The Dissertation Committee for Agoston Petz
certifies that this is the approved version of the following dissertation:

Context-Based Adaptation in Delay-Tolerant Networks

Committee:

Christine Julien, Supervisor

Scott Nettles

Lili Qiu

Sanjay Shakkottai

Sriram Vishwanath

Context-Based Adaptation in Delay-Tolerant Networks

by

Agoston Petz, B.S.E.E., M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2012

This dissertation is dedicated to my family, especially to my parents Ernő and Marianna, who have ever encouraged and supported me in all of my pursuits, most of them not nearly as practical as this one.

Acknowledgments

This dissertation would not have been possible without a great deal of help from a great number of sources, all of whom I am indebted to. I would especially like to thank my advisor Christine, who is undoubtedly the best advisor a graduate student could possibly hope for and who's direction, advice, and support has shaped me into the researcher I am. I would also like to thank my dissertation committee, especially Scott—our many conversations helped focus my perspective on systems research in general. I would like to thank all of my fellow students in the Mobile and Pervasive Computing lab, especially Drew, Seth, and Vasanth—our friendship and (often irreverent) conversations broke up the occasionally monotonous day-to-day humdrum. I'd also like to thank my fellow graduate student Robert, while it goes without saying that I am grateful for our lasting friendship, I am certainly also grateful for our moments of graduate student commiseration. Everyone can use a sympathetic ear and a good wallowing from time to time. Thank you to Mariposa as well, you have put up with a lot of career uncertainty from my end with remarkable flexibility, and I realize I haven't been the most easy-to-get-along-with person this past year. Thanks for your patience and love!

There is no way even a small part of the results in this dissertation would have been possible with the help of Chien-Liang. Thank you for your enthusiastic support of my experiments, your incredible work ethic, and your willingness to wake up at ridiculous hours of the night on the off-chance that the wireless channel might

be better at four in the morning. I would also like to thank my often-collaborator Brenton. Our hack-a-thons have been some of the most fruitful weeks of my entire graduate career, and I will never forget that it is you who first introduced me to ExtremeCom, the single most amazing conference I have ever heard of, much less taken part in! I am thrilled to have hiked up Indrahar Pass with you.

I would also like to thank two professors from my undergraduate days who in a large part paved my way to the PhD. Bill Bard, who taught me nearly everything I know about networks, and Brian Evans, who set the professorial model and the pedagogical bar to which I will aspire if I ever go the way of academia. You have both inspired me to finish what I started.

Finally, I would like to thank my parents Ernő and Marianna. Thank you for your encouragement, for your unwavering enthusiasm, and for your continued support of my education.

Agoston Petz

The University of Texas at Austin
December 2012

Context-Based Adaptation in Delay-Tolerant Networks

Agoston Petz, Ph.D

The University of Texas at Austin, 2012

Supervisor: Christine Julien

Delay-tolerant networks (DTNs) are dynamic networks in which senders and receivers are often completely disconnected from each other, often for long periods of time. DTNs are enjoying a burgeoning interest from the research community largely due to the vast potential for meaningful applications, e.g., to enable access to the Internet in remote rural areas, monitor animal behavioral patterns, connect participants in mobile search and rescue applications, provide connectivity in urban environments, and support space communications. Existing work in DTNs generally focuses either on solutions for very specific applications or domains, or on general-purpose protocol-level solutions intended to work across multiple domains.

In this proposal, we take a more systems-oriented approach to DTNs. Since applications operating in these dynamic environments would like their connections to be supported by the network technology best suited to the combination of the communication session's requirements and instantaneous network context, we develop a middleware architecture that enables seamless migrations from one communica-

tion style to another in response to changing network conditions. We also enable context-awareness in DTNs, using this awareness to adapt communications to more efficiently use network resources. Finally, we explore the systems issues inherent to such a middleware and provide an implementation of it that we test on a mobile computing testbed made up of autonomous robots.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Delay-tolerant networks (DTNs) are networks of intermittently connected nodes whose topologies are subject to constant change due to mobility or adverse networking conditions. Because the endpoints of a communication session in a DTN may never be connected in the traditional sense, communication is often supported through the ferrying of messages by mobile nodes that opportunistically encounter sources and destinations. In a sense, DTNs are an extreme variant of mobile ad hoc networks (MANETs). In the latter, high-levels of node mobility cause the network's routing topology to change, but applications in MANETs can generally assume that *some* connected path exists from the source to the destination at any point in time. Communication in DTNs, on the other hand, often relies on *temporal* connectivity—the idea that a source node may encounter a node that will, *in the future*, encounter the destination (or a node that will be able to pass it off to such a node after some number of such transitions). The eventual delivery of a message in a DTN can take any number of these "hops", and routing in such networks can be thought of as a *best effort* paradigm; there is no guarantee of success; DTN routing algorithms attempt to increase the probability of successful delivery.

DTN architectures potentially apply in many application instances. In an

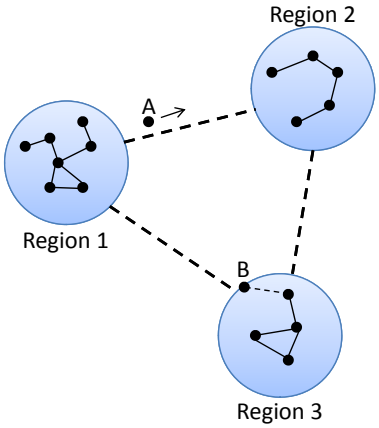


Figure 1.1: A generic DTN target environment

urban setting, commuter transportation systems can be used to carry messages from one region of a city to another [18]. A remote village may be well-connected internally but less reliably connected to the wider world [43]. In a military network, devices within a post or base camp may have good connectivity to one another but may be connected to devices in other regions only by roving UAVs or convoys that relay messages from time to time. Researchers moving among tagged herds of zebras can collect and share information about zebra behavior and movement [25]. Pockets of well-connected sensor networks can be periodically connected via roving mobile robots [42]. Nodes at the fringe of cellular networks can still access the Internet through cooperation with other nearby users.

Figure 1.1 shows an abstraction of the common operating environment shared by these disparate applications. The figure shows three well-connected regions that are interconnected transiently via delay-tolerant links (dashed lines in the figure). In the deployments depicted, nodes can break off from the well-connected regions and transit between them (such as node A, which is transiting between regions 1 and 2, and node B, which has just arrived at region 3). Research issues relating to routing and understanding network topologies are well-studied [7,23,31,35], and great strides

have been made with respect to supporting the new paradigm of mobile computing that DTNs exhibit. In this dissertation, we study a more practical systems issue, namely the ability of networked devices in DTNs to adapt their connections to the changing operating environment.

Since the connectivity among devices in a delay-tolerant network (DTN) is widely varying and unpredictable, it is unlikely that any particular routing protocol will be ideal for every situation. It is necessary to integrate DTN solutions with traditional mobile computing solutions to ensure the availability of the best communication support possible at any given moment. Applications executing in DTN environments would like their connections to be supported by the network technology best suited to the combination of the communication session's requirements (e.g., for throughput and delay) and the instantaneous network context. While previous work has investigated choosing the best implementation strategy for a communication session at its inception, the more difficult problem of adapting an ongoing communication session is largely unexplored. Several aspects make this a challenging problem, including 1) efficiently sensing the appropriate network and data context to allow for adaptation decisions; 2) aggregating and organizing context in the right place to allow for intuitive adaptation strategies to be developed; 3) enabling intelligent cross-layer design to support the tuning of the network stack to allow adaptation; and 4) providing seamless transitions for applications (i.e., an application should not need to be aware of the change in underlying implementation).

In this dissertation we focus our work on mechanisms to seamlessly adapt network communications to suit the properties of the (changing) network. We base our adaptation on a combination of *network*, *application*, and *system* context, describing such a context in terms of easily and cheaply sensed metrics. We develop a middleware architecture and prototype to support applications in delay-tolerant

environments by adapting the underlying network, transport, and routing protocols. As part of this architecture, we also develop a flexible framework for collecting context which is capable of effecting changes to the network stack in response to changing context. Finally, we deploy our system on real-world mobile robots and provide several evaluations showing the benefits of such an architecture.

1.1 Research Contributions

This dissertation addresses the challenges of supporting applications in a dynamic delay-tolerant network setting by automatically adapting connections and underlying network protocols. Specifically, this dissertation makes the following contributions:

1. Develop an architecture for dynamic connection migration in delay-tolerant networks and demonstrate its utility on a real system. We design and build two prototypes to examine the benefits and trade-offs using two separate evaluation environments (network emulation using real hardware and software, and real world, small-scale, indoor testbed evaluation using autonomous robots). These two architectures and subsequent prototypes allow us to focus our efforts in two important ways. First, they offer insight about what types of network stack adaptations are beneficial, and even *where* in the networks stack the adaptations should be implemented to most benefit delay-tolerant network systems with the least overall system complexity. Second, they focus our efforts in designing context sensing and aggregation strategies by offering insight into how such software should integrate with the system.

2. Create a context-sensing framework for delay-tolerant networks. We examine both the design of the context framework, and the types of context that can be sensed, incorporating both *passive metrics* which can be sensed from ex-

isting network communication and thus do not inquire a "sensing overhead", and *active metrics* which offer increased accuracy at the cost of increased network communication. We also provide a implementation of context-sensing framework capable of context sharing across multiple nodes, and that supports publish/subscribe mechanics.

3. Design and implement a complete systems solution that incorporates concepts from Research Task 1 with the context framework from Research Task 2 to adapt a real delay-tolerant network stack.

4. Use the Pharos mobile computing testbed to design and perform a series of real-life application validation and evaluation studies using the system developed in Research Task 4.

1.2 Overview

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the Pharos Testbed, a mobile computing testbed comprised of autonomous robots that we use for a platform for most of the results presented in this dissertation. Chapter 3 presents work detailing our efforts towards a dynamic DTN architecture that allows applications' ongoing connections to seamlessly migrate between two different communication stacks: a traditional mobile ad hoc networking stack (for use in well-connected regions such as those shown in Figure 1.1) and a dedicated DTN stack (for use in transient communication situations). Chapter 4 covers the design and implementation of a context framework that collects and shares context in delay-tolerant networks. It allows for designers to easily and quickly create adaptation algorithms to effect the network stack through multi-threaded "agents" that can act on changing context information by tuning network stack parameters. In this chapter we also present a compelling case study for using such context even

in mixed cellular/DTN networks to improve the coverage and capacity of cellular networks. In Chapter 5, we present the design and implementation of a complete system solution that ties all of these ideas into a single, modular, and easily extensible middleware, and Chapter 6 describes our validation and evaluation of the whole system using the autonomous robots of the Pharos Testbed. Finally, Chapter 7 concludes.

Chapter 2

The Pharos Testbed

The Pharos Testbed [45,53] is a platform for interdisciplinary experimental research for mobile and pervasive computing. In particular, the Pharos project aims to create repeatable experiments for environments with mobile participants. We use the autonomous robots of the Pharos Testbed for nearly every experiment conducted to support this dissertation; this chapter presents an overview of the testbed and the robots in order to provide the necessary background knowledge for the experiments discussed later in this dissertation. Neither the Pharos testbed nor the Proteus nodes themselves are contributions of this dissertation—this section is simply included for the benefit of the reader.

2.1 Overview of Pharos Testbed

The Pharos vehicular testbed consists of numerous autonomous vehicles called *Proteus*. Designed for modularity and economy, Proteus uses commercial-off-the-shelf (COTS) equipment to maximize robustness, flexibility, and cost-effectiveness. Below we discuss the design in four major functional sections: software support, mobility, behavior, and interaction.

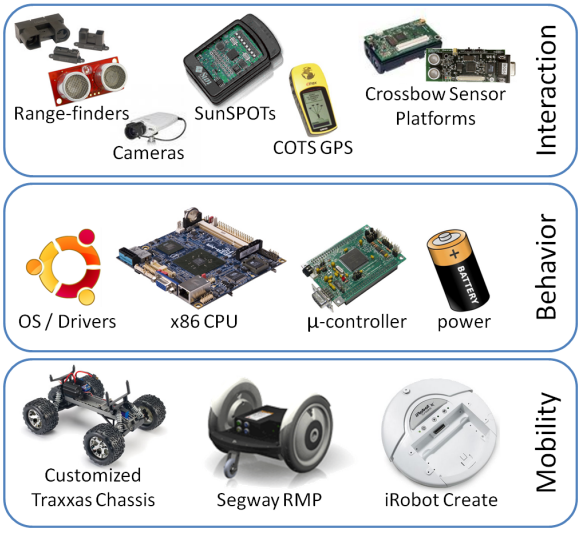


Figure 2.1: Hardware architecture of Proteus Nodes

2.1.1 Physical Mobility

Physical mobility is provided through one of three options: iRobot Create, Segway RMP50, or customized Traxxas Stampede. The Create is a low cost, low speed, differentially steered robot with a simple serial control interface. The RMP50 is based on Segway's popular self-balancing products and is controllable over a CAM bus or USB port. It is more expensive than the Create but offers higher speeds, higher payload capacity, and long-range outdoor use. The third mobility option is a customized Traxxas Stampede. The Stampede is a high-performance remote controlled car with Ackerman steering and 4-wheel-independent suspension. Each platform provides its own power to reduce dependencies and interference with the node's other components. While the Traxxas is not a COTS component, the low cost, light weight, outdoor compatibility, and range of speeds makes it a desirable option for experimenters. The Traxxas mobility platform is controlled by the on-board microcontroller described in the next section [1]. In this dissertation we use

[1]Details on the hardware, assembly, and software are all available at `http://www.pharos.ece.utexas.edu/`.

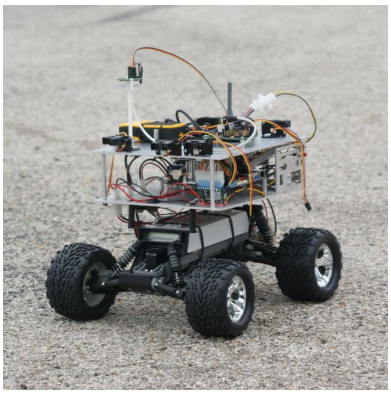


Figure 2.2: The Proteus Mobile Node

only the Stampede variant of the Proteus node, an example of which is shown in Figure 2.2.

2.1.2 Behavior and Communications

A low-power x86 Linux-based motherboard coupled to a Freescale microcontroller provides the platform for Proteus node behaviors. This dual architecture approach offloads many of the real-time tasks to the microcontroller while allowing the x86 system to focus on higher level aspects. The two-level approach also opens a wide range of I/O options for connecting sensors and other peripherals. Basic communications are provided by an onboard 802.11 b/g wireless NIC with a 5.5dBi antenna.

Typically, mobility commands are executed by user-level applications through the Player/Stage API running on the x86 computer. Depending on the platform, these commands are then sent to the mobility platform via serial interface, USB, or the microcontroller. Sensor data is collected in much the same way via serial, USB, or the microcontroler.

2.1.3 Interaction

The third functional area of the Proteus node is sensing and actuating. We currently support various range-finding sensors, digital compass, GPS, and cameras. Most of the sensors we use are specifically supported by third-party drivers for the Player API. The remaining sensors have matching interfaces in the Player API and only require us to implement device-specific drivers. These drivers typically reside on the microcontroller and are exposed to the x86 through the existing serial connection. The sensed data can then be used not only to influence the node's mobility but also in applications running on the x86 computer. This dissertation makes use of the digital compass and GPS for outdoors navigation, and the camera for indoor navigation using line-following algorithms.

2.2 Pharos Software Architecture

The Pharos testbed's software architecture is shown in Figure 2.3. At a high-level, it consists of three main components: a Pharos client residing on a laptop that wirelessly communicates with one or more Proteus nodes, the Pharos Server running on each Proteus's x86 computer, and sensor/actuator drivers that reside within Proteus's micro-controller.

Pharos Client. The Pharos client is written in Java™ and serves as the experiment coordinator. It assigns motion scripts to Proteus nodes and initiates the execution of the motion script. Upon receiving the motion scripts and experiment configuration, which contains the node specifications and motion script assignments, the Pharos client wirelessly connects to the Pharos servers on each node, configures them, and coordinates the start of the experiment. At this point, the nodes may move out of range of the Pharos client, and it has no further role until the experiment is over when it collects log files, organizing them by experiment identifier and node

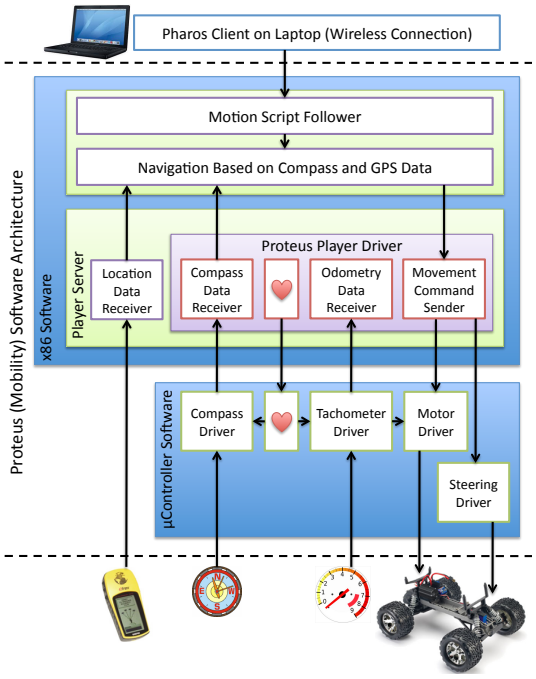


Figure 2.3: The Proteus Mobility Architecture

ID.

Pharos Server. The Pharos server consists of a Motion Script Follower and a Navigation component. The Motion Script Follower informs the Navigation component of the next waypoint and desired speed; upon arrival it pauses for the specified amount of time and repeats the process. The software that implements the network protocol being evaluated runs in parallel with the Motion Script Follower and can influence the sequence of waypoints that a node visits and the speed at which it travels. The Navigation component requires compass and GPS data, using both to adjust the steering angle and speed. The Navigation component obtains the sensor data and issues the movement commands through a Player server that also runs on the x86. Hardware actuation and feedback is accomplished through a combination of the well-known Player Server robot API and custom micro-controller drivers. The server is also capable of exposing the robot heading, location, speed, and destination

via a service that provides this information every two seconds on a local TCP socket. We use this capability in this dissertation to collect this geographical context from the mobility controller.

2.3 Conclusion

Using the Pharos Testbed to support this dissertation allows us to leverage a Linux-based mobile autonomous system for our experiments. This allows us to build real systems solutions, and to test them under real-world conditions using commodity hardware. At the same time, the Pharos Testbed allows us to repeat the exact mobility patterns across experiments, allowing for comparability between independent experiments— something that would not be possible if human subjects were used to provide the "mobility" of the devices.

Chapter 3

Dynamic Network Stack Adaptation

In this chapter, we study the ability of networked devices to adapt their network interactions to the changing operating environment. We present two approaches to test our ideas: a Linux network stack based approach that we evaluate using a purpose-built delay-tolerant network emulator (hereafter referred to as the *Dynamic Stack Swapper*, or DynSS) [47], and an advanced, modular approach (hereafter referred to as the *Middleware for Delay-Tolerant Mobile Ad-Hoc Networks*, or MaDMAN) [48] that we evaluate on a real-world, delay-tolerant network using nodes from the Pharos Testbed. With an aim to support networks such as the one depicted in Figure 1.1, both approaches focus on transitioning between two stacks: a traditional stack (for use within the connected regions in the figure) and a DTN stack (for use in transient communication situations). Starting with the shared goal and related work, we present both approaches and their respective implementations and evaluations.

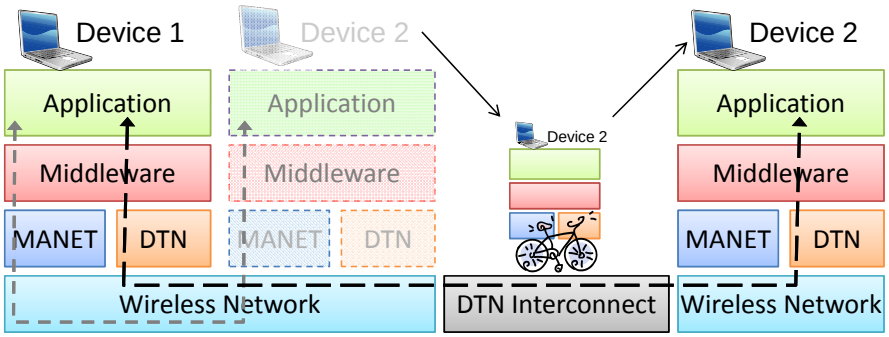


Figure 3.1: Operation of a dynamic delay-tolerant network architecture

3.1 Goal

Our goal is to distance the developer from network implementation details. This makes it simpler to write programs because the programming constructs are more intuitive. It also separates the implementation of the conversation from the available underlying primitives, allowing applications to delegate responsibility for filling in the best fit communication primitives. In the remainder of this chapter, we will speak mostly in terms of a single "application" that comprises two end points and the (potentially dynamic) connection between them. Aspects of the ideal resulting architecture are depicted and described in Figure 3.1. As the figure shows, applications within the same local area (e.g., within the same village) use the underlying MANET communication support (dashed gray connection). When the network conditions change to make MANET communication unreasonable or impossible (e.g., Device 2 begins to move away from the village), the communication session is automatically migrated to the DTN network technology. This migration occurs transparently to the application; the ongoing communication is not interrupted (though its quality of service may change). We focus on enabling the mechanics of this transition and the systems and communication issues that must be resolved to enable seamless transitions. The remainder of this chapter is organized as follows. In the next section,

we provide background information and related work. Section 3.3 describes the architecture for the entire Dynamic Stack Swapper (DynSS) system, while Section 3.4 presents our evaluation setup and custom-built delay-tolerant network emulator. Section 3.5 presents the results from this evaluation, demonstrating the potential benefits of swapping stacks in particular situations, and Section 3.6 presents our conclusions based on the experience designing and implementing DynSS. Section 3.7 motivates and introduces our second approach, the Middlware for Delay-Tolerant Mobile Ad-Hoc Networks (MaDMAN), while Section 3.8 presents the architecture, and Sections 3.9 and 3.10 present the implementation and evaluation of the same. Finally, Section 3.14 concludes.

3.2 Related Work

Given our goals are (1) to provide applications the semblance of a communication session in a delay-tolerant network and (2) to maximize application performance by dynamically selecting the best end-to-end approach based on applications' requirements and the network conditions, related approaches can be divided into two categories: approaches that seek to provide end-to-end connection semantics in challenged environments and approaches that seek to abstract complex underlying semantics while still exposing expressive protocol behavior.

End-to-End Connectivity in Dynamic Environments. Existing applications often expect end-to-end connectivity, which can be unreasonable in dynamic or unpredictable environments. Transport layer extensions have been developed that enable end-to-end connectivity in cases when it would otherwise be impossible [3, 61]. These approaches are not suitable for the types of networks envisioned in delay-tolerant networking, as applications' connections still timeout if they experience extended disconnections. Other approaches use proxies as delegates to shield applications from disconnection [22, 34]. Such approaches assume every application

will eventually reconnect to a particular central service, making them unsuitable to the highly dynamic and unpredictable delay-tolerant networking environment.

Abstracting Diverse Protocols. In response to the need to evolve network architectures to suit emerging applications, architectures that maintain interfaces for existing applications but expose new functionality have been developed. The Overlay Convergence Architecture for Legacy Applications (OCALA) [24] defines an overlay architecture that allows existing (legacy) applications to continue to function, even if the underlying network architecture and implementation are fundamentally changed. OCALA is similar in spirit to our approach but offers a different end goal, i.e., to enable evaluating new network overlays' support of existing applications. The Overlay Access System for Internetworked Services (Oasis) [33] also attempts to enable legacy applications to continue to function on top of overlays that offer more expressive interfaces to newly introduced applications. In contrast to both OCALA and Oasis, DTN applications such as the one described previously require the ability for the network architecture to dynamically reevaluate the choice of underlying implementation, adapting not only to the application's stated requirements but also to the changing network situation and the changing network path from the source to the destination.

Some DTN solutions define an API that allows applications to choose "traditional" end-to-end connections or DTN-based communication protocols [44]; this choice can be based on information about the availability of end-to-end connectivity [40]. This approach intimately intertwines the two communication approaches, which has the potential to significantly increase the overhead of communication. The Haggle project identified a set of architectural principles underlying the design of *pocket-switched networks*, which encounter many of the same challenges as DTNs [65]. Haggle constructs an unlayered network architecture that focuses on application messages and requirements, enabling applications to be communication

protocol agnostic [65]. Applications running on Haggle hand off *application data units* to the middleware to be forwarded by the best possible forwarding algorithm for the particular message at the particular time. In Haggle the redesign of the network "stack" requires revisiting protocol implementations and specifically tailoring them to be included in Haggle (the prototype described in [65] includes "direct" and "flooding" communication). We argue that a layered approach to integrating multiple communication approaches makes it easier to add new functionality "off-the-shelf."

ParaNets matches our vision, defining a *protocol tree* instead of a single stack in which the application, as the root of the tree, can be supported (potentially simultaneously) by multiple stacks with different capabilities and characteristics [19]. We argue that not only should these networking technologies be run in parallel but they should be capable of being dynamically swapped in support of a single application's long-lived conversation, complete with state commonly associated with network communication sessions. Our approach is also similar in spirit to the MANETKit project [55], which componentizes the behavior of MANET routing protocols and allows multiple protocols to execute in a single network in parallel. MANETKit allows applications in MANETs to dynamically switch the protocol used to support a communication task based on the application's operating conditions.

Design considerations relating to the impact of delay-tolerant network underlays on end-to-end application semantics have also been explored [39]. We move a step further by looking at a specific technical challenge within these new requirements; specifically, we enable the underlying network implementation to intelligently alternate between a traditional MANET implementation and a less reliable DTN implementation. The goal is to provide a single "session" interface to the application and dynamically fill in the nature of the connection based on 1) the conditions and quality of the network and the path(s) to the destination and 2) the application's re-

quirements. Existing applications should be able to use the resulting network stack without modification; "DTN-aware" applications should ultimately be able to tailor their interactions to take full advantage of the added, context-aware functionality.

3.3 Dynamic Stack-Swapping Architecture

The DynSS architecture is based on the idea that the capabilities of a delay-tolerant network are best utilized by a network stack specifically tailored for this new style of network. This implies that DTN-specific PHY, MAC, network, and transport protocols will all need to coordinate to fully utilize the network. While applications for DTNs commonly experience extended periods of high delay, this may not be the common operation, and it is definitely not the only mode of operation. For this reason, we designed a middleware for delay-tolerant applications that allows them to function when delays are high, using delay-tolerant protocols, but also maximizes their performance when delays are not high, using more traditional means of communication. In this way, our approach is not dissimilar to OCALA [24]. What makes DynSS unique is the overarching design decision to allow applications to use both a traditional network stack (e.g., TCP/IP) and the delay-tolerant stack simultaneously and to dynamically swap application connections between the stacks as dictated by changing network conditions. Our ultimate goal is to maintain connection state across these stack transfers to enable seamless migrations (e.g. to prevent retransmissions of delivered data and aid in the parameterization of the new protocols). The DynSS architecture has four distinct parts: (1) a service daemon, or "core" that implements most of the middleware's functionality, (2) a user library that provides applications with an API to access the service daemon, (3) a delay-tolerant router and transport protocol, and (4) a context aggregator to gather information about the network to determine the optimal stack to use for a given connection. The following four subsections describe the functions of each of these

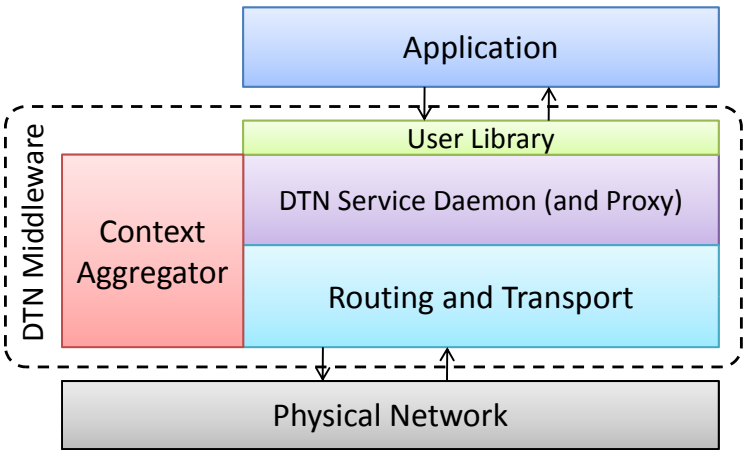


Figure 3.2: Dynamic Stack Swapping Middleware Components

subsystems, and Figure 3.2 illustrates their conceptual relationships.

Service Daemon. The service daemon implements most of the functionality of DynSS and should be viewed as the "core" of the system. It is responsible for interfacing with applications, managing dynamic resources, and negotiating outbound and inbound connections. The service daemon is intended to run on all nodes participating in the delay-tolerant network.

User Library. The user library provides the API by which applications interact with the service daemon and, through it, the delay-tolerant network. It also allows the application to provide a small amount of context to DynSS. We have experimented with application provided data priority as context, which can be used to optimize the available bandwidth or to determine what data can be dropped or rescheduled for later transmission if the network cannot currently handle the throughput.

Routing and Transport Layer. The routing component implements the algorithms and protocols necessary for routing packets, which vary depending on the exact routing protocol used. It is where various existing and future routing

protocols can be plugged in. As described previously, much existing work focuses on creating good DTN routing protocols, and this component approach allows DynSS to selectively take advantage of this work. In an effort to make this easier, we placed "delay-tolerant routers" in user-space, enabling them to interact with the kernel using Linux's built-in netfilter user-space queue. This is how AODV-UU [38] and many other routing protocol implementations interact with the kernel. We have also intentionally separated routing from transport to enable the incorporation of end-to-end semantics specific to the new DTN model of communication.

Context Aggregator. The context aggregator shown in Figure 3.2 is responsible for monitoring and processing network context to accurately characterize the state of the network. Examples of such context include connectivity, throughput, and latency; the context aggregation mechanism is discussed in detail in Chapter 4.

3.4 Evaluation of Dynamic Stack Swapping

We evaluated our dynamic connection migration idea by designing and building a DynSS prototype consisting of a "connection swapping shim" in C++ on the Linux 2.6 kernel. To evaluate whether it is beneficial to dynamically swap the network stack out from under an application connection and to determine the network conditions under which it makes sense to do so, we created a DTN network emulator. The emulator provides a model of a delay-tolerant network that we can use to perform end-to-end connectivity tests using real systems.

3.4.1 DynSS Implementation

The following subsections provide specific implementation details for the various components and provide a step by step example of how an application uses DynSS to set up a connection.

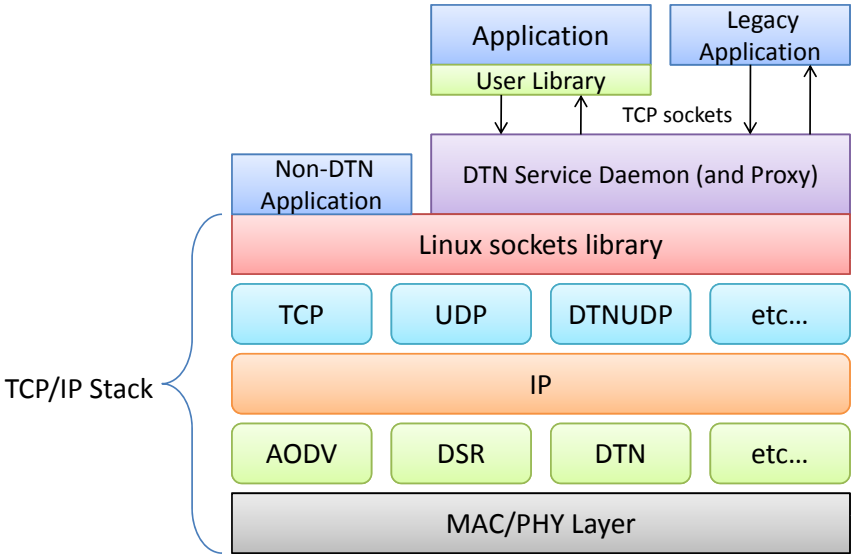


Figure 3.3: Architecture Overview

Service Daemon

The service daemon described in the previous section is implemented as a multi-threaded user-space server, where each thread generally encompasses one component of the DynSS architecture. The service daemon pulls them all together to allow them to be managed in a unified space. As depicted in Figure 3.2, the service daemon interacts with the application, the context aggregator, and the routing and transport components. Through the user library (discussed next), the daemon employs sockets to send and receive application messages. It then uses the Linux socket API to interact with the lower layers. The specific functions of the context aggregator and the routing and transport component are discussed in more detail below.

User Library

The user library is a C library that mimics the API of the C sockets library. Most of the available function calls are enhanced versions of standard socket calls, for example, `write` is augmented with optional priority information. Applications must currently link to our library to use DynSS.

DTN Router and Transport Layer

In our current implementation, we are not concerned with any specific routing protocols, only in making sure that the appropriate hooks are in place to plug different routing protocols into DynSS. As mentioned previously, we implement a generic router using Linux's `libipq` library to queue incoming messages in user-space where the router component can examine, modify, and selectively allow or drop packets. Existing MANET routing protocols and specialized DTN routing protocols can be plugged in to provide different styles of service as the context aggregator deems necessary. With respect to the transport layer protocols, we use a standard TCP implementation for the MANET stack, and our DTN transport layer protocol does not currently implement any functionality beyond what UDP already provides. We refer to this implementation as "UDPDTN."

User Datagram Protocol for Delay-Tolerant Networks (UDPDTN). Because a running TCP connection contains a significant amount of state information, it is important for a DTN transport protocol to be able to take advantage of this information in the best way possible. During a stack transition, DynSS logs information available from one transport protocol (e.g., the last acknowledged data, the window size, the sequence number, etc.) and inputs it into the connection's new transport protocol. The protocol that has been swapped in can use this context information to adjust its own parameters. We have created a very simple first cut at a delay-tolerant transport protocol, which we refer to as the User Datagram

Protocol for delay-tolerant Networks (UDPDTN). UDPDTN is basically UDP with sequence numbers; it uses the last *acknowledgement number* from the TCP stack to bootstrap where in the application data stream it begins sending. Note that this is usually a different point from the last data accepted by the TCP socket (and thus what might be considered "sent" from an application's perspective) since there can be large amounts of data sitting in the low-level TCP buffers that has been transmitted already but not yet acknowledged. The lack of acknowledgement of any given data can mean that the data or the acknowledgement itself was lost. In either case, in order to minimize data loss, UDPDTN starts sending at the position of the last received acknowledgement.

Context Aggregator

The context aggregation engine is implemented as a special thread in the service daemon. We are currently modifying the TCP/IP stack in the Linux kernel to export various data about the network and transport layers into a custom `/proc` file. This file will be read by the context aggregator and used to influence decisions about when to transition from traditional protocols to DTN protocols and vice versa. We also plan on implementing feedback mechanisms whereby the context aggregator can modify the behavior of the TCP/IP and DTN stacks. The `proc` file system is the most logical means to accomplish this. In the implementation used for our feasibility study in the next section, our context aggregator is a very simple component with an omniscient view of when transitions are best made. In Chapter 4 we explore the design and implementation of a comlete context aggregator for delay-tolerant networks.

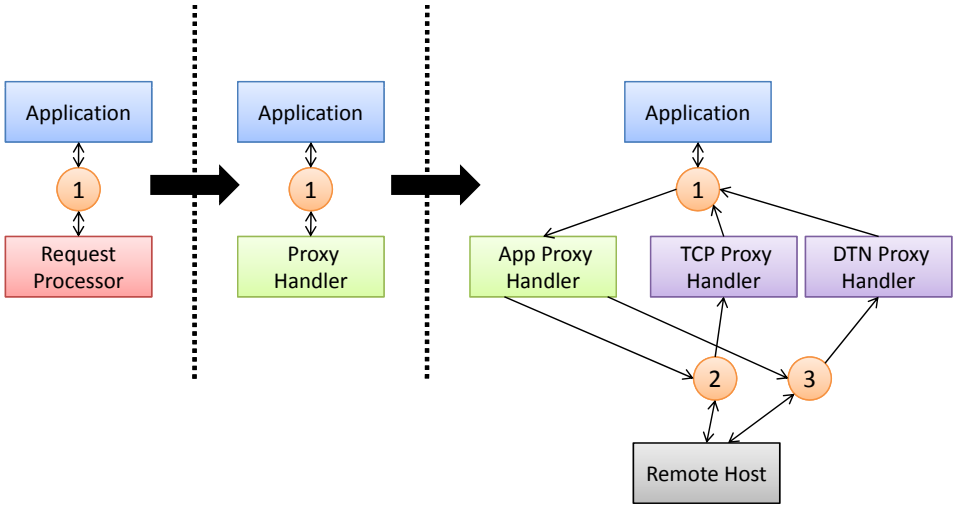


Figure 3.4: Outbound (Proxied) Request

Example connection setup and teardown

DynSS has two modes of operation, one for handling outbound connections to remote DTN-enabled hosts (shown in Figure 3.4), and one for handling inbound connections (shown in Figure 3.5). Applications and middleware components are shown as rectangles, sockets are shown as circles, and arrows show data flow between the sockets and components. To explain how the different components of DynSS interact, we provide a step-by-step description of how an outbound connection is set up. The operations involved in processing the inbound connection are analogous.

1. The application requests an outbound socket using our user library.

2. The library translates the socket request into a socket request to the middleware (using configuration information such as the service daemon's host and port) and sends the request. This socket (labeled '1' in Figure 3.4) is then used to communicate between the application and DynSS.

3. The *request processor* determines that the request is for an outbound connec-

tion, spawns a *proxy handler* thread to handle the connection, and passes the previously created socket to the handler.

4. The proxy handler negotiates the specifics of the connection, which includes reserving resources, connecting to the remote address, and determining priority information.

5. The proxy handler, using input from the context aggregator, opens either an outbound TCP or UDPDTN connection to the remote host and starts forwarding data from the application. These connections are sockets '2' and '3' in Figure 3.4. The proxy handler also receives feedback from the context aggregator over the duration of the connection (not shown) to determine if and when to migrate the connection between stacks.

6. The proxy handler also spawns two threads to listen to the sockets and forward incoming data back to the application. These threads are responsible for ensuring in-order delivery and generating ACKs.

Connection setup for listening services (Figure 3.5) happens in a similar fashion, except that for each incoming connection (regardless of the type) DynSS opens a return connection to the service application.

3.4.2 Delay-Tolerant Network Emulator

Our network emulator is similar to NIST Net [10] and allows us to introduce arbitrary packet delays and packet drops into the network. We apply a delay to every packet using probabilistic distributions, which can be thought of as delay vs. time curves that capture the properties of the delay-tolerant network from the perspective of an end-to-end connection. Our emulator does not model a particular underlying topology nor a particular routing protocol. We simply emulate the

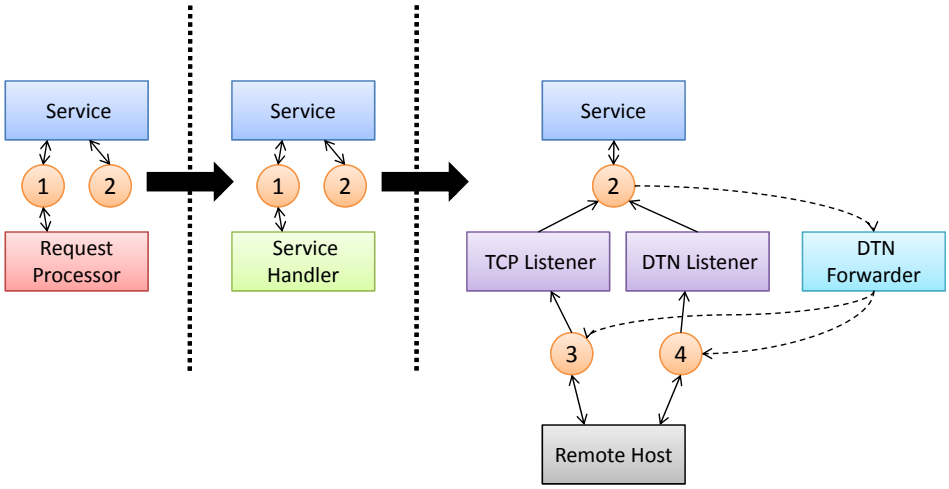


Figure 3.5: Service Request

end-to-end behaviour of a generic delay-tolerant network in order to better understand the trade-offs of dynamically swapping stacks in terms of possible application throughput. Changing the delay vs. time curve allows us to model all end-to-end throughput characteristics of the network. We tried many different delay vs. time curves, and the motivations behind the curves we ultimately selected are provided below.

The emulator does not explicitly model queues or network storage—data loss is modeled using probabilistic packet drops. Each delay vs. time curve also has a drop probability vs. time curve associated with it, and any packet scheduled by the emulator also has a chance to be dropped equal to this probability. In practice, our drop probability is simply a function of the delay. For the purposes of this simulation, we naïvely assume that, as delays increase, so does the probability that a packet traversing the network will get lost, queue dropped, or run into some other problem. Our drop probabilities vary between .01% to 5% and are linear functions of the delay. Since TCP is a reliable protocol, packet drops did not cause data loss, but as can be seen from the results, this reliability comes at a very high price in

environments with hefty delays. UDPDTN in its current iteration does not provide any retransmit capabilities and relies on the best-effort guarantees of the underlying network. As stated previously, this is not an ideal mechanism but leads to low overhead and better throughput, as demonstrated below.

Scenario 1—Short Delay

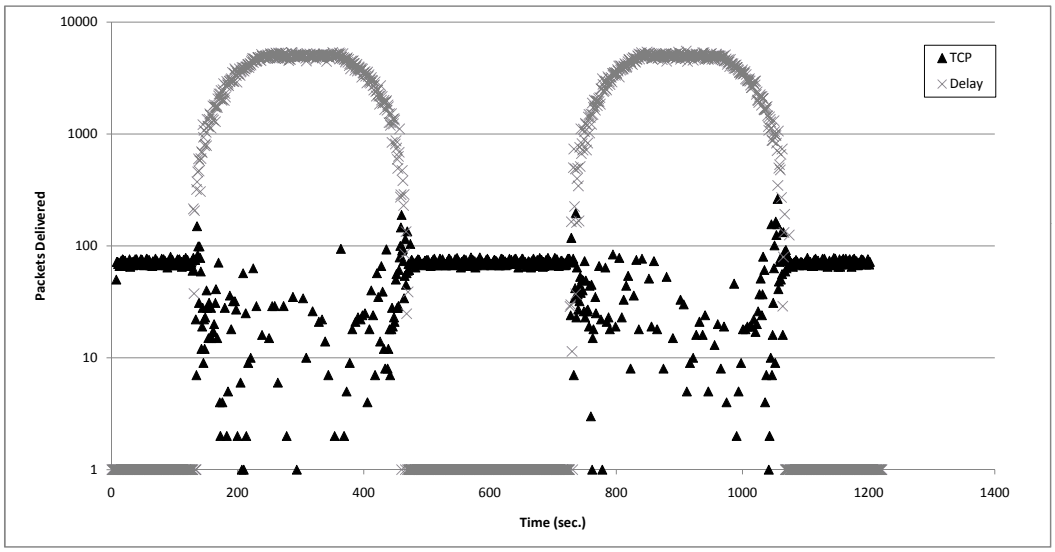
In reporting our results, we use two specific delay scenarios to compare a mixed DTN/reliable delivery protocol swap with a basic TCP-only approach. The first of these scenarios models a situation where a well-connected user wanders away from the well-connected portion of the network and operates in disconnected mode for a short period of time, then wanders back. This is modeled by a rapidly increasing delay from 0 ms to 5000 ms, followed by a period of two minutes where the delay stays around 5000 ms and then comes back down. This cycle happens twice during the course of the simulation, which lasts 20 minutes. This delay curve can be seen in Figure 3.6(a) and (b).

Scenario 2—Long Delay

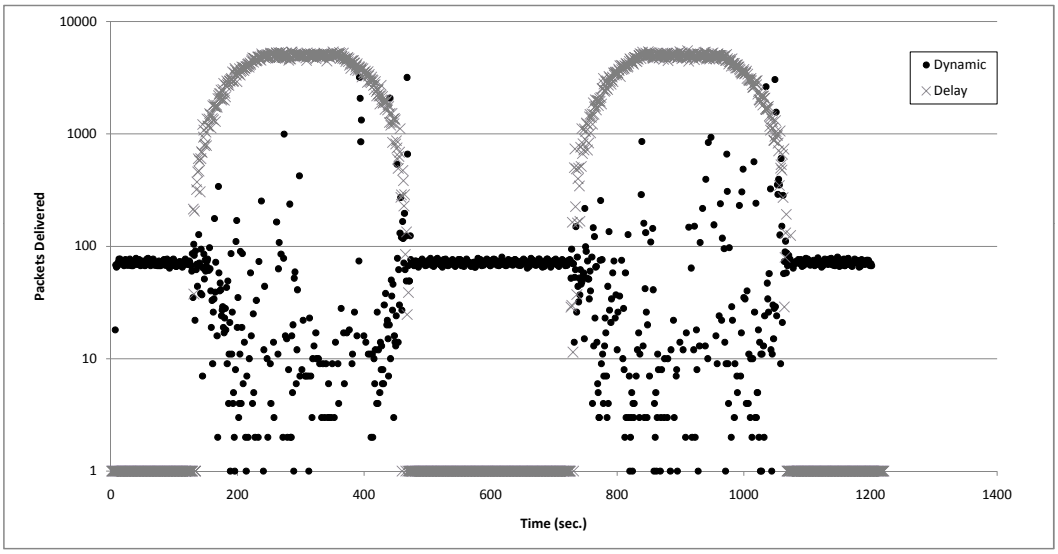
Our second scenario models a situation where a user leaves the well connected portion of the network for a longer period of time and then comes back. The DTN in which the user spends the interim time also experiences lengthier delays. The user spends 10 minutes operating in this high-delay mode where the packet delays average two minutes. This delay curve is depicted in Figure 3.7(a) and (b).

3.5 DynSS Results

For each scenario, we ran three different tests: one using only TCP between the end hosts, one using only UDP, and one using our middleware to swap stacks dynamically

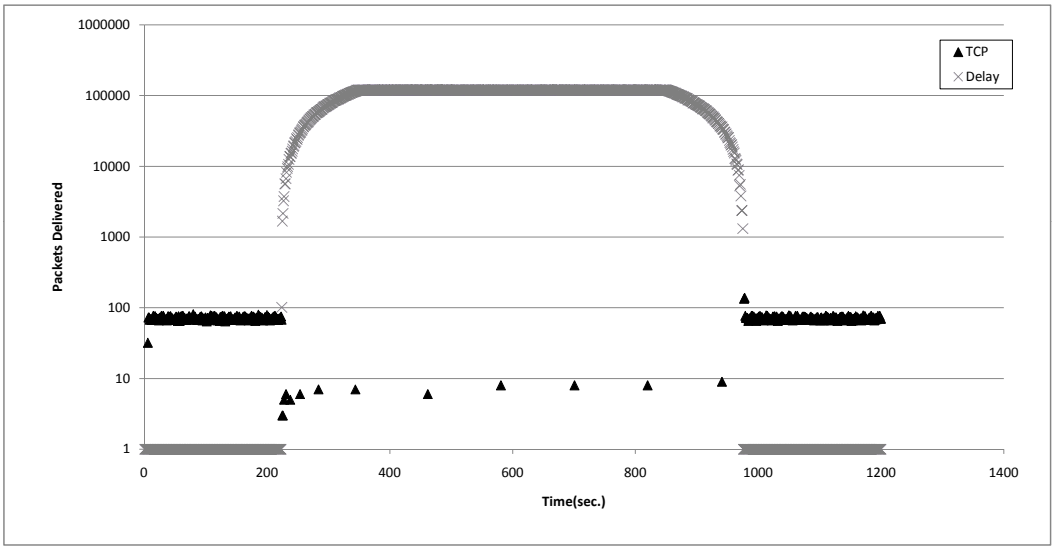


(a) TCP-only over Scenario 1

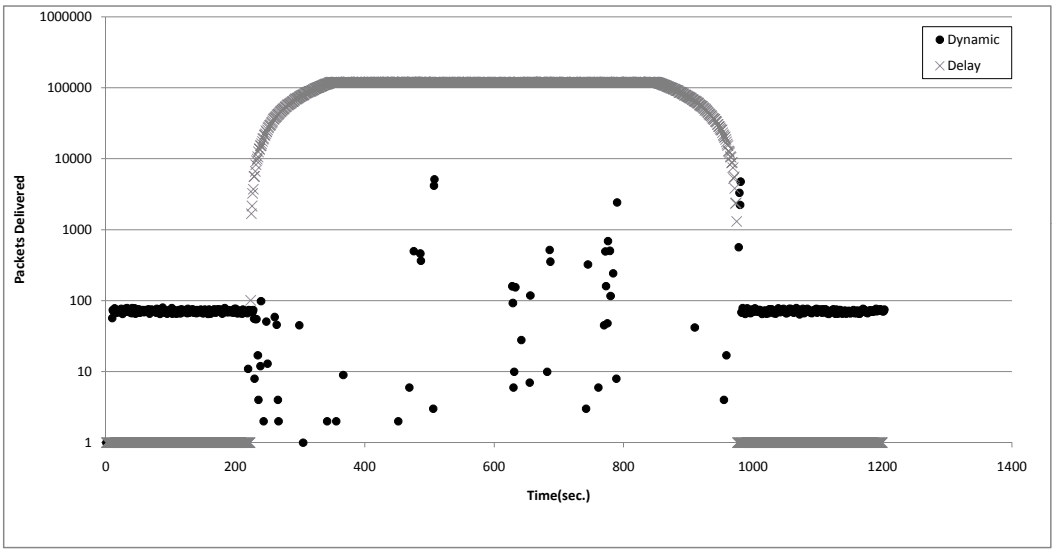


(b) Adaptive TCP/DTN over Scenario 1

Figure 3.6: Throughput vs. Simulation Time for Scenario 1. Each graph shows the delay (in gray and measured in ms) imposed on the packets and the measured throughput (in black) of either the TCP-only approach (a), or the mixed TCP/DTN approach (b).



(a) TCP-only over Scenario 2



(b) Adaptive TCP/DTN over Scenario 2

Figure 3.7: Throughput vs. Simulation Time for Scenario 2. Each graph shows the delay (in gray and measured in ms) imposed on the packets and the measured throughput (in black) of either the TCP-only approach (a), or the mixed TCP/DTN approach (b).

during the connection. We make the assumption that both TCP and UDP can take advantage of the delay-tolerant routing protocol and thus all packets can be routed into and out of disconnected portions of the network. Therefore TCP connections do not break under conventional IP route failures and instead continue operating over the DTN. However, as can be seen from the simulations, TCP throughput suffers during the delayed portions of connectivity and gets progressively worse as the per-packet delays increase. Versions of TCP optimized for ad-hoc wireless networks may perform slightly better, but even those rely on timely acknowledgements, which are often impossible to guarantee in delay-tolerant networks. Since UDP does not have flow control, we had to limit the data rate to avoid overflow behaviour. For this limit, we chose to use the data rate of TCP when TCP is transmitting during the non-delayed portion of the simulation. Our reasoning behind this is simple: TCP increases the data rate until it maximizes the capacity of the network, and the average maximum capacity of the network during disconnected operation will not be higher than the average maximum capacity of the network during fully connected operation. In this way, we run UDP at a very optimistic data rate since we are assuming that the delay tolerant routing and network layers can handle as much traffic as TCP and IP routing layers can handle. The following sections present the results of both tests.

Scenario 1—Short Delay

For the TCP throughput test, the TCP socket buffer was kept full, and TCP was allowed to send as much data as the network allowed. Each packet contained 1kB of data. We used the standard TCP implementation in the Linux 2.6 kernel (TCP CUBIC [57]). The results of the tests are shown in Figure 3.6(a). As expected, the throughput of TCP decreases dramatically during the delayed phases of the simulation, dropping from around 90 packets per second to 10-30 packets per second.

Figure 3.6(b) shows the results of switching to the UDPDTN protocol when the throughput of TCP drops and switching back when the throughput of TCP rises again. The packet arrival rate is very bursty during the 'disconnected' portion of the simulation, but it is easy to see that a large number of packets still arrive during the delayed phases of the simulation. TCP was able to deliver around 47,000 packets with no data loss during the 20 minute run. The middleware, using dynamic stack swapping, was able to deliver around 85,000 packets during the same simulation since the dynamic stack was unconstrained by TCP's flow control mechanism and could progress further into the available data. This resulted in an 80% increase in throughput with 0.01% packet loss (950 lost packets). We did not graph the UDP-only results; they are largely similar to the dynamic results. However, UDP is only a best-effort protocol, so in the periods of low-delay, it cannot take advantage of TCP's added end-to-end guarantees. On the contrary, by dynamically switching between TCP and a UDP-like protocol, UDPDTN garners the advantages of both.

Scenario 2—Long Delay

The results for the second scenario are shown in Figures 3.7(a) and 3.7(b). The results are analogous to those for the first scenario; the disconnected phase is marked by a dramatic drop in TCP throughput. Once again, the dynamic stack switching outperforms TCP-only, this time resulting in 60,000 packets delivered vs. TCP's 31,766. This is a 90% increase in throughput, and it comes at a cost of 1.74% packet loss, with all packet losses occurring during the disconnected phase. We once again omit the UDP-only results for the same reasons as above.

3.6 Conclusions based on DynSS Experience

Dynamic TCP/DTN stack swapping greatly improves network utilization when large delays are present in end-to-end connections. Through the above scenarios, it is

easy to see that in scenarios with longer delays and more time spent operating under "challenged" conditions, the benefits of stack swapping can only increase. Future mobile computing network deployments will demand the ability to dynamically migrate application connections from one communications technology to another. Specifically, as DTNs become commonplace, application components will move between periods of good connectivity, where traditional networking protocols can be employed, and weaker connectivity, where taking advantage of emerging DTN protocols will offer better performance. In building DynSS, we adopted a systems perspective on enabling applications' connections to be seamlessly moved from one network stack to another. By building this architecture in a real operating system and emulating the end-to-end delay characteristics of a delay-tolerant network, we demonstrated that it is not only reasonable, but potentially highly beneficial to employ such an adaptive network architecture.

However, as we discovered, the DynSS approach has a number of drawbacks. In leveraging complete network stack implementations (e.g., the TCP and UDP implementations in the Linux kernel), we are forced to adhere to the common API presented to applications by these implementations. This leaves us with less control over the network stack than we need to accomplish true dynamic stack manipulation. For example, tweaking protocol parameters, which could very well yield further improvements in efficient network utilization, can only be accomplished through limited user-level API calls, and the limited parametrization made available by the kernel through the variables exposed in the `/proc` filesystem. Additional control points would have to be coded directly into the Linux kernel's monolithic TCP/IP codebase. Furthermore, we are forced to use "entire" monolithic network stack implementations thus limiting the granularity at which we can compose custom protocol stacks to support the unique requirements of delay-tolerant networks. It would be far better to utilize a modular implementation of the network stack,

in which each component is exposed, and can be changed, in isolation. This would allow for a greater degree of control over the composition of network stacks. Additionally, there is the issue of integrating a "vertical" cross-layer context aggregator. This would also be simplified by using a network stack implementation that ran in privileged *user-mode* instead of *kernel-mode* since inter-process communication primitives such as sockets and shared memory would be readily available to the programmer. In fact, the context aggregator *should* reside in userspace— this would serve to broaden the context-sensing possibilities of such a component without the risk of introducing unstable code directly into the operating system's kernel. We examine these issues in the next section.

3.7 Middleware for Delay-Tolerant Mobile Ad-Hoc Networks

. The Middleware for Delay-Tolerant Mobile Ad-Hoc Networks (MaDMAN) builds on our initial efforts in dynamic DTN architectures (see DynSS—Section 3.3) and similarly, it allows applications' ongoing connections to seamlessly migrate between two different communication stacks: a traditional mobile ad hoc networking stack (for use in well-connected regions and a dedicated DTN stack (for use in transient communication situations). The DynSS architecture had a number of drawbacks as discussed in Section 3.6; we created MaDMAN to address these. It improves upon DynSS in the following ways:

- MaDMAN uses a modular network stack implementation, allowing for more flexible compositions of protocols. Furthermore, the modularity allows for easier manipulation of both internal and external protocol parameters.
- The MaDMAN protocol stack code-base is based on an external router framework; it is both object-oriented and independent of the Linux kernel, allowing

for cleaner development.

- MaDMAN can run both in user-mode, and kernel-mode as a kernel module, allowing for easier debugging and testing, and better system stability in case of bugs.

- MaDMAN is tightly integrated with the Bundle Protocol, a capable and popular transport protocol designed specifically for delay-tolerant networks. The Bundle Protocol is explained in Section 3.8.

We have implemented MaDMAN's stack-switching capabilities using the Click modular router [36]; we describe this implementation in detail in Section 3.9. The use of Click allows for modular stack compositions and enables future integration with a wide variety of network context measurement capabilities that can influence MaDMAN's understanding of the dynamic environment; this added intelligence will enhance MaDMAN's ability to respond to changes in its environment. We have evaluated this implementation using the autonomous robots of the Pharos testbed [53] (for a description of the testbed and its capabilities, see Chapter 2). The MaDMAN middleware we present in this chapter represents an essential step in realizing the integration of emerging delay-tolerant networks with existing mobile computing capabilities in actual deployments.

3.8 MaDMAN Architecture

The architecture of MaDMAN is largely based on our DynSS work but with a few significant changes. Due to the continuous emergence of new and better approaches to communication, it is essential that MaDMAN enable simple and intuitive integration of new physical, media access, network, transport, and application layer protocols, without requiring these approaches to be redesigned or reimplemented to fit inside the middleware. To this end, we designed MaDMAN to operate within the

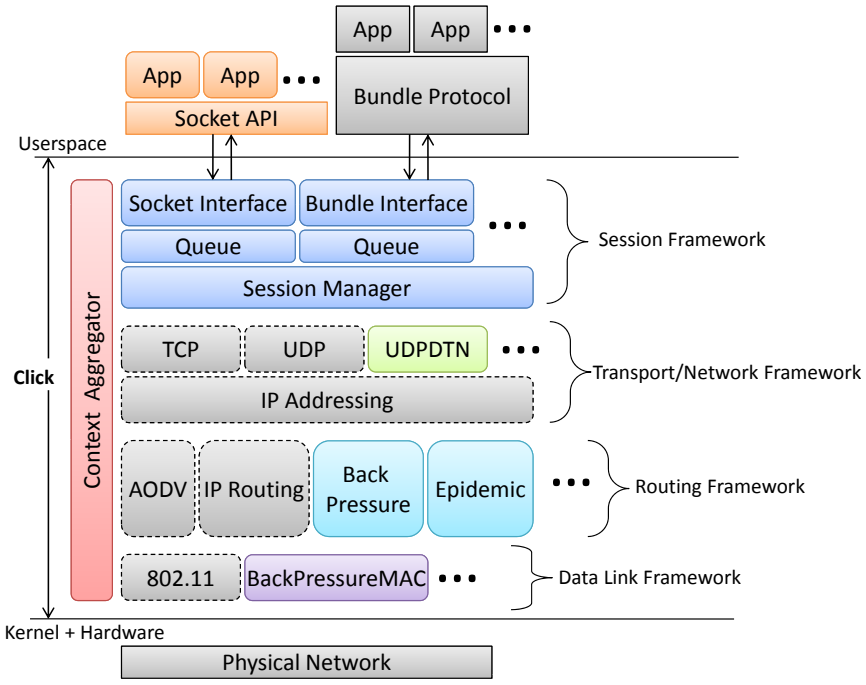


Figure 3.8: The High-Level MaDMAN Architecture

Click Modular Router [36], a flexible, modular, and stable framework for developing routers. Click has a large base of available network protocols and is itself a good framework for implementing experimental protocols. We further discuss our reasons for selecting Click in Section 3.9.1.

Figure 3.8 shows the components of the MaDMAN architecture. The grayed components in the figure (with dotted outlines) are components that already exist within the Click implementation. The main components of the middleware are the *Context Aggregator* and the *Session Manager*. The former handles MaDMAN's understanding of the operating environment's conditions and presents context information to the *Session Manager*, which uses the context information to dynamically select and reselect the best communication strategies. MaDMAN's *Session Framework* includes its various application interfaces and the *Session Manager* itself, which manages all of the application session resources. The application interfaces

themselves (*Socket Interface*, *Bundle Interface*, etc.) implement the inter-process communication between applications and MaDMAN (accomplished with the help of various API libraries). In the remainder of this section, we provide a conceptual overview of MaDMAN's functionality in four parts: 1) the application interface, 2) the connection logic, 3) the transport, network, and routing components, and 4) the context aggregator.

The Bundle Protocol

We have designed MaDMAN to work with the Bundle Protocol [58], a popular application layer protocol designed to support communication in delay-tolerant networks (DTNs). Besides working in these challenged environments, the bundle protocol (BP) provides a generalized model for communication via arbitrary data units called "bundles". One advantage of the bundle protocol is that it abstracts away the network stack and choice of network protocols from the application developer, and instead presents a unified API suitable for use in a variety of environments. This makes the bundle protocol an ideal vehicle for the development, integration, and testing of alternative transport, network, and link layer protocols. The key feature of the bundle protocol that makes this possible is the modular interface to one or more "convergence layers" which act as interfaces between bundles and the network stack. As defined in RFC5050 [58], convergence layers have two responsibilities: to send and receive bundles on behalf of a Bundle Protocol Agent (BPA). Our integration of the reference implementation of the Bundle Protocol and the Click Modular Router, done in order to support the Bundle Protocol in our middleware, is described more fully in Section 3.9.3.

3.8.1 Application Interfaces

MaDMAN's application interface consists of several separate components (*Socket Interface*, and *Bundle Interface* in Figure 3.8). These serve various types of applications, allowing non-Click code to interface with the Click-implemented MaDMAN modules. For example, the *Socket Interface* allows application programmers access to MaDMAN through the use of a standard user library (*SocketAPI* in Figure 3.8) that provides a socket-style interface in the form that network applications programmers are accustomed to. The *Bundle Interface* provides an interface specific to the bundle protocol and serves as a convergence layer for the bundle protocol to allow interoperability with emerging delay-tolerant network applications [58]. These application interfaces and their associated API libraries expose a small amount of context acquisition to the application programmer to facilitate the development of delay-tolerant "aware" applications. Through this interface, applications can optionally provide *data priority* information as context, which MaDMAN's connection intelligence can use to guide decisions to allocate resources to the connection and to optimize the available resources among connections. MaDMAN can also selectively schedule low-priority connections on a lower reliability network stack to free up resources for higher priority connections. Applications can also provide information about future bandwidth requirements and the application's desires for rescheduling transmissions to further help optimize data delivery. The MaDMAN application interface includes a SOCKS proxy to enable legacy applications to entirely bypass the *Socket API* user library. While this prevents these applications from sharing context information with the middleware, it enables reuse and interoperability.

3.8.2 Connection Logic

The connection logic lives inside of the *Session Manager* in Figure 3.8. As part of the *Session Framework*, it sits above the transport layer and is responsible for engaging

the best combination of lower-layer protocols to use for a given connection. The module then passes the application data along to the correct combination of Click modules to establish communication. The *Session Manager* tracks each connection, demultiplexing inbound data to the correct application stream. It also reconfigures the protocol modules used to support each communication session when changes in the environmental or network context dictate an adaptation. The intelligence that uses context information to determine the best combination of network modules to use at any given time resides in the *Context Aggregator* (described below), but the *Session Manager* handles the mechanics of closing and opening remote connections and maintaining and sharing state among protocols' queues.

3.8.3 Transport, Network, and Routing

MaDMAN's network stack, in the traditional sense, consists of all possible transport, network, routing, and data link layers available on a device. There are many possible compositions of these elements into a working stack, only some of which are functional and useful. The possibilities are enumerated by the connection manager, and the elements that can work together to constitute various stacks are determined before MaDMAN applications run. Adding new functionality in these layers simply requires adjusting this enumeration.

MaDMAN's key differentiator is its ability to exchange the network stacks in the middle of an application's communication session. To support this, MaDMAN transfers state between various transport protocols to maintain connection state in the transition process. For example, well-connected networks will rely heavily on TCP connections for reliability and throughput scaling characteristics. When the connectivity becomes less reliable, MaDMAN will exchange this TCP connection for a less reliable delay-tolerant connection based on, for example, epidemic routing. While this exchange loses TCP's reliability guarantees, MaDMAN does maintain

some of the TCP connection's state (e.g., the last acknowledged data packet, the TCP window size, the current sequence number). This information is made available through the *Session Manager* to the other communication protocols. Conversely, when the connection migrates back to the reliable network stack, similar information can be used to bootstrap the new TCP connection with a buffer that reflects the data that was delivered while TCP was unavailable. The decision to switch from one stack to the other can be initiated by either end of the connection. It is a local decision per node, and thus we can end up with nodes communicating using asymmetric protocol stacks. One node could use a certain combination of protocols to transfer data to another node but receive its responses using an entirely different set. The *Session Manager* is responsible for sorting the data into a single stream to send to the application. Out-of-order delivery is a question of policy for the *Session Manager* and can be selected per application session.

3.8.4 Context Aggregator

The *Context Aggregator* is a cross-layer component responsible for gathering network performance data from the various elements in the network stack. Examples include node connectivity, mobility statistics, instantaneous and averaged throughput, latency, and protocol-specific parameters (e.g., TCP window size). The *Context Aggregator* also contains decision processes that use this context information to determine the best combination of low-level modules to use for the applications' active connections. The *Context Aggregator*'s intelligence can be based on complex user-specified policies dependent on the wide variety of available context information (see Chapter 4); for the purposes of evaluating MaDMAN in this chapter, we rely on a simple timing based decision process described in Section 3.9.2.

3.9 MaDMAN Implementation

We have implemented the model described in the previous section using the Click modular router. In this section, we describe our implementation in detail, beginning with some background information on Click.

3.9.1 Background on Click

The MaDMAN middleware, with the exception of the external application library, is implemented as a collection of Click elements within the Click Modular Router [36], whose software architecture is well suited to our goals of adaptability and dynamic reconfigurability. Click is written in C/C++, runs on Linux and BSD, and includes numerous components to manipulate the network stack from the hardware drivers to the transport layer. This frees us from re-implementing common protocols yet offers the flexibility of easily swapping in alternatives or modifying existing protocols without breaking the host operating system's network layer.

Click "routers" consist of *elements* with explicitly defined input and output ports that are "wired" together by configuration files to define a network stack. MaDMAN runs entirely within Click and is thus composed from Click elements using a configuration file. Figure 3.9 shows the configuration of MaDMAN elements in our current implementation. Every element, with the exception of the context aggregator, operates on packets moving up or down the stack. Adjacent modules are connected to each other and can pass packets back and forth. Where drawn, arrows explicitly define the directions in which packets can move; solid arrows represent the passing of packets between modules, and dotted arrows represent element handler calls between elements. We have defined two separate network stacks, a standard TCP/IP stack and a delay-tolerant stack. In the remainder of this section, we describe these MaDMAN middleware components.

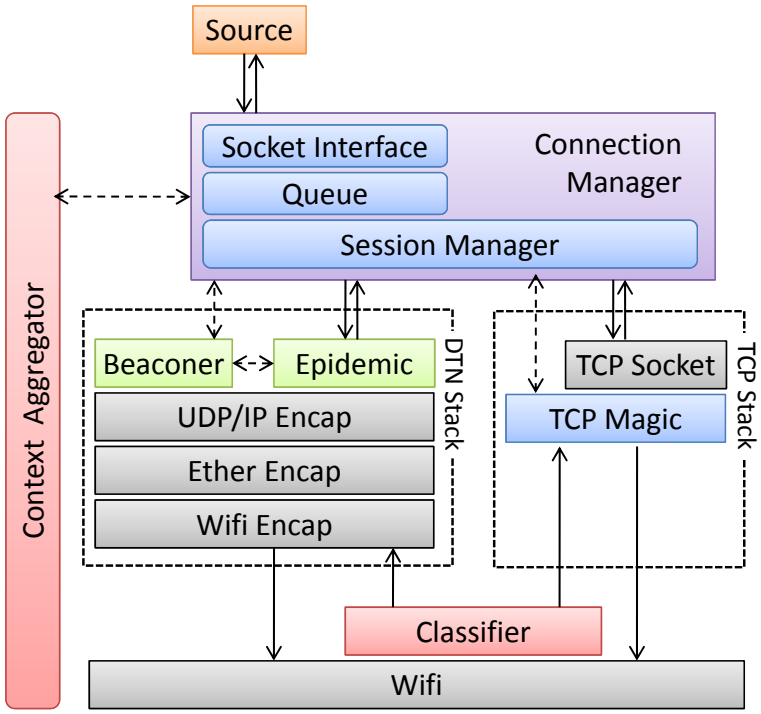


Figure 3.9: MaDMAN Middleware Components

3.9.2 Middleware Components

Figure 3.9 highlights the specific Click middleware components we implemented to demonstrate and validate the MaDMAN middleware. In comparison to Figure 3.8, this figure explicitly separates the elements of the TCP Stack and the DTN Stack. In addition, the functionality provided by the components labeled *Session Framework* in Figure 3.8 is collected into a single Click element, the `Connection Manager`. We discuss these components, the Context Aggregator, and the wiring of MaDMAN elements in the remainder of this section.

TCP Stack

Our TCP stack uses Click's `Socket` element to open and close standard TCP/IP connections. Although we would have preferred composing a TCP stack out of Click elements to enable reading and writing of internal TCP state, these modules are not currently available in the standard Click distribution. Our TCP connections are handled by the Linux kernel's TCP implementation, and, as a result, MaDMAN's TCP stack is the standard Linux TCP stack. Since we needed to collect state information from the TCP connections (window sizes, sequence numbers, acknowledgments, etc.), we wrote a Click element (`TCP Magic`) that parses incoming and outgoing TCP packets and provides state information to the `Connection Manager`. `TCP Magic` is a passive element and does not affect the TCP connections. To track packet deliveries, `TCP Magic` correlates incoming TCP ACKs with outbound data packets to determine how much of the outbound data was delivered successfully. This information is used by the `Connection Manager` to avoid sending data that has already been transferred over an old connection. Connections operating on the TCP stack use standard IP Routing. This stack provides the reliable connectivity that, in our conceptual model, is used when a source and destination have an actual MANET path connecting them (e.g., between devices in the same region in Figure 1.1).

DTN Stack

The delay-tolerant stack consists of our implementation of epidemic routing [67], a popular routing protocol for delay-tolerant networks given its effectiveness and simplicity. In epidemic routing, when two nodes connect, they share *message digests* that contain information about the packets each node carries. Nodes examine the digests and specifically request the packets they have not yet received. In this way, packets are spread around as nodes come into contact with one another, and mobile

nodes can thus ferry data to the disconnected portions of the network as they move.

Our implementation consists of two Click elements, `Beaconer` and `Epidemic`. `Beaconer` sends and receives application layer beacon packets on a user-configurable interval to build a picture of which other nodes are currently connected. The beacon packets are broadcast via UDP, and `Beaconer` issues responses to establish whether functional two-way communication is possible. When it is, a node considers the remote node connected and makes this information available to the `Connection Manager` via a handler call. When `Beaconer` misses a set number of beacons in a row (given by a user-configurable time-out interval), it logically disconnects the node. Although there is clearly an overhead associated with sending application layer beacons, this does have the advantage of establishing application-layer connectivity and in doing so provides a better guarantee that when a node is logically connected, it is actually capable of sending and receiving packets. An alternative approach would have been to use the neighbor table provided by the MAC implementation, however we discovered that, in practice, just because two nodes have discovered each other at the MAC layer does not mean they can achieve any reasonable level of "goodput".

The `Epidemic` element has its own internal queue of packets and holds on to all of the packets it receives in case it encounters a node that has not yet received them. When `Beaconer` discovers a new connection, the `Epidemic` element initiates a message exchange. For the duration of the connection, any new data that the node creates is directly delivered to all of the connected nodes, in addition to being buffered for possible delivery to new contacts. The delay-tolerant stack encapsulates all of the epidemic routing packets in UDP/IP.

Connection Manager

The `Connection Manager` implements the logic described in Section 3.8.2 and is responsible for multiplexing incoming and outgoing connections to the right network stack. It relies on cues from the `Context Aggregator` to know when to swap connections between stacks. The `Connection Manager` tracks delivered data for all of its connections, which it uses to "prime" new stacks so that they continue delivering data where the previous stack left off. In our first iteration of MaDMAN, the `Connection Manager` has two possibilities available to it: the TCP stack and the DTN stack described above, although we foresee having many combinations of lower layer protocols. When triggered to swap from the TCP stack to the DTN stack, the `Connection Manager` starts sending data to the epidemic routing protocol's data buffer at the point where TCP last received an ACK. When triggered to swap back to TCP, the solution is somewhat trickier since epidemic routing has no delivery confirmations. Instead, the `Connection Manager` on the receiving end sends a re-establishment message with the sequence number of the last received epidemic packet, thus allowing the remote node to restart the TCP connection where the DTN stack left off. There is a possibility here for data loss since epidemic routing does not provide guaranteed delivery, but should a reliable delay-tolerant transport protocol be desired, the missing data chunks are enumerated by the `Connection Manager` and could easily be requested from the source so long as it is still buffering the data.

Context Aggregator

In our MaDMAN implementation, the `Context Aggregator` is a simple element that provides cues for the `Connection Manager` to swap from the TCP stack to the DTN stack on a user-provided time-based trigger schedule. This first iteration of the `Context Aggregator` does not collate network statistics or provide an

intelligent stack swapping algorithm (hence the lack of handler connections to it in Figure 3.9), though it is sufficient for the purposes of demonstrating the feasibility of our approach. Our complete Context Agent Framework solution, presented in Chapter 4, addresses all of these issues. When swapping from DTN to TCP, the context aggregator waits for available connections. When the `Beaconer` element reports a direct path to the destination, the `Context Aggregator` initiates a swap back to the TCP stack. A better algorithm would be to trigger the switch from TCP to DTN as soon as the average throughput drops below a certain figure, based on application requirements and data generation rate; the exploration of such smarter connection switching algorithms is the subject of Chapter 4.

Wiring of MaDMAN Elements

Click itself takes a router configuration file as input to determine the types and wiring of all of the elements that comprise a "router," or in our case, the MaDMAN middleware. The advantage is that new configurations are very easy to test, and functionally identical modules can be swapped out without recompiling. For example, in Figure 3.9 the {`TCP Socket / TCP Magic`} combination can be swapped for a plain `Unix Socket` should the user wish to test higher layer functionality using a local data stream instead of a full network stack. The change involves two lines of the router configuration files which must be changed from:

```
source
     -> Socket()
     -> TCPMagic()
```

to:

```
source
     -> UnixSocket()
```

Click checks the gates of all elements to make sure that all gates are connected and nothing is connected twice; nevertheless, it is possible to wire elements together in an order that Click will allow, but that will not function as intended. An evaluation of the MaDMAN is presented in Section 3.10.

3.9.3 Integration with the Bundle Protocol

As mentioned previously, the Bundle Protocol shows great promise as a general purpose application-layer protocol for delay-tolerant networks (DTNs) and has found many adopters within the research community. As an application layer protocol, domain-specific transport, network, routing, and lower-level protocols are also required to deliver bundles between nodes. In the Bundle Protocol specification, the domain-specific protocols that support the Bundle Protocol are collectively referred to as a *convergence layer*. A given system implementation might have several convergence layers to choose from (for example UDP/IP/IP-Routing, TCP/IP/DSR-Routing, TCP/IP/AODV-Routing, etc.) and it is generally the network designers job to choose the correct one. We wanted MaDMAN to be able to act as a *flexible* and *reconfigurable* convergence layer for DTNs, thus allowing for the Bundle Protocol as one of many possible application layer protocols. We provide such an implementation, dubbed the Click Convergence Layer [51] for the popular DTN2 Bundle Protocol reference implementation [8] and present a performance comparison between our Click-based convergence layer and native DTN2 convergence layers in Section 3.10. The following presents the design and implementation of the Click Convergence Layer.

Designing the Click convergence layer (ClickCL) was challenging for a number of reasons. First, we needed to understand the interface between the DTN2 and its convergence layers, which in practice is not as clean as RFC5050 would lead one to believe. Additionally, unlike monolithic convergence layers which exist entirely

within the DTN2 codebase, ours must interface DTN2 to a separate process (since Click runs as a standalone process). Therefore we needed to design a protocol to interface DTN2 (itself a multi-threaded process) and Click by means of inter-process communication. We also needed to split the functionality of the convergence layer between the Click Convergence Layer Adapter (ClickCLA) which runs in DTN2 and Click itself.

Another fundamental problem was deciding what services to include in the interface between the DTN2 and Click. The formal requirements put on any convergence layer adapter regarding the sending and receiving of bundles are very open-ended, and many convergence layers provide additional services. These additional features lead to more elaborate integration of the CLA with the DTN2 implementation. Since we wanted to ensure that any conceivable convergence layer could be built using the ClickCL, we needed to design an open-ended interface that could be easily extended. To accomplish this we designed a control protocol to carry such metadata between Click and DTN2. To demonstrate the viability of this implementation, we present an example convergence layer using the ClickCL, the details of which are covered next.

Since Click does not provide such a well-documented API to external applications, the Click Convergence Layer Adapter within DTN2 (ClickCLA) cannot call Click's functions directly. Instead, the ClickCLA must transfer the bundles to Click by other means of inter-process communication. We designate two separate channels to transfer bundles between the bundle daemon and Click: a control channel, which carries control messages and information about incoming and outgoing bundles, and a shared memory "channel" for the transfer the actual bundles. We chose POSIX shared memory mapping to transmit bundles since they can be up to 2GB in size according to the bundle protocol specification, and it is the most efficient way to handle such large data chunks. This design, shown in Figure 3.10, puts an extra

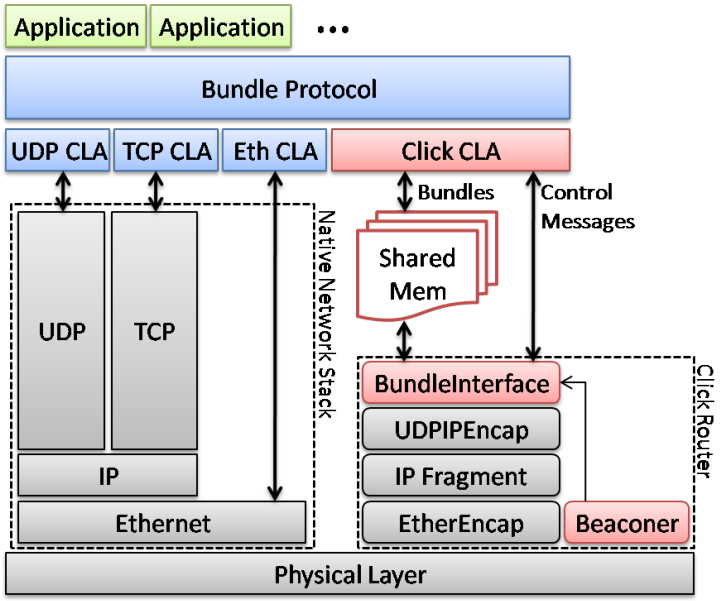


Figure 3.10: Architecture of the Click Convergence Layer

burden on our Click module to recognize the bundle format and primary bundle block header. DTN2 provides two calls, `produce()` and `consume()`, to receive and send bundles in increments appropriately chosen by the convergence layer module without the convergence layer having to worry about the format or content of the bundle.

In order to test the Click Covergence Layer, we implemented a custom MaDMAN configuration. For the purposes of testing against an existing DTN2 convergence layer, we constructed a UDP stack entirely in Click to compare against DTN2's internal UDP Convergence Layer (UDPCL), which itself uses the native Linux UDP implementation. Figure 3.10 depicts our MaDMAN stack configuration to test the ClickCL module. The grey elements in the figure (`UDPIPEncap`, `IP Fragment`, and `EtherEncap`) are all elements available in the default Click installation. Together, they package all incoming packets into a UDP datagram and send

it out via the wireless interface. In addition to using these elements, we have added the `BundleInterface`, which passes bundles and control messages to and from the bundle daemon, and the `Beaconer`, which provides neighbor discovery through the use of beacons from which a table of currently connected nodes is built. This beaconing feature is not provided by the native UDPCL.

The Click Convergence Layer Adapter

A Convergence Layer Adapter passes outgoing bundles onto the network and receives incoming bundles from the network. ClickCLA is the DTN2's interface to Click and provides the functionality of a generic convergence layer. We defined a custom control protocol to communicate between the ClickCLA and the `BundleInterface`. The interface currently supports four types of control messages:

BUNDLE_READY, BUNDLE_SENT, LINK_UP, and LINK_DOWN.

For our implementation, we chose to use the Linux universal TUN/TAP interface to transmit the control messages between the `BundleInterface` and the ClickCLA in the bundle daemon. To illustrate the functionality of the Click Convergence Layer, we describe the process of sending one bundle between two nodes **A** and **B**.

- **A** generates a bundle destined for **B**, but since they are not connected, the bundle gets queued by the bundle daemon
- **B** moves in range and **A**'s `Beaconer` discovers **B**
- The `Beaconer` notifies the `BundleInterface` of a new link
- The `BundleInterface` generates a LINK_UP control message and sends it to ClickCLA

- The ClickCLA creates a link in the bundle daemon for node B and adds a next-hop route to the bundle daemon's routing table
- **A** sends the bundle destined for **B** out on the newly available link through the ClickCLA
- The ClickCLA copies the bundle into a shared memory block and sends the block's identifier in a BUNDLE_READY control message to the `BundleInterface`
- The `BundleInterface` receives the BUNDLE_READY message, processes the bundle into a Click packet, and passes it down to the `UDPIPEncap`, which encapsulates the bundle in a UDP frame, and again in an IP frame
- The `IPFragmenter` fragments the packet into MTU-sized chunks, adds its own headers, and passes it to the `EtherEncap` element to encapsulate the fragments into an Ethernet frame
- `EtherEncap` passes the ready packets to the wireless driver, which sends it on the network

The IP fragments are then reassembled, the headers are stripped, and the bundle is delivered by **B**'s Click instance to **B**'s bundle daemon in a similar fashion. When the two nodes part ways, the `Beaconer` times out their connection and LINK_DOWN messages are generated by both parties to disable the links and remove the routes. For the purposes of sending and receiving bundles, the functionality of our Click-based UDP convergence layer is identical to that of the native UDP convergence layer, and thus they are interoperable with one another. Additionally, the Click-based UDPCL is really just a particular configuration of MaDMAN, and thus is completely modular and easy to modify, add, or remove functionality from. In contrast, one would have to delve into the UDP implementation within the Linux kernel to modify the native UDPCL. Changing any Click-based convergence layer

requires only simple manipulation of existing elements or inclusion of new custom elements in the processing stream.

As mentioned previously, we evaluated the performance of this new convergence layer for the DTN2 reference implementation against an internal convergence layer in order to show that the associated shared memory and inter-process communication overhead is within acceptable bounds for operation in real-world delay-tolerant networks. This evaluation is covered next in Section 3.10.

3.10 MaDMAN Evaluation

This section covers the validation of MaDMAN using the Pharos mobile computing testbed [53]. Chapter 2 provides an overview of the Pharos testbed itself, whereas the hardware and software configuration used for these evaluations are covered below. The section is presented in two parts; first we provide an evaluation of the Bundle Protocol convergence layer implementation described above, and second we present an evaluation of the main MaDMAN implementation. Both are relevant if the MaDMAN-style architecture is to be accepted as a viable alternative to existing monolithic delay-tolerant network middlwares.

3.10.1 Evaluating the Click Convergence Layer for the Bundle Protocol Reference Implementation

To evaluate our Click convergence layer, we sought to do a performance test against a native convergence layer. We built a UDP convergence layer in Click that is analogous to the UDP convergence layer in the bundle daemon and tested the performance of each in both a wireless and wired environment.

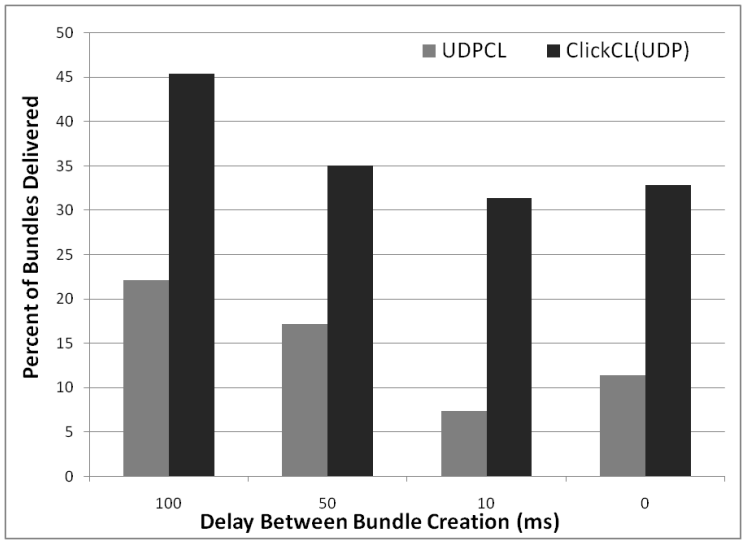


Figure 3.11: Bundle Delivery Ratios

Wireless Experiments

Our wireless experimental setup consists of two nodes with VIA NR10kEG nano-ITX motherboards, 1GHz VIA C7 processors, and 1GB DDR2 RAM each equipped with Atheros 802.11bg wireless cards. The computers themselves are identical to the computers on the Proteus nodes of the Pharos Mobile Computing Testbed. In all wireless tests, the nodes were placed 10 feet apart in an indoor lab space.

We used the **dtnsource** and **dtnsink** applications to generate fixed size bundles at regular intervals and record bundle receptions at the destination. Both programs are distributed with DTN2. Though the UDPCL encapsulates bundles in UDP datagrams, which have a maximum size of 65kB, we found that using anything larger than 48kB bundles resulted in a throughput of close to 0 due to the high probability that at least one fragment of the complete UDP frame would be lost during transfer. Therefore we used 48kB bundles in all of our tests.

We ran several tests with the source creating bundles with 100ms, per 50ms, per 10ms, and 0ms pauses between bundles. Figure 3.11 shows the delivery ratios

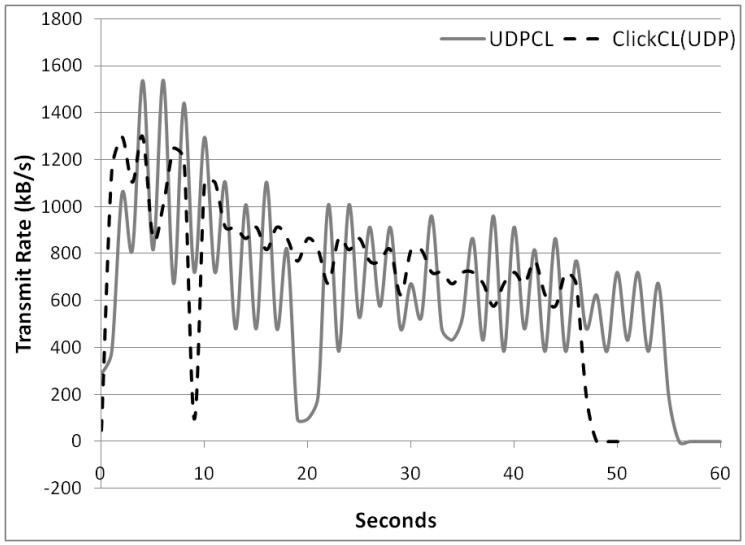


Figure 3.12: Transmission Rates in kB/s

for one set of such tests. Our results were consistent between different tests. In all cases, the Click convergence layer implementing UDP, denoted ClickCL(UDP) in the graphs, delivered a larger percentage of the bundles than the native UDP convergence layer. We sought to understand why this might the case, since the performance should have been nearly identical. We discovered that the ClickCL(UDP) has a much less bursty transmission rate at the wireless card than the native UDPCL, which we suspect contributes to the successful delivery and reassembly of more UDP datagrams.

Figure 3.12 shows the transmission rates at the wireless card for an experiment in which dtnsource was not throttled. We observed this phenomenon in all of the tests, but it was especially evident when the bundle creation rate was not throttled. We do not plot the receiver's data rate since it is nearly identical to the transmitter's. We hypothesize that Click's smoother transmission rate is a side-effect of the extra processing it must do over the native UDP implementation of the kernel and the polling nature of the Click wireless device driver interface. This

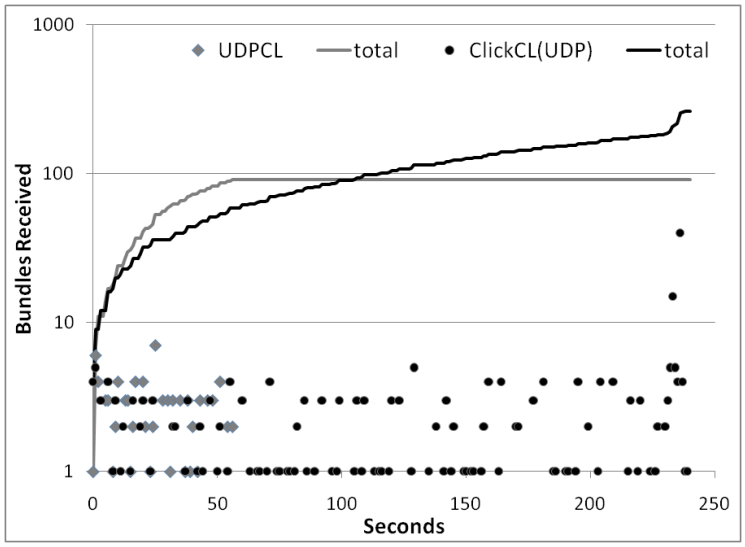


Figure 3.13: Bundle Delivery Latencies (800 48kB Bundles)

seems to have a positive effect on packet delivery ratios when using unreliable protocols over lossy channels. We do not claim this as a contribution of our work, but it is an interesting result nonetheless.

We also used dtnsink to study bundle delivery latency. Figure 3.13 shows the number of bundles received by dtnsink every second for the same test as Figure 3.12. The diamonds and circles indicate the number of bundles delivered for UDPCL and ClickCL(UDP) respectively, and the lines indicate the total number of bundles received up to that point in the test. Note the logarithmic y-axis. This graph shows that DTN2 with UDPCL delivers bundles to dtnsink about as fast as the source can send them, with little delay. The ClickCL(UDP) has a much longer latency for processing incoming bundles and delivering them to the dtnsink application. In fact, as you can see in Figure 3.12, the transmission of packets (and the receipt of them at the receiver's wireless card) is finished approximately 60 seconds into the test, whereas ClickCL(UDP) continues to deliver bundles to dtnsink until around 250 seconds. However, due to its less-bursty transmission rate, the ClickCL(UDP)

convergence layer was able to deliver more of the 48kB bundles.

We discovered that incoming bundles, although processed very quickly by Click, and copied very quickly to shared memory, were experiencing long delays because of the design of the ClickCLA in the bundle daemon, which spent a lot of time working through the queue of control packets to process all of the available bundles. We suspect that this is due mostly to our decision to encapsulate control packets into Ethernet frames and pass them between the ClickCLA and the `BundleInterface` in click via the TUN/TAP device in Linux. Though this seemed like a simple way to accomplish the exchange of control packets, and allowed us to reuse existing code from other DTN2 convergence layers, it introduces unnecessary delays in the processing of incoming bundle data while the frames are delivered by the TUN/TAP device and processed by the ClickCLA.

Wired Experiments

Our wired experiments were designed to test the maximum performance of DTN2 with the two UDP convergence layers. We had reason to believe that the act of sending and receiving bundles through the API incurs a non-trivial amount of overhead, so we designed the experiment to isolate the performance of the CL independent of the API. Our results led to some interesting discoveries about what happens when convergence layers are overloaded. Our experimental setup consisted of seven nodes: three sources, three sinks, and one central "choke-point" or "hub" node as illustrated in Figure 3.14.

Since the central hub node does not have to pass bundles through the DTN2 API, its performance is only a function of the CL's ability to send and receive bundles, and the BPA's ability to process them. Because it handles three unthrottled flows of bundles at once, we are sure to test the maximum performance of the stack. In this experiment we used Dell Studio Hybrids with Intel T8100 2.1 GHz

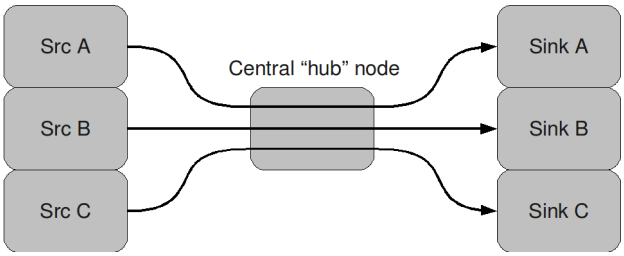


Figure 3.14: Wired ClickCL Evaluation Setup

CPUs connected to a single Gigabit Ethernet switch. We used the **dtnsource** and **dtnsink** applications to generate and dispose of bundles. In this experiment, each source generated 2000 48kB bundles destined for its sink.

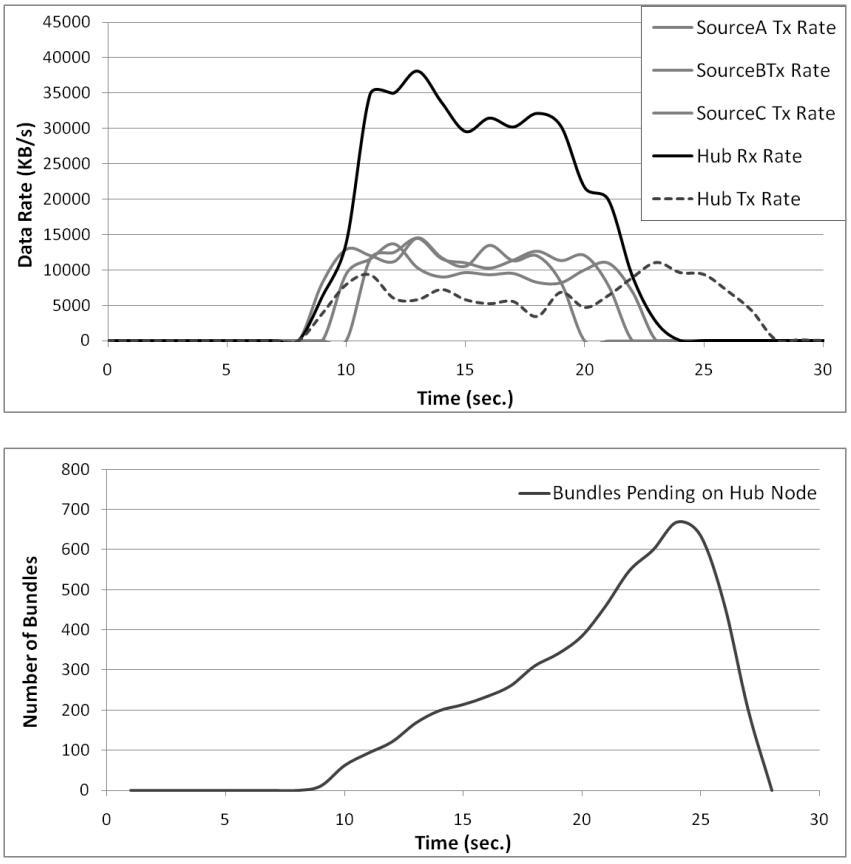


Figure 3.15: Three Flow DTN2 Performance Evaluation on Gigabit Ethernet

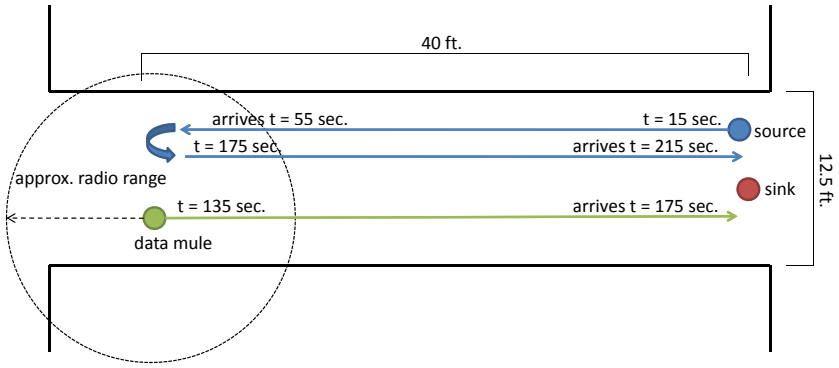


Figure 3.16: Experiment Scenario

Figure 6.14 illustrates the performance of the native DTN2 UDPCL in a typical experiment. The upper plot shows the transmit rate of the three source nodes and the receive and transmit rates of the hub node. We immediately notice that the hub node is receiving bundles at a much higher rate than it is transmitting them. This leads to a backlog of bundles on the hub node. We also notice that the hub's transmit rate is higher when it is done receiving bundles from the sources. What is surprising is that the hub node's pending bundles list continues to grow for at least several seconds after the sources stop transmitting. This indicates that a considerable backlog of incoming data is queued by the UDP stack, and this data is processed as incoming bundles are accepted by the BPA.

Running the same test using the ClickCL(UDP) on all the nodes, we observed about a 4x degradation in throughput for these high-speed wired tests. As observed in our wireless tests, the cause seems to be that when the stack is heavily loaded, our control messages between the Click CLA and Click are delayed considerably.

3.10.2 MaDMAN Evaluation

To evaluate the MaDMAN implementation described in Section 3.9, we used the Pharos testbed. The Click MaDMAN implementation and the configuration files

Figure 3.17: The Experimental Setup

we used for these experiments are available at `http://mpc.ece.utexas.edu/madman`. We used three of the Proteus nodes shown in Figure 2.2, whose mobility is provided by a customized Traxxas Stampede (see Section 2.1.1 for a detailed description of the hardware setup). Although the robots can navigate using GPS, we performed these experiments indoors on a smaller scale for ease of control and reproducibility. We relied on the Proteus's ability to move (fairly) straight on smooth surfaces.

We used the three nodes to create a small-scale delay-tolerant network; the spatial and timing characteristics of the experimental environment are shown in Figure 3.16. To ensure node disconnections (and re-connections) in a 40 foot hallway, we had to attenuate the radios' transmit power. We accomplished this through a combination of a modified `ipw2200` Linux Intel wifi driver that allowed us to set the transmit power of the wireless cards to 3dBm and aluminum foil around the antennae. The combination gave us a connection range of between five and fifteen feet, depending on the channel characteristics; for our experiments, we were able

to reliably establish connections within five feet and to guarantee a node moving down the hallway would eventually disconnect from one at the end of the hallway. Each experiment consists of a stationary data sink, a mobile data source, and a mobile data mule. Only the mobile data source creates data packets. Specifically, MaDMAN's `Source` module (see Figure 3.9) generated 1448B packets at an experiment-specific rate. In all experiments, epidemic beacon packets of size 64B were sent every 200 milliseconds, and the epidemic disconnection time-out interval was set to 700 milliseconds.

3.10.3 Mobility Pattern

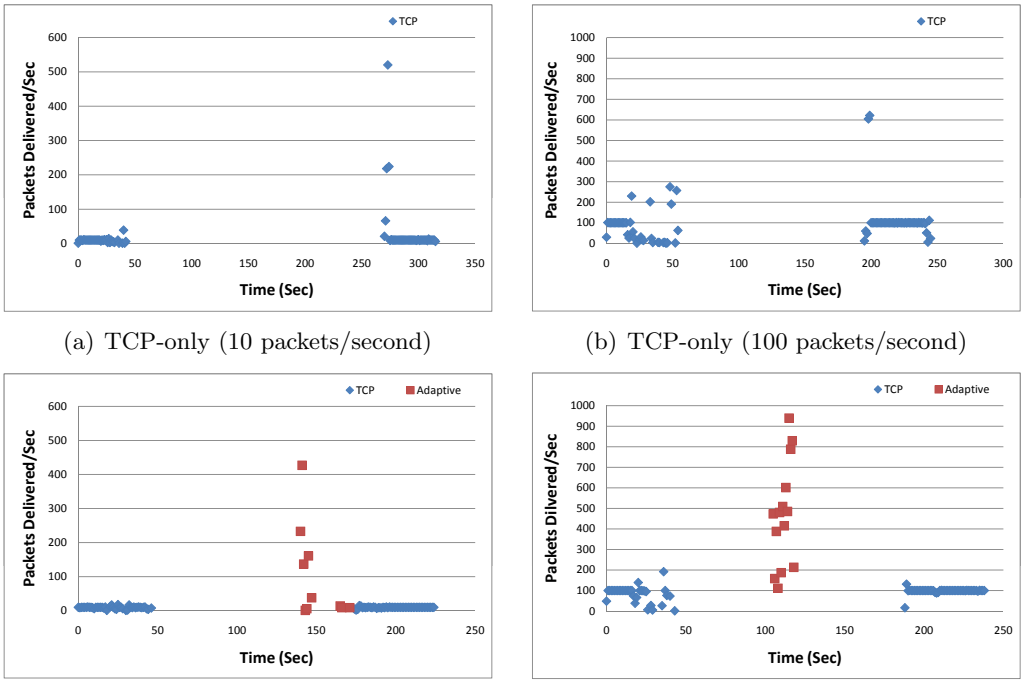
We used the same mobility pattern in all of the results we discuss in this paper. At the beginning of the experiment, both the mobile data source and the stationary sink nodes are next to each other and connected. In this situation, MaDMAN chooses to send all of the data packets generated by the `Source` application via TCP. At $t = 15$ seconds, the source node begins moving towards the data mule at the other end of the hallway at a speed of 1 foot/second. It eventually disconnects from the sink node; due to the reduced quality of data delivery, MaDMAN swaps the connection underpinnings from using TCP over a reliable routing stack to using UDP over an epidemic routing stack. Around $t = 55$ seconds, the mobile data source arrives at the other end of the hallway and connects to the mobile data mule. The source node generates packets at the specified rate for the entire duration of the experiment; because the application's connection is engaged in epidemic routing, the mobile data source sends queued packets to the mobile data mule with the optimistic hope that it will encounter the data sink before the source is connected to it again. At $t = 135$ seconds, the mobile data mule begins to move in the direction of the stationary data sink, again at a speed of 1 foot/second. The mobile source node follows 40 seconds later at $t = 175$. When the mobile data source reconnects to the sink directly,

MaDMAN responds by switching the connection's supporting protocol stack back to the reliable TCP one. The state from the delay-tolerant network stack is used to bootstrap the new TCP connection, which ultimately continues the application's communication session from the last packet the stationary data sink received from the mobile data mule.

3.10.4 Results

In this section, we provide results for two sets of experiments executed using the setup described above. These two experiments differ in the rate with which packets are generated by the `Source` module; the first experiment generated 1448B packets at a fixed rate of 10 packets per second, while the second experiment generated 1448B packets at a fixed rate of 100 packets per second. For simplicity in viewing the results, we show results from only one run each of these experiments; results were consistent across multiple runs. Each experiment was run once using only the reliable TCP network stack without any intelligent swapping of the underlying communication protocol structure and once with MaDMAN's stack swapping capabilities in place, with the connection support changing as described in the previous section. Figure 3.18 shows the results of these experiments. In the figures, the diamonds represent packets delivered by the reliable TCP-based stack, and the squares are packets delivered by the delay-tolerant UDP stack with epidemic routing.

MaDMAN enables "early" delivery of disconnected data. In the first experiment, the data rate was slow enough that both approaches were able to deliver all of the data the application generated by the end of the experiment. In this case the adaptive nature of MaDMAN does not improve the packet delivery ratio or the throughput as measured over the entire experiment window. However, as Figure 3.18(c) shows, MaDMAN's adaptive stacks do deliver a set of packets early, when the mobile data mule reaches the stationary data sink in advance of the



(a) TCP-only (10 packets/second)

(b) TCP-only (100 packets/second)

(c) Adaptive (10 packets/second)

(d) Adaptive (100 packets/second)

Figure 3.18: Packets delivered vs. Time

mobile data source's return. Specifically, the data mule delivered 1035 data packets approximately 40 seconds in advance of the mobile data source's return to connectivity with the stationary data sink (as indicated by the peak in the adaptive data around 150 seconds into the experiment). In general, this result demonstrates that, while using a reliable connectivity infrastructure is ideal when possible, the use of an opportunistic communication structure when the reliable one becomes unavailable can improve the delay incurred in the delivery of application data packets.

MaDMAN improves application-level throughput through opportunistic delivery. In the second experiment, the `Source` module in MaDMAN generated approximately 150KB of data per second (100 packets of size 1448B per second). This data rate was high enough that the TCP connection alone could

not reliably deliver all of the generated application data. Figure 3.18(b) shows the results when only the reliable TCP stack was employed; in this experiment, the sink node received only 38% of the total generated application data. The remainder of the data was lost almost exclusively due to buffer overflows resulting from the high rate of data generation. On the other hand, Figure 3.18(d) shows the results when MaDMAN adapted the communication implementation in response to the experiment's changing operating conditions. In this case, the two modes of communication combined to deliver a total of 65% of the total generated application data, a significant improvement over TCP alone. The degree to which epidemic delivery of packets by a data mule can improve throughput is heavily dependent on the sizes of the epidemic data buffers, which in the case of this experiment were not a performance-limiting factor. In general, this experiment's results show that intelligently swapping the communication implementation in a mixed delay-tolerant mobile ad hoc network can have a significant impact on the total throughput of application data.

3.11 Conclusions based on MaDMAN Experiments

Our two part evaluation of MaDMAN in the Pharos testbed has shown that the benefits of dynamically adjusting the communication paradigm that supports an application's ongoing communication sessions include decreasing the overall delay and increasing throughput. In this section, we discuss a few points that arose in the execution of our evaluation.

First, we have demonstrated that our Click-based implementation of the MaDMAN middleware can easily run on commodity hardware. MaDMAN's modular design ensures its independence from any particular protocol implementations. We have demonstrated the middleware using Click's interface to the TCP implementation in the Linux stack and our own implementation of the popular delay-tolerant

epidemic routing protocol. Other protocols implemented within Click or with interfaces to Click can be integrated in a similar manner. We have also demonstrated that MaDMAN interoperatures with the Bundle Protocol by showing that it can act as a fully capable convergence layer for the Bundle Protocol Reference Implementation. This is an important objective if the MaDMAN architecture is to be accepted as a viable architecture by the delay-tolerant network community.

At a more detailed level, we found that when we pushed the packet generation rate much beyond 100 packets per second, the results became inconsistent between runs due to internal buffer limitations within the Click framework; the buffers that hold data destined from one internal element to another experienced queue-drops due to the latency of handling the data load within Click itself. We also found a wide variation in the time it took for two nodes to reconnect to each other once they were within communication range. In some extreme cases, the nodes required up to a minute after entering each other's radio range to establish link-level communication. We are confident that these are not bugs in the MaDMAN implementation, but external limitations due to our choice of framework (Click) and the Linux ad-hoc device driver implementations.

In the testbed evaluation, our experiments showed positive results with respect to two metrics: 1) we enabled early delivery of application data that was generated in the disconnected state, and 2) we increased the total throughput in high traffic situations by opportunistically delivering data when the source and sink were not directly connected. We were able to show these significant benefits even given the simplistic nature of the experimental setup. Specifically, the mobile data source remained disconnected from the stationary data sink only for about three and a half minutes; it is easy to imagine situations where the source node is operating in the opportunistic mode for much longer, in which case MaDMAN's benefits will be amplified. In addition, we used only a single data mule to deliver opportunistic

data. In larger mixed delay-tolerant mobile ad hoc networks, it is likely that several mobile nodes will be able to ferry opportunistic data among sources and sinks, which would also increase the benefits of the MaDMAN middleware.

3.12 Research Contributions

This chapter makes the following research contribution:

Research Task 1: We develop an architecture for dynamic connection migration in delay-tolerant networks and demonstrate its utility on a real system. We designed and built two prototypes to examine the benefits and trade-offs using two separate evaluation environments (network emulation using real hardware and software, and real world, small-scale, indoor testbed evaluation using autonomous robots). These two architectures and subsequent prototypes allow us to focus our efforts in two important ways. First, they offer insight about what types of network stack adaptations are beneficial, and even *where* in the networks stack the adaptations should be implemented to most benefit delay-tolerant network systems with the least overall system complexity. Second, they focus our efforts in designing context sensing and aggregation strategies by offering insight into how such software should integrate with the system.

3.13 Impact

Our work in this chapter is, to our knowledge, the first to define an architecture for general-purpose context-based network stack adaptation for delay-tolerant networks. Context-informed stack reconfiguration using modular stack elements has not been attempted for delay-tolerant network stack compositions, and these experiments represent the first results in this space. We were also the first to define a general purpose convergence layer interface between the DTN2 Reference Implementation

and the Click Modular Router [51]—this particular work led to a *best paper* award at the 2010 Conference on Extreme Communication, a delay-tolerant networking conference.

3.14 Chapter Summary

This chapter presented two unique architectures, the DynSS network stack-swapping concept and the the MaDMAN architecture. Both enable adaptive communication infrastructure in mixed delay-tolerant and mobile ad hoc networks. To handle and take advantage of the changing operating conditions that applications in these dynamic environments experience, we enable the intelligent exchange of the protocol stacks that implement a communication session in the middle of an ongoing session. Both DynSS and MaDMAN accomplish this in ways that are seamless from the application's perspective. Our treatment of delay-tolerant networks as changing environments that must be supported by a combination of traditional ad-hoc networking protocols and delay-tolerant network-specific protocols is, to our knowledge, unique. The approaches presented in this chapter are a fundamental step in smoothly integrating DTN technologies and benefits into our existing mobile computing infrastructure. They prove—from a systems perspective—that such adaptations are beneficial and possible using commodity hardware and furthermore that they can be integrated with existing work in DTNs.

However, this is not yet a complete solution. The information used to trigger the stack swapping or re-composition must be sensed from the DTN environment. So far we have assumed an omniscient perspective with respect to the points at which the transition between protocols should occur. How this can be actually accomplished is covered extensively in the following chapter.

Chapter 4

Context Sensing and Aggregation for DTNs

In this chapter we will design and build a context-sensing framework for delay-tolerant networks. We build on the work of Chapter 3, where we showed that context awareness can benefit nodes in delay-tolerant networks by allowing them to adapt their communications to changing network conditions. Specifically, in Chapter 3 we looked at swapping network stacks between "conventional" stacks built for ad-hoc network environments, and delay-tolerant network stacks based on DTN-specific routing and transport protocols. This adaptation has to be informed by awareness of networking conditions (e.g., *network context*.) In this chapter we examine how this context can be collected and interpreted to effect systems changes.

Our approach consists of two parts. First, since network resources are particularly scarce in delay-tolerant environments, we examine how network context be collected efficiently [49]. Second, we examine how to store and share context and expose it to allow for the implementation of context-based adaptation algorithms. To accomplish this, we design and build a general purpose framework for context collection, aggregation, sharing, and adaptation and prove its utility on a

real system. This chapter starts with a brief overview of context and potential uses for context in adapting communication. We present our work on efficient context sensing in Section 4.2, and the design and implementation of a general purpose context framework in Setion 4.3. We also provide a compelling use case for context, even in non-traditional delay-tolerant networks [50], in Section 4.4, treating areas of bad connectivity in cellular networks as a sort of delay-tolerant region and showing that, through the application of context and delay-tolerant transport and routing concepts, we can improve the coverage and capacity of cellular networks.

In the next chapter (Chapter 5), we use the context framework and the lessons learned from Chapter 3 to implement a complete context-aware delay-tolerant system and present an evaluation of context-based adaptation in a real delay-tolerant environment in Chapter 6.

4.1 Context and its Uses

As we have alluded to previously in this dissertation, there are many types of context, and many ways in which context can be used to adapt network communications. With respect to acquiring and using context, there are two key issues: *(i)* what context to acquire and *(ii)* how to distribute and respond to context. There are many mechanisms for sensing, storing, and aggregating context. The key questions are what types of context information are useful for a given networking scenario, and how that context information should affect protocol behavior. For example, when a network cache relies on opportunistic node contacts, context may include knowledge about contact patterns (e.g., duration of contacts or number of unique contacts) in order to prioritize who to forward data to. Alternatively, when multiple users are requesting the same data, context may include aggregate activities of local groups of nodes in order to more efficiently utilize network resources. We give additional concrete examples in Table 4.1. In this chapter we focus on *general* mechanisms

Type of Context	Examples	Usage
System Context	battery level, charging status, CPU load, free memory	selectively enable/disable a clients participation in mobile caching and content sharing
Network Context	network type, roaming status, calling status, WiFi state	can influence sharing patterns; enables content prediction, which is required for advanced delivery
Location Context	GPS location, speed, heading	can provide common mobility patterns for prediction
Aggregate Context	common activities, social connections	servers can learn popular locations for caches and popular data to cache
Data Context	creation time, data size, time-to-live, priority labels	influence which data can be off-loaded depending on the network cache capabilities

Table 4.1: Potentially Useful Concrete Context Metrics

to efficiently collect context, and a *general-purpose* framework for context-based adaptation. This enables a wide variety of approaches using a wide variety of context types, including those listed in Table 4.1, and potentially many more.

4.2 Passive Context Sensing for Delay-Tolerant Networks

Adaptive delay-tolerant protocols and applications are heavily dependent on the availability and accuracy of contextual information. However, there is a trade-off between accurate context sensing and resource utilization. This is especially important in regards to network resources, which can be expensive in terms of battery usage, and a limited, shared commodity that many devices must share. Traditional mechanisms for collecting context rely on *active* metrics, or metrics that generate additional network traffic in order to measure context (for example latency) or at the very least exchange information such as location (for example to measure

node mobility). However, much useful context can be measured through *passive* means by eavesdropping on existing network traffic. Passive sensing is beneficial in that it conserves the already precious bandwidth available in challenged networking situations. We examine passive context sensing for delay-tolerant networks and find—unsurprisingly—that passive metrics are not as accurate as their actively sensed counterparts. However, several metrics can be correlated (for example packet error rate and load) to increase the sensing accuracy. We create a Passive Sensing Suite to facilitate the development of passively sensed context estimators, and find through experimentation that passively sensed metrics can be good estimators of their actively sensed counterparts. The rest of the this section is organized as follows. Section 4.2.1 presents an overview of our work on passive context sensing, and Section 4.2.2 provides related work in the area. Section 4.2.3 covers the design and implementation of our passive context sensing framework as well as the specific metrics themselves, and finally Section 4.2.4 provides an evaluation of the passive context suite.

4.2.1 Passive Context Sensing Overview

The ability to respond to the condition of the network is crucial in DTNs. Network-awareness is especially important for protocol adaptation as it allows communication protocols to change their behavior in response to the immediate network conditions or the available network resources. Network context can also be used directly by applications, for example to change the fidelity of the data transmitted when the available bandwidth changes.

Traditional means of measuring context are *active* in that they generate extra control messages or require nodes to exchange meta-information. Metrics that report message latency require nodes to exchange ping messages, measuring the amount of latency these messages experience. Traditional measures for determining the degree

of mobility in a mobile network require nodes to periodically exchange location and velocity information. The extra network traffic these mechanisms generate places an increased burden on the already taxed network, making it difficult to justify the use of context-awareness in the common case. If the overhead of sensing context information can be reduced, the benefit of the availability of the information is increased.

We define a framework for defining passively sensed context metrics based on network eavesdropping. Our approach focuses on sensing context with zero additional communication overhead. Our context metrics do not provide the exact measure of context that their active counterparts may provide, but we demonstrate the measures' fidelities match traditional measures of context. We use this framework to create instantiations of three common network context measures. For each of these metrics, we evaluate the specificity of the passively sensed context metric with respect to a simulated ground truth. Our work shows that passive sensing of network context can inexpensively provide information about the state of the network and that, especially when these metrics are correlated with each other, enable adaptive applications in delay-tolerant environments where traditional *active* context sensing is cumbersome.

4.2.2 Related Work in Passive Context Sensing

Much work has focused on supporting software engineering needs through frameworks and middleware that provide programming abstractions for acquiring and responding to context. For example, Hydrogen [20] defines a completely decentralized architecture for managing and serving context information. Hydrogen's abstractions are unconcerned with *how* context is sensed; clearly, performing context acquisition efficiently is important to the success of such a framework. Many other projects have also looked at reducing the cost of context sensing. Several of these take an

application-oriented perspective, identifying what high-level information the application desires and only acquiring information necessary to support an application's desired fidelity [70]. SeeMon [27] reduces the cost of context by only reporting *changes* in context; other time- and event-based approaches also limit overhead this way [13].

Active network monitoring has been explicitly separated from passive network monitoring. Komodo [56] defines *passive context sensing* as any mechanism that does not add network overhead. Komodo requires knowledge of the entire network (even, and especially, network links not currently in use), so the project implements an active sensing approach. Given that we focus on mobile networks based on wireless communication, we promote an approach that takes advantage of the inherent broadcast nature of communication, passively gathering information about links that may not be present at the application level. Passive measurement of network properties has been explored in a scheme that uses perceived signal strength to adapt a routing protocol [5]. This approach requires that nodes are able to easily discern the signal strength of incoming packets and relies on the use of protocols that already send periodic "hello" messages to monitor their neighbor set, which adds network overhead. A different approach monitors packet traffic to provide routing protocols information about packets dropped at the TCP layer [73]. This information allows protocols to more quickly respond to route failures. We undertake a similar approach in this work but focus on gathering a local measure of network properties instead of boosting performance on a particular end-to-end flow.

These related projects lay the foundation for our work in developing a comprehensive framework for passively sensing network context information. These previous projects have demonstrated 1) a need for context information to enable adaptive communication protocols and applications; 2) a requirement for the acquisition of context to be extensible and easy to incorporate into applications; and 3) a

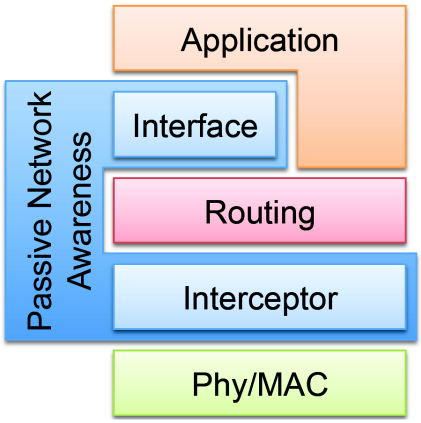


Figure 4.1: Architecture for Passive Context Sensing

desire to accomplish both of the above with low network communication overhead.

4.2.3 Passive Context Sensing Framework Design

In this section, we introduce a framework for adding passive context sensing into delay-tolerant network architectures, as well as the metrics we have implemented. A schematic of our architecture is shown in Fig. 4.1.

Physical and MAC layer implementations handle packet reception and transmission. Our framework inserts itself in two places: first between the MAC layer and the routing layer, and second above the routing layer before the application. The former point serves as an *interceptor* that allows eavesdropping on existing communication. The information overheard through this interceptor will be used to infer various context metrics as described below. The portion of the framework inserted between the routing and application layers exposes the passively-sensed context information to the application, enabling it to adapt to the current context.

Passive Metrics: Some Examples. The following three metrics each measure a dynamic condition of the physical or network environment. In all three cases, the sensed information can be useful to communication protocols that adapt

their transmission rates or patterns, and to applications that adapt high-level behaviors.

Network load. The simplest metric in our passive metric suite provides a direct measure of the local traffic on the network. Adapting to this information, applications can prioritize network operations, throttling communication of low importance when the network traffic is high. Communication protocols can also change routing or discovery heuristics in response to changing amounts of network traffic to avoid collisions.

Network density. In dynamic DTNs, a node's one-hop neighbors can constantly change, and applications can adapt their behavior in response. When the number of neighbors is high, common behaviors can increase collisions and therefore communication delay, while when the number of one-hop neighbors is low, conservative communication can lead to dropped packets and loss of perceived connectivity. To most easily measure the local network density, nodes exchange periodic hello messages with one-hop neighbors. While some protocols already incur this expense, adding proactive behavior to completely reactive protocols can be expensive. We devise a metric for passively sensing network density regardless of the behavior of the underlying protocol(s).

Network Dynamics. Our final example passive metric measures the mobility of a node relative to its neighbors. Traditional measures of relative mobility require nodes to periodically exchange velocity information. We approximate this notion of relative mobility by eavesdropping on communication packets to discern information about links that break. We show how this simple and efficient metric can correlate well with the physical mobility degree in dynamic mobile ad hoc networks.

The Specificity of Passive Metrics. A major hurdle in passively sensing context information is ensuring that the quality of the measurement sensed passively (or the *context specificity*) closely approximates the value that could have been

sensed actively for increased cost. This may differ from the actual value for the context metric since even active metrics may not exactly reflect the state of the environment. For each of the passive metrics we define, we generate its context specificity by comparing its performance to a reasonable corresponding active metric (if one exists). This not only allows us to determine whether the particular passive metric is or will be successful, but it also helps us tune our approaches to achieve better specificity.

Adaptation Based on Passive Metrics. One of the most important components of our framework is its ability to make passively sensed context information available to applications and network protocols. As shown in Fig. 4.1, we provide an interface that delivers passively sensed context directly from the sensing framework.

4.2.4 Passive Context Sensing Framework Implementation and Evaluation

To acquire context information at no network cost and little computation and storage cost, we created a passive network suite in C++. Our implementation takes network packets received at a node, "intercepts" them, and examines their details, all without altering the packets or their processing by the nodes. Our implementation also provides an event-based interface through which applications can receive information about passively sensed context. We describe the concrete architecture and implementation of our passive metric suite and look in detail at the specifics of our three sample metrics.

Implementing Passive Metrics. Fig. 4.2 depicts our implemented passive context sensing framework. Solid arrows represent the movement of packets. Specifically, packets no longer pass directly from the radio to the MAC layer or from the MAC layer to the network layer; instead they first pass through the passive context sensing framework. Dashed arrows indicate potential uses of the passively sensed

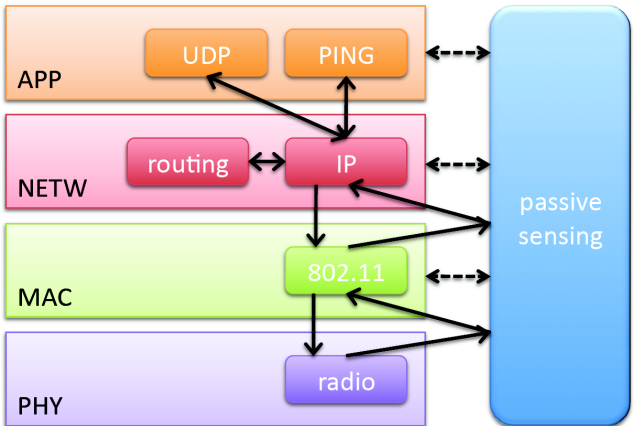


Figure 4.2: Implementation of Passive Context Sensing Suite

context in the form of event registrations and callbacks.

In our passive sensing suite, the interceptor (*passive sensing* in the figure) eavesdrops on every received packet. For each of the passively sensed metrics, the framework generates an estimate of the metric's value based on the information from the data packets in a specified time interval, ν. This time interval can be different for each passively sensed metric depending on its sensitivity in a particular environment. To define a passive metric, a new handler for the metric must be provided that can parse a received packet. The handler defines its own data structures to manage the necessary storage between estimation events. When any packet is intercepted, a copy is passed to the handler for each instantiated metric, and the handler updates its data structures.

Each new metric must also define an estimator that operates on the context information stored in the metric's data structure and generates a new estimate. When the passive framework is instantiated, each metric is provided a time interval for estimation (ν). The framework then calls the metric's estimator every ν time steps to generate a new metric estimate. Larger intervals result in lowered sensing overhead (in terms of computation) but may result in lower quality results (as

discussed below).

The Passive Metrics

For each metric, our interceptor takes as input the sensed context value at time t and the estimated value at time $t - \nu$ and creates an estimate of the next value of the time series. For each metric, this results in a moving average, in which previous values are discounted based on a weight factor γ provided for each metric. When γ is 0, a new estimate for time t is based solely on information sensed in the interval $[t - \nu, t]$.

Network Load. Network load can be sensed directly by measuring the amount of traffic the node generates and forwards. The network load metric's handler eavesdrops on every received packet, logging the packet's size in a buffer. To generate an estimate, the metric's estimator function simply totals the number of bytes seen in the interval ν and adjusts the moving average accordingly. Specifically, the network load metric nl_i of a node i is defined as the total of the sizes of the packets that the node has seen within a given time window $[t - \nu, t]$:

$$nl_i(t) = \gamma nl_i(t - \nu) + (1 - \gamma) nl_i^m(t - \nu)$$

where $nl_i^m(t - \nu)$ denotes the total size of packets seen by the node in the time interval $[t - \nu, t]$ (i.e., the measured value).

Network Density. Our second metric measures a node's network density, or its number of neighbors. This metric's handler examines each packet and logs the MAC address of the sender. When the estimator is invoked at time t, it tallies the number of unique MAC addresses logged during $[t - \nu, t]$. The network density of a node i is estimated by calculating the number of distinct neighbors of the node:

$$nd_i(t) = \gamma nd_i(t - \nu) + (1 - \gamma) nd_i^m(t - \nu)$$

where $nd_i^m(t-\nu)$ calculates the number of distinct neighbors observed in the previous time window. Node i was isolated during $[t-\nu, t]$ when $nd_i^m(t-\nu) = 0$.

Network Dynamics. Our third metric captures the relative dynamics surrounding a particular node. This metric is, to some degree, a measure of how reliable the surrounding network is. We can approximate this notion by eavesdropping on communication packets to discern link quality [64]. A node can do this by observing the quality of the received packets directly or by looking at the semantics of packets that indicate link failures.

In the former case, a node observes packets transmitted by neighboring nodes to determine the link quality lq_i^j, which is a normalized representation $\in [0, 1]$ of the quality of the link from node j to node i:

$$lq_i^j(t) = \gamma lq_i^j(t-\nu) + (1-\gamma) lq_i^{j,avg}(t-\nu)$$

where $lq_i^{j,avg}(t-\nu)$ calculates the average of the link quality values of the packets received from node j in the current window.

In our implementation, covered next, instead of directly measuring link quality, we rely on the presence of *route error* packets in the communication protocol to indicate faulty links. The metric's handler eavesdrops on every packet, counting those indicating route errors. When the context estimator is invoked, it returns the number of route error packets seen per second in the time interval $[t-\nu, t]$:

$$lq_i^j(t) = \gamma lq_i^j(t-\nu) + (1-\gamma) nre_i^{j,m}(t-\nu)$$

where $nre_i^{j,m}(t-\nu)$ is the number of route error packets from j in $[t-\nu, t]$.

We implemented the passive sensing metrics using the Click Modular Router [36],and we evaluated our implementation on autonomous robots from the Pharos Testbed. The following describes our implementation, and our experimental setup and results.

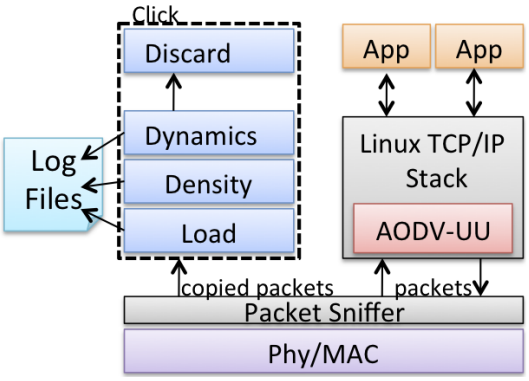


Figure 4.3: Click Passive Sensing Implementation

Implementation in Click. We implemented three context sensing elements, `PCS_Load`, `PCS_Density`, and `PCS_Dynamics`, which implement the three passive sensing metrics described in Section 4.2.4. Each element also has an external handler to allow other elements or processes to retrieve the computed context value. We have made our implementation available for download[1]. Fig. 4.3 shows the configuration we used in our experiments. The three passive sensing elements are connected such that all inbound packets are copied and processed by all three elements; the copy of the packets is then discarded. Although it is possible to configure Click to run as a kernel module so it can process the original packets instead of copying them to user-space, this was an unnecessary optimization for our experiments.

The Pharos Testbed. To fully evaluate our passive sensing implementation, we used the Pharos testbed [53], a highly capable mobile ad hoc network testbed at the University of Texas at Austin that consists of autonomous mobile robots called Proteus nodes [54]. We used eight of the Proteus nodes running Linux v.2.6, each equipped with an Atheros 802.11b/g wireless radio. The robots navigate autonomously using their onboard GPS and a digital compass.

[1]Our implementation is at `http://mpc.ece.utexas.edu/passivesensing`

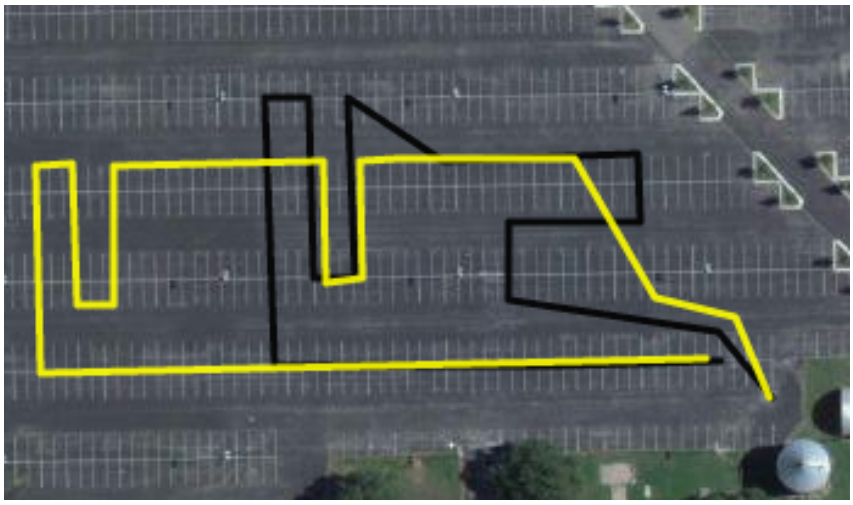


Figure 4.4: Waypoints for Experiments

Experimental setup. In addition to the passive context sensing suite, each node was running the AODV routing protocol [38] implementation from Uppsala University and sent UDP beacons to every other node at 1s or 10s intervals (depending on the run). This beaconing was independent of the passive sensing suite, and simply provided network load. Our reasoning for running AODV instead of a delay-tolerant specific routing protocol is simple. Prior work established simulated "ground-truth" metrics for the passive context metrics we implement, and these simulations used the AODV implementation from Uppsala University. Following suit in our real-world implementation allowed us to directly compare the results we got from the passive context suite with previously established results from simulation. This allowed us to reason about the accuracy of our implementation in regards to the prior work. We used two mobility patterns, a short pattern (shown in black in Fig. 4.4), which took about 5 minutes to complete, and a long pattern (shown in yellow), which took about 10 minutes [2]. Each pattern had a series of longer jumps punctuated by 2 series of tight winding curves. The robots were started 30 seconds

[2]Waypoints generated using `http://www.gpsvisualizer.com`

apart and drove at 2m/s (though this varied based on course corrections and imperfect odometer calcuations), and the winding curves were designed to trap several robots in the same area to ensure the formation of a dynamic ad hoc network. To ensure occasional link-layer disconnections in our 150m x 200m space, we turned the transmit power on the radios down to 1dBM (using the MadWiFi stack[3]).

Results. Fig. 4.5 shows values of the passively sensed metrics for one robot navigating the longer mobility model with 1s beacon intervals, the weight factor (γ) set to 0, and the time interval ($[t - \nu, t]$) set to 10 seconds to better show the instantaneous context values. Fig. 4.6 shows a different run with seven robots, the beacon interval set to 10s (instead of 1s), and with each robot instantiating a 1MB file transfer to one randomly chosen destination every 10 seconds. Seventy-eight total file transfers were attempted, of which 43 succeeded and 35 eventually timed out or were interrupted. Although the raw data is not extremely meaningful in isolation, it does show the degree of variation of context observed by a single node even in a small experiment. There are obvious correlations between node density and load and node density and network dynamics that were evident in our real world tests—some of this can be seen in the figures as well.

Comparing real-world results to simulation. To compare the real-world experiments to the simulated results, we took the recorded GPS trace of the Proteus nodes' exact locations in time and created trace files that were compatible with OMNeT++. In this way, we could simulate the exact mobility pattern executed by the robots, including variation from the intended waypoints due to GPS and compass error, steering misalignment, and speed corrections. Figs. 4.7 and 4.8 show the simulated results for the same node as Figure 4.5. We used the same simulation setup as in the previous section, but we set the simulated transmit power to 0.001mW in order to simulate the same number of neighbors on average for each

[3] `http://madwifi.org/`

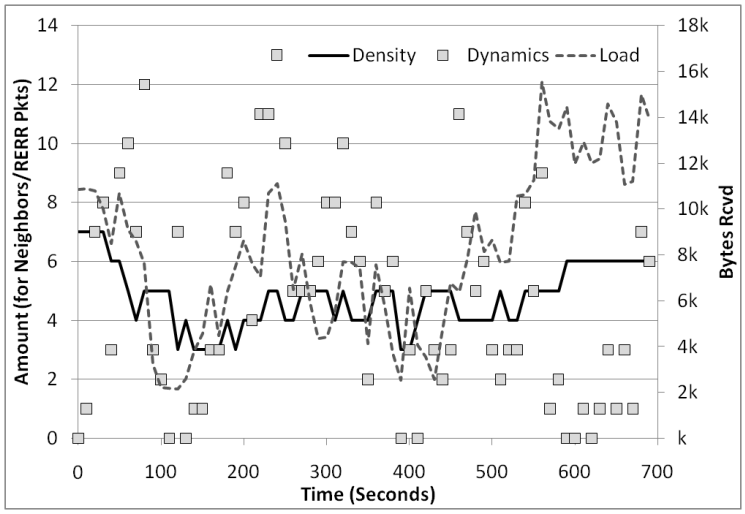


Figure 4.5: 8 nodes, 1s beacons, no file-tx

node—this value of 0.001mW was empirically determined by comparing simulations with the observed number of neighbors from the real-world experiments. We were able to correlate the node density between simulation and the real-world well on average, but the number of route error packets seen by the nodes differ significantly. We assume this is due to inaccuracies in the wireless model used in the OMNeT++ simulator.

Adapting to Passively Sensed Context. We have made our passively sensed context metrics available through an event based interface. Upper-layer protocols and applications can register to receive notifications of changes in passively sensed context metrics and adapt in response. Nodes in delay-tolerant networks must integrate with and respond to the environment and the network. Previous work has demonstrated 1) a need for context information to enable this type of expressive adaptation; 2) the ability to acquire context information with little cost; 3) the ability to easily integrate new context metrics as they emerge; and 4) software frameworks that ease the integration of context information into applications and

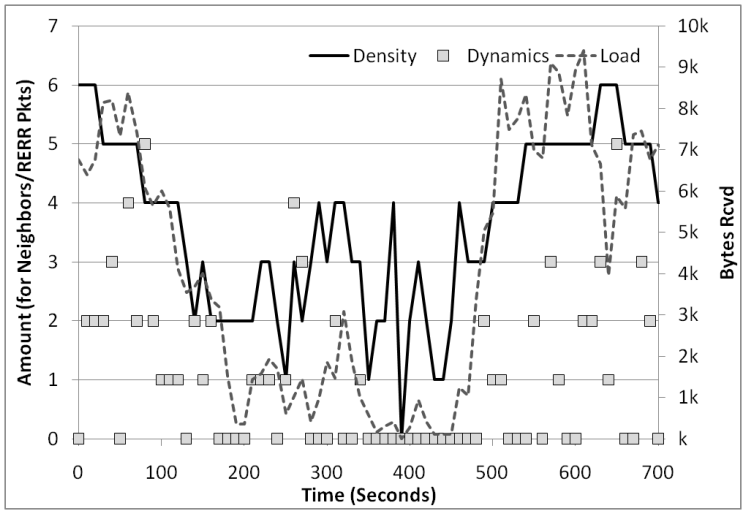


Figure 4.6: 7 nodes, 10s beacons, file-tx

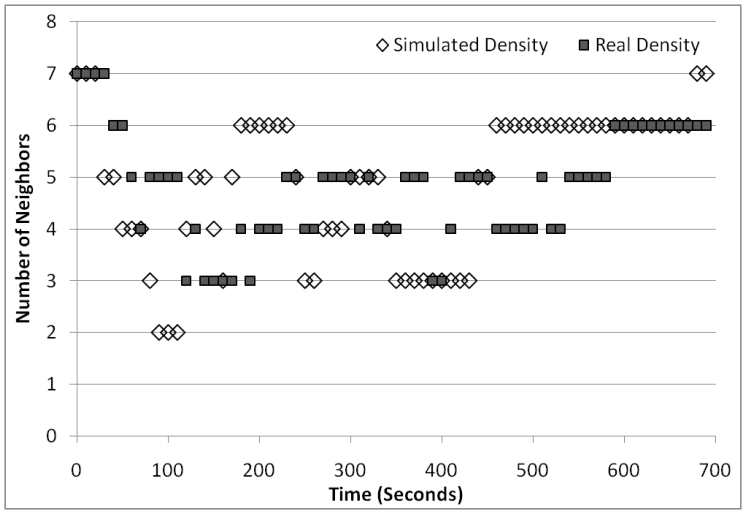


Figure 4.7: Sim. vs. real world density

protocols. In this section, we have described a framework that achieves all of these goals by enabling the *passive* sensing of network context. Our approach allows context metrics to *eavesdrop* on communication in the network to estimate network

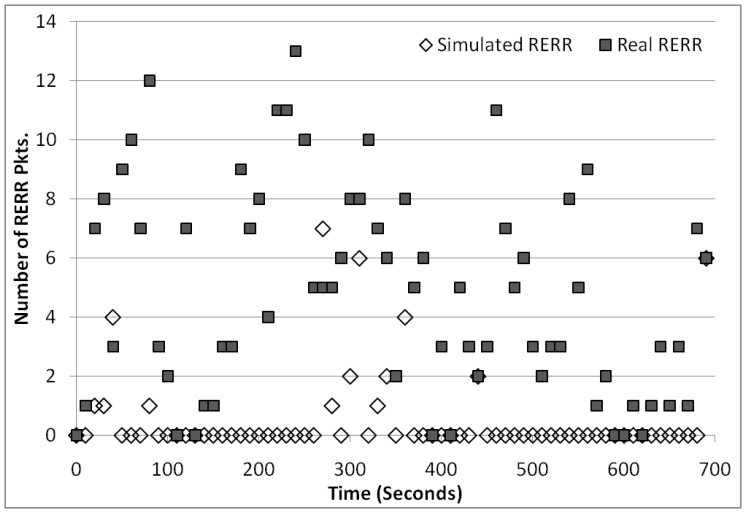


Figure 4.8: Sim. vs. real world route errors

context with no additional overhead. We have shown that our framework can be easily extended to incorporate new metrics and that the metrics we have already included show good specificity for their target active metrics in both simulation and a real network deployment. Additionally, we have shown that applications can even adapt the context sensing framework by correlating the results of multiple passively sensed context metrics. This information enables adaptive applications and protocols in environments where active approaches are infeasible or undesirable due to the extra network traffic they generate.

The next section takes a more systems-oriented approach to context and presents a general framework that can not only take context provided by the Passive Context Suite, but context from other sources as well to provide a unified framework for context-based adaptation. In the next section we also discuss how to store and share context, and how to provide mechanisms by which the context can be used to affect system adaptations.

4.3 The Context Agent Framework

Our prior work using context to inform stack adaptation decisions in opportunistic ad-hoc networks (see Chapter 3) led us to work on a general systems framework for context aggregation and context-based adaptation. Dubbed the Context Agent Framework (CAF), this work satisfies the need for a generic way of aggregating context information from various sources and sharing that information with various applications. The scope of the framework is broad; it embodies several concepts:

- Context types and values cannot be entirely known *a priori*, therefore any universal context solution must consider dynamic typing in order to remain flexible to new context types.
- Broad categories of context exist (system, data, user, network, etc.) and they each have appropriate aggregation strategies. A good context framework allows for all such strategies to coexist in a single framework.
- Adding context-based adaptation strategies to a system should be straightforward and simple. Adding new context types and their associated collection, aggregation, and (if useful) sharing mechanisms should as well.
- Any good context framework should support multiple programming abstractions, allowing for ease of use, as well as context sharing across multiple nodes to allow for automatic context distribution.

With these concepts in mind, we have designed and implemented the *Context Agent Framework* (CAF), a modular and flexible framework for collecting, aggregating, sharing, and adapting to context. The following section presents the CAF architecture and describes our design. Section 4.3.2 presents our implementation of the CAF architecture as a multi-threaded user-space context daemon.

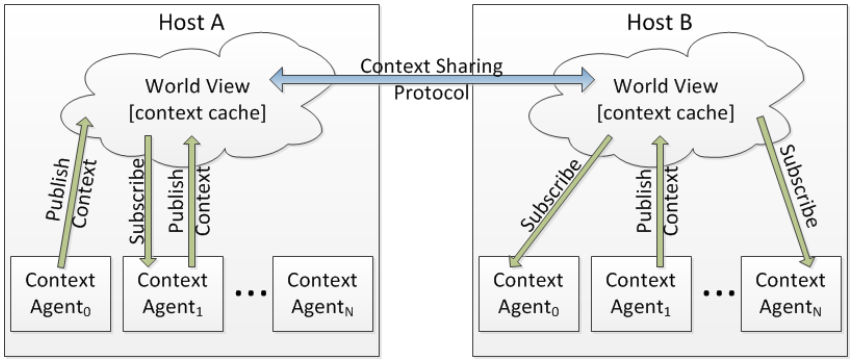


Figure 4.9: Context Agent Framework Architecture

4.3.1 Context Agent Framework Architecture

The Context Agent Framework is a multi-threaded architecture comprised of one or more *Context Agents* that either "produce" context updates (by sensing, aggregation, or both) or "consume" context to produce context-based system adaptations; context agents can be both producers and consumers of context. Figure 4.9 shows a high-level view of the CAF architecture. The framework is designed to run as a privileged *user-space* process, and each element of the architecture is embodied by one or more threads within the framework. This was done to provide concurrency among multiple context agents collecting independent context samples, as well as to allow blocking within the implementation of any given agent. The following describes the elements of our framework.

Elements of the Context Agent Framework

The following describes the purpose of each element of the architecture, and provides the mechanisms by which the elements can communicate with one another and interact with the system.

Context Agents. The Context Agent Framework is made up of one or more *context agents*. A context agent is responsible for one or more concrete context types;

its purpose is two-fold: gather context from the system, user, or network, and/or use context to adapt network communications. Some agents only collect context in order to make it available for other agents—these are pure context producers. Similarly, pure consumer agents only *use* available context in order to adapt network communications. Agents can act as both producers and consumers and can thus fill both roles for particular context types. The Context Agent Framework (CAF) brings together multiple individual context agents and provides a shared context cache called the World View to facilitate the sharing of context between agents. This component is discussed below. The mechanisms by which agents collect context depend on the exact type of context to be collected and the specific implementation of an agent. We present concrete examples of how gathering can be accomplished for particular context types in Section 4.2, however the CAF architecture is intended to be flexible regarding the exact mechanisms. It is intended as a framework for quickly and easily developing new context agents rather than an exhaustive set of all useful context agents.

Each agent within the framework is embodied by at least one thread in the CAF, although agents are allowed to spawn sub-processes and thereby have multiple threads implementing them. For context-producing agents there are two basic subtypes, *listeners*, and *gatherers*. As their names imply, listener agents implement a server process that waits for incoming connections from outside sources to provide context. The CAF supports any kind of blocking service implementation, although in our implementation we have limited ourselves to TCP sockets. Naturally, the *listener* must understand the exact format of the incoming context in order to be able to parse it, and this format must be agreed upon before the CAF is instantiated. The other type, *gatherer*, implements a proactive agent that fetches context. The means by which a *gatherer* can gather context is limited only by the possibilities available to a privileged user-level process. For example, a gatherer context

agent could read battery life by calling a battery monitoring daemon, or by reading the appropriate node in the Linux `proc` filesystem directly, or even by opening a file that stores battery life samples taken in the past. It could probe the Passive Context Sensing Suite discussed in the previous section for context updates. The main difference between *gatherers* and *listeners* is that *gatherers* proactively fetch context, usually on a timer-based trigger. Both types of agents can choose to post the gathered context to the World View, where it will be stored and made available to any local agents that subscribe to the resultant type.

The second purpose of Context Agents is to use context to adapt network communications. Essentially this can be thought of as evaluating some *context-adaptation function* using the available context in the World View to generate some output—the output is then used to inform some system change, for example to adapt communication. Once again, the CAF is intended to be open-ended in regards to how exactly this is accomplished. Since each agent runs as a separate thread within the CAF, context-based adaptations themselves can be anything a privileged, user-space process can accomplish—so far in our experiments we have limited ourselves to adapting the parameters of routing protocols, but there is no limit to the possibilities. Similarly to context aggregation, context-based system adaptation can be accomplished by any combination of system calls, file or socket writes, inter-process communication, etc. The re-evaluation of the adaptation function can be triggered in two ways; either with a timer, or on a context sample update.

A Word on Context Formats and Types. The CAF is purposefully open-ended in regards to the formatting of context samples. However, all context samples must conform to a loose standard based on the tuple concept [14]. Every sample must be associated with a type, which is itself just string representation. A context sample is formatted as: {*type*:*timestamp*:*value(s)*}. For example, location might be encoded as {**location** : *timestamp* : *longitude* : *latitude*}. There are two general

classes of context available in CAF: local context and global context. Local context is relevant only to the local system and potentially to nearby neighbors. Examples might include a device's battery life, or the intended destination of a mobile node. When context is shared between two nodes, local context is exchanged but not stored in the World View. This limits its distribution to a single hop, since nodes do not re-share gathered local context; they only share their own local context samples. Global context is context that is intended to be shared across the entire network. Examples might include the locations of static nodes or access points, passively-sensed node density or congestion estimates, or other context types that could be useful in adapting *network-wide* behaviors; All global context is stored in the World View. In reality, global context can be shared only among nodes that meet, so there is no guarantee of coverage. Global context types are also generally tagged with the geographic location at which they were sampled in order to assign the context samples to *containers* within the World View that represent discrete geographic locations. The size of these geographical context containers is user-configurable, and is discussed further on.

World View. The World View acts as the context cache. Context agents within the CAF store their gathered context samples in the World View, and "use" existing samples by retrieving them from the World View. The World View supports two main operations, *publish*, and *subscribe*. Publish operations allow context agents to add new context samples to the World View by means of the *publish* operation (analogous to the **out()** operation of tuple spaces). Subscribe allows agents to register their context interests with the World View, and in doing so receive updates analogous to the **read()** operation of tuple spaces. An agent can pass in a list of types, even type wildcards, and that agent automatically receives updates to any sample, or any new samples, that conform to the type. Contrary to traditional publish/subscribe systems, our publish and subscribe primitives are only available

locally—agents can only subscribe to context updates from their local World View. They cannot subscribe to updates from another node's World View. This limitation exists because of the nature of delay-tolerant network links, which cannot be predicted or relied upon with any level of certainty. However, since context samples are spread across the network by means of the World View sharing and merging capabilities, agents on one node *can* and do receive context samples generated on other nodes.

The World View is called as such because as the Context Agent Framework collects and processes context (either from its local context agents, or from context samples shared by other nodes), it generates a "world view" of the operating environment. This world view representation is made up of one or more geographically associated *context containers*, or more precisely *context sets*. Each context container is associated with a geographical area, and those context samples that are geo-tagged are mapped to the appropriate context container. The size, shape, and number of context containers is user-configurable to suit the needs of the network deployment; additionally, there is a single non-geographically associated container for storing global context samples that do not have geographically identifying information, for example queue length (if such a context type is desired to be shared at a global level). Containers are sets because no two samples with identical *types* may be stored in the same context container. In effect, there is a one-to-one mapping of context types to context samples, and the type itself uniquely identifies the given sample. This exclusive typing has two advantages: 1) it simplifies subscriptions since there is only ever zero or one sample returned for a given concrete subscription type, and 2) it simplifies the logic of adding and removing samples since a concrete context sample can be modified, removed, or replaced by using only the *type* information to identify it, requiring no additional unique information about the sample (for example: who added it, or when it was added). In practice, this one-to-one mapping is

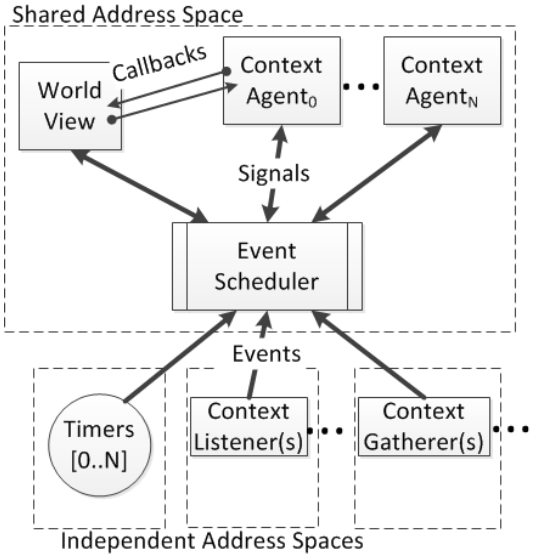


Figure 4.10: Context Agent Thread Model

not restrictive due to the free-form type identifiers, it only forces the type designer to include uniqueness within the type identifier to tell the sample apart from others. For example, battery levels could by typed as {**"battery-nodeID"** : *timestamp* : *mW-hours-left* : *mW-hours-total*}. In this example, the unique ID of the node is encoded in the type identifier.

Sharing Context. In order to accomplish context distribution, every node periodically shares its World View with its neighbors, and when a node receives another's view, the two are merged according to a merge algorithm. In practice, this merge algorithm is generally replacement (the sample with the latest timestamp is the one that is kept) but could easily be a rolling average or a similar operation. This sharing and merging of views allows the Context Agent Framework to approximate the global context. There are ways to efficiently share large amounts of context [16]; our method of World View sharing is discussed in the implementation section.

4.3.2 Context Agent Framework Implementation

We implemented the Context Agent Framework architecture in Linux using the Perl programming language. Perl was chosen for its rapid prototyping ability, its dynamic variable typing, and the ease of interaction with external processes and the system. The following describes the implementation of each of the elements of the CAF architecture.

Context Agent Implementation

As mentioned previously, the CAF is designed to be multi-threaded in order to allow blocking within the individual context agents themselves, and to allow for the concurrent gathering and distribution of context. Figure 4.10 shows the thread model of the CAF. Each square (or circle) in the figure represents a separate thread. Our implementation splits context agents into two separate threads, one which implements the context gathering functionality, and one which implements the context-based adaptation functionality. Those context agents which *only* gather context (pure producers) can be implemented by a single context gathering thread; likewise those context agents which *only* use existing context (pure consumers) can be implemented as a single thread. However agents that both produce and consume context need two threads—this allows for the simultaneous gathering and usage of context and prevents the adaptation routines from being preempted by incoming context data.

Intra-CAF Communication and Coordination. Figure 4.10 also shows the main modes of communication available to threads within the architecture. There are two mechanics provided by the CAF: *events*, which provide asynchronous communication, and *callbacks*, which provide synchronous communication. Asynchronous communication is used to connect the *gatherer* and *listener* threads to the agents on whose behalf they gather context. When new context is available, an event is created notify the agent that there is new context to process, and potentially pub-

lish to the World View. Events also provide for a mechanism to trigger timer-based routines. Example events include `BroadcastView` which triggers the *World View* to share its context with the network. All events go through the Event Scheduler that processes events according to their handlers and in the order that they are posted. Callbacks provide synchronous communication, and are used whenever asynchronous processing is either not required, or explicitly not desired. Example callbacks include `Subscribe` which passes a subscription for a set of context types to the World View, along with the address of the Context Agent who requested it.

Event Scheduler. Context aggregated by the listeners and gatherers is passed to the respective agents via *events* that are posted to the Event Scheduler. Effectively, the Event Scheduler acts as the gateway between the individual processes implementing the context gathering routines, and the threads that implement the agents and World View. The Event Scheduler is implemented as a simple FIFO queue. Each event has a separate handler that dispatches the event to the right module (agent, World View, or other thread).

World View Implementation. The World View element is implemented in its own thread within a shared address space that can be accessed by the context agents (allowing for agents to communicate with the World View using either events or callbacks). It implements a two-dimensional context "map" of the environment, that is indexed via (x,y) coordinates. The size of the map, and the size of the individual squares within it are user-configurable—each square of the map represents a separate geographically associated context container. Since the index is Cartesian, real-world GPS coordinates must be mapped to Cartesian coordinates before geotagged context samples can be inserted into the World View. In practice, this is done by a separate system process that provides location information as well as a bounding box that encompasses the entire mobility space. The bounding box is then used to translate GPS coordinates into their respective (x,y) values. This is

not the only way in which a location mapping can be accomplished, it is simply the mechanism we chose since it translates well to our real-world experiments. The World View stores context samples in their respective containers indexed by the (x,y) location in which the samples were taken—or in the case of non-geo-tagged samples, it stores them in a single "global" container.

Context Sharing. Context sharing is accomplished through a periodic broadcast of the serialized contents of the World View in a single UDP frame. This sharing mechanism is simple and can be inefficient in the case of a large number of context samples. There are opportunities here for efficiently sharing context (aside from our naïve periodic beaconing), for example Grapevine [16], however work in efficiently representing and distributing context is orthogonal to this dissertation.

Context Merging. Our context merging protocol is based on *timestamps*. When a node receives another node's summarized World View, it incorporates all of the context samples into its own World View. As discussed before, no two samples of the same type can exist in single context container within the World View, so in the case of duplicates, the sample with the newer timestamp is kept, and the older one thrown away. In our implementation, we rely on reasonably tightly synchronized clocks across all of the nodes, which, given the on-board GPS devices that are required for mobility purposes, is not an unreasonable assumption.

4.3.3 Context Agent Framework Conclusions

In summary, our Context Agent Framework represents a holistic systems-based approach to context collection, aggregation, storage, and adaptation. It allows for run-time context typing and wildcard-based context subscriptions so the complete taxonomy of context types does not need to be determined ahead of time. It presents a modular and flexible framework for developing context agents to gather and respond to context, and allows for agents to effect systems changes in whatever way

the programmer desires. The next chapter presents the concrete implementation of several context agents within the CAF to support the evaluations presented in Chapter 6.

4.4 A Use-Case for Context-Based Adaptation in Mixed Cellular/Delay-Tolerant Networks

The following is a use-case motivating context-based adaptation for mixed cellular/DTN networks. Although not the "standard" mobile delay-tolerant network that exists in rural areas or emergency situations, cellular networks present a compelling use-case for context-based adaptation. Specifically, the following work highlights the benefits of using network, data, and location context in order to offload data from the over-burdened cellular networks to higher bandwidth WiFi networks. We present the Mobile Advanced Delivery Server (MADServer), a novel DTN-based architecture that enables intelligent data offloading, caching, and querying solutions that can be incorporated in a manner that still satisfies user expectations for timely delivery. At the same time, MADServer allows for users who have low-quality or expensive connections to the cellular network to leverage multi-hop opportunistic routing to send and receive data. We present the MADServer concept and architecture, along with a preliminary implementation and real-world performance evaluations.

4.4.1 Motivation for MADServer Project

The recent explosion in cellular data traffic (due largely to the popularity of smartphones and flat-rate data subscriptions) is generating capacity problems for operators, both in the wireless spectrum and in cellular access networks. We develop a novel DTN-based web server architecture that alleviates these problems and pro-

vides a better end-user experience. Our architecture: (*i*) offloads "heavy" content transfers from the cellular network; (*ii*) makes use of the *context agents* to provide client mobility patterns and predictions and to find suitable network resources to enable advanced delivery of content; (*iii*) incorporates the *Context Agent Framework* architecture to generate offloading decisions; and (*iv*) provides a simple way for developers to prioritize content to show which can be easily offloaded and which is time-critical. In this way, we extend our work on context-based adaptation, and incorporate DTN concepts with cellular networks in which nodes are generally considered well-connected but in which mobile nodes can benefit greatly from opportunistic content sharing based on DTN principles. The following are some motivating example situations for this work.

Mobile Video-on-Demand. Services such as YouTube and those offered by local TV channels have become immensely popular. With the advent of smartphones, users also want to use these services on their mobile devices, straining cellular networks. Potential local similarities in requests for this content (commuters on the same train all wanting to catch up on last night's episode of a popular TV series or watch the latest viral video) generate great potential gains for caching and data offloading.

Events with large crowds. Some events entail crowds in areas where networks are provisioned for fewer people. Such events include big outdoor sporting events (such as marathons, which are spread over a large area, making it hard to deploy extra capacity at a particular location), or the recent royal weddings in Sweden and the UK, where large crowds gathered to see the newlyweds but still wanted to be able to watch the wedding on their mobile devices. Many in the crowd will have similar interests and request the same data; local caching and opportunistic exchange of data has great potential benefits.

4.4.2 MADServer Overview

MADServer uses context information about the mobility of content consumers combined with information about the network and data content itself. Specifically, our architecture makes it possible to send different pieces of web content over different network technologies, enabling offloading of "heavy" content from the cellular networks, in particular if such content is not time critical. Mobile devices can recombine the pieces before providing them to the user. Decoupling the methods of content delivery from user requests (in contrast to the World Wide Web model of immediate request/response) allows for the delivery of content to places where a user *will* be instead of where a user currently is. In today's highly mobile environment, this pre-caching takes full advantage of offloading opportunities. By integrating this with a DTN query mechanics, the needs of multiple users can potentially be served by a single transfer over the access network, and users can also collaboratively share content locally to satisfy requests.

4.4.3 MADServer Concept and Architecture

MADServer enables context-based data offloading without requiring new software at cell towers and with minimal changes at clients and servers; this allows for quick adoption of data offloading and eases the burden on network programmers. We split web content into two conceptual pieces: large content (pictures, streaming video, music, etc.), and the rest of the HTML frame, which itself can contain smaller content items (news tickers, feeds, etc.). We also distinguish two delivery vectors, *3G* (which can really be any cellular communication technology) and the *content offloading vector*, which can take many forms, although it will require some local content-cache and will generally rely on WiFi for its "last-hop" delivery.

We transfer small content over 3G and offload large content when beneficial and within delay constraints. A user's active application and transport sessions

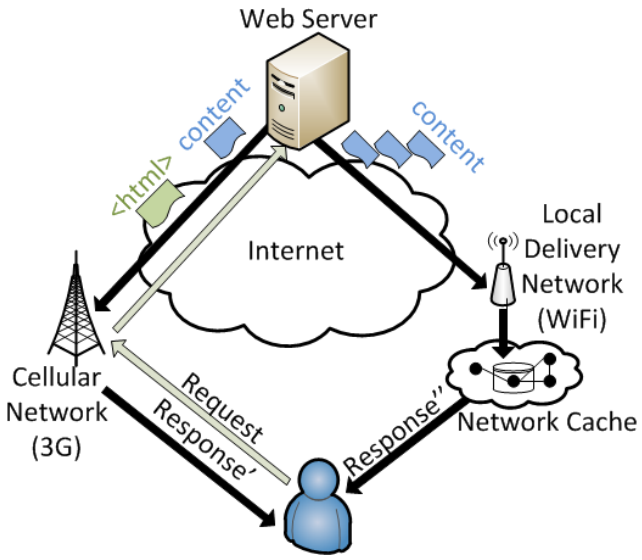


Figure 4.11: High-Level MADServer Architecture

need not be terminated and restarted when switching technologies, which disrupts user experience and leads to re-transferring partially received pages. Instead, when an alternate delivery vector is available, the web server can offload the bulk of communication but still provide a seamless session with minimal 3G usage. Figure 4.11 shows the high-level architecture. Client requests are sent over 3G, although large requests like file uploads could be offloaded. The server response is split into two, *Response′* and *Response″* to be sent over 3G and the offloading vector, respectively. *Response′* contains HTML frames to be served in the traditional way with the content tags re-written to point to future offloaded locations; *Response″* contains the content with its meta-data. The decision of when to offload content and which delivery vector to use depends on the context of the user and the context of the data.

It is up to the client to determine which context to share with the server; this will largely depend on what is available through the Context Agent Framework.

Since context is being collected on the client node and stored locally in the World View, the client has complete control over what context it shares with a given web service, and what it does not. The web services themselves have no special access to the World View except in terms of what context samples the client chooses to forward to the web service. This is a significant benefit since it ensures that the system does not violate a user's privacy requirements. The use of context is not entirely new in web services; existing web services that provide tailored services such as streaming bitrate adaptation have a similar reliance on context [6].

The context that a user acquires about her situation must be sent to the web server to enable intelligent data offloading. A client's context can be piggybacked on the client's (HTML) request. Our architecture leverages existing approaches for succinctly summarizing context [26] to prevent the transmission of context from overburdening 3G connections. Aggregated context information about a group of nodes can be similarly shared. Alternatively, context can be shared through the network cache and back to the server (through the reverse of the process of delivering *Response″*). Again, we do not construct new context acquisition and distribution mechanisms for MADServer but instead integrate with our context agent and Context Agent Framework approach.

Content Offloading Vector

In our architecture, time-critical content is delivered in the traditional manner across the cellular network so the user experience in not degraded, but less critical content can selectively be pushed across an alternative delivery mechanism, especially when the associated delivery delay is within tolerable bounds [4]. Our vision combines delay-tolerant networking principles with traditional client-server web services, and relies on asynchronous, opportunistic communication. Although the MADServer architecture is itself independent of the particular content offloading vector, our

implementation uses the DTN2 with the BPQ extension.

Publish/Subscribe with BPQ. In publish-subscribe systems, senders *publish* messages with topic labels, and these are distributed through the network according to *subscriptions*. In general, subscriptions are distributed to the entire network to form a routing structure; however, maintaining this routing structure in the face of topological changes [9, 21, 37, 72] is not always feasible in a mobile network.

The Bundle Protocol [59] (see Section 3.8) is the de facto standard application session protocol for DTNs, and the Bundle Protocol Query (BPQ) extension block [12] allows for intelligent in-network content caching, in essence providing a publish/subscribe system over DTNs. BPQ queries are sent towards the original content publisher, who responds to it. BPQ-enabled nodes on the path back to the requestor will, depending on space availability and local policies, cache the content. If another node makes a query that can be satisfied by the same content, the request can be served by these intermediate caching sites directly instead of forwarding the request all the way to the publisher.

Server-Side Architecture

Our server-side architecture, shown in Figure 4.12, has three main components, a *Request Processor*, a *Context Manager*, and a *Response Processor*, all of which live in a middleware "shim" layer directly below the web application. The client inserts its context into its HTML requests. The *Request Processor* looks for specific context tags, strips them from the request, and sends the context information to the *Context Manager*, which tracks each user by his 3G IP address. This context processing is the server-side equivalent of the Context Agent Framework—however the server does not provide the capabilities of the complete CAF architecture for two reasons. First, web services do not need context agents to collect context on their behalf—the

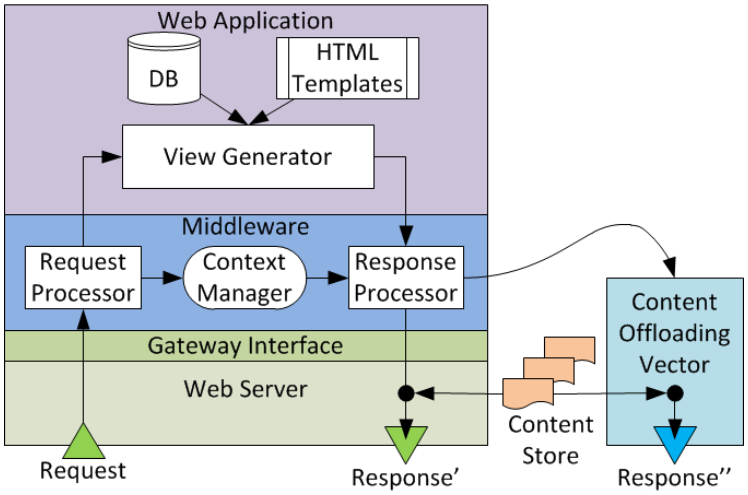


Figure 4.12: Server-Side Architecture

client nodes provide this. Second, it would be too great of a performance penalty to rely on an external process to manage the context—it must be done in the web service implementation itself; the necessary routines of the CAF are re-implemented in the web service middleware. Other than introducing this MADServer middleware "shim", requests proceed as normal, with no changes required to the web application itself. Once the HTML response is generated, the *Response Processor* rewrites the response according to the pre-defined rules and the user context provided by the *Context Manager*. The rules control if, and how much of, the content is removed from the response to be sent over the offloading delivery vector. Consider the following two content items:

1. <a href="http://**server-location.url**/media/largeVideoFile.flv" id=embedStreamPlayer>

2. <a href="http://**server-location.url**/images/largeImageFile.jpg">

Normally, both content items would be served by the web server over the 3G network. The *Response Processor* could rewrite the above response to:

1. <a href="http://**localhost/tmp-cache**/largeVideoFile.flv"
   id= embedLocalStreamPlayer>
2. <a href="http://**localhost/tmp-cache**/largeImageFile.jpg">
3. < LOOKUP CONTENT "dtn://*/largeImageFile.jpg",
   "dtn://*/largeVideoFile.flv" USING DTN_BPQ>

The largeVideoFile and largeImageFile urls now point to the local client cache, and the embedded video player changed to the local streaming service. The client looks for both content items in the local DTN network cache using the BPQ extension, and, once the content arrives, the client can stream the video from its own local cache.

MADServer Client-Side Architecture

In MADServer, clients must provide context to the server. This requires two elements on clients, a *context aggregator*, and a *MADServer browser plugin*. The Context Agent Framework (described in Section 4.3) serves as the *context aggregator* and provides the clients mobility information in addition to the locations of known offloading content caches. The resulting context is then sent to the servers via a browser plugin that automatically detects servers that have data offloading capabilities and inserts context information from the CAF into HTTP requests. Data privacy is a natural advantage of client-controlled context—the user retains full control over what context she shares and with whom. In our implementation we simply offload data using the bundle protocol when a WiFi connection is available;

this is naïve and could lead to performance degradation [4], but it is sufficient for a proof of concept.

4.4.4 MADServer Prototype and Evaluation

This section describes our MADServer proof-of-concept implementation and the evaluations performed using it. Our goal is to demonstrate that, using context information about offloading possibilities, the MADServer architecture can improve 3G bandwidth usage and content delivery latency for a realistic web service.

Content Offloading Service. We used the DTN2 Reference Implementation[4] of the Bundle Protocol to implement the data offloading delivery vector. This allows us to address content independent of a user's current IP address (instead using its globally unique endpoint ID) and to pre-cache content in places where a node might visit as long as there is at least one host there who implements the bundle protocol and can accept and forward bundles. If there is a network of such nodes, then content is disseminated to all nodes according to the the DTN2 forwarding rules. When the user eventually comes into contact with a node caching its content, the content is delivered.

Web Server and Interface. We use Apache[5] (since it is the open-source standard for deployment web servers) interfaced with our web application using Python Web Server Gateway Interface (WSGI).[6]

Middleware. We implemented the middleware layer and context manager in the Django Web Framework.[7] In our experiments, the CAF provides three pieces of context for each HTTP request:

$$\{\textbf{Offload}? < y/n >, \textbf{DestIPAddr} < addr >, \textbf{DTNEndpointID} < eid >\}.$$

[4] http://www.dtnrg.org
[5] http://apache.org/
[6] http://wsgi.org/wsgi/
[7] https://www.djangoproject.com/

When the CAF, or more specifically the purpose-built *offloading agent* within the CAF, on the client determines that mobile advanced delivery of content is beneficial, it asks the server to start offloading the content and provides a destination IP address where the content should be sent. Note that this need only be a destination running a bundle protocol router that can accept and forward bundles, and can thus act as an in-network content cache. The available content caches and their locations must have been previously sensed and stored in the World View. Additionally, the *offloading agent* must be able to guess the destination of the mobile user. These are very reasonable assumptions given that humans tend to conform to regular and predictable mobility models [15, 41]. The client also provides its globally unique bundle protocol endpoint identifier. The server encapsulates the requested content items in their own bundle, which is addressed to the client's DTNEndpointID and forwarded over the Internet to the provided IP address. Once the bundle is transferred to the destination, hop-by-hop opportunistic routing will deliver the content. We do not implement a predictive location context service on the client; instead for the evaluation we loaded this context *a priori*. However, the *offloading agent* could collect this context easily enough by leveraging recent work in energy efficient cellular phone-based predictive location awareness [41].

Web Application. For the application we wanted to use a realistic web service, so we built a fully-functional social networking website in the Pinax rapid web application development framework[8] with a MySQL database back-end and static file system for the content (pictures, video, etc.).

3G-Only vs. Offloading Experiment

The first experiment studies the potential for faster content deliver latency through adaptive offloading. This is practically feasible *only* through context-based adapta-

[8] `http://pinaxproject.com/`

tion of the client-server communiation. It is beneficial, since offloading not only frees expensive cellular bandwidth, it can deliver content faster even without pre-caching. We issued 50 requests for three different web pages with minimal HTML frames containing images of sizes {512 KB, 1 MB, and 5 MB}. The client is a Linux-based laptop with a USB 3G Modem and WiFi card located in Europe, and the server is located in the central United States. We measured average delivery latency using only 3G connectivity and with content offloading using our bundle protocol based offloading vector—the last hop link of which was over 802.11b via a WiFi access point. Figure 4.13 shows the latency results with their standard deviation for two different WiFi access points, one with a relatively poor signal strength and many users and one with good signal strength and very few users. WiFi is not necessarily faster than 3G [4], although this strengthens the argument for user context-based data offloading: if the cellular connection is currently expensive (e.g., the user is roaming), the extra latency may well be worth the money saved. Conversely, if the data is of high priority, using 3G (depending on the available WiFi bandwidth) may be better.

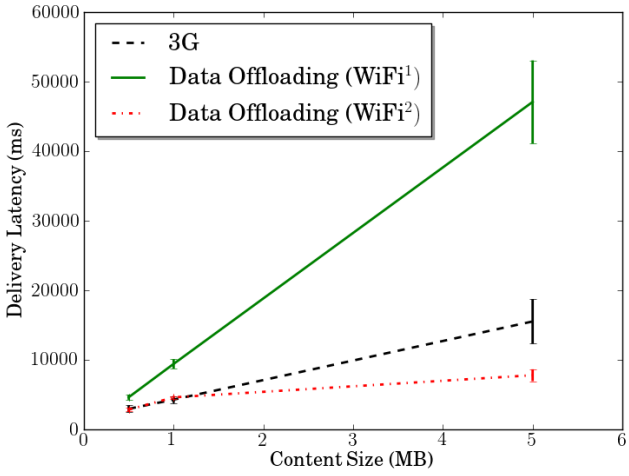


Figure 4.13: Impact of offloading on response time

3G Bandwidth Savings

In this experiment we looked at the 3G bandwidth savings enabled by context-based data offloading. Figure 4.14 shows the 3G bandwidth usage during regular operation and with data offloading. In this experiment, the client makes three requests for a page containing a 16MB video file. In the 3G-only case, the video is streamed over the 3G connection. In the data-offloading case, the video is bundled and sent to the client using our DTN2 content delivery vector; the final hop is over 802.11g from a WiFi access point. The client receives the bundle, inserts it into its local cache, and the video is "streamed" locally. As the results show, the 3G savings are significant (three orders of magnitude). Furthermore, the latency of the WiFi connection setup and bundle protocol client registration handshake is only a few seconds and is not detrimental to the user experience. The local video file stream can be started as soon as the video file starts arriving on the client; there is no need to wait for the whole file to arrive in order to start streaming from the file descriptor. The combined request-to-video-start latency is thus only a few seconds more than when using only 3G. Again, these bandwidth savings are not possible without the context information to enable them.

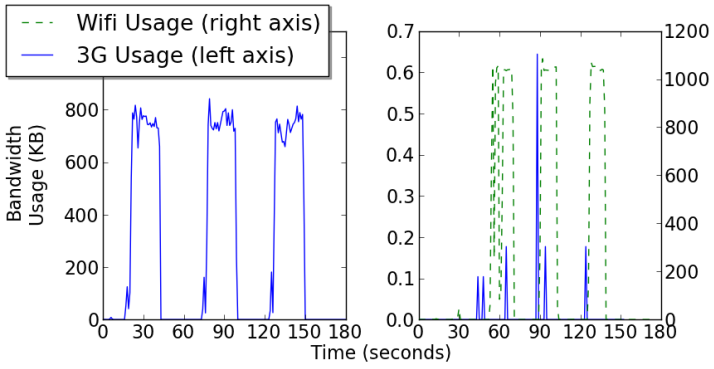


Figure 4.14: Bandwidth (3G-only vs. 3G + Offloading)

Advanced Delivery

Our MADServer implementation also enables context-based mobile advanced delivery of content. If the client's *offloading agent* determines that a node is about to connect to a content cache (using a predictor based on mobility data and the locations of known content caches stored in the World View), it can request that current and future content requests be serviced by the cache instead of downloaded over 3G. Figure 4.15 shows the results of an experiment in which a user makes 20 independent requests for web pages each including a 5MB content item. The left-hand graph plots the 3G usage over time if the content is requested and delivered via 3G only. The right hand plot shows a scenario where after the first five requests, the user determines that it will soon connect to a mobile advanced delivery cache (implemented by DTN2), at which point all content should be forwarded to the cache. The server responds to two of the requests over 3G regardless because the data is *high priority* according to its meta-data tags. The rest is forwarded to the content-cache and served to the client when it connects over 802.11b. When the client disconnects from the content cache, it sends a context update, and the remaining five requests are serviced over 3G. In this experiment, the *user context* was predetermined and provided ahead of the experiment. In the complete MADServer architecture, this context would be based on the World View's context samples providing the node's location and the locations of known content caches, and would be generated by the *offloading agent*'s adaptation routines. This experiment studies the benefits of such context, and not only does the MADServer architecture save 3G bandwidth, in this case reducing the 3G load from 108 MB to 60.7 MB given only a 51 second connection to a content-cache, but it is able to deliver all twenty content items in almost half the time.

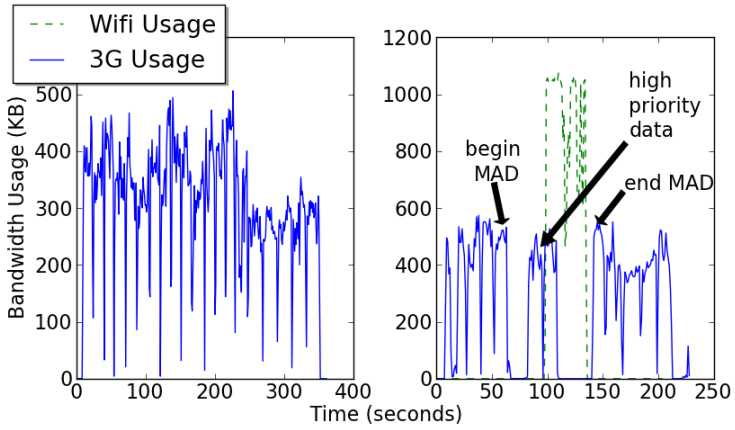


Figure 4.15: 3G-Only vs. Mobile Advanced Delivery

4.4.5 MADServer Conclusions

MADServer, a web server architecture for mobile advanced delivery, performs adaptive content offloading by splitting web responses into pieces based on user and data context, and delivering them to the client using different *delivery vectors*. It is enabled by our Context Agent Framework architecture and implementation—using the CAF (or a CAF-style architecture) allows for the appropriate context to be collected, and provides the mechanisms by which clients can use it to adapt their web services to suite their mobility patterns and offloading opportunities. Context-based content offloading addresses critical capacity problems in cellular data networks but must be done in context sensitive ways so as not to deteriorate the user experience. This architecture is a first step in integrating context metrics with data offloading decisions, and our implementation provides tangible results of the benefits. Additionally, MADServer shows us that an adaptive network stack architecture for delay-tolerant networks has potential benefits beyond the classic definition of delay-tolerant networks. As this work shows, such concepts can be easily extended to other domains, such as cellular networks, and can yield positive results in these

domains.

4.5 Research Contributions

This chapter makes the following contributions:

Research Task 2: We create a context-sensing module for delay-tolerant networks that supports efficient context collection using passive metrics. Using the lessons learned from the passive context sensing module, we also create a complete context framework for delay-tolerant networks capable of many types of context aggregation and sharing, incorporating both *passive metrics* which can be sensed from existing network communication and thus do not inquire a "sensing overhead", and *active metrics* which offer increased accuracy at the cost of increased network communication. We provide an implementation of context-sensing framework capable of context sharing across multiple nodes, and that supports publish/subscribe mechanics for ease of use.

4.6 Impact

Our work on passive context sensing is the first to deploy passive context sensing metrics in a real-world mobile network and to compare the results to established "ground-truth" metrics by replaying the mobility paths of the autonomous nodes in a simulator [49]. Our work on the Context Agent Framework, although not unique in its goals to be a unified context aggregation framework, is nevertheless the first such framework purpose built to adapt a delay-tolerant network stack. Our MADServer architecture is the first to combine delay-tolerant networking concepts and cellular networks (which are traditionally well-connected networks) to provide data offloading opportunities without requiring purpose-built content caches in the network (instead relying on DTN protocols and node mobility) [50]. It is also the

first to provide data offloading capability without requiring infrastructure changes in the carriers' networks.

4.7 Chapter Summary

In this chapter we presented an overview of context in delay-tolerant networks, two separate but related context sensing approaches, and a case study to examine the benefits of context even for non-DTNs. We studied efficient *passive* context sensing primitives that work by snooping on existing network traffic to estimate network context and built the Passive Context Sensing Suite implementation for Linux to test our ideas in a real mobile network. We compared the Passive Context Sensing Suite's metrics to previously established "ground truth" metrics using simulation. In this chapter we also presented the Context Agent Framework, a unified context sensing, aggregation, and adaptation framework that wraps all of the ideas on context-based adaptation presented in this dissertation so far into a flexible and easily programmed system. We designed the CAF around the concept of individual *context agents* that are responsible for collecting or adapting to specific context types, and that share their context with each other (and with other nodes in the network) using an associative context cache called the World View. We built a prototype of the Context Agent Framework as a multi-threaded user-level application for Linux. Finally, we presented a compelling use case for context-based adaptation to improve the capacity of cellular networks. In treating areas of bad connectivity in cellular networks as a sort of delay-tolerant region, we showed that context based adaptation can inform data offloading decisions, leading to better bandwidth utilization, and under some conditions, better deliver latencies.

Chapter 5

A Complete System Implementation: Context-Aware Delay-Tolerant Networks

In this chapter we present a full system integrating our work on adapting delay-tolerant network stacks with context awareness. Specifically, we connect our Context Agent Framework implementation from Chapter 4 with an adaptable DTN router (described below) and present the ways in which context is used to tune the runtime network stack parameters. This work presents the culmination of our goals in this dissertation—a complete system implementation for context-aware adaptation in delay-tolerant networks. We modify a popular delay-tolerant middleware implementation in order to allow for the dynamic updating of its internal router parameters and present our interface to connect the stack to our Context Agent Framework [46]. We also present the specific context types we collect in order to

adapt the network stack in a real-world delay-tolerant network and the mechanisms by which they are collected and shared. This chapter represents the *focusing* of our research goals in creating a flexible Context Agent Framework into the realization of a concrete context-agent created for a specific adaptation task. It also presents the specific means by which we adapt a real-world delay-tolerant network stack. It is in some ways a limited representation of the total capability of the Context Agent Framework and the dynamic DTN stack architecture as a whole. However, given the remarkable simplicity of the following implementations, what this chapter provides is proof that the framework and the whole concept of the adaptive network stack for DTNs makes sense in the real world. Chapter 6 presents our results using this complete system implementation.

5.1 Overview

For our complete system implementation for context-aware adaptation in delay-tolerant networks, we needed both a complete network stack implementation that provides routing and application layer capabilities for delay-tolerant networks, and a mechanism to dynamically update the parameters of the network stack in reaction to changes in network context. Our Context Agent Framework from Chapter 4 provides the means to sense, aggregate, and adapt to context. For the network stack, we chose the DTN2 Reference Implementation. The following sections provide an overview of DTN2 and the reasons we use it in lieu of MaDMAN, our previous middlware design. We then present our work on integrating DTN2 with the CAF.

Overview of DTN2 Reference Implemenetation. The DTN2 Reference Implementation [8] is a middleware solution for DTNs built in C++. Its architecture is designed for developing, evaluating, and deploying DTN protocols. DTN2 is also the reference implementation of the Bundle Protocol [59], an application-layer protocol for delivering messages (called *bundles*) between endpoints in a DTN. It is

a full-featured middleware solution for delay-tolerant networks, complete with an application programming interface that supports DTN-aware applications, allowing them to send and receive data that is automatically bundled and forwarded by the middleware. Routers within DTN2 control a node's forwarding strategy and govern how a node determines which bundles (or potentially bundle fragments) should be sent along any given link between two nodes. For our system implementation, we have focused our efforts on adapting network coded routing for delay-tolerant networks, a new class of protocols that show significant promise [28, 30, 60, 71, 74]. We use an implementation that we helped develop alongside our sponsors. Both network coding and our implementation are covered in detail below.

5.2 The Delay-Tolerant Network Stack

Our experiences using DTN2 both as an application layer over our MaDMAN Middleware (Section 3.7), and to support cellular network data offloading for our MAD-Server architecture (Section 4.4) led to our decision to use DTN2 as delay-tolerant network stack upon which we build our systems solution. Although in some ways our own DTN middleware, MaDMAN, is more modular and thus more flexible to implement on top of, it has a tendency to drop packets internally under very high loads (Section 3.9.3). It also lacks the necessary hooks to fully support a DTN-specific application interface. This is a constraint of the underlying Click Router platform, which was really intended more to support experimental routers than to support real application workloads. DTN2 is less modular, less stable, has a bigger in-memory footprint, and runs more slowly than MaDMAN. However, it has many more delay-tolerant specific routers already implemented, it has a full-featured application interface, and it is used by a large number of researchers in the delay-tolerant networking domain. It is also the reference implementation of the Bundle Protocol, the most wide-spread application layer protocol currently in use for delay-tolerant

networks, so it is up-to-date with the latest research to come out of the Bundle Protocol space. We want our contributions to have maximum benefit to the delay-tolerant research community; embracing DTN2 allows us to add to a vibrant and growing codebase used by DTN2 researchers around the world. DTN2's properties make it an ideal candidate for our system implementation despite the attraction of the slimmer, more modular MaDMAN. Additionally, since we built a Click-based convergence layer for DTN2, MaDMAN components can still act as network layer components of DTN2—for a discussion of this, see Section 3.9.3.

5.2.1 Coding-Aware Routing in DTNs

In network coded routing, intermediate nodes not only forward incoming packets, but also "mix" packets from multiple sources to increase information content in forwarded packets. Such approaches are particularly useful in DTNs, where opportunities to exchange data are intermittent and unpredictable. Network coding can reduce both routing overhead and delivery latencies compared to probabilistic routing without coding [69]. This is intuitive, since an intelligent coding (and re-encoding) scheme engenders *innovative* content in the fragments exchanged, increasing the likelihood that a received fragment increases the receiver's total information. The benefits of network coded routing in DTNs have been extensively studied and simulated. Erasure coding can improve the worst-case delay in DTNs [2, 30, 68, 69], and network coded routing compares favorably with probabilistic routing in addition to having lower overhead [71]. Combining random linear coding with epidemic routing has achieved better transmit power versus delay performance, especially when buffer sizes are constrained [74]. Network coding can increase throughput even in networks with non-homogeneous mobility [11].

A Note on Network Coding vs. Erasure Coding. Network coding and erasure coding are often (incorrectly) used interchangeably in the literature. They both

operate on the same basic principles. Both split a data unit, in our case a *bundle*, into fragments and create linear combinations of fragments to send to other nodes. The original bundle is never sent unencoded; different combinations of fragments are disseminated, and the destination needs to only receive some number of linearly independent encoded fragments to reconstruct the original bundle. The two coding techniques differ in which nodes generate encodings. In erasure coding, only the source generates encodings while network coding allows intermediate nodes to generate new random linear combinations of received fragments, resulting in increased "mixing" of information in the network and, theoretically, a more robust randomized routing protocol.

Before describing our routing implementation, we present an overview of our terminology:

Bundle: the fundamental data unit of the bundle protocol [59]. Bundles too large to be transferred in a single contact are fragmented; we encode across these fragments. Every bundle has a globally unique identifier (*GUID*).

Fragment x_i: a bundle is split into M (non-encoded) fragments of k bits, such that $x_i \in \mathsf{GF}(2)^k$. Each fragment is associated with its parent bundle's *GUID*.

Coefficient vector $\boldsymbol{c}$: a vector, $\boldsymbol{c} = \langle c_1, c_2, \ldots, c_M \rangle$, where $c_i \in \mathsf{GF}(2)$ and $i \in [1, M]$, controls which fragments to combine (`xor`) to create an encoded fragment.

Encoded fragment / Codeword $w_{\boldsymbol{c}}$: an encoded bundle made up of some linear combination of fragments such that $w_{\boldsymbol{c}} = \sum_{i=1}^{M} c_i x_i$ for some coefficient vector $\boldsymbol{c}$.

Re-encoding vector $\boldsymbol{d}$: on a node with r encoded fragments, a re-encoding vector $\boldsymbol{d} = \langle d_1, d_2, \ldots, d_r \rangle$ can create a new linear combination $w_{\boldsymbol{c'}} = \sum_{i=1}^{r} d_i w_i$. Re-encoding vectors are only used in network coded routing, and re-encoding is

only allowed for encoded fragments associated with the same original bundle (i.e., with the same *GUID*).

5.2.2 Network-Coded Router Implementation

The coded routing implementation we helped develop, called *SimpleNCRouter*, relies partially on DTN2's ability to break large bundles into fragments. This work is reported in [52]. Coded routing protocols can create encoded fragments from bundle fragments and distribute these encoded fragments independently using opportunistic connections that are inherent to DTNs; when a receiver has acquired *enough* pieces of information, it can reconstruct the original data. It is not necessary for the receiver to acquire *all* of the original fragments. With respect to DTN2, both the original (application) data and the encoded fragments are stored in bundles that move through the modules of the architecture implementation. In our version, when a bundle is received from the network or through the application programming interface (API), the node checks to see if the received bundle is an encoded fragment. If not, and the bundle is larger than some threshold, the node splits it into fixed-size fragments. Each fragment is tagged with its corresponding coefficient vector ($\boldsymbol{c}$) and the *GUID* of the original bundle for record keeping. Routers disseminate bundles containing an encoded fragment inside the payload.

The Essential Data Structures

To enable coding-aware routing in DTN2, we created data structures for managing and manipulating encoded fragments distributed in bundles. The following data structures enable us to intelligently store and forward encoded fragments and to easily reassemble the original bundles:

Network Coding Metadata Extension Block (NCMD Block) This metadata extension block is attached to encoded bundles. The information con-

tained within is based partly on a preliminary Internet-Draft [1] and relies on DTN2's extension block and metadata extension block support [66]. An NCMD Block carries information about the coefficient vector used to generate the payload and the *GUID* corresponding to the original bundle. Putting this data in an extension block, as opposed to the bundle payload, gives the router access to the encoding information without loading and parsing the payload from the data store.

Network Coded Bundle (NC Bundle) The `NCBundle` class is a wrapper for encoded bundles; it is a simple aggregation of a pointer to a bundle and an associated NCMD Block. To keep from processing the extension block repeatedly, we store the fields parsed out from the NCMD Block in a data structure. The bundles encapsulated as NC Bundles contain encoded fragments generated from the original, larger application bundles.

Network Coded Bundle Collection (NC Bundle Collection) Our central data structure is `NCBundleCollection`, a table of collections of NC Bundles, indexed by *GUID*. Figure 5.1 shows how a node handles incoming bundles, interacting with the appropriate NC Bundle Collection to store encoded fragments and assess the rank of each application-level bundle. A received NC Bundle is first sorted by its *GUID* into the correct collection, and the coefficient vector used for the encoding is copied from the NCMD Block into a row-reduced matrix whose rows span the space of the current collection. The matrix is echelonized and the rank checked. If the rank of the collection increased, the NC Bundle is retained and otherwise discarded.

If the row-reduced coefficient vector matrix in an NC Bundle Collection reaches full rank, the matrix is inverted. The columns of the inverse matrix contain the coefficients needed to sum the NC Bundle payloads together to decode the original bundle fragments. Since the coefficient vectors are chosen randomly, they may not be linearly independent, and sometimes a node must receive more

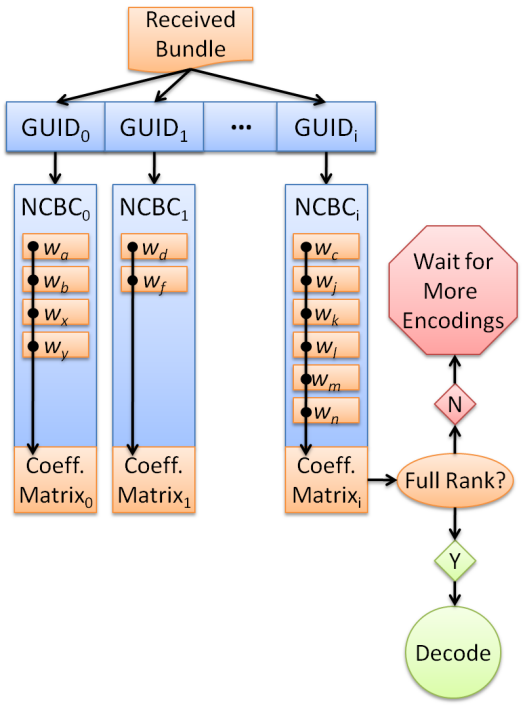


Figure 5.1: Incoming Bundle Data-Flow Diagram

than N encoded fragments before it can decode the set. The NC Bundle Collection class uses the `m4ri` library [32] to do fast binary linear algebra computations. Our routers can also operate in *non-rank-checking mode* to support experimentation. In *non-rank-checking mode*, matrix manipulations are disabled and an NC Bundle is discarded only if that exact bundle has been seen previously.

Intermediate nodes generate new "mixes" from received encoded fragments, increasing the innovative encodings in the network. Instead of simply selecting an existing bundle to send from the NC Bundle Collections, `SimpleNCRouter` creates a new encoded fragment to send. It first randomly selects an NC Bundle Collection, then chooses a random re-encoding vector and `xors` together the payloads of the NC Bundles indicated in the re-encoding vector. It then sends the new bundle on

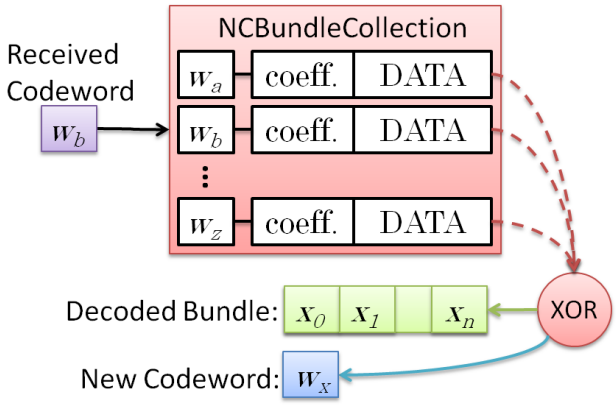


Figure 5.2: Encoding/Decoding

the available link. The configurable weight of the re-encoding vector defaults to the log of NC Bundle Collection rank. Figure 5.2 shows this process of generating a new NC Bundle and how stored NC Bundles in an NC Bundle Collection are combined to recreate the original bundle. In general, there are many ways to create encoded fragments (*e.g.*, arithmetic over larger finite fields); we use `xor` for simplicity.

Configuration Options. `SimpleNCRouter` has several configuration options to experiment with performance and functionality (default values are in parentheses):

rank_check (true): if set, the router will discard received NC Bundles that are not innovative.

reencode (true): if set, router will generate new NC Bundles with re-encoding vectors of weight more than 1.

auto_decode (false): if set, router immediately decodes when an NC Bundle Collection reaches full rank.

keep_original_bundles (true): if set, router retains original bundles, even after fragmenting into NC Bundles.

chunk_size (50000B): size of the fragments.

max_weight (0): max weight of re-encoding vectors. If 0, the weight is the log of the rank of the collection.

Evaluating the *SimpleNCRouter* implementation is beyond the scope of this dissertation, as it is not claimed as a contribution [1]. However, we use this router extensively in designing and building a context-aware version of the network coded router, which is covered in the following sections.

5.3 Integration with the Context Agent Framework

The previous section described our network coding router, called *SimpleNCRouter*. In this section, we extend the *SimpleNCRouter* to be adaptable and context-aware, presenting a modified version of it called the *Context-Aware Network Coding (CANC) Router* and focusing on using context to dynamically adjust protocol behavior. We focus on network coding for reasons described in Section 5.2.1, but the specifics of the dynamic protocol reconfiguration are applicable to many types of routers within DTN2. We integrate our *CANCRouter* with the Context Agent Framework described in Chapter 4 by providing a context agent for the CAF, the *CANCR Agent* built for the purpose, and describe specifically how context is used to adapt communication in our DTN system implementation. We also provide details on a second context agent, the *Geo-Context Agent*, which communicates with the Proteus robots' mobility controller and provides context information about node location and destination using GPS.

Figure 5.3 shows the software components. The novel contributions of this section are in the following three components:

[1] Interested parties may wish to read [52]

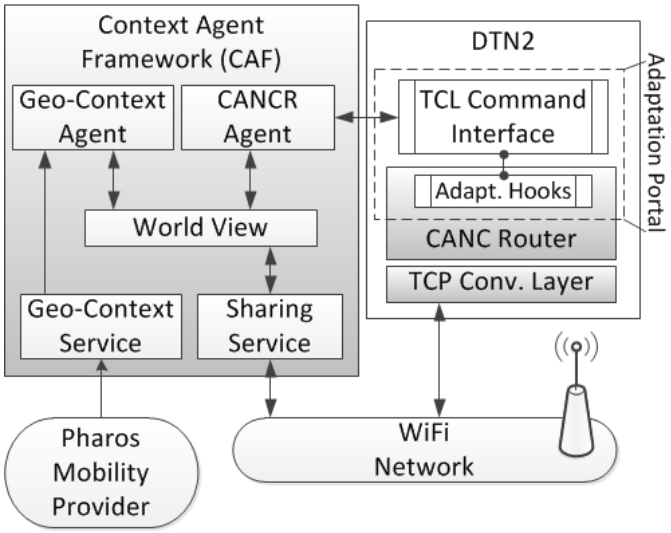


Figure 5.3: CANC Router Implementation

- the *CANCR Agent*, which collects and processes context information for adapting a network coded router
- an *Adaptation Portal*, which exposes configuration hooks into a Bundle Router within the DTN2 middleware, generally extendable to other routers
- a highly configurable *CANCRouter*, which makes an extensive set of configuration parameters available to the CANCR Agent. Our prototype extends the SimpleNCRouter.

The following sections describe the new context-aware network coding router, the *CANCRouter*, as well as the *CANCR Agent* within the CAF, and the *Adaptation Portal* through which the CANCR Agent can reconfigure the CANCRouter.

5.3.1 CANC Router

We extended the SimpleNCRouter into the CANCRouter by implementing functions to handle updates to a few key configuration parameters. The parameters

themselves are covered in the following subsection. The changes to the codebase of SimpleNCRouter itself were fairly minimal, and yet as Chapter 6 will show, a little context-based parameter tweaking makes a big difference in the capabilities of the router. This serves to reinforce our ideas that context-based adaptation should happen *outside* the implementation of any given router and that only the parameter update "hooks" need to be provided in order to make it work.

5.3.2 Adaptation Portal

In general, an *Adaptation Portal* specifies the interface between a *Context Agent* within the Context Agent Framework (CAF), which acquires and assimilates context information, and configuration hooks in the underlying Bundle Router, in the form of assignable parameters. Different implementations of the Bundle protocol will provide different mechanisms for this connection. Within DTN2, the most obvious option is to use the *TCL Command Line Interface*. Ultimately, any Adaptation Portal serves as a bridge over which information transits between the Context Agent and the Bundle Router (sending routing protocol parameters in one direction and DTN context information in the other direction). It could easily take the form of a socket-based communication. In this case, it was better to use the *TCL Command Line Interface* for the communication since this is the standard DTN2 router interface to the outside; it also allowed for simple and highly expressive human debugging of the entire system, allowing us to manually enter parameter changes on the TCL Command Line—this allowed for debugging the CAF-side and the DTN2-side of the implementation separately.

In creating the concrete *CANC Router Adaptation Portal*, we identified three configuration hooks that are immediately useful for controlling the flow of encoding bundles when the underlying routing protocol is the *CANC Router*: weight, rate, and balance. Each parameter can be configured for each globally-unique identifier

(GUID) and for each neighbor. The CANC Adaptation Portal tunes these parameters through the DTN2 TCL command line interface. Figure 5.4 illustrates the effects of these parameters.

- **Rate.** This controls how fast a node sends encodings to a neighbor relative to how fast the neighbor sends to it. For a rate $r \geq 1$, the node can send r encoding bundles for the specified GUID for every 1 it receives in return. For a rate $0 < r < 1$, the node can send an encoding bundle from the specified GUID when it has received at least $1/r$ from its neighbor. A rate of zero ceases sending, and a rate of -1 causes unconstrained sending.[2] In Figure 5.4, at the top, Node A chooses a higher rate to send to Node B than vice versa. .

- **Weight.** If a node is carrying encoding bundles associated with multiple GUIDs, the weight parameter allows the Context Agent to bias the selection of which GUID's bundles to favor. In the SimpleNC router this selection was uniform, which was inefficient if one GUID had a much higher rank than another or if one GUID was much newer than another. Weight parameters are taken into account after GUIDs are checked for eligibility based on rate counters. The weights can be set using the configuration hooks or automatically based on relative GUID ranks. The example depicted in the center of Figure 5.4 shows a higher weight for GUID2 because its rank is higher.[3]

- **Balance.** If a node has two or more neighbors, the links share the same limited bandwidth. A router may want to bias the bandwidth to a particular neighbor; the balance parameter enables this. The balance parameter may

[2] If nodes A and B have rates of $r_A > 0$ and $r_B > 0$ for a particular GUID, this algorithm is subject to deadlock unless $r_A \cdot r_B \geq 1$. We added a timer so that each node's send counter is reset every second, allowing it to send at least one encoding bundle per second, unless the rate is zero.

[3] We favor the GUID with higher rank because the sender has more information about that GUID and is more likely to be able to complete the receiver's entire application-level bundle. Alternative rationales for assigning weights are possible.

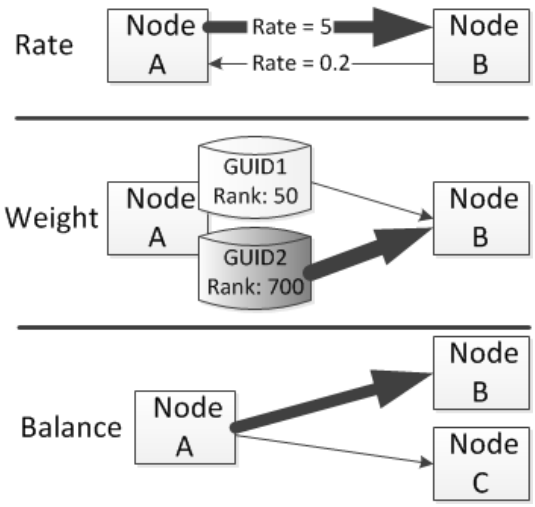


Figure 5.4: The configuration hooks for the CANC Router

be approximated by coordinating the rate parameters.[4] In the bottom of Figure 5.4, Node A biases the bandwidth to Node B. This may be because Node B is in an information-starved part of the network.

SimpleNCRouter is similar to flood router in trying to disseminate all bundles to all neighbors, which can be unsustainable in many practical deployments. *CANC Router*'s adaptation hooks let us give directionality to the flow of data in the network, minimizing wasted bandwidth.

5.3.3 The Context Agents

This section presents the two context agents within the Context Agent Framework (CAF) that accomplish the context collection and interpretation to inform the parameter changes sent to the CANCRouter through the Adaptation Portal.

[4] An available MAC-layer broadcast convergence layer would make this parameter irrelevant.

Geo-Context Agent

The Geo-Context Agent, built for evaluating the CANC Router in the Pharos Testbed, provides location information to the CANCR Agent by posting it to the CAF's World View. It is composed of two parts: the Geo-Context Listener and the agent itself. The Listener is a service process that listens for periodic (about every 2 seconds) GPS location updates from the Proteus Mobility Controller. These updates include the location of the node (e.g., its GPS coordinates) and the destination of the node (also GPS coordinates) if the node is a mobile node. The Geo-Context Agent itself interprets these GPS locations (using information about the "bounding box" of the experiment space) to provide Cartesian coordinates mapping the node's location to a context region (or *box*) within the World View's context grid. Whenever a location update arrives from the Proteus controller, the context tuple that stores the node's current location in the World View is updated. In this way, the CANCR Agent can subscribe to location updates and receive them as locations change. The location and destination context of a node is stored in the *local* context cache of the World View—that is, it is shared with single-hop neighbors but not disseminated across the network.

Context Aware Network Coded Routing (CANCR) Agent

Given our use of network coded routing, the routing process itself can be considered an information dissemination problem where the goal is to move data from where there is high information density towards the sink (which starts with zero information). In our implementation, we focus on context that represents nodes' mobility (i.e., their current location, their destination, and whether they are mobile or static) and the network coding router state (i.e., for each known GUID, the current rank of the decoding matrix and the source and destination of the bundle). Nodes both share this information with each other and collect this information from their one-

hop neighbors. As the CANCR Agent collects and processes context, it generates geographically tagged samples of the ranks of the decoding matrices of the nodes it encounters. These rank-samples are tagged with the location at which they were observed and the timestamp and posted to the World View. Only static nodes are sampled, that is mobile nodes' locations are not predictable so their rank-samples are not stored in the World View. Through the sharing of World Views as nodes meet, the global rank information is disseminated, and nodes get an idea of the information diversity of the network.

Using this opportunistically gained global information, the *CANC Context Agent* uses mobility and router context to set the *rate* at which a node will send encodings to a given neighbor. We implemented two variants of the rules, one that only considers the relative ranks of two nodes (ignoring location and mobility context) and one that also considers node mobility, as shown in Figure 5.6. Although this *relative rate* scheme is specific to network coding, in general, controlling the sending/receiving balance between a pair of nodes is a baseline control mechanism for many bundle routing protocols. These updates to the *rate* at which encoded bundles should be sent are communicated to the CANCRouter using the adaptation portal.

5.4 Research Contributions

This chapter makes the following contributions:

Research Task 3: Design and implement a complete systems solution that incorporates concepts from Research Task 1 with the context framework from Research Task 2 to adapt a real delay-tolerant network stack.

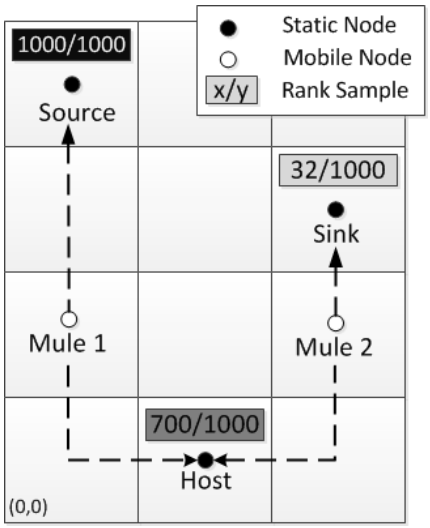


Figure 5.5: Context World View (with sample nodes and waypoints)

5.5 Impact

To our knowledge, the work in the chapter represents the first real-world implementation that adapts the behavior of a network coded router using context, and the first system which interfaces an external application (the Context Agent Framework) to the widely-used DTN2 Reference Implementation of the Bundle Protocol through its TCL Command Line Interface in order to control it [46].

5.6 Chapter Summary

In this chapter we presented a full system implementation for adaptive delay-tolerant networks. We combined our from Chapter 3 on adapting delay-tolerant network stacks with the Context Agent Framework. We provided an implementation which uses the the Context Agent Framework to adapt the behavior of a delay-tolerant network-coded router using context, and presented the specific context types that are used to tune the runtime network stack parameters, the methods of adaptation, and

```
if neighbor.rank == MAX_RANK:
→neighbor.rate = 0; # do not send to any full-rank node
else if neighbor.eid in bundle.sinks
→neighbor.rate = MAX_RATE; # unbounded rate to sink
else if neighbor.rank == 0:
→neighbor.rate = MAX_RATE; # neighbor has no encodings
else if self.rank == MAX_RANK:
→ neighbor.rate = MAX_RATE # I have full rank
else:
  # default case, use relative rates
→neighbor.rate = self.rank/neighbor.rank
```

(a) Basic rank-aware rules

```
if neighbor.rank == MAX_RANK:
→neighbor.rate = 0; # do not send to any full-rank node
else if neighbor.eid in bundle.sinks:
→neighbor.rate = MAX_RATE; # unbounded rate to sink
else if neighbor.type == MOBILE:
→boolean destinationsFull := true
→foreach destination in neighbor.destinations:
→→if WorldView.destination.rank != MAX_RANK:
→→→destinationsFull := false
→if destinationsFull == true:
→→# mule's destinations all have full rank, don't send
→→neighbor.rate = 0;
→else: skip # move to next rule
else if neighbor.rank == 0:
→neighbor.rate = MAX_RATE; # neighbor has no encodings
else if self.rank == MAX_RANK:
→ neighbor.rate = MAX_RATE # I have full rank
else:
  # default case, use relative rates
→neighbor.rate = self.rank/neighbor.rank
```

(b) Extended mobility-aware rules

Figure 5.6: *CANCR Context Agent* Rules

the interface that allows the Context Agent Framework to affect the delay-tolerant stack. The work in this chapter represents the culmination of the goals set out in the beginning of this dissertation. It represents a complete system implementation for context-aware adaptation in delay-tolerant networks. In the following chapter, we prove the benefits of our system by deploying it in a real delay-tolerant networking using autonomous robots from the Pharos Testbed.

Chapter 6

Validation using the Pharos Testbed

This chapter presents the validation of our systems solution for delay-tolerant networks. We present results gathered with the implementation described in Chapter 5 using autonomous robots from the Pharos Testbed. The main goal of this chapter is to validate our complete systems implementation in real-world operating conditions, on commodity hardware, employing real mobility through the autonomous navigation capabilities of the Pharos Testbed nodes. This chapter also proves the benefits of a flexible context aggregation and adaptation framework like our Context Agent Framework (CAF) by showing how a minimal number of context-based adaptation "rules" can be used to great effect in improving routers in delay-tolerant networks. In short, it proves that our ideas about context-based manipulation of internal network stack variables by means of our CAF can and do work in a real-world deployment. This chapter is not, however, an exhaustive list of all of the beneficial adaptation possibilities for delay-tolerant networks nor a complete benchmarking of the entire system. Such an exploration is well beyond the scope of this dissertation. Instead, we aim to prove that our total system implementation works

as we intended and in this way lay the groundwork for researchers to build their own adaptive routers and protocols using our architectural ideas and our Context Agent Framework.

6.1 Overview of Experiments

The experiments below compare our context-aware network coded router (CANC Router) with the basic network coded router (SimpleNC) in a variety of situations, under a variety of constraints. They clearly show that context-based adaptation can benefit nodes in delay-tolerant networks by improving latency and lowering overhead. We present a variety of experiments below. For the remainder of this chapter, the following terminology applies:

1. **SimpleNC:** refers to the implementation of network coded routing developed in [52] and described in this dissertation in Section 5.2.2.

2. **CANC Router:** sometimes indicated as *CANCR* for short, this implementation is as described in 5.3.1. CANCR improves upon the SimpleNC router by allowing for a variety of internal parameters to be controlled by the Context Agent Framework (CAF) during runtime, allowing for context-based adaptation of the routing protocol behavior.

We are somewhat limited by the number of consistently operational robots in the Pharos Testbed, and therefore the experiments use between three and five nodes. In order to provide consistency and comparability among results, the following properties apply to *all* of the experiments described in this chapter.

1. *Communicating Parties:* All of the experiments are comprised of some combination of *static* nodes and *mules*. The static nodes act as either senders, receivers, or in some cases both. As implied, they do not move. The mules

act as the data ferries (or data mules) and navigate autonomously between waypoints in order to "bridge" disconnected parts of the network, or to move data between static nodes.

2. *Data:* All data sent in our experiments is bundled into 100M bundles. In all cases, the 100MB bundle is fragmented into 10kB units which are used to code across. Thus, a complete bundle requires 1000 unique linear encodings to represent it in its entirety. Nodes send one or more 100MB bundles to each other in the experiments.

We perform four categories of experiments: indoor, testbed, outdoor, and characterization experiments. The indoor experiments use limited mobility in a controlled indoor environment–they are presented in Section 6.2. The testbed experiments employ real hardware running in a virtualized wireless environment. The environment and the testbed experiments are covered in Section 6.3. The outdoor experiments offer greater range of mobility and different wireless channel characteristics from the indoor experiments; these are presented in Section 6.4. Finally, in Section 6.5, we present our characterization experiments; in these experiments we isolate variables to examine the *system* performance of our CANCR solution versus SimpleNC, and gauge the system performance tradeoff between the context-enabled CANC Router and the plain SimpleNC Router in situations where context-based adaptation can be expected to have little to no benefit.

6.2 Indoor Experiments

We performed two experiments with the autonomous Pharos nodes, one using three nodes (*Source*, *Mule 1*, and *Sink*), and one using five nodes (*Source*, *Mule 1*, *Intermediate*, *Mule 2*, and *Sink*). See Figure 5.5 for a visual overview of their placement and mobility paths—the dimensions of the hallway were 25m by 42m; in all cases,

only the mules moved (with speeds of 0.57 and 0.65 meters/second for *Mule 1* and *Mule 2*, respectively). In the five-node test, the *Intermediate* was a stationary node otherwise identical to the *Source* and *Sink* in its configuration. In both experiments the source generated a 100MB bundle at the beginning of the experiment, which it split into 1000 fragments to encode over. The effective wireless connectivity distance between nodes was around 10 to 20 meters, less around corners, and the *Source*, *Intermediate*, and *Sink* were mutually disconnected for the duration of all the experiments despite their physical proximity. They could only send data between each other via the mules. We compared our CANC Router (employing the Context Agent Framework) with SimpleNC.

CANC Router Configuration. For the CANC Router, we used the adaptation rules defined in Figure 5.6(b). As shown in Figure 5.5, the mobility space is divided into twelve distinct "squares" for the purposes of geo-tagging context tuples with their Cartesian coordinates. Since this was an indoor test, GPS-based location was impossible, and we relied on the Pharos Mobility Controller to estimate the robot's location as it controlled the robot's movements. In practice, this turned out to be sufficient for these experiments since we did not need a high degree of accuracy in the location estimate given the low node density.

6.2.1 3-Node Indoor Experiments

Figure 6.2 shows the results of the three-node experiment. The graph shows the rank of the decoding matrix at each node vs. time. Although the SimpleNC mule reached full rank before the CANCR mule, the CANCR sink reached full rank more than 1000 seconds before the SimpleNC sink. This was due to to the CANCR Context Agent's control of the rates, which kept the mule from being overwhelmed by encodings from the sink (as happened in SimpleNC). CANCR was able to significantly improve the overhead of network coding by intelligently limiting rates. SimpleNC

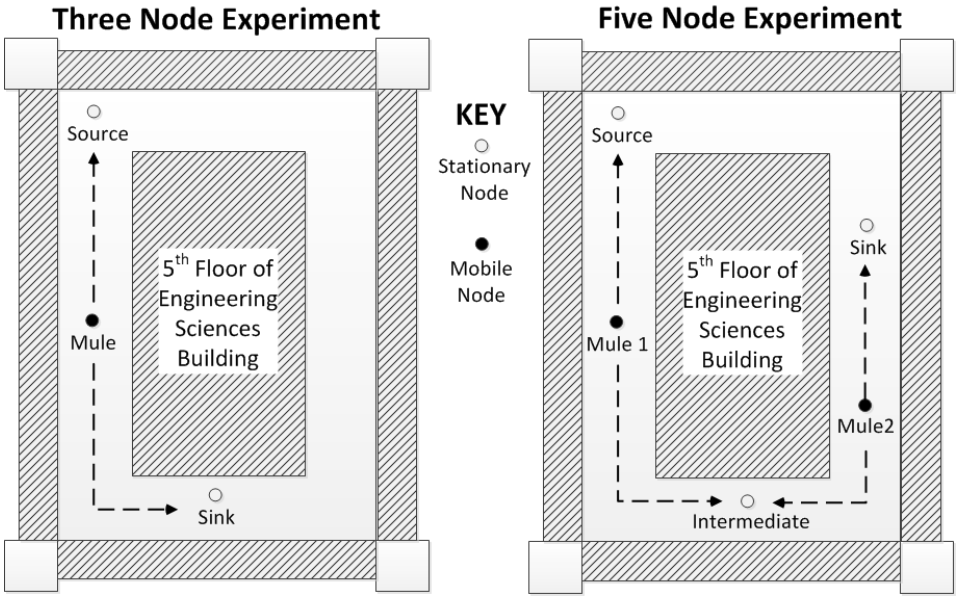


Figure 6.1: Indoor Experiment Setup

resulted in a combined total of 5951 encoded fragment transmissions compared to CANCR's 2149, making CANC almost three times more efficient. CANCR achieved close to the absolute minimum number of transmissions needed, which is 2000 (1000 to the mule and 1000 to the sink).

6.2.2 5-Node Indoor Experiments

The results from the five node experiment, graphed in Figure 6.3, also show that CANCR outperforms SimpleNC. The ranks of the sources and mules are omitted for clarity. The CANCR sink was able to reach full rank faster than the SimpleNC intermediate. The larger performance gain over the three-node experiment is due to the increased congestion at the intermediate, where three nodes (*Mule 1*, *Intermediate*, and *Mule 2*) were often vying for the wireless channel simultaneously. Adapting the send rates in such a situation yielded even greater benefits than when

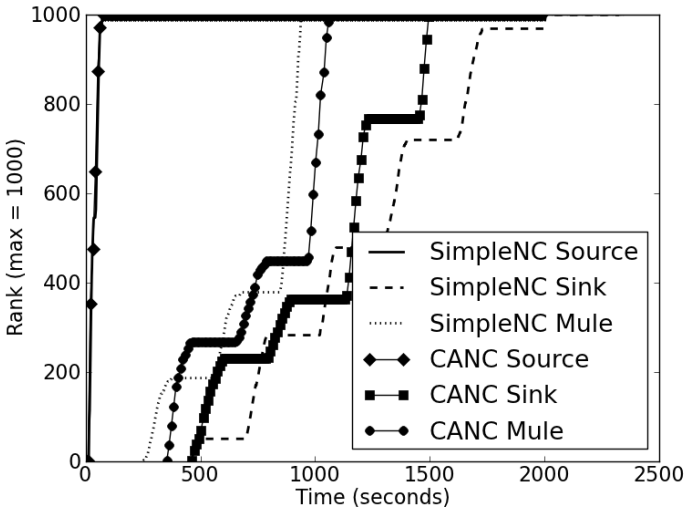


Figure 6.2: Indoor Single-Bundle 3-Node Experiment

only two nodes were connected at once, and we believe that as the number of neighbors grows, the benefits of adaptive rate control will grow as well. Similarly to the three-node experiment, CANCR also provided massive overhead gains; it sent 7149 total encodings across all nodes, compared to SimpleNC's 16765—resulting in a 2.3 times efficiency gain.

6.3 Virtualized Testbed Experiments

We also ran several experiments on the VirtualMeshTest (VMT) mobile wireless testbed [17,29]. VMT allows us to subject Linux-based real wireless nodes with commodity wireless hardware to emulated mobile environments. The wireless testbed is effectively an analog channel emulator based on an array of programmable attenuators. Given a desired physical arrangement of nodes, the system computes the expected path loss between nodes and programs the attenuators to achieve those path loss properties. By updating the attenuations every second, VMT can emulate

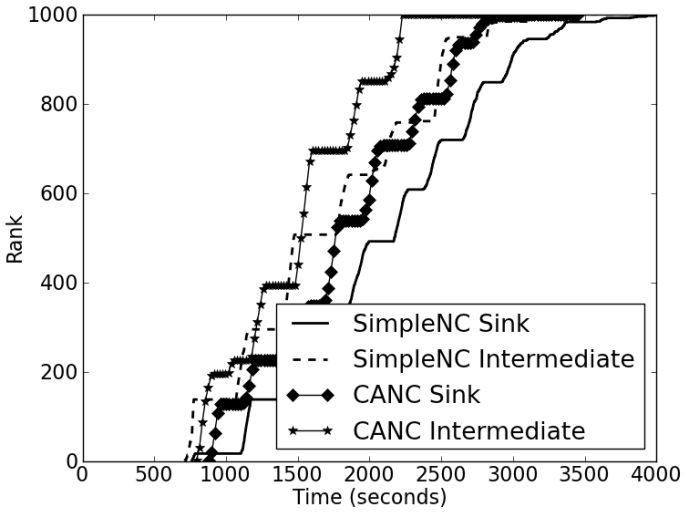


Figure 6.3: Indoor Multi-Bundle 5-Node Experiment

a mobile wireless environment for real wireless nodes.

VMT Experimental Setup. We used the same three-node and five-node scenarios described above with some minor changes. Since the effective range of each node was approximately 500m, the emulated distances had to be much greater to achieve disconnections; however the roles and movements of the nodes remain the same. The stationary nodes were spaced 1200m apart (ensuring that the *Source*, *Intermediate*, and *Sink* depicted in Figure 5.5 were mutually disconnected), and the mobile nodes (*Mule 1*, and *Mule 2*) moved between them at a rate of 10m/s. As with the Pharos experiments, the source created a 100MB bundle at the beginning of the scenario that it split into 1000 fragments to encode over, and as before we compared our CANCR framework against SimpleNC.

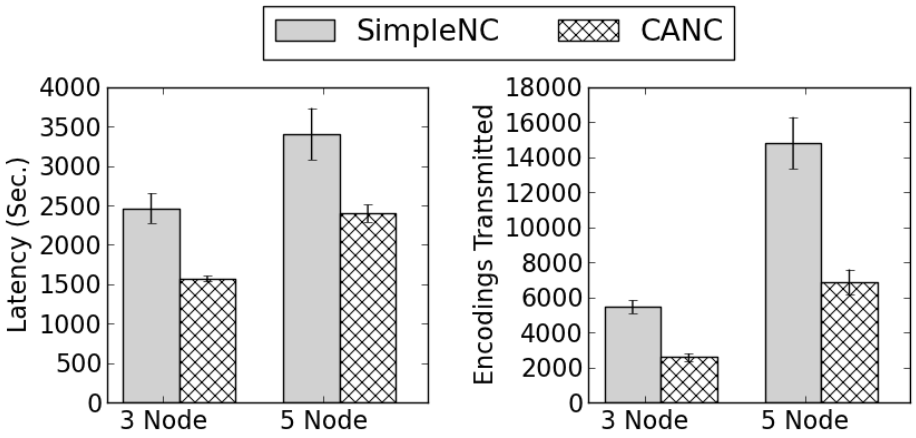


Figure 6.4: Overhead and Latency Results from VMT Testbed

6.3.1 VMT Results

Several runs of both the three-node and five-node experiments were averaged and Figure 6.4 shows the average time it took for the sink to receive enough encodings to decode the 100MB bundle (latency) and the average number of bundles transmitted across all nodes (overhead) in the network with the standard deviations. As was the case for the Pharos results, CANCR resulted in a lower latency and much fewer total transmitted bundles than SimpleNC. It is interesting to note that although the same number of experiments were run for each router, the standard deviations are much smaller for CANCR. This confirms that our CANCR Context Agent's rate adaptation results in more stable and predictable behavior.

6.4 Outdoor Experiments

We performed a series of outdoor experiments using the autonomous robots in the parking lot of the Dell Diamond Baseball Stadium in Round Rock, TX. We tried several variations on the above indoor experiments. Similar to the indoor tests,

there are two or more static nodes separated by enough distance to ensure mutual disconnection, and these distances are traversed by mobile data mules that carried data on behalf of the static senders and receivers. We performed several runs of each experiment. Each outdoor experiment takes between 30 minutes to an hour to complete, and requires another 10-20 minutes to set up in *ideal* conditions. In practice, this setup time was usually much longer. Additionally, the requirement of a large, flat, unobstructed space necessitated travel to the Round Rock Dell Diamond Baseball Stadium (with the entire set of robots, support equipment including generators, laptops, tables, chairs, spare parts, etc.) which limited the number of experiments that could be performed within reasonable time constraints. In total, this dissertation is the culmination of 62 successful real-world experiments and countless failures. We naturally observed a small degree of variability between independent and otherwise identical experiments due to variations in the wireless channel, variations in the exact trajectories taken by the mules, and other factors out of our control. However, the difference between identical experiments was always minimal in comparison to the difference between the CANC Router and SimpleNC. We present a representative selection of the total set of experiments below.

Context Agent Framework Configuration. The outdoor experiments, with noted exceptions, also used the context rules defined in Figure 5.6(b). The mobility space of the nodes, shown in Figure 6.5, was divided into three blocks for the purposes of assigning geo-tagged context values to their respective Cartesian coordinates. A higher degree of granularity is definitely within the capabilities of both the Context Agent Framework and the Pharos Mobility Controller but unnecessary for the purposes of our adaptation rules.

A Note on Robot Naming. For the purposes of the outdoor experiments, the robots were given unique names to identify them. The robots we used are: Guinness, Czechvar, Spaten, Manny, and Ziegen. In all cases, Guinness, Czechvar,

and Spaten were *static* nodes, and Manny and Ziegen were used as the *mobile* nodes. We attempted to maximize comparability between tests by using the same robots in the same locations across all of the tests to account for any variations in their behavior.

6.4.1 3-Node Outdoor Experiments

This experiment is very similar to the three-node indoor experiment described above. One static node (Spaten) is separated by a distance of 245 meters from another static node (Guinness), and one 100MB bundle is sent from one to the other. A mobile node (Ziegen) repeatedly traverses the distance between the two nodes at a rate of 1 m/s until the entire 100MB bundle is successfully transferred. Figure 6.5 shows the positions and mobility paths of the robots.

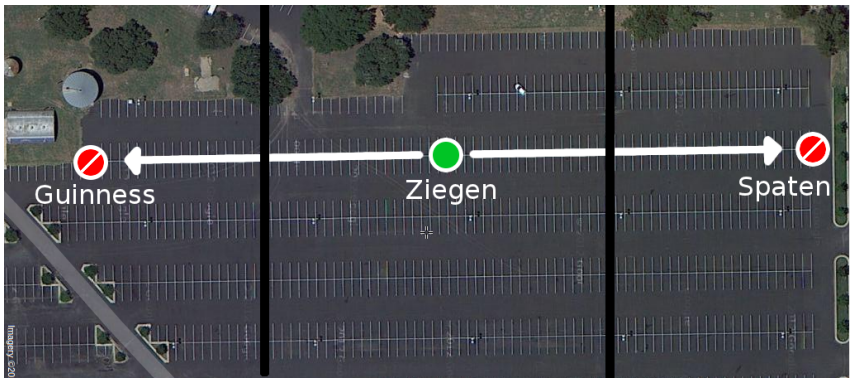


Figure 6.5: 3-Node Outdoor Experiment Setup Showing Three Geographic Regions Mapped by the Context Agent Framework

Single Mule, Single Bundle

In this experiment, only one 100MB bundle is sent between the source and the sink. Figure 6.6 shows the combined graph of the rank at each node vs. the experiment time. Using the adaptive CANC Router, the Sink was able to receive enough linearly

independent encodings to decode the entire bundle in less than half the time it took SimpleNC. These results are consistent with 2 additional runs of this experiment (not graphed).

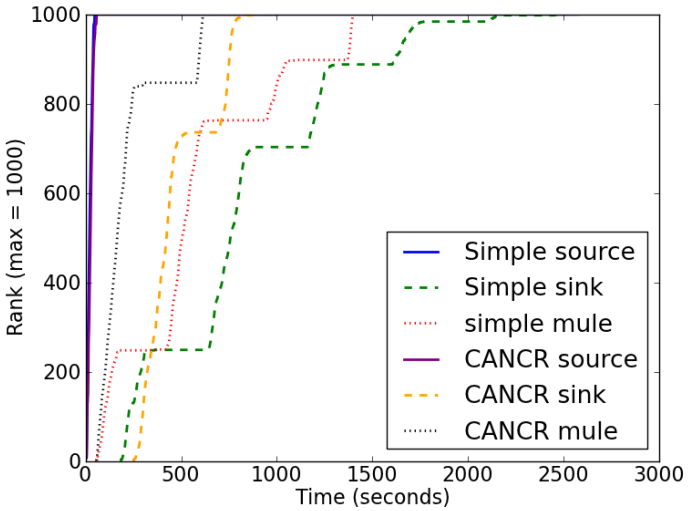


Figure 6.6: Single-Bundle 3-Node Outdoor Experiment

Single Mule, Multiple Bundles

In this experiment, two 100MB bundles are sent. Both Guinness and Czechvar create one 100MB bundle destined for the other at the beginning of the experiment, and a single mule (Ziegen) carries data between the two static nodes. Figure 6.7 shows the results of this experiment. Also graphed is the position of Ziegen along its mobility path—Guinness is located at the 0m position, and Spaten is located at the 245m position. The difference in delivery latency between SimpleNC and the CANC Router is significantly larger than in the single bundle case. This is due to increased efficiency in channel utilization enabled by our context-based adaptive rules. There are several reasons for our increased efficiency. First, the reactive

randomized encoding performed by the bundle sender is fairly resource intensive. Thus, as the number of possible bundles to send grows, it becomes increasingly important to prioritize the "correct" bundle since sending the "incorrect" bundle can starve the system for resources. Second, a sender can *starve* a receiver if the rate at which encodings are transferred is unbounded (as is the case with SimpleNC). This starvation occurs for a similar reason as above. The receiver node is busy decoding the incoming bundles, and becomes resource starved due to the rate at which they are coming in. This becomes a bigger problem as the number of bundles increases. Both of these issues are solved by the *relative rate* rule triggered on the last line of the rules in Figure 5.6, allowing for a much "fairer" distribution of the limited channel capacity between senders and receivers. In a broader sense, this underscores the power of the Context Agent Framework. CAF enables *simple* but *powerful* adaptation strategies due to its open-ended context collection and aggregation capabilities, and the flexible and highly capable design of its interface to the network stack.

6.4.2 4-Node Outdoor Experiments

We also wanted to study the benefits of the Contex Agent Framework model in mutiple-mule situations. We present two sets of such experiments. The first uses multiple mules, with one bundle transferred between Guinness and Spaten. In the second set, we transfer multiple bundles and also vary the wireless signal attenuation. In practice, the wireless range of our Proteus robots is between 100-150 meters depending on conditions. This ensures that the static nodes remain connected to the mobile nodes for long periods of time as the nodes drive up to a static node, turn around, and start driving back towards the other static node. In fact, there is a long period of time when the mobile mule actually acts as a "bridge" between the two static nodes (e.g., the static nodes can both communicate with the mobile

node at the same time). In order to better emulate a real-world delay tolerant network, we introduce hardware signal attenuators to limit the range of the wireless antennas. We present multi-mule, multi-bundle experiments using no attenuation, 6dB attenuators, and 10dB attenuators and discuss the results. The positions of the two static nodes (Spaten and Guinness) are identical to that depicted in Figure 6.5, but instead of a single mule, we have two mules (Ziegen and Manny) which move along the same mobility path.

Multiple Mule, Single Bundle

This experiment is very similar to the 3-node outdoor experiments above, but instead of a single mule we use two mules (Ziegen and Manny) that follow the same mobility path. The two mules start in between the static source and sink (and are thus approximately 120m from each) and head in opposite directions, crossing paths in the middle of the mobility space as they head back and forth between the static nodes. Figure 6.8 shows the decoding matrix rank of the nodes for both SimpleNC and CANCR. As before, the rate adaptation provided by the Context Agent Framework significantly improves the delivery latency at the destination.

Multiple Mule, Multiple Bundle

The following three experiments use the same two mules, but instead of a single bundle sent from one static node to the other, we send two bundles (each static node acting as the source for one 100MB bundle, and the destination for the other). Figure 6.9 shows the results with no hardware attenuators attached to the 802.11 wireless radios, Figure 6.10 shows the results with 6dB hardware attenuators, and Figure 6.11 shows the same experiment with 10dB hardware attenuators. The results show that at no attenuation and 6dB, the results have similar trends. Both show CANCR providing a significant improvement to the delivery latency of bundles

(examining the rank of the destination node). However, at 10dB attenuation, the difference between CANCR and SimpleNC is minimal. This is due to the poor channel characteristics. We performed further tests to ensure our hypothesis is correct; at 10dB the wireless channel becomes the dominant bottleneck in communication due to the high rate of transmission errors; neither node is able to send enough encodings to guarantee the delivery of the complete 100MB bundle. Neither this experiment, nor any other 10dB attenuated experiment was able to completely deliver a single 100MB bundle, and under such extreme wireless channel conditions, the effect of encoding rate adaptation is minimized.

6.4.3 5-Node Outdoor Experiments

The final set of outdoor experiments is an analog to the 5-node indoor experiment described in Section 6.2. The positions and waypoints of the nodes are shown in Figure 6.12. A single source (Guinness) generates two 100MB bundles, one destined for Czechvar, and one destined for Spaten. Two mules operate in this experiment: Ziegen moves between the Guinness and Czechvar, and Manny moves between Guinness and Spaten. We used 6dB attenuators to limit the wireless range of the nodes to ensure that all three static nodes were mutually disconnected from one another, and to ensure that the mobile mules could not act as bridges between the *Sources* and *Sinks*. Figure 6.13 shows the results from one run of this experiment. Given that all of the nodes grow their rank for both bundles, the graphs clearly indicate that SimpleNC spreads encodings around randomly (as is intended according to the theoretical research behind network coded routing for DTNs). On the other hand, CANCR is able to prioritize which bundle should go to which mule after the experiment has been running for a short while. This highlights the importance of sharing the *World View*s between nodes. The *Source*, through the collection of information diversity samples by the mules and the sharing thereof, is able to learn of the

identities and locations of the two *Sinks*. From that point on, it is able to use the *destination* location of the mules to prioritize which bundle should be sent to which mule. This context-based adaptation was enabled by the addition of a single rule to the CANCR Context Agent to consider both the intended path of the mobile node as well as the location of the sink for a particular bundle (if they are known). The rule is implemented as follows:

```
if neighbor.type == MOBILE AND neighbor.path includes sink.location:
→rate = MAX_RATE;
else:
→rate = 0;
```

In fact, we made this change to the CANCR Context Agent *in the field*, underscoring yet another benefit of our framework. Do to the decoupling of context gathering and context-based adaptation rule evaluation, new context-based network stack adaptations can be added quickly and easily by adding simple logical statements evaluated against existing context elements. This is a powerful feature of the CAF, since inevitably even researchers who carefully design adaptive routers will find interesting and new ways to adjust the routers' parameters during deployments. Our system allows for these to be done *outside* the code-base for the router, allowing for quickly prototyping new adaptation rules without recompiling or re-deploying the entire code base. The expressiveness of our rule-based context adaptations make them more powerful than simple configuration options, allowing for greater control of the router logic through our general Adaptation Portal interface.

6.5 System Characterization Experiments

We also performed a series of experiments to isolate aspects of the system implementation in order to better understand the differences between SimpleNC and CANCR. In order to isolate the wireless channel as a potential bottlenck, we remove the wire-

less links and replace them with wired connections thus connecting all of the nodes together through a single switch. In the second experiment, we remove the mobile nodes and instead place four static nodes within wireless range of each other. This second experiment removes mobility as a variable and instead shows what would happen if the delay-tolerant mobile network ever became a well-connected ad-hoc network for a period of time. In all of these experiments the World View "sees" all of the nodes as occupying the same coordinate within the grid representing the entire experimental space.

6.5.1 Comparison using Wired Experiments

For this experiment four nodes (Spaten, Czechvar, Manny, and Ziegen) were connected via Gigabit ethernet. Two 100MB bundles were sent, one from Spaten to Czechvar and one from Czechvar to Spaten. Figure 6.14 shows the results from this experiment. The difference in delivery latency between CANCR and SimpleNC is less drastic. This is due to the fact that efficiently utilizing the limited channel is no longer a factor. Instead, as was alluded to in previous experiments, the resource utilization during encoding and decoding becomes the dominant factor in the overall delivery latency. CANCR is able to prioritize delivery of encodings to the "correct" node (e.g., the destination of the bundle) and thus yields faster delivery latencies to the sinks.

6.5.2 Non-Mobile Wireless Outdoor Experiments

We also wanted to eliminate node mobility as a factor in the delivery latency, and ran several experiments with the same four nodes connected via wireless links, but all within range of each other. Figure 6.15 shows the results from one such experiment. It is clear that the CANC Router is able to deliver the bundles to their respective destinations faster because it wastes less of the precious wireless band-

width sending encodings to the non-destination nodes (Manny and Ziegen). Since overall information diversity is the goal of our network-coded implementation, all of the nodes receive all of the encodings regardless, but the node to which the bundle is destined is given priority through the context-based rate adaptation rules.

6.6 Discussion of Results

These results clearly show that a network-coding router for delay-tolerant networks can be improved through the external manipulation of its parameters using our Context Agent Framework. More importantly, they show that our architectural ideas and framework developed in the previous chapters work in real-world situations. In our experiments, we have focused on a few simple context types: *information diversity*, which represents the total rank of the decoding matrix for a given encoded bundle, and *location*, which represents the geographical location of nodes, location of information diversity samples, and the mobility paths of the mobile mules. Using our general purpose Context Agent Framework, we were able to collect these context elements and share them among nodes as they moved through the network. Using a purpose-built context agent (the CANCR Context Agent), we were able to use the collected context to change the operating parameters of the CANC Router in order to improve the overall latency and overhead of moving information between the sources and sinks, specifically by changing the rate at which nodes send encodings to each other. We were able to improve the usage of the limited shared wireless channel.

This chapter is essentially a case-study for one possible use of the Context Agent Framework. There is no limit to the types of context-based adaptation that could be implemented using rules as simple as our own and no limit to the number and types of routing protocols that can be adapted using our DTN2 Adaptation Portal. The most important contribution of this chapter is to *validate* that

our implementation works under real-world constraints: real mobile nodes, a complete Linux-based system, commodity wireless hardware, using popular and widely-deployed delay-tolerant transport protocols, and under realistic data loads. We have met this goal through a variety of experiments intended to test the capabilities of our complete system implementation.

6.7 Research Contributions

This chapter makes the following contributions:

Research Task 4: Use the Pharos mobile computing testbed to design and perform a series of real-life application validation and evaluation studies using the system developed in Research Task 3.

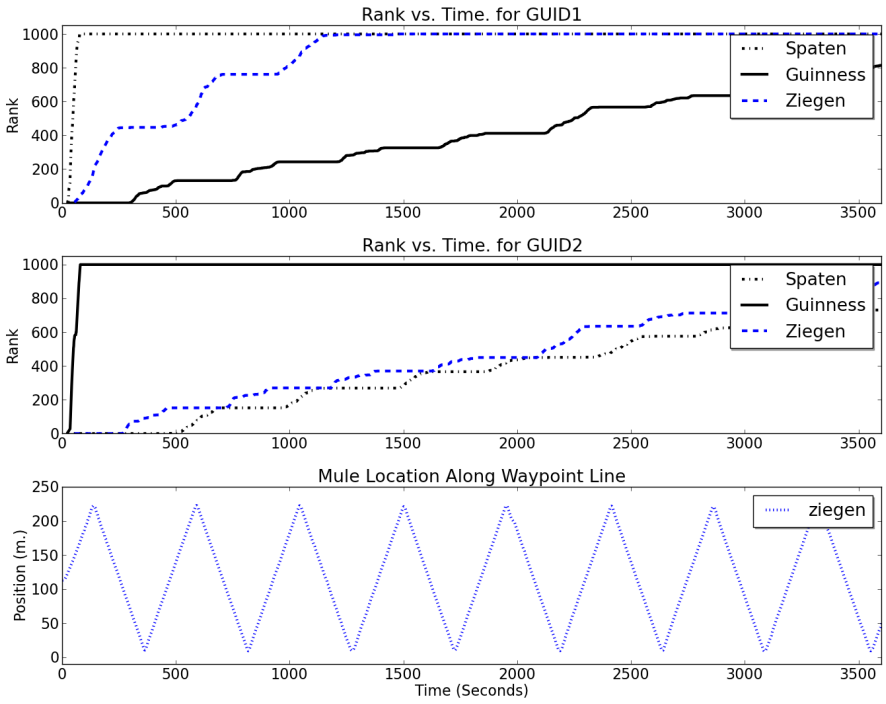
6.8 Impact

To our knowledge, these are the first experiments to use externally-gathered context information to manipulate a delay-tolerant routing protocol. They are also the first experiments to manipulate the parameters of a delay-tolerant routing protocol *during* operation in response to changing network situations and represent the first validation of a system that interfaces an external application (the Context Agent Framework) to the widely-used DTN2 Reference Implementation of the Bundle Protocol in order to affect its behavior.

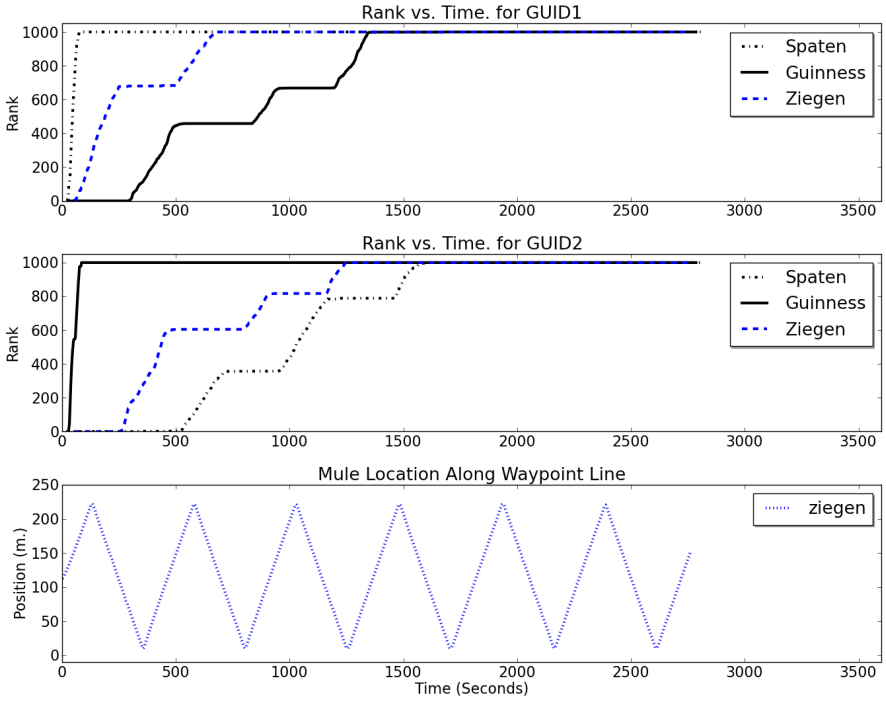
6.9 Chapter Summary

This chapter presented the validation of our complete systems solution for delay-tolerant networks. We presented a broad selection of experimental results gathered with the implementation described in Chapter 5 using autonomous robots from the

Pharos Testbed. We showed that our system is able to adapt the delay-tolerant network stack (employing the popular DTN2 Reference Implementation as the stack itself) on commodity hardware using mobile nodes connected over 802.11 b/g wireless radios. This chapter also showed the power of the Context Agent Framework by comparing network coded routing characteristics (throughput, latency, overhead) to an adaptive network coded router controlled by an agent within the CAF. We showed how even fairly minimal adaptation rules can result in significant improvements in delay-tolerant networks.

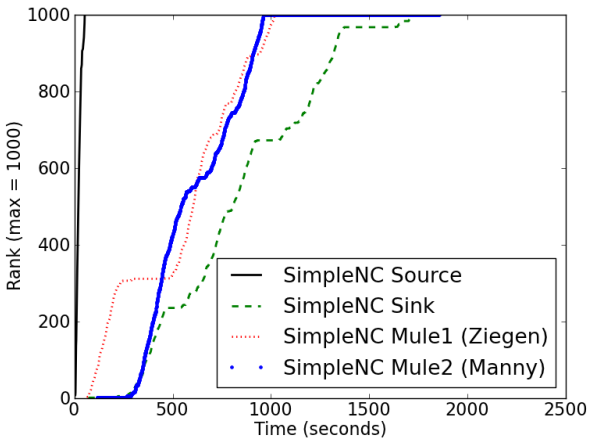


(a) 1 Mule, 2 Bundles, No Attenuation, SimpleNC Router

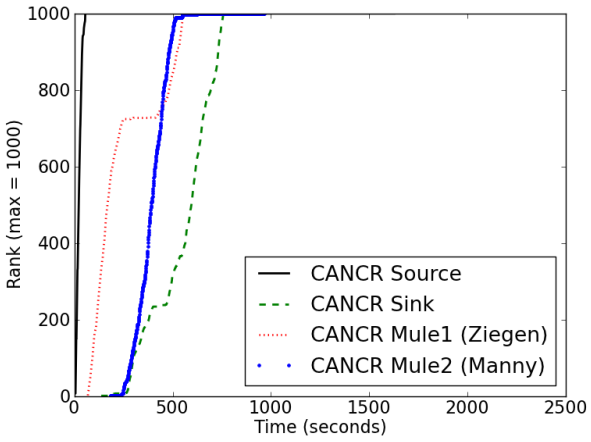


(b) 1 Mule, 2 Bundles, No Attenuation, CANC Router

Figure 6.7: Multi-Bundle 3-Node Outdoor Experiments (0dB)

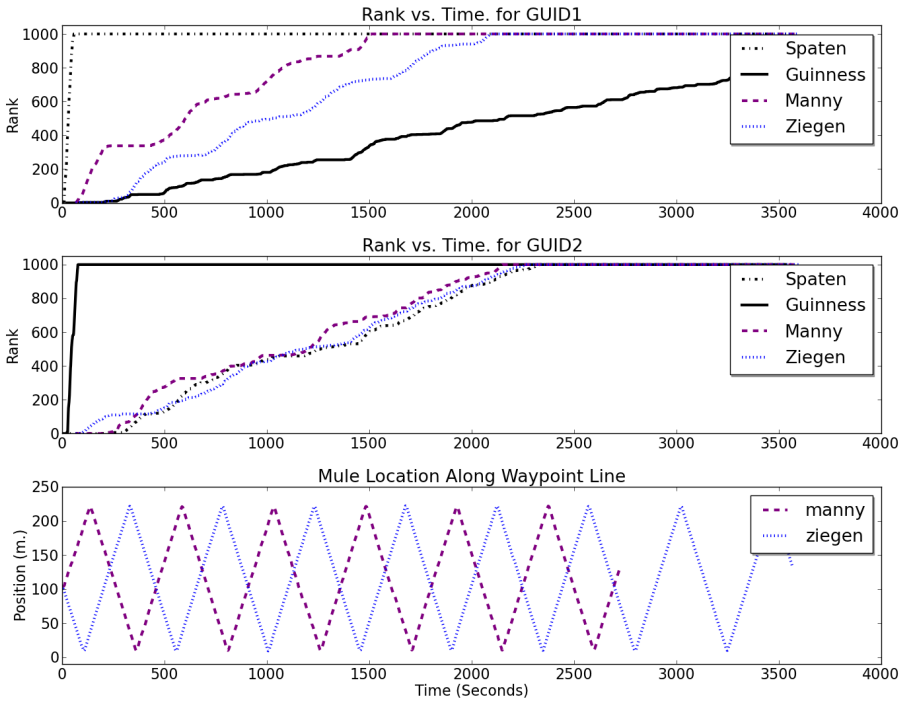


(a) 2 Mules, 1 Bundle, No Attenuation, SimpleNC Router

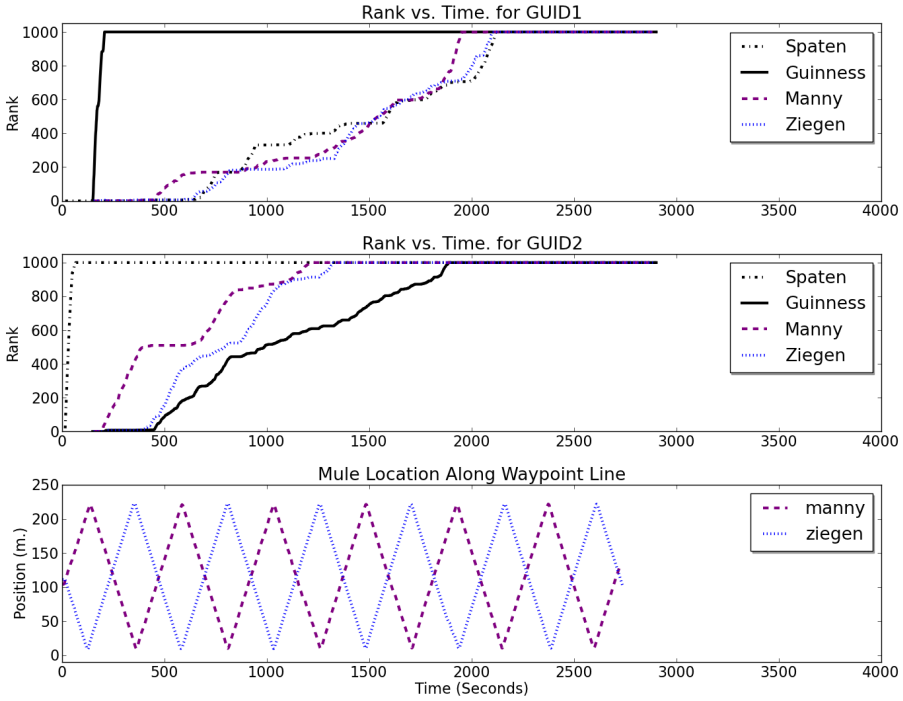


(b) 2 Mules, 1 Bundle, No Attenuation, CANC Router

Figure 6.8: Single-Bundle 4-Node Outdoor Experiments (0dB)

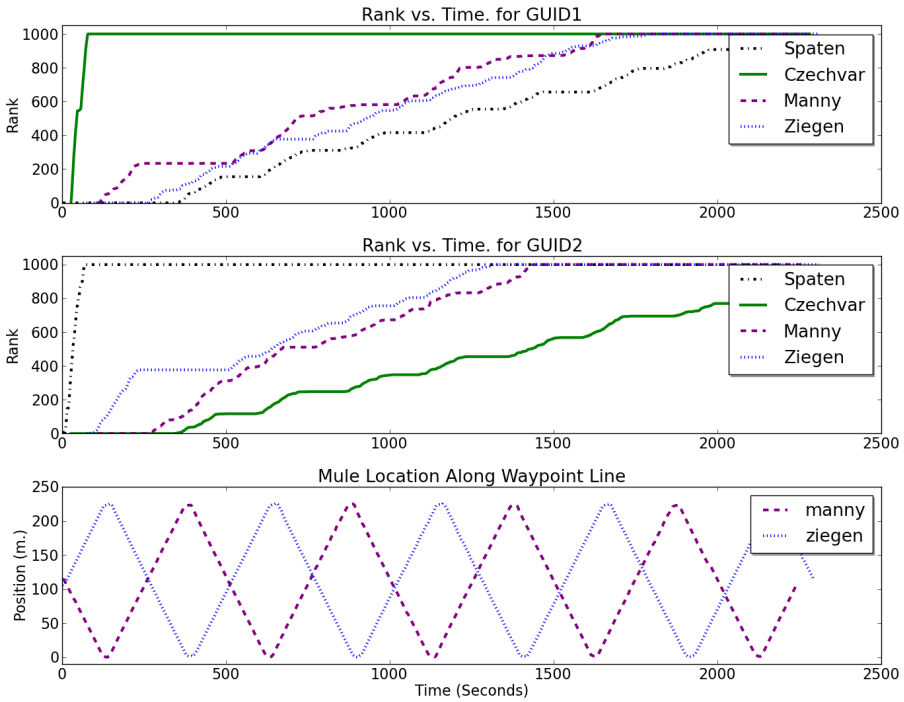


(a) 2 Mules, 2 Bundles, No Attenuation, SimpleNC Router

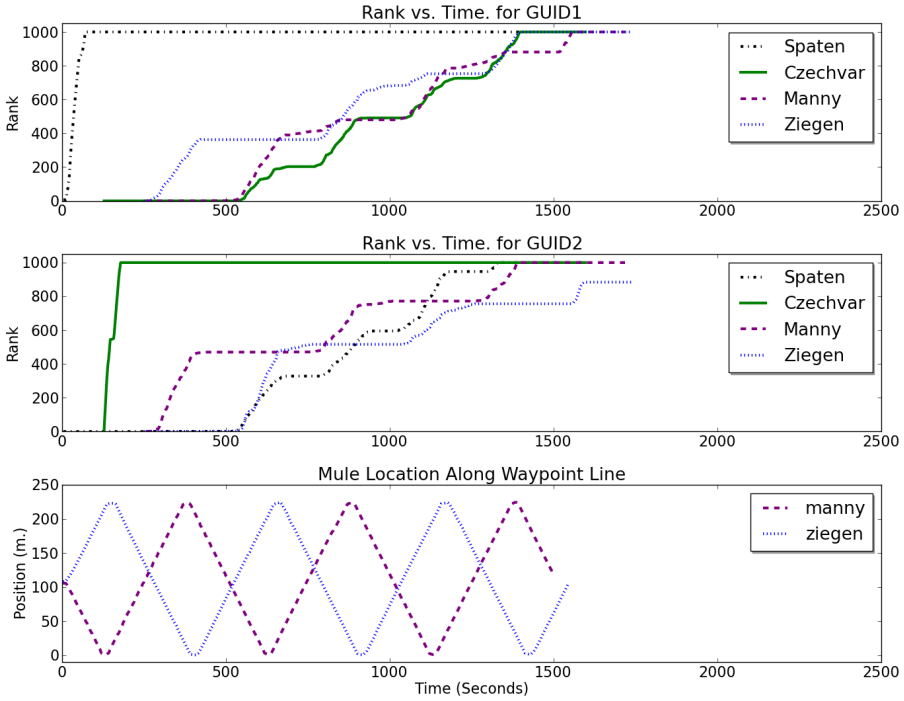


(b) 2 Mules, 2 Bundles, No Attenuation, CANC Router

Figure 6.9: Multi-Bundle 4-Node Outdoor Experiments (0dB)

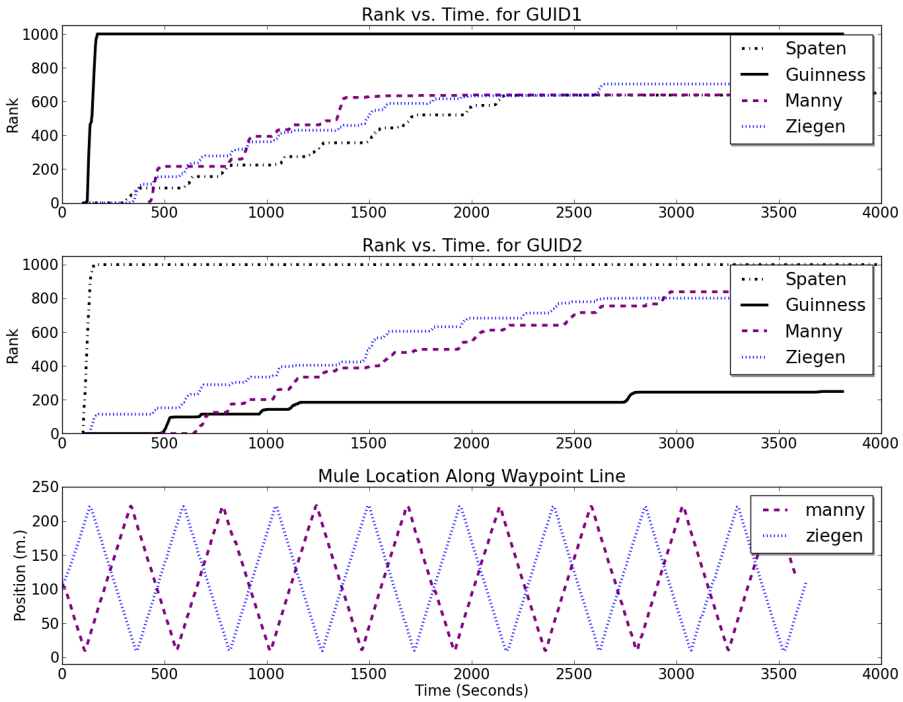


(a) 2 Mules, 2 Bundles, 6dB Attenuation, SimpleNC Router

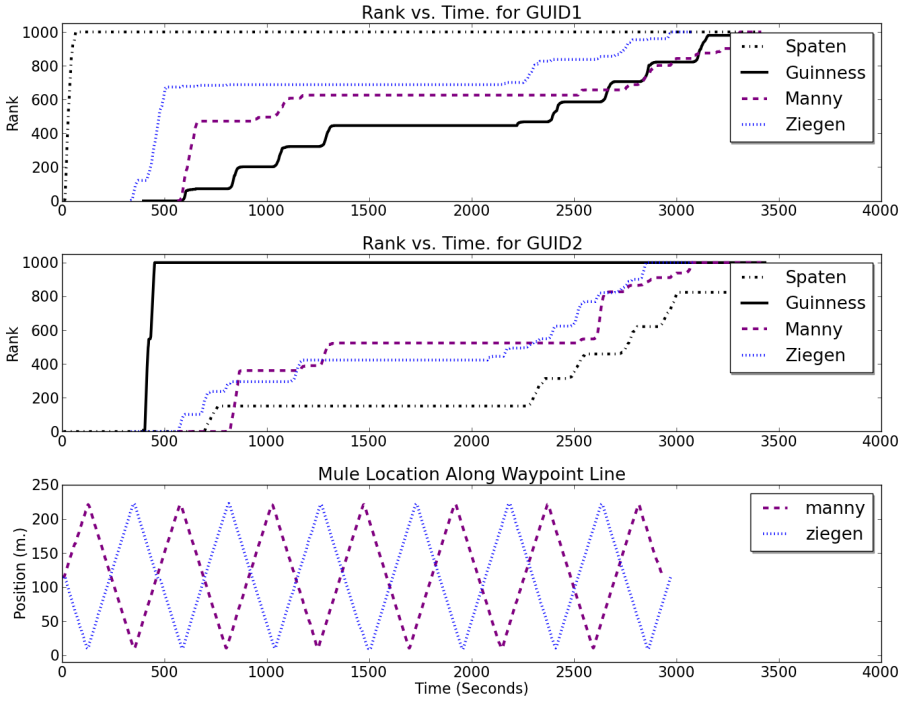


(b) 2 Mules, 2 Bundles, 6dB Attenuation, CANC Router

Figure 6.10: Multi-Bundle 4-Node Outdoor Experiments (6dB)



(a) 2 Mules, 2 Bundles, 10dB Attenuation, SimpleNC Router



(b) 2 Mules, 2 Bundles, 10dB Attenuation, CANC Router

Figure 6.11: Multi-Bundle 4-Node Outdoor Experiments (10dB)

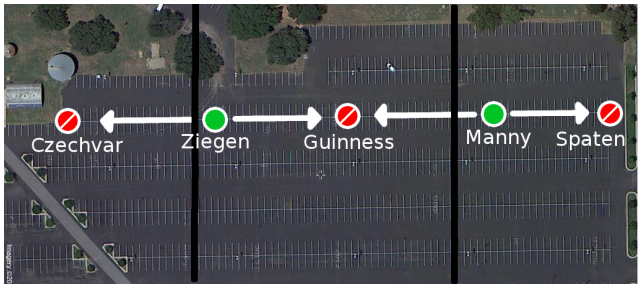
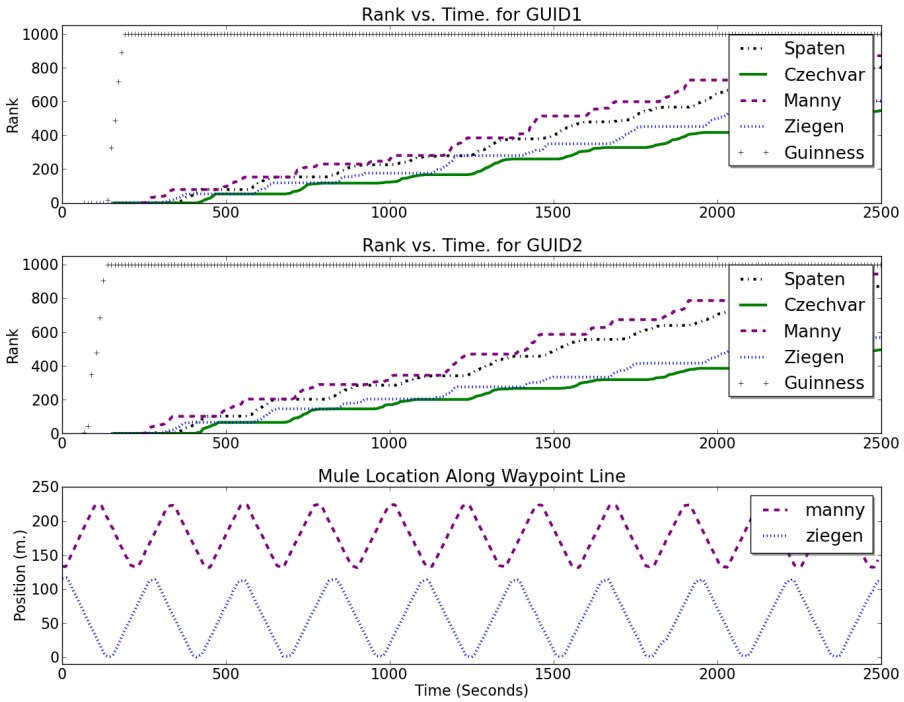
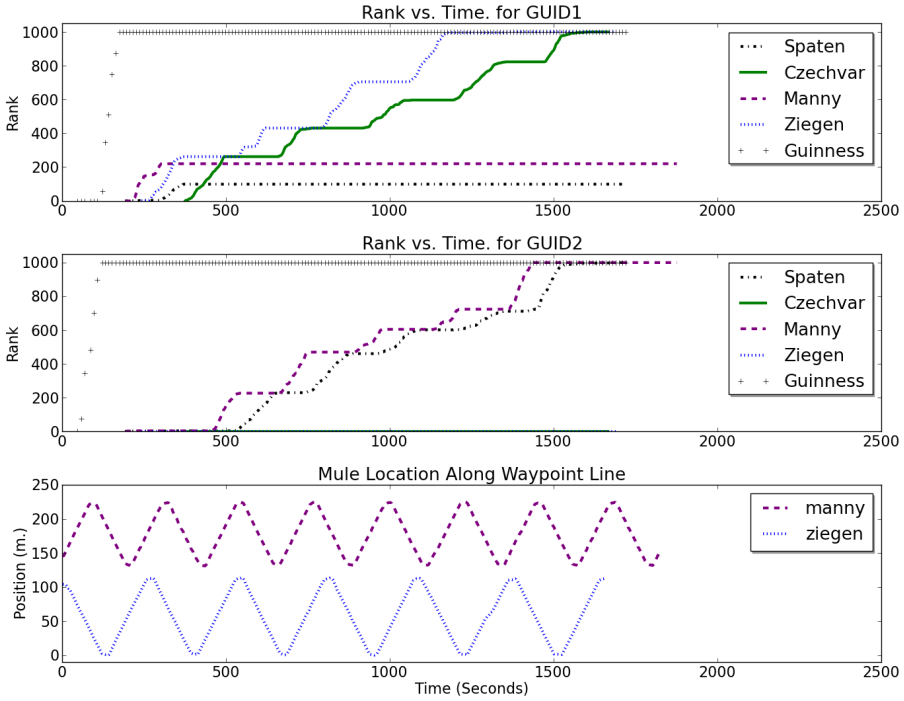


Figure 6.12: 5-Node Outdoor Experiment Setup

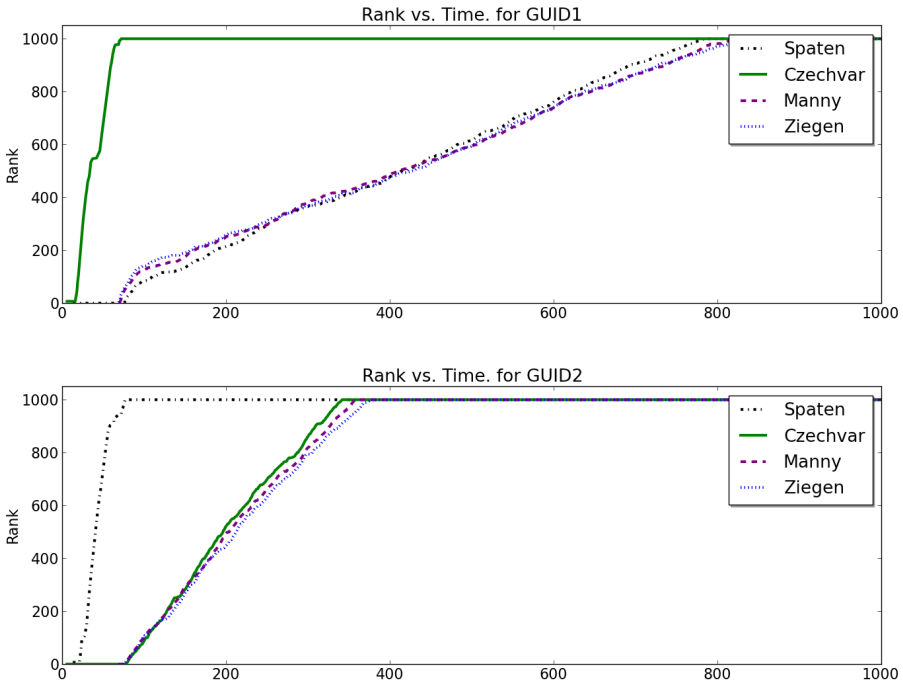


(a) 2 Mules, 2 Bundles, 6dB Attenuation, SimpleNC Router

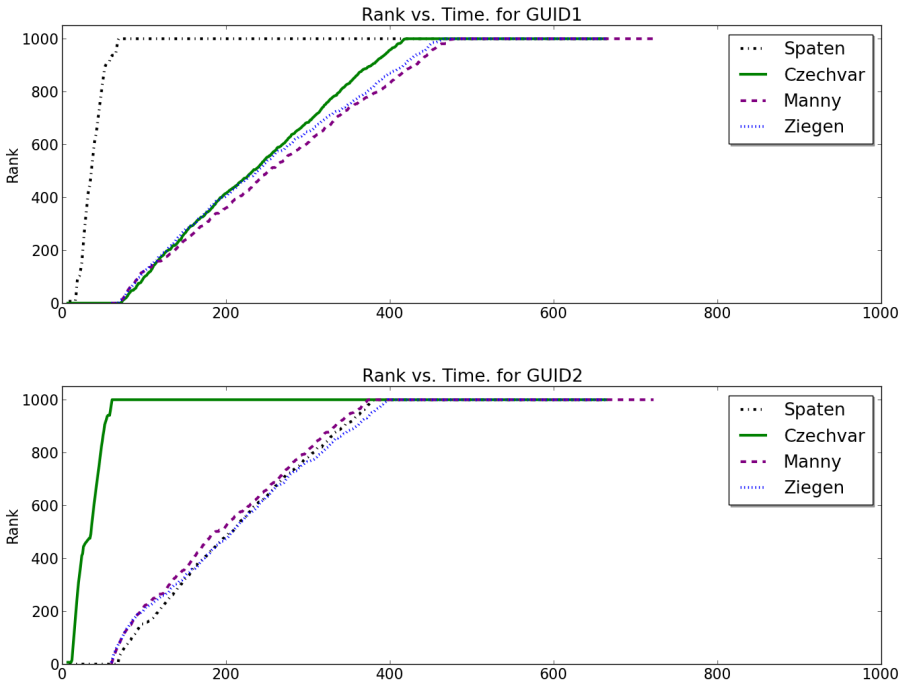


(b) 2 Mules, 2 Bundles, 6dB Attenuation, CANC Router

Figure 6.13: Multi-Bundle 5-Node Outdoor Experiments

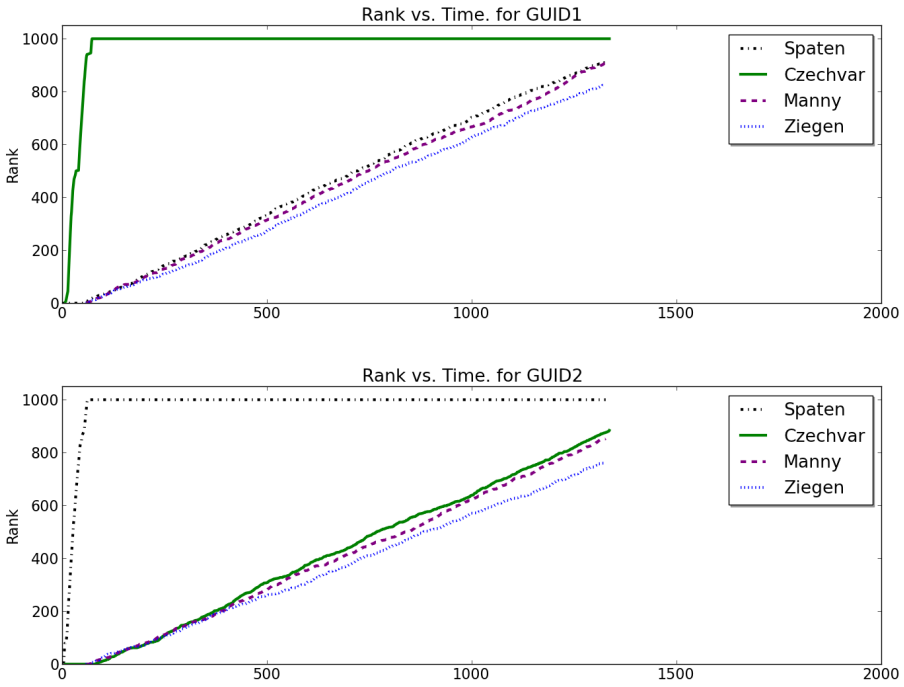


(a) 4 Static Nodes, Wired, SimpleNC Router

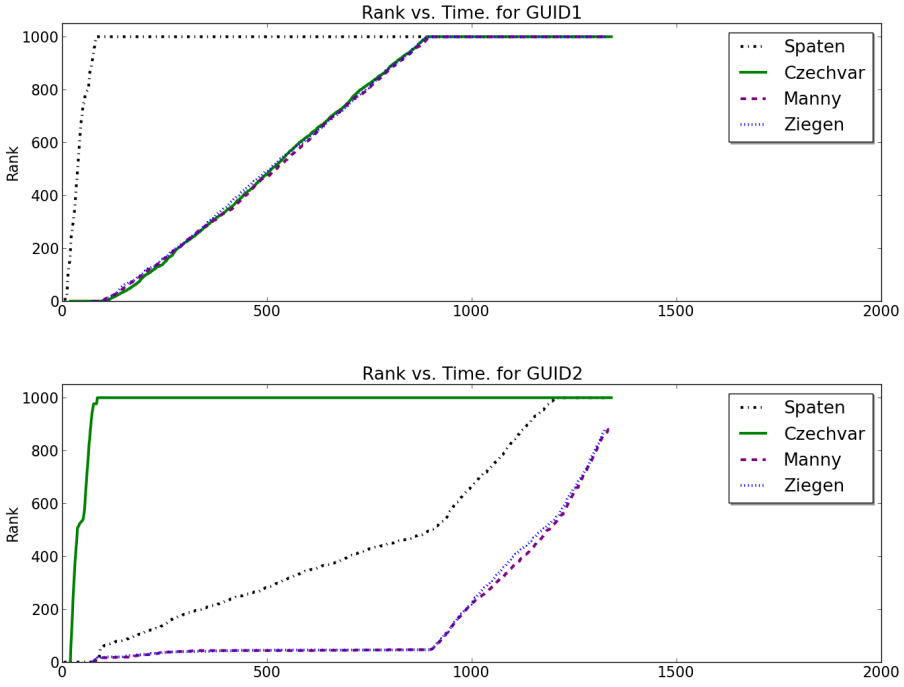


(b) 4 Static Nodes, Wired, CANC Router

Figure 6.14: Static, Wired Indoor Experiments



(a) 4 Static Nodes, Wireless, Outdoors, SimpleNC Router



(b) 4 Static Nodes, Wireless, Outdoors, CANC Router

Figure 6.15: Static, Wireless, Outdoor Experiments

Chapter 7

Conclusion

In this dissertation, we have taken a practical systems-oriented approach to solving problems in delay-tolerant networks using context-driven adaptation of the network stack. In DTNs, nodes are intermittently connected and mobile, thus the topology of the network is subject to constant changes. Since there is no guarantee of end-to-end connectivity between communicating parties, the general approach towards routing in these types of networks has taken the form of probabilistic *store-and-forward* algorithms. DTNs are an "extreme" form of mobile ad-hoc networks (MANETs) and potentially apply to rural networking situations, disaster recovery scenarios, and even space communications. The commonality is that connectivity is widely varying and unpredictable.

However, communicating parties operating in delay-tolerant networks would like their connections supported by the *best* possible technologies for the given networking situation. Current approaches generally deploy a single set of protocols with a pre-determined set of parameters to support these networks. However, since delay-tolerant networks are constantly changing due to node mobility, we argue it is better to be adaptive in response to changing networking conditions. This dissertation uses context collection, aggregation, and adaptation to improve the behavior of

delay-tolerant network stacks through the context-based adaptation of their composition and behavior. This entailed solving several novel challenges including 1) how to efficiently sense the appropriate network and data context to allow for adaptation decisions; 2) how to aggregate and organize context to allow for intuitive adaptation strategies to be developed and easily deployed; 3) how to enable intelligent cross-layer design to support tuning the network stack; and 4) how to provide seamless transitions for applications as the network stack changes. We have focused on the mechanics of the adaptations and the mechanics of context collection, aggregation, and sharing. We have explored two separate architectures (DynSS and MaDMAN) to tackle this sort of adaptation and have provided proof-of-concept implementations and evaluations of both of them. We have also designed and implemented a general-purpose context aggregation system, the Context Agent Framework, that provides the mechanism by which new context types can be created and new adaptation can be implemented in Linux-based delay-tolerant networks. We have proven the benefits of context-based stack adaptation for traditional delay-tolerant networks and have provided use cases even for non-traditional DTNs (MADServer). We have also combined the Context Agent Framework with a complete delay-tolerant networking stack, and have provided extensive evaluation of our implementations using autonomous robots from the Pharos Testbed. Our middleware prototypes represent the first systems of their kind in the delay-tolerant research community, and our results clearly show the benefits of our approach.

7.1 Future Work

This dissertation opens the door for many exciting future research possibilities. The following is a selection of potential future work enabled by this dissertation.

7.1.1 Optimizing Context Sharing for Delay-Tolerant Networks

This dissertation provides a general framework for the collection and dissemination of context as well as for context-based adaptation. However, as the number of context types and context samples grows, our Context Agent Framework entails considerable network overhead in sharing the context (the World View) between nodes. There is further work to be done on efficiently summarizing context to reduce the load on network resources and developing mechanisms to share context efficiently. For example, Grapevine [16] summarizes and shares simple context; devising a similar efficient summary data structure for the World View is still unexplored. Any such approaches could be easily incorporated into the Context Agent Framework in the future.

7.1.2 Complete Taxonomy of Context Types Relevant to DTNs

Another avenue for future work lies in developing a complete taxonomy of context types relevant to delay-tolerant networks. We have explored many types of context in this dissertation including information diversity, node density, network load, relative mobility, geographical location, and destination, but there are many more relevant types to be explored. Work in completing the *context picture* is orthogonal to this dissertation. However, any further types could be easily incorporated into the Context Agent Framework. In fact, we believe the CAF would be a useful tool to any researchers who seek to develop new context types and to define mechanisms by which they can be collected, aggregated, and stored. The primitives for such operations are already available in the CAF.

7.1.3 Improving Network Coded Routing Through Further Context-Based Adaptation

In this dissertation we have explored a number of context-based adaptations for network coded routing in delay tolerant networks. Specifically we have adapted the *rate*, *weight*, and *balance* between different senders and different sets of encodings according to context changes in node mobility and information diversity. Further adaptations of network coded routing are no doubt possible, and, although we have had great success with our adaptation, it is likely that further context-based adaptation can yield even greater benefits to the overall delivery latency and network bandwidth utilization.

7.1.4 Exploration of Further Context-Based Network Protocol and Physical/Link Layer Selection

Although we have explored many possibilities for adapting delay-tolerant network stack composition using context, there are possibilities for further research in this direction. We do not claim to have found all of the useful ways and mechanisms by which delay-tolerant network stacks can be combined and recombined in response to changing network conditions. Rather, this dissertation highlights that such adaptation can be beneficial to nodes operating in delay-tolerant environments. A fruitful avenue for future work lies in exploring additional ways in which context can inform stack recomposition. These would possibly include adaptating properties of the physical layer and link layer protocols in addition to further network, transport, and routing protocol adaptations.

7.1.5 General-Purpose Adaptive Delay-Tolerant Routing Protocol

Finally, this dissertation paves the way for a general-purpose adaptive delay-tolerant routing protocol. We have shown that our Context Agent Framework can effectively

control the behavior of routers built in the DTN2 Reference Implementation through our implementation of the Context-Aware Network Coded Router (CANC Router). We believe that many existing DTN routing protocols such as PRoPHET [31], Spray-and-Wait [63], and even the mobility-assisted Spray-and-Focus [62] could be more efficiently (in terms of the number of lines of code and debugging complexity) implemented using a combination of a simple *base* DTN router and the Context Agent Framework. All of these routing protocols are all built using similar mechanics—some local decision factors in to whether or not a particular bundle (or fragment of a bundle) is forwarded to some particular neighbor at any given time, and they all have different means by which this decision is reached. The behavior of each of these routing protocols, and many interesting new ones, could be coded within the CAF using simple context-based rules. If this was done, one general-purpose bundle router within DTN2 could be used to implement all of them. This approach would have several key advantages. First, a higher level language (Perl) could be used to write the rules that determine if and when a given bundle should be forwarded, reducing the programming complexity. Second, routing protocol developers would not have to separately collect and aggregate every context type they need to make routing decisions; many would already be available in the World View and made available through the publish/subscribe semantics of the CAF. Finally, a unified general-purpose *base* router could serve the needs of every DTN routing protocol thus resulting in less redundancy between routers and a more stable codebase. This is a significant piece of future work, and would leverage our general purpose CAF along with our Adaptation Portal architecture for DTN2 to solve many routing protocol design issues for delay-tolerant researchers.

7.2 Dissertation Summary

In summary, this dissertation addresses several challenges of supporting applications in a dynamic delay-tolerant network setting. We examine how to automatically adapt connections and underlying network protocols, how to collect context efficiently, how to organize the context and make it available to programmers who wish to adapt the network stack, how to incorporate context-based network adaptation with existing delay-tolerant network systems (i.e., the DTN2 Reference Implementation), and we show how to design network protocol adaptation algorithms using out solution. Our solutions represent the first application of general-purpose context-based adaptation in delay-tolerant networks, and our prototypes are the first attempts at building such solutions into real-world systems. We are also the first to validate DTN solutions using autonomous robots in a real-world setting. As such, this dissertation represents a significant step in the evolution of delay-tolerant network systems and paves the way for new and exciting research ideas that can be designed leveraging our novel architectures and built using our frameworks.

Bibliography

[1] A. Caro and J. Zinky. DTN erasure coding protocol (preliminary notes). To appear as Internet Draft.

[2] E. Altman, F. De Pellegrini, and L. Sassatelli. Dynamic control of coding in delay tolerant networks. In *Proc. of INFOCOM*, 2010.

[3] A. Baig, M. Hassan, and L. Libman. Prediction-based recovery from link outages in on-board mobile communication networks. In *Proc. of IEEE Globecom*, pages 1575–1579, 2004.

[4] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *Proceedings of MobiSys*, 2010.

[5] P. Basu, N. Khan, and T. D. Little. A mobility based metric for clustering in mobile ad hoc networks. In *Proc. of the International Workshop on Wireless Networks and Mobile Computing (WNMC2001*, pages 413–418, 2001.

[6] A. Begen, T. Akgul, and M. Baugher. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing*, 15:54–63, 2011.

[7] C. Boldrini, M. Conti, J. Jacopini, and A. Passarella. Hibop: a history based routing protocol for opportunistic networks. pages 1 –12, jun. 2007.

[8] Bundle protocol reference implementation. `http://www.dtnrg.org/wiki/Code`.

[9] M. Caparuscio, A. Carzaniga, and A. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. on Software Eng.*, 29(12):1059–1071, Dec. 2003.

[10] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[11] M. Chuah, P. Yang, and Y. Xi. How mobility models affect the design of network coding schemes for disruption tolerant networks. In *Proc. of NetCod*, 2009.

[12] S. Farrell, A. Lynch, D. Kutscher, and A. Lindgren. Bundle protocol query extension block. Technical Report draft-farrell-dtnrg-bpq-01, IRTF, Mar. 2012.

[13] A. Ferscha, S. Vogl, and W. Beer. Ubiquitous context sensing in wireless environments. In *Distributed and Parallel Systems: Cluster and Grid Computing*, volume 706 of *The Springer Int'l Series in Eng. and Comp. Science*, pages 98–106, 2002.

[14] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, 1985.

[15] M. Gonzalez, C. Hidalgo, and A. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, 2008.

[16] E. Grim, C. Fok, and C. Julien. Grapevine: Efficient situational awareness in pervasive computing environments. In *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 475 –478, march 2012.

[17] D. Hahn, G. Lee, B. Walker, M. Beecher, and P. Mundur. Using virtualization and live migration in a scalable mobile wireless testbed. *SIGMETRICS Perform. Eval. Rev.*, 38:21–25, January 2011.

[18] K. Harras, K. Almeroth, and E. Belding-Royer. Delay tolerant mobile networks dtmns: Controlled flooding in sparse mobile networks. In *Proc. of the 4th Int'l. IFIP-TC6 Networking Conf.*, pages 1180–1192, May 2005.

[19] K. Harras, M. Wittie, K. Almeroth, and E. Belding. Paranets: A parallel network architecture for challenged networks. pages 73 –78, mar. 2007.

[20] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, and W. Regschitzegger. Context-awareness on mobile devices: the hydrogen approach. In *Proc. of HICSS*, 2003.

[21] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6):643–652, Nov. 2004.

[22] O. J and D. Kutscher. A disconnection-tolerant transport for drive-thru internet environments. In *Proc. of IEEE INFOCOM*, pages 1849–1862, 2005.

[23] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. *ACM SIGCOMM Computer Communication Review*, 34(4):145–158, October 2004.

[24] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *Proc. of NSDI*, pages 267–280, 2006.

[25] P. Juang, H. Oki, W. Want, M. Maronosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. *ACM SIGPLAN Notices*, 37(10):96–107, October 2002.

[26] C. Julien. The context of coordinating groups in dynamic mobile networks. In *Proc. of COORDINATION*, 2011.

[27] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. SeeMon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proc. of Mobisys*, pages 267–280, 2008.

[28] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft. XORs in the air: Practical wireless network coding. In *Proc. of SIGCOMM*, 2006.

[29] Y. Kim, K. Taylor, C. Dunbar, B. Walker, and P. Mundur. Reality vs emulation: Running real mobility traces on a mobile wireless testbed. In *HotPlanet 2011 (to appear)*, 2011.

[30] Y. Lin, B. Li, and B. Liang. Efficient network coded data transmissions in disruption tolerant networks. In *Proc. of INFOCOM*, pages 1508–1516, 2008.

[31] A. Lindgren, A. Doria, and O. Schelen. Probabilistic routing in intermittently connected networks. In *Proc. of the 1st Int'l. Wkshp. on Service Assurance with Partial and Intermittent Resources*, pages 239–254, 2004.

[32] M4RI(e). `http://m4ri.sagemath.org/`.

[33] H. Madhyastha, A. Venkataramani, A. Krishnamurthy, and T. Anderson. Oasis: An overlay-aware network stack. *SIGOPS Operating Systems Review*, 40(1):41–48, January 2006.

[34] Y. Mao, B. Knutsson, H. Lu, and J. Smith. DHARMA: Distributed home agent for robust mobile access. In *Proc. of INFOCOM*, pages 1196–1206, 2005.

[35] T. Matsuda and T. Takine. (p,q)-epidemic routing for sparsely populated mobile ad hoc networks. *Selected Areas in Communications, IEEE Journal on*, 26(5):783 –793, jun. 2008.

[36] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[37] L. Mottola, G. Cugola, and G. Picco. A self-repairing tree overlay enabling content-based routing in mobile ad hoc networks. *IEEE Trans. on Mobile Computing*, 7(8):946–960, 2008.

[38] E. Nordstrom and H. Lundgren. Aodv-uu implementation from uppsala university.

[39] J. Ott. Application protocol design considerations for a mobile internet. In *Proc. of the 1st ACM/IEEE Int'l. Wkshp. on Mobility in the Evolving Internet Architecture*, pages 75–80, 2006.

[40] J. Ott, D. Kutscher, and C. Dwertmann. Integrating DTN and MANET routing. In *Proc. of CHANTS workshop*, pages 221–228, 2006.

[41] M. Papandrea. A smartphone-based energy efficient and intelligent multi-technology system for localization and movement prediction. In *PerCom Workshops*, pages 554–555, 2012.

[42] P. Pathirana, N. Bulusu, A. Savkin, and S. Jha. Node localization using mobile robots in delay-tolerant sensor networks. *IEEE Trans. on Mobile Computing*, 4(3):285–296, May–June 2005.

[43] A. Pentland, R. Fletcher, and A. Hasson. Daknet: Rethinking connectivity in developing nations. *IEEE Computer*, 37(1):78–83, January 2004.

[44] C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Proc. of WMCSA*, pages 90–100, 1999.

[45] A. Petz, C.-L. Fok, and C. Julien. Experiences using a miniature vehicular net-

work testbed. In *Proceedings of the ACM International Workshop on VehiculAr Inter-NETworking, Systems, and Applications*, 2012.

[46] A. Petz, A. Hennessy, B. Walker, C.-L. Fok, and C. Julien. An architecture for context-aware adaptation of routing in delay-tolerant networks,. In *Proceedings of the 4th Extreme Conference on Communication*, 2012.

[47] A. Petz and C. Julien. An adaptive architecture to support delay tolerant networking. In *Proceedings of the 7th Workshop on Adaptive and Reflective Middleware*, December 2008.

[48] A. Petz and C. Julien. The MaDMAN middleware for delay-tolerant networks. In *Poster at HotMobile 2010 (Proceedings of the 11th Workshop on Mobile Computing Systems and Applications*, 2010.

[49] A. Petz, T. Jun, N. Roy, C.-L. Fok, and C. Julien. Passive network-awareness for dynamic resource-constrained networks. In *Proc. of DAIS*, 2011.

[50] A. Petz, A. Lindgren, P. Hui, and C. Julien. MADServer: An architecture for opportunistic mobile advanced delivery. In *Proceedings of the ACM MobiCom Workshop on Challenged Networks*, 2012.

[51] A. Petz, B. Walker, C. Ardi, and C. Julien. The click convergence layer: Putting a modular router under DTN2. In *Proceedings of the 2nd Extreme Workshop on Communication*, September 2010.

[52] A. Petz, B. Walker, C.-L. Fok, C. Ardi, and C. Julien. Network coded routing in delay tolerant networks: An experience report. In *Proceedings of the 3rd Extreme Conference on Communication*, 2011.

[53] http://mpc.ece.utexas.edu/pharos/.

[54] http//proteus.ece.utexas.edu.

[55] R. Ramdhany and G. Coulson. Manetkit: A framework for manet routing protocols. pages 261 –266, jun. 2008.

[56] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In D. Milojicic, F. Douglis, and R. Wheeler, editors, *Mobility: Processes, Computers, and Agents*, pages 567–581, 1999.

[57] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *PFLDNet05*, 2005.

[58] K. Scott and S. Burleigh. Bundle protocol specification. *IETF Draft, draft-irtf-dtnrgbundle-spec-01. txt, October*, 2007.

[59] K. L. Scott and S. Burleigh. Bundle protocol specification. Technical Report RFC5050, IRTF, Nov. 2007.

[60] S. Sengupta, S. Rayanchu, and S. Banerjee. An analysis of wireless network coding for unicast sessions: The case for coding-aware routing. In *Proc. of INFOCOM*, pages 1028–1036, 2007.

[61] A. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. of Mobicom*, pages 155–166, 2000.

[62] T. Spyropoulos. Spray and focus: Efficient mobility-assisted routing for heterogeneous and correlated mobility. In *Proc. of ICMAN*, 2007.

[63] T. Spyropoulos, K. Psounis, and C. S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proc. of WDTN*, 2005.

[64] R. Srinivasan and C. Julien. Passive network awareness for adaptive mobile applications. In *Proceedings of the 3rd International Workshop on Managing Ubiquitous Communications and Services*, pages 22–31, 2006.

[65] J. Su, J. Scott, P. Hui, J. Crowcroft, E. De Lara, C. Diot, A. Goel, M. H. Lim, and E. Upton. Haggle: seamless networking for mobile applications. pages 391–408, 2007.

[66] S. Symington. Delay-Tolerant Networking Metadata Extension Block. RFC 6258 (Experimental), 2011.

[67] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, April 2000.

[68] B. N. Vellambi, R. Subramanian, F. Fekri, and M. Ammar. Reliable and efficient message delivery in delay tolerant networks using rateless codes. In *Proc. of MobiOpp*, 2007.

[69] Y. Wang, S. Jain, M. Martonosi, and K. Fall. Erasure-coding based routing for opportunistic networks. In *Proc. of WDTN*, pages 229–236, 2005.

[70] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proc. of Mobisys*, pages 179–192, 2009.

[71] J. Widmer and J.-Y. L. Boudec. Network coding for efficient communication in extreme networks. In *Proc. of WDTN*, 2005.

[72] E. Yoneki and J. Bacon. An adaptive approach to content-based subscription in mobile ad hoc networks. In *Proc. of MP2P*, 2004.

[73] X. Yu. Improving tcp performance over mobile ad hoc networks by exploiting cross-layer information awareness. In *Proceedings of the 10th annual international conference on Mobile computing and networking (MobiCom '04)*, pages 231–244, New York, NY, USA, 2004. ACM.

[74] X. Zhang, G. Neglia, J. Kurose, and D. Towsley. On the benefits of random linear coding for unicast applications in distruption tolerant networks. In *Proc. of WiOpt*, pages 1–7, 2006.