Enabling Innovation Below the Communication 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-2009-141
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-141.html

October 19, 2009

Acknowledgement

This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley Reliable Adaptive Distributed systems lab sponsors: Sun Microsystems, Google, Microsoft, HP, Cisco, Facebook and Amazon.

Enabling Innovation Below the Communication API

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

†University of California at Berkeley, ‡Nicira Networks

ABSTRACT

Innovation in the network is notoriously difficult due to the need to support legacy applications. We argue that this difficulty stems from the API used to access the network. The ubiquitous Sockets API lets applications choose from a number of communication mechanisms, but binds them tightly to their chosen mechanism (e.g. specifying a destination using IPv4). Applications must therefore be modified in order to benefit from new network technologies. To address this problem, we propose a new communication API called NetAPI that lets applications specify their communication intents without binding to particular network mechanisms, enabling evolution below the API. We have built a NetAPI prototype for the iPhone, and use it to show that we can add disconnection tolerance, content shaping and power saving policies under NetAPI without application modifications.

1 Introduction

Virtually all network applications access the network through the *Sockets API* [17], which was developed for BSD UNIX over twenty years ago. The Sockets API lets clients select between a number of network technologies, but it binds them tightly to their chosen mechanism (e.g. specifying a destination host using IPv4). 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. The Internet for which Sockets were designed was primarily used for file transfer between static hosts. Today's Internet usage is far more diverse, with a variety of end devices (e.g., servers, desktops, laptops, phones) accessing a variety of content (e.g., documents, voice, video) through a variety of applications (e.g., email, web, IM) and network technologies (e.g., Ethernet, 802.11, cellular). The challenges introduced by this diversity, including mobility, naming, content delivery, multiple network interfaces, and intermittent connectivity, have inspired numerous research efforts over the past decade [3,12,15,16,21]. However, these advances have proven difficult to deploy without application modifications, and so remain largely unadopted.

To understand the limitations of the Sockets API, consider another API used to access system resources: the filesystem API for accessing storage. Although it is nearly as simple as Sockets, the filesystem API has fostered far more innovation since the time it was introduced. Today, applications using the filesystem API can take advantage not just of new types of storage media and placement algorithms, but also of filesystems cached in memory (buffer cache), served over the network (NFS), striped redundantly across disks (RAID), partitioned across commodity machines (GFS [26]), or employing advanced deduplication logic to conserve space (NetApp). The same `ls`, `cat` and `vi` applications written for BSD UNIX will work over this wide range of storage systems without even needing to be recompiled.

What features of the storage API made it so much more capable of supporting innovation than the network API? First, the filesystem API *hides storage technology details* from the application (e.g. locations are given using paths, not block numbers), whereas the Sockets API requires applications to invoke specific network mechanisms (e.g. choose among IPv4 and IPv6 address families, use TCP or UDP). Second, the filesystem API captures 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), whereas the Sockets API exposes no communication semantics to the network stack beyond raw byte streams or datagrams. Stretching the filesystem analogy, the Sockets API resembles what might have happened if the filesystem API included primitives for accessing blocks and inodes rather than files: there would be little room for architectural evolution. Our goal in this paper is to define a communication API that is conducive to innovation.

To show the real-world need for a richer network API, we note that while researchers been exploring new architectures, practitioners have not been idle. Today's practical solution to the limitations of Sockets is HTTP. HTTP has well-defined request semantics (e.g. GET vs. POST) that let middleboxes understand how to cache responses, speeding content delivery. Redirection and DNS load balancing provide some flexibility in naming. HTTP provides limited disconnection tolerance by letting partial-content requests resume a transfer in the middle of a file. Finally, cookies let applications establish sessions, which can be used to track users across disconnections and to load balance stateful services. These are all examples of HTTP *exposing application intent* to the network stack. As a result of these features and of the widespread availability of HTTP middleboxes, many applications that used to have separate protocols now run over HTTP, including file transfer, RPC (SOAP), instant messaging (XMPP), and streaming video (YouTube).[1] Nonetheless, HTTP is limited in flexibility because it is an application layer protocol. For example, HTTP cannot facilitate deployment of new naming mechanisms or transport protocols. Furthermore, the caching and session features of HTTP are implicit conventions tacked on over time. Their implementation in middleboxes is architecturally clumsy, and much of the network stack is unaware of these features.

Our Contribution In this paper, we propose a new communication API, called NetAPI, designed to enable flexibility in the network. NetAPI provides an interface against which applications can specify communication intents without binding to particular network mechanisms. Because no general interface can be expected to express all communication semantics, NetAPI uses entities called *schemes* to encapsulate specific classes of communication (e.g. web browsing, voice, RPC). Applications open a NetAPI connection by providing an URI of the form *scheme://resource*. A *scheme implementation* in the OS interprets the resource name, allowing flexibility in naming. Applications then read and write *messages* (ADUs [9]) on the connection. The semantics of these operations are given by the scheme *definition*,[2] but how these operations map onto wire protocols is a function of the scheme *implementation*. NetAPI exposes only the high-level communication semantics to the application, not low-level details which may change over time.

The primary benefit of NetAPI is that it *hides network technology details* from the application, encapsulating them in the scheme implementation. Figure 1 illustrates this shift. Today, applications must manage a variety of low-level network mechanisms, including selecting a name resolution mechanism (e.g. INS [21]) and a transport protocol (e.g UDP), encoding messages over this protocol, selecting network interfaces to use, and ensuring security (through e.g. SSL). Applications must therefore be ported manually when new network technologies emerge. NetAPI shifts network responsibility for managing low-level network mechanisms to the scheme implementation, leaving the application responsible only for breaking content into messages, specifying destination names, and specifying a high-level security policy. This separation of concerns simplifies application development and porting, while letting scheme implementations use new network technologies without application modifications.

