Application Support for Opportunistic Communication on Multiple Wireless Networks

A. Seth[1], S. Bhattacharyya[2], S. Keshav[1]

[1]School of Computer Science
University of Waterloo, Ontario, Canada

[2]Sprint Advanced Technology Labs
Burlingame, California, USA

Abstract— **Many wireless network technologies are in use today. However, each technology has to make a difficult but unavoidable tradeoff between the area covered by a single access point and the capacity available to an individual user. Fortunately, current and future mobile devices are likely to come equipped with multiple wireless interfaces that can be used either singly or in parallel. By opportunistically using these multiple wireless interfaces a mobile device can mitigate the coverage-capacity tradeoff of the underlying networks. However, actually using multiple interfaces– requiring sessions to be maintained across interface switches, disconnections, and device shutdowns–is complex. Consequently, to be useful in practice, we believe that application writers need to be shielded from these details, while still exercising fine-grained control over network usage policies. In this paper we describe a system that allows applications running on a mobile device to seamlessly exploit multiple heterogeneous wireless networks. We have designed, implemented, and evaluated the performance of an Opportunistic Connection Management Protocol (OCMP) that allows such applications to opportunistically communicate on multiple network interfaces, switch across interfaces, aggregate their bandwidths, remain disconnected or powered off for arbitrarily long periods, package and move their state across devices, and interoperate with legacy applications and servers. The implementation is in J2ME so that it can be run on any Java-based mobile device. Extensive policy control for interface selection is also provided to applications, along with a very simple API for application developers. This paper explains the design, architecture, and implementation of OCMP, and illustrates its benefits through both analysis and field experiments. Our results are encouraging and suggest that application support for multi-network opportunistic communication is both achievable in current systems, and of significant practical value.**

I. Introduction

The past few years have seen an explosive growth in the number of mobile devices such as cellphones, PDAs, and laptop computers. These devices can use a variety of wireless access technologies. These range from wide-area technologies such as GPRS, EDGE, CDMA 1xRTT, EV-DO, and satellite access, to local-area technologies such as 802.11a/b/g and short-range technologies such as Bluetooth, Zigbee, etc. However, any wireless access technology must make a difficult tradeoff between the coverage of an access point and the capacity available to a user in that access point's coverage area. To offer wireless access in a given geographical area, wide-area wireless access technologies require fewer access points but offer inherently lower per-user capacity. Short-range infrastructure access networks can offer large per-user capacity, but the capital cost to offer coverage in large geographical areas can be prohibitive. Although recently 802.11-based mesh networks have attempted to provide near-ubiquitous wireless broadband access, in practice success has been limited because of uncontrollable external interference in the 2.4GHz band and a dramatic reduction in capacity when the multi-hop count is large. Consequently, we think that no single wireless access technology can be expected to provide ubiquitous, high-bandwidth coverage. For example, high-speed 802.11 a/b/g access coverage is typically confined to WLANs inside buildings and public hot-spots. In contrast, lower-speed WWAN technologies such as CDMA 1xRTT and GPRS provide far wider coverage, although even such technologies cannot be expected to be available everywhere and coverage can be decidedly spotty inside enclosed areas. Note that, besides this coverage-capacity tradeoff, managed wireless technologies impose limits on the number of simultaneous users in a given geographic area. This may prevent a user from using a network even when it is available.

Fortunately, future mobile devices will almost surely come equipped with multiple wireless interfaces that can be used either singly or in parallel. *Our key insight is that mobile devices equipped with multiple radio interfaces can opportunistically use one or more wireless networks to increase their overall communication capacity.* However, actually using multiple interfaces–requiring sessions to be maintained across interface switches, disconnections, and device shutdowns–is complex. To be useful in practice, we believe that application writers need to be shielded from these details, while still exercising fine-grained control over network usage policies. In the rest of the paper, we elaborate on this insight and use it to design, analyze and implement an Opportunistic Connection Management Protocol (OCMP).

The rest of the paper is laid out as follows: In Section II we describe our design goals. Sections III and IV present first an overview, then a detailed description of our system. We evaluate our design using analysis in Section V and field measurements in Section VI. Section VII is a survey of related work. We conclude with a description of our contributions, and an outline of future work in Section VIII.

II. Design goals

Our top-level design goal is to allow an application to opportunistically exploit the presence of one or more wireless connections between a mobile and a server. However, in meeting this goal, we ran into several non-trivial problems. For instance, how should a mobile stripe its data across multiple interfaces? How should it interact with legacy servers that cannot identify multiple streams originating from a single mobile as a single striped stream? How can an application direct the communication subsystem to obey constraints on pricing and/or delay bounds? These question led to the following sub-goals:

- **Application-directed intelligent use of multiple networks:** Today's mobile devices can detect the availability of one or more access networks, but a *single* interface is selected by default at the IP layer. Because IP chooses a next hop based solely on the destination address, fine-grained multi-path routing of application data segments on multiple access networks is difficult. Instead, we'd like a mobile device to potentially use *multiple* access networks simultaneously even for a single application, opportunistically switching to the best available access network. When multiple interfaces are in use for a single application, data has to be striped across these interfaces. Decisions about which networks to use and how to stripe data across them has to take into consideration the bandwidth and congestion states of the available access networks, energy-efficiency of the wireless radios, provider pricing policies[1], and application requirements. So, we would like to allow an application developer or user to specify fine-grained (i.e. per-application data unit) policies.
- **Application session persistence across disconnections:** Consider a mobile device that has established a connection to a server using one of several network interfaces. Suppose the mobile decides to power itself off or switch to a different interface; then reconnects to the same server with a different IP address. With existing systems, the server would be unable to recognize that the two connections correspond to a single ongoing data transfer session, and therefore would not be able to migrate persistent application state from the old connection to the new. For network access to be truly opportunistic and seamless, an application should be able to exchange data with a server when changing network interfaces or even when faced with intermittent loss of connectivity. This requires the maintenance of persistent application state at both the server and the client so that data transfer can resume from the point where it stopped once connectivity is restored.
- **Support for legacy servers:** We recognize that existing servers are heterogeneously administered, so it is unlikely that a solution that requires changes at a server will ever be deployed in practice. Consequently, we would like a solution that works well with existing servers.
- **Ease of application design and implementation:** Application designers who are familiar with the socket-bind-connect approach to writing distributed applications cannot deal well with systems where connections may fail arbitrarily, be resumed arbitrarily, and exhibit large variations in bandwidth depending on the currently available network. We would like to insulate application developers from these problems and provide them with a simple and intuitive communication interface.

We address these goals by means of our system architecture and Opportunistic Connection Management Protocol (OCMP). We present an overview of the system architecture next, in Section III, deferring details of OCMP to Section IV.

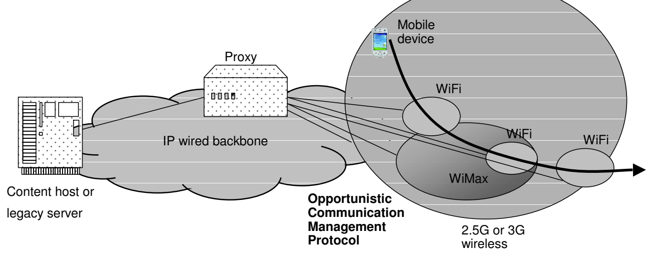
Mobile device
Proxy
WiFi
IP wired backbone
WiFi
WiFi
WiMax
Content host or legacy server
Opportunistic Communication Management Protocol
2.5G or 3G wireless

Fig. 1. Architecture overview

III. Architecture

Figure 1 presents an overview of the system architecture. The main components are the content host, the proxy–which runs an OCMP server–and the mobile host, which runs the OCMP client. We describe each component next.

A. Content host

At the left of Figure 1 is the content host, a server that either provides content such as video or stored voice to a mobile device or receives uploads and content requests from the mobile. This represents popular web sites like google.com, or yahoo.com as well as media servers that provide audio and video content. Content hosts reside in a data center at the core of the Internet. These servers are connected to a wired, high-capacity, and global IP backbone. Existing content servers run legacy applications and do not support disconnection resilience or parallel transport connections over multiple networks for a single application session. We would like to provide a feasible path for supporting opportunistic communication *without* requiring modifications to legacy servers. We achieve this via the deployment of network-based proxies which are described next.

B. Proxy servers

Proxy servers allow interworking between legacy servers and our protocols. A proxy is located in the communication path between a mobile device and a content host. It serves as the termination point for the multiple transport connections opened by the mobile host over multiple network interfaces. The proxy server hides data striping and multiple connections from the content host. It can also provide fine-grained and application-specific connection management, as will be described in Section IV.

The proxy can either be provided by an internet service provider, or by an enterprise on behalf of its employees. Proxies should be placed so that the round trip time from the proxy to the large majority of the mobile devices is as low as possible. For example, cellular providers could naturally keep the proxies adjacent to the PDSNs in CDMA or the GGSNs in GPRS networks, or on the backhaul point to the Internet core. On the other hand, if a third party provides a proxy as a value-added

[1]Provider pricing policies can have a huge impact on the usage of cellular data networks. For instance, Rogers Telecom in Canada charges up to CDN$21/MB for some rate plans! In contrast, Airtel in India charges only CDN$0.135/MB for its most expensive plan, which is 155 times cheaper.

service, it should place the proxy in a well-connected data center. We only require that the proxies have one or more globally-reachable public IP addresses, or dynamic DNS registrations.

