other systems [35] due to its flexibility and extensibility. We expect some schemes to support naming systems that others do not (e.g. looking up a file in a peer-to-peer system), while other naming systems will be shared among schemes (e.g. DNS).

Lastly, the task of standardizing and implementing schemes is left to the community. For instance, the World Wide Web could be implemented by a set of distinct schemes, each providing a section of a traditional Internet communication (hypertext, audio, video). However, this may hinder optimizations, such as caching, that can shared among schemes. Similarly, if multiple types of communication are supported over a single protocol (e.g. interactive web browsing and non-interactive file downloads over HTTP), there is a choice between having a single scheme for this protocol and hinting application requirements to it through options, or having separate web browsing and file download schemes. These tradeoffs are inherent, and as such we leave them to domain experts in standards bodies. NetAPI aims only to ensure that there is flexibility in defining schemes, and our PANTS prototype provides one example of a functional set of schemes.

2.2 Messages

Messages in NetAPI are application-defined data units (ADUs) [17] containing binary data and a list of key-value *properties*. Properties are message fields interpreted by the scheme implementation, whose semantics are defined by the scheme. For example, in a *video://* scheme, each message may be a frame, and properties may include a `type` field saying whether the frame is a keyframe and a `timestamp` field.

The use of ADUs allows applications to divide content into logical units which can be treated individually by the network stack, similar to files in a filesystem and request-response pairs in HTTP. Properties let applications express semantic information about content without being coupled to a specific protocol encoding. NetAPI does not mandate how messages are encoded and ordered. For example, properties may map to flags in RTP or headers in HTTP. Messages may be concatenated into a TCP stream in a *web://* scheme, or may be unordered UDP packets in a *video://* scheme. Both encodings and transport protocols are free to evolve. One final advantage of a message-oriented API is that message reconstruction is performed by the scheme, eliminating a common source of code complexity, bugs and security vulnerabilities in networked applications.

2.3 Server Operations

Thus far, this section has focused on the needs of "client" applications, i.e. those interested in retrieving data or interacting with a service. For applications acting as a provider, the task is to respond to these client requests. One simple way to implement providers is a pair of listen and accept operations as in Sockets. The application calls **listen()**, providing a "listen URI" to listen on, which includes a scheme and possibly a local identifier such as a port. It then repeatedly calls **accept()** to obtain client connections, and call **get()**, **put()** and **control()** on the handles returned. However, NetAPI can also support network architectures with radically different content distribution mechanisms, such as DONA [31], which uses a publish-subscribe paradigm. Either the implementation may provide a **publish()** operation to publish messages, or it may require servers to open a client connection to a special URI and post messages. However, because content providers actively manage their servers and have an incentive to move to improved content distribution mechanisms, we do not strive to define a general-purpose content publishing API in NetAPI. Instead, we are content with a general-purpose client API and with identifying API features that create flexibility (schemes and messages).

3 Usage Examples

We now show several examples of how NetAPI supports popular Internet applications. We start with an in-depth look at web content retrieval, to demonstrate that all the details of the application can be accommodated by NetAPI. We then present several other applications for breadth.

3.1 Web Content Retrieval

The *web://* scheme is used for retrieving web content, with a naming format similar to HTTP URLs, *i.e.*, `web://<url>` plus optional URL-encoded key-value query parameters.

A web page is returned over multiple messages, because it may be arbitrarily large. The application calls **get()** repeatedly to receive chunks of the page, in the same way that the **read()** call on sockets returns chunks of bytes. These chunks are annotated with metadata such as position in the file, total file length and an end-of-file flag to aid reconstruction.

In addition to URIs, web servers also use cookies and HTTP headers like the user agent when they generate content for a request. These parameters may be passed to the **open()** call as options. In the same manner, the application may ask for security features like server identity authentication and content protection through options to **open()**. For access controlled resources, the application may sup-

ply a username and password. Exposing the credentials to the API allows the implementation to select security mechanisms most appropriate for a particular operating environment and to evolve them over time. For example, the implementation could add support for a single sign-on protocol such as Liberty [45] or use IPSec [44] without application modifications.

As an example, a web browser executes the following Python-like pseudocode to download a web page, supplying a cookie and requesting authentication of the source:

```
page = ""
handle = open("web://my.site.com/home.html",
              authenticate_origin = true,
              cookies = {"username": "john"})
while True:
  message = get(handle)
  page += message.data
  if message.is_end_of_file:
    break
print(page)
```

Clients can also submit data to a server through HTML forms that processes it and returns a response (as in HTTP POST). The application calls **put()** with the form data before calling **get()** to retrieve the response.

Network bindings. In well-connected environments, the *web://* scheme can be implemented over HTTP/1.1. The, **open()** call triggers an HTTP get request, and data is returned in **get()**. Submitting data to a server happens via HTTP POST. A successful response is encoded as a NetAPI message, mapping the HTTP headers into the key/value properties. A failure maps into an API error. If encryption is requested, the transactions are layered over SSL/TLS. HTTP authentication may also be used. Like today's HTTP clients, the implementation can use persistent connections and pipelining to optimize performance, letting applications use asynchronous **get()** calls to request multiple objects in parallel.

NetAPI also makes it natural to run the *web://* scheme using other network technologies, such as DOT [27], DTN [34], and BitTorrent [14]. To download a web object using BitTorrent, an initial resolution step identifies the content hash and location of the appropriate tracker for the publication URI, and then initiates the download process from peers. DOT would similarly transfer the object over the most appropriate transport method after locating it. In a DTN context, a well-connected proxy polls pre-selected (or on-demand subscribed) web sites for updates and proactively pushes the most recent versions to an offline client-side proxy that responds to **get()** calls with a locally cached copy.

Finally, unlike current HTTP libraries, NetAPI makes it possible to install central policies across *all* applications that use the *web://* scheme. For example, a user interested in blocking ads in both their web browser and their news reader may install a global proxy, without having to configure each application to use the proxy.

3.2 Multimedia Streaming

Multimedia content is naturally accommodated in NetAPI, enabling efficient distribution protocols in a wide variety of environments. For example, a URI in a *media://* scheme might contain a set of track descriptors, each with a reference to a URI in the *video-data://* or *audio-data://* scheme for streams with various encodings and various levels of quality. These per-track streams in turn contain multiple messages, one for each frame of the audio/video, with properties defining the frame content type and a time offset relative to the start of the video. A media player selects the tracks it desires, opens them, and calls **get()** repeatedly to receive frames. The player may also move forward and backward in the stream by calling a seek **control()** operation, and may obtain network statistics such as lost frames or jitter by calling a get-statistics **control()** operation.

Network bindings. On the Internet, video content is delivered in multiple ways, including HTTP, streaming protocols layered on UDP, and peer-to-peer protocols. NetAPI enables any of these methods to be used efficiently. Furthermore, the fact that name resolution happens below the API lets NetAPI take advantage of novel approaches for determining the nearest content server without application modifications.

In a DTN context, instead of streaming the video frame-by-frame, the whole video might be packed up into a single DTN bundle, then unpacked at the client and delivered frame-by-frame in response to **get()** calls.

Finally, on a mobile device, NetAPI can take into account the quality of the available network connection (cellular vs Wi-Fi) and any user policy settings (e.g. best battery life vs. best performance) to choose an appropriate level of quality for the stream, again with no application changes.

3.3 Syndicated Content

Syndication protocols such as RSS distribute news, blog posts, or other periodically updated items. NetAPI is a natural fit for this publish/subscribe design pattern. In the *news://* scheme, each URI identifies a news stream publication that contains news items (one per message), corresponding to the items in an RSS feed. Message properties identify feed origin, timestamps, and references to the complete articles.

As with multimedia content, clients call **get()** to obtain news items one at a time as individual messages in the order that they were published. If a client initializes having previously obtained and displayed some items, it can also check for new items by calling a **control()** operation, passing the identifier of the last message received.

Network bindings. RSS and ATOM are obvious choices to implement the *news://* scheme in well-connected environments. While a client has an open publication handle, the implementation periodically polls the feed over HTTP, parsing the XML items into NetAPI messages. Alternatively, the implementation could use a true subscription-based protocol like Corona [32], or bundle multiple items together in

Operating Environment	Resolution Mechanism(s)	Message Transfer Protocol(s)
Fixed Internet	DNS, BitTorrent Tracker, DHT, DONA	All current (and future) transport and application protocols
Delay Tolerant Networks	Opportunistic routing, configured proxy	DTN bundle protocol
Mobile Ad-Hoc Networks	Flooding, swarming, gossiping	Wireless broadcast protocols, gossiping
High-Performance Computing	Hard-coded lookup table	Protocols on a lambda

Table 1: Examples of environment-specific resolution and message transfer mechanisms.

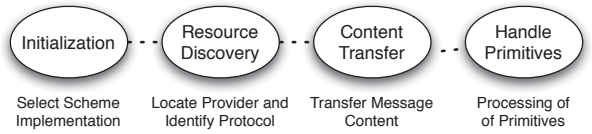

Figure 2: General tasks required from an implementation of NetAPI.

a single transfer over an intermittent network. The *news://* scheme can also naturally support publish-subscribe, data-oriented network architectures such as DONA. The server portion of NetAPI might support a **publish()** operation for these schemes as described in Section 2.3.

4 Implementing the API

In this section, we discuss the tasks involved in implementing NetAPI, and in the process underline the applicability of NetAPI across diverse network environments. Figure 2 shows the four high-level tasks in implementing NetAPI: initialization, resource discovery, content transfer and handling of primitives. Potential implementations differ widely depending on the needs of the application and the operating environment. Table 1 provides some examples of implementation options.

Initialization. On the **open()** call, the implementation identifies the appropriate scheme from the URI, loads the scheme module and returns a connection handle. One appealing aspect of NetAPI is that the scheme implementation is resolved at runtime, so it is possible to support dynamically loaded modules. This means that schemes may be implemented in user space as in FUSE [37], and even that scheme implementations may be updated while the system is running. In the latter case, existing connections over the scheme would continue using their instance of the old scheme module.

Resource Discovery. Once initialized, the implementation needs to locate the name specified in the URI. Name resolution depends on the infrastructure available in the operating environment. In common scenarios, the implementation uses an infrastructure like DNS to resolve the names. Resolution in dynamic environments like mobile ad hoc networks happens through flooding, swarming or gossip.

Content Transfer. The implementation must choose the right mechanisms for transferring the data depending on the semantics and needs specified in the scheme. For example, if a scheme requires reliability, the implementation must use a reliable transport protocol (e.g., TCP). In mobile devices with constantly changing network characteristics and multiple network interfaces, the implementation must pick the right communication interface. For example, a disruption-sensitive scheme should use the cellular interface as it likely has ubiquitous connectivity, while a throughput-oriented scheme such as file transfer can opportunistically use WiFi access points. When multiple applications are running, a central module may need to allocate resources between them.

Handle Primitives. Finally, the implementation must convey application data from and to the network. This requires transforming application messages into the format used by the transfer protocol or performing transformations of the data itself (e.g. compression or encryption). The degree to which this process is burdensome depends on how well-suited the scheme is to the particulars of the transfer protocol, in terms of its message units, formats and requirements. The implementation must also handle local buffering and state management, to handle rate mismatches between the network and application consumption. Finally, once the application closes the handle, the implementation can either release the allocated resources, or cache them opportunistically for future use.

5 PANTS: NetAPI for Mobile Phones

To demonstrate the benefits of NetAPI, we implemented a system for mobile devices. Networking for mobile is highly challenging, as devices must regulary switch between network interfaces and access points. This is often done at the cost of battery life and application usability, much to the chagrin of users. From a developer's point of view, managing mobile networking is complex, leading to buggy applications and poor performance. The mobile environment thus provides an ideal usage scenario for NetAPI. We built a mobile phone implementation of NetAPI called Protocol Aware Network Technology Selector (PANTS). We designed PANTS for the jailbroken iPhone platform, which gave us a BSD environment and the ability to use many existing network libraries. We used PANTS to build disconnection tolerance, content-shaping, and power-saving features in our scheme implementations without modifying applications.

