CafNet: A Carry-and-Forward Delay-Tolerant Network

by

Kevin W. Chen

S.B. C.S., M.I.T., 2006

Submitted to the Department of Electrical

Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 2007

Author . . . .
Department of Electrical Engineering and Computer Science
February 2, 2007

Certified by . . . .
Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

CafNet: A Carry-and-Forward Delay-Tolerant Network

by

Kevin W. Chen

Submitted to the
Department of Electrical Engineering and Computer Science

February 2, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a delay-tolerant network stack that allows applications to send messages to other network nodes when no end-to-end connectivity is present. CafNet, the Carry-and-Forward Network, is a delay-tolerant network stack with a CafNet Transport Layer, a CafNet Network Layer, and one or more Mule Adaptation Layers, corresponding to the traditional transport, network, and link layers. Nodes can connect to other nodes through a variety of link mechanisms, and in some cases, the data itself can be physically carried, such as on a USB key or a PDA. CafNet prioritizes messages such that data with a higher priority is sent during short bursts of connectivity. The stack was tested on embedded PC systems used in cars for the CarTel project to determine its performance.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

Acknowledgments

I would like to thank my thesis advisor, Hari Balakrishnan, for the opportunity to work on this project and for the advice and guidance he provided throughout the project.

I am deeply grateful to Vladimir Bychkovsky, who continually met with me, worked with me to get started with the project and understand the problems to be addressed, helped me set deadlines to stay on track, gave suggestions on how to structure parts of my implementation, and taught me the importance of unit testing.

I would like to thank Hongyi Hu and Katherine Lai for their work with me on CafNet as part of a 6.829 Computer Networks project, in which we developed a simplified CafNet prototype using Bluetooth on cell phones, PDAs, and computers.

Thanks also to Sam Madden and Bret Hull for their useful comments on the design of CafNet, Yang Zhang for working with me to integrate CafNet with IceDB as part of CarTel, and Jakob Eriksson for reading and commenting on this thesis and assisting with the Soekris boxes.

This work was supported by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, and by the National Science Foundation under grants CNS-0205445 and CNS-0520032.

Contents

1 Introduction **13**

1.1 Delay-Tolerant Networks 13

1.2 Motivating Application: CarTel 14

1.3 CafNet Requirements 16

1.4 Thesis Contributions and Outline 17

2 CafNet Design **19**

2.1 Endpoint Naming 21

2.2 CafNet Applications 21

2.3 CafNet Transport Layer 22

2.4 CafNet Network Layer 26

2.5 Mule Adaptation Layer 27

3 Implementation **29**

3.1 Messages 29

3.2 Interprocess Communication 31

3.3 Communication Between MALs 32

3.4 CafNet Layers 33

3.4.1 Mule Adaptation Layer 33

3.4.2 CafNet Network Layer 35

3.4.3 CafNet Transport Layer 40

3.5 Combined CafNet Stack 42

3.6 CafNet Transport Layer as a Library 43

3.7 Bluetooth, Java, and Cell Phones 43
3.7.1 JSR-82: Bluetooth in Java 44
3.7.2 J2ME: Java on Cell Phones 44
3.7.3 JSR-75: J2ME Java Filesystem access 45

4 Performance 47

4.1 XML-RPC 47
4.2 Plain Sockets 50

5 Related Work 53

5.1 Addressing 53
5.2 Routing 53
5.3 Social Routing 54
5.4 Mobile Sensor Networks 55
5.5 Transport Layer 56
5.6 Full DTN Design 56

6 Conclusion 59

6.1 Future Work 59
6.1.1 Routing 59
6.1.2 Custodial Transfers 60
6.1.3 Bluetooth 60
6.1.4 Security 62
6.2 Conclusion 62

List of Figures

1-1 CarTel Architecture 15

2-1 CafNet Stack Design 20

2-2 CTL Message Lists and Message Flow 23

2-3 CTL Message Lists and Message Flow (Continued) 24

3-1 MAL Communication Format 32

4-1 CafNet Performance: Varying number of messages 49

4-2 CafNet Performance: Varying message sizes 52

List of Tables

2.1 CafNet application API exposed to the CTL 22

2.2 CTL API exposed to CafNet applications 22

2.3 CTL API exposed to the CNL 25

2.4 CNL API exposed to the CTL 26

2.5 CNL API exposed to each MAL 26

2.6 MAL API exposed to the CNL 27

3.1 CafNet message headers 30

3.2 CafNet stack configuration file directives 42

4.1 Initial CafNet file transfer results on a Pentium 4 48

4.2 Initial CafNet file transfer results on a Soekris box 48

4.3 CafNet performance for sending 1000 messages of different sizes . . . 51

Chapter 1

Introduction

In today's world, people carry around a variety of devices with portable storage — PDAs, USB keys, laptop computers, and more. While some devices can connect to the Internet, others can only communicate with one another. The Carry-and-Forward Network (CafNet) protocol aims to take advantage of the mobility and local networking of these devices and use them to forward messages (data) toward the target destination, passing them off to other intermediary devices as needed.

1.1 Delay-Tolerant Networks

CafNet can be best described as an implementation of a delay-tolerant network (DTN), a concept first discussed in Fall [7]. Unlike conventional networks, DTNs do not necessarily provide end-to-end connectivity between two endpoints, and nodes may experience greater periods of disconnectivity than periods of connectivity. The latency to send a message may be on the order of hours or more, rather than fractions of a second. Sensor networks, for example, may collect data to send to a central server, but given power limitations and remoteness may only be able to infrequently transmit the data. These "challenged networks" can make use of a DTN can be used to send the data when connectivity is available.

A DTN addresses the problems posed by challenged networks by creating an overlay network on top of other protocols. Bluetooth and other local wireless technologies, for example, may be used for local transmissions where Internet access is not present, whereas TCP may be used to transmit data to remote systems when Internet connectivity is available. Although each of these protocols has its own requirements and addressing schemes, the DTN can use link-specific modules to operate on top of each of them.

Each node in the DTN can potentially act as a “data mule,” a device that is physically carried from one location to another [16, 19]. These mules behave like “store-and-forward” nodes on the Internet in some ways, but deliver messages not by immediately forwarding them to a predetermined next-hop, but instead, waiting until they encounter another device and then potentially forwarding the data, hence the name “carry-and-forward networking” (CafNet).

1.2 Motivating Application: CarTel

The CarTel[1] project features a collection of vehicles, each containing an embedded PC collecting data from sensors, providing information such as GPS, Wifi availability, and OBD-II (On-Board Diagnostics) data from the vehicle [2, 10]. This data is then sent to a central server for analysis, allowing users to track and plot their routes, find congestion points, and see traffic patterns at different times of day. When they want to visit an unfamiliar location, they can see the anonymized data of other users to plan routes. The central server can also notify the user of any vehicle problems reported by OBD. Cars can also communicate with each other, perhaps streaming audio to one another, or sharing information about open parking spaces, potential carpool routes, or nearby store or restaurant specials. Figure 1-1 shows this overall CarTel architecture.

The way in which CarTel has transmitted the data to the central server has evolved over time. Very early on, data was stored on a USB key, and the USB key would be hand-carried between cars and offices. Once the USB key was plugged into the office computer, software on the computer would automatically upload the data to

[1]http://cartel.csail.mit.edu/

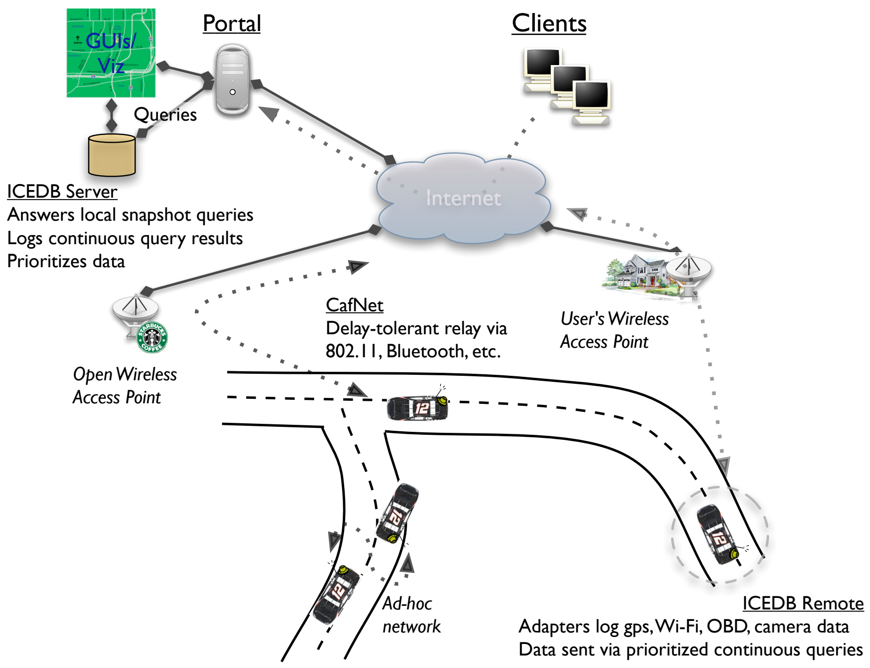

GUIs/ Viz
Portal
Clients
Queries
ICEDB Server
Answers local snapshot queries
Logs continuous query results
Prioritizes data
Internet
CafNet
Delay-tolerant relay via 802.11, Bluetooth, etc.
User's Wireless Access Point
Open Wireless Access Point
Ad-hoc network
ICEDB Remote
Adapters log gps, Wi-Fi, OBD, camera data
Data sent via prioritized continuous queries

Figure 1-1: CarTel Architecture [10]

the server without any further interaction required. At the end of the day, the USB key would be carried back to the car and plugged back in to store more data.

The embedded PC systems in the vehicles support Wifi, and in the next stage they were configured to upload data opportunistically through access points whose owners had chosen to opt-in. Periods of connectivity could be brief, however, as the car might be quickly driving by, and so schemes to quickly scan and associate with access points were developed [2].

In the future, it may be useful to send to PDAs or laptops with Bluetooth. The Bluetooth device sits in the car while moving, and the embedded PC continuously transmits new data to the device. At the destination, the user takes the device to an office, at which point the device retransmits the data over Bluetooth to a computer, which then sends it over the Internet to the central server.

Each of these methods of transmitting data requires the data collection application to implement completely different protocols and neighbor discovery to determine which data to send where. CafNet aims to simplify application development by providing the CafNet applications with a simple interface and implementing robust delivery protocols.. Given increasing storage capacities, mules can carry large amounts of data and therefore act as a high-bandwidth but high-latency channel. For data that can tolerate high latency, mules can be effective data carriers.