The proxy acts as a store-and-forward agent for data downloads to a mobile device. A download starts with a mobile application initiating a data transfer request, for instance, an HTTP GET request. This request is intercepted by the OCMP client on the device and forwarded to the proxy. The OCMP server on the proxy supports an *application plugin* (described in more detail in Section IV-D) that allows it to understand how to process application-specific data transfer requests. If the request is from an application supported by the proxy, the plugin at the proxy processes the request and then uses legacy protocols to contact the content host on behalf of the mobile device. Thus the content host is shielded from details of communication between the mobile and the proxy.

Once data is downloaded from the content host to the proxy, the proxy caches the data and looks for available transport connections to the mobile device. No such connection may be available at that time if the mobile device is temporarily disconnected from all access networks. If so, the proxy holds the downloaded data in persistent storage until the device reconnects. Alternatively, the proxy can use an out-of-band mechanism (e.g., an SMS message if the mobile device is a smart phone) to inform the mobile device about the availability of data. When the mobile device reconnects using one or more transport connections, the proxy segments the application data into *bundles* (which are similar in spirit to the bundles in Delay Tolerant Networking [6]) and routes the bundles over these connections. The routing policy over multiple connections is negotiated in advance between the OCMP peers on the mobile client and the proxy.

Similarly, when data is being uploaded from the mobile device to a content host (e.g., blog or picture uploads), the proxy receives bundles from a single application over multiple transport connections from the mobile, reassembles them into a single stream, opens a connection to the content host and forwards the data using legacy protocols. Thus the OCMP client on the mobile host and the OCMP server on the proxy implement analogous functions for segmentation and reassembly of application data as well as policy-based multi-path routing of application data segments.

The multi-connection state between a mobile and a proxy can be packaged and moved to a different proxy to allow a mobile to always use a ‘nearby’ proxy, greatly improving performance. Similarly, the state on a mobile device can be retained persistently across arbitrary periods of disconnection or power loss. The state can even be transferred to a different end-point like a home or office desktop, and unpackaged to recreate an operating state identical to the state on the mobile prior to disconnections. This can be used to provide semantics similar to that provided by Internet Suspend and Resume [8].

C. OCMP

The proxy and a mobile are connected by multiple heterogeneous wireless networks that differ in coverage, capacity, pricing, and availability. An OCMP server-side protocol running on the proxy coordinates access on these networks with an OCMP client-side protocol running on each mobile. OCMP defines a message format for encapsulating application data segments which are striped across multiple interfaces. This allows encapsulation of application data segments that are created at the mobile or the proxy for transfer over multiple transport connections. For example, the OCMP header contains the sequence number for each segment, which allows reassembly at the other end. OCMP also has control messages, i.e., messages that consist of only an OCMP header and an empty body. For example, control messages are exchanged between the OCMP client and the proxy to coordinate policies regarding data striping across multiple transport connections.

Connections between an OCMP client and OCMP server are always initiated by the client. A new transport layer connection is created each time the device connects on a new network and torn down when the device disconnects. Each network interface is associated with a single transport connection that is shared by all application data units assigned to that interface.

Using OCMP, data can be transferred on multiple connections in parallel, under fine-grained application control. Essentially, OCMP clients and servers choose the connection to be used for each application-level data unit using application-specified policies. Moreover, if a connection abruptly terminates, or even if *all* the connections terminate, the OCMP client and server gracefully recover from the failure, providing applications the illusion of seamless connectivity.

Unlike past work, OCMP does not depend on TCP semantics of the underlying connections, as long as the transport layer provides end-to-end reliability. An underlying connection can be a standard TCP/IP connection or can be a transport protocol optimized for wireless networks, such as erasure-coded UDP. OCMP can therefore exploit systems that compress and transcode data on wireless links to optimize bandwidth use [23].

Besides working with the server-side, the OCMP client-side also has the additional responsibility of detecting network connections and disconnections. It uses application-specific policies to decide whether it should initiate a connection to the server side when a connection opportunity arises. It also has a notification mechanism to inform an application if there is any data that has arrived for it.

D. Mobile

The proxy identifies a mobile device (and all connections originating from it) by a globally unique identifier (GUID). This GUID can be drawn from an existing namespace such as the IMSI numbers for mobile phones or IPv6 addresses. The GUID serves several purposes. First, as in HIP [10], it decouples device addressing (a GUID) from routing (in terms of IP addresses). This solves the problem of IP address changes due to mobility and/or disconnections. Second, it enables the proxy to maintain persistent data transfer state even when a mobile application uses parallel transport connections over different access networks. Finally, it enables the proxy to stripe data intended for a single mobile device across all the transport connections belonging to the device.

The OCMP client on a mobile device provides two application interfaces. The first is meant for legacy applications that are designed in the socket-bind-connect paradigm. For such applications, data download requests are intercepted by the OCMP client and dispatched to a client-side application plugin, which sends a message on a control connection to its peer running on the proxy. The plugin at the proxy, acting on behalf of the client, initiates a connection to the server using legacy protocols (e.g., TCP/IP) and downloads the data. It then transmits the data to the mobile device striped across multiple connections. The OCMP client layer on the mobile device reassembles the data before delivering it to the application. For data uploads, the plugin at the proxy reassembles data received from the device over multiple connections and transmits to the server over legacy protocols.

We have also built a new application interface for disconnection- and delay-tolerant applications. It takes the form of a 'communication directory', which is a standard directory in the file system. An application writer drops a file into this directory, and is guaranteed that the file will appear at a destination directory at some point in the future. This is described in more detail in Section IV-A. We have found that this API is both robust and easily understood by application developers.

We have developed a Java-based prototype implementation for our system and evaluated its performance on laptops with multiple wireless access technologies such as 802.11b/g, CDMA 1xRTT, and GPRS EDGE. We chose to implement our system in Java due to the rapid proliferation of J2ME-compatible mobile devices. Our system is platform independent and can be simply downloaded to a suitable end point device. We encountered a number of implementation problems in going from a paper design to a working prototype. These problems are described in detail in [13] and are elided from this presentation due to considerations of space.

IV. OCMP DETAILS

This section describe the OCMP client and server protocols in greater detail.

A. Client side communication API

OCMP interacts with applications on the client side either by means of a 'communication directory' or by intercepting socket calls made by legacy applications.

A communication directory is simply a directory in the file system that contains application data. Each file in this directory has a sequence number and the files are transferred to the proxy in order of the sequence numbers. To send data, an application creates a new file in the directory with the next unused sequence number. A 'watcher' process periodically looks for modifications to the last modified time of the directory. If the modification time is more recent than the last time the directory was checked, the newly created files are sent to the OCMP stack using the OCMP API.

The watcher also registers itself as the default receiver with OCMP, much like inetd. When called, it accepts data and writes them to a file in the appropriate communication directory. A client can simply read this file to get its incoming data.

Each communication directory has two special files. The *config* file has application-specific configuration parameters. For example, for the blog-upload application, this is the user-

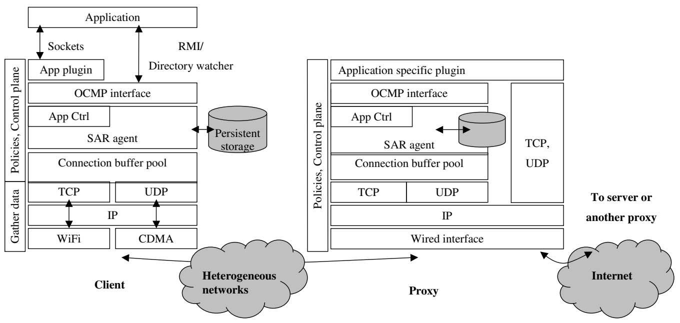

Fig. 2. The OCMP protocol stack at a mobile and a proxy. Only two NICs are shown.

name and password for the user. The config file also contains application-specific policies to control the interface(s) used for transferring data for that application. These policies are passed to OCMP by the watcher. The other special file is the *status* file. This file has one entry for each file in the communications directory and contains the status of that file. The status of a file can be, for example, 'ready to send', 'partially sent', or 'sent'. An application that wants to know the status of a file's transfer can read the status file. This can be used, for example, to update an icon in the GUI.

The use of a communication directory simplifies application development. An application writer has to only create a send and receive directory and the associated config and status files. After that, all communication is achieved by writing files or reading files from the directory.

In addition to the communication directory, we support legacy Java applications by intercepting 'socket' calls in the Java API. These calls are instead handled by OCMP, specifically by the application plugin associated with that application. OCMP guesses the plugin associated with a socket call by looking at the destination port number as well as the first few bytes of the written data. We describe this in more detail in Section IV-D. After the interception, the remainder of the processing is identical as with the communication directory as is described next.

B. OCMP protocol stack

The OCMP client and server stacks that run on a mobile and on a proxy respectively are shown in Fig. 2. We assume that applications or their associated plugins can categorize their communications into either a control or one or more data *streams*. The application control stream provides an explicit control channel between the application plugin peers running on the mobile and proxy. For example, it is used to tell a receiver about the length of the bulk data sent on a data stream, or application parameters required by a peer plugin. It can also convey to the mobile the status of the data transfer between the plugin on the proxy and legacy servers.

Each application data stream is assigned to a *SAR (Segmentation and Reassembly) agent* that segments incoming data into one or more bundles to support data striping across interfaces. These bundles are enqueued at a *connection pool* shared buffer.

The connection pool is an entity that maintains a list of active transport layer connections, one on each interface, and has a shared buffer from which the *OCMP scheduler* can remove bundles. The scheduler sends each bundle on one of the transport-layer connections depending on network availability and the application-specified policy. The scheduler can also decide what kind of a transport layer to use over which interface and send connection requests to the proxy. In order to support mobile devices that switch themselves off to save energy, all bundles in the connection pool are also stored in persistent storage.