A second benefit is that because NetAPI captures more *application intent* than Sockets, it can enable more intelligence in the network stack. For example, NetAPI is a logical place to build disconnection tolerance that is sharable across applications. As another example, a mobile 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 type of central policy would be impossible if the network stack did not understand application semantics.

[1]Some of this is also due to firewalls blocking non-HTTP ports.
[2]We expect schemes to be standardized by bodies like the IETF.

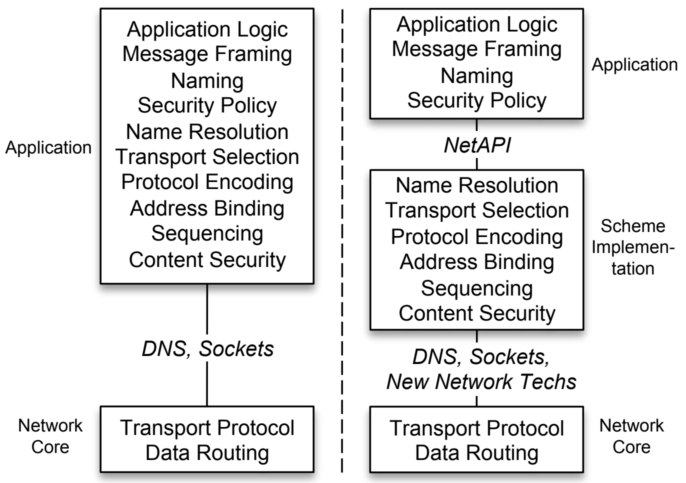


Figure 1: Block diagram showing how the division of responsibilities shifts from the current model (left) to a new functional model (right) with adoption of NetAPI.

We started this project with a clean-slate approach, and in our first attempt we proposed a pub-sub API that was quite different from Sockets [13]. However, after more than a year of additional research, including extensive implementation, we have ended up with a design whose syntax is far closer to the Sockets API. NetAPI essentially replaces addresses with names, (optional) address binding with (optional) transport binding, and extends ADUs with properties. These changes, while seemingly minor (and oft-discussed in the literature), make the interface far more declarative and thus allow network technology to evolve without changes to applications.

Implementation To evaluate NetAPI, we built a NetAPI prototype called PANTS (Protocol Aware Network Technology Selector) for the iPhone, aimed at mobile networking challenges like disconnection tolerance and multiple interfaces. PANTS runs on the client only and interacts with legacy servers. We implemented two applications using PANTS, a file downloader and a news reader, and took advantage of NetAPI to add disconnection tolerance, power-saving policies and content shaping to these applications without modifying them. We also implemented the global "best performance" versus "best battery life" setting explained above.

Outline This paper is organized as follows. We start with an overview of networking challenges that have arisen as the Internet has evolved in Section 2. We describe NetAPI in Section 3. In Section 4, we show how NetAPI supports popular applications. We describe our NetAPI prototype, PANTS, in Section 5, and evaluate it in Section 6. We discuss the design rationale behind NetAPI in Section 7. We survey related work in Section 8. Finally, we conclude in Section 9.

2 Modern Networking Challenges

We make a case for NetAPI by surveying some major challenges in today's Internet. We argue how, despite the recognition of these challenges and the solutions developed, the inflexibility of the Sockets API limits the pace of innovation.

Naming and Addressing: Research has advocated the decoupling of naming of a resource from the address at which it can be accessed. There are numerous advantages to doing this that have been noted in literature – improved performance by transparently choosing the address of the nearest source (e.g., *i3* [11]), expressive naming of resources (e.g., Intentional Naming System [21]), and easy deployment of newer address formats (e.g., IPv6). We believe that providing a common substrate that accommodates various naming schemes and decouples addressing would greatly facilitate development and deployment of such technologies.

Content Delivery: Prior research (e.g., DOT [15], DONA [19]) has proposed the decoupling of the name of a resource from the mechanism by which it is transferred. This is motivated by the fact that a resource can often be obtained from multiple sources and that can improve performance. The increasing popularity of mobile devices also presents an opportunity to query nearby devices for the resource [12].

Multiple Network Interfaces: Mobile devices have multiple connectivity options, e.g., Wi-Fi, cellular, and Bluetooth, with Wi-Max networks on the horizon. These interfaces have different characteristics in terms of power, throughput and range [25], that vary spatially and temporally [5].

Expecting the application to specify the local interface before starting communication clearly leads to sub-optimal choices. On the other hand, innovations have been happening in how devices pick the best available interface [5]. With low-throughput applications, they may prefer to remain connected to cellular during times of good connectivity. High-bandwidth applications may wish to avoid the cellular connections altogether, conserving battery life, or to change protocols on lower-bandwidth links. Proposals have also suggested using interfaces in tandem to increase throughput.

For these, and any future solutions that combine multiple network interfaces, the current model of tying communication to a single address is problematic. This has led to solutions that utilize multiple interfaces being implemented in an architecturally unclean manner and facing deployment challenges as they invariably require network proxies to mask changes in the address of the mobile device.

Mobility: Mobility in devices leads to disruptions in connectivity that require application-specific handling. Disruption tolerant applications like email synchronizers could wait until they get optimal (e.g., energy efficient) connectivity options before resuming their activities, while others like streaming might prefer a seamless transfer of connectivity to another interface. Temporary loss of connectivity is troublesome as it requires maintenance of state during that period and must deal with the possibility of the mobile device resuming communication using a different IP address. This is an active research area, with projects such as DTN [22], Haggle [12], and KioskNet [6].