1.3 CafNet Requirements

The following requirements drive the design of CafNet:

- CafNet must be able to handle variable-size messages (application data units, ADU) ranging from a few bytes to a few megabytes.

- Messages sent through CafNet must be time-insensitive as they may be subject to long delays. Applications requiring end-to-end delivery acknowledgments must also be able to handle delays in receiving the acknowledgments. Since retransmission may be necessary if an acknowledgment is not received, the application (and not the transport layer) must also retain a copy of the message available for potential retransmission.

- Each message should have an associated priority. Contact time with other mules or Wifi access points can potentially be very short, on the order of seconds, and so not all the data can always be transferred. Higher priority data should be sent quickly during these brief intervals. When a mule has limited space, higher priority data should preempt lower priority data.

- Although some applications may require end-to-end acknowledgments, others may not. CafNet should be able to provide acknowledgments for those applications that require them, but simply provide best-effort delivery for messages from other applications, and drop lower priority messages without retransmission if space is needed.

1.4 Thesis Contributions and Outline

At the start of my research, a preliminary CafNet design had been proposed, but no implementation had yet been started. I wrote the first implementation, correcting flaws in the design as they were discovered. This improved design is discussed in Chapter 2. Next, I integrated this implementation with IceDB, a CarTel component and one of the first CafNet applications, and made some additional changes to the CafNet implementation to make it easier for IceDB to implement the necessary CafNet methods, discussed in Chapter 3. I ensured CafNet worked over a variety of conditions, such as between multiple computers, and over GPRS with unexpected disconnections, and then tested the stack's performance. After using CafNet on the embedded PC used in CarTel revealed performance problems, I replaced the inter-process communication CafNet uses with a different protocol, discussed in Chapter 4. Chapter 5 explores related work, and Chapter 6 looks at future work and a summary of my research.

Chapter 2

CafNet Design

The CafNet stack is divided into three layers: the CafNet Transport Layer (CTL), the CafNet Network Layer (CNL), and the Mule Adaptation Layer (MAL), analogous to the traditional transport, network, and link layers. CafNet applications use application-level framing and schedule and later transmit application data units (ADUs) to the CTL [5]. The CTL passes the message to the CNL, which then buffers the message until it can send the message to an appropriate node. At that point, the CNL passes the message to the appropriate MAL, which sends it over the appropriate network interface.

Many CafNet layers feature the use of callbacks or upcalls [1, 4]. Since messages may not be immediately sent, lower layers perform upcalls to higher layers to inform them when certain actions can be done. The CTL performs application callbacks when it needs to get data, and the CNL performs CTL callbacks when it can buffer more data.

Unlike the traditional link layer, the MAL is also responsible for determining the network neighbors of the current device and notify the CNL of any updates in these neighbors. A Bluetooth MAL might do this by periodically scanning to find nearby Bluetooth devices that are also running the CafNet stack. A USB key could store a list of devices it has been connected to previously, and therefore has a reasonable chance of seeing again. A device connected to the Internet might use a MAL that that uses TCP to communicate with other CafNet stacks over the Internet. Such

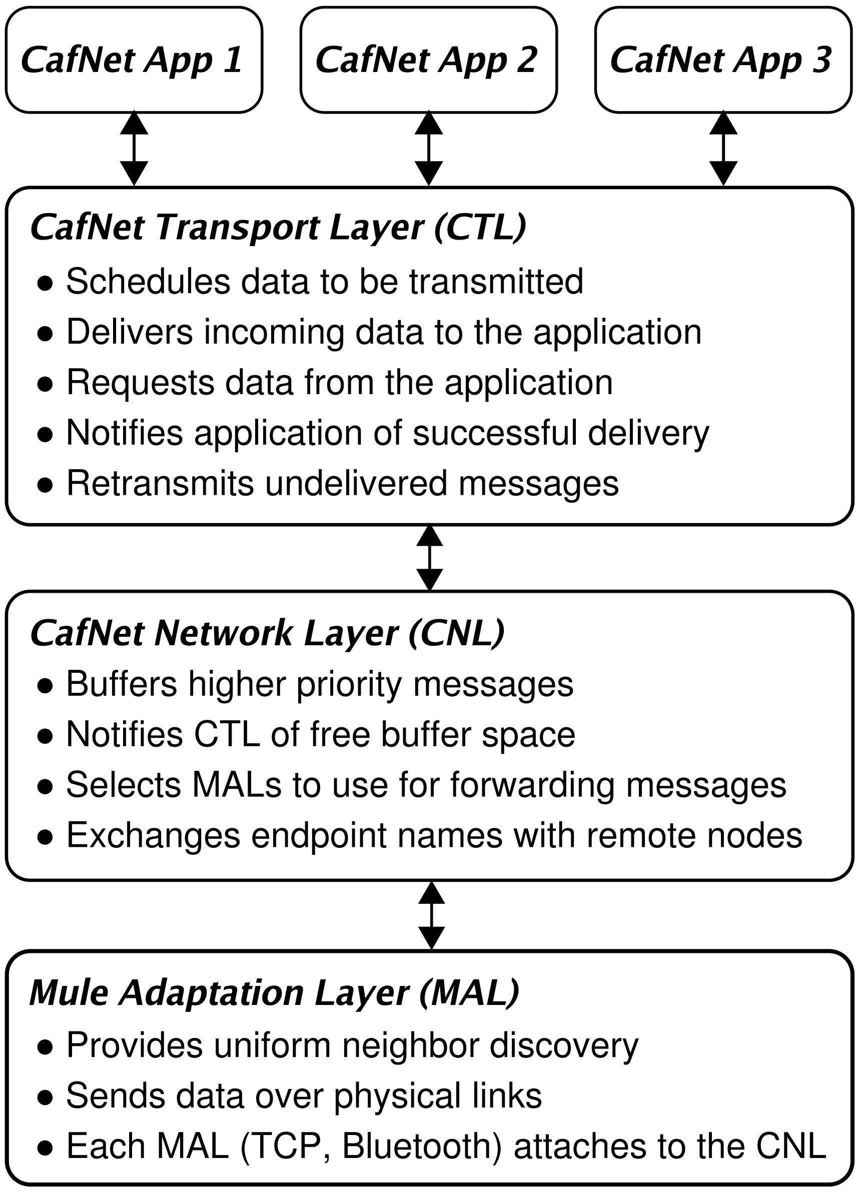

CafNet App 1
CafNet App 2
CafNet App 3
CafNet Transport Layer (CTL)
● Schedules data to be transmitted
● Delivers incoming data to the application
● Requests data from the application
● Notifies application of successful delivery
● Retransmits undelivered messages
CafNet Network Layer (CNL)
● Buffers higher priority messages
● Notifies CTL of free buffer space
● Selects MALs to use for forwarding messages
● Exchanges endpoint names with remote nodes
Mule Adaptation Layer (MAL)
● Provides uniform neighbor discovery
● Sends data over physical links
● Each MAL (TCP, Bluetooth) attaches to the CNL

Figure 2-1: CafNet Stack Design

a MAL might have a list of known neighbors or broadcast its presence to the local network.

The CafNet stack consists of one CTL, one CNL, and one MAL per link mechanism (such as Bluetooth or TCP), as shown in Figure 2-1. An application binds to the CTL, after which it can send messages to the CTL. On the receiving end, the CTL can pass up messages it receives that are addressed to the application.

While traditional networks are stream-oriented, CafNet is message-oriented instead, an architecture better suited for messages with potentially high latency [3, 7]. Applications send messages by breaking them up into smaller chunks on their own if desired and then scheduling the chunks for transmission. The CafNet stack does not buffer the messages at this point — since connectivity may not be present, if the stack were to buffer everything applications wished to send, the buffers would quickly fill up. Because connectivity may only be present for a short amount of time, the stack must reorder messages and send out high priority messages first. To do this, the network layer buffers higher priority messages; the transport layer stores only metadata and then requests additional messages from the applications as space opens up in the network layer.

2.1 Endpoint Naming

The CNL and CTL of each CafNet stack running on a node are globally addressed by a string derived from a public key. For historical reasons described in Section 3.6, the CTL and CNL addresses must be the same. These strings are flat and contain no other semantic meaning [18]. In the future, these keys can be used for authenticating and encrypting messages in transit because they are self-certifying [14, 15].

2.2 CafNet Applications

CafNet applications run in a separate process from the CafNet stack, since multiple CafNet applications can use a single stack. A CafNet application must provide the following callback API so that the CTL can communicate with it:

```
int msg_received(String source, String destination,
      int priority, String ackType, int aduId);
void ack_received(int aduId, String type);
Message get_msg(int aduId);
```

Table 2.1: CafNet application API exposed to the CTL.

When the CTL receives a message for a CafNet application, the CTL calls the application's `msg_received` method, passing on the message, as well as the CNL source address, the CNL destination address, priority, ACK type requested, and the ADU ID. Similarly, when the CTL receives an acknowledgment, it calls the application's `ack_received` method with the message's ADU ID and the type of ACK. Finally, when the CTL is ready to get contents of a message from the application after being signaled by the CNL, it calls the application's `get_msg` method.

2.3 CafNet Transport Layer

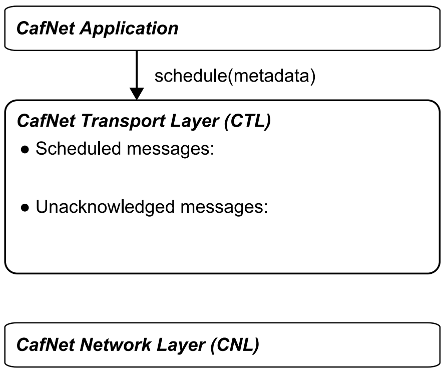
The transport layer serves two purposes; first, it maintains metadata for messages that CafNet applications have scheduled and requests full messages from the applications as necessary. Second, the CTL retransmits messages periodically. The CTL adds the metadata of all messages that are sent with END-TO-END acknowledgments requested to a list of unacknowledged messages. If a message has remained unacknowledged for a period of time, the CTL buffers the message again in the CNL.

The CTL presents the following API to CafNet applications:

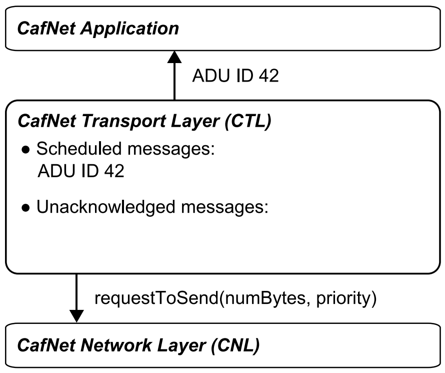
```
int bind(String appName, int port);
int unbind(String appName, int port);
int schedule(String appName, String destination,
     int priority, String ackType, int size);
void cancel(String aduId);
```

Table 2.2: CTL API exposed to CafNet applications.