Applications select the interface for each bundle by registering application callback methods with the OCMP scheduler. These methods are called when the scheduler has to select an outgoing interface for bundles belonging to that application. The handlers for these callbacks can make application-specific decisions with as much control as desired. Naive application writers can simply return the default interface, while a more sophisticated programmer can take into account variables such as the current time, the energy remaining at the mobile or other relevant factors. For example, applications can send application control messages over a cellular connection and application data on WiFi connections. Other policies can include an intelligent striping mechanism that takes the cost and power consumption on different interfaces into account. Similar policies are also supported in the application plugins running on the proxy, and the policy parameters are conveyed to the proxy by encapsulating them in application control messages.

At a proxy, incoming bundles are processed by a symmetric stack and eventually handed to an application-specific plugin. These plugins can be loaded into OCMP dynamically to use the OCMP API directly. The plugin can then take application-specific actions to transfer the data to a legacy server. The plugin can also fetch data from a legacy server on behalf of an application and store it in the connection pool buffer for the mobile. When a mobile opportunistically connects with the proxy, bundles in the connection pool buffer are enqueued on the appropriate transport layer connection and sent to the mobile.

C. *Session-level reliability*

Due to the presence of a send buffer in the network stack, write calls that enqueue data into a non-empty send buffer return successfully, making an application think that the data was reliably delivered to the receiver, even though it might not be delivered at all if the connection closes prematurely! In this case, the data in the send buffer is actually lost after a connection termination is announced to the application. Similarly, on the receive side, bundles that have been acked by TCP, are not actually passed to the OCMP agent on an unclean disconnection. Hence, with any buffered protocol stack, there is always a possibility that data equal to the sum of the send and receive buffers is actually lost, even though the sending side believes the data to have been delivered successfully. For this reason, transport layer semantics are insufficient for reliable deliver, and a session level reliable data transfer protocol is needed to recover from lost data.

To avoid the overhead of a per-bundle ack or nak protocol, an OCMP sender keeps track of the order in which it transmitted

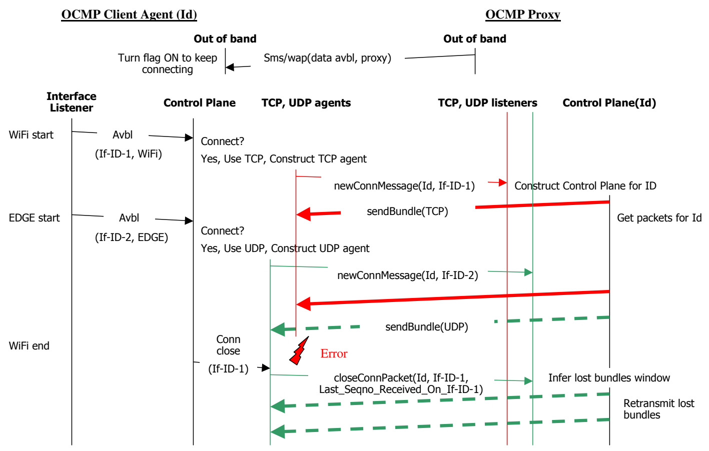

Fig. 3. Control flow sequence diagram

data on each network interface, and retains, in persistent store, all data that might possibly get lost in transit. When a connection closure is detected, the receiver informs the sender of the last sequence number bundle it successfully received on that connection. The allows the sender to infer the set of bundles that were not successfully received. The sender therefore queues them for transmission on a working interface, or marks them as undelivered for subsequent retransmission.

The ability for one end of a connection to inform the other of unclean connection termination on an alternate interface is a useful feature of OCMP. This is because we have found that in practice, one of the ends knows about a disconnection far sooner than the other. This technique allows both ends to reason correctly about the disconnection and to take corrective action. Typically, disconnections are due to wireless failures, which the mobile device finds out about much faster than the proxy. The mobile then sends a disconnection notification, along with the last sequence number it received on the WiFi interface, on its cellular (EDGE) interface. If the proxy was sending some data to the mobile on the WiFi interface, it can then immediately retransmit the data sent on the failed interface after the last sequence number received by the mobile. The proxy responds to a disconnection message with a reply disconnection message that carries the last sequence number it received on the failed interface. In case the mobile was uploading data, it can now retransmit everything it sent after the last sequence number that was received by the proxy. This allows us to quickly recover from a broken connection. We evaluate the performance of this technique in Section VI.

The discussion is illustrated in a sample scenario shown in Fig. 3 for a mobile device that encounters intermittent WiFi connectivity and uses both WiFi and EDGE for data transfer. The protocol begins when the OCMP proxy notifies the mobile device that it has data waiting to be picked up by the device. We assume that these notifications can be sent through an out-of-band mechanism, such as SMS. When the mobile receives this notification, it asks the interface listener module to raise an event whenever the device connects to a new network. Thus, when the

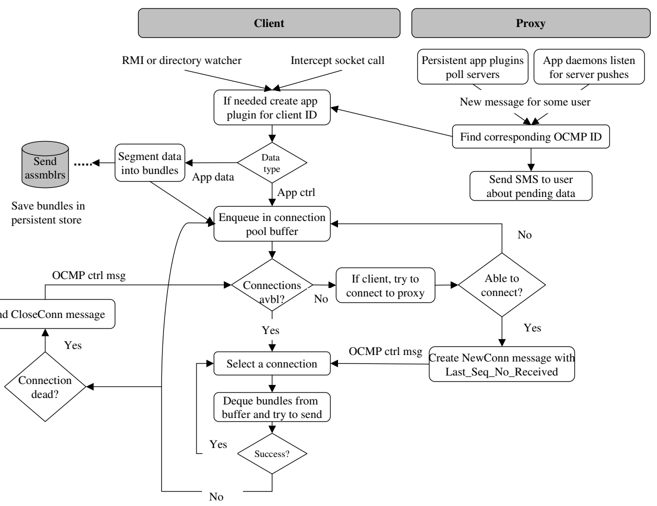
Client
Proxy
RMI or directory watcher
Intercept socket call
Persistent app plugins poll servers
App daemons listen for server pushes
If needed create app plugin for client ID
New message for some user
Find corresponding OCMP ID
Send SMS to user about pending data
Send assmblrs
Segment data into bundles
Data type
App data
App ctrl
Save bundles in persistent store
Enqueue in connection pool buffer
No
OCMP ctrl msg
Connections avbl?
No
If client, try to connect to proxy
Able to connect?
Send CloseConn message
Yes
Yes
Yes
OCMP ctrl msg
Select a connection
Create NewConn message with Last_Seq_No_Received
Connection dead?
Deque bundles from buffer and try to send
Yes
Success?
No

Fig. 4. Processing steps on the sender side

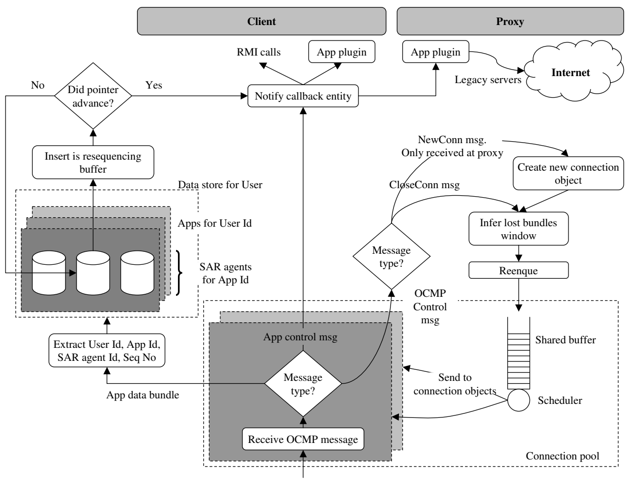

Fig. 5. Processing steps on the receiver side

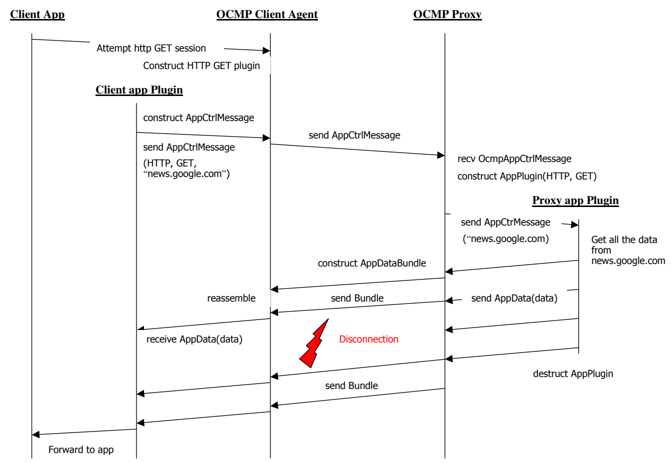

Fig. 6. Data transfer sequence diagram

mobile connects to a WiFi hotspot, the OCMP control layer decides to use TCP as a transport layer on WiFi to connect to the proxy. The connection is initiated through a control message, which first instantiates an OCMP connection pool entity for the mobile on the proxy if it did not exist already. The connection is then added into the connection pool. Similarly, a new transport layer connection is created when the mobile enters into EDGE coverage, this time using a reliable UDP protocol. The proxy can now stripe data on both connections, or use policy feedback from the application to regulate the relative data rates on each connection. If one connection breaks uncleanly, the other connection is used to send control messages to the proxy so that the proxy does not have to wait until a TCP timeout to detect the connection failure.

