Probabilistic Routing in Delay Tolerant Networks

Jeff Wilson
November 16, 2007

Jump to: navigation, search

Opportunities for digital communication are often limited due to impoverished resources or the expense of using established infrastructure. One possible solution involves the use of message relay over ad hoc, peer-to-peer communication channels. Wireless-enabled, mobile devices are quickly becoming commonplace, increasing the potential for interpersonal networking apart from infrastructure. In areas lacking infrastructure, mobile communication technologies can be affixed to vehicles and personnel. Such networks may never have a contemporaneous path between source and destination, instead making use of communication opportunities to form such paths asynchronously, one hop at a time. To minimize delay and enhance chances of delivery, routing techniques rely on mobility and message relay to move a message "closer" to its destination, with limited or no knowledge of global topology. Traditional network-layer protocols fail to accommodate such extreme conditions. Research on intermittently connected networks accommodates long periods of isolation, but these solutions provide little support for dynamic interpersonal networking. We contribute dynamic discovery and routing to one such project as well as support for native Bluetooth, a widely available commodity wireless technology.

Contents

1 Introduction

Data networks continue to extend their reach over copper, fiber, and wireless infrastructure. Perhaps more rapidly, personal computing devices (e.g., PDAs, cell phones, etc.) continue to improve and to expand their feature set, to the point that recent models can be expected to have some form of personal wireless networking, such as WiFi or Bluetooth. We carry these devices with us wherever we go, discovering in our mobility that fixed networks do not yet offer easy access. Some regions without infrastructure suffer from impoverished resources; other regions suffer excessive rates imposed on existing infrastructure, effectively rendering the network inaccessible. Irrespective, some kind of novel peer-to-peer opportunistic networking is possible using these wireless-enabled devices.

1.1 Challenged Networks

"Challenged networks" form a broad category of work, loosely described as any attempt at data communication apart from fixed infrastructure. As compared to the operation of the Internet Protocol, many key assumptions do not hold when applied to challenged networks: an end-to-end path may never exist between communicating pairs; round trip delay may be arbitrarily long (hours, days, weeks); some messages may never successfully reach their destination; etc. [1] To further contrast with IP networks, the links that make up challenged networks can expect to face any combination of high latency, low bandwidth, high error rate, shorter node longevity, and reduced path stability. The bleak reality in challenged networks is that disconnection is often much more common than connection.

In such challenged networks, fixed networks are inaccessible, or possibly available via some bridged node at the edge of the challenged region. Connection-rich regions are well served by MANETs' mesh network protocols[2], but these fail to compensate for the greater challenges faced by connection impoverished regions. In these more isolated regions, links form when peers opportunistically encounter each other after some potentially lengthy delay. Due to the short range of whatever personal wireless technology is in use, mobility is essential for overcoming the challenge of isolation. Delay between encounters are likely to be common, and may last indefinitely. Communicating endpoints may never directly encounter each other. Even if a mesh protocol serves to interconnect all nodes within range, it is unlikely that an end-to-end path would ever be discovered between sender and receiver. For traditional (IP) routing protocols, such non-existent paths cause packets to be dropped; clearly IP cannot function in such environments.

1.2 Criteria

In contrast to IP's failure to communicate in connection impoverished challenged networks, a working solution could queue messages to persistent storage through the duration of solitude. As mentioned earlier, mobility is necessary to overcome the challenge of isolation; in addition, relay is also essential to enhance the probability of message delivery and to minimize delay. A message is relayed when the sender (or some intermediary) entrusts a copy of the message to some node other than the destination (an intermediary) for the express purpose of delivering to the destination or relaying to some other intermediary "closer" to the destination. When a node relays the same message to more than one peer, the message is replicated through the network; the higher the degree of replication, the greater the chance that a given message will be delivered. However, unbounded replication will consume significant storage and bandwidth resources.

Some delay is to be expected when routing messages across connection impoverished challenged networks. In fact, no guarantee can be made that a given message can be delivered to the intended destination before the message TTL expires. A routing solution for such networks must take advantage of a short-range personal wireless technology for transport. Regardless of underlying link technology, the solution should have an automated means of discovering when peers come into proximity. The routing strategy should seek to balance and optimize minimal delay against maximal delivery probability. At the same time, an optimal queuing policy should honor the storage constraints faced by participating nodes, so that message evictions balance storage quota against message delivery probability.

Fall's proposed DTN architecture[1] has the potential of fulfilling these criteria. Designed as a peer-to-peer overlay architecture, DTN provides a store-and-forward messaging service across challenged networks. The basic data unit is known as a "bundle." By operating above network transport protocols in the application layer, DTN can function as a gateway between dissimilar networks, bringing together various naming semantics into a unified bundle addressing scheme. An end-to-end path need not exist between sender and receiver; instead, routes form across a cascade of time-dependent communication opportunities. Bundles are queued to persistent storage through arbitrary delay; whenever a peer is encountered, some routing protocol determines which bundles to forward. To increase chances of delivery, bundle TTL can be set for an arbitrary length of time. DTN abstracts the differences between transports used across a variety of challenged networks by proposing a transport-specific convergence layer. To adapt the bundling service to an arbitrary network transport, the convergence layer provides mechanisms to assure (or subscribes to services in the transport layer that provide) reliable delivery, connection management, flow control, congestion control, and message boundaries. By shrinking the time dependency from the full length of the path to just one hop at a time, DTN opens up possibilities for networking across many different kinds of challenged areas.

The IRTF's DTN research group[3] (DTNRG) makes available the Delay Tolerant Networking Reference Implementation[4] (DTNRI). However, the DTNRI falls short in two significant criteria: no transport is available across wireless networks, and no automated peer discovery is available. Without dynamic peer discovery, no dynamic routing protocols can be implemented.

Simply deploying DTN with its current capabilities is not sufficient. One of our criteria states that a solution will leverage a personal wireless technology for transport. To satisfy that criterion, we select from today's marketplace a technology that is widely available and well supported: Bluetooth[5]. In addition to the advantage of standardized hardware that is widely deployed, Bluetooth also enjoys wide adoption in the Linux development community, with built-in support shipping in most modern distributions[6]. Also helpful to our solution, Bluetooth has a neighbor discovery protocol built in to its hardware specification, called Inquiry.

1.3 Proposed Solution

We propose a solution to the problems faced by connection impoverished challenged networks made up of stationary nodes mixed with undirected mobility. Our solution satisfies the following requirements: queue messages through arbitrary delay; utilize wireless links when available by dynamically discovering neighbors as they come into range; increase delivery probability and reduce delay using replication; and honor storage constraints imposed by participating nodes. At present, no such solution exists.

While finite battery life is a concern, we ignore its limitations by assuming that node longevity will outlast message lifetime. This assumption leads us to further assume that any link disconnection is caused by mobility, not battery failure. We assume that, due to the challenges faced by connection impoverished regions, message lifetime may expire before successful delivery. We assume that any solution to such challenges will experience greatly reduced performance when compared to fixed networks: greater delay, greater possibility of delivery failure, and perhaps higher overhead per payload. However, we further assume that, when compared to the prospect of no communication at all, reduced performance is acceptable to network participants. We only consider the case of cooperative relay, ignoring the case where relay is refused. We

assume that storage is limited, which affects our choice of forwarding strategy and queuing policy. And finally, we limit our focus to undirected mobility and optimize our delivery strategy accordingly.

We contribute this solution in the form of extensions made to the DTNRI. Specifically, we extend the DTNRI to

- include a new Convergence Layer to add Bluetooth support
- contribute a neighbor discovery mechanism (prerequisite to any dynamic routing protocol) that is generally applicable to arbitrary link technologies
- include an implementation of the PRoPHET dynamic routing protocol

In Chapter 2, we present a survey of the literature. Chapter 3 describes our solution in detail. The final two chapters offer a User's Manual and a Developer's Guide.

2 Related Work

Recent research in challenged networks proposes several novel approaches. The first part of this chapter introduces the reader to a brief taxonomy, with examples from a survey of related works, such as lessons learned from optimizations in protocols designed for automated nodes, as well as protocols more adapted for undirected mobility. Next we explore existing work in the area of dynamic routing and discovery, techniques we need for our solution. Finally, we close the chapter with a brief overview of available personal wireless technologies.

2.1 Types of Challenged Networks

In the previous chapter, we define "challenged networks" as any extension of data communication beyond fixed infrastructure. Challenged networks can be classified by the nature of the links interconnecting the network participants. When links between nodes form a connected graph, such a network is connection rich. By contrast, when nodes spend the majority of time in isolation, they form a connection impoverished challenged network.

2.1.1 Connection Rich

In "connection rich" challenged networks, we are given an abundance of in-range peers from which to construct an interconnected mesh. Note that this does not imply contemporaneous connectivity between ultimate source and destination. While participants may be mobile or stationary, any solution for a mobile environment should easily reduce to apply to the more simple case of stationary nodes. Solutions for mobile populations in these environments typically make the following assumptions: 1) a dynamic population forms a volatile connected graph; 2) the destination lies within this graph, or 3) the destination is reachable through some interconnection at the edge of the dynamic region. The strategy for connection rich challenged networks is to quickly form a mesh topology, optimizing route discovery in order to reduce startup latency.

Stable solutions exist for mobile ad hoc wireless networks (MANETs)[2], which form a broad category within connection rich challenged networks. Within this category, routing protocols focus on quickly discovering and converging to a mesh topology. Once a route is computed across the dynamic population of mobile wireless relays, traditional IP transports data from sender to receiver. MANET protocols are either proactive or reactive. Proactive protocols reduce startup latency at the expense of a constant chatter of polling for network topology changes. In contrast, reactive protocols (such as AoDV[7]) only poll the network topology "on demand", trading the constant overhead of polling for the occasional startup latency.

MANET solutions do not scale to meet the concerns of connection impoverished challenged networks. When topology is trivial, such as the meeting of two nodes, MANETs topology discovery needlessly creates a significant overhead. A different strategy is needed for such simple encounters. MANETs have the advantage of leveraging the robustness and maturity of Internet Protocol and its Sockets API. IP and Sockets are widely adopted and well understood. However, in environments where no path exists between sender and receiver, such as the challenges faced by connection impoverished networks, IP routing fails; messages are dropped, not queued.

2.1.2 Connection Impoverished

Participants in connection impoverished networks spend the majority of their time without any in-range peers, and connectivity only involves a small number (typically one) of simultaneous peers. Participants in connection impoverished networks are likely to face other challenges, such as storage limitations or finite battery life. In networks with stationary nodes, clearly some degree of mobility is necessary to accomplish communication across isolated regions. Even with mobility, however, performance will be poor when compared to fixed networks, in that messages will be indefinitely delayed, possibly never successfully delivered (prior to TTL expiration).

In connection-impoverished networks, mobility can be directed, as in autonomous nodes (such as robotics) or scheduled routes (such as public transportation). Other types of mobility are undirected, such as mobile phones and PDAs, or sensors used in tracking wildlife. Strategies differ between directed and undirected mobility. For example, directed mobility has the advantage of optimizing movement patterns to maximize delivery rate.

2.1.2.1 DTN

Delay tolerant networking[8] is an emerging field of study that focuses on the problems faced by connection-impoverished, challenged networks. As proposed by Fall[1], DTN is a general purpose, message-oriented reliable overlay architecture that seeks to bring together and bridge across heterogeneous link technologies. The design focus is on interoperability and robustness. By subscribing to various protocol stacks from the application layer, DTN provides a store-and-forward gateway function between dissimilar networks. For example, over the Internet, the overlay uses TCP for transport and DNS for name resolution, but when faced with mobility, the transport of choice is Bluetooth using its associated protocols. Interoperability between dissimilar networks is made possible by DTN gateways bridging them together.

DTN is based on an abstraction of message switching. Messages switched across this overlay network are called Bundles. Similar in operation to a postal service, DTN allows for asynchronous Bundle delivery: no end-to-end path may ever connect sender to receiver. Instead, Bundles are forwarded along a cascade of communication opportunities, so that the path forms over time. Bundles are spooled to persistent storage until such a forwarding opportunity arises. Also similar to postal service, DTN offers a coarse-grained class of service, such as delivery priority of low, normal, and high. Return receipt and custody transfer offer some assurance of reliability. To minimize the delay incurred with each round trip, Bundles should contain all necessary information to request or fulfill a given service. (Contrast this to the round trips needed for a typical web request: name lookup, TCP handshake, GET, etc.) Custody transfer allows DTN to renegotiate the contract of fate-sharing from the terms of fixed networks' end-to-end principle to challenged networks' hop-by-hop approach. When a sending node requests custody transfer from a peer, the fate of delivery is delegated to the peer for the Bundle in question, essentially shrinking the path by one hop. (This option is especially useful when forwarding a Bundle from a storage impoverished to a storage rich node, so that the sender may reclaim storage after transferring custody.)

The transport channels between DTN peers are called Links. Links are classified as Always-On, On-Demand, and Opportunistic, reflecting the predictability of each. Link state reflects whether a communication opportunity is currently available. Links between DTN nodes can be expected to face any combination of challenges, such as low bandwidth, high error rate, or high latency. Since protocols and services will vary widely across the technologies deployed in challenged networks, DTN adopts the strategy of normalizing each network to a Convergence Layer adapter (see Figure 1). By subscribing to network services at the application layer, DTN fills in with the Convergence Layer whatever is lacking in the network layer: reliable delivery, connections (and the related indications of connection failure), flow control, congestion control, and message boundaries. For TCP, where connections are managed by the protocol, the Convergence Layer need only restart connections if they are lost; in addition, the TCP Convergence Layer must manage message boundaries in light of TCP's stream semantics. Routers form the decision plane of DTN, directing which Bundles are forwarded across which Links according to some protocol. Bundle and Link dynamics drive the routing protocol, whether the protocol is statically configured or adaptively learns.

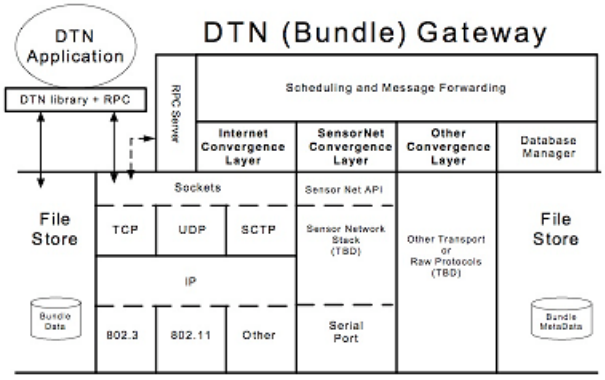
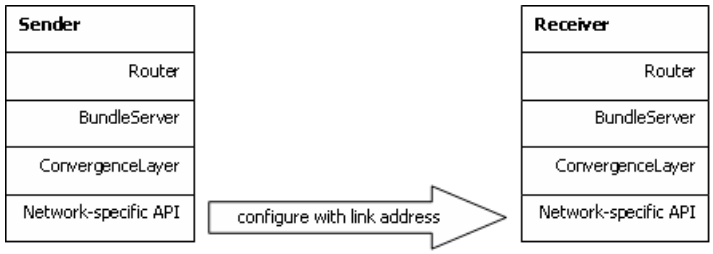
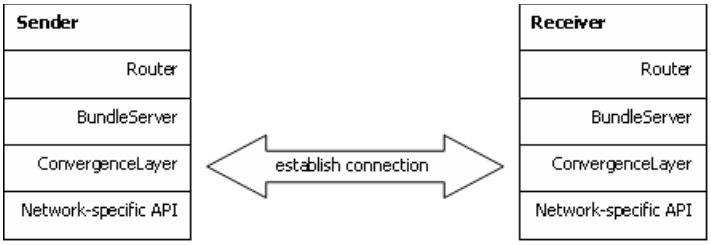


Figure 1. DTN Architecture[1].

A Bundle is transferred between DTN peers using the following procedure (see Figure 2). First, a Link must be configured using the network-specific address of the peer. Next, the Link must be opened by issuing an open command. To open a Link, the Convergence Layer performs the necessary operations on the specific network technology to prepare reliable transport of Bundles to the peer. (For example, the TCP Convergence Layer would issue a connection to the peer's IP address and TCP port.) Once the Convergence Layer completes the Link initialization (including any on-the-wire application-layer protocol for establishing a peering session between endpoints), the Convergence Layer signals that the Link is open. When the Router receives notice that the Link is open, it will queue any Bundles destined for or routed through the peer. Then each Bundle is sent over the Link. The Router issues the send command on the Link, causing the Bundle to be queued up in the Convergence Layer. The Convergence Layer serializes the Bundle over its underlying transport in a reliable fashion. Once the peer acknowledges receipt of the full Bundle, the Convergence Layer signals that the Bundle has been transmitted. The bundling system updates its forwarding statistics, and the next Bundle is queued up for transmission.

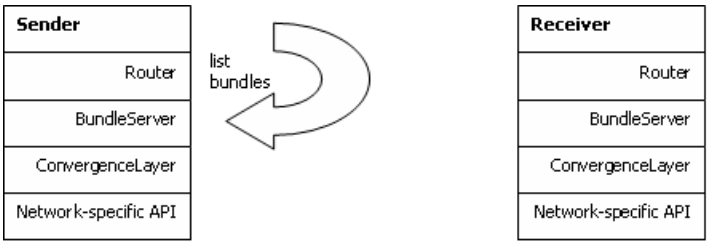


1. Configure sender with receiver's network specific address

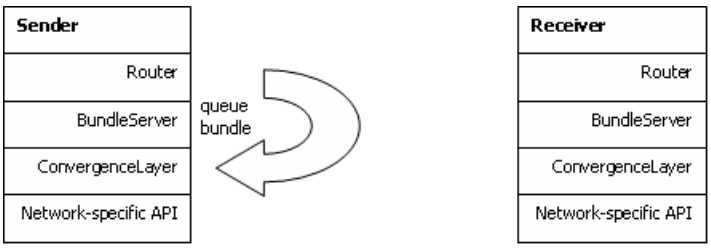


2. Establish application-layer connection between convergence layers

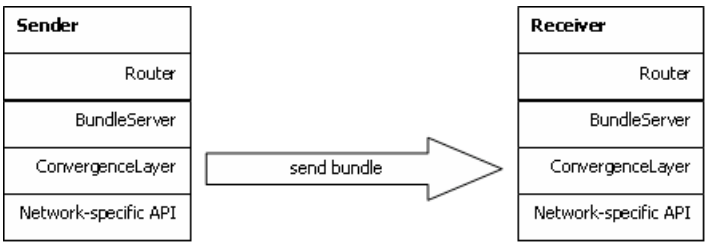
---



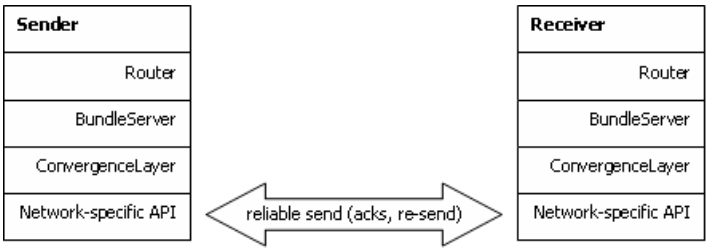
3. Router enumerates bundles in local store



4. Router enqueues outbound bundle with convergence layer



5. Convergence layer serializes bundle to network

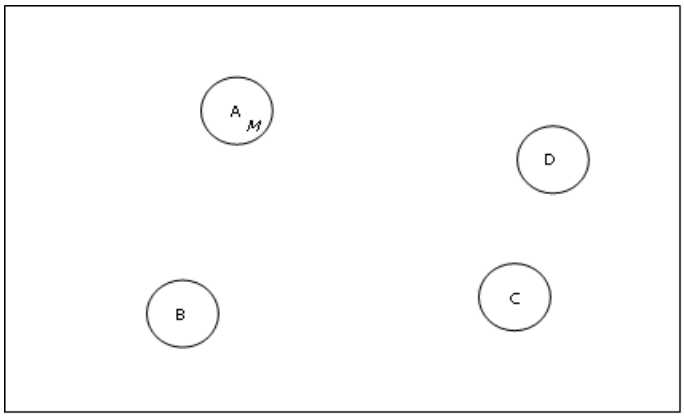


6. Acknowledge and retransmit as necessary to effect reliable transport

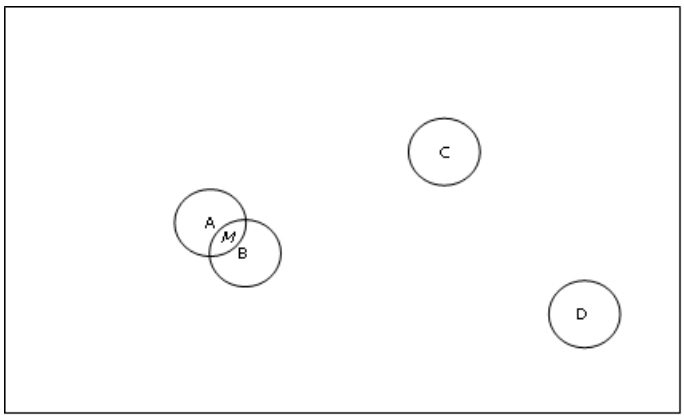
---

Figure 2. Bundle transfer.

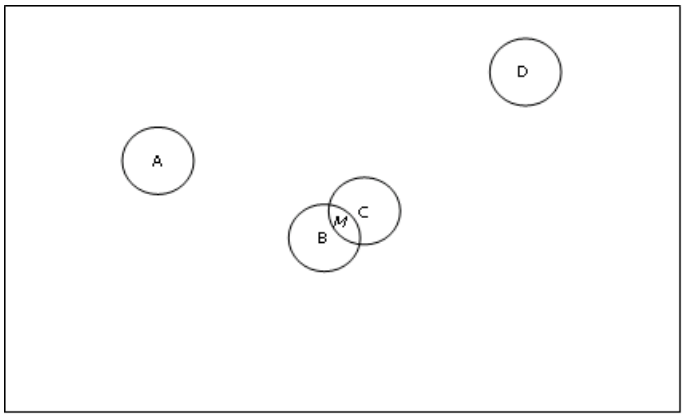
We recognize great potential in DTN for solving the problem of messaging across challenged networks of impoverished connectivity with some degree of mobility. Consider the following allegory: Alice has a message *M* for Donna. (See Figure 3.) By chance, Alice encounters Betty, who agrees to transport the message to Donna. On her way to see Donna, Betty happens to meet with Charlotte. Charlotte volunteers to take the message from Betty and deliver it to Donna. Soon after, Donna encounters Charlotte, who delivers the message from Alice.



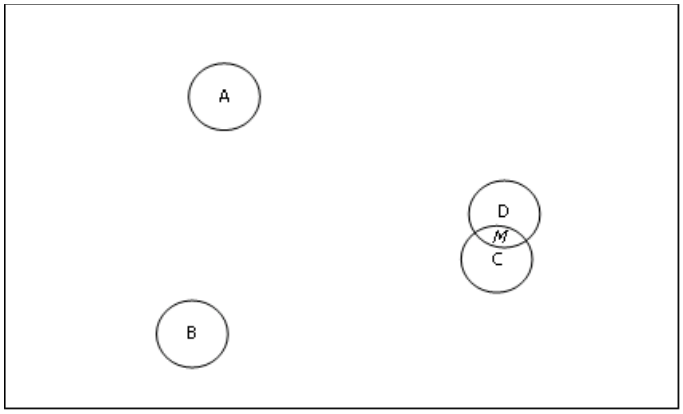
(a) Time t



(b) Time t + dt



(c) Time t + 2 * dt



(d) Time t + 3 * dt

Figure 3. Transitivity[9].

The DTN architecture provides an excellent starting position for implementing cooperative, transitive message relay between peers. Furthermore, an example implementation of DTN is available, the DTNRI (Delay Tolerant Networking Reference Implementation)[4]. The DTNRI forms a well-designed, cleanly written, and easily extensible code base, and is made available for public download. Convergence Layer adapters allow DTNRI to be extended to support arbitrary link technologies. Currently, DTNRI convergence layers exist for UDP and TCP. Arbitrary routing protocols can be added to the DTNRI by extending a new Router. At present, DTNRI has only a static router that depends on manually configured routing entries. No dynamic routers exist; this is directly attributable to DTNRI's lack of dynamic neighbor discovery. Neighbor discovery automates the discovery of peers and the configuration of links to those peers. (To state the same thing in different words: without neighbor discovery, Step 1 in Figure 2 above is a process of manual configuration.) Without neighbor discovery, DTNRI is not a viable solution to the problem of message routing in mobile populations of connection impoverished networks. Cooperative, transitive message relay requires dynamic discovery of peers, automated link configuration of discovered peers, and routing protocols that can effectively build dynamic routes by learning from patterns observed in such interaction between peers.

2.1.2.2 Directed Mobility

Research into directed mobility highlights the advantage of non-random mobility in connection impoverished challenged networks. Message ferries[10] use a combination of location awareness (i.e., GPS) and short bursts over long-range radio to arrange rendezvous points. Autonomous nodes (i.e., robots) move towards these rendezvous points to come into range with the message ferry to exchange messages via short-range radio. MV[11] improves on message ferries by tracking history of encounters in order to optimize ferry motion for maximum delivery rate throughout the network.

Other types of directed mobility leverage scheduled routes as opposed to robotic movements. DTLSR[12] is an adaptation of standard IP link state routing protocols to the challenges of unreliable links. DTLSR serves well in an environment where topology does not change rapidly, such as networks in which the dynamics stem from link instability.

2.1.2.3 Undirected Mobility

The harder challenge is faced by connection impoverished networks in environments of undirected mobility, since mobility patterns are less predictable. Because of the scarcity of links, global topology cannot be determined. Since there is no expectation of directly encountering the destination of each message, and topology is not available to advise routing decisions, messages are cooperatively relayed via encountered peers. Strategies turn to replication instead, relaying messages to more than one peer to increase chances of delivery. However, in addition to increasing the chances for delivery, replication also adds a burden to participants' limited storage resources as messages are queued through the duration of link scarcity. (Recall our assumptions of cooperative relay and limited storage.)

The extreme case of replication, Epidemic[13], relays a copy of every message in inventory to every peer encountered. In small populations with low message load, this strategy is effective. However, such unbounded replication has consequences for both storage and bandwidth resources as population size grows or message load increases. Epidemic cannot scale in networks of nodes with limited storage and bandwidth-challenged links: more storage resources are consumed queuing messages, and links are choked with the bandwidth requirements of replication.

A modified replication strategy, Probabilistic Routing Protocol using History of Encounters and Transitivy (PRoPHET[14][9]) proposes optimizations to Epidemic to scale to higher message load and larger populations. PRoPHET introduces a queuing policy to prevent storage utilization beyond a user configured quota. Forwarding decisions are guided by a heuristic that estimates the probability of future encounters for each message's destination; messages are only relayed to peers with higher probability of encountering the message destination.

Observing that human mobility is non-random, PRoPHET tracks the history of encounters with peers as an indicator for the probability of future meetings. This predictability metric also advises the queuing policy, which optimizes quota enforcement against delivery probability. Transitive routes are learned by exchanging routing information bases between peers. If peers are not encountered very often, then their routes should reflect a lower probability; this is accomplished by periodically aging the learned routes.

In addition to queuing policy, PRoPHET also utilizes acknowledgments to truncate message queues. Once a message is successfully delivered to its destination, PRoPHET floods the acknowledgment through the network so that every subsequently encountered peer can remove the message from its queue. Using simulations, Lindgren demonstrates that PRoPHET achieves higher efficiency (fewer dropped messages, higher delivery rate, lower delay) than Epidemic. No such router has been implemented for DTNRI.

2.2 Dynamic Routing in DTN

Since dynamic neighbor discovery is missing from DTNRI, no dynamic routing protocols have yet been contributed. DTNRI has well-defined concepts of Bundle (message unit) and Link (communication opportunity). We seek to bring in the concept of discovery in a manner independent of the underlying link technology. This is a considerable challenge, since each technology will have its own unique method of determining when peers enter the local area.

We introduce the concept of Discovery and Announce. The Discovery object represents the link technology, either by binding to a port to listen for beacons, or by regularly polling its local area. The Announce object represents an individual listening Convergence Layer instance, by sending out beacons or responding to polls with rendezvous information (such as IP address and TCP port). Once a Discovery object becomes aware of a neighbor's Announce object, the Discovery object creates a new Opportunistic Link and initializes it with the rendezvous information read from the Announce object. The DTN daemon receives a signal that the new link is available (ContactUpEvent). By subscribing to the ContactUpEvent handler, DTN routers are now capable of dynamic routing. Since our contribution of neighbor discovery, other researchers have contributed dynamic routers to DTNRI, including Epidemic (flood) and DTLSR.

2.3 Personal Area Wireless Technologies

Many personal area wireless technologies flood today's market: WiFi, Zigbee, and Bluetooth, just to name a few. Among the contenders, attributes vary, such as power consumption, bandwidth, and range. Not all technologies are as mature, and only a few enjoy a wide level of adoption.

We choose Bluetooth[5] as the personal wireless technology for this implementation. It is a mature technology with wide adoption: the hardware specification has been adopted by many vendors, the chipsets are embedded in many different form factors and different types of products, such as cell phones, PDAs, laptops, and even desktops. Development support is embedded in most modern Linux distributions as an extension to the Sockets API. (This support is known as the BlueZ project[6].) The RFCOMM[15][16] profile defined within the Bluetooth specification allows for in-order byte-stream semantics, similar to TCP, providing a straightforward path for integration into DTNRI as a Convergence Layer.

Bluetooth is designed as a personal area network with low power and short range, ideal as a wire-replacement technology (similar in function to USB). The encoding for Bluetooth's transport is based on time division multiplexing (TDM), requiring that one device act as timing master and allowing no more than seven other devices to join in to form an ad hoc piconet[17]. To span networks with more than eight members, piconets can be joined together in scatternets. However, throughput between piconets is limited to one-half of the bandwidth of the piconet; when a timing master becomes the bridge node, its piconet's throughput is also cut in half.

Research shows that Bluetooth scatternets are not the optimal choice for MANET mesh protocols. When optimizing scatternets for connectivity, deciding whether a connected graph exists that satisfies Bluetooth's degree constraint (no more than 7 partners in a timing-master's piconet) is NP-Hard[18]. Even using heuristics to simplify the decision to a polynomial problem, mesh convergence in Bluetooth involves a high enough delay that MANETs are unlikely to adopt this technology. Since our work is focused on connection-impoverished situations that typically involve interaction with one peer at a time (hop by hop), we do not attempt to construct scatternets, and therefore we are not concerned with the consequences of this particular optimization.

3 Solution

DTN[1] provides a first step in solving the problem of message routing in challenged networks of impoverished connectivity with undirected mobility. Specifically, the proposed design of DTN partially addresses the problem with bundling architecture that queues messages to persistent storage to endure arbitrary delay caused by link outages. Unfortunately, the current implementation of DTN (the DTN reference implementation[4], or DTNRI) lacks dynamic routing and neighbor discovery. The DTNRI also lacks substantive support for personal area wireless networking.

We propose extending DTNRI to address these issues. We contribute code modifications to implement support for the following three improvements to the DTNRI project:

1. implementation for personal area wireless technology, using the BlueZ[6] support available under Linux
2. design and implementation of a neighbor discovery architecture
3. implementation of a dynamic router based on the PRoPHET protocol based on work by Lindgren, et al.[14].

With these three modifications to the DTNRI project, we bring together a solution to the problem of message routing in challenged networks of impoverished connectivity with some degree of undirected mobility.

3.1 DTN Architecture

Our solution merges together existing technology from three distinct disciplines: we bring BlueZ and PRoPHET support into the DTNRI project. To introduce each of our three contributions, we present them as part of an overview of the DTNRI's architecture. After the overview of DTN presented in this section, the next sections move through the details of Bluetooth, neighbor discovery, and the PRoPHET protocol. The remaining sections present the details of implementing each contribution.

The DTNRI is based on the DTN architecture as set forth by Fall[1] and is designed as a peer-to-peer overlay application. Bundles are exchanged between peers in a store-and-forward fashion, with an emphasis on storage. By subscribing to each available networking technology on its host, DTN is able to bridge across heterogeneous link technologies. Bundles may be buffered to persistent storage for an arbitrary length of time. Delivery may be asynchronous, meaning that participants in the relay path need not be connected simultaneously. Using a technique called custody transfer, DTN proposes to relax IP's fate-sharing principle by delegating responsibility for a given message on a hop-by-hop basis. The overall design is interoperable and robust when faced with a wide variety of link technologies and operating conditions. (See Section 2.1.2.1 for greater detail.)

The DTNRI is a reference implementation whose objective is to "clearly embody the components of DTN architecture" and to provide "robust and flexible software framework for experimentation, extension and real-world deployment."[4] The DTNRI architecture is event driven, using a Tcl command interpreter as the event loop. Drawing directly from [1], the main actors in this event loop are Links, Contacts, and Bundles. A Link captures the state of the logical link between DTN peers (whether Scheduled, OnDemand, or Opportunistic). A Contact represents an active communication session over a Link. A Bundle is the basic messaging unit switched between peers. A Bundle's metadata includes the message's source address, destination address, creation time, expiration time, and class of service. (CoS is coarse-grained, similar to the postal service: priority, standard, bulk, etc.) As Links become available, DTN relies on a Router to decide which Bundles (if any) to forward to its peer.

Links utilize a software adapter layer, Convergence Layer, to normalize services offered by the underlying network technology to match expectations of bundle router. The bundle router then does not have to worry with the preservation of message boundaries, or in-line application-layer acknowledgments. Adaptations made by the Convergence Layer allow the Link to standardize on a simple API with the router, with primitives such as `interface_up`, `interface_down`, `open_contact`, `close_contact`, and `send_bundle`. An Interface represents the passive listener side of a Link. The primitive, `open_contact`, initiates an outbound connection, although a Contact can represent either inbound or outbound connections. Between the two sides of Contact, whether passive or active, the expectation is that Convergence Layer will handle any break in communication, monitoring link state to re-establish contact whenever the peer becomes available again. Later in this chapter, we present the details of how we implement a Bluetooth extension to the Convergence Layer API.

DTNRI's architecture is designed to be portable across many host architectures. To aid in code portability, system calls in DTNRI are implemented as object-oriented adapters in a separate namespace: oasys. For example, network support in the TCP Convergence Layer is accomplished by using the oasys TCPClient and TCPServer objects. Throughout the DTNRI architecture, object serialization (i.e., persistent storage) is available via extending the oasys interface, SerializableObject. Other interfaces within oasys are available for time calculations, threads, signals, and other system calls. Later in this chapter, we present the details of how we extend the oasys library to include support for Bluetooth sockets using Linux BlueZ (specifically, using the RFCOMM profile) and how we use these oasys Bluetooth objects to implement the Bluetooth Convergence Layer.

In the work done by Lindgren, et al., the PRoPHET protocol[9] specifies that neighbor discovery must exist as a service provided by the bundling host. The existing DTNRI does not have dynamic neighbor discovery. We design and implement a new module to DTNRI, the Discovery module. We specify an architecture independent of underlying network technology, to allow a common registry process for advertising to and discovery of DTN peers. In support of IP-based neighbor discovery, we also contribute to the oasys IPSocket family to introduce support for broadcast and multicast. In support of Bluetooth-based neighbor discovery, we contribute further to the oasys BluetoothSocket family, to introduce support for Inquiry and SDP. We present these contributions in further detail in a later section of this chapter.

The designers of the DTNRI architecture call out the BundleRouter interface as a list of event handlers. As DTNRI components touch and modify Links, Contacts, and Bundles, events are posted back to the bundling system. The central processing component is BundleDaemon, which coordinates all DTN services. BundleDaemon forwards most events to the BundleRouter interface. No dynamic router yet exists in the DTNRI. We contribute a dynamic router by implementing the PRoPHET protocol as an extension to the BundleRouter interface. Further details of our dynamic router implementation are presented in this chapter in a later section.

The following sections provide greater detail on each of our three contributions to the DTNRI project. First, we describe the Bluetooth specification and the support available in Linux via BlueZ. We describe how we integrate BlueZ support into the DTNRI project. Next, we present the details of our neighbor discovery architecture. We discuss the implementation of IP-based and Bluetooth-based discovery modules. To close the chapter, we discuss the PRoPHET dynamic routing protocol. We present the details of how we extended the DTNRI BundleRouter interface to contribute ProphetRouter.

3.2 Bluetooth

In this section, we present the details of how we extend the DTNRI project to include Bluetooth support. First we review a summary of the Bluetooth specification. Next we introduce the Linux BlueZ protocol stack and discuss its extension to the Sockets API. Finally we discuss the implementation of the BluetoothConvergenceLayer using RFCOMM socket objects. In a later section of this chapter, we discuss the implementation of Bluetooth neighbor discovery.

The Bluetooth specification[17] defines both a hardware and software interface. The interface between these two is the Host Controller Interface (HCI). Data passes over radio signals between two hosts as packetized stream in TDM slots. Further details are available in Appendix A.

The BlueZ project implements Bluetooth support in Linux in the form of an extension to the Sockets API. The address family is specified as PF_BLUETOOTH. The language of the Bluetooth specification defines "profile" where TCP/IP would use the term "protocol." For example, a UDP-like packet service is available via the L2CAP profile. A TCP-like byte-stream service available via the RFCOMM[15] profile. To integrate the Sockets API with Bluetooth hardware, BlueZ modifies the Linux kernel to translate the underlying system calls into HCI commands.

We contribute Bluetooth support to the DTNRI project. We extend the oasys library to include a Bluetooth client and server, using the RFCOMM profile. We also extend oasys to contribute Inquiry support for native Bluetooth neighbor discovery. Details of how the Bluetooth Inquiry process works are available in Appendix A. We extend oasys to contribute Service Discovery Protocol (SDP) support for registering and querying service availability. More details on Inquiry and SDP are presented in a later section in this chapter.

We contribute support for Bluetooth transport to the DTNRI by implementing a Convergence Layer based on the Bluetooth RFCOMM profile. The stream-based connection-oriented semantics of RFCOMM are similar to TCP. Due to this similarity, we modeled the implementation of the Bluetooth Convergence Layer after the DTNRI's existing TCP Convergence Layer. So much underlying functionality is common between these two that the DTNRI maintainers have factored out this commonality into the following hierarchy. The base class is ConnectionConvergenceLayer, which provides a framework for connection establishment and management. Much of the work of ConnectionConvergenceLayer is carried out by a helper class, CLConnection, that provides on-the-wire convergence layer protocol for session initiation and inline application-level acknowledgments. StreamConvergenceLayer extends ConnectionConvergenceLayer to provide management for message boundaries and other stream-related state. Our contribution is to extend StreamConvergenceLayer and the CLConnection helper class to use our oasys adapters for the RFCOMM transport.

In summary, our contribution of Bluetooth support begins by extending the oasys library. We provide Bluetooth transport by contributing oasys adapters for RFCOMM client and server objects. Using our contributed RFCOMM objects, we contribute the Bluetooth Convergence Layer to the DTNRI. We also contribute oasys adapters that perform Inquiry discovery. We contribute oasys adapters that perform SDP registration and query. Later in this chapter, another section details how our Inquiry and SDP contributions are integrated into our Bluetooth neighbor discovery implementation.

3.3 Dynamic Neighbor Discovery

The DTNRI lacks dynamic neighbor discovery services. We contribute a neighbor discovery architecture as well as IP-based and Bluetooth-based implementations. As noted previously, DTNRI has the potential of bridging together many distinct and unrelated networking technologies by subscribing to each technology from the application layer. To realize this potential, the DTN transport API is simplified to a message switching model. In similar fashion, neighbor discovery must be based on an abstract model. No single proximity technique will work for each network. Some technologies already have proximity detection built in. We seek to contribute a discovery model that abstracts these services across the heterogeneous technologies supported by DTN convergence layers.

Our solution begins with a generic architecture of Discovery and Announce. The Discovery object represents the networking technology; in terms of Socket API, there is a concrete class of Discovery for each address family: AF_INET, AF_BLUETOOTH, etc. Each Announce represents an Interface (i.e., the listening Convergence Layer instance). For each Interface, an Announce is registered with the Discovery object, detailing the network address for the Interface, as well as a polling or beaconing frequency. (The current implementation requires manual configuration of each Interface to be registered as an Announce. Future work may see this registration automated, perhaps as a flag passed in to the configuration of each Interface [e.g., set discoverable].) When neighbors come in range of the Discovery object, each of the Announce objects is enumerated. For polling implementations, Announce objects are learned by querying the remote registry. For beaconing implementations, the Discovery object passively listens, and Announce objects are learned as the corresponding beacons are received. Using the information discovered from the Announce object, the neighbor initiates contact to the advertised Interface by creating a new OpportunisticLink and configuring it with the provided address. The DTN router is responsible for opening OpportunisticLinks as they become available. Once a Link has been opened successfully, a ContactUpEvent is posted to the BundleDaemon. We discuss later in the chapter how ProphetRouter subscribes to ContactUpEvent to use as its "New Neighbor" signal.

Our contribution for IP-based neighbor discovery is a simple beaconing protocol. The IPDiscovery object is fully parameterized to accept broadcast, multicast, or unicast destinations; the default (if unspecified) destination is broadcast. We contribute modifications to the existing `oasys::IPSocket` object to support multicast and broadcast support. Beacons are sent as UDP packets. The UDP payload is a C struct that specifies the advertised Interface's IP address, protocol (TCP vs. UDP), and port number. The IPDiscovery object is set up as a UDP listener. One IPDiscovery object represents all instances of the IP Convergence Layer address family, whether TCP or UDP. Each IPAnnounce object serves as a registration: no beacons are sent out by the IPDiscovery object until at least one IPAnnounce object is registered. Each IPAnnounce object represents an individual Convergence Layer instance. The creation parameters for IPAnnounce serve to set listening IP, protocol, and port, as well as beacon frequency. When the IPDiscovery listener receives an Announce packet from a peer, a new OpportunisticLink is formed. The UDP payload distinguishes which type of ConvergenceLayer (whether UDP or TCP) and what IP address and port to use. Once the Link is initialized, the Discovery process posts a LinkChangeState event to the BundleDaemon. This event is forwarded to BundleRouter, which requests the Link to open. The router's command to open the Link is forwarded to the Convergence Layer, which attempts to initiate Contact with the peer. Once Contact is successfully initiated, the Convergence Layer posts a ContactUpEvent to inform the routing process that a new neighbor has been discovered. To summarize, our contributions of IPDiscovery and IPAnnounce integrate with existing DTNRI objects and interfaces (e.g., BundleDaemon, BundleRouter, Link, ConvergenceLayer, Contact). Using our contribution of broadcast and multicast support, and our contribution of an IP-based beaconing service (an extension of our contribution of the Discovery architecture), DTNRI is now capable of dynamically forming TCP- and UDP-based transports as peers come into proximity.

Our contribution of Bluetooth-based neighbor discovery is a wrapper to Bluetooth's built-in Inquiry and Service Discovery Protocol (SDP) processes. Similar to the design of IPDiscovery, BluetoothDiscovery serves as a registry for BluetoothAnnounce objects. BluetoothDiscovery iterates over each BluetoothAnnounce object to determine when the next poll occurs. Because Inquiry is implemented by the Bluetooth hardware, only one Inquiry can run at a time. As a consequence, we choose to implement BluetoothDiscovery polling as the minimum of the polling frequencies, should there be a difference in the configuration of the BluetoothAnnounce objects. When the poll time expires, BluetoothDiscovery initiates Bluetooth inquiry (using our contribution of `oasys::BluetoothInquiry`) to poll the local area for any in-range devices. The result of a successful Inquiry is an array of Bluetooth addresses (along with timing information needed to synchronize TDM, unique to each device). Upon creation, each BluetoothAnnounce object performs SDP registration with the local host to register its Bluetooth Interface. This SDP registration tells the host's SDP daemon to answer any SDP queries for DTN (as described by a 128-bit universally unique identifier, or UUID) with this Interface's RFCOMM channel. For each device discovered by Inquiry, BluetoothDiscovery opens an SDP query to determine whether the DTN UUID is registered. If the DTN UUID is advertised by the peer's SDP daemon, then BluetoothDiscovery uses the adapter's address and the SDP-registered RFCOMM channel to create a new OpportunisticLink using the Bluetooth Convergence Layer. To summarize, similar to how IPDiscovery integrates with existing DTNRI objects, our BluetoothDiscovery implementation allows DTNRI to dynamically form RFCOMM-based transports as Bluetooth peers come into proximity. Our BluetoothDiscovery implementation uses our contributions of Bluetooth Inquiry and SDP to extend our Discovery architecture.

Other proposals for neighbor discovery are under discussion in the DTN mailing lists[19]. Neighbor discovery is already well-developed in MANET groups[20]. When compared to discovery mechanisms in MANET, the simplicity of our neighbor discovery solution is justified, given that we only anticipate trivial MANETs of two or three nodes. While it is conceivable that our contribution of neighbor discovery may be superceded in the near future, our timely contribution serves to stimulate development in dynamic routing. To sustain our contribution, future work should focus on determining whether the concept of Discovery belongs at a per-address-family perspective or at a per-adapter perspective, such that multi-homed hosts have one IPDiscovery per IP address, and one BluetoothDiscovery per Bluetooth adapter.

3.4 PRoPHET Dynamic Routing Protocol

In this section, we first review the PRoPHET protocol as proposed by Lindgren, et al.[14] Next, we present the BundleRouter interface as defined within the DTNRI. We conclude the section with a description of how we extend the BundleRouter interface to bring PRoPHET functionality to the DTNRI.

The PRoPHET protocol was first proposed as an improvement to Epidemic[13], a simple replication-based router. Best suited for chaotic environments, simple replication routers attempt to improve delivery rates by copying each message to each node encountered. With increasing numbers of messages and nodes, this strategy does not make effective use of resources. To improve on Epidemic's performance, PRoPHET limits replication by making use of non-random human mobility, tracking encounters and computing a delivery predictability (0 <= P <= 1) for each known destination that indicates the probability of delivering a message to that destination. Each time a peer is encountered, summary vectors are exchanged, detailing predictability for other peers and listing which bundles are available for relay. PRoPHET regards each direct encounter with a peer as an indication that the peer is likely to be encountered in the future. This probability is calculated as follows, where node A is local, node B is the remote, and P_encounter is a tunable startup constant between 0 and 1:

```
P_(A,B) = P_(A,B)_old + ( 1 - P_(A,B)_old ) * P_encounter
```

Conversely, nodes are not useful relays if not encountered frequently; therefore, age is periodically calculated, where gamma is a tunable aging constant (0 <= gamma <= 1) and K is the number of elapsed time units:

```
P_(A,B) = P_(A,B)_old * gamma^K
```

PRoPHET also tracks transitive encounters. If A frequently encounters B, and B frequently encounters C, then B is good relay when A has messages for C (beta is a tunable transivity scaling factor, 0 <= beta <= 1):

```
P_(A,C) = P_(A,C)_old + ( 1 - P_(A,C)_old ) * P_(A,B) * P_(B,C) * beta
```

Replication highlights the strain between competing goals within the network. One goal is to maximize the chances of delivery for each message while reducing its delay. Higher degrees of replication translate to higher chance of delivery and shorter delay. Another strategy for improved chances of delivery is to buffer the message to persistent storage: the longer a message is buffered, the greater the chance it has of eventually being delivered. However, this competes directly with another goal, which is to optimize resource utilization within the network, with the understanding that participating nodes may have minimal bandwidth and storage resources. As the network scales up in the number of messages and participating nodes, unbounded replication will result in total consumption of all storage and bandwidth resources. The PRoPHET protocol proposes optimizations to enable the user to tune delivery improvement against resource consumption. Forwarding strategies address the optimization of bandwidth resources, and queuing policies address the optimization of storage resources. See Appendix B for more details on these optimizations.

Each encounter with a peer triggers a protocol exchange between PRoPHET nodes. After a brief negotiation of protocol version and timeout parameters, the protocol moves through an exchange of known routes, then negotiates which bundles to relay. Mixed in with the bundle exchange are PRoPHET acknowledgment, signifying that a bundle has reached its final destination within the PRoPHET region. Upon receipt of a PRoPHET ACK, a node may delete the associated bundle to free up its resources. These ACKs are flooded through the network with each peering session. (Further details of the PRoPHET protocol are available in Appendix B.) These exchanges take place across the data plane as Bundles exchanged between the peers, encoded as TLVs (as specified by the PRoPHET Internet Draft[9]).

3.4.1 DTNRI BundleRouter

In this section, we first present an overview of the BundleRouter API defined within the DTNRI architecture. Next, we describe the subset of handlers defined by this interface that we override to implement our contribution. We then transition into a discussion of how our PRoPHET implementation extends this interface.

The BundleRouter interface extends the BundleEventHandler interface. (Recall that the DTNRI architecture is built around a core event loop. Events posted to this loop reflect changes made to the main three actors in the DTNRI system: Bundles, Links, and Contacts.) As events are posted to the BundleDaemon, most events are forwarded to BundleRouter. The exhaustive list of events is defined in the DTNRI source code in `servlib/bundling/BundleEvent.h`.

Our implementation (`servlib/routing/ProphetRouter.h`) only overrides a subset of the event handlers inherited from BundleRouter. Recall that neighbor discovery integrates with BundleRouter through the Contact up and down events; these are seen by the router as `handle_contact_up` and `handle_contact_down`. (These handlers map directly to the PRoPHET interface requirements discussed in Appendix B, "New Neighbor" and "Neighbor Gone.") BundleRouter's `handle_bundle_received` signals an inbound bundle received via a Link (i.e., from a peer). Conversely, BundleRouter's `accept_bundle` affords veto power over local applications: refusal means that a local application cannot submit new bundles until queue space becomes available. The handler, `handle_bundle_delivered` signals that an application has received a bundle, meaning that the bundle has reached its final destination. PRoPHET responds to this signal by generating a PRoPHET ACK for the affected bundle. Another handler is `handle_bundle_expired`, which signals the end of a Bundle's TTL. The

handler, `handle_bundle_transmitted`, means that a bundle has been sent out over a Link. Another handler, `handle_link_available`, is how BundleRouter knows to open dynamically discovered links. The successful opening of a link triggers a ContactUpEvent. While there are other event handlers made available by BundleRouter, we have only mentioned the handlers used by our dynamic router implementation.

In the following section, we discuss the motivations behind our choice of the Façade design pattern. We present the consequences of this choice, such as the need to keep the DTN namespace distinct from that of our router contribution. We close with an overview of our implementation design.

3.4.2 Façade

We contribute ProphetRouter to the DTNRI. We implement the PRoPHET protocol as set forth by [9], by extending the DTNRI's BundleRouter interface to override the abovementioned event handlers. In order to make our PRoPHET protocol implementation more easily portable, we minimize its API by using the Façade design pattern. We use this pattern to decouple the namespace and functionality of our module from the DTNRI by forcing all calls through the "façade" storefront. Our goal is to contribute the Façade implementation back to the authors of the PRoPHET protocol as a reference implementation. The end result is that only our integration piece, the BundleCore interface, needs to be re-implemented when porting our PRoPHET contribution to a new DTN platform.

Using the Façade design pattern, we open the possibility of contributing our implementation of the PRoPHET dynamic routing protocol to other DTN implementations in addition to the DTNRI. The core logic of our contribution is the Encounter, an object that encapsulates the finite state machine defined by the PRoPHET Internet Draft[9]. The interface to Encounter is through Controller, which aggregrates the Encounters and demultiplexes messages between them. Each Encounter represents a peering session, and is indexed through the Controller interface by the Link over which the protocol Bundles are exchanged. The Controller is a member of ProphetRouter. The interface to Controller includes `new_neighbor`, `neighbor_gone`, `accept_bundle`, `handle_bundle_received`, `handle_bundle_transmitted`, and `ack`. The Encounter interface is simplified to `handle_timeout` and `receive_tlv`.

As a consequence of our design decision to decouple our namespace from DTNRI, any events or their related objects that pass through ProphetRouter must be translated to Façade wrappers before being forwarded through the Controller interface. In our DTNRI implementation, this means that `dtn::Bundle` must be wrapped in `prophet::Bundle`, and `dtn::Link` must be wrapped in `prophet::Link`. The DTN's Bundle and Link attributes and methods are forwarded to the corresponding Façade interface.

As mentioned earlier, protocol messages between PRoPHET nodes are exchanged as bundles in the data plane. The PRoPHET Internet Draft specifies TLVs for encoding the messages of each protocol phase. These TLVs may be concatenated. Our implementation defines a base class, BaseTLV, from which to derive specialized objects to represent individual protocol messages. We define ProphetTLV to represent the overall concatenated TLV header information; each ProphetTLV contains one or more individual protocol messages (as derived from BaseTLV). Serialization between bundles and TLVs is handled by the methods of BaseTLV and ProphetTLV.

The Encounter object tracks the state of a peering session between PRoPHET nodes. As PRoPHET nodes are encountered, whether directly or transitively, each is represented by a Node object. Node tracks the name of the node (i.e., DTN URI), the delivery predictability for reaching that node, and the timestamp of the Node's last update. Nodes are aggregated by the Table object. Table also implements convenience methods for search and updating routes, as well as a method for aging all nodes. Controller owns the Table object, but makes it available to Encounters through the Oracle interface, which also makes available the Stats, AckList, and BundleCore interfaces. Stats are used to compute FwdStrategy and QueuePolicy, and are updated by ProphetRouter as Bundles are transmitted. AckList aggregates Ack, which represents a Bundle that has been successfully delivered, and can therefore be purged from buffer space. Repository serves as PRoPHET's queue manager for stored Bundles and is based on priority queue, using an ordering imposed by QueuePolicy.

Events flow through the DTNRI BundleRouter interface into our ProphetRouter implementation. These events are posted as objects within DTNRI are modified. In the stream of events to which ProphetRouter subscribes by overriding the appropriate event handler, PRoPHET protocol messages are detected, the affected objects are translated from dtn namespace to prophet, and then the prophet objects are passed into our Controller interface to be processed by the relevant Encounter object. Conversely, in order for Encounter to effect changes on DTNRI objects, we contribute the BundleCore interface to abstract the core services implemented by the DTN host.

The BundleCore interface represents the abstraction of the system services offered by the DTN host to the Façade elements. For example, some of the services mapped through the BundleCore interface include `drop_bundle`, `send_bundle`, enumerate bundles (using an STL list), commit route to persistent storage, remove route from persistent storage, and timer functions. In addition to mapping DTN services to the Façade namespace, native DTN data types are translated to Façade intermediary data types using BundleCore's functionality. Further details about integrating BundleCore and Façade interfaces are available in the chapter entitled Developer Guide.

In this section, we have discussed how we implement the BundleRouter interface with our ProphetRouter contribution, and we have described how ProphetRouter decouples DTN namespace to integrate with the Façade's Controller for the processing of peering sessions as represented by Encounter. Conversely, our BundleCore interface provides a decoupled control path for Façade objects to manipulate objects within the DTN host. By decoupling our logic with the Façade design pattern and BundleCore interface, we make our code more easily portable to other DTN systems.

3.5 Conclusion

The DTN architecture provides message switching and queuing services that conceivably enable routing across connection impoverished challenged networks of undirected mobility. The DTNRI is a partial implementation of the services needed to solve this problem. Our contributions of Bluetooth, neighbor discovery, and the PRoPHET dynamic router extend DTNRI to enable it to meet the need for these networks.

Many researchers have turned their attention to the problems faced by challenged networks: MANETs and DTNs approach different facets of

challenged networking. Where connectivity is rich, MANET protocols adapt to loss of infrastructure by discovering the existing topology and facilitating IP semantics within the graph. Where connectivity is impoverished, DTNs adapt to asynchronous paths, encountering delays which likely violate the IP contract. Recent work[21] suggests a hybrid approach that prefers MANET solutions when available, and falls back to DTNs as a last resort. Future extension of our work would be well-advised to look further into this hybrid approach.

Future work may consider how to continue moving PRoPHET forward by researching bridging considerations between Internet regions and mobility regions. This bridging might be accomplished by the simple incorporation of static routes, such that part of the routing information base is not affected by the transience of learned routes. Additional improvements to the Epidemic protocol have been proposed by Musolesi, et al.[22], perhaps surpassing the advancements proposed by Lindgren, et al.[14].

4 User Manual

The purpose of this chapter is to guide the reader through the process of installation, configuration, and maintenance of the DTN overlay routing application. It is beyond the scope of this document to address a wide variety of deployment scenarios, so we limit our discussion to the following. We assume that the reader has a basic understanding of Linux system administration and access to physical hardware on which to install an operating system. We discuss the basic options for installing Debian Linux. Next we present the simple steps necessary to install the DTNRG Debian package. We move on to review the configuration options as specifically pertains to our contributions of Bluetooth links and the PRoPHET router. We describe the automation made available by our contribution of the neighbor discovery module, then we discuss the specific configuration for using the module. In closing, we present the configuration we used to set up a live demonstration of routing Bundles over Bluetooth links with the PRoPHET protocol.

4.1 Linux

The first step in our discussion of setting up a DTN router is to install and configure the Debian distribution of Linux. Begin by reading the installation instructions for the Etch version of Debian[23]. Full support for Debian installation is beyond the scope of this document; however, the reader is advised to observe the following suggestion. Ensure that DHCP or other network configuration is available, since Internet access is required for the Debian installation, as well as for later phases of the installation. When installing Debian, answer "Yes" to the question about using network mirrors, then select a site geographically nearby.

The following instructions are paraphrased from [4]; if there is a conflict between the following and the DTNRG site, the DTNRG site is authoritative. Once the Debian installation is complete, login to the system as root and modify the application management (apt-get) configuration file. Add the following lines to the file, `/etc/apt/sources.list`:

```
deb http://www.dtnrg.org/debian etch contrib
deb-src http://www.dtnrg.org/debian etch contrib
```

Reload the configuration with the command, `apt-get update`. Install the DTN platform by running `apt-get install dtn`. This concludes the steps necessary to install the application binaries for DTN support on this system.

In order to configure Bluetooth support for DTN, confirm BlueZ support under Debian. (BlueZ support should be enabled by default on Debian installations.) As root, execute the following command: `/usr/sbin/hciconfig -a`. If an error occurs similar to "No such file or directory", execute the following command as root:`apt-get install bluetooth`. To configure BlueZ to run in discoverable mode (for neighbor discovery), edit `/etc/bluetooth/hcid.conf` as root and add the following line to the `device` stanza:

```
discovto 0;
```

Reload the Bluetooth configuration with `killall -HUP hcid`. To confirm the change in settings requires a second host with Bluetooth capabilities. From the second host, run the command, `/usr/bin/hcitool inq`. A successful inquiry, returning the same Bluetooth adapter address reported by the `hciconfig` command above, means that discoverability is now enabled.

This concludes the steps necessary to configure Linux in preparation for Bluetooth support under DTN. The next section continues with further details of DTN configuration.

4.2 Configuring DTN

In this section, we present DTNRI configuration details that pertain to our contributions. The Debian package installed in a previous step from the DTNRG site is pre-configured, for the most part. To add Bluetooth support, add the following to `/etc/dtn.conf`:

```
interface add bt0 bt
discovery add btdisc0 bt
discovery announce bt0 btdisc0 bt interval=30
route set type prophet
```

The `interface` command instructs DTN to install a Bluetooth Convergence Layer to listen for inbound DTN requests on RFCOMM channel 10 on the default Bluetooth adaptor (hci0). The `discovery add` command sets up the default Bluetooth adapter (hci0) as an Inquiry agent to poll for neighboring DTN nodes within Bluetooth range. The `discovery announce` command associates the Bluetooth Convergence Layer with the local Bluetooth SDP server, registering the DTN UUID at RFCOMM channel 10, and commissions the Inquiry to poll for neighbors at 30 second intervals. Finally, the `route` command sets up a PRoPHET router instance. Further details on some of these commands are expanded

below.

The abovementioned minimal DTN configuration enables the reader to repeat the demonstration described later in this chapter. However, many more configuration options are available. For example, runtime configuration changes can be made by interacting with the DTN console, using `telnet localhost 5050`. A list of registered commands is revealed by typing `help`. Further instructions for each command is available by typing `help <cmd>`. Details on commands pertaining to Bluetooth, neighbor discovery, and Prophet are presented in the following sections. For more information on commands not covered below, visit the DTNRG website http://www.dtnrg.org or post your question to the DTN users mailing list at dtn-users@mailman.dtnrg.org (mailto:dtn-users@mailman.dtnrg.org) .

4.3 Configuring Bluetooth

Bluetooth support for DTN relies on BlueZ libraries in the underlying Linux operating system (see Section 4.1 for details on installing Linux) and is made available via the Sockets API, similar to how clients and servers operate over TCP/IP. Bluetooth servers bind to a socket to listen for incoming requests, and Bluetooth clients create an outbound socket to connect to a listening server.

Bluetooth support for DTN is put into effect through the Bluetooth Convergence Layer, which operates as a client/server pair. The `interface` command instructs DTN to install a Bluetooth server to listen for incoming requests. The `link` command instructs the DTN to initiate a client connection to a specified Bluetooth server (the listening `interface` of a neighboring DTN node). The astute reader has already noticed the absence of any previous reference to the `link` command in the DTN configurations we present. Due to the dynamic nature of PRoPHET and its reliance on neighbor discovery, no `link`s are created in advance, but only as they are discovered. Link creation and configuration is handled dynamically by the `discover` mechanism. For testing and troubleshooting purposes, however, it can be useful to statically configure Bluetooth `link`s, especially when used with the `static` router.

To list existing `interface`s, issue `interface list` at the DTN console. To delete an `interface`, note the name listed by the `list` command then issue `interface del <name>`. To override the use of the default Bluetooth adapter when adding an interface, pass in the address of the desired adapter by appending to the `interface add` command as `local_addr=<bluetooth address>`. Similarly, the option for overriding the default RFCOMM channel is appended as `channel=<number>`.

To list existing `link`s, issue `link dump` at the DTN console. To delete a `link`, note the name listed by the `dump` command, and issue `link delete <name>`. Links can be opened, closed, and set to available or unavailable via the `link` console command. Similarly to how interface options are specified, link defaults can be overridden by passing in desired parameters, appended to the `link add` command. The `link add` command has several required parameters (see Figure 4, below).

position	name	description
1	name	the handle used by DTN to refer to this link
2	next hop	transport address to remote peer (e.g., Bluetooth address and RFCOMM channel)
3	type	whether OPPORTUNISTIC, ONDEMAND, or SCHEDULED
4	conv layer	which transport is used by this link (bt for Bluetooth, tcp for TCP, etc)
ff	args	any CL specific options can be overridden at this point

Figure 4. `link add` required parameters.

For example, to create an ONDEMAND link to a known Bluetooth neighbor on RFCOMM channel 1, issue the following command:

```
link add bt0 11:22:33:44:55:66:01 ONDEMAND bt
```

(N.B. the link will not initialize properly unless the neighbor is already running a Bluetooth DTN interface listening on channel 1.)

4.4 Configuring Neighbor Discovery

Dynamic neighbor discovery agents in DTNRI are available for IP, Bonjour, and Bluetooth. The minimal configuration described above alludes to the `discovery` command family. The discovery framework is designed to accommodate both polling and beaconing strategies for locating nearby neighbors. A `discovery` agent is commissioned analogous to a Convergence Layer's listening Interface. Depending on the underlying transport's strategy, the `announce` either polls or beacons according to its `interval` parameter. Only one `discovery` is needed per protocol family. For each logical interface (i.e., Convergence Layer listener), a distinct `announce` should be registered to advertise its presence. (N.B. the Bonjour discovery module was not contributed by the author, so it is not discussed in detail in this work.)

The Bluetooth discovery strategy is based on polling, using the hardware-native Inquiry method. Similar to the Bluetooth Convergence Layer options available in the `interface` command, the `discovery` command allows the user to override the use of the default Bluetooth adapter by supplying `local_addr` to the `discovery add` command. The IP discovery strategy is based on UDP beaconing. The following parameters (see Figure 5) tune various functionality for the discovery module.

position	name	description
1	name	handle used by DTN for referring to this discovery instance
2	address family	bt for Bluetooth, ip for Internet Protocol
3	port	for IP, listening UDP port must be specified
opt	continue_on_error	(boolean) attempt to continue operating when faced with link errors

opt	addr	specify destination for IP beacons (can be broadcast, multicast or unicast)
opt	local_addr	specify which adapter for UDP listener to bind to
opt	multicast_ttl	how many hops to allow multicast beacons to propagate
opt	unicast	(boolean) if false, broadcast to beacon addr

Figure 5. Parameters for `discovery` command.

To remove a module, type `discovery del <name>` at the DTN console. Any associated `announce` objects will also be decommissioned and cleaned up.

The `announce` module serves as a registration for a Convergence Layer's listening interface. For IP, each `announce` registration becomes a separate UDP beacon payload, sent out at the interval configured by the `announce` creation. The beacon contains IP address, protocol type (whether UDP or TCP), port, and local DTN name (local_eid). When a listening `discovery` receives a beacon, it passes the information to the base class' neighbor discovery handler. For Bluetooth, each distinct registration is listed with the local SDP service under a Universally Unique Identifier (UUID), listing the local DTN name and RFCOMM channel. The interval becomes the frequency at which a Bluetooth Inquiry scans the local range for neighbors. If a neighboring Bluetooth device is discovered, the next step is to initiate an SDP query to determine whether the DTN service is offered. A successful SDP query is passed in to the base class' neighbor discovery handler. The following table (Figure 6) shows the parameters used by the `announce` command.

position	name	description
1	cl name	name of CL to advertise
2	discovery name	name of discovery to associate this announce with
3	cl type	whether TCP, UDP, or Bluetooth
4	interval	how often to repeat a beacon or initiate a poll
opt	cl_addr	address of CL to advertise
opt	cl_port	port of CL to advertise

Figure 6. Parameters for `announce` command.

To remove an `announce`, first list all discovery components with `discovery list`. Each `discovery` will list out its `announce` registrations. Remove the desired registration with the command, `discovery remove <announce-name> <disc-name>`.

Once the base class neighbor discovery handler receives a new event, it has all the information needed to initiate a Link to the discovered peer. First, ContactManager is queried for an existing Link matching the discovery. If none exists, a new Link is commissioned, and then a request to make the Link available is posted to the BundleDaemon. The end result is that once a Contact is formed over the discovered Link, a ContactUpEvent is posted. Dynamic routers (such as PRoPHET) subscribe to the ContactUpEvent to be informed of the arrival of new neighbors. If a Link is disrupted, a ContactDownEvent signals the end of communication over the associated link. (PRoPHET uses ContactDownEvent for the "Neighbor Gone" signal.)

4.5 Configuring PRoPHET

When installed as described in a previous section, the DTN package comes with support for the PRoPHET routing protocol. The PRoPHET protocol is designed to conform to a variety of situations, and as such, has many user-configurable parameters. The detailed PRoPHET user guide below is also available on the DTNRG site[24]. Many options are available for tuning the PRoPHET protocol, as listed below (with default initial values). Future work may lead to better understanding of ideal parameterization.

The first category of parameters is modified with the command, `prophet set`. The following chart lists currently available parameters.

name	domain	range	description
age_period	seconds	[0 .. 2^32)	timer used by PRoPHET to calculate node age and ACK expiry; defaults to 180 seconds
beta	fraction	[0 .. 1]	scaling factor for transitivity of learned routes; defaults to 0.25
encounter	fraction	[0 .. 1]	initial value for directly contacted routes; defaults to 0.75
epsilon	fraction	[0 .. 1]	minimum value for retaining routes; defaults to 1/255 or 0.0039
gamma	fraction	[0 .. 1]	aging constant for diminishing routes over time; defaults to 0.99
hello_dead	integer	[0 .. 2^32)	maximum number of silent HELLO_INTERVALs before peer is considered unreachable; defaults to 20
internet_gw	boolean	0 or 1	indicates whether PRoPHET node bridges to Internet; defaults to 0
kappa	integer	[0 .. 2^32)	number of milliseconds in a given time unit; defaults to 100
max_forward	integer	[0 .. 2^32)	maximum forwarding per bundle allowed by GTMX; defaults to 5
min_forward	integer	[0 .. 2^32)	minimum forwarding before eviction under LEPR policy; defaults to 3
relay_node	boolean	0 or 1	indicates whether node will relay messages for other nodes; defaults to 1

Figure 7. Parameters for `prophet set` command.

The next category of parameters relates to the queue policy to be enforced by PRoPHET. The default policy is FIFO. The chart below lists the

full set of available queue policies available via `prophet queue_policy=<name>`.

name	description
fifo	First In First Out: evict oldest first
mofo	evict most forwarded first
mopr	evict most probable (most favorably forwarded) first
lmopr	evict most favorably forwarded first (linear increase)
shli	evict shortest lifetime first (the bundle that will expire soonest)
lepr	evict least probrable first (lowest predictability)

Figure 8. Parameters for `prophet queue_policy` command.

Similar to the form used by queue policy, PRoPHET forwarding strategies are set by the `prophet fwd_strategy` command. The default strategy is GRTR.

name	description
grtr	forward if remote's predictability is greater than local's
gtmx	forward if grtr and the bundle has been forwarded less than max_forward
grtr_plus	forward if grtr and remote's predictability is greater than max predictability seen so far
gtmx_plus	forward if grtr_plus and the bundle has been forwarded less than max_forward
grtr_sort	forward if grtr and sort descending by P_remote - P_local (subtract local predictability from remote's predictability)
grtr_max	forward if grtr and sort descending by remote's predictability

Figure 9. Parameters for `prophet fwd_strategy` command.

Finally, the `hello_interval` describes the longest acceptable delay between protocol messages, and can be set to any value between 1 and 255 inclusive, representing time units of 100 ms. The maximum configurable delay is 25.5 seconds. The default setting is 20 (2 seconds).

4.6 Running a DTN

With our contribution of neighbor discovery, DTNRI is self-configuring. Nodes are aware when peers come into proximity, and automatically configure links to one another. With our contribution of PRoPