

# E-mail4B: An E-mail System for the Developing World

Michael Demmer, Bowei Du, and Matthew Piotrowski

*Computer Science Division  
University of California, Berkeley  
Berkeley, CA 94720*

{demmer, bowei, pio}@cs.berkeley.edu

## Abstract

We present E-mail4B, a prototype e-mail delivery system targeted for developing regions. Effectively deploying networked applications in these challenging environments requires adopting a new architecture design to address the inherent cost and infrastructure demands. Our project evaluates the Technology and Infrastructure for Emerging Regions (TIER) architecture through an initial development effort. While we conclude that the architecture is in general well-suited to the task at hand, we also conclude that several key aspects of the protocols and APIs should be altered, and present concrete suggestions for the evolution of the architecture.

## 1 Introduction

The Internet's E-mail infrastructure has emerged in much of the developed world as an indispensable communications medium. Many disparate societal groups, from corporate executives to teenagers, have embraced the inexpensive and convenient nature of E-mail messaging. As the communications medium has grown in reach, it has become crucial to commercial transactions, financial services, and electronic banking.

As pervasive as the spread of E-mail has been in the developed world, it has been correspondingly lacking in the developing world. These regions suffer from financial and infrastructure constraints that have been largely prohibitive of the development of a reliable networking infrastructure to enable E-mail communication.

Yet the potential impact of an E-mail infrastructure on the developing world is likely to be as compelling, if not more so, than it has been in the developed world. Much of the developing world suffers from a lack of a reliable communications infrastructure of any form. Per capita telephone and networking penetration is low, and communication technologies in general are unreliable, expensive, and can suffer from long outages.

To date, the efforts to "bridge the digital divide" have largely consisted of extending the traditional Internet in-

frastructure to reach communities in the developing world. Yet many of these efforts suffer from technical and financial constraints that have limited their effectiveness.

The E-mail4B system is intended to address the specific problem of bringing an E-mail infrastructure to the developing world. Rather than simply following the design paradigms of the traditional Internet, this system addresses the constraints of the developing world by adopting a new technology platform specifically targeted for developing world applications. As many other projects have found, this region is a harsh one for networked application development: unreliable power infrastructure, little network connectivity, an uneducated and largely illiterate end-user population, and orders of magnitude smaller per-capita income are just some of the challenges faced in this arena.

The E-mail4B system is being developed as part of the Information and Communications Technology for Billions (ICT4B) [1] and Technology and Infrastructure for Emerging Regions (TIER) [8] research initiatives. The central goals of these initiatives are to provide the technical and social resources necessary to effectively develop applications for the billions of people who currently have very limited access to technology.

This paper describes our initial efforts toward evaluating the goals of this infrastructure. The central goal of the project is to develop a prototype E-mail delivery system that follows the design guidelines of the TIER platform. Through this initial implementation, we aim to evaluate the APIs and assumptions of the platform to gain insights into aspects of the platform that need future work.

The rest of the paper is organized as follows. Section 2 describes the TIER platform and describes the key motivations for the project. Sections 3 and 4 describe our initial prototype implementation and some evaluation experiments of the prototype. Section 5 outlines the results and insights we gained about the platform. Finally, Section 6 describes our future work.

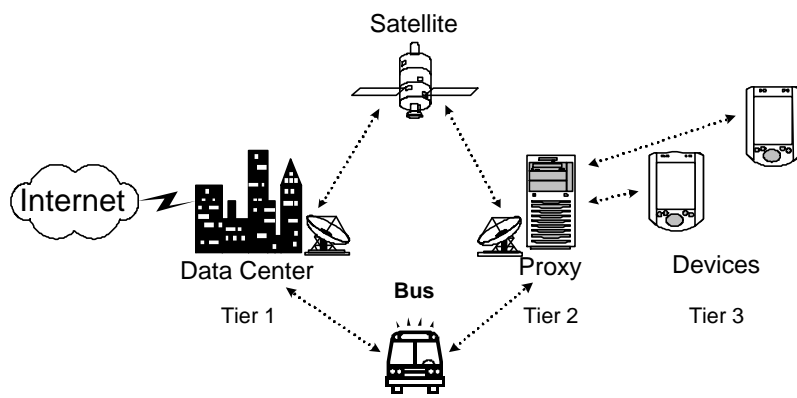


Figure 1: TIER Infrastructure

## 2 Background / Motivations

### 2.1 ICT4B Initiative

The central goal of the ICT4B initiative is investigate ways in which technology can improve the lives of people in developing regions. This idea is not new; on the other hand, our approach to achieving this goal is. Unlike other projects, we do not seek to simply bring old technology from the first world to the third world. We believe that the needs and the environment of these regions are fundamentally different, and that they can greatly benefit from technology designed specifically for them.

One of these benefits is cost. We seek to move as much processing and storage as we can away from the end user and into the infrastructure. This will allow for cheap end user devices, and common infrastructure that is highly utilized and shared, amortizing costs over a large number of users.