The reader may note that we have cited existing solutions to all of these problems. We aim to deal with the overarching problem of *adoption*. The limitations of existing network interfaces mean that innovative new technologies have an extremely hard time leaving the research phase. By placing these technologies under a common network API, we can ease adoption and create an environment supporting innovation while still expressing communication intent. It is to this end that we have developed NetAPI.

3 NetAPI Design

NetAPI provides five basic high-level operations:

- **open(***scheme://resource*, *options***)** $\Rightarrow$ *handle*
- **put(***handle*, *message*, *options***)** $\Rightarrow$ *result*
- **get(***handle*, *options***)** $\Rightarrow$ *message*
- **control(***handle*, *options***)** $\Rightarrow$ *result*
- **close(***handle*, *options***)** $\Rightarrow$ *result*

These operations may be exposed to the application by NetAPI language bindings in multiple ways, e.g. through both synchronous and asynchronous versions of the calls. Errors can be reported through exceptions or return codes.

The operations in NetAPI are very similar to those in filesystems and Sockets, making the API easy to understand for developers. Our contributions are twofold: providing separate semantics for separate classes of communications through schemes, and making the API flexible and declarative through the use of key-value options and opaque names.

Users start a connection through **open()**, which returns a connection handle. Instead of asking for an address or DNS name in **open()**, NetAPI takes a Uniform Resource Identifier (URI) [18] of the form *scheme://resource*. The scheme portion of the URI selects one of several *communication schemes*. Each scheme represents a class of network service, such as *web://*, *video://*, or *voice://*. The scheme defines the high-level semantics of NetAPI operations and defines what types of messages and options can be used. The scheme is also responsible for defining how the resource name in the URI is resolved; NetAPI does not enforce any specific naming mechanisms. This approach to naming has been adopted by other systems [23] due to its flexibility and extensibility. We explain schemes in greater detail in Section 3.1.

The **put()** and **get()** operations are how a NetAPI application inserts and retrieves data from the network. We refer to one bundle of data as a *message*. NetAPI messages are application-defined data units (ADUs) [9], such as individual 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.

Get() is also used to accept connections in servers. Calling **get()** on a server scheme returns a handle to the next client connection. The application can then call **get()** and **put()** on this handle to communicate with the client.

The **control()** function is used perform operations that are not part of the data stream, such as seeking in a streaming video scheme, or querying the scheme about the packet loss rate observed. It takes an *options* argument, which is a list of key-value pairs interpreted by the scheme. It returns a *result* object which may also contain key-value pairs, such as the loss rate queried. The other NetAPI operations also support key-value options, and return *result* objects for schemes that wish to provide status information. Example uses include setting transport requirements in **open()** or signaling end-of-file in **get()**. The **close()** call ends a connection.

The rest of this section describes and motivates the main design elements of NetAPI: schemes and messages. We end with a discussion about compatibility and evolution.

3.1 Schemes

A general communication API must support applications with vastly different requirements from the network: media applications desire low jitter and tolerate some losses, file download applications may accept a file from multiple providers, some applications tolerate disconnections, and so forth. The solution to this problem in Sockets is to provide a low-level API and ask applications to manage communication details themselves, but this hinders evolution. The solution in NetAPI is to separate the definition of semantics for different types of communication into schemes. Schemes are a key feature of NetAPI, so we explain and motivate them in detail.

What is a Scheme? A scheme is a protocol between the application and the network stack that captures the high-level structure of a particular form of communication. Schemes capture communication structure that is unlikely to change over time, rather than details of how the communication is implemented on the wire. For example, a *web://* scheme for web clients may specify the following protocol: Applications call **open()** on an URI of the form *web://location/resource*, optionally call **put()** to post HTTP POST parameters (specifying the name of each parameter as a `name` property on the message passed to **put()**), and repeatedly call **get()** to receive the file until the message returned has the `end_of_file` property set. Similarly, a *video://* scheme may specify that messages returned by **get()** are keyframes in a certain encoding, may list a set of standard properties on these messages that indicate position in the file, and may define **control()** operations for seeking and changing bit rate.

The primary benefit of schemes is that scheme implementations can change underneath an application as long as they adhere to the scheme. We refer to this as "vertical" evolution. For example, the *web://* scheme can choose to use a different naming mechanism to resolve the *location* portion of its URI. Similarly, the *video://* scheme could choose to use a more efficient transport protocol or encoding; as long as it provides keyframes in the encoding defined in the scheme standard, applications will continue to work. Furthermore, the choice of encoding can depend on the network interfaces available, or on a user setting such as optimizing for battery life, without the application being aware of these policies. Scheme definitions themselves can evolve by adding new options; for example, on a mobile device, the *web://* scheme may support an option to **open()** called `max_delay` indicating that a request is delay-tolerant. Finally, if a scheme must change in a non-backwards-compatible way to support new technology, a new scheme with a different name can be created. We call this "horizontal" evolution.

Scheme Responsibilities. Scheme implementations are responsible for resolving names, binding to addresses, selecting transport protocols, encoding messages, and ensuring communication security. 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 [14], new transport protocols [30], BitTorrent [8], DTN [22], or a clean-slate architecture like DONA [19]. 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 to the **open()** call.

Nonetheless, the ability of schemes to select communication mechanisms does not entail a loss of control in applications. For example, the system can provide *tcp://* and *udp://* schemes for applications that desire the same amount of control over networking that Sockets provide. These schemes would 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 supported later.

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-server://* scheme may support an option called `secure` in the **open()** call, which forces the server to use HTTPS, 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 high-level 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 encrypt and authenticate data manually.

Finally, schemes define the semantics of messages, allowing scheme implementations to perform bundling and reordering of messages, caching, and content modification (e.g. changing encoding). The use of semantic information can also go beyond simple caching and ordering, 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. Finally, a media scheme on a mobile phone might adjust content quality depending on the network interfaces available as in BARWAN [16]. These actions may be controlled by global setting in the OS, like a "best battery life" versus "best performance" setting.

Standardizing Schemes. 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 (e.g. hypertext, audio, and video). However, if multiple types of communication are supported over a single protocol (e.g. interactive web browsing and non-interactive file downloads over HTTP), it may be cleaner to have a single scheme for this protocol and hint application requirements to it through options. The tradeoffs involved in defining schemes are inherent, and as such we leave them to domain experts in standards bodies. NetAPI aims only to provide flexibility in defining schemes. Our PANTS prototype provides two examples of functional schemes in Section 6.

Because of the great flexibility available to scheme implementations, we expect scheme specifications to include a list of permissible implementation choices, similar to todays Internet RFCs.

3.2 Messages

Messages in NetAPI are application-defined data units (ADUs) [9] containing data and a list of key-value *properties*. 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.

A second advantage of a message-oriented API is that message reconstruction is performed by the scheme, eliminating a common source of bugs and security problems.

3.3 Compatibility and Evolution

As network technologies and schemes evolve, new and old scheme implementations will have to be compatible. In addition, schemes must be compatible with existing non-NetAPI applications. Currently, each form of communication is tightly coupled with an existing wire protocol (e.g. the web and HTTP). To ensure compatibility while supporting evolution, we propose that all schemes initially be defined in terms of wire protocols such as HTTP and RTP that are standardized independent of NetAPI. Initially, scheme implementations would interoperate with each other and with legacy applications over a single legacy protocol. Over time, as schemes gain more options for transport and naming, a protocol negotiation phase can be added, with a default protocol chosen if the negotiation cannot be performed.

Evolving naming options will likely be the most difficult task. The name formats a scheme expects can change, and applications would need to be modified to take advantage of any new naming mechanisms the scheme decides to present. However, if the application is designed to be name-format-agnostic (e.g. users specify strings for names), then it can benefit from new naming mechanisms with no code changes.

4 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.

4.1 Web Content Retrieval

The *web://* scheme is used for retrieving web content. It accepts a URI (`web://<url>`) and optional URL-encoded key-value 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. Timeouts are signalled via *options* fields, with **get()** returning immediately if the HTTP connection fails.

Web servers also need access to cookies and HTTP headers, like the user agent, when they generate content for a HTTP request. Clients may pass these parameters to the **open()** call as options. In the same manner, the client may ask for security features like server identity authentication and content protection through options to **open()**. For access controlled resources, the client may supply 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 [31] without application modification.

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:

```
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 [15] or BitTorrent [8]. 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.

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 filter, without having to configure each application to use the filter.

4.2 Web Server

Naming is an extremely difficult problem for web services. Though all major services use DNS as the primary naming mechanism, the limitations of that technology are well known. Allowing a web server to transparently use different naming mechanisms and transports would allow for a greater rearchitecting of the Internet without a major overhaul of the services involved.

Our solution is to build a NetAPI scheme that abstracts naming and transport from the web server application. To do this, the scheme maps from the name provided to the **open()** call (e.g., `open(web-server://foo)` to names appropriate for the new technologies (e.g., `dtn://foo`). The application may also specify specific naming technologies and names in the options, signaling to the scheme that it should use these names for those technologies. This allows for the application to make use of new technologies with default name mappings, yet still customize existing ones. The following snippet shows how this may map into code:

```
h = open("web-server://foo",
    {"dtn" = "dtn://not-foo"})
```

When the scheme receives a message for the server, the appropriate information is translated and handed to the application. The application may be aware of what technology is being used, for instance sending less data over delay tolerant links. Following this, the application sends data to the client, which is bundled and forwarded over the appropriate transport mechanism.

When publishing static content, the application no longer needs to be informed of requests. If we wanted to optimize this publishing of static content, we may opt for horizontal evolution. Although publishing could be an extension to the *web-server* scheme, it is distinct enough to warrant a new *web-publish* scheme. Under this new scheme, the application calls **put()** to insert static content into the network. This data can then be aggressively cached and/or replicated with a content distribution network (CDN) [1], BitTorrent [8], or DONA [19]. The scheme may automatically load balance the content, replicating more aggressively or to different locales when load increases. Any client requesting it will be directed by the scheme to a close, lightly loaded machine.

4.3 Multimedia Streaming

Multimedia content is naturally accommodated in NetAPI, enabling efficient distribution protocols in a wide variety of environments. For example, the data associated with a URI in a *media://* scheme might link to 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. For example, 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.

On a mobile device, NetAPI can take into account the quality of the available connection (cellular vs Wi-Fi) and any user policy settings (e.g. best battery life vs. best performance) to choose the right level of quality for the stream.

4.4 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 [20], or bundle multiple items together in a single transfer over an intermittent network.

5 PANTS: NetAPI for Mobile Phones

To evaluate the feasibility, usability and flexibility of NetAPI, we implemented a prototype of it for mobile phones. Mobile devices present significant networking challenges, as they must regularly switch between network interfaces and access points. This is often done at the expense of battery life and application usability, much to the chagrin of users. For developers, mobile networking is complex, leading to bugs, poor performance, or applications that simply do not attempt to tolerate disconnections or conserve battery life. Mobile phones thus provide an ideal usage scenario for NetAPI.

We built a mobile phone implementation of NetAPI called Protocol Aware Network Technology Selector (PANTS) for the jail-broken iPhone platform. We implemented two schemes on PANTS – *web* and *voice* – and used them in three sample applications. Both schemes interoperate with legacy servers. We then used PANTS to add disconnection tolerance, content-shaping, and power-saving features in our scheme implementations below NetAPI, without modifying applications.

This section describes the goals and architecture of PANTS. Then, in Section 6, we describe our sample schemes, applications, and experience adding functionality below NetAPI.

5.1 PANTS Goals

In designing PANTS, we wanted to build a research platform that would not only implement NetAPI, but would also be able to host various cross-application policies for managing multiple network interfaces proposed in the literature [6, 12, 16, 32]. We set the following goals for PANTS:

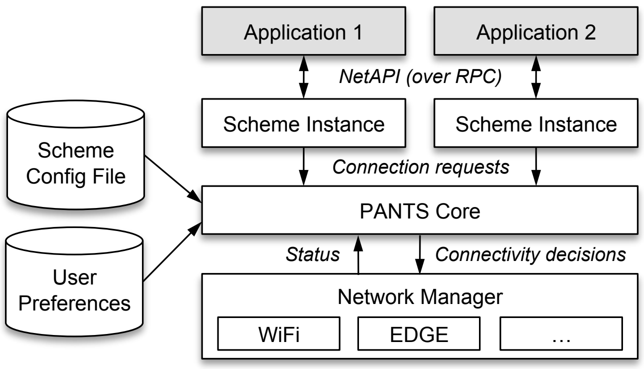


Figure 2: PANTS architecture.

1. Provide a NetAPI interface to applications.
2. Provide a location for centralized network interface selection based on application needs and user preferences.
3. Make schemes and policies pluggable.
4. Support "real" applications talking to unmodified servers.

5.2 PANTS Architecture

PANTS is implemented as a Python daemon using Twisted Python [36]. The primary supported platform is the jail-broken iPhone, but PANTS also runs on Nokia's N810 Linux platform and on Linux and Mac OS X laptops. Figure 2 illustrates the architecture of PANTS. The system has the following main components:

Application Interface. Applications communicate with the PANTS daemon by invoking the NetAPI operations (**open()**, **get()**, etc) over local RPC. We provide both XML RPC, which has implementations in most programming languages, and a more efficient binary protocol for Python clients.

Schemes. Each scheme implementation in PANTS is a Python class. An XML configuration file maps each scheme name to a class name. An instance of the appropriate scheme class is created for each client connection after the client calls **open()**. This instance implements the other NetAPI operations, such as **get()** and **put()**, by invoking lower-level network mechanisms.

Connection Requests. Scheme instances do not open TCP and UDP connections to servers directly. Instead, they request a connection from the core module, allowing it to give them the "best" available network interface. We also let schemes provide hints about their bandwidth, latency, and disconnection tolerance needs to aid in interface selection. These hints could be used for various network selection policies proposed in the literature [6, 7, 12, 16, 32]. However, we have not implemented any of these policies in PANTS yet, because we have focused on implementing other interesting functionality under NetAPI. We have only implemented a simple policy for disruption-intolerant applications described in the PANTS Core section below. We also let schemes manually request particular network interfaces.

Network Manager. PANTS has a network interface driver object for each physical network interface on the phone. This driver wraps platform-specific networking libraries to expose network interface status to PANTS (e.g. available WiFi networks and signal strength) and to let PANTS turn the interface on and off. The status information to can be used to select interfaces in schemes or in the PANTS Core.

User Preferences. PANTS provides a simple binary setting for user preferences – best performance or best battery life. This setting is available to the PANTS core and to schemes.

PANTS Core. The PANTS Core module is responsible for creating scheme instances in response to connections, selecting network interfaces in response to schemes' connection requests, and turning interfaces on and off.

We currently support only a simple network interface selection policy, meant to illustrate that PANTS can implement cross-application policies. Schemes may say, for each connection request, whether the connection is *disruption-tolerant*, i.e. could be resumed if we switch network interface. Applications such as file download and email are disruption-tolerant, but low-latency applications like VOIP may not be. This policy is in place because the iPhone OS only allows one of EDGE and WiFi to be active at any time. For disruption-tolerant applications, PANTS returns the best-performing network interface available (preferring WiFi over cellular), unless a disruption-intolerant application is running, in which case the current interface is used. For disruption-intolerant applications, PANTS chooses EDGE when possible, unless another disruption-intolerant application is running over WiFi.

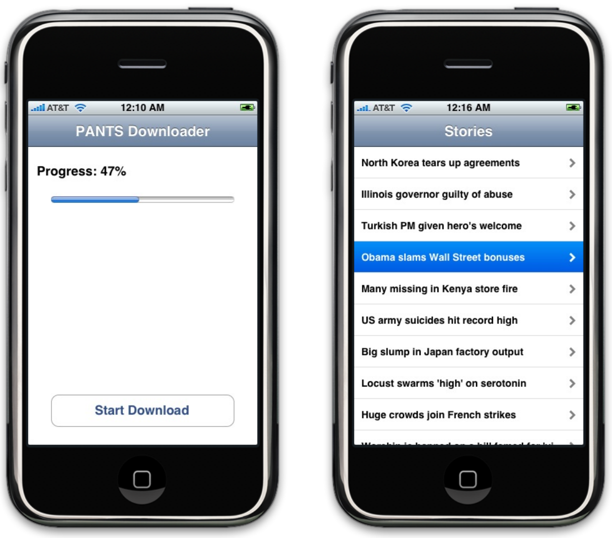
6 Evaluation

To evaluate our NetAPI prototype, we implemented two sample schemes, *web* and *voice*, and three applications over these schemes. We then added five policies in the scheme implementations under NetAPI, to demonstrate that these policies can be implemented cleanly without modifying applications. Section 6.1 describes the *web* scheme, to which we added support for disconnection tolerance, content shaping, and a power saving policy. Section 6.2 describes the *voice* scheme, to which we added encoding selection and security mechanism selection. Finally, we benchmark PANTS to show that the overhead from NetAPI is tolerable in Section 6.3.

6.1 Web Scheme

We implemented a *web* scheme for accessing content over HTTP. Applications use this scheme by opening a URI of the form *web://host:port/resource*, then successively calling **get()** to obtain chunks of content. The message properties for each chunk include the chunk's position in the file and some HTTP parameters such as content length.

We built two sample applications over the *web* scheme: a File Downloader and a News Reader. We briefly describe



North Korea tears up agreements
Illinois governor guilty of abuse
Turkish PM given hero's welcome
Obama slams Wall Street bonuses
Many missing in Kenya store fire
US army suicides hit record high
Big slump in Japan factory output
Locust swarms 'high' on serotonin
Huge crowds join French strikes

(a) Downloader (b) News Reader

Figure 3: PANTS applications using the *web* scheme.

these applications and demonstrate how extending the *web* scheme provides benefits to the applications.

File Downloader. Our first application was a File Downloader (Figure 3(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 3(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 invoke NetAPI. The following listing shows the relevant code from the Downloader (in Python); the News Reader is similar:

```
self.fileData = ""
connection = PANTS.open(self.url)
while True:
  message = connection.get()
  self.fileData += message["data"]
  self.progress = (message["position"] /
                   message["fileLength"])
  if message["isEndOfFile"]:
    break
```

The code simply opens a handle and calls `get` on it repeatedly to receive chunks of the file. The `message` returned by `get` contains a field called `"isEndOfFile"` on the last chunks. It also contains fields for the current position in the file and the file length, which are used to display progress.

6.1.1 Features Implemented Under Web Scheme

Disconnection Tolerance. The original *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 also plan to support a timeout on `get` and an optional parameter to prevent blocking.
2. On reconnection to the server after a disconnection, the scheme implementation 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 download managers in 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* scheme semantics already defined all `get` calls as blocking.

Power Management. To further illustrate the flexibility of PANTS, we extended the *web* scheme with a feature that is missing in mobile download applications that we are aware of (e.g. iTunes on the iPhone): the ability to constrain downloads to only occur over Wi-Fi, to save power[3]. This required a very simple code change to the scheme implementation – when the scheme requests a connection from PANTS, if the "best battery life" user setting is on, it explicitly requests WiFi. Because of the download-resumption functionality in the previous section, the semantics seen by the application are unchanged – it receives a new chunk of data each time it calls **get**().

Content Shaping. We also demonstrate PANTS's ability to transform the content received by the application. When the "best battery life" policy is selected by the user, we alter the page content received by the News Reader 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.

6.2 Voice Scheme

To test the applicability of PANTS and NetAPI to multimedia tasks, we implemented a voice-over-IP (VoIP) scheme. This scheme utilizes Twisted Python's SIP protocol libraries to register and communicate with an IP private branch exchange (PBX) system, using authentication credentials (username and password) provided to the **open**() function. When the PBX receives a call destined for our application, PANTS and the PBX negotiate an RTP connection using the Session Description Protocol (SDP). Because the VoIP registration and session negotiation protocol is more complex than the HTTP logic in the *web* scheme, the *voice* scheme provided a good opportunity to verify that NetAPI can encapsulate complex session setup protocols inside a scheme.

We built a very simple application on the *voice* scheme that registers with a PBX server, providing a username and password, but does not attempt to parse RTP data or respond to calls. This was sufficient to test that the full connection bootstrap process can be done inside NetAPI. We plan to build an application that actually plays media and allows the user to speak in the future, but it was difficult to find suitable required media libraries for Python on the iPhone.

Encoding Selection. When receiving 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 policy was simple to implement in PANTS, through 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.

In the current *voice* scheme, the application must still be aware of the encoding used in RTP, so it cannot be fully agnostic to VOIP protocol evolutions. However, it would also be possible to have PANTS decode the RTP data and give the application content in a single lossless encoding.

Security Policies. During the SIP registration, the PBX server lists available authentication mechanisms. Again, PANTS can make intelligent decisions without the application's involvement. For secure VoIP communications, PANTS can disallow communications over unknown or unsecured WiFi networks. It can also request encrypted communication. If PANTS were implemented in the OS, security policies could also be set at the OS level by system administrators in an enterprise, and would thus affect any NetAPI-based VOIP application that users installed, including applications that administrators may not be aware of.

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. Nonetheless, we evaluated the performance of PANTS to show that overheads are tolerable. We compared the time it takes to download a file through PANTS to downloading it through `wget`. Table 1 shows the results for four scenarios – a laptop downloading a small file over Wi-Fi, a laptop downloading a large file, and an iPhone connecting over Wi-Fi or EDGE. We see no statistically significant difference between PANTS and `wget` in all scenarios except for Wi-Fi on the iPhone. This slowdown is due to PANTS becoming CPU-bound. We expect a native implementation of NetAPI to be as fast as `wget`.

[3]WiFi is more costly than GPRS per second, but less costly per byte. This is due to the transmission costs and bandwidths.

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 1: PANTS download performance compared to `wget`. Standard deviations shown in parentheses.

7 Discussion

NetAPI Design Rationale NetAPI sprang out of an earlier clean-slate project to replace the Sockets API with a more flexible pub-sub interface [13]. After a year of discussions and implementation, what surprises us most is how similar our final result is to Sockets. Rather than attempting to change basic communication abstractions in NetAPI, we decided to make the existing read/write interface more *declarative*, through features such as URIs for names (instead of addresses) and key-value options. The resulting interface is simple and familiar to programmers. In addition, we allowed separate interfaces for different classes of communication to be created through schemes, which define the high-level semantics of NetAPI operations but support evolution both "vertically" within scheme implementations and "horizontally" through the introduction of new schemes.