The goal of PANTS is to demonstrate the flexibility, adaptability, and usability of NetAPI. For flexibility, PANTS takes

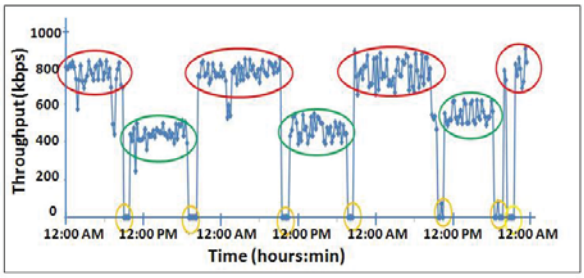

Figure 3: Variation in Wi-Fi Throughput for *DS* as a function of time. Data collected at similar locations are marked in clusters.

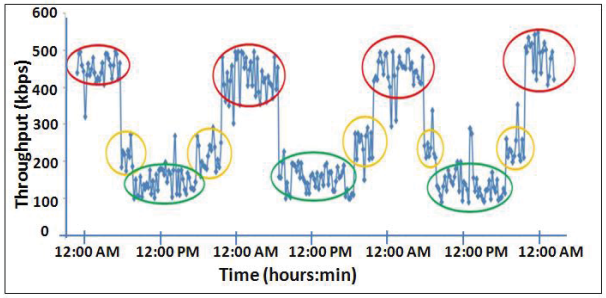

Figure 4: Variation in throughput for cellular interface for *DS*. Data collected at similar locations and times are marked in clusters.

advantage of network characteristics and history to provide applications with better service. For adaptability, PANTS provides an area to implement innovative new schemes. This allows NetAPI applications to utilize these innovations without modification. Lastly, PANTS demonstrates the simplicity of writing these NetAPI applications.

In this section, we first give a brief overview of the difficulties and opportunities available in mobile computing. This is used as a justification for this particular use case. We then go into depth about the PANTS implementation itself. Implementation of the schemes providing benefits to mobile devices follows in section 6.1.

5.1 Motivation

We present the variability as well as the repeatability of cellular and Wi-Fi characteristics, motivating prediction modules, network selection, content shaping, security, and disconnected operation. We used a data set (*DS*), collected by a user in his normal mobility patterns, for a period of three days in the San Francisco Bay Area.

DS was collected by a working professional who spent the significant fraction of his time either at his residence, work or commute. *DS* used active profiling, i.e., downloading a file across the internet, to measure the throughput on both the Wi-Fi as well as the cellular interfaces. We use only the "preferred" networks for Wi-Fi association. *DS* logged the location as well as throughput every ten minutes.

Wi-Fi Throughput Our data set indicates high spatial correlation and low temporal variance.

Wi-Fi throughput exhibits low spatial variability, i.e., at a given location, the Wi-Fi throughput is relatively invariant. Figure 3 plots the best available throughput as a function of time for *DS*. Data collected at roughly similar locations are clustered. The top clusters correspond to throughput at home and the bottom clusters correspond to throughput at work. Throughput drops to zero during commute as there is no Wi-Fi connectivity available using preferred access points. Note the low variability in values among points in the same cluster and location.

Cellular Throughput Cellular throughput exhibits both spatial as well as temporal variance. Figure 4 plots the cellular throughput with time. The throughput observed while in his residence (top and middle clusters) is distinctly higher than while at work (bottom clusters) and this is likely because the load on the cellular network is much higher in industrial areas with multiple simultaneous users. Another interesting observation is the high throughput experienced during off-hours (top clusters). These results indicate that the cellular throughput is a function of space as well as time of day. Also, note that the cellular throughput never drops to zero indicating ubiquitous connectivity with no disruption.

Conclusion These results demonstrate that there are many advantages available for mobility. To take advantage of these, we need a centralized network selection daemon that knows both the connection requirements of the application and the likely properties of the available connections. With low-throughput applications, they may prefer to stay on cellular during areas of good connectivity. High-bandwidth applications may wish to avoid the cellular connections all together, lengthening the device's battery life, or change protocols to use lower-bandwidth links. Security can be enforced by only allowing trusted connections we expect along our path. Lastly, delay tolerant applications can wait for network characteristics that allow for a pleasant user experience, rather than attempting connections over shoddy links. NetAPI allows for all of these.

5.2 System Description

The Protocol Aware Network Technology Selector (PANTS) is implemented as a Twisted Python [54] daemon. It runs on numerous mobile computing platforms. The primary supported platform is a jail-broken Apple iPhone. The PANTS daemon also runs on Nokia's N810 Maemo mobile Linux platform [46], as well as Linux and Mac OS X laptops. Figure 5 is details the architecture of PANTS.

We also have an implementation of PANTS in the i-mate HTC PDA running Windows Mobile 5.0 using the C#.NET framework. It implements the most important PANTS features.

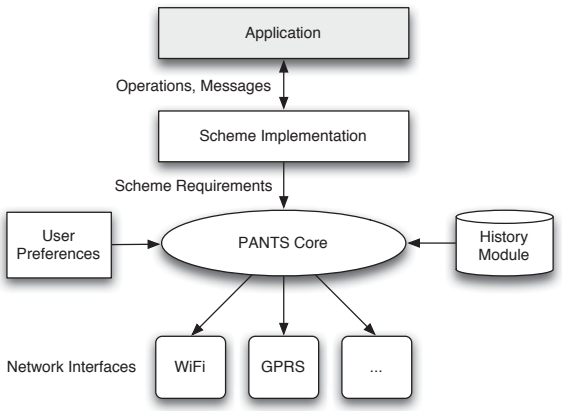

Figure 5: The PANTS architecture

Application Interface. Applications communicate with PANTS over any one of three different RPC protocols: text, XML, and python pickling. Each of these is abstracted from the schemes, so that any RPC format may be used to communicate with PANTS. Most languages provide basic XML RPC libraries, allowing them to communicate with PANTS. Extending PANTS to use new RPC or IPC mechanisms is simple.

User Preferences. The user preferences module informs PANTS about what network properties the user would prefer. This module is configurable, allowing for a wide range of user input. We decided to implement a low-granularity solution, allowing the user to optimize for performance or battery life. This data is periodically read by PANTS and used to assist in network selection.

Network Manager. PANTS requires a network interface object for each physical network interface. This object interfaces with the specific interface, informing PANTS of the properties of that interface. For instance, the iPhoneWifi object scans the available Wi-Fi access points, as well as computes the expected bandwidth of each access point. This information is then returned to PANTS to assist in selecting the appropriate network interface. The interface may also cache the last scan or return the expected available access points determined by querying a location database with GPS data (see Section 5.1).

PANTS itself queries each configured network interface object for available networks and their properties. To select a network interface, it determines the most appropriate network (see Section 5.2).

History Module. The history module of PANTS is in the process of being developed. We have only a simple implementation that provides a place to store and compile statistics. The module is queried before every network selection and informed of the choice following it. This would allow us to skip networks we know require MAC or browser authentication, or are highly intermittent.

Schemes. Schemes implement the NetAPI primitives: **open**(), **close**(), **put**(), **get**() and **control**(). The scheme is the primary area for semantics specifications, as the particulars of the communication with the client API is defined by the scheme itself. On a command requiring a connection, the scheme asks PANTS for a network connection. PANTS returns this connection, as well as the properties of this connection. With this information, the PANTS scheme is able to intelligently schedule data transfers to optimally deliver content to the application.

PANTS Core. The PANTS core has a number of jobs. It creates schemes to handle user connections. It merges user and applications requirements, network history, and network properties to select a network connection. Lastly, it periodically checks if any of these have changed, and if so, select a new optimal network interface.

For the first task, we create a configurable table of schemes. When PANTS receives an open command, it parses the URI to determine the correct scheme handler and instantiates it. This handler is given the client information, and all communication from the client application now goes through the handler.

For the second, we merge all of the application requirements into one set of requirements. This is done by taking the maximum of each particular application demand. This treats all schemes as equals, which need not be the case. However, this fulfills the requirements of all of the individual schemes.

Next, we apply heuristics to decide if the user or the application requirements take precedence. The first heuristic takes into account that the user likely wants their applications to continue. Thus, if any application is not disruption tolerant, we either choose the more mobile connection or keep the existing connection. Secondly, the user's demand for power or performance is viewed as more important. We then decide on the appropriate interface based solely on that metric. Without user input, we just order the application demands by strength, and then select based on the most important feature. With this, we are able to select the optimal interface.

Lastly, the periodic network check is scheduled for every ten seconds. It simply reruns the network selection algorithm, and compares the result to the existing interface. If they are the same, we do nothing. If they differ, the current open schemes are informed that their connection is about to be revoked, allowing for them to make changes before the adoption of the new interface.

6 Evaluation

We implement two schemes, *web* and *voip*, and show how applications can easily and automatically take advantage of the features in the schemes. We also measure the implementation complexity and performance overhead of PANTS.

6.1 Support for Innovation

To evaluate our PANTS prototype, we implemented two sample schemes, *web* and *voip*. Our sample applications using these schemes could automatically take advantage of the features in the schemes. To achieve backward compatability, we also implemented a *socket* scheme and an http proxy over the *web* scheme.

6.1.1 Web Scheme

We implemented the scheme itself and two applications using it: a File Downloader and a News Reader. We begin by briefly describing the *web* scheme and then the sample applications. Later we explain how we extended the *web* scheme to add various networking functionalities like disconnection-tolerance and power-efficiency, automatically resulting in applications taking advantage of them.The *web* scheme, in its basic form, is mostly a wrapper around Twisted Python's HTTP libraries, with some extra logic that we have added below to extend the basic scheme. The scheme handler makes an HTTP request, parses select fields of the header to compute the file size, and passes back the data read to the application until the download is complete.

File Downloader: Our first application was a File Downloader (Figure 6(a)) that fetches a large file over HTTP. This is representative of applications such as music stores, video stores, and software updaters. This is an example of an application that would clearly benefit from features like disconnection tolerance and smart resumption of downloads.

News Reader: The News Reader (Figure 6(b)) represents a more interactve application. It downloads an RSS feed every 60 seconds and displays a list of stories. The user may click a story to view its summary in a HTML content control. In addition to the disconnection tolerance, we used PANTS to implement a power-saving policy: download media files (like images or audio) only when Wi-Fi is available.

Both applications use less than 10 lines of code to interact with PANTS. The following listing shows the relevant code from the Downloader application (in Python):

```
fileData = ""
client = PantsClient(self.url)
while True:
  result = client.get()
  fileData += result["data"]
  curPos = float(result["currentPosition"])
  fileLength = float(result["fileLength"])
  self.progress = curPos / fileLength
  if result["done"]:
    break
```

The code simply opens a client and calls `get` on it repeatedly to receive portions of the file. The `result` object returned by `get` contains a field called `"done"` on the last portion. It also contains fields indicating the current position in the file and total file length, which are used to display progress. The code for the News Reader is very similar.

(a) Downloader (b) News Reader

Figure 6: PANTS sample applications.

Adding Disconnection Tolerance The initial version of the *web* scheme attempted to open a connection right away and return the data to the client, raising an error otherwise. We made the scheme resilient to disconnection through two mechanisms:

1. If a connection to the server cannot be made, or is broken, the scheme will retry connecting later. Any `get` calls made by the application during this time will block. We consider putting a timeout on the `get` calls or an optional parameter to prevent blocking.

2. On reconnection to the server after a disconnection, the scheme handler uses the HTTP `Range` parameter, supported by most web server implementations, to request only the range of bytes starting from the last received position to the end of the file (similar to web browsers like Mozilla Firefox).