Another benefit is usability. Many people in these regions are illiterate or semi-literate. Thus, one can imagine difficulty with standard user interfaces. To solve this problem, we are developing low-cost, multilingual speech recognition [7]. We are also starting to work very closely with sociologists and local organizations that can help bridge the usability gap.

### 2.2 TIER Infrastructure

One of the central motivations of the E-mail4B project is to evaluate the TIER infrastructure. In this section, we describe this infrastructure, which can best be thought of as having three levels, or “tiers” (see Figure 1).

The first tier consists of a large data center. This system is intended to be capable of supporting millions of users. Clustering technology will be employed to ensure high availability. It can be thought of as having essentially unlimited computing power and unlimited reliable storage. As indicated in Figure 1, the data center will likely reside

in a major urban area with a reliable power source and a standard connection to the Internet.

The second tier consists of smaller, PC-class proxies. They are capable of supporting hundreds or thousands of users and contain a reasonable amount of processing power and storage. In general, they are connected to the data center through intermittent network connections. Examples of such connections include a low earth orbiting (LEO) satellite, a solar-powered wireless router with intermittent sunlight, or a wireless router attached to a bus. We intend that proxies will be owned and operated at a village level.

The third tier consists of small, handheld devices (similar to a PDA but even more resource-constrained). They are capable of supporting one or a handful of users. They contain just enough processing power and storage to relay information between the user and the proxy and to support a simple user interface. Devices have an intermittent connection to the proxy that is governed by the user’s mobility (basically, the device and proxy are connected when the user is within wireless range in the village).

The aim of this tiered architecture, as stated above, is to reduce cost. The idea behind this is that the device will offload processing and storage to the local village proxy; proxies, in turn, will offload processing and storage to the data center. In fact, we view the data center as having the only true copy of data in the system and the proxies and devices as simply caches of this data. Such a distinction relaxes the reliability requirements of the proxies and devices because we can always restore them if necessary. In this setup, cost savings occur because the more expensive, resource-rich entities are also the more highly shared entities, allowing costs to be amortized over a large number of users.

In developing the E-mail4B system, we aim to evaluate the effectiveness of this tiered architecture. Since the goal of this infrastructure is to move processing and storage away from the end user devices, it is useful to take an actual application and see how well it can accomplish this. Thus, we fully embrace the idea of a data center, proxies,

and devices in our implementation.

## 2.3 Delay Tolerant Networks

Note that in this TIER infrastructure, each tier is connected to the next tier via intermittent links, i.e. links that come and go. Traditional Internet protocols were not designed with this type of network in mind. They depend on the existence of a relatively low-latency path all the way from the source of the communication to its destination. Yet the TIER infrastructure expects that this is an unrealistic goal, and what is needed is a protocol that realizes such a path may not exist.

Fortunately, this problem is being worked on in the context of Delay Tolerant Networking (DTN) [2][3][4]. The basic idea of the DTN is that at each point in the network, the data to be transmitted is stored in case it can't be transmitted immediately. More advanced aspects of the network layer seek to answer questions like the following:

*What data should be sent if everything can't be sent?*

*What notifications should be given to senders about the success or failure of delivery?*

*Should the sending of data on a high latency link be postponed in expectation that a low latency link will soon become available?*

## 2.4 TIER Traffic Class API

To hide some of the complexity of managing the DTN's features, an abstraction layer of network Traffic Classes [6] has been developed. This API acts as an intermediary between the logical network requirements of the application and the details of the DTN. The core offering of the API is a logical separation of network traffic into classes.

There are a number of parameters on which this classification can be done. Some of these include:

- priority level of the traffic
- the direction of the traffic
- the desired reliability
- the utility of a piece of data over time
- the desired bandwidth

As mentioned previously, one of the key motivations of the E-mail4B project is to evaluate the design of the TIER API. We want this API to be as useful and as easy to use as possible for potential application developers. Indeed, one of the goals of the ICT4B/TIER initiatives is to have others build upon our work. This has a much higher chance of happening if we provide good, easy-to-use tools. So E-mail4B set out to use the TIER API to create an entire mail system. By utilizing the API for a trial application development, strengths and weaknesses of the API would be exposed. Feedback can then be propagated back to the TIER and DTN designers, improving the infrastructure elements.

## 2.5 Why Email?

We chose e-mail as a test application for evaluating the TIER infrastructure for several reasons. First, e-mail is inherently a delay tolerant application. In contrast with interactive applications such as online web browsing or video conferencing<sup>1</sup>, E-mail fits more with a paradigm of postal mail, in which users can expect delays in communication on the order of hours to days. Thus, we can expect that a usable e-mail system should be the baseline application which can be deployed with the TIER infrastructure.

Second, e-mail is an application that is well suited to exercising TIER traffic classes. Small mail headers and status updates are naturally distinct from larger mail bodies and attachments. There is also a natural distinction between the one-way delivery of mail from data center to proxy and the two-way interactivity of a user checking mail in a village.

Finally, email fits in nicely with the overall goal of the ICT4B project: to improve the lives of people in developing regions through technology. Despite its simplicity, email can be used to carry vital information. For example, it is well-known that farmers are often cheated by middlemen because of a lack of information regarding crop prices. A simple e-mail system to find out pricing information would go a long way to realizing more efficient markets and could substantially improve the income and quality of life of poor rural farmers.

Another example is personal communication. People in developing regions often have relatives who have left the local area, and they are willing to pay a substantial fraction of their income to communicate with these relatives. E-mail has the potential to be a much cheaper way to do this. And even if email as we think of it is unusable because of the literacy problem, an email system can easily be converted to carrying voice messages instead of text messages.

## 3 Prototype Implementation

### 3.1 Implementation

The E-mail4B system consists of three applications running at the data center, proxy and device layers in the TIER architecture (see Figure 2). We implemented the E-mail4B system in C++ on top of the Linux operating system. The server, proxy and device code was written using an event driven model which integrated the TIER and DTN network layer events with other file descriptors and system timers. A simple mock up GUI for interacting with the e-mail on the device end was created using Python/Tk. This GUI supported basic operations such as sending, checking and reading e-mail.

Reliable, crash recoverable data storage in the data center and proxy was achieved by using an off the shelf

---

<sup>1</sup>Note that real-world deployments of such interactive applications on the Internet do occasionally experience significant latency and data loss, resulting in less-than-ideal service quality.

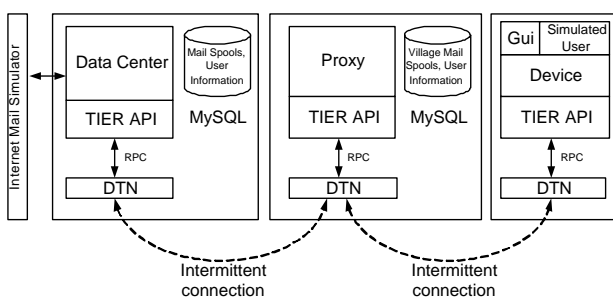


Figure 2: E-mail4B architecture

MySQL relational database to store permanent state. The choice of a database system for storing e-mail was motivated by two factors. First, the use of the database system took care of concerns for the integrity of the data held in the data center and proxy in the event of faults. Second, it is likely that many of the applications (including e-mail) will have similar reliable storage requirements, i.e. that the database will provide an easy centralized means of administering data storage. However, we only rely upon basic database features, and selected a relational system due to the reliability semantics and ease of use as opposed to a need for advanced querying and indexing features.

Communication between the three tiers of the system uses the TIER/DTN networking protocols. The DTN network layer provides automatic store and forward functionality in the event of network disconnection. The TIER api offers differentiated traffic classes for network traffic. Differentiated traffic classes allows us to prioritize network communication between small, time sensitive traffic and large bulk data transfers. For instance, we used the `EVENT` class of TIER messages to communicate message requests and status updates, while mail content was transferred using `DATA` class. The rationale behind these assignments is that status updates, because they are smaller and more time critical, should have higher priority over mail content. Mail content, being larger and having static content, should be viewed as a bulk data transfer.

Because the TIER/DTN api did not provide connection notification, we were forced to create an out of band mechanism to detect the presence of a low-latency network. This connection status monitor was implemented by sending heartbeat ping messages over TCP between connection endpoints. If heartbeat messages did not arrive within a certain latency, the network link was considered to be down. It is important to note that the heartbeat messages would not have been appropriate to send via the DTN protocol. This is because connection monitoring packets do not need the store and forward functionality of DTN and have a very short useful lifetime. Connect status was used as an indication to the user as to when an upgraded user experience was possible because of the existence of a low latency connection between the device and the proxy.

## 4 Experiments

We evaluated the performance of our system in terms of a comparison between e-mail delivery in a system with intermittent connectivity accomplished using E-mail4B and the optimal time of possible delivery. Optimal is defined to be the time of arrival of a mail message given that network delivery is instantaneous. This implies that a message is forwarded to its next hop as soon as the link between the hops is available.

All experiments were performed on a four processor Intel Xeon 2 GHz PC, with the applications communicating using loopback interfaces.

### 4.1 Simulation Environment

Network connectivity conditions are simulated using `iptables` firewall rules driven from a connectivity simulation script. Periods of disconnectivity/connectivity were simulated by inserting/deleting firewall rules to drop all packets between a particular pair of loopback interfaces. Periodicity was determined by a preset event schedule which simulated daily periods of connectivity, e.g. the arrival of data mules or connection due to having a LEO satellite overhead. We ran our simulations using a scaled passage of time, that is one second of wall clock time corresponded to a scale factor number of seconds passing in simulation time.

External and internal mail sent during the simulation was generated by random processes in the data center and device. The following tables contain the parameters of the random mail generation engines:

#### External Mail Parameters:

Parameter	Description
<code>user id</code>	User that the e-mail is addressed to
<code>delay</code>	Delay between mail sending events
<code>size</code>	Length of the e-mail messages sent

#### Device Mail Parameters:

Parameter	Description
<code>user id</code>	User that the e-mail is destined for
<code>delay</code>	Delay between sending/checking mail
<code>size</code>	Length of the e-mail messages sent
<code>read %</code>	Fraction of e-mails read
<code>delete %</code>	Fraction of e-mails deleted

Each of the parameters was determined by selecting a value from a probability distribution. Additionally, the device only generated and checked mail while it was “connected” to the proxy, as determined by the connectivity monitor.

## 4.2 Experimental Setup

We ran our system using a single data center, proxy and device. The intermittence of the connection between the data center and proxy and the proxy and device was controlled with the following connectivity simulation script:

```
# user1 comes to town at 10am and stays
# for five hours, repeats every 10 hours
User1 <-> Prox1 periodic start=10:00:00m \
  period=10:00:00 duration=05:00:00

# proxy to data center connection up
# 1/2 the day from 6pm to 6am
Data1 <-> Prox1 periodic start=18:00:00m \
  period=24:00:00 duration=12:00:00
```

These schedules of intermittency was chosen such that at some times, it is necessary to use a store and forward mechanism to deliver the e-mail, whereas at other times, an end-to-end link is available from the data center, through the proxy, to the device.

The experiment was run with a scaling factor of 300, meaning that every second of wallclock time was simulated to be 300 seconds, or 5 minutes of realtime.

The external mail generation parameters were:

Parameter	Value
user id	Set to 1, as there was only one device.
delay	Chosen from a normal distribution having mean of 20 minutes and a standard deviation of 5 minutes.
size	Uniform from 100 to 500 bytes.

Device mail generation parameters were:

Parameter	Value
user id	1
delay	5 to 15 minutes
size	100 to 500 bytes
read %	95%
delete %	20%

## 4.3 Experimental Results

Figure 3 is a graph comparing the optimal e-mail delivery time for an optimal network versus the delivery time of our implementation. The first graph is e-mail delivery from the data center to the device. The second graph is the e-mail delivery from the device to the data center.

Maximum	59 s.
Average, device to data center	11.6 s
Standard deviation	12.32 s
Average, data center to device	19.16 s
Standard deviation	11.77 s

The above table shows the maximum, mean and standard deviation of the differences in delivery time between

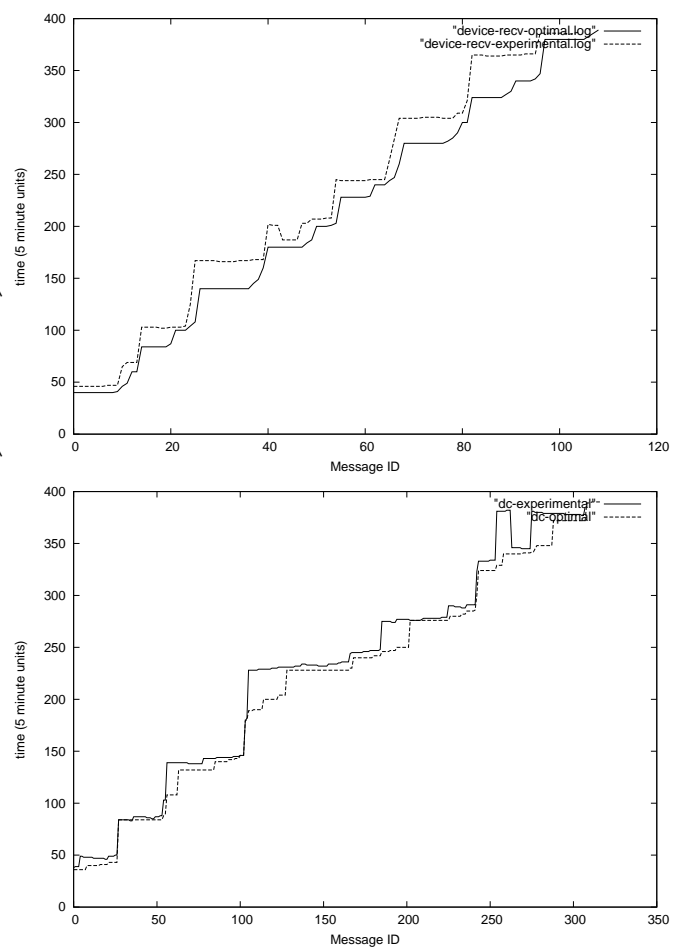


Figure 3: DC to Device and Device to DC mail arrival times

the actual experimental value and the optimal mail delivery time. The units of the difference is given in wall clock seconds. There are several interesting results that can be seen from the data.

First, the differences between the experimental and optimal results when translated to simulated time is quite large. We found that while we were able to speedup the rest of the system by the simulation scaling factor, the DTN code sometimes introduced delays on the order of tens of seconds of wallclock time. The hard real delay introduced by DTN greatly dominated simulated time. In an actual deployment, (i.e. running the simulation with a scale factor of 1), the DTN delays would have little impact on actual system performance.

Second, there are several small dips in the delivery time of the mail. The experimental message delivery time is not a monotonically increasing function. This can be explained by the fact that DTN code reorders packets sent. While mail messages were generated with in order message id's, the order in which DTN delivered the messages is may not follow the same ordering.

Finally, we note that the standard deviation in delivery

time of the data center to device direction is much greater than that of the device to data center direction. One explanation of this phenomenon is the protocol which was used for e-mail delivery on the device. The device only checked and created e-mail while it could sense that it was connected to the proxy. This meant that e-mail flowing from the device would only be subjected to variations in link behavior on the data center to proxy link. On the other hand, e-mail flowing from the data center to the proxy is subject to the conditions both network links in addition to randomness in the frequency of mail checking by the device.

## 5 Results / Lessons

As mentioned previously, the central goal of our project was to evaluate the TIER architecture and associated APIs as a platform to develop applications that target developing regions. In general, our conclusions are positive, as we found that the architecture was appropriate for the task of developing an email system.

One metric by which to evaluate a development platform is the time and effort that it takes to implement an application within the environment. In the span of less than two months, we developed a fully functional prototype implementation of the email system. The TIER traffic class API was well-suited to support the communication needs of the application, and we were able to take advantage of the DTN layer to handle the intermittent networking.

The general approach of developing applications for a multi-level architecture (i.e. data center, Proxy, Device) is well suited to an email system. Relying on the data center as the only truly reliable storage simplified our coherency algorithms, as the data center can resolve potential inconsistencies and conflicts. In addition, the device implementation was greatly simplified by the ability to rely on the proxy to handle some processing on its behalf. This benefit would be even more pronounced in a more advanced email system, in that the proxy could handle complicated transcoding or other operations, essentially reducing the device to just the user interface and simple cache management operations.

However, we have identified several key improvements and additions to the TIER and DTN layers that would improve the applicability of those infrastructure elements. With specific improvements to the traffic class API, the addition of connectivity state information to the DTN API, some other minor improvements to the DTN api, and the addition of a general cache management API, the overall TIER platform can be even better suited to support an email system as well as many other developing world applications.

### 5.1 Traffic Classes

The TIER traffic class networking API (see Table 1) is still in its formative stages, and as such, is still lacking in core functionality. Specifically, the API is currently just a thin layer atop the DTN client API, and does not actually implement differentiated handling for message classes. In the current implementation, the `confset` parameters are ignored and the separate `tier_send_[CLASS]` functions map to the same underlying functionality. Though our simple evaluation experiments did not depend on differentiated service offering from the various traffic classes, a more fully-functional implementation could leverage a more capable network layer to simplify the application processing.

One example where this lack of real traffic classes occurred in our experiments relates to the protocol between the device and the proxy. One basic operation of the device is to poll the proxy to see if any new mail has arrived for it<sup>2</sup>. If the network layer becomes disconnected, then it is possible that the device may send multiple query packets during the disconnected period. The DTN agent will queue these request packets, and when the connection is re-established, the proxy will get a series of redundant query packets in succession, all requesting the same message data.

This problem exposed a weakness in our application logic. In the “new mail query” message, the device includes the set of message headers that it is requesting, and the the proxy simply responds to the query with any matching messages. This means that if a number of messages arrive at the proxy while the device is disconnected, and the device subsequently connects, the proxy will actually send the new set of message headers to the device multiple times in a row, clearly wasting network resources.

While this problem could easily be addressed with enhanced application logic, a better approach would be to offer a traffic class that provides a form of at-most-once semantics for a message type. With this traffic class, the aforementioned problem could be easily solved by simply designating the “new mail query” packets in this periodic traffic class. The semantics of this class would ensure that even in the case where a device sent multiple queries when it was disconnected, multiple copies of the message that end up queued for delivery would be eliminated. Therefore upon being reconnected, only one such query would be sent to the proxy, and the redundant mail delivery traffic would be eliminated. In fact, with this traffic class available, the device could have a trivial update algorithm that wouldn’t need to keep track of whether or not it could communicate with the proxy or not; the device could just send query packets at a fixed interval and the network layer would handle the delivery semantics.

---

<sup>2</sup>An alternative protocol design would be for the proxy to proactively forward new message headers to a device. The polling functionality is used to allow the device to control the number of messages that it receives in a single burst. We plan to revisit this design choice further in a future implementation.

Function	Description
<code>tier_initialize(localname, dtnhost, dtport)</code>	Set up the connection to the bundle agent
<code>tier_close()</code>	Shut down the connection
<code>tier_register_receive_callback(callback, cdata)</code>	Register a function to be called on bundle arrival
<code>tier_load_confset(name)</code>	Retrieve the named confset
<code>tier_store_confset(name, confset)</code>	Store a mapping of the name to the confset
<code>tier_send_EVENT(confset, msg, len, dst)</code>	Send a bundle in the EVENT traffic class
<code>tier_send_DATA(confset, msg, len, dst)</code>	Send a bundle in the DATA traffic class
<code>tier_send_PERIODIC(confset, msg, len, dst)</code>	Send a bundle in the PERIODIC traffic class
<code>tier_send_TWOWAY(confset, msg, len, dst)</code>	Send a bundle in the TWOWAY traffic class

Table 1: Tier Traffic Class API v1 (simplified)

There are two key design reasons to place this type of functionality into the network layer. First of all, it vastly simplifies the requirements on the application developer. Secondly, this type of network service can be used in a range of applications. Besides the above example, this type of traffic class is a very natural fit for periodic information dissemination applications, such as price updates, stock quotes, or weather forecasts. In all these cases, historical data is of little value; the true goal of the application is to deliver the most recent state of affairs to the intended recipient.

We conclude that the TIER network API should be further developed to provide both the traffic classes that are currently exposed in the API, as well as a new traffic class that offers at most once delivery semantics.

## 5.2 DTN Connectivity State

The core goal of the Delay Tolerant Networking specification is to provide a networking layer that addresses the needs of environments in which traditional Internet assumptions do not hold. Specifically, the model of communication more closely resembles that of a postal service, than a connection-oriented network communications layer. Data messages are sent asynchronously as “bundles”, and are routed through the infrastructure, potentially encountering long delays at hops along the route.

In the course of developing our system, we found that this communication abstraction was insufficient to provide the application features of the system. One feature of the system is that a device has a small cache of the user’s mail. This ensures that when a user is disconnected from the network, the email on the device can be accessed, while the set of messages that do not fit in the cache are stored at the proxy. When the device is within communication range of the proxy, it is desirable that the user can access all of the messages that are stored at the proxy.

To implement this feature, the device clearly needs some notion of whether or not it is “connected” to the proxy. If our implementation were to use standard Internet protocols, this information would be readily accessible from the sockets networking library, as a TCP socket is connection ori-

ented, either side of the connection can determine within a reasonable time bounds when the other side comes into or out of connectivity.

On the other hand, the DTN API does not provide any such connectivity information to the application; essentially the only exposed operations are those to send and receive data bundles (see Table 2). Yet internally, the DTN agent daemon maintains multiple lists of “contacts”, and when it has a bundle to route to or through a contact, it periodically retries to establish the connection to that contact. This periodic retry ensures that when the contact is re-established (i.e. the network connection is available), the agent can deliver the bundle to the contact.

This information of whether or not a contact is reachable is the minimal piece of information that should be exposed to the application through the DTN api. This would allow an application to take different actions depending on whether or not it could expect to reach the peer agent and interact with it.

Exposing the connectivity state also implies that the connectivity test parameters should be exposed and adjustable by the application. The DTN agent maintains internal timers with various periods, used to retry a connection attempt to a lost contact. There are also various timeouts used to determine when there are no more bundles required for an ONDEMAND contact, leading to the contact connection being dropped. For an application to take advantage of knowledge about the connectivity state, the application should be able to adjust these timers to get its desired behavior. Of course, given that multiple applications may be using the DTN infrastructure simultaneously, care must be taken to resolve conflicting settings.

Besides simple reachability, richer semantics could be exposed as well. Bandwidth and/or round trip time estimations could be performed to expose the link characteristics. In addition, certain neighbors in the DTN routing infrastructure can be designated as an `Intermittent - Scheduled Contact`. This means that the DTN agent is informed of periodic schedule that the contact is expected to follow. An example of this type of contact is a Low Earth Orbit satellite connection that is only overhead of the target area for some portion of the day, yet the period and duration

Function	Description
<code>send_bundle_mem(buf, len)</code>	Send a bundle from memory
<code>send_bundle_file(filename)</code>	Send a bundle from a file
<code>bundle_register(endpoint)</code>	Register the application to receive bundles addressed to the given endpoint
<code>bundle_poll()</code>	Poll for new bundle arrivals
<code>bundle_getinfo()</code>	Query the agent for configuration information

Table 2: DTN API v1 (simplified)

of this schedule is well known and predictable. For these types of periodic contacts, the API could expose not only the current connectivity state, but aspects of the schedule, such as the next available delivery time.

### 5.3 TIER Traffic Class Availability

Given that in the TIER platform, an application does not directly interact with the DTN API, but instead uses the traffic class API (see Table 1), the addition of connectivity notification to the DTN must be propagated through the traffic class API as well.

Yet the notion of traffic classes fits very naturally with exposing this information. The TIER API should be augmented with the ability to query which traffic class characteristics can currently be offered by the infrastructure. In other words, the various characteristics of the connectivity status would be converted into the notion of whether or not a particular traffic class is available. In the case of the aforementioned issue in our email application, the availability of a TWOWAY traffic class is a natural abstraction for the notion of whether or not the device can rely on a relatively low latency communication dialog with the proxy.

In general, the information exposed by the DTN api can be arbitrarily matched with the availability of traffic classes. For example, the TIER api could be augmented with a new traffic class, DEADLINE, representing the ability to deliver a certain message by a specified real time deadline. Were the DTN API to expose the periodicity information of a connection, this traffic class could be offered to the application. The notion of whether or not that particular traffic class is available could be determined based on the exposed DTN scheduling information.

One final note regarding the exposed connectivity status draws from our experiences implementing the prototype email delivery system. As mentioned previously, our device needed to know whether or it could reach the proxy; thus we implemented a simple system that periodically tried to establish a TCP connection to a host, then sent short messages back and forth to maintain the connection. This connection monitor is then used by the device to determine whether or not it is “in range” of the proxy. It then uses the TIER/DTN APIs to do the communication.

This implementation is actually a contributing cause to the problem mentioned previously related to a buildup of message query packets. The application’s notion of

connectivity from the connection monitoring system and the DTN’s notion of connectivity from the internal timers maintaining connection lists became out of sync and could remain that way for several seconds. Thus the application expected that it would get a response to its query within a few seconds; the delay of up to ten seconds caused multiple queries to be sent before the proxy received any one of them.

We therefore conclude that given that application features depend on knowing the connectivity status to a peer application, and that determining this information through alternate means leads to inconsistencies between the connectivity probing and the network layer, the notion of connectivity within the DTN agent should be exposed through the DTN and TIER networking APIs.

### 5.4 Caching in TIER API

A key goal of the TIER API is that applications for the developing world will be network-centric, leveraging the shared data center and proxy infrastructure to enable rich application functionality in a highly constrained device. Our email system design follows this paradigm, in that the device uses the proxy as a larger storage source, and the data center uses the proxy as a village area storage service. Through the implementation of the email system, we found that a majority of the protocol messages were devoted to cache coherence of the data between the tiers and that e-mail specific pieces of the application was limited to the communication endpoints. E-mail data within the system could be treated as opaque objects which need to be updated and replicated within the system with some manner of caching semantics.

Our experiences with e-mail motivate the hypothesis that applications written for TIER will have their structure centered around the notion of caching. This follows from the disconnected nature of the architecture, as well as the assumption that all permanent data is stored in the data center. Because there is no end to end connectivity, information to be delivered from the data center to a device must be cached at the local proxy. The limited storage available on the device also means that device memory can only hold a limited view of a user’s data, meaning that the data that cannot fit on the device must be held in the proxy.

In light of these common application requirements, the TIER API should be augmented with mechanisms for

generic cache coherence operations. While we have not worked out the specific details of what the API should look like, based on our experience with the email system, the caching API should offer the following basic services:

#### **Overview:**

Our caching system will consist of data objects called tables which contain elements which either hold data values, or are links to other shared objects in the system. Remote clients can register to have a shared view of a subset of the elements in a table. The TIER caching API will then automatically maintain cache coherency between the owner of the table and clients that are registered with the table.

#### **Create and Destroy table:**

Creating a table involves specifying a schema describing basic type information of the table elements. One important notion is that table elements may contain keys to other tables. For example, a given email message may contain a body and zero or more attachments. Our planned email system design uses a unique identifier space for the larger body regions, enabling a message header to be sent independently of the body regions. Thus an entry into a *message header table* would contain references to one or more entries in the *message body table*. This type of functionality may also be used in an offline web browsing application as part of a hyperlinks implementation.

**Insert, Delete, and Lookup elements:** Clearly, creating a table implies that there is some means to manipulate the table. These operations are local to a single location (i.e. one of data center, proxy, device). The lookup functionality should be able to query on a number of fields in the table definition. This is similar to a relational database system, though would likely be constrained to some number of designated key fields as part of the table creation.

**Register and Unregister replications:** Simply having a table layer does not provide caching semantics. The notion of a *replication* implies that table elements matching some set of criteria should be replicated to a cache version of the table on another host. A registration should be “sticky”, implying that whenever new elements are inserted to a table that match a given registration, then the new element is forwarded to the entity matching the given registration. The replication should also correlate with the rest of the API, as the application should be able to specify the traffic class parameters that table replication messages should follow.

**Register and Unregister data callbacks:** Given that the API layer is responsible for managing the replication of table elements, this layer should expose a hook to the applications so they can be made aware of changes to the table, specifically when new data arrives.

These operations follow naturally from specific operations we developed into the prototype email service; it is apparent that the email system implementation would be much simpler were these operations in the infrastructure. For instance, a “sticky” registration scheme couples

a database insertion with a scheduled transfer. In our current implementation, upon a new message arrival, the data center first inserts the message headers into the appropriate table, does a series of lookups to determine which proxies that should get a copy of the mail headers, then sends the headers on to the registered proxies. It repeats this same process for the mail body data. Building the forwarding functionality as a network service would clearly simplify the implementation of the email server, as an incoming mail message would simply be inserted into the appropriate table, the infrastructure would handle the data transfer, and the proxy would be notified when a new message arrives in its table.

## **6 Future Work**

### **6.1 Full System Implementation**

Our prototype implementation indicated that the TIER infrastructure is effective for developing an email system that has promise to be functional in a developing world environment. We plan to take the lessons learned from this initial implementation, and to implement a more complete implementation of the system, such that this new implementation may be actually deployed in a developing region.

As mentioned previously, we plan to make significant changes to the DTN and TIER infrastructure which should make the email system itself relatively simple to implement.

### **6.2 “Multicast” Delivery Protocol**

Optimizing the delivery of repeating mail message data to a set of users provides the opportunity for significant bandwidth savings. The main source of inspiration for these optimizations stems from the operation of wide area filesystem coherence algorithms in systems such as [5] and [9]. The element that these (and several other) systems have in common is the use of a strong hash to represent data identity more efficiently than the whole data block itself.

We plan to implement an enhancement to our message transfer protocols that takes advantage of this technique to limit redundant transfers of identical mail data.

Specifically, when a message is injected into the mail delivery system, it is parsed into a header region and zero or more body data regions (called *chunks*). This parsing is trivial, as the standard Internet mail protocol already defines a boundary identifier between attachments for sending a multi-attachment message. For each *chunk*, we compute a strong hash of the message contents and use the hash value as the *bodyid*. The list of *bodyid* values for a given message are transmitted along with the message headers, allowing the body data to be sent separately, perhaps in a separate traffic class.

Furthermore, given a hash key, the data center can determine whether at some point in the past it had already

transmitted the given *chunk* to a given proxy. It can then omit re-sending the chunk, relying on the proxy to use the bodyid in the mail header message to resolve the correct chunk of data corresponding to that body id.

In terms of applicability, this feature would clearly be most effective in the case where a large attachment is sent to a number of users in the same village, as the attachment data only needs to be sent once to the proxy in the village. This optimization can also be applied in the opposite direction; if an end-user receives a large attachment and then replies or forwards it back to the data center, the proxy need not send the body data back to the data center since it can assume that the data center already has a copy of the data and can omit sending it.

One final note about this feature is that it is strictly an optimization and cannot affect correctness. This is because we plan to add a new message type that can be used to resolve a failed reference in the case that the optimization was applied too aggressively. For example, in the case where a proxy crashes and loses all mail data, it may subsequently get bodyid references to body data that it no longer has, due to the fact that the data center is not aware of the data loss. In this case, the proxy can detect the unresolvable reference and can send a request back to the data center, which will respond with the missing data. Clearly this is not the desired mode of operation, given that it involves a full extra round trip, and that the cost of a round trip may be a very long time. However, the expectation is that most of the time, the data center can have a good idea of which body regions are likely to be present at a proxy, and can omit sending redundant traffic.

## 7 Conclusions

In this paper we describe the E-mail4B system, a prototype email delivery system targeted at the developing world. The goals of this project were to evaluate the general TIER platform for application development. We implemented a fully functional prototype system that handles intermittent networking operation. Through this implementation, we learned some key lessons regarding the platform, though overall, found that the platform's design paradigms were appropriate to the task of developing our email delivery system.

## 8 Acknowledgments

We would like to thank Eric Brewer for advice and guidance throughout the project. We would also like to thank Rabin Patra and Sergui Nedeveschi for their work developing the Tier API, and Kevin Fall, Robert Durst, and Keith Scott for their work on the DTN implementation.

## 9 References

### References

- [1] Eric A. Brewer. A Scalable Enabling IT Infrastructure for Developing Regions (ICT4B). *NSF Proposal*, 2003.
- [2] V. G. Cerf, S. C. Burleigh, A. J. Hooke, L. Torgerson, R. C. Durst, K. L. Scott, K. Fall, and H. S. Weiss. Delay-tolerant network architecture. Internet Draft, March 2003. <ftp://ftp.rfc-editor.org/in-notes/internet-drafts/draft-irtf-dtnrg-arch%01.txt>.
- [3] Kevin Fall. A delay tolerant network architecture for challenged internets. In *ACM SIGCOMM*, February 2003.
- [4] The Delay Tolerant Networking Research Group. <http://www.dtnrg.org/>.
- [5] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001. <http://citeseer.nj.nec.com/muthitacharoen01lowbandwidth.html>.
- [6] R. K. Patra and S. Nedeveschi. Traffic classes API for ICT. Draft, October 2003. [http://tier.cs.berkeley.edu/docs/traffic\\_classes.pdf](http://tier.cs.berkeley.edu/docs/traffic_classes.pdf).
- [7] R. Patra S. Nedeveschi and S. Kim. Hardware speech recognition for low cost, low power devices. *CS252 Graduate Computer Architecture report*, 2003. <http://www.cs.berkeley.edu/~rkpatra/spring03/cs252/cs252.pdf>.
- [8] Technology and Infrastructure for Emerging Regions (TIER). <http://tier.cs.berkeley.edu/>.
- [9] A. Tridgell and P. MacKerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.