Fig. 4 and Fig. 5 summarize this discussion and show the processing steps involved on the sender and receiver side.

D. Application-specific plugins

Both the OCMP client and the server support application-specific *plugins*. These short-lived code modules are invoked to carry out application-specific actions for each client-server interaction. All applications need a plugin at the proxy, and legacy application need a plugin at the client end as well. For example, a legacy web browser request on a mobile is associated with an instance of a HTTP plugin both on the client on the proxy that initiates an HTTP GET on its behalf. The proxy-side plugin stores the results in persistent storage and communicates them to the client over opportunistic links. Other examples are a blog plugin to support upload from a mobile device to a blog [22], and a flickr plugin to upload a photograph to flickr [24]. Application plugins attempt to mask a mobile's disconnections from legacy applications either on the mobile or at the content host. Of course, long disconnections that last for hours or days cannot be masked, particularly from interactive applications. However, delay-tolerant applications, such as email and music download, are ideal for opportunistic communication. Note that client side plugins are needed only for legacy applications. If the applications are rewritten to use the 'communication directory', then application plugins are not required on the client.

An instance of a plugin is created on the mobile if OCMP intercepts a socket call made by legacy applications. The destination port number or the first few bytes written into the socket are used to disambiguate different applications from each other, and a corresponding plugin object is created to handle the connections. Whenever a new plugin is created, or a new file is dropped into the 'communication directory', an application control message is also sent to the proxy to ask it to dynamically instantiate a peer plugin on the proxy. We illustrate this through a sample data transfer sequence diagram shown in Fig. 6. Note that application control messages have an application ID and application type field to uniquely identify the correct plugin and the type of the plugin. The plugin then collects *one-time* data from the legacy server, hands it to a SAR agent, and finally destructs itself. A distinct plugin object is therefore associated with each client interaction with the server. Although somewhat heavyweight, this allows us to cleanly handle communication state in the event of a disconnection. We are looking into more lightweight techniques for state persistence in current work.

Persistent application daemons can also be created at the proxy that either monitor legacy servers for updates, or receive 'push'-style updates from the servers. The data from these updates is then handed to an application plugin, which hands over the data to SAR agents in the usual way. If the mobile is already connected to the proxy, an application control message is sent to the mobile to notify it about pending data lying at the proxy. The mobile now either downloads the data into the 'communications directory', or instantiates the appropriate application plugin to handle the incoming data. If the mobile is not connected to the proxy, potentially an out-of-band SMS message can be sent to the mobile to indicate pending data. The client OCMP running on the mobile receives this SMS and tries to connect to the proxy whenever connection opportunities arise.

E. OCMP identifiers

As described earlier, OCMP identifies each mobile device by a unique GUID such as its IMSI [19]. The proxy uses this ID to demultiplex bundles belonging to different users. A different class of identifiers is needed for some applications. Consider a proxy that registers itself as the email server for a set of mobile users using a DNS MX record. When receiving incoming email, the proxy needs to find the user's OCMP-GUID. Therefore, the proxy needs to maintain a mapping from the user's application-specific address, such as an email address, to the user's GUID so that when the user connects to the proxy, it can send data to the correct user.

We have defined a framework on the proxy to support translation from application-IDs to OCMP-GUIDs. A registered application can create a daemon on the proxy that maintains mappings from application identifiers to the OCMP identifiers for all users of that application. This daemon is also registered to receive content from legacy servers. So, when a content server pushes data to the application daemon, it can instantiate an application specific plugin with the correct OCMP-GUID for the user, and redirect the incoming data to the plugin. The plugin caches the data in the usual way and delivers it to the mobile whenever it connects.

Note that each bundle carries a *GUID* to distinguish bundles belonging to different users, an *application identifier* so that bundles can be routed to the correct application, a *SAR agent identifier* for each data stream, and a *sequence number*. A concatenation of the first three identifiers defines a unique *session identifier*.

F. Adaptive striping

An important feature of our work is the ability to stripe data over multiple interfaces. This approach has been used by several systems in the past. However, they have either suffered from poor performance, as described in detail in [7] or require that the

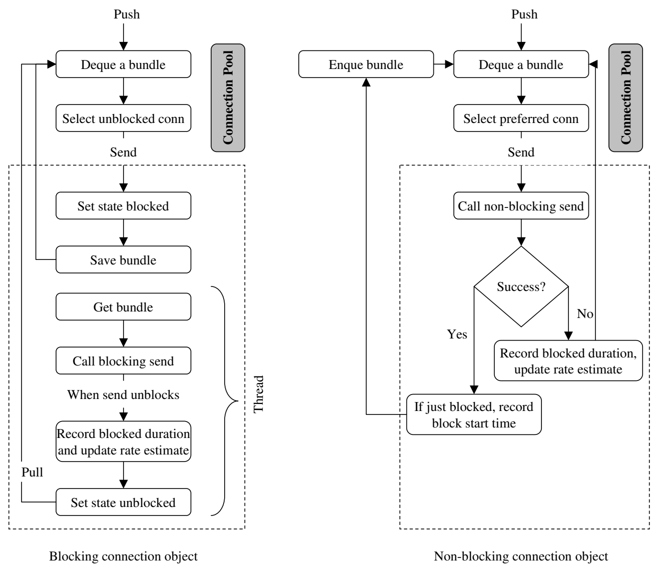

Fig. 7. State maintenance with blocking and non-blocking transport layer implementations

underlying reliable protocol be TCP. Our transport-layer agnostic and rate-based approach to stripe data is described in detail next.

The key problem with application-layer (i.e. transport agnostic) striping, as pointed out in [7], is that data 'stuck' on a slow interface can cause data sent on a faster interface to be held indefinitely in a resequencing buffer at the receiver, causing the buffer to grow, potentially without bound. To begin with, we believe that this is not a significant problem in OCMP because our focus is on delay tolerant applications. This allows us to have essentially infinite resequencing buffers by placing them in secondary storage. Nevertheless, to minimize disk accesses and save power, OCMP uses a simple algorithm to detect when an interface stalls, and then reassigns in-flight data from the stalled interface on to unstalled interfaces. This allows the system to quickly recover from stalled interfaces. We explain this algorithm next.

The first step is to estimate average transmit rates at each interface. This is done through a standard EWMA estimation using a weight parameter of 0.8. The algorithm is triggered immediately after an OCMP bundle is dispatched on an outgoing interface. This is shown in Fig. 7 for two cases, namely, when the transport layer implementation has blocking write calls, and when the write calls are non-blocking [1].

Once the rates are estimated, Algorithm 1 is used to detect stalled interfaces and reassign data. The connection pool object keeps track, for each interface, of how much data had been sent on *other* interfaces while that interface was blocked. If this amount exceeds a stall threshold, then it determines that the interface is stalled, and reassigns a fixed amount of data from the

[1]Blocking transport layer implementations need a thread for each connection. Our implementation uses thread sleeps and waits so that thread overhead is kept at a minimum.

Algorithm 1 Detect stalled interface and reassign bundles

```
$\epsilon$ // Constant: Safety factor
$B$ // Constant: Maximum size of resequencing buffer
$W_j$ // Constant: Expected window size of $j^{th}$ interface
$rate_j$ // Estimated value of rate for $j^{th}$ interface
$stalled_j$ // Whether $j^{th}$ interface is stalled or not
$trackdata_j$ // Data dispatched on interfaces other than $j^{th}$
Function $detect\_stall(D)$ // $D$ = Size of bundle dispatched
// Called on $i^{th}$ interface when some data is dispatched on it
  for each j s.t. $j \neq i$ do
    $trackdata_j \Leftarrow trackdata_j + D$
    $sumrates \Leftarrow \sum_{k \neq j} rate_k$
    if $trackdata_j > B - \epsilon - sumrates * W_j / rate_j$ then
      $stalled_j \Leftarrow true$
      $rate_j \Leftarrow 0$
      reassign $W_j$ bytes of data sent on $j^{th}$ interface
    end if
  end for
  $trackdata_i \Leftarrow 0$
end Function
```

stalled interface to unstalled interfaces. We now discuss how to determine appropriate stall thresholds.

We will illustrate our algorithm by considering just two interfaces, say WiFi and EDGE. We will denote by a *window* the set of bundles that have been sent by OCMP, but that have not been acknowledged by its peer. Let W_{edge} be the maximum window size of EDGE. Then the maximum amount of data that could have been sent on WiFi while the EDGE pipe is being filled up = $r_{wifi} * W_{edge}/r_{edge}$. This is also the maximum amount of data that could be waiting in the resequencing buffer on the receiver for a bundle from the EDGE interface. If the EDGE connection stalls, the data in the resequencing buffer will grow linearly at a rate of r_{wifi}. We wish to avoid this, by transferring bundles away from the EDGE interface to the WiFi interface when a stall is detected.

We know that at time *t* after the EDGE connection stalls, the size of the resequencing buffer is $r_{wifi} * W_{edge}/r_{edge} + r_{wifi} * t$. This has to be always kept less than the maximum size of the resequencing buffer = *B*, within some safety threshold = ϵ. Thus, $t < (B - \epsilon)/r_{wifi} - W_{edge}/r_{edge}$. Now, the amount of data dispatched on WiFi since the EDGE connection got stalled = *D* = $r_{wifi} * t$. Hence, we can substitute for *t*, and claim that the reassignment should take place when $D > B - \epsilon - r_{wifi} * W_{edge}/r_{edge}$. The amount of data to reassign from EDGE to WiFi is simply W_{edge}. This logic is incorporated in Algorithm 1 reflecting the fact that a mobile may have more than 2 interfaces.

We chose conservative values for our parameters. In our system, we use W_{edge} = 16Kb, which is the commonly used default window size in most OSes. Since we multiplex all application data onto the same transport layer connections, we are also able to afford a large resequencing buffer of 2Mb shared across all connections between a client and a proxy.

Note that the need for reassignment is more severe in case of a complete disconnection from some interface. In such cases, we send a disconnection notification on a working interface, instead of relying on the transport layer of the broken interface to detect the disconnection through transport layer timeouts.

V. Analysis

This section presents a mathematical analysis of the performance of a mobile device in a fairly limited environment that supports both (and only) WiFi and EDGE or 1xRTT. Our goal is to compute the expected throughput achieved by the mobile as a function of its mean residence time in the WiFi network and the delay in connecting to a wireless network. We are also able to analytically compare the performance of, for instance, a policy that only uses WiFi with that of a policy that uses both networks when possible. Though preliminary, and with some strong assumptions, we believe that our analysis allows us to get a good intuitive grasp of the expected performance of our system. We compare this analysis with an actual field measurement of the system in Section VI.

For the purpose of analysis, we assume that the connectivity schedules of a mobile device with WiFi networks can be modeled by two random variables: residence time *R* in a WiFi network (with mean μ_r), and disconnection time *D* from a WiFi network (with mean μ_d). We also assume that the device *always* has EDGE coverage, even though it may choose not to use this coverage to save on costs. We consider the connection establishment latencies to be fixed and L_{wifi} and L_{edge} for WiFi and EDGE networks respectively. We also assume fixed throughputs for EDGE and WiFi networks of r_{edge} and r_{wifi} respectively.

In general, the distribution of R and D are strongly dependent on the mobility model. Because this is unknown, we proceed in two steps. First, we compute the expected performance of five different connection policies as functions of some quantities determined by these distributions. These performance measures can therefore be computed as long as we are given the distribution of R and D. Several ongoing research projects are modeling user mobility through raw traces on mobility [5]: we anticipate that we can use their results to compute these values. Second, we show quantitative results comparing these five policies assuming that R is exponentially distributed. We hasten to add that these are merely illustrative. We realize that we cannot draw strong conclusions about the relative performance of the policies based on this assumption. Nevertheless, the numerical results give us an intuitive understanding of the relative performance of the policies that are likely to hold for any distribution of R.

We first assume that we are (somehow) given the following quantities:

- $P_{R>L_{wifi}}$ = Probability of a successful WiFi connection
- $E_{R<L_{wifi}}$ = Expected value of *R* during which WiFi connections cannot be established
- $E_{R>L_{wifi}}$ = Expected value of *R* during which WiFi connections are possible

We use these values to compare the following five policies:

1. **Bandwidth aggregation** Use EDGE all the time and WiFi whenever available.

$$\overline{r} = r_{edge} + r_{wifi}(E_{R>L_{wifi}} - L_{wifi})/(\mu_r + \mu_d)$$

2. **Handoff with connection overlap** Use either WiFi when available or EDGE otherwise. Here, we assume that an ongoing connection is maintained during the time in which the other is establishing a connection. However, after a connection has been established, only one network can be used. This is a reasonable assumption to make because two radios cannot be used simultaneously on legacy devices without making changes to the routing tables.

$$\overline{r} = \{r_{edge}(\mu_d + E_{R<L_{wifi}} - L_{edge}P_{R>L_{wifi}}) \\ +r_{wifi}(E_{R>L_{wifi}} - L_{wifi})\}/(\mu_r + \mu_d)$$

Note that the first expression for the connected duration in EDGE is written as the mean disconnection time from WiFi (=μ_d), plus the time during which a WiFi connection is being established (=$E_{R<L_{wifi}}$), minus the time it takes to reconnect to EDGE if the device was able to switch to WiFi in the first place (=$L_{edge}P_{R>L_{wifi}}$).

3. **Handoff with orthogonal connections** This policy is similar to the previous one, except that we assume that EDGE and WiFi networks cannot be used simultaneously during the connection establishment phase. This scenario is true for some devices with limited capabilities.

$$\overline{r} = \{r_{edge}(\mu_d - L_{edge}P_{R>L_{wifi}})+ \\ r_{wifi}(E_{R>L_{wifi}} - L_{wifi})\}/(\mu_r + \mu_d)$$

4. **WiFi only**

$$\overline{r} = r_{wifi}(E_{R>L_{wifi}} - L_{wifi})\}/(\mu_r + \mu_d)$$

5. **EDGE only**

$$\overline{r} = r_{edge}$$

Given these expressions, which are independent of the distribution of R, in the second step, to generate illustrative quantitative results, we assume *R* and *D* to be exponentially distributed random variables. This allows us to compute the required expressions as follows:

- $P_{R>L_{wifi}} = \frac{1}{\mu_r}\int_L^\infty e^{-x/\mu_r}dx$
- $E_{R<L_{wifi}} = \frac{1}{\mu_r}\int_0^{L_{wifi}} xe^{-x/\mu_r}dx$
- $E_{R>L_{wifi}} = \mu_r - E_{R<L_{wifi}}$

We compared the five schemes in two different scenarios: when the proxy is located *close* to the correspondent host, and when it is located *far* from the correspondent host. A large distance between the mobile and the proxy increases the RTT, reducing the throughput on a connection. We model this through an inverse relationship between the RTT and throughput, as indicated in [9]: $r_{edge} = r_{wifi}/RTT$. Here RTT is the Round Trip Time between a mobile and the proxy. We plot the values of $\overline{r}$ for different schemes and scenarios in Fig. 8- Fig. 12. The parameter values we used for our analysis are listed in Table I. We chose these values based on experimental observations as indicated.

In Fig. 8, we keep the mean residence time in WiFi, i.e. μ_r

[1]In our experiments, we observed a small correlation between signal strength and link acquisition delays. However, DHCP delays always took approximately the same value, indicating that commercial 802.11 hardware have good support for power and rate adaptation depending on the link conditions.

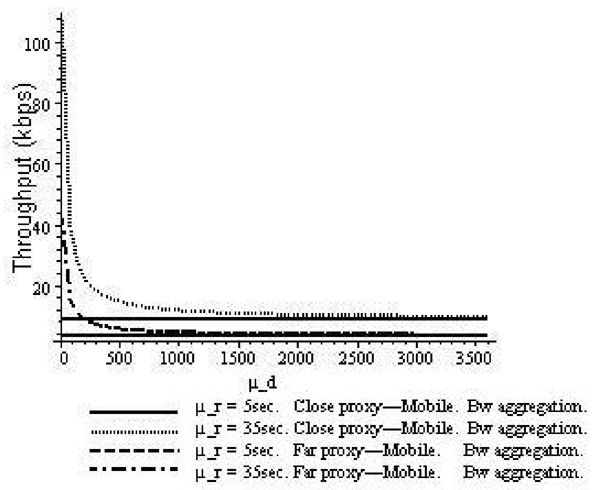

Fig. 8. Increasing disconnection time for fixed mean residence time μ_r

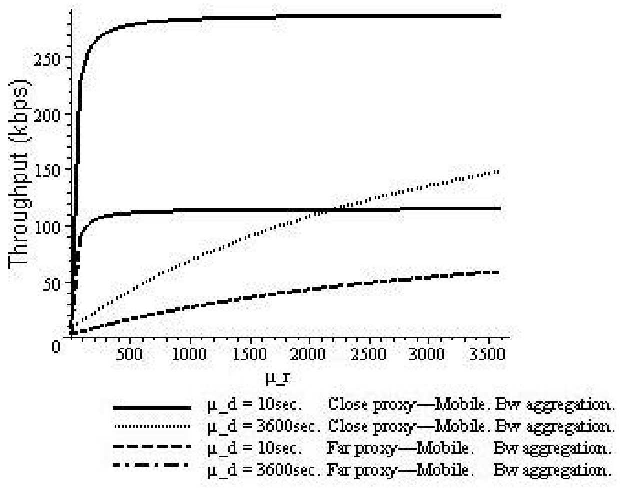

Fig. 9. Increasing WiFi residence time for fixed mean disconnection time μ_d

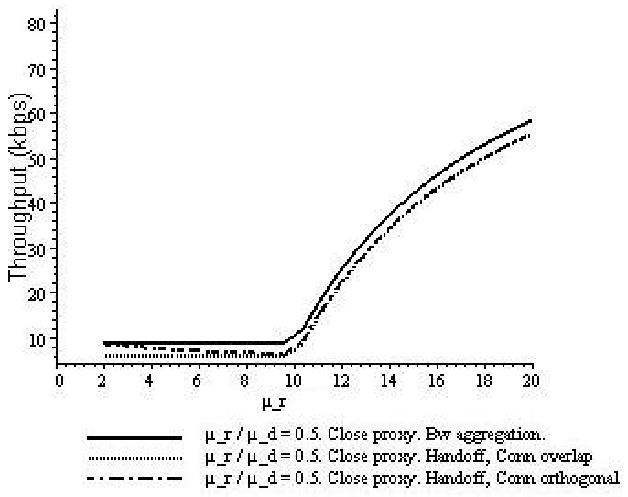

Fig. 10. All schemes, fixed mean degree of connectivity μ_r/μ_d

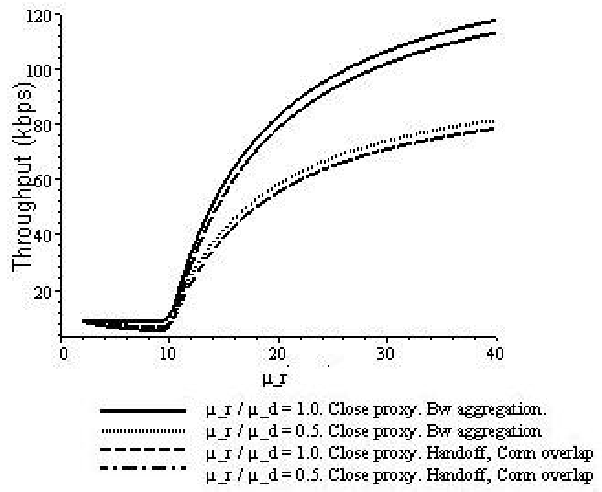

Fig. 11. Different μ_r, fixed mean degree of connectivity μ_r/μ_d

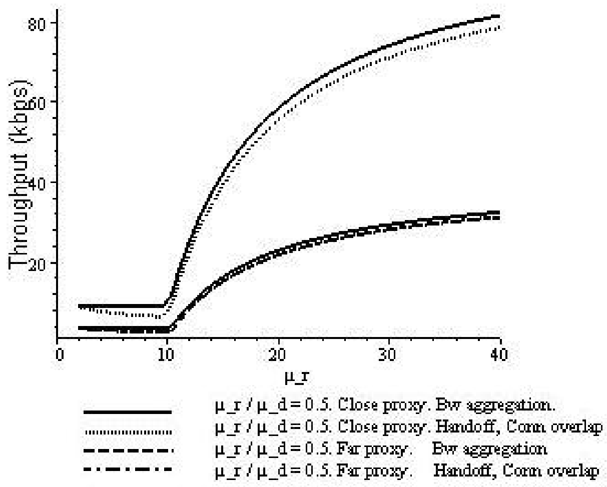

Fig. 12. Fixed mean degree of connectivity μ_r/μ_d

Param	**Value**	**Reason**
r_{edge}	8 Kbps	Expt. observations
r_{wifi}	290 Kbps	Expt. observations
L	8 sec	Expt. observations:[1] 1. Link acquisition = 3 to 5sec 2. DHCP = 2 to 2.5sec 3. Client initializations = 80 to 150ms 4. TCP, proxy initializations = 400 to 500ms
$RTT_{closeproxy}$	40 ms	Ping measurements
$RTT_{farproxy}$	100 ms	Ping measurements

TABLE I
CONNECTION ESTABLISHMENT DELAYS

fixed at 5 sec and 35 sec respectively, and vary μ_d for the bandwidth aggregation policy with the proxy located at two different locations. We see that the mean throughput drops off reciprocally with respect to the amount of disconnection. The asymptotic value is the throughput derived by only using EDGE, which also happens when $\mu_r < L_{wifi}$. We also observe that the proxy placement has a large effect on throughput. Similar behavior is likely for any policy that opportunistically exploits WiFi, that is, achieving nearly r_{wifi} when $\mu_r >> \mu_d$ and asymptoting to r_{edge} when $\mu_d >> \mu_r$.

In Fig. 9, we keep μ_d fixed at 10 sec and 3600 sec, to indicate two different scenarios of dense WiFi connectivity and sparse WiFi connectivity respectively. We observe that as μ_r crosses L_{wifi}, the throughputs for the bandwidth aggregation policy with μ_d = 10 sec saturate at a value of $\mu_{edge} + \mu_{wifi}$. Our analysis indicates that for μ_d = 3600 sec, the saturation will occur much later. Again, we expect these trends to hold relatively independent of the distribution of R and D for any policy that opportunistically exploits WiFi connectivity.

In Fig. 10, we keep the value of μ_r/μ_d fixed at 0.5, and vary μ_r for the first three policies. We notice that in the third policy, where WiFi and EDGE cannot function simultaneously, severe thrashing takes place when $\mu_r < L_{wifi}$ because the device simply switches between networks without getting much useful work done. As μ_r is increased from 1s to L_{wifi}, the probability of switching into a WiFi network increases, and hence the amount of thrashing also increases. Therefore, we see a dip in the value of mean throughput which starts rising after L_{wifi}, and finally catches up at around 11 sec. This can be used to find the *make up* time introduced in [17], which is the minimum connection duration of a WiFi network required to justify the switching latencies and reconnection delays encountered in network switches.

In Fig. 11, we plot graphs for different values of μ_r/μ_d, and we see that the degree of connectivity (i.e ratio of μ_r to μ_d has a greater effect in performance than the choice of scheme.

Finally, in Fig. 12, we keep μ_r/μ_d fixed at 0.5, and plot graphs for the first and second schemes with the proxy placed at different locations for each scheme. We again observe that the closeness of proxy influences performance more significantly than the choice of policy. This corroborates the need for the application state packaging/unpackaging feature of OCMP to relocate data to the OCMP proxy closest to a mobile device. We also see that after a sufficiently large μ_r much greater than L_{wifi}, the throughput averages out to a value of $r_{edge} + r_{wifi}(\mu_r - L_{wifi})/(\mu_r + \mu_d)$ for the first scheme. The asymptotic values of other schemes can be calculated similarly under the limiting conditions of $\mu_r \to \infty$, leading to $P_{R>L_{wifi}} \to 1$, $E_{R>L_{wifi}} \to \mu_r$.

Based on this analysis, we draw the following broad conclusions:

- First, for any policy that opportunistically uses WiFi, the critical parameter is the degree to which WiFi is available. If a mobile device resides for longer durations in a WiFi network, its throughput asymptotically reaches r_{wifi}.
- Second, policies that use WiFi opportunistically do not differ very much from each other: the ability to use the WiFi interface

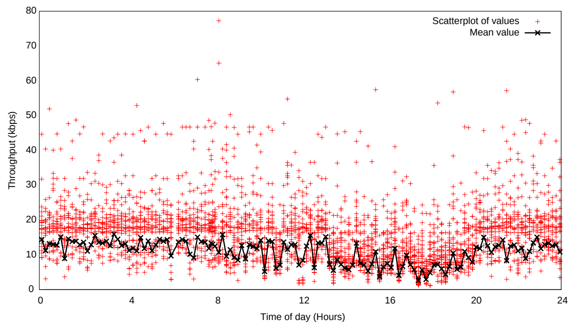

Fig. 13. Scatterplot of 1xRTT application layer throughput

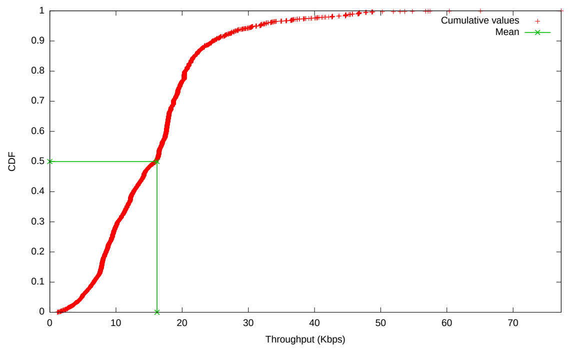

Fig. 14. CDF of 1xRTT application layer throughput

is more significant than the exact choice of policy.

• Third, the placement of a proxy critically influences performance. Ideally, a proxy should be located as close to a mobile as possible.

• Fourth, if a mobile is present for a short time only in a WiFi network, then care should be taken to 'make then break'. Otherwise, thrashing will occur.

VI. Evaluation

A. *Throughput of cellular networks*

We verified the erratic behavior of data on cellular networks [2] by collecting traces of 100Kb file downloads repeated every 10 minutes on an 1xRTT network at different times of the day. As shown by the scatterplot in Fig. 13, the throughput varies considerably. Surprisingly, there appears to be a little correlation between the mean throughput and the time of day.

Fig. 14 shows a cumulative frequency distribution of the throughput traces. The CDF is approximately linear from 5Kbps to 25Kbps, indicating uniform deviations of almost 100% from the mean value of 16Kbps.

Although we did most of our subsequent experiments on EDGE instead of 1xRTT,[2] we expect EDGE networks to exhibit a similar erratic behavior as well. Note that our measurements indicated that the mean capacity on EDGE was only 8 kbps.

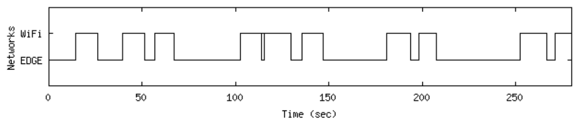

Fig. 15. Connectivity schedules used for experiments

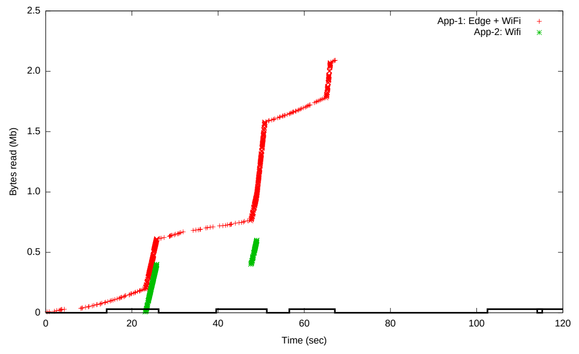

Fig. 16. Download: Client and proxy geographically close

B. *OCMP performance*

For our experiments, we replayed the connectivity schedules calculated using the simulator described in more detail [14]. Briefly, the schedules are generated by modeling a random walk of a mobile on a 2D map of heterogeneous access networks. This network map is constructed by randomly laying out networks with uniformly distributed radii of coverage. We chose parameters to model drive-thru scenarios with small connection durations so that we can illustrate the advantages of OCMP even with brief connection opportunities. Our analysis shows that with larger residence times in a network, the performance will get only better because the WiFi throughput is an order of magnitude higher than the EDGE throughput. The sample connectivity schedule we used for our experiments is shown in Fig. 15.

For our experiments, we used a last hop WiFi link connected to a shared DSL backhaul connection, and an EDGE connection from a commercial cellular service provider. We ran our experiments in two scenarios: (a) client and proxy located close to each other with mean ping times of 30ms, and (b) client and proxy located far apart with mean ping times of 70ms.

[2]This was because of unexpected problems with our 1xRTT NIC, forcing us to replace it with an EDGE NIC

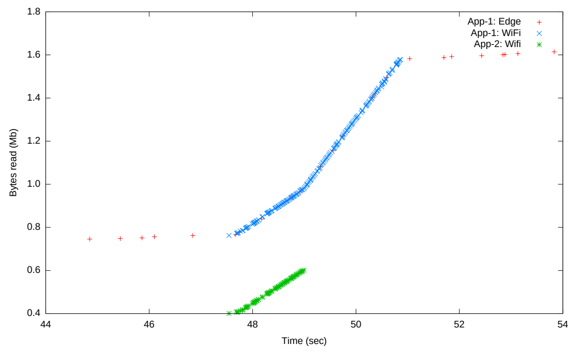

Fig. 17. Download: Client and proxy geographically close: Zoomed in

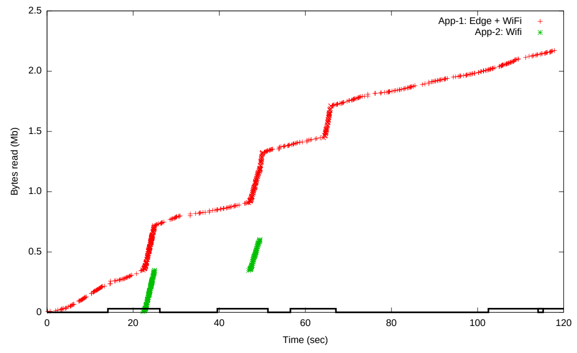

Fig. 18. Download: Client and proxy geographically very far

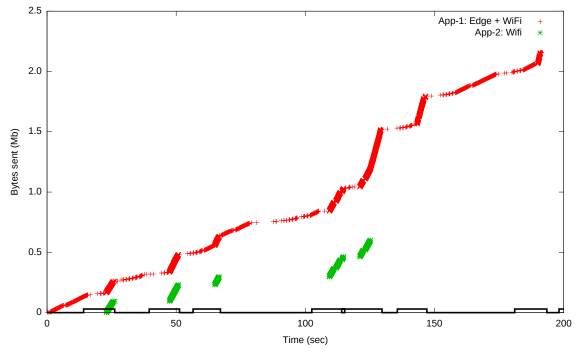

Fig. 19. Upload: Client and proxy geographically close

We first studied download behavior. Fig. 16 shows the trace of an experimental run where the mobile uses WiFi and EDGE simultaneously, and the client and the proxy are located close together. The graph shows two applications with different policies: App-1 uses both EDGE and WiFi, whereas App-2 uses only WiFi opportunistically. During the trace, App-1 downloads 1.9 MB and App-2 downloads 600KB. The 'mesa' markings on the X-axis indicate periods of WiFi network coverage. The trace clearly illustrates the benefits of opportunistic communication: even through a 8-9 sec delay is incurred in WiFi connection establishment, brief connection opportunities can still offer large download spurts.

Fig. 17 shows the same graph with a detailed view from 44 to 54 seconds, during which a WiFi connection is available. Notice the kink at approximately 47 sec, when App-2 terminates and App-1 gets the entire WiFi bandwidth to itself.

Fig. 18 shows the same connectivity schedule replayed with the client and proxy situated geographically far away from each other. The effect of a larger RTT results in lower throughput on both the EDGE and WiFi networks and a general flattening of the graph.

Fig. 19 and Fig. 20 show a trace of application behavior during an *upload* with the same connectivity schedules and application requirements as the download experiments. We observe that uploads are slower than downloads, which is because upload rates on both a DSL backhaul and on EDGE are slower than download rates.

#	Network	Throughput	Delay = *L*	*R-L*
1	EDGE	8.8 Kbps	0	48 sec
	WiFi-I	291 Kbps	9.276 sec	2.714 sec
	WiFi-II	292 Kbps	8.665 sec	3.335 sec
	Average		43.3 Kbps	
2	EDGE	7.5 Kbps	0	48 sec
	WiFi-I	296 Kbps	8.985 sec	3.015 sec
	WiFi-II	293 Kbps	9.246 sec	2.754 sec
	Average		40.9 Kbps	
3	EDGE	7.7 Kbps	0	48 sec
	WiFi-I	287 Kbps	9.276 sec	2.724 sec
	WiFi-II	288 Kbps	8.685 sec	3.315 sec
	Average		42.0 Kbps	

TABLE II
THROUGHPUTS MEASURED WITH A FIXED CONNECTIVITY SCHEDULE

C. Comparison with analysis

In order to verify our measurements with the analysis, we ran two experiments. In the first experiment, we used a fixed schedule for WiFi availability. The connectivity schedule was as follows: 0..12 sec (only EDGE), 12..24 sec (EDGE+WiFi), 24..36 sec(only EDGE), 36..48 sec (EDGE+WiFi). During both connection intervals, WiFi took approximately 9 sec to establish connections and start download.

We find the measured values in Table II to be very close to theoretically calculated values. Assuming that r_{edge} = 8 Kbps, r_{wifi} = 290 Kbps, L = 9 sec, μ_r = 12 sec, μ_d = 12 sec, we calculate:

$$r_{edge} + r_{wifi}(\mu_r - L)/(\mu_r + \mu_d)$$

to be 44.25Kbps, which is very close to the measured values for all three experimental runs.

We also experimented with exponentially distributed WiFi residence times. We computed 30 values of *R* with μ_r = 12 sec offline, and ran experiments with a fixed value for μ_d = 10 sec. The distribution of *R* is shown in Fig. 21. The figure also shows the connection establishment latencies and the actual time available for data transfer. Measured values of the amount of data transferred on EDGE and WiFi is shown in Fig. 22. OCMP downloaded approximately 16.5Mb of data in 8 minutes, giving an average throughput of 35Kbps. We find that the measured value agrees closely to the value obtained from the equation de-

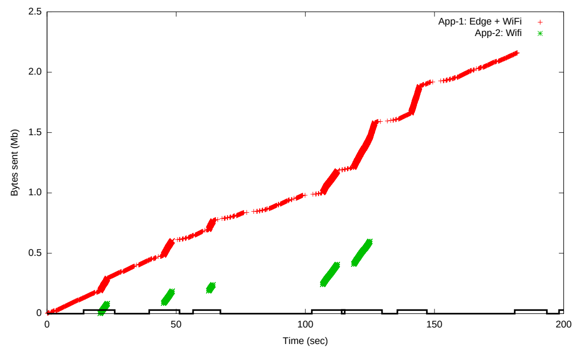

Fig. 20. Upload: Client and proxy geographically far apart

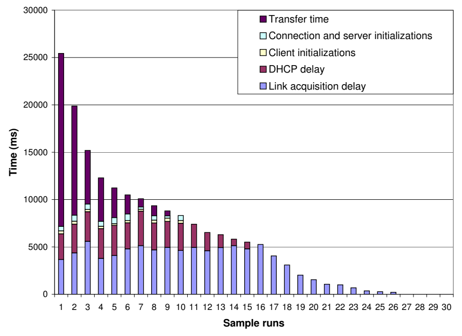

Fig. 21. Transfer times with 30 sample runs

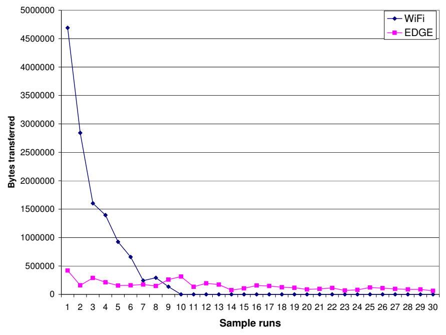

Fig. 22. Bytes transferred with 30 sample runs

rived in Section V, with r_{edge} = 8Kbps, r_{wifi} = 290Kbps, L = 8 sec, μ_r = 12 sec, μ_d = 10 sec. We calculate:

$$\overline{r} = r_{edge} + r_{wifi}(E_{R>L_{wifi}} - L_{wifi})/(\mu_r + \mu_d)$$

to be = 37.9Kbps.

We conclude that our analysis, though rough and ready, is able to accurately predict actual performance.

VII. Related Work

We are not aware of any other work that provides the full set of functionality provided by our system. However, our work is closely related to, and builds on, the insights of several threads of past work in this area, as described next.

Our use of many wireless interfaces in parallel is similar in spirit to pTCP [7]. Unlike pTCP, which assumes an underlying TCP connection, OCMP is transport-agnostic. We are able to make this tradeoff because we are primarily interested in delay-tolerant applications that can deal with a large resequencing buffer. In contrast, pTCP supports interactive applications, and must therefore exploit TCP structure to reduce the size of the resequencing buffer. Unlike pTCP, we do not require any protocol changes in TCP, and do not require any modification to legacy servers, which makes our system more likely to be deployed. Finally, we have built, deployed and evaluated the performance of the system in a testbed, instead of relying on simulations, as in pTCP.