With these changes, our sample applications automatically became tolerant to disconnections due to exiting a coverage area or moving between two hotspots. No changes to the applications were required, because the *web* semantic defines all `get` calls as blocking.

We tested this functionality by starting a download in the Downloader application over EDGE and placing the phone inside a Faraday cage that stopped the cellular wavelengths to simulate a disconnection. We waited until the phone indicated "No Service" and then took the phone out of the container to check that the download resumes where it left off.

Constraining Downloader to Wi-Fi To further illustrate the flexibility of PANTS, we extended the *web* scheme with a feature that is missing in other downloader implementations we are aware of: the ability to constrain downloads to only occur over Wi-Fi to optimize for battery life. This required

a very simple code change to the scheme handler – when the handler requests a network interface to open a connection on, it only uses a connection of type Wi-Fi if the power saving policy is enabled. Because of the download-resumption functionality from the previous section, the semantics seen by the application are unchanged – it receives the next chunk of data every time it calls `get`, and the `get` call is blocking. This let the Downloader application opportunistically download a file over Wi-Fi only, with no code modifications.

Adding Content Cognizance to News Reader We also demonstrate PANTS's ability to transform the content received by the application. When a power-saving policy is selected by the user, we alter the page content received by the application to remove the HTML image tags, disabling images and thus conserving download bandwidth. This is a basic example meant simply to convey the point that PANTS can control content quality to enforce policies.

We have also extended the News Reader to be one-level recursive in not only fetching the RSS feed but also the webpages and enclosures in the RSS items and storing it in the local cache. Such a feature is particularly useful when the mobile device has intermittent network connectivity as it insulates the user's experience from real-time connectivity characteristics. To enforce the"best battery life" policy, our news reader downloads the html content over the cellular interface but fetches the media (like images, podcasts) files only when it has Wi-Fi coverage.

Nonetheless, modifying HTML pages in-flight is not unheard-of: for example, ad-blocking proxies such as Privoxy [19] filter ads out of web pages to make the browsing experience faster and more pleasant. In future work, we plan to extend this content-modification ability to a multimedia application, and to explore other ways to apply it to web browsing, such as downloading lower-quality versions of pages by switching the browser agent to a mobile browser, or prioritizing content to download text before images. There is also an opportunity for caching content inside PANTS.

HTTP Proxy The PANTS software ships with a limited HTTP proxy that receives HTTP communications and forwards them through the *web* scheme. This is as simple as it seems, with the only difficulty being the lack of communications between the application and PANTS. Because of this, we assume all HTTP proxy communications to have the same needs, those being high bandwidth, disruption tolerant links. However, all HTTP traffic that uses the proxy gains the benefits of the *web* scheme, with disruption tolerance, content caching, and bandwidth-informed content contraints.

6.1.2 VoIP Scheme

To test the applicability of PANTS and NetAPI to multimedia tasks, we implemented a voice-over-IP (VoIP) application. The application itself is very simple, sending a URL that defines the server to connect to, with username and password.

The application makes use of the *voip* scheme. This scheme utilizes Twisted Python's own sip protocol libraries to register and communicate with an IP private branch exchange (PBX) system. When the PBX receives a call destined for our application, PANTS and the PBX negotiate an RTP connection using the Session Description Protocol (SDP). With this scheme, we aim to demonstrate the protocol selection and security benefits of PANTS.

Adding Encoding Selection When receving a new call, both the PBX and PANTS signal the encodings available for use in this communication. We decided to try to limit the bandwidth used when on a GPRS connection, while using a high bit-rate encoding on Wi-Fi networks.

This is simple in PANTS, we add a check in the call reception handling code. Because SDP is used to negotiate the encoding, we filter the high-bandwith encodings from that communication when on a low-bandwidth link. This forces the PBX to use only low-bandwidth encodings. Likewise, when on Wi-Fi, we do no such filtering.

With this, the application is able to dynamically reduce its network load based on network properties. In this specific case, the application will still need to know about the wire protocols, limiting the effectiveness. However, the encodings themselves can be moved into PANTS, providing the application with one concrete encoding regardless of the one chosen by the SDP negotiation. Then the application can utilize new, lower-bandwidth links or encodings with no modification.

Adding Security During the SIP registration, the PBX server offers available authentication techniques. Again, PANTS can make intelligent decisions for this choice, without the application's involvment. For secure VoIP communications, PANTS may disallow communications over unknown or unsecured wireless communications. It may select higher security authentication, such as encrypting the entire communication, when on unsecured or untrusted mediums. These policy decisions can be enforced by companies or governments, forcing security onto their employees.

6.1.3 Socket Scheme

We implemented the *socket* scheme, which allows for a traditional socket interface to PANTS, and facilitates any legacy application using sockets to be easily ported to use the PANTS architecture. This scheme performs no optimizations as we do not expect applications to specify their needs or semantics. This scheme is essentially for porting legacy applications enabling easy adoption of PANTS.

6.2 Implementation Complexity

As shown at the start of Section 6.1, the amount of code required in our PANTS client applications that used the *web* scheme was minimal – about 10 lines in each. The basic *web* scheme implementation took about 100 lines of code, mostly by reusing Twisted Python's HTTP library. The final *web* scheme, with the features of disconnection tolerance, downloads only over Wi-Fi networks and content cognizance, was

Feature	Lines
PANTS code in sample apps	10
Initial *web* scheme	100
Disconnection tolerance / resuming downloads	100
Wi-Fi-only downloads	5
Remove images in news reader if GPRS	10
Final *web* scheme	170
Initial *voip* scheme	310
Encoding selection	15
Final *voip* scheme	325
socket scheme	160
Socket client	110
iPhone interface drivers	160

Table 2: Implementation complexity of various PANTS features, in lines of Python code.

170 lines. Table 2 summarizes these results.

Although our implementations of the *web* scheme, sample applications, interface drivers, and PANTS in general are prototypes and may miss certain corner cases, the results show that PANTS is a promising architecture for supporting multiple schemes and new policies for these schemes.

6.3 Performance

Our primary goal with PANTS was to explore the flexibility of NetAPI, not to achieve high performance. As such, we used a rapid prototyping language (Python), whose performance is worse on a mobile phone. We were also unable to use optimized Python interpreters such as Psyco [3] which do not have an iPhone port. Nonetheless, we evaluated the performance of PANTS to show that overheads are nonexistent in some situations and tolerable for most applications.

We evaluated the performance of PANTS by measuring the time it takes to download a file through PANTS compared to downloading it through `wget` (and sending the output to `/dev/null`). Table 3 shows the results for four different scenarios – a laptop connecting over Wi-Fi downloading a small file, the same laptop downloading a large file, and an iPhone connecting over Wi-Fi and over EDGE. We see no statistically significant difference between PANTS and `wget` in all scenarios except for the Wi-Fi running on the iPhone. The slowdown here is due to PANTS becoming CPU-bound and not being able to process the data as fast as it arrives. However, the results on the laptop and over EDGE show that there is no fundamental performance limitation in PANTS. We expect a native implementation (unlike the unoptimized interpreted Python environment) to be able to download data over Wi-Fi as fast as `wget`.

We are also building a Windows Mobile implementation of PANTS using C#. It currently supports fewer features than the Python version but can perform file transfers. Table 4 shows the results of downloading a small and large file through the Windows Mobile version of PANTS on an iMate cell phone. We see significantly smaller overheads (at most 10%) because of C#'s highly optimized runtime environment.

Scenario	File	PANTS Mbps	Direct Mbps
Laptop Wi-Fi	440 KB	2.02 (.24)	1.8 (.12)
Laptop Wi-Fi	6.7 MB	4.13 (.56)	3.83 (1.48)
iPhone Wi-Fi	440 KB	1.01 (.13)	1.49 (.13)
iPhone EDGE	440 KB	0.04 (.01)	0.03 (.01)

Table 3: PANTS download throughput performance compared to `wget`, for different file sizes over different network interfaces. Standard deviations shown in parentheses.

Scenario	File	PANTS Mbps	Direct Mbps
Cellular	300 KB	0.04	0.04
Cellular	4.2 MB	0.05	0.05
Wi-Fi	300 KB	2.66	3
Wi-Fi	4.2 MB	3.32	3.32

Table 4: PANTS download throughput performance compared to direct downloads, for different file sizes over different network interfaces for our Windows Mobile implementation using C#.NET.

7 Discussion

In this section, we discuss the implications of NetAPI adoption in terms of the design and deployment of applications.

7.1 Capturing Application Intent

The core benefit of NetAPI – the decoupling of applications from network protocols – presents its main challenge: how to best capture application intent, while still providing sufficient flexibility for scheme implementations to use network protocols efficiently. The NetAPI interface exposes three primary mechanisms for applications to communicate their intent: *i)* the content structure (connections, messages, and their properties), *ii)* primitives (actions), and *iii)* the scheme (and its specification).

Scheme-related design decisions involve clear tradeoffs, to which we do not claim to have a universal recipe. Indeed, for any major application we believe these decisions will be made in forums such as IETF working groups, probably engendering significant debate. We would view this as a positive outcome, as it testifies as to the generality of NetAPI: while the contracts between applications and the network are subject to non-trivial specification work, the foundations of the API should sustain future applications and future networking technologies.

7.2 Implications of Adoption

NetAPI decouples applications from protocol implementations, and hence, presents a new deployment model for applications that could have ramifications in its adoption. Protocol designers may be required to write two specifications: a scheme specification to define the semantics of the NetAPI

primitives and the application model, and a protocol specification to define a particular network protocol that implements the scheme. Although clearly there is an increased specification burden, specifying the abstract communication needs separately from the actual network protocol is exactly what is required to enable the innovation and evolution of the network. In most cases, we expect developers to utilize existing schemes. The number of existing schemes in the operating system would increase with NetAPI adoption. Once this has reached critical mass, creating applications will become easier, as many desired features (such as delay tolerance) will be implemented by the schemes.

Today the task of deploying a radically new network technology that affects multiple layers is next to impossible. Although certainly not the only challenge, the need to support the already-deployed and in-use applications is clearly one of the key barriers to adoption of disruptive network technologies. With NetAPI this task would be much easier, as applications would not be responsible for supporting the new network technology, rather the network manager would be. Hence, as a part of experimenting with or deployment of a new network technology, critical popular scheme implementations would be updated to take advantage of it.

Interestingly, NetAPI itself is incrementally deployable, because it can wrap existing protocols. Our PANTS prototype interacts with legacy servers and provides a proxy that can be used by existing HTTP applications (Section 6.1.1) as well as a *socket://* scheme that lets NetAPI applications use TCP and UDP (Section 6.1.3). Our API could therefore be used as the networking API of a mobile operating system. In the mobile phone space in particular, we believe that NetAPI-like concepts have a potential to be adopted because application developers are used to working against restricted APIs and platform developers need control over application behavior to ensure smooth functioning of the device (e.g. ability to take calls at any time). One commercial API with high-level networking abstractions is the WebOS platform developed by Palm [49], which will launch in 2009. WebOS requires applications to be written in JavaScript and HTML like web pages, but gives them disconnection tolerance through the HTML 5 client-side storage API [56].

8 Related Work

Research proposals. A number of systems have been proposed in the research literature that both inspired certain aspects of the design of the API and also serve as examples of the types of systems that would be enabled by widespread adoption of our API.

DONA [31] is a clean-slate networking design built around a name-based anycast abstraction to access data objects without knowledge of their location in the network. The DOT proposal [27] provides a framework by which largely unmodified applications can leverage a dynamic mapping to a particular transport method when transferring large data objects. Their work supports our belief that many applications are agnostic of the particulars of transfer methods, and that a dynamic binding of such methods is beneficial for optimal behavior in a range of environments.

To operate effectively in environments with long round trip times, DTN systems use larger "bundles" of data that they can transmit as entire units, thus requiring that the system have a better notion of the application's intent when interacting with the network. Haggle [25] shares several characteristics with DTN. It argues that mobile ad-hoc networks require a new way of thinking about data and networking, relying on the communications stack to opportunistically select one of a variety of transport methods to convey an object from point to point. KioskNet [12] provides multiple transport methods in the setting of rural Internet kiosks.

A handful of other proposals have demonstrated the benefits of expressing application communication semantics to the network stack, including Scalable Data Naming [50] and Structured Streams [36]. Declarative networking [21] shares our goals of expressing the intent instead of a precise mechanism for the network, but focuses more on the implementation of network protocols rather than the expression of a wide range of application-relevant semantics.

Middleware systems. Many middleware systems have been developed that offer applications a higher-level API than Sockets, and several adopt the publish/subscribe paradigm (*e.g.*, Tibco [53], IBM WebSphere MQ [41], DCOM, CORBA, and J2EE [52]).

Our proposal does not aim to compete with these or any other middleware systems; we advocate a new programming interface, not a proposal or mandate for a specific implementation of that interface. NetAPI is a closer match to the language-specific messaging interfaces of modern programming languages (*e.g.*, Java Message Service [51]).

Tuple space systems such as Linda [38], JavaSpaces [22] and T Spaces [57] all provide a generic and powerful interface, to the extent that they have been proposed as a generic communication interface [38]. However, they require an agreed upon address or naming format and have scalability limitations due to their semantics.

One final commercial platform of particular interest is the Palm WebOS [49] for mobile phones. In WebOS, the high-level API offered to developers of mobile applications is HTML5, CSS, and Javascript; developers write applications as if they are web pages. However, applications also gain disconnection tolerance through the HTML 5 client-side storage API [56] (which is similar to Google Gears [2]). Although applications must manually control which data they place in the client-side database provided by the storage API and when they synchronize the database with the Internet, this example illustrates the need for disconnection tolerance in mobile applications and the willingness of commercial developers to forsake the Sockets API for a higher-level API.

Network selection systems. Network selection was an early research problem, tackled in large part by the Berkeley BAR-

WAN project [28]. In this project, techniques directly related to our work, such as handoff between wireless networks [16], were developed. With increasing proliferation of Wi-Fi networks, detecting the available access points and selection of the best among them has become crucial. Wardriving databases [5, 6, 9, 15] provide a simple mapping between Wi-Fi beacons and GPS coordinates. Virgil [8] and Context-for-Wireless [11] scan and learn various characteristics of the access points like throughput and latency for future prediction, as well as scheduling network activities, e.g., Breadcrumbs [7] and MobiSteer [55].While we expect that our network manager and schemes will highly utilize many of the above-mentioned techniques, our goal is to design the system along with the API to make this possible.

Most mobile operating systems have daemons that control the network, allowing for transitions between Wi-Fi and cellular connections, e.g., Nokia's ICD [47] and Gnome's Network Manager [40]. Some applications, like web browsers, use these to implement disruption-tolerance by switching to local operations until informed of a new network connection. Our system is more general, allowing applications to influence network selection based on their requirements.

Mobility Aware Applications. Dealing with changing network characteristics in mobile settings, and informing applications to adapt accordingly has been proposed in prior work. The framework in [10] detects the available network interfaces and its changing characteristics and presents them to applications. Odyssey [13], Mobiware Toolkit [48] and the frameworks in [42] and [18] focus on the complementary aspect of defining mechanisms for applications adaptation. Odyssey [13] models the adjustment of applications to general changes in resources around the high-level concepts of agility and fidelity. In addition to involving network elements (e.g., routers) in detecting mobility, Mobiware [48] defines a utility function relating the application's quality and bandwidth changes. Likewise, the framework in [42] provides applications with a feedback loop that helps map from network-centric quality to application-centric quality. While application adaptability is a key concept proposed in these papers, they concentrate primarily on network bandwidth as a resource, and adaptation consequently is in terms of consuming lower-quality objects (e.g., lower quality video stream, or compressed images). In contrast, NetAPI is generic and concrete with larger goals: (1) it allows varied kinds of adaptation like postponement (e.g., DTN) and enables applications to present their own requirements and semantics systematically through schemes, and (2) it manages resources for a larger set of metrics, including bandwidth, power-efficiency, latency, jitter and disruption.

9 Conclusions

NetAPI is an interface that decouples applications from network mechanisms to foster innovation below the API. Unlike the Sockets API, NetAPI *hides implementation mechanisms* from the application and *captures application intent* to let the network stack understand application requirements. Apart from facilitating deployment of new network technologies, NetAPI is attractive because it lets the network stack *adapt* to environmental conditions and it enables *centralized policies* over how the network is used.

We demonstrated the utility of NetAPI through a prototype for mobile phones called PANTS. PANTS selects the network interface appropriate for each application based on the communication scheme it uses, and can provide features such as disconnection tolerance, content shaping and power-saving policies without application modifications. It also simplifies user management of mobile networking by providing a global "best performance" versus "best battery life" setting. Furthermore, although NetAPI is an architectural idea, it can be deployed incrementally. Our prototype interacts with legacy servers over existing protocols.

Although NetAPI is a first step towards a more flexible communication API, we believe that it captures key features that enable an API to support innovation: communication schemes, extensible names (URIs), and messages.

10 References

[1] Gmail filesystem. `http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html`.

[2] Google gears api. `http://code.google.com/apis/gears/`.

[3] Psyco. `http://psyco.sourceforge.net/`.

[4] Ssh filesystem. `http://fuse.sourceforge.net/sshfs.html`.

[5] Wi-Fi Hotspot Locator. In *http://jiwire.com*.

[6] WIGLE: Wireless Geographic Logging Engine. In *http://wigle.net*.

[7] A. J. Nicholson and B. D. Noble. BreadCrumbs: Forecasting mobile connectivity. In *Mobicom*, 2008.

[8] A. J. Nicholson et al. Improved access point selection. In *ACM/USENIX MobiSys*, June 2006.

[9] A. LaMarca et al. Place lab: Device positioning using radio beacons in the wild. In *Proc. Pervasive 2005*, 2005.

[10] A. Peddemors et al. A Mechanism for Host Mobility Management supporting Application Awareness. In *MobiSys*, 2004.

[11] A. Rahmati and L. Zhong. Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer. In *Proceedings of ACM/USENIX MobiSys*, June 2007.

[12] A. Seth, D. Kroeker, M. Zaharia, S. Guo, S. Keshav. Low-cost Communication for Rural Internet Kiosks Using Mechanical Backhaul. In *Proc. MOBICOM 2006*, September 2006.

[13] B. D. Noble. System Support for Mobile Adaptive Applications. In *IEEE Personal Communications*, 2000.

[14] BitTorrent, January 2008. `http://www.bittorrent.com`.

[15] Bychkovsky, V. et al. A measurement study of vehicular internet access using in situ wi-fi networks. In *Proc. 12th Ann. Int. Conf. Mobile Computing and Networking (MobiCom)*, 2006.

[16] Ramón Cáceres and Venkata N. Padmanabhan. Fast and scalable handoffs for wireless internetworks. In *MobiCom '96: Proceedings of the 2nd annual international conference*

on Mobile computing and networking, pages 56–66, New York, NY, USA, 1996. ACM.
[17] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. of ACM SIGCOMM '90*, pages 200–208, Philadelphia, USA, 1990.
[18] D. Andersen et al. System Support for Bandwidth Management and Content Adaptation in Internet Applications. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.
[19] Privoxy Developers. Privoxy. `http://www.privoxy.org/`.
[20] B. Ford et al. Persistent Personal Names for Globally Connected Mobile Devices. In *Proc. of OSDI 2006*, Seattle, WA, USA, November 2006.
[21] B. T. Loo et al. Declarative routing: extensible routing with declarative queries. In *Proc. of ACM SIGCOMM '05*, pages 289–300, Philadelphia, PA, USA, 2005.
[22] E. Freeman et al. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Professional, July 1999.
[23] H. Balakrishnan et al. A Layered Naming Architecture for the Internet. In *Proc. of ACM SIGCOMM '04*, pages 343–352, Portland, OR, USA, August 2004.
[24] I. Stoica et al. Internet indirection infrastructure. In *Proc. of ACM SIGCOMM '02*, August 2002.
[25] J. Su et al. Haggle: Clean-slate Networking for Mobile Devices. Technical Report UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, January 2007.
[26] M. Walfish et al. Untangling the Web from DNS. In *Proc. of NSDI '04*, pages 225–238, San Francisco, CA, USA, March 2004.
[27] N. Tolia et al. An Architecture for Internet Data Transfer. In *Proc. of NSDI '06*, pages 253–266, San Jose, CA, USA, May 2006.
[28] R. H. Katz et al. The bay area research wireless access network (barwan. In *In Proceedings Spring COMPCON Conference*, pages 15–20, 1996.
[29] S. Leffler et al. An Advanced 4.4BSD Interprocess Communication Tutorial.
[30] T. Berners-Lee et al. RFC 3986: Uniform Resource Identifier (URI): Generic syntax. RFC 3986, IETF, January 2005.
[31] T. Koponen et al. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of ACM SIGCOMM '07*, Kyoto, Japan, August 2007.
[32] V. Ramasubramanian et al. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *Proc. of NSDI '06*, San Jose, CA, USA, May 2006.
[33] W. Adjie-Winoto et al. The Design and Implementation of an Intentional Naming System. In *Proc. of SOSP '99*, Charleston, SC, USA, December 1999.
[34] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proc. of ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003.
[35] K. Fall and S. Farrell. Dtn: An architectural retrospective. *IEEE Journal on Selected Areas in Communications*, 26(6), June 2008.
[36] B. Ford. Structured Streams: a New Transport Abstraction. In *Proc. of ACM SIGCOMM '07*, Kyoto, Japan, August 2007.
[37] Filesystem in Userspace (FUSE), January 2008. `http://fuse.sourceforge.net`.
[38] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of ACM*, 35(2):97–107, 1992.
[39] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.
[40] Gnome. Network Manager. `http://projects.gnome.org/NetworkManager/`.
[41] IBM. WebSphere MQ, January 2008. `http://www.ibm.com/software/integration/wmq/`.
[42] J. Bolliger and T. Gross. A Framework-based Approach to the Development of Network-Aware Applications. In *IEEE Transactions on Software Engineering*, 1998.
[43] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, 2002.
[44] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, IETF, November 1998.
[45] Liberty Alliance Project, January 2008. `http://www.projectliberty.org`.
[46] Maemo Community. Maemo mobile linux. `http://maemo.org`.
[47] Nokia. Internet connection daemon. Unpublished.
[48] O. Angin et al. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. In *IEEE Personal Communications*, 1998.
[49] Palm. webos. `http://developer.palm.com/`.
[50] 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*, pages 391–400, Bristol, England, September 1998.
[51] Sun Microsystems. Java Message Service (JMS), January 2008. `http://java.sun.com/products/jms/`.
[52] Sun Microsystems. Java Platform Enterprise Edition (EE), January 2008. `http://java.sun.com/javaee/`.
[53] Tibco. Tibco Enterprise Messaging Service, January 2008. `http://www.tibco.com/software/messaging/`.
[54] Twisted Matrix Labs. Twisted event-driven network engine. `http://twistedmatrix.com/trac/`.
[55] V. Navda et al. MobiSteer: using steerable beam directional antenna for vehicular network access. In *ACM/USENIX MobiSys*, June 2007.
[56] WHATWG. Html 5 draft recommendation - structured client-side storage. `http://www.whatwg.org/specs/web-apps/current-work/multipage/structured-client-side-storage.html`.
[57] P. Wyckoff. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.