



An Adaptive Architecture to Support Delay Tolerant Networking

Agoston Petz
Christine Julien

TR-UTEDGE-2008-002



© Copyright 2008
The University of Texas at Austin



An Adaptive Architecture to Support Delay Tolerant Networking

Agoston Petz and Christine Julien

Department of Electrical and Computer Engineering

University of Texas at Austin

agoston.petz@gmail.com, c.julien@mail.utexas.edu

Abstract

Delay Tolerant Networks (DTNs) are emerging as a new form of networks in which sending and receiving nodes may not be reliably connected to each other in a traditional sense but must instead rely on mobile nodes to ferry messages through the network. Future mobile computing deployments are likely to require such DTN deployments to operate alongside more traditional mobile ad hoc networking approaches. Applications operating in these dynamic mixed 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. In this paper, we explore the systems issues related to enabling such a symbiotic network architecture. Specifically, we develop an adaptive architecture that enables connections to seamlessly migrate from one communication style to another in response to changing network or application conditions. We delineate the properties of the architecture, describe an initial implementation, and provide a simulation analysis that demonstrates that, given perfect context-awareness, significant performance improvements can be made by using such an adaptive architecture.

1 Introduction

Evolving mobile computing environments consist of unpredictable mobile entities connected via wireless networks whose topologies are subject to constant change. The use of these networks in both commercial and military deployments necessitates a new paradigm of communication and message exchange. *Delay tolerant networking* (DTN) has evolved in which protocols asynchronously route messages from a source to a destination over networks that lack continuous connectivity. In these networks, connections come and go dynamically, the source and destination may never be directly connected, and nodes' movements are unpredictable.

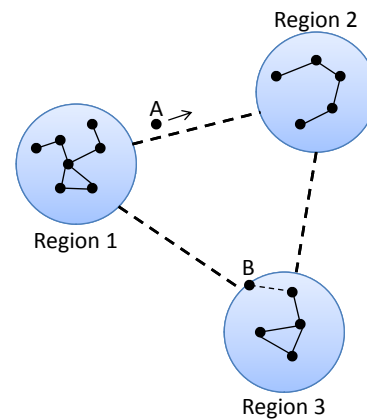


Figure 1. A generic DTN target environment

DTN architectures potentially apply in many application instances. In an urban setting, commuter transportation systems can be used as storage to carry messages from one region of a city to another [5]. A remote village may be well connected internally, but less reliably connected to the wider world [20]. 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 jeeps 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 [10]. Pockets of well-connected sensor networks can be periodically connected via roving mobile robots [19].

Figure 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 [8, 11, 25] and opti-

mizing topologies are well-studied, and great strides have been made with respect to supporting this new paradigm of communication. In this paper, we study a more practical systems issue, namely the ability of networked devices to adapt their network interactions to the changing operating environment. Specifically, we present a network architecture and implementation that allows one network stack to be dynamically swapped in for another while not experiencing (from the application level) a disruption in connectivity. With an aim to support networks such as the one depicted in Figure 1, we 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). We focus on enabling the mechanics of this transition and the systems and communication issues that must be resolved to enable seamless transitions. For now, we ignore issues related to what context information should be used to determine when to transition and the functions that relate the context to the transition decision and focus instead on demonstrating that, given good knowledge of the connection state, it is beneficial to make transitions in the first place.

The remainder of this paper is organized as follows. In the next section, we describe the targeted problem and provide background information. Section 3 describes the architecture for the entire system, while Section 4 discusses the implementation details of our prototype system. Section 5 provides some evaluation of our approach through emulation, demonstrating the potential benefits of swapping stacks in particular situations. Section 6 concludes.

2 Background and Problem Definition

In this section, we first detail a single example application that falls in the general category depicted in Figure 1. We then look at related approaches and use them as a springboard to precisely define the problem we address in this paper.

2.1 A Motivating Scenario

As touched upon in the previous section, delay tolerant networks are relevant to a wide variety of applications. To further explore the requirements of applications for delay tolerant networks, we hone in on a particular example that fits the model shown in Figure 1, specifically using DTNs to connect remote villages to each other and to the Internet through distant hubs [3, 20]. In such an application, devices are deployed within small remote villages (e.g., via the one laptop per child project [16]) where the available infrastructure is severely limited. Within the village, the devices have good connectivity among themselves, supported by traditional mobile ad hoc networking protocols or

other wireless grid approaches. Connectivity to the wider world, however, is more difficult; without infrastructure networks, long-range wireless communication is too energy intensive for the villages' constrained resources. However, people transit between villages and larger cities to deliver goods and supplies and to visit family and friends. Some of this movement is on a predictable schedule; other movements are more spontaneous. By leveraging the storage and communication capabilities of devices traveling with people among villages, computer users in remote areas can achieve access to the wider networked world.

This network access, however, is not of the same nature as the Internet access most people enjoy. Connections are asynchronous, demanding an entirely new network architecture to support these devices. Delay tolerant networks have emerged as the obvious candidate; in such networks, applications that can tolerate significant latencies can be supported even under the duress of the conditions described above. However, much work remains to be done to provide applications the same kinds of abstractions they are used to in traditional networks given this vastly different underlay.

2.2 Related Approaches

Given the application described previously, an overarching goal of our work is to provide applications the semblance of a communication session in a delay tolerant network, even when no end-to-end connection is even possible. In addition, we aim to maximize application performance by dynamically selecting the best end-to-end approach based on applications' (potentially competing) requirements, the communicating parties, and the network conditions. Given these goals, 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 in Dynamic Environments. The nature of existing applications commonly stresses a view of 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, for example by predicting when link failures will occur in mobile networks and mitigating their impact [1] or by using dynamic DNS updates to support TCP connection migration [24]. 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 [14, 7]. 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 suite emerging applications, architectures that maintain interfaces for existing applications but expose new functionality have been developed. The Overlay Convergence Architecture for Legacy Applications (OCALA) [9] 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 functions below the transport layer, presenting legacy applications with a standard IP-like interface and hiding a sub-layer that translates between these traditional network interactions and interactions in new network overlays. By design, OCALA enables applications utilizing different overlays to interact with each other but must use well-defined gateways to transition between overlay types. 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 Internetnetworked Services (Oasis) [12] also attempts to enable legacy applications to continue to function on top of overlays that offer more expressive interfaces to newly introduced applications. Oasis inserts a layer that manages available overlays for applications, allowing the “best” performing overlay to be dynamically selected at the beginning of a connection. 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.

An approach specific to delay tolerant networks effectively defines an API that allows an application to choose a “traditional” end-to-end connection (provided by the AODV MANET routing protocol [21]) or a DTN based communication protocol based on information about the availability of the end-to-end connectivity [18]. The underlying assumption is that, if possible, end-to-end semantics are always preferable. This approach exposes network information (e.g., information about routing paths and or responsible intermediaries) to the application to use in subsequent decision making. Because the approach intimately intertwines AODV and DTN routing, the DTN routing algorithm can take advantage of AODV knowledge in its forwarding decisions. However, this approach makes the two communication styles dependent on each other, which has the potential to significantly increase the overhead in comparison to the individual protocols.

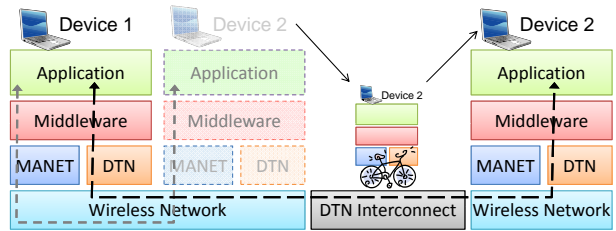


Figure 2. Operation of a dynamic delay tolerant network architecture.

2.3 Problem Definition

Design considerations relating to the impact of a delay tolerant network underlays on end-to-end application semantics have begun to be explored [17]. We move a step further by looking at a specific technical challenge within these new requirements; specifically, we enable the underlying network implementation to vacillate intelligently 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 requirements. 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.