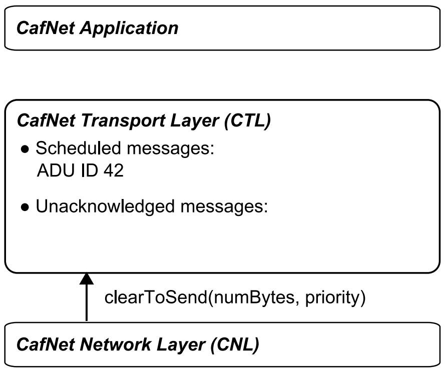
When a CafNet application first starts up, it `bind`s to the CTL, specifying the


CafNet Application
schedule(metadata)
CafNet Transport Layer (CTL)
● Scheduled messages:
● Unacknowledged messages:
CafNet Network Layer (CNL)

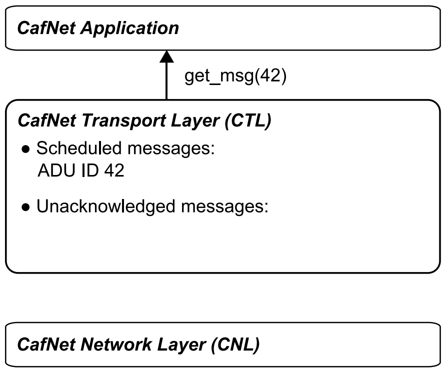
(a) The CafNet application schedules a message by sending metadata only.


CafNet Application
ADU ID 42
CafNet Transport Layer (CTL)
● Scheduled messages:
ADU ID 42
● Unacknowledged messages:
requestToSend(numBytes, priority)
CafNet Network Layer (CNL)

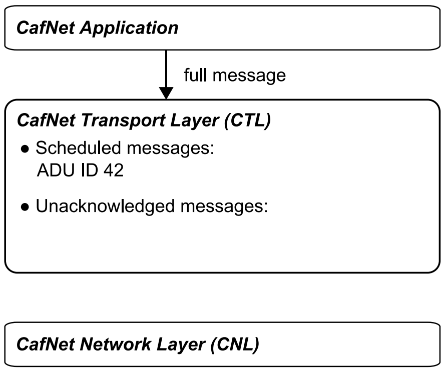
(b) The CTL returns an ADU ID corresponding to the message and requests that the CNL send it.


CafNet Application
CafNet Transport Layer (CTL)
● Scheduled messages:
ADU ID 42
● Unacknowledged messages:
clearToSend(numBytes, priority)
CafNet Network Layer (CNL)

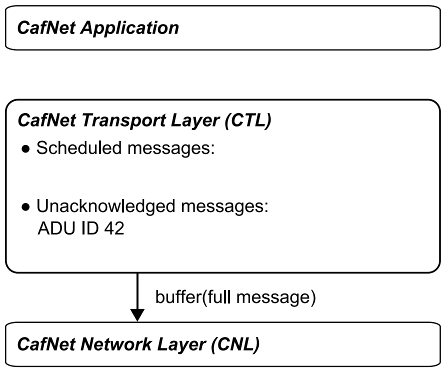
(c) When CNL buffer space frees up, the CNL clears the CTL to send the message.


CafNet Application
get_msg(42)
CafNet Transport Layer (CTL)
● Scheduled messages:
ADU ID 42
● Unacknowledged messages:
CafNet Network Layer (CNL)

(d) The CTL gets full message data from the application.

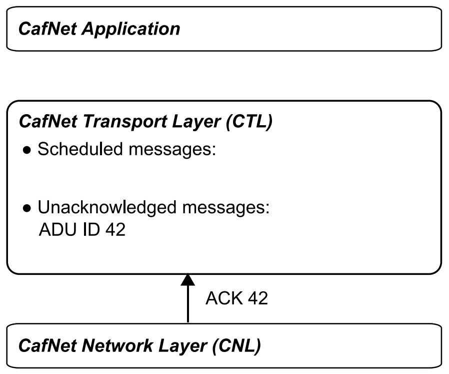

CafNet Application
full message
CafNet Transport Layer (CTL)
● Scheduled messages:
ADU ID 42
● Unacknowledged messages:
CafNet Network Layer (CNL)

(e) The application returns the full message data.

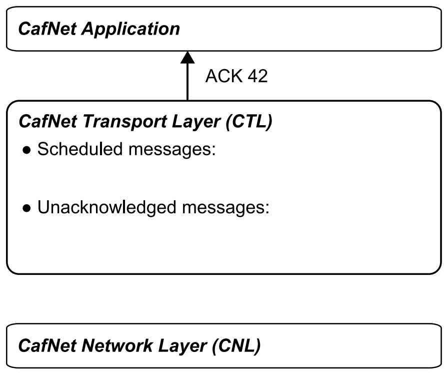

CafNet Application
CafNet Transport Layer (CTL)
● Scheduled messages:
● Unacknowledged messages:
ADU ID 42
buffer(full message)
CafNet Network Layer (CNL)

(f) The CTL buffers the message in the CNL.

Figure 2-2: CTL Message Lists and Message Flow


CafNet Application
CafNet Transport Layer (CTL)
• Scheduled messages:
• Unacknowledged messages:
ADU ID 42
ACK 42
CafNet Network Layer (CNL)

(g) At a later point, CafNet receives an ACK for the message.


CafNet Application
ACK 42
CafNet Transport Layer (CTL)
• Scheduled messages:
• Unacknowledged messages:
CafNet Network Layer (CNL)

(h) The CTL passes the ACK up to the application.

Figure 2-3: CTL Message Lists and Message Flow (Continued)

name of the application and a TCP callback port for the CTL to use to communicate with the application — the TCP callback port is necessary because the CafNet application is a separate process. The application name allows the CTL to differentiate messages destined for different applications, and avoids the limitations of port number used in TCP and UDP, where multiple applications may attempt to use the same number.

After the application has bound to the CTL, the application is ready to send and receive messages. The CTL uses this binding when a message or ACK arrives at the CTL; the CTL checks to see if the correct application is bound, and if so, passes the message or ACK up to the application.

When the application wishes to transmit a message, it calls the `schedule` method with the name of the application, the destination endpoint name, the priority of the message (a higher number indicates a higher priority), the type of ACK desired (END-TO-END or NONE), and the number of bytes of data; the CTL then returns an ADU ID number. The application associates the message with this ADU ID; when the CTL later calls the application's `get_msg` with this ID, the application returns it.

The CTL keeps a collection of message metadata that all bound applications have scheduled. The CTL calls the CNL's `requestToSend` with the number of bytes and the priority of the highest priority message. At any point, if an application schedules a message with higher priority, the CTL then requests to send that message by calling the CNL's `requestToSend` method.

If the application wishes to cancel a message, it calls `cancel` with the ADU ID of the message. If the message has either not yet been buffered, or has been buffered but not yet been acknowledged, the message's metadata is removed from the CTL. The CTL also calls the CNL's `cancel` method to remove it from the CNL buffer, if it is present there. If the message has already been forwarded, `cancel` has no effect in the CNL.

The CTL presents the following API to the CNL:

```
void clearToSend(int bytes, int priority);
void bufferResult(boolean result, Message msg);
void msgReceived(Message msg);
```

Table 2.3: CTL API exposed to the CNL.

When the CNL is ready to get a message from the CTL, the CNL calls the CTL's `clearToSend` method with the number of bytes and the priority corresponding to the previous `requestToSend`. At this point, the CTL adds the destination to the message's headers and calls the CNL's `buffer` method to add the message to the CNL's buffer. The CNL then asynchronously calls the CTL's `bufferResult` informing the CTL whether buffering the message succeeded.

When the CNL receives a message, it passes it to the CTL by calling the CTL's `msgReceived`. The CTL then passes it up to the application by calling the application's `msg_received` method.

Finally, when the CTL successfully buffers a message in the CNL, it adds the message to a list of unacknowledged messages. Periodically, the CTL attempts to re-buffer any messages that are still unacknowledged.

Figures 2-2 and 2-3 show the lists of messages the CTL maintains and the message flow between the CafNet layers.

2.4 CafNet Network Layer

The buffer that the CNL maintains allows the CNL to send messages quickly when network connectivity is established. The CNL also identifies the endpoint names of remote nodes and forwards messages to appropriate nodes.

The CNL presents the following API to the CTL:

```
void buffer(TransportLayer ctl, Message msg);
void cancel(String msgId);
void requestToSend(TransportLayer ctl,
     int bytesToSend, int priority);
```

Table 2.4: CNL API exposed to the CTL.

When the CTL calls `requestToSend`, the CNL checks to see if there is sufficient space in the buffer to add the message to the buffer. When there is enough space, it calls the CTL's `clearToSend` method.

When the CTL calls `buffer`, it generates a unique Message-ID for the message. The CNL checks to see if a message with the same Message-ID already exists in the CNL buffer. If so, the new message replaces the old one. Otherwise, the CNL checks to see if there is space in the buffer, removing lower priority messages if necessary. The CNL then calls the CTL's `bufferResult` informing the CTL whether buffering the message was successful. Section 3.4.2 explains the details of the CNL's buffer management.

Finally, if the CTL calls the `cancel` method, the message corresponding to the specified Message-ID is removed from the cache, if it is present.

The CNL presents the following API to each MAL:

```
void neighborUpdate(MuleAdaptationLayer mal, Collection neighbors);
void msgReceived(MuleAdaptationLayer mal, Message msg);
void sendResult(boolean success, Message msg);
```

Table 2.5: CNL API exposed to each MAL.

When each MAL starts up, and when the neighbors for the MAL change, the MAL calls the CNL's `neighborUpdate` with a collection of neighbors. This collection only contains information about the presence of neighbors, and does not contain endpoint names. Upon seeing previously unseen neighbors, the CNL will send a `CNL-Self-Identify` message to each new neighbor, informing the neighbor what the CNL's endpoint name is. This allows the CNL to forward messages appropriately.

When a MAL receives a message, it simply passes it up to the CNL by calling the CNL's `msgReceived` method. Since `CNL-Self-Identify` messages are passed up by the MAL just as other messages are, the CNL's `msgReceived` needs to sort them out and save the CNL addresses it received. For all other messages, the CNL just passes them up to the CTL.

After a CNL asks a MAL to `send` a message, the MAL will asynchronously call the CNL's `sendResult` method to notify the CNL whether the send was successful. If the send was successful, the CNL can remove the message from its cache. Otherwise, the CNL can try again later. This asynchronous callback is used to avoid having the CNL blocked, and is discussed more in section 3.4.2.

2.5 Mule Adaptation Layer

Each Mule Adaptation Layer communicates with other MALs to transmit messages. Given that different link mechanisms can work very differently, the overall design of each MAL can be very different. Nevertheless, they present a common interface to the CNL, which does not differentiate between MALs:

`void send(String dest, Message msg);`

Table 2.6: MAL API exposed to the CNL.

When a CNL wants to send a message, it simply calls the MAL's `send` method. The MAL then copies the message to the relevant output stream. If this successfully happens, the MAL calls the CNL's `sendResult` to indicate that the send was successful. Note that it is possible for `sendResult` to indicate a success but not have the message actually send to the remote MAL, since it just means that the message was added to the link mechanism's buffer. If the connectivity that was present disappears, but the buffer has not yet filled up, the `sendResult` will still be successful. In this case, the message will be removed from the CNL buffer, and the CTL will eventually retransmit the message.

Chapter 3

Implementation

When CafNet was first implemented, it was designed to not only work over TCP connections, but also over Bluetooth, including on cell phones. Nokia cell phones support Java, Python, and C++. Both Java and Python easily support Bluetooth on the cell phone — I chose Java for the first CafNet implementation given my familiarity with it.

3.1 Messages

The CafNet stack stores messages internally as a byte array of payload data, and a hash of headers, mapping header names to their values. Table 3.1 describes the headers that the stack uses.

Although all headers are stored in memory, the data can be stored either in memory or on disk. Devices with limited memory, for example, may choose to store messages on disk so they can use a larger CNL cache. This implementation of the CafNet stack uses messages with data stored in memory, but can easily be modified to use messages with data stored on disk.

Header	**Description**
Content-Length	Number of bytes in the payload.
MAL-Message-Type	Differentiates between data and MAL-internal messages: Valid values are DATA, FIN, and FINACK.
CNL-Source	The address of the CNL sending the message.
CNL-Destination	The address of the CNL receiving the message; value set to `CNL-Self-Identify` when a remote CNL is identifying itself for the first time.
CTL-Source	The address of the source CTL.
CTL-Destination	The address of the destination CTL.
CTL-ACK-Requested	The kind of ACK requested upon receipt of a message; valid values are END-TO-END and NONE.
CTL-Message-Type	Differentiates between data and CTL-internal messages: valid values are DATA and END-TO-END.
Priority	The priority of the message, expressed as an integer. Higher numbers represent higher priorities.
Application	The name of the destination application.
App-Id	An integer used by the application and CTL to uniquely identify each message within a CTL.
Message-ID	A string used by the CTL and CNL to identify a message in the CNL buffer.
Operation	Application method being called by the CTL.
Other headers	Parameters for application methods being called; dependent on the operation header.

Table 3.1: CafNet message headers

3.2 Interprocess Communication

Although the three layers of the CafNet stack are executed within a single process, CafNet applications run in other processes, and they need to communicate with the CTL. Initially, we used XML-RPC to perform this communication, as implementations are available for all common languages, so applications could easily implement the needed methods, and then simply add XML-RPC.

While this worked well on desktop and laptop computers (such as a Pentium M, 1.7 GHz machine), XML-RPC performed abysmally on the Soekris[1] 266 MHz embedded PC used in CarTel, in some cases transferring data at 9 kilobytes per second locally. Even though the data transferred quickly from the source CTL to the destination CTL, the messages got queued up between the destination CTL and the destination application. More details are available in section 4.1.

CafNet now instead communicates with applications using TCP sockets with its own binary protocol. The CTL and all CafNet applications each listen on a port for incoming requests. When a new TCP connection is established, the CTL or CafNet application reads from the connection, executes the requested method, and then returns the result (if any). As in XML-RPC, each TCP connection is used for only one method call[2].

This protocol uses 32-bit signed integers in big endian (most significant byte first; also network byte order), the format used by Java's DataOutputStream. Strings are represented by an integer indicating how many bytes are in the string, followed by that number of bytes. Message payloads are transmitted in the same way as strings.

When performing a method call, the string containing the name of the method to be called (`msg_received`, `ack_received`, or `get_msg` when the CTL communicates with the application, or `bind`, `unbind`, `schedule`, or `cancel` when the application communicates with the CTL) is written to the socket first. Then, each parameter is written out in the order specified by the method's signature. The result, if any, is then returned in the same format.

[1]`http://www.soekris.com/`

[2]Some XML-RPC implementations support reusing connections, though not all do.

3.3 Communication Between MALs

Communication between MALs, on the other hand, uses a text-based protocol, very similar to HTTP. Each MAL's link mechanism (the Bluetooth Serial Port Protocol or TCP, for example) is assumed to provide reliability and ordering. Each message starts with several lines of headers in the format `Name: Value`. One required header, `Content-Length`, indicates how many binary bytes of message payload follow. The end of the headers is signified with a newline, and then any binary data follows. It is possible for a message to have 0 bytes of data, as occurs with control messages. The other required header is the `MAL-Message-Type` header, which can be one of `DATA`, `FIN`, or `FINACK`. `FIN` and `FINACK` messages always have a `Content-Length` of 0.

```
Message-ID: client-server-ZPerformanceTest-741795
Application: ZPerformanceTest
CNL-Source: client
Content-Length: 15
MAL-Message-Type: DATA
Priority: 1
MAL-Interface: net1
CTL-Message-Type: DATA
CTL-Source: client
CNL-Destination: server
CTL-ACK-Requested: NONE
App-Id: 741795
CTL-Destination: server

This is a test.
```

Figure 3-1: MAL Communication Format

Communication between MALs with established connections is bidirectional — after a connection is made between two MALs, regardless of which MAL connected to the other, each MAL can send as many `DATA` messages as it can while the connection is established. When one MAL has finished sending its messages, it will send a `FIN` message. Once the other MAL has received the `FIN` message, it will respond with a `FINACK` message. Once a MAL has received both a `FIN` message (indicating that the remote MAL has finished sending messages) and a `FINACK` message (indicating that the remote MAL has received all the messages that this MAL has to send), the MAL will close the connection.

Not all MALs use `FIN` and `FINACK` messages. The Bluetooth MAL uses them as it needs to constantly scan for new devices and connect to them (since it can only connect to one other Bluetooth device at a time), and so it cannot continue to maintain a connection to a particular Bluetooth device. The TCP MAL, on the other hand, maintains connections with all known neighbors until the TCP connection is broken by the physical connectivity disappearing, and so never sends any `FIN` or `FINACK` messages.

3.4 CafNet Layers

The CTL, CNL, and MALs each run in their own threads so that each layer can determine how to prioritize its own requests. This allows the stack to run more efficiently than processing events from all layers sequentially, without having to implement a complex system of prioritizing events. When other layers make requests, in most cases, the requests are not directly executed — instead, they are added to a queue. Each layer has an event loop that processes items in the queue, prioritizing events from lower layers over events from higher layers, but executing events from each layer sequentially. This removes the need to address many potential concurrency issues that would result if other layers directly executed the methods they called.

3.4.1 Mule Adaptation Layer

In addition to acting as a traditional link layer, each MAL also informs the CNL of neighbors, that is, other CafNet stacks that this stack can communicate with. The MAL assigns each neighbor a unique string identifier (similar to Linux network interfaces, like `eth0` or `wlan1`), and then provides the CNL with these identifiers.

This CafNet implementation features two different kinds of MALs: a Bluetooth MAL and a TCP MAL. Given the properties of each protocol, the MALs are quite different.

Bluetooth MAL

The Bluetooth MAL runs two threads: a server thread, and a scanning thread; unlike the TCP MAL and the other CafNet layers, it does not have an event loop thread. The server thread advertises a Bluetooth Serial Port Protocol service with a UUID unique to CafNet (71f711bfb6214ac8a592ad39250ba141) so that other Bluetooth MALs can discover this Bluetooth MAL. The scanning thread scans for other MALs advertising the CafNet UUID, and rescans every 3-7 minutes. It is not possible to continuously scan, as scanning and sending data cannot happen simultaneously; Bluetooth also uses the same 2.4 GHz frequency spectrum that Wifi does, so continuously scanning would be disruptive to other applications using Wifi on the same system and other nearby Wifi devices.

When the CNL asks the Bluetooth MAL to send a message, the MAL connects to the remote MAL using the Bluetooth Serial Port Protocol. Once connected, the MAL starts a thread to process any incoming messages, and then sends the requested message. When the message has been sent, the MAL sends a FIN message to notify the remote MAL that it has finished sending messages; the remote MAL should return a FINACK message acknowledging the FIN message. Once the remote MAL also sends a FIN, indicating that it has sent all the messages it needs to send, the MAL closes the connection.

Although not very efficient, the Bluetooth MAL currently only sends one message per connection for simplicity, and because without additional setup, Bluetooth only allows a device to connect to one other device at a time. Section 6.1.3 discusses the possibility of connecting to multiple devices at a time, and how that can be accomplished.

The Bluetooth MAL was also designed prior to `CNL-Self-Identify` messages being developed, and so is not yet integrated into the new design. Since the CNL does not send any messages on an interface without having previously received a `CNL-Self-Identify` message, but the Bluetooth MAL does not associate incoming messages with a particular Bluetooth device, the CNL will not send any messages to the Bluetooth MAL under the current design. Section 6.1.3 also discusses potential ways of fully integrating the Bluetooth MAL into the CafNet stack.

TCP MAL

In addition to the event loop thread, the TCP MAL has two primary threads: a server thread, and a trigger thread. The server thread listens for incoming TCP connections from other MALs. The trigger thread listens on a local TCP port for connections from an external program notifying it of the presence or absence of external connectivity.

The external program sends a single line with the text `IP=ON` or `IP=OFF` indicating changes in connectivity. Like other events, the trigger events get added to a collection of pending events; unlike other events, however, trigger events get executed before any other pending events. If both `IP=ON` and `IP=OFF` events are pending, they are executed in the order they are requested. If multiple `IP=ON` events are pending, they are coalesced into one event and the earlier ones are ignored; the same is true for multiple `IP=OFF` events.

When an `IP=ON` event is processed, the TCP MAL attempts to connect to all known peers, as listed in the CafNet configuration file. After each connection is made, a new thread is started for sending messages over the connection, the MAL updates the CNL with the new collection of neighbors, and then another new thread is started for receiving messages over the collection. The ordering of these events, as well as the creation of these extra sending and receiving threads, is important, and will be discussed later in section 3.4.2.

When an `IP=OFF` event is processed, the MAL closes all open connections, and then updates the CNL with an empty list of neighbors.

3.4.2 CafNet Network Layer

Forwarding

The CNL forwards messages to each of the MAL neighbors that it knows about. To do this forwarding, the CNL must know the CNL addresses corresponding to each of the MAL neighbors. Whenever the CNL learns about a previously unknown neighbor, the CNL sends a message with no payload to the new neighbor. To differentiate this address announcement from a normal message, the CNL sets the destination CNL address to `CNL-Self-Identify` — this is also done since the CNL does not know the remote CNL's address and cannot address the message properly. When the remote CNL learns of this CNL, it too will send a `CNL-Self-Identify` message,

The CNL associates the remote CNL addresses with the MAL identifiers. Messages buffered in the CNL that have destination CNL addresses corresponding to one of the neighbors are then sent to those neighbors. Otherwise, the messages are held in the CNL buffer until a neighbor with the specified CNL address is seen. At this time, messages can only be forwarded directly to the destination, and so forwarding is not supported. Section 6.1.1 explains some of the options available for future work.

Message Priority

The CNL uses a priority queue for its buffer. Messages are sorted by their priority — messages with higher priority are sent before messages with lower priority. Messages with lower priority may also be preempted from the buffer by messages with higher priority if insufficient space is available.

One potential problem is that it is possible that an application could send many messages and fill the CNL buffer. If the messages are low priority, other messages can be buffered and preempt the messages with unreachable destinations. If the messages are high priority, however, all outgoing communication may become blocked, although incoming messages would still be processed normally.

Message Flow

The CTL can have up to one `requestToSend` request at a time. If a request is still pending and the CTL calls `requestToSend` (perhaps with higher priority data), the CNL replaces the previous request with the new request. After the request is processed, the CNL checks to see if there is space available in the CNL buffer.

The CNL determines if space is available for a pending message first by seeing if the remaining space in the buffer plus the sizes of all buffered messages of lower priority is greater than the amount of space requested. If so, the CNL calls the CTL's `clearToSend` method with the number of bytes and priority requested. Otherwise, the CNL will try again later.

When a message in the buffer has a CNL destination matching the remote CNL address of any of the neighbors, the CNL calls the appropriate MAL's `send`, specifying the message and the MAL identifier to use to send the message. The MAL then asynchronously calls the CNL's `sendResult` method to indicate whether the `send` was successful; if successful, the message is then removed from the CNL's buffer.

MAL Blocking and Disconnected Connections

Originally, the MAL's `send` returned whether sending the requested message was successful, rather than asynchronously calling the CNL's `sendResult`. This caused the CNL to block while each message was being sent. While this approach worked reasonably for short messages on fast TCP connections, it would not work very well for larger messages on lower-bandwidth links, such as the Bluetooth MAL — the CNL would be unable to send messages out on other links simultaneously, possibly missing small windows of time in which it can send data on a higher-bandwidth connection.

More importantly, it is possible for the TCP MAL to think its connections are still connected when in fact they are not. If the physical link goes down while the interface remains up, for example, connection reset signals are not sent, and the TCP MAL does not realize the connections are broken until they time out a few minutes later. In such cases, if `send` requests were performed synchronously, connections breaking in these instances will block the MAL for several minutes. By having separate sending threads for each remote MAL in the TCP MAL, one connection blocking also does not prevent messages from being sent to other destinations.

In some cases, a MAL may attempt to connect to a remote MAL after a network disconnection, while the remote MAL still thinks a previous connection is open. The MAL processes the new connection in the same way as any other new connection, assigns it a MAL identifier, and then informs the CNL of a new neighbor. When the

CNL informs the remote CNL of its CNL address, the remote CNL will see that the same CNL is connected twice, and it will close the previous MAL connection. This prevents the CNL from attempting to send messages over to the old, no longer usable, connection.

An Attempt at Non-Blocking I/O

Initially, instead of using an asynchronous `send` and having separate sending threads for each remote MAL, CafNet was going to use non-blocking I/O. The CNL would attempt to `send` a message, and if the `send` failed because it would block, the CNL would try again after 5 seconds, up to three times. In the meantime, the CNL would attempt to send messages to other destinations. If the `send` would still block after 15 seconds, the CNL would close that particular connection, assuming that the connection was broken.

After attempting to implement this design, I encountered several problems:

- It is not possible to have blocking reads and non-blocking writes on the same socket so modifying the previously existing blocking reads complicated the situation. The workaround I used was to use non-blocking sockets and write a wrapper class to make the non-blocking reads blocking.

- In contrast to C/C++, non-blocking writes in Java are not all or nothing — instead, writes write what they can. Therefore, under the non-blocking scheme, the CNL would have to keep track of how many bytes of each message that it has sent, and then when it retries, only send the portion that has not yet been sent. Even if writes were all or nothing, some messages may be larger than the socket buffer, and would have to be sent in chunks, leading to the same problem.

- It is unclear when the CNL should retry sending the remainder of a message if only part of was sent. The socket may be functioning normally and not be stuck, but cannot immediately send additional bytes.

- Partial writes mean that the CNL (for `CNL-Self-Identify` messages) and the

MAL cannot send messages on their own without their messages going through the main message queue, because the CNL might have only sent a fraction of a different message to the same MAL, and that message needs to be completed first before sending another message.

- If `CNL-Self-Identify` messages were to go through the main CNL message queue rather than be passed to the MAL directly, they would need a CNL destination header so the CNL can pass them to the correct MAL. However, the remote CNL address may not yet be known.

Event Loop

The event loop for the CNL is a bit more complicated than the loops for the CTL and the MAL. In those layers, the event loops simply checks the queue for pending events; if any exist, it executes the first event and removes it from the queue. In addition to processing queued events, however, the CNL must also `clearToSend` any pending messages (by running an internal `clearPending` method) and send any buffered messages that can now be sent (by running an internal `sendAll` method).

Most events (all but non-`CNL-Self-Identify` `msgReceived` calls) require the CNL to perform these actions after the event is executed since these events change the state of the CNL. When this happens, the CNL will call both `clearPending` and `sendAll` as long as either results in something happening — `clearPending` has the potential of adding a message to the buffer that can be sent, and `sendAll` may have opened up space to `clearToSend` another message.

The `sendResult` call is an exception — it does not directly cause the CNL to call `clearPending` and `sendAll`. Instead, it sets a flag that tells the CNL to do so, which it checks after each action. Since many messages may be sent every time `sendAll` is called, it does not make sense to call `clearPending` and `sendAll` directly after every message is sent — instead, other actions are allowed to run and the `clearPending` and `sendAll` calls from multiple `sendResult` actions finishing are coalesced.

3.4.3 CafNet Transport Layer

The CTL keeps track of message metadata that the application has scheduled, and calls the CNL's `requestToSend` with the metadata with the highest priority.

Other than the main event loop, the CTL has one thread used for processing requests from applications. This separate thread allows the CTL to prioritize actions without having to decide when to read from the application, especially when a high number of application requests are pending. Applications communicate with the CTL as described in Section 3.2. This thread parses requests, calls the appropriate method, and then returns the result. To avoid complicating the API for the application, most of these methods (`bind`, `unbind`, and `schedule`) bypass the event loop and instead return results synchronously. `cancel` does not return anything, and is processed by the event loop normally.

Message Flow

When the application wants to schedule a message, it calls the CTL's `schedule` method with the metadata for the message. The CTL then adds this metadata to a list of pending metadata. If the priority of the message scheduled is greater than the priority of the priority that the CTL has requested the CNL to send, the CTL replaces its previous `requestToSend` with a new request at higher priority.

The `schedule` call returns an application ID for the application to associate with the message. This ID is an integer and is generated simply by adding 1 to the previous ID. Application IDs are unique within each CTL (until they get reused because of overflow).

At some later point, the CNL then calls the CTL's `clearToSend` to indicate that it can buffer a message with the specified priority and number of bytes. The CTL then calls the application's `get_message` method with the appropriate application ID, and then asks the CNL to `buffer` the returned message.

Right before message is passed to the CNL, though, if the application requested an END-TO-END ACK, the CTL adds a `CTL-ACK-Requested` header with value

END-TO-END to the message and adds the metadata to a list of unacknowledged messages. When the destination CTL receives the message, it will pass the message to the application. If the application indicates that it successfully received the message, the CTL will then send a message with a `CTL-ACK-Hash`[3] header whose value is the `App-Id` of the original message, indicating which message is being acknowledged. Upon receiving the END-TO-END ACK, the origin CTL will then remove the metadata from the list of unacknowledged messages.

The ACK is sent directly with the CTL calling the CNL's `buffer` method. Because the CNL's size is based on the total message payload size, and ACKs have no message payload, ACKs can always be buffered in the CNL even if it is otherwise full. ACKs are sent with the same priority as the original message.

Every few minutes, the CTL will then go through the list of unacknowledged metadata, get the messages from the application, and ask the CNL to buffer those messages. The messages are not passed through the normal `requestToSend` and `clearToSend` process, as some of these messages may still be in the CNL buffer, and `requestToSend` would have checked to see whether additional space is available; rebuffering messages that may happen to still reside in the CNL will replace the previous copy of the message with the new (likely identical) message, and not take any additional space. On the other hand, if the message no longer resides in the CNL buffer because it has been successfully sent, but the END-TO-END ACK has not yet been processed, the CNL will add the message to its buffer, removing lower priority messages if necessary. If necessary space cannot be made, the message is not buffered.

If the application wants to cancel transmission of a message, it can do so by calling the CTL's `cancel` method with the corresponding application ID. The CTL removes the metadata from its pending and/or unacknowledged message lists, and then calls the CNL's `cancel` method. If the message exists in the CNL's message buffer, the CNL removes it. No attempt is made to bring back a message that has already been sent and no longer in the CNL cache.

[3]This name is historical; previously, hashes of message data were used instead of an `App-Id`.

Directive	Options	Details
verbosity	verbosity	Controls the output verbosity. 0 indicates errors only, while 100 indicates almost everything.
NetworkLayer	cnlAddress [,cacheSize]	Starts a CNL with the specified CNL address. Takes an optional integer argument specifying the size of the CNL buffer in bytes.
TCPMAL	listenPort, triggerPort, peers	Starts a TCP MAL. Listens for connections from other TCP MALs on port *listenPort*. Starts a trigger server to listens for connectivity updates on port *connectivityTriggerPort*. Connects to other TCP MALs listed in the space separated list of peers, specified as *hostname:port*.
BluetoothMAL		Starts a Bluetooth MAL. Takes no options.
TransportLayer	ctlAddress, appPort [,resendRate]	Starts a CTL with CTL address *ctlAddress*. Listens on TCP port *appPort* for incoming requests from applications. Resends unacknowledged messages every *resendRate* seconds. If *resendRate* is not specified, the CTL defaults to a rate of 60 seconds.

Table 3.2: CafNet stack configuration file directives.

3.5 Combined CafNet Stack

The CafNet stack is started by running a Java JAR file, which reads in a plain text configuration file to configure one or more CafNet stacks. Each line starts with the name of a directive (either a CafNet layer or overall stack option), followed an equals sign and the options, if any, for that layer. Options are separated by commas. Table 3.2 describes these options.

For any particular CafNet stack, the CNL must be listed before any other layers. MALs and CTLs listed after the CNL are then associated with that CNL.

For historical reasons (see Section 3.6), CTLs and CNLs have separate addresses. These addresses must be the same for the stack to work correctly.

3.6 CafNet Transport Layer as a Library

The original CafNet design featured CTLs as a library that applications would link against, rather than a layer directly part of the stack. Each CTL would have a different CTL address, and CTLs would bind to the CNL.

This design evolved into the current implementation, where applications instead bind to a CTL, and only one CTL is associated with each CNL. The current code partially supports both multiple CTLs per CNL and multiple applications per CTL. However, the CNL assumes a `CNL-Destination` that is the same as the `CTL-Destination`, so it is not possible to actually send messages to a CTL whose address is different from the address of the CNL it is associated with.

Because multiple CTLs, each potentially generating conflicting application IDs, would bind to the same CNL, it was necessary to generate Message-IDs for each message to differentiate messages in the CNL buffer (so that the appropriate message would be canceled if a CTL requested to cancel a message). The Message-ID is generated by concatenating the origin CTL address, the destination CTL address, the name of the application, and the application ID. With only one CTL per CafNet stack now, however, the distinction between application ID and `Message-ID` is no longer necessary.

3.7 Bluetooth, Java, and Cell Phones

Java is very commonly supported on both personal computers and mobile devices, one of the reasons CafNet was implemented in Java. Personal computers generally run J2SE (Java 2 Standard Edition), while many portable devices, including Nokia cell phones, run a more limited edition, J2ME (Java 2 Micro Edition). J2ME implements MIDP (Mobile Information Device Profile), which defines the Java environment for mobile devices. Given the limitations of mobile devices, MIDP is limited to provide compatibility to a wide range of devices that may use it. This limitation, however, made implementing CafNet on a Nokia cell phone infeasible.

3.7.1 JSR-82: Bluetooth in Java

J2SE does not have support for Bluetooth, so a library that provides APIs for using Bluetooth on normal computers is required. The JSR-82 specification[4] defines these APIs, but does not provide an implementation. Some of these implementations are OS-dependent, though there are implementations available for Windows, Linux, and MacOS from different vendors. CafNet uses AvetanaBluetooth, available for Linux from `http://sourceforge.net/projects/avetanabt/`.

3.7.2 J2ME: Java on Cell Phones

Because the phone has limited memory, it is not possible to store very many messages (< 1 MB) in memory. Instead, one could potentially write the messages to the phone's filesystem. In attempt to provide device independence, however, J2ME by itself does not allow access to the phone's filesystem. Instead, it allows long-term storage by providing a simple database system called a record store to write data to memory.

Unfortunately, it is not possible to append to a record store in the database system, and so records can only be created entirely at once — that is, from a message in memory. Since some messages are larger than the maximum amount of memory Java can use, however, it is not possible to use the record store without breaking messages up into separate records, introducing undesired complexity.

Another issue with the record store is that it is only accessible by Java applications. While the computer version of CafNet communicates with CafNet applications with sockets, MIDP 1.0 does not support sockets, and so on the cell phone, CafNet must communicate with applications. Only Java applications can read from the record store, however, and so the record store cannot be used for CafNet applications in general.

[4]http://www.jcp.org/en/jsr/detail?id=82

3.7.3 JSR-75: J2ME Java Filesystem access

One alternative to the record system is to use the FileConnection (JSR-75) package[5], which is an optional API for J2ME that allows reading and writing from the device's filesystem. The Nokia 7610, however, does not support this API. The Nokia 6682, which I later obtained, does support this API, but at that point, I started focusing on getting CafNet to work under J2SE instead. JSR-75, however, would allow the phone to hold messages larger than the available memory in the device, since it does not have to allocate memory to hold the entire message at once.

[5]http://www.jcp.org/en/jsr/detail?id=75

Chapter 4

Performance

After implementing the CafNet stack, I conducted performance testing on several machines, including the Soekris 266 MHz embedded PC used in CarTel, to determine what throughput the stack could deliver. Initial testing revealed that the XML-RPC protocol, used for interprocess communication between the CTL and CafNet applications, reduced the potential performance of the stack when running on the Soekris box, and so I replaced XML-RPC with a custom binary protocol over plain TCP sockets.

4.1 XML-RPC

As mentioned in section 3.2, the initial design of the CafNet stack used XML-RPC to communicate between the CTL and CafNet applications. Once the CafNet stack was fully implemented with the XML-RPC communication, `cncp`, a Python program designed to use CafNet to transfer files, was used to test transferring files between two CafNet stacks on the same host.

`cncp` divides files that it sends into smaller blocks of data and sends each chunk as a message. The initial tests varied the size of the chunks to test what the overhead of message headers for each chunk and message processing in the CafNet stack was. Transferring an approximately 50 MB file on a Pentium 4, 2.4 GHz machine with a CNL cache of 1 MB yielded the following results:

Block Size	**Time**	**Transfer Rate**
1 MB	73 seconds	662 kilobytes per second
100 KB	85 seconds	569 kilobytes per second
10 KB	203 seconds	238 kilobytes per second

Table 4.1: Initial CafNet file transfer results on a Pentium 4

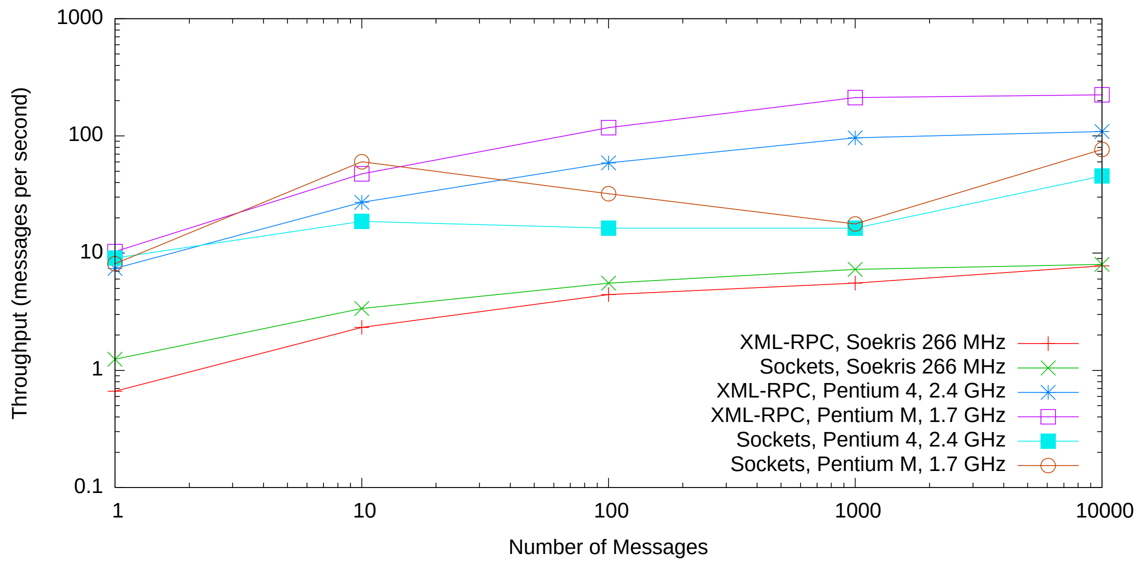
While these transfer rates seem reasonable, they become much worse on the 266 MHz Soekris box used in CarTel:

Block Size	**Time**	**Transfer Rate**
1 MB	N/A seconds	15 kilobytes per second
100 KB	N/A seconds	9 kilobytes per second

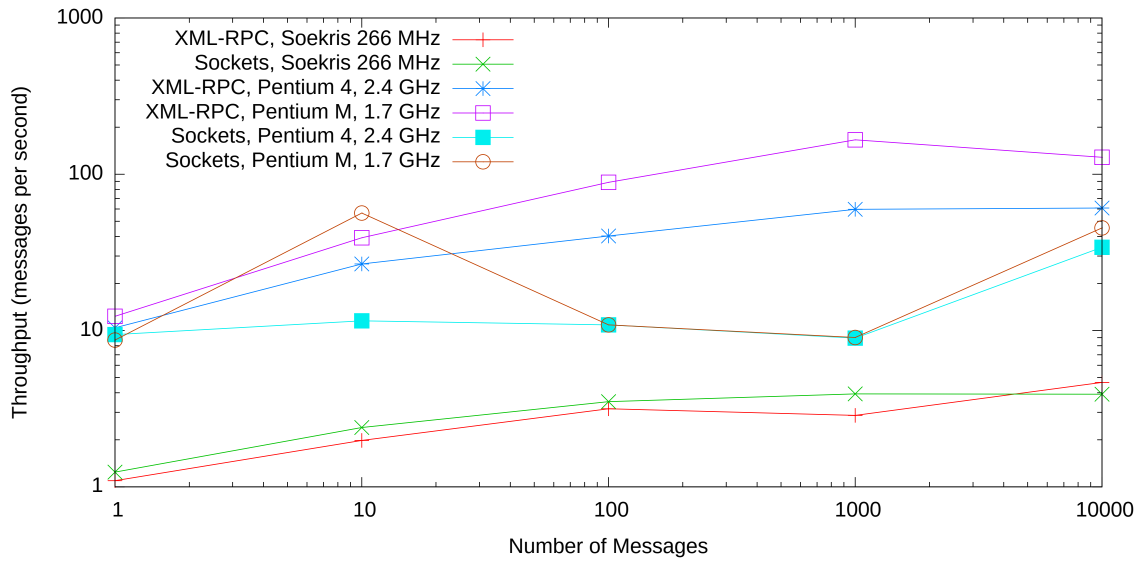
Table 4.2: Initial CafNet file transfer results on a Soekris box

On the Soekris box, once the source application passes messages to the source CTL, the messages quickly flow to the source CNL, between the MALs, reaching the destination CNL and then the destination CTL, where they become queued. The application receives these queued messages quite slowly, indicating that the transmission between the destination CTL and the destination application is a bottleneck in message delivery. In the case of the tests shown in Figure 4.2, no total transfer time is available because so many pending messages accumulated at the CTL, and the CafNet stack exited from using too much memory, as the Soekris box only has 128 MB of memory.

Since communication between the CTL and application is performed using XML-RPC, and there is no performance problem between the two MALs, the test suggests that XML-RPC is inefficient. XML-RPC is a text-only protocol, so any field that might contain binary data must be base64-encoded. During each message transfer, the message must be base64-encoded and base64-decoded twice, leading to this inefficiency.



(a) CafNet Performance: Unidirectional, Varying Number of Messages



(b) CafNet Performance: Bidirectional, Varying Number of Messages

Figure 4-1: The CafNet stack scales reasonably well with increasing number of messages and shows increased performance as more messages were sent. The stack was tested with a 1 MB CNL cache with different numbers of 100-byte messages with no ACKs requested. The number of messages shown is the number of messages sent in each direction.

4.2 Plain Sockets

To increase CafNet's performance, I replaced XML-RPC with a custom protocol as described in section 3.2. Like XML-RPC, both CafNet applications and the CTL open server sockets to process incoming requests. Although it would be convenient to make requests using some standard format like XML or YAML, no common format allows the use of binary data, and XML and YAML both require base64 encoding. CafNet, therefore, must make use of a custom protocol to avoid the overhead of converting message payload data to text.

Figure 4-1 shows the performance of the previous XML-RPC CafNet stack and the current plain socket CafNet stack. Not surprisingly, the time it takes to send messages of a fixed size scales somewhat linearly with the number of messages.

Unlike with the Soekris box, however, the Pentium 4 and Pentium M machines in many cases performed better with XML-RPC than with plain sockets. This is likely because the plain socket implementation opens a new TCP connection for every method call between the CTL and the application. While some XML-RPC implementations (such as the Python implementation) do the same thing, other XML-RPC implementations (such as the Java implementation CafNet used[1]) have a `Keep-Alive` option to reuse a single connection. The tests shown by Figure 4-1 were performed with a Java CafNet application, and therefore made use of the `Keep-Alive` feature. In the future, the custom plain socket protocol could be improved to support a similar feature.

Figure 4-2 shows the performance of the XML-RPC and plain socket CafNet stacks with varying message sizes. In each test, a number of messages was sent such that the total payload data would equal 1 MB. As expected, the larger messages had less overhead, since they have less header data to transmit, and since there are fewer messages for the CafNet stack to process.

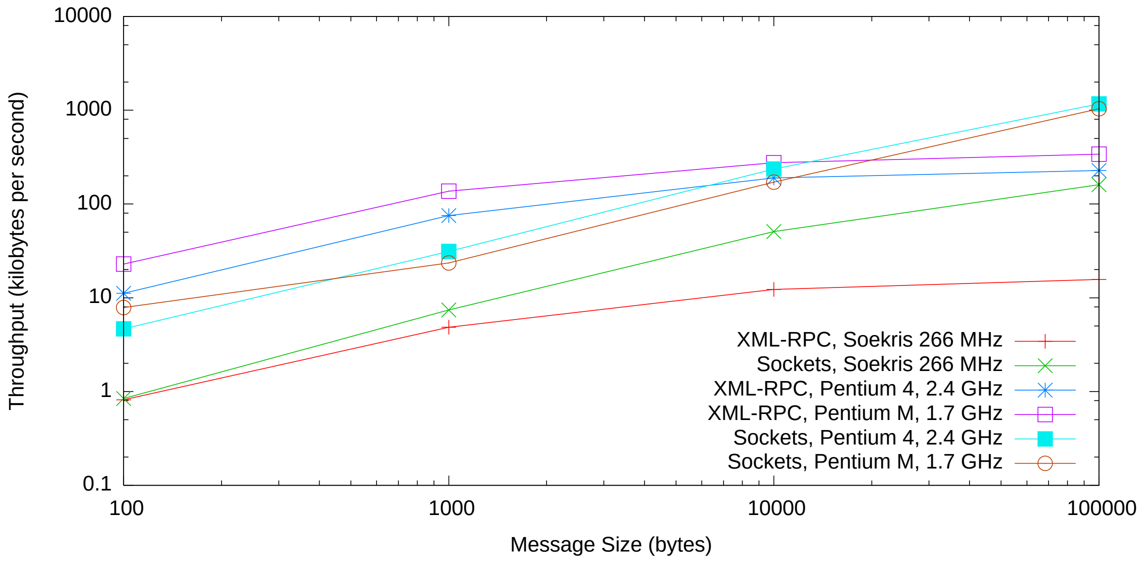
Table 4.3 also suggests that most of the overhead in the CafNet stack is the number of messages, not the size of the message payload. The 1000-byte messages

[1]Apache XML-RPC version 2; `http://ws.apache.org/xmlrpc/`

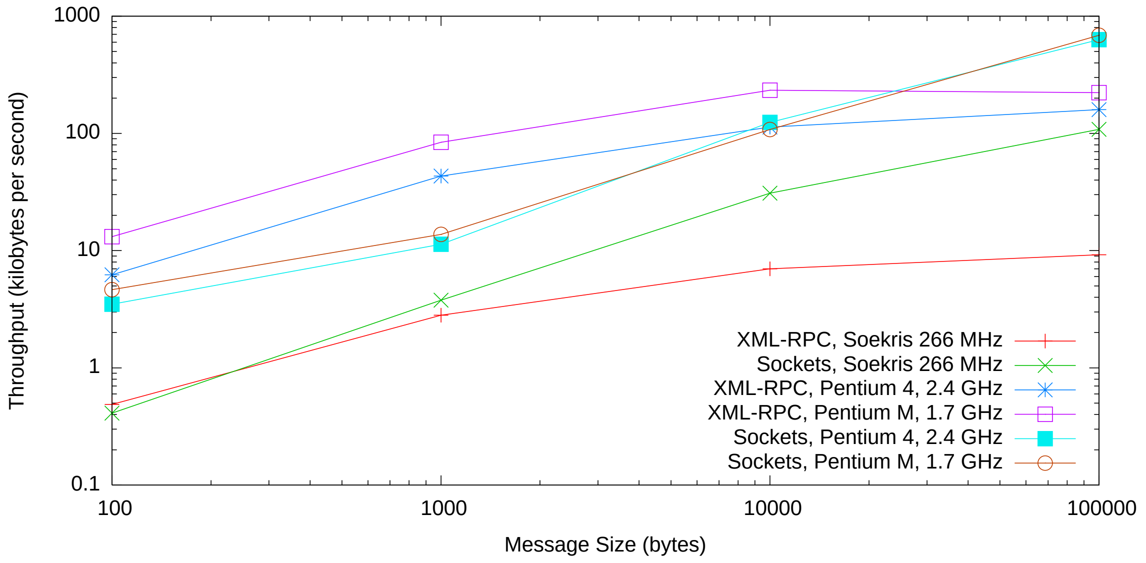
Machine	**100-byte Messages**	**1000-byte Messages**
Pentium 4, Sockets	61 seconds	33 seconds
Pentium M, Sockets	56 seconds	43 seconds
Soekris, Sockets	137 seconds	138 seconds
Pentium 4, XML-RPC	10 seconds	13 seconds
Pentium M, XML-RPC	5 seconds	7 seconds
Soekris, XML-RPC	180 seconds	211 seconds

Table 4.3: CafNet performance for sending 1000 messages of different sizes.

did not take appreciably longer to send, and in the case of the non-Soekris machines, experimental variation caused the 1000-byte messages to be sent more quickly than the 100-byte messages in some cases.



(a) CafNet Performance: Unidirectional, Varying Message Size



(b) CafNet Performance: Bidirectional, Varying Message Size

Figure 4-2: CafNet's overhead depends on the number of messages transmitted, not the total payload data transmitted. The stack was tested with a 1 MB CNL cache with different sizes of messages sent with no ACKs requested. For each size, the total payload data sent in each direction equaled 1 MB.

Chapter 5

Related Work

Much work has been done in delay-tolerant networking and similar fields. Some parts of CafNet build upon ideas from this previous work, while other parts use other designs.

5.1 Addressing

Research performed at Intel describes a DTN setup that divides nodes into regions with globally unique names [6, 7]. Within each region, nodes communicate with a common network protocol. To send messages to nodes in other regions, nodes transmit data to a DTN gateway at the border between two regions, which converts the data transmission between protocols if necessary. Nodes are identified with "node tuples" containing a region name and an entity name; the entity name is opaque outside the region. When sending messages, the message is first delivered to the destination region. Once there, internal routing is used to deliver messages to the local entity name.

5.2 Routing

Jain [11] explores the difference between proactive and reactive routing in DTNs. In proactive routing, routing tables are computed and pushed to nodes as updates in the network topology occur. DSDV (Destination Sequenced Distance Vector) and OLSR (Optimized Link-State Routing), used in ad-hoc networks, are both proactive routing protocols. Protocols that use reactive routing, on the other hand, calculate routes as they are needed. When a node sends a message to another node, the route is determined on-demand. AODV (Ad-hoc On-demand Distance Vector) and DSR (Dynamic Source Routing) are reactive routing protocols in ad-hoc networks.

While proactive routing may be more efficient, the routing tables required would be rather large and may consume both significant memory and network resources. On the other hand, reactive routing may be slower, given that end-to-end connectivity is not present and so additional hops may be required to determine the path.

One of the algorithms explored in Jain [11] is a form of reactive source routing. The minimum total delay for a route is calculated by summing the Minimum Expected Delay (MED) between each pair of two consecutive nodes, which takes into account the average waiting time, propagation delay, and transmission delay. Unlike many other algorithms, MED does not require knowledge of future encounter times, and instead, simply relies on the average expected time.

Jones [12] expands on MED and establishes the Minimum Estimated Expected Delay (MEED). Like MED, each node computes expected delays to other nodes and propagates that information to the rest of the network. Unlike MED, MEED dynamically routes messages at each hop and dynamically updates the routing table whenever neighbors arrive. If the sending node unexpectedly comes in contact with a neighbor that can get the message to the destination faster than the expected neighbor, the node passes the message to the unexpected neighbor rather than holding on to it and passing it to the expected neighbor as it would have otherwise done.

5.3 Social Routing

Herrmann [9] explores a mobility model that focuses not on geographic positions, but instead on social mobility and groups of people coming together at times and interacting with each other. This model assigns each mobile node, representing a user, a list of meeting points to visit. For each meeting point, the model assigns a meeting time while ensuring that the time does not overlap with any other times the user is busy. Simulations of data sets created with this model exhibit properties similar to those of real social networks. Such a model can be used in a DTN where data is primarily passed between users to predict how data will flow, and to route messages appropriately.

In contrast with these models, Su [17] uses actual data from interactions between users to determine whether using mobile devices to form an ad-hoc network is feasible. First, users were given devices that would record encounters with other such users. This trace data was then used to run simulations of message delivery using epidemic routing, in which data is sent to every node encountered and flooding the network. Not all intermediate nodes were equally successful at delivering the messages, suggesting that this information can be used to improve routing.

5.4 Mobile Sensor Networks

While many sensor networks involve stationary nodes, ZebraNet [13] is a sensor network that tracks zebra movements by recording GPS data and sending it through other mobile zebra nodes to roaming researchers. Since end-to-end connectivity is not present, mobile ad-hoc network protocols cannot be used. The researchers are continuously moving and have unpredictable positions, so data cannot simply be forwarded to a known position.

Given the power requirements of the ZebraNet project, data cannot simply be flooded to every node each zebra encounters. Instead, ZebraNet uses known zebra movement patterns (such as grazing, searching for water, and lack of significant sleeping) and previous history to predict expected zebra encounters and routing the collected data appropriately.

5.5 Transport Layer

Harras [8] discusses transport layer operations in DTNs using epidemic routing and different ways of ensuring messages are reliably transmitted. The first way is hop-by-hop reliability, in which the next hop takes responsibility for delivering the message. Fall [7] calls this mechanism "custodial transfer," which is discussed further in Section 6.1.2. The second approach is "active receipt," which is what CafNet uses — the acknowledgment message is sent back as a separate message and forwarded through the DTN. While effective, this approach uses more DTN resources as it sends two messages through the network, rather than just the original one. A third approach is "passive receipt" — nodes do not send explicit acknowledgments, but if they encounter another node sending a message that has already been delivered, they inform the node to stop sending that message. Eventually, a node that has stopped sending the message will encounter the source and ask the source to stop sending the message, completing the acknowledgment process.

5.6 Full DTN Design

Demmer [6] describes a different complete DTN design. The primary component of this design is the Bundle Router, which receives information from many events, and then uses this information to schedule messages. Messages to be forwarded are then sent to a Bundle Forwarder, which executes the instructions received from the router and sends messages to one of several Convergence Layers. Each Convergence Layer is an adapter between the DTN and an underlying link mechanism, similar to a CafNet Mule Adaptation Layer.

Operations that take place in the CafNet Network Layer are separated out into several modules in this design. The Persistent Store, similar to the buffer in the CafNet Network Layer, holds messages to be forwarded later. The Contact Manager maintains the status, performance, and history of each link.

Finally, like the CafNet Transport Layer, this design communicates with DTN applications using interprocess communication. The Registration Module communicates with the application, processes application messages, and passes messages the stack receives to the appropriate application.

Chapter 6

Conclusion

6.1 Future Work

While CafNet is functional enough for use in CarTel, more work remains to be done to make it a fully-featured DTN.

6.1.1 Routing

At the moment, the CafNet stack will only route messages directly to their destination — that is, if a CTL with address A wants to send a message to a CTL with address B, CTL A will pass it to CNL A, and CNL A will only pass it to another CNL if its CNL address is B.

While CNLs already exchange `CNL-Self-Identify` messages so they can identify themselves to each other, they will also need to identify what neighbors they are likely to see, and when they next expect to encounter them. The sending CNL can then determine whether it should send certain messages through the remote CNL, even if it is not the destination. Implementing routing such as MEED [12] as mentioned in Section 5.2 will then allow CafNet to forward messages through other nodes.

6.1.2 Custodial Transfers

Some DTN applications may have data they want to ensure gets to the destination, but for space or power limitations, may be unable to save a copy of the message in case an END-TO-END ACK is not received. Fall [7] describes a system of custodial transfers, in which the responsibility of delivery is shifted to another node. The application requests a custodial ACK; after an intermediate node that supports custodial transfers sends an acknowledgment back, the sending node can remove its copy of the message. From that point on, the intermediate node is responsible for delivering the message and retransmitting as necessary, unless it passes that responsibility to another node that supports custodial transfers and completes the transfer.

Not all nodes need to support custodial transfer for the transfer to work. Suppose a message travels from A to B, B to C, and C to D, and the message requests a custodial transfer, and only node C supports custodial transfers. A sends the message to B, and B sends the message to C. C returns a custodial ACK to A indicating it has accepted custody of the message. At this point, A no longer has to retransmit the message and can remove it from memory, and C is responsible for ensuring the message reaches its destination.

CafNet currently does not implement custodial transfers, although it would be useful for it to do so. This could be implemented by having a special application for each CTL that stores the messages acquired through custody transfer from other nodes. This application then schedules all the messages as normal while requesting END-TO-END ACKs and optionally requesting custody transfers itself.

6.1.3 Bluetooth

As mentioned in section 3.4.1, work on the Bluetooth MAL remains to be done before it is fully integrated into the CafNet stack. A single connection should allow MALs on both ends to send multiple messages at the same time, without establishing a new connection for each direction. One way in which this might be done is to stop using FIN and FINACK messages to indicate the end of message transmission.

Instead, if one endpoint has stopped transmitting messages for 5-15 seconds or so, it is considered finished and the connection can be closed if the other endpoint is also finished. Connection status, however, is currently done in the Message class, so a bit of work may be needed to gain access to the connection status. Alternatively, a timeout could be added to the appropriate part of the appropriate Message constructor.

When the Bluetooth MAL discovers other Bluetooth devices that support the CafNet protocol, it should connect to each one in turn so that `CNL-Self-Identify` (and potentially other messages) can be exchanged. After this initial connection, the CNL can maintain the association between the MAL ID and remote CNL address until the remote device goes out of range. This makes the Bluetooth MAL act more like the TCP MAL, even if it does not continuously maintain the connections to each of its neighbors.

Piconets and Scatternets

As previously mentioned, Bluetooth by default only allows connecting to one other device at a time. However, it is possible to form small ad-hoc networks called piconets, which are formed by a master device and up to seven slave devices. In the case of normal computing devices, like a Bluetooth keyboard and mouse, the computer is the master, and the keyboard and mouse are each slave devices. Having one CafNet node act as a master and up to seven other nodes act as slaves would allow a form of multicasting such that a sender could send to seven other devices simultaneously, and the other seven devices to send to the master at the same time, saving time and power by not having to make separate transmissions. Using piconets also reduces collisions that might occur, since these devices will not all try to connect the other devices at the same time.

It is possible for devices to be in multiple piconets, forming a scatternet. A node can be a slave in two different piconets at the same time, or be a slave in one piconet and a master in another, so even with the 8 node per piconet limit, it is still possible to have all local Bluetooth devices communicate each other without continuously disconnecting and establishing new connections. However, some method of determining which devices should be masters and which devices should be slaves will need to be determined.

6.1.4 Security

Security will also be an issue. As it is now, the network is extremely susceptible to denial of service attacks in the form of message floods. A malicious sender may send a large number of supposedly high-priority messages to attempt to force the device to drop previous legitimate messages in favor of the junk messages.

Another security issue is the privacy and protection of the data getting sent. The messages will be kept on mobile devices that may belong to absolute strangers. In such a case, there is little that the protocol can do to prevent strangers from reading the data. Privacy and protection will thus need to be implemented on an end-to-end basis, probably by signing and encrypting the message before sending it. Using public keys as CTL addresses is a first step to do so, since it makes it easy to encrypt messages sent over CafNet.

6.2 Conclusion

In this thesis, I designed and implemented CafNet, a delay-tolerant network stack, for use with CarTel. While writing the first CafNet implementation, I corrected flaws in and improved upon the pre-existing design. Initial versions of this implementation revealed many unexpected issues, such as properly dealing with link mechanism disconnections and ensuring the stack does not block when they occur. Performance testing of the network stack showed that the Soekris box used in CarTel did not perform well with the XML-RPC stack used in the implementation, and so it was replaced with a custom protocol over plain TCP sockets.

While more work, such as implementing muling by supporting forwarding to neighbors other than the final destination, remains to be done to make CafNet a fully-featured delay-tolerant networking stack, the CafNet stack now has good performance and can be used in CarTel and with other delay-tolerant network applications.

Bibliography

[1] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *ACM SIGCOMM '99*, Cambridge, MA, September 1999.

[2] Vladimir Bychkovsky, Bret Hull, Allen K. Miu, Hari Balakrishnan, and Samuel Madden. A Measurement Study of Vehicular Internet Access Using In Situ Wi-Fi Networks. In *MobiCom '06: Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, Los Angeles, CA, September 2006.

[3] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, E. Travis, and H. Weiss. Interplanetary Internet (IPN): Architectural Definition. May 2001. `http://www.ipnsig.org/reports/memo-ipnrg-arch-00.pdf`.

[4] David D. Clark. The structuring of systems using upcalls. In *SOSP '85: Proceedings of the 10th ACM Symposium on Operating Systems Principles*, New York, NY, 1985.

[5] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM '90: Proceedings of the ACM Symposium on Communications Architectures & protocols*, New York, NY, 1990.

[6] Michael Demmer, Eric Brewer, Kevin Fall, Sushant Jain, Melissa Ho, and Rabin Patra. Implementing Delay Tolerant Networking. *Intel Research Technical Report IRB-TR-04-020*, Dec 2004.

[7] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, New York, NY, 2003.

[8] Khaled A. Harras and Kevin C. Almeroth. Transport Layer Issues in Delay Tolerant Mobile Networks. In *IFIP Networking*, Coimbra, Portugal, May 2006.

[9] Klaus Herrmann. Modeling the Sociological Aspects of Mobility in Ad Hoc Networks. In *MSWiM '03: Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, New York, NY, 2003.

[10] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen K. Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A Distributed Mobile Sensor Computing System. In *SenSys '06: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, Boulder, CO, November 2006.

[11] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a Delay Tolerant Network. In *SIGCOMM '04: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, New York, NY, 2004.

[12] Evan P. C. Jones, Lily Li, and Paul A. S. Ward. Practical Routing in Delay-Tolerant Networks. In *WDTN '05: Proceeding of the 2005 ACM SIGCOMM Workshop on Delay Tolerant Networking*, New York, NY, 2005.

[13] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *ASPLOS*, 2002.

[14] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating Key Management from File System Security. In *SOSP '99: Proceed-*

ings of the 17th ACM Symposium on Operating Systems Principles, New York, NY, 1999.

[15] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. *IETF RFC 4423*, May 2006.

[16] Rahul C. Shah, Sumit Roy, Sushant Jain, and Waylon Brunette. Data MULEs: Modeling a Three-tier Architecture for Sparse Sensor Networks. In *Proceedings of the First IEEE Workshop on Sensor Network Protocols and Applications*, May 2003.

[17] Jing Su, Alvin Chin, Anna Popivanova, Ashvin Goel, and Eyal de Lara. User Mobility for Opportunistic Ad-Hoc Networking. In *WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications*, Washington, DC, 2004. IEEE Computer Society.

[18] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the Web from DNS. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[19] Wenrui Zhao, Mostafa Ammar, and Ellen Zegura. A Message Ferrying Approach for Data Delivery in Sparse Mobile Ad Hoc Networks. In *MobiHoc '04: Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, New York, NY, 2004.