The concept of schemes led to some interesting debates within our group. First, by making standards bodies responsible for defining scheme semantics, are we just shifting the problem of designing an evolution-friendly API to them? We believe that this is not the case. The problem of defining future-friendly communication interfaces is inherently complex, so there can be no "free lunch" – someone will have to make tough decisions, and standards bodies like the IETF are well suited to understand such decisions given their experience defining protocols. However, schemes separate the problem into tractable pieces and provide an avenue for gradual evolution. Useful schemes for simple forms of communication such as email and HTTP could be defined today based on protocol specifications, and would allow developers to take advantage of features such as disconnection tolerance that can be built below NetAPI. These schemes could then be extended over time with new options, or succeeded by new versions. Furthermore, schemes for different communication classes can be developed independently.

Second, how will different scheme implementations and versions interact? Currently, each class of communication is tightly coupled with an existing wire protocol (e.g. HTTP for the web, or RTP for media). Our proposal is let wire protocols be standardized independent of NetAPI and define NetAPI schemes in terms of these wire protocols. Initially, scheme implementations would interoperate with each other and with legacy applications through a single protocol. Over time, as schemes gain more options for transport and naming, a protocol negotiation phase can be added, with a default protocol chosen if the negotiation fails.

Property	Sockets	HTTP	Filesystem	NetAPI
App. intent captured	low	high	medium	high
Technology detail hidden	low	medium	high	high
Naming flexibility	low	medium	high	high
Generality across apps.	high	medium	high	high
Implementation freedom	low	medium	high	high

Table 2: Comparing the interfaces provided by Sockets, HTTP, filesystems, and NetAPI.

Aside from schemes, the main design principle behind NetAPI is *separation of concerns* between the application and the scheme implementation. Rather than being responsible for both determining high-level communication goals (e.g. download a file) and invoking the network mechanisms that accomplish them, as under the current model, applications under NetAPI only specify high-level goals, and scheme implementations decide how to meet them. This lets scheme implementations use new network mechanisms without application changes, and simplifies application development. Providing a high-level interface for network operations is of course not unique to NetAPI; many communication libraries, such as .NET's *HttpWebRequest* [27], perform the same function. Our contribution is to define an interface that we believe is general enough to work across applications, across devices, and across very different network technologies, making it a candidate for a new operating system API.

This leads us to a final question: should NetAPI be a system interface or an application library? Although it is possible to implement NetAPI in a library, implementing it in the OS has advantages. First, the NetAPI implementation can coordinate traffic across multiple applications, e.g. to prioritize media traffic. Second, the system can make centralized decisions, such as turning network interfaces on and off.

Comparing NetAPI to Other System APIs To summarize the design principles that went into NetAPI, we compare it with several other APIs along several axes in Table 2:

- **Sockets** provides a low-level interface to the network, which makes it generally applicable across applications but reduces the amount of freedom that implementations have. Little application intent is exposed to the network stack, and addresses are used instead of names.
- **HTTP** (which is a protocol, but has essentially become an API through various libraries) captures more application intent than Sockets. This allows implementations to be more flexible and to hide technologies such as caching from applications. However, HTTP is less broadly applicable than Sockets – for example, it is not suitable for low-jitter applications that prefer UDP over TCP. HTTP also provides no flexibility in naming hosts, only in naming resources on hosts.
- The **filesystem API** for storage contains several fea-

tures that make it both generally applicable and flexible: paths for naming, mount points for seamlessly integrating new filesystems, file properties such as writability, and a means of capturing application intents through parameters such as the mode in which a file is opened.

- **NetAPI** incorporates the properties that make the filesystem API flexible (textual names, properties and options). It is applicable to more applications than HTTP because it does not mandate TCP or DNS, and it provides freedom in naming both hosts and resources.

NetAPI Adoption. Like previous Internet architecture proposals, NetAPI requires application modifications in order to be adopted. However, NetAPI aims to be the "last" major interface change for some time, by providing a high degree of flexibility, being similar enough to Sockets to be familiar to developers, and supporting existing protocols (as shown by PANTS). Whether NetAPI is adopted also depends on how useful it is to developers. We believe that NetAPI has the potential to solve real problems today in the space of mobile devices, by encapsulating complex functionality such as disconnection tolerance and management of multiple interfaces into scheme implementations that can be used by many applications. Mobile developers are also already used to working with higher-level APIs in a sandbox environment (e.g. Palm's WebOS [2] requires applications to be written in JavaScript and HTML5). We plan to pursue a fuller implementation of NetAPI for a mobile platform in future work.

8 Related Work

Rethinking the communication API. NetAPI is the evolution of a proposal presented in [13], where the authors argued that a publish/subscribe API would serve the needs of today's Internet applications better than Sockets. NetAPI contains multiple improvements over this early proposal.

First, rather than attempting to replace the low-level Sockets API, we focus on enabling separation of concerns between a high-level layer providing a communication interface to the application (NetAPI) and the network stack below it. NetAPI can thus interoperate with Sockets applications. Second, the put/get interface in NetAPI is a more natural fit for many applications than a pure publish-subscribe API. For example, for a request-response interaction with an HTTP server, [13] suggests having the client create a temporary publication to use to receive its reply and post a reference to this publication to the server. In NetAPI, the request-response exchange is simply a put followed by a get. We do note that schemes can use pub-sub mechanisms if desired. Third, we have implemented a prototype of NetAPI that can run "real" applications interacting with existing servers, which we have used to explore the design space and to verify that useful functionality can be implemented below the API.

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

DONA [19] 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 [15] 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.

DTN [22], Haggle [12], and KioskNet [6] attempt create networks over unreliable links. These technologies are particularly applicable to cellular networks and developing regions, where network connectivity is often spotty. Abstracting these technologies *below* the network API would allow for much greater adoption of technology in disadvantaged areas as well as simplifying the development of delay-tolerant applications.

A handful of other proposals have demonstrated the benefits of expressing application communication semantics to the network stack, including Scalable Data Naming [33] and Structured Streams [24]. Declarative networking [10] 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 [35] and IBM WebSphere MQ [28]). 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 language-specific messaging interfaces like the Java Message Service [34].

One commercial platform of particular interest is the Palm WebOS [2] 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 [37]. 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 communication interface.

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 [4] detects the available network interfaces and its changing characteristics and presents them to applications. Odyssey [7], Mobiware Toolkit [32] and the framework in [29] focus on the complementary aspect of defining mechanisms for applications adaptation. Odyssey [7] 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 [32] defines a utility function relating the application's quality and bandwidth changes. Likewise, the framework in [29] provides applications with a feedback loop that helps map from network-centric quality to application-centric quality.

NetAPI implementations can benefit from many of the above-mentioned techniques. However, our goal is to design a generic interface that allows these and other network technologies to be deployed without application modifications.

9 Conclusion

NetAPI is a communication interface designed to enable innovation in the network. Unlike the Sockets API, NetAPI *hides implementation mechanisms* from the application and *captures application intent* to let the network stack serve application requests intelligently. This design allows new network technologies to be deployed below NetAPI without application modifications. Furthermore, application development is simplified because complex management of network mechanisms can be encapsulated into shared scheme implementations. We have demonstrated the utility of NetAPI through a prototype for mobile phones called PANTS, showing that disconnection tolerance, content shaping and power saving policies can be added transparently under NetAPI.

10 References

[1] CoralCDN. `www.coralcdn.org`.

[2] Palm WebOS. `http://developer.palm.com/`.

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

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

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

[6] 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.

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

[8] BitTorrent. `http://www.bittorrent.com`.

[9] 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.

[10] 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.

[11] I. Stoica et al. Internet indirection infrastructure. In *Proc. of ACM SIGCOMM '02*, August 2002.

[12] J. Su et al. Haggle: Clean-slate Networking for Mobile Devices. Technical Report UCAM-CL-TR-680, University of Cambridge, Computer Laboratory, January 2007.

[13] M. Demmer et al. Towards a Modern Communications API. In *Proc. of Hotnets '07*, November 2007.

[14] M. Walfish et al. Untangling the Web from DNS. In *Proc. of NSDI '04*, San Francisco, CA, USA, March 2004.

[15] N. Tolia et al. An Architecture for Internet Data Transfer. In *Proc. of NSDI '06*, San Jose, CA, USA, May 2006.

[16] R. H. Katz et al. The bay area research wireless access network (barwan). In *In Proceedings Spring COMPCON Conference*, pages 15–20, 1996.

[17] S. Leffler et al. An Advanced 4.4BSD Interprocess Communication Tutorial.

[18] T. Berners-Lee et al. RFC 3986: Uniform Resource Identifier (URI): Generic syntax. RFC 3986, IETF, January 2005.

[19] T. Koponen et al. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of ACM SIGCOMM '07*, Kyoto, Japan, August 2007.

[20] 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.

[21] W. Adjie-Winoto et al. The Design and Implementation of an Intentional Naming System. In *Proc. of SOSP '99*, Charleston, SC, USA, December 1999.

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

[23] K. Fall and S. Farrell. DTN: An architectural retrospective. *IEEE Journal on Selected Areas in Communications*, 26(6), June 2008.

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

[25] G. Ananthanarayanan and I. Stoica. Blue-Fi: Enhancing Wi-Fi Performance using Bluetooth Signals. In *Proceedings of ACM/USENIX MobiSys*, June 2009.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03*, 2003.

[27] HttpWebRequest. `http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.aspx`.

[28] IBM. WebSphere MQ, January 2008. `http://www.ibm.com/software/integration/wmq/`.

[29] J. Bolliger and T. Gross. A Framework-based Approach to the Development of Network-Aware Applications. In *IEEE Transactions on Software Engineering*, 1998.

[30] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, 2002.

[31] Liberty Alliance. `http://www.projectliberty.org`.

[32] O. Angin et al. The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking. In *IEEE Personal Communications*, 1998.

[33] 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.

[34] Sun Microsystems. Java Message Service (JMS), January 2008. `http://java.sun.com/products/jms/`.

[35] Tibco. Tibco Enterprise Messaging Service, January 2008. `http://www.tibco.com/software/messaging/`.

[36] Twisted Matrix Labs. Twisted event-driven network engine. `http://twistedmatrix.com/trac/`.

[37] WHATWG. Html 5 draft recommendation - structured client-side storage. `http://www.whatwg.org/specs/web-apps/current-work/multipage/structured-client-side-storage.html`.