In realizing the above goal, we make a few assumptions. First, we delegate destination discovery to the underlying network protocols (be they traditional MANET routing protocols or DTN routing protocols). We instead focus on the systems issues involved in being able to dynamically and robustly switch an ongoing connection from one style of communication to another. We also assume that the ideal “DTN-aware” application can specify, as an application parameter, a maximum tolerable delay. This information can be used to determine whether the underlying systems should even attempt to satisfy the request, given the current network conditions. In the remainder of this paper, we will speak mostly in terms of a single “application” which comprises two end points and the (potentially dynamic) connection between them. Aspects of the ideal resulting architecture are depicted and described in Figure 2. 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 automati-

cally 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).

3 An Architecture for Seamless Delay Tolerant Networking

The architecture we propose 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 model 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. Our middleware is architected to allow new and experimental physical and MAC layer protocols, network layer protocols, and transport and application layer protocols to easily plug into the middleware framework. It is designed to present a single, uniform communication interface to applications while at the same time allowing researchers to easily modify the internal components. In this way, it is not dissimilar to OCALA [9]. What makes our middleware 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. The middleware’s model incorporates the use of application and network context to automatically determine the appropriate time to switch stacks on a per-connection basis.

In the remainder of this section, we describe the middleware’s components at a conceptual level. More detailed discussions of our particular implementation are handled in Section 4. The middleware has four distinct parts: (1) a service daemon 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 subsystems, and Figure 3 illustrates their conceptual relationships.

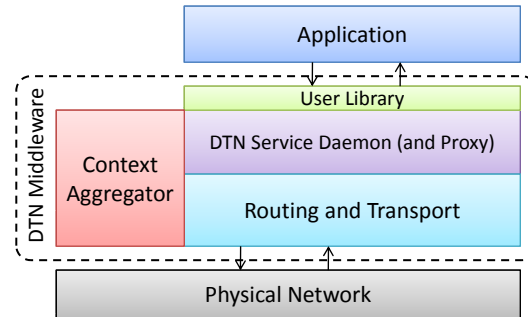


Figure 3. DTN Middleware Components

3.1 Service Daemon

The service daemon implements most of the functionality of the middleware, and should be viewed as the “core” of the system. It is the most complex component and the component that communicates with all of the other pieces of the system. It is responsible for interfacing with applications, managing dynamic resources, and negotiating out-bound and inbound connections. The service daemon is intended, but not required, to run on all nodes participating in the delay tolerant network.

Our middleware architecture specifies that the service daemon should behave like a proxy and communicate with applications using standard libraries (e.g., sockets or some other network capable IPC). We maintain that a standardized approach to IPC is a good idea for two reasons: (1) it allows user applications written in any language to easily interface with the middleware so long as they have an appropriate user library, and (2) it allows the middleware to exist on a different machine than the application. The second reason is important since the ultimate architecture of a network relying on our middleware may vary from a completely distributed deployment to one that includes delay tolerant “gateways” [4, 6]. In the latter case, our middleware could be hosted on a gateway machine and could service a number of clients on the local network. An unfortunate drawback of our proxy-style architecture is that it breaks end-to-end semantics since application data is acknowledged by the middleware before it is actually delivered to the destination. This is also an advantage of the middleware, since it can keep trying to deliver data long after the sender has closed the socket and given up on waiting for the acknowledgement. A future implementation of the middleware’s service daemon could incorporate a mechanism similar to that explored in TCP Splice [13] that provides a lock-step implementation that preserves the full end-to-end semantics, should such a strong semantics be required.

3.2 User Library

The user library provides the API by which applications interact with the delay tolerant network. The user library is intended to be as small (and portable) as possible and functions like a wrapper for the sockets library. Our user library only implements the necessary functionality to translate socket requests to service daemon requests. It does this by first packing the application socket request parameters into a separate request destined for the service daemon, sending that request to the daemon, and returning the result to the application.

The user library also allows the application to provide a small amount of context to the middleware. Currently, applications can provide 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. Priority hints are helpful since a low priority connection can be selectively scheduled on the delay tolerant stack to free up TCP/IP resources. We are also considering feedback-hints about future bandwidth requirements, application suggestions for re-scheduling transmissions, etc.