The use of location-independent identifiers for resuming a session was proposed earlier in the context of Rocks and Racks [21] and TCP Migrate [15]. However, these earlier solutions are not only TCP-centric but also support only a single interface. Our use of an almost-always-available cellular connection for the transmission of control messages (i.e. data available, and link down) distinguishes us from these proposals. Also, unlike these proposals, we have designed and implemented a session-level reliability protocol.

Our use of a proxy for dealing with session disconnections and the aggregation of multiple transport connections into a single connection is similar to that proposed in PCMP [12]. However, OCMP differs from PCMP in several ways. First, unlike PCMP, OCMP supports the use of multiple NICs in parallel. Second, unlike PCMP, OCMP nodes can be powered down because application data and control is persistently stored. OCMP allows session state to be encapsulated and transferred from one proxy to another. This allows us to reassign a mobile to the closest available proxy, greatly improving performance. Finally, servers can push data to OCMP proxies, or plugin daemons can poll legacy servers to pull data, and the data can then be picked up opportunistically by mobile devices. We believe that these differences make OCMP much more suited to a dynamic multi-network environment than PCMP.

Last but not the least, our work is complementary to, and extends, recent work in the area of implementing a router for delay tolerant networks (DTN) [4]. Our notions of session persistence, data persistence, bundling, and multi-network support originate in this seminal work. However, we have made several non-trivial extensions. These include the support for fine-grained policy control, the notion of application plugins, the use of a proxy, the separation of the data and control planes, and the the use of Java. Some of our detailed design decisions also differ from that made in the DTN reference implementation. For instance, DTN associates a 'convergence layer' with each transport protocol, which means that all NICs that support TCP would use the same convergence layer. In contrast, we associate a connection with each NIC, allowing us to exploit network heterogeneity even though all transport networks may support TCP. Similarly, OCMP supports a control channel to communicate disconnection and reliable data transfer information between peers. This is not currently possible in DTN. We observe that our work is motivated by a narrower set of problem areas than DTN, which allows us to exploit the inherent problem structure to make these optimizations. In current work, we are extending OCMP to use a 'DTN' transport, so that we can interoperate with a DTN network. Essentially, this allows the OCMP scheduler to act as a DTN node, so that the path from a mobile to a proxy can accommodate multiple disconnected hops.

VIII. Conclusions and Future Work

We have described the design, implementation, analysis, and field evaluation of a system that allows a mobile device to opportunistically communicate using one or more wireless interfaces. The OCMP protocol allows an application designer to take as much control over the details of the underlying communication mechanisms as he or she desires, allowing specification of non-trivial high-level policies. Yet, naive applications can be shielded from the underlying complexities by means of the 'communications directory' abstraction.

Our system makes the following contributions:

- Intelligent application-directed use of multiple network interfaces by means of application-specific plugins and scheduling across multiple transport connections across different interfaces.
- Support for session persistence across disconnections by means of a session-level reliability protocol, data persistence, and the use of session identifiers.
- Support for legacy servers by means of a proxy and application-identifier to OCMP-identifier translation at the proxy.
- Ease of application design and implementation using a 'communication directory'.

These meet the design goals we stated in Section II.

We have implemented the system on a testbed and have used it to develop three simple applications. A 'mobile blog' application allows a user to create a text of photograph blog entry in a communication directory, which is then uploaded to a blog at blogger.com. The 'opportunistic Jabber' application is a port of the XMPP protocol used in Jabber to use OCMP instead of TCP. Finally, OCMP has also been used to create a disconnection-tolerant RSS newsreader. These lead us to believe that our system provides a broad foundation for building new applications that run on smart mobile devices in an environment with multiple wireless networks.

We now outline some avenues for future work:

- **Policy design:** Although OCMP provides support for fine-grained policy control, the actual specification of non-trivial control policies continues to be an open problem. We are currently investigating how to balance parameters such as the cellular pricing plans, power consumption, mobility prediction, and application requirements to design sophisticated policies. To obtain continuous indications from the lower layers about power levels and battery life is yet another issue that still remains to be solved.
- **Detection of WiFi networks:** Our work assumes that a mobile can easily detect the presence of a wireless network. However, keeping a mobile powered on awaiting an opportunistic connection wastes power. Indeed, leaving a WiFi NIC on at all times in idle mode consumes 200mW and even sleep mode consumes 60mW [1]. Proposals have recently been made for hierarchical radios [16] where a low power WiFi detector is first used to detect WiFi signals and then power on the rest of the radio. We hope to incorporate these ideas into our system.
- **Detection of disconnections from WiFi networks:** Detecting when a network is no longer accessible is a hard problem because the signal strengths of wireless networks vary dramatically even within a coverage area, introducing uncertainty whether the mobile is temporarily in a fade, or actually out of range. Recent work proposes that three consecutive retransmission failures most likely indicates a disconnection [20]. We hope to use this insight into our system to efficiently detect disconnection.
- **Power efficient device design:** We believe that not only should a radio interface on a mobile device be power efficient, perhaps the entire device should be restructured to allow opportunistic access. For example, a mobile device can be powered down when not in use, and rapidly boot up it only when a wireless network is detected. This could be accomplished by a stripped down OS that is only capable of sending and receiving data and can be booted nearly instantaneously from flash memory. The rest of the OS should be loaded only when it is required, i.e. when there is data to send or receive.

To sum up, we believe that our work opens up several avenues for future work. We intend to pursue these and other related efforts in our quest to provide application support for multi-network opportunistic communication.

References

[1] Atheros Communications, "Power Consumption and Energy Efficiency Comparisons of WLAN Products," www.atheros.com/pt/whitepapers/atheros_power_whitepaper.pdf, 2003.

[2] R. Chakravorty, A. Clark, I. Pratt, "GPRSWeb: Optimizing the Web for GPRS Links," Proc. ACM/USENIX MOBISYS, 2003.

[3] M. Chan and R. Ramjee, "Improving TCP/IP Performance Over Third Generation Wireless Networks," Proc. IEEE INFOCOM, 2004.

[4] M. Demmer, E. Brewer, K. Fall, S. Jain, M. Ho, and R. Patra, "Implementing Delay Tolerant Networking," Intel Research, Berkeley, Technical Report, IRB-TR-04-020, Dec 2004.

[5] R. Jain, D. Lelescu, M. Balakrishnan, "Model T: An Empirical Model for User Registration Patterns in a Campus Wireless LAN" Proc. ACM MOBICOM, 2005 .

[6] K. Fall, "A Delay-Tolerant Network for Challenged Internets," Proc. ACM SIGCOMM 2003.

[7] H. Hsieh and R. Sivakumar, "A Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-homes Mobile Hosts," Proc. ACM MOBICOM, 2002.

[8] M. Kozuch and M. Satyanarayanan, "Internet suspend/resume," Proc. Workshop on Mobile Computing Systems and Applications, 2002.

[9] J. Kurose and K. Ross, "Computer Networking," Addison Wesley, 3rd Edition, pp.271, 2004.

[10] R. Moskowitz, P. Nikander. P. Jokela, T. Henderson, "Host Identity Protocol," http://www.potaroo.net/ietf/ids/draft-ietf-hip-base-00.txt, 2004.

[11] D. Nystedt, "Intel Slashes PC Power-up Time," http://www.pcworld.com/news/article/0,aid,123053,00.asp, Oct 2005.

[12] J. Ott and D. Kutscher, "A Disconnection-Tolerant Transport for Drive-thru Internet Environments," Proc. IEEE INFOCOM 2005.

[13] A. Seth, S. Bhattacharya, S. Keshav, "Opportunistic Communication Over Heterogeneous Access Networks," Technical report, Sprint Labs, CA, April 2005.

[14] A. Seth, N. Ahmed, S. Keshav, "Mobility Decisions in Heterogeneous Wireless Access Networks," Manuscript, University of Waterloo, Dec 2004.

[15] A. Snoeren and H. Balakrishnan, "An End-to-End Approach to Host Mobility," Proc. ACM MOBICOM 2000.

[16] J. Sorber, N. Banerjee, M. Corner, S. Rollins, "Turducken: Hierarchical Power Management for Mobile Devices," Proc. ACM/USENIX MOBISYS, 2005.

[17] M. Stemm and R. Katz, "Vertical Handoffs in Wireless Overlay Networks," In Mobile Networks and Applications, Volume 3, Number 4, Pages 335-350, 1998.

[18] O. Tickoo, V. Subramanian, S. Kalyanaraman, K. Ramakrishnan, "LT-TCP: End-to-End Framework to Improve TCP Performance over Networks with Lossy Links," Proc. IEEE International Workshop on Quality of service (IWQoS), Jun 2005 .

[19] V. Vanghi, A. Damnjanovic, B. Vojcic, "The CDMA2000 System for Mobile Communications," Prentice Hall, 1st Edition, pp.224, 2004.

[20] H. Velayos, "Autonomic Wireless Networking," Doctoral thesis, TRITA-S3-LCN-0505 , ISSN 1653-0837, ISRN KTH/S3/LCN/–05/05–SE, Stockholm, Sweden, May 2005.

[21] V. Zandy, and B. Miller, "Reliable Network Connections," Proc. ACM MOBICOM 2002.

[22] "Blogger API," http://www.blogger.com/developers/api/1_docs.

[23] "Bytemobile," http://www.bytemobile.com.

[24] "Flickr API," http://www.flickr.com/services/api.