In the future, the middleware will also act as a SOCKS proxy so that legacy applications will be able to entirely bypass the user library. This will obviously prevent the sharing of application context and hints since that functionality is captured by the user library API but it allows existing applications to coexist with new DTN-aware applications.

3.3 Routing and Transport Layer

The routing component implements the algorithms and protocols inherent to the routing layer, and it is where various existing and future protocols can be plugged in. As described previously, much existing work focuses on creating good DTN routing protocols, and this component approach allows the middleware to selectively take advantage of this work. In an effort to make this easier, the middleware places “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 [15] and many other routing protocol implementations interact with the kernel. We have also intentionally separated routing from transport (see the more detailed Figure 4) to enable the incorporation of end-to-end semantics specific to the new DTN model of communication. We have created a very simple first cut at this transport protocol, which we refer to as the User Datagram Protocol for Delay Tolerant Networks (UDPDTN). An important aspect of the use of a transport protocol in support of a DTN connection within our middleware is the ability to transfer state between the DTN network stack and the traditional network stack as applications’ connections transition

between the two. We briefly describe this transport protocol next.

3.4 User Datagram Protocol for Delay Tolerant Networks

UDPDTN is a stateful but connectionless protocol and is intended to simplify the seamless transition between the traditional network stack and a DTN-aware network stack. 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, the middleware 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. Our initial cut at the UDPDTN protocol is simple and uses little of this information; future DTN transport protocols may rely more heavily on it. In the future, we also hope to run the DTNRG bundle overlay protocol [23] over our enhanced UDPDTN “underlay”, and the added DTN-specific information in the transport layer will benefit the bundle protocol and other DTN overlay protocols.

3.5 Context Aggregator

The context aggregator shown in Figure 3 is responsible for monitoring and processing network context to accurately characterize the state of the network. This information will be used to determine the optimal stack to use for a given connection at a given time and will depend on some combination of network layer statistics, transport layer statistics, and the current application activity. Examples include connectivity, throughput, and latency, although defining this context and providing an algorithm to determine the “appropriateness” of using either stack is beyond the scope of this paper.

4 Implementation

We have implemented our middleware in C++ on the Linux 2.6 kernel. The following subsections provide specific implementation details for the various components and Section 4.5 provides a step by step example of how an application uses the middleware to set up a connection.

4.1 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

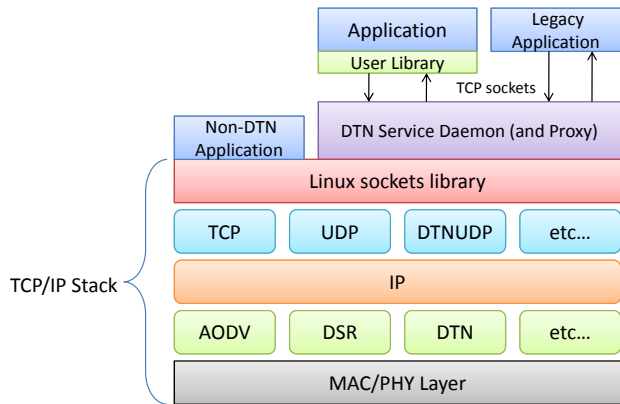


Figure 4. Architecture Overview

middleware. The service daemon pulls them all together to allow them to be managed in a unified space. Specific functions of these threads will be described in Section 4.5 as part of the example. As depicted in Figure 3, 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.

4.2 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 the middleware; future work will add a SOCKS proxy interface to the middleware for legacy application support.

4.3 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 the middleware. As mentioned previously, we implement a generic router using Linux’s `libipq` library to queue incoming messages in user-space where the router component of the middleware 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 “UDPDN.”

4.4 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 a god-like view of when transitions are best made.

4.5 Example connection setup and tear-down

The DTN middleware has two modes of operation, one for handling outbound connections to remote DTN-enabled hosts (shown in Figure 5), and one for handling inbound connections (shown in Figure 6). Applications and middleware components are shown as rectangles, sockets are shown as circles, and arrows show directional data flow between the sockets and components. To explain how the components of the middleware 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 5) is then used to communicate between the application and the middleware.
3. The middleware’s *request processor* determines that the request is for an outbound connection, 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 middleware resources, connecting to the remote address, and determining priority information.

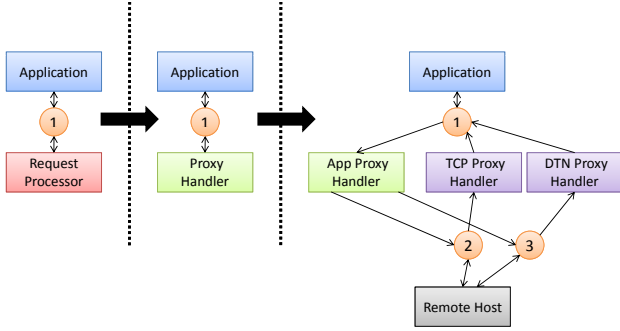


Figure 5. Outbound (Proxied) Request

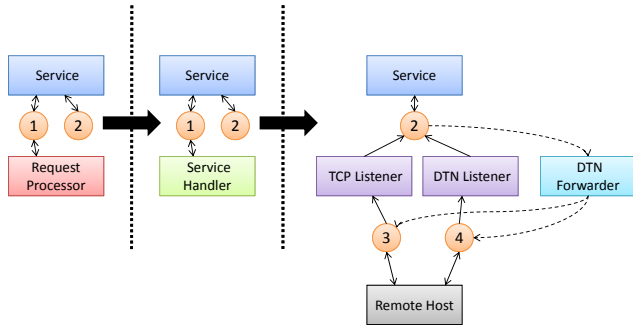


Figure 6. Service Request

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 5. 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. Future work will explore a using a worker-thread model instead of a thread per socket to increase connection efficiency.

Connection setup for listening services (Figure 6) happens in a similar fashion, except that for each incoming connection (regardless of the type) the middleware opens a return connection to the service application.

5 Evaluation

To evaluate our middleware's performance, but more importantly to discover whether it is beneficial to dynami-

cally 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.

5.1 Network Emulator

Our network emulator is similar to NIST Net [2] and allows us to introduce arbitrary packet delays and packet drops into the network. As in NIST Net, our emulator is a user-space program that interfaces with the kernel using the `libipq` library. Incoming and outgoing packets are first passed to the emulator, which assigns a delay to the packet and schedules the packet for later transmission. This allows us to capture and control the behavior of arbitrary DTN deployments in a single piece of software. Our implementation is independent of any particular DTN routing algorithm or particular network topology. Instead, we apply a delay to every packet using on a probabilistic distribution based on a given mean. These distributions are given as inputs to the emulator and can be thought of as delay vs. time curves which capture the properties of the delay tolerant network from the perspective of an end-to-end connection.

5.1.1 Packet Scheduling

Each packet is assigned a delay when it enters the network, and the value of the delay is calculated from a delay vs. time curve at a 1/4 second granularity. For example, if a packet enters the system at time $t = 14$ seconds, and the delay at 14 seconds is given by the delay vs. time curve as a normally distributed random number x with mean 10,000 ms and a variance of 500 ms, the packet will be queued and delivered at $t + x$ seconds. All packets that enter the network at the same time are assigned a delay using the same distribution. At quarter-second intervals, the delay distribution changes to a different mean and variance, allowing us to control the network characteristics at a relatively fine level of granularity. In this scheme, out of order delivery is entirely possible since adjacent delays are completely independent. During the course of our simulations, we tried many different delay vs. time curves, and the motivation behind the curves we selected is provided below.

5.1.2 Packet Loss

To model packet drops, we incorporated a probabilistic packet drop scheme in the emulator. 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

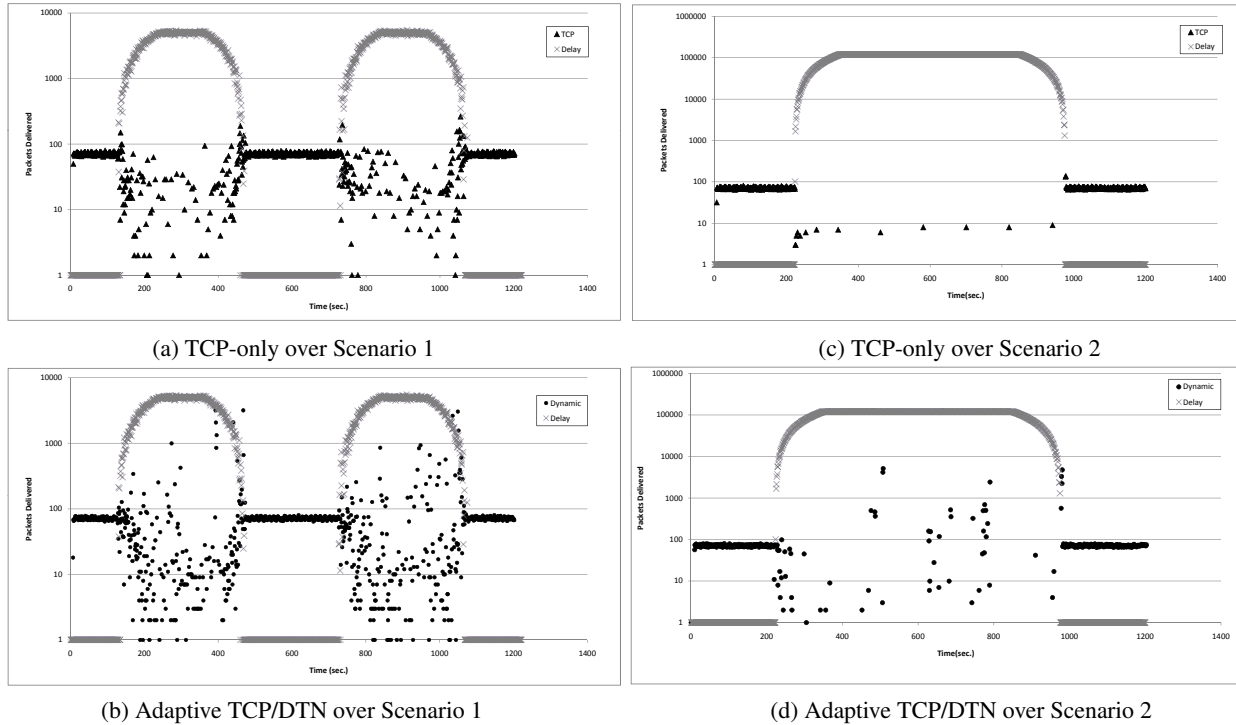


Figure 7. Throughput vs. Simulation Time for Scenarios 1 and 2. Each graph shows the delay (in gray) imposed on the packets and the measured throughput (in black) of either the TCP-only approach ((a) and (b)) or the mixed TCP/DTN approach ((c) and (d)).

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 performance in terms of throughput, as demonstrated below.

5.1.3 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 7(a) and (b).

5.1.4 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 7(c) and (d).

5.2 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 during the connection. In our tests, we make the assumption that both TCP and UDP can take advantage of the delay tolerant routing protocol and can thus 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. Since UDP does not have flow control, we had to limit the data rate to avoid overflow behavior. 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.

5.2.1 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 [22]). The results of the tests are shown in Figure 7(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 7(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 with a data loss of around 950 packets. This is an 80% increase in throughput with 0.01% packet loss. 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.

5.2.2 Scenario 2—Long Delay

The results for the second scenario are shown in Figures 7(c) and 7(d). 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.

5.2.3 Summary

The adaptive TCP/DTN stack swapping greatly increases throughput in both of the above scenarios, and it is easy to see that in scenarios with longer delays, and more time spent operating over DTNs, the benefits of stack swapping can only increase.

6 Conclusions

Future mobile computing network deployments will demand the ability to dynamically migrate application connections from one communications technology to another. Specifically, as Delay Tolerant Networks (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 boost performance. In this paper, we adopted a systems perspective on enabling applications’ connections to be seamlessly moved from one communication style to another. By building this architecture in a real operating system and emulating its performance over a real network, we demonstrated that it is not only reasonable, but potentially highly beneficial to employ such an adaptive network architecture. In comparison to a reliable end-to-end protocol, a protocol stack that adapts can take advantage of the reliability when feasible but achieve better throughput in periods of long delay by temporarily switching to a best-effort style protocol. When the network improves, the protocol stack can switch again, reaping the benefits of both styles of communication.

This work has focused on developing a modular software architecture for enabling adaptation between DTN-aware protocols and more traditional communication styles and demonstrating that such an adaptive architecture can be beneficial to applications. We have assumed an omniscient perspective with respect to the points at which the transition between protocols should occur. Ongoing work is exploring how less perfect information about the network conditions can be incorporated to maintain the benefit of an adaptive network stack. In addition, we are incorporating real, representative DTN transport and routing protocols into the architecture. We expect the performance of the architecture to improve even further, as the performance of these tailored protocols should exceed the naïve UDP-like implementation we have used. In total, the architecture we have presented in this paper is a fundamental first step in smoothly integrating DTN technologies and benefits into our existing mobile computing infrastructure.

Acknowledgements

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded in part by the DoD. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

- [1] A. Baig, M. Hassan, and L. Libman. Prediction-based recovery from link outages in on-board mobile communication networks. In *Proc. of IEEE Globecom 2004*, pages 1575–1579, 2004.
- [2] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.
- [3] A. Doria, M. Uden, and D. Pandey. Providing connectivity to the Saami nomadic community. In *Proc. of the 2nd Int'l. Conf. on Open Collaborative Design for Sustainable Development*, 2002.
- [4] K. Fall. A delay-tolerant network architecture for challenged internets. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, 2003.
- [5] 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.
- [6] T. Hyyryläinen, T. Kärkkäinen, C. Luo, V. Jaspertas, J. Karvo, and J. Ott. Opportunistic email distribution and access in challenged heterogeneous environments. *Proceedings of the second workshop on Challenged networks CHANTS*, pages 97–100, 2007.
- [7] O. J and D. Kutscher. A disconnection-tolerant transport for drive-thru internet environments. In *Proc. of the 24th Annual Joint Conf. of the IEEE Computer and Communications Societies*, pages 1849–1862, 2005.
- [8] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. *ACM SIGCOMM Computer Communication Review*, 34(4):145–158, October 2004.
- [9] 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 the 3rd Symp. on Networked Systems Design and Implementation*, pages 267–280, 2006.
- [10] 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 zebra-net. *ACM SIGPLAN Notices*, 37(10):96–107, October 2002.
- [11] 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.
- [12] 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.
- [13] D. Maltz and P. Bhagwat. MSOCKS: an architecture for transport layer mobility. *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3.
- [14] Y. Mao, B. Knutsson, H. Lu, and J. Smith. DHARMA: Distributed home agent for robust mobile access. In *Proc. of the 24th Annual Joint Conf. of the IEEE Computer and Communications Societies*, pages 1196–1206, 2005.
- [15] E. Nordstrom and H. Lundgren. AODV-UU Implementation from Uppsala University.
- [16] One laptop per child. <http://laptop.org>, 2007.
- [17] 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.
- [18] J. Ott, D. Kutscher, and C. Dwertmann. Integrating DTN and MANET routing. In *Proc. of the 2006 SIGCOMM Wkshp. on Challenged Networks*, pages 221–228, 2006.
- [19] 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.
- [20] A. Pentland, R. Fletcher, and A. Hasson. DakNet: Rethinking connectivity in developing nations. *IEEE Computer*, 37(1):78–83, January 2004.
- [21] C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Proc. of the 2nd IEEE Wkshp. on Mobile Computer Systems and Applications*, pages 90–100, 1999.
- [22] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *PFLDNet05*, 2005.
- [23] K. Scott and S. Burleigh. Bundle Protocol Specification. *IETF Draft, draft-irtf-dtnrgbundle-spec-01.txt*, October, 2003.
- [24] A. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. of the 6th Annual Int'l. Conf. on Mobile Computing and Networking*, pages 155–166, 2000.
- [25] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, April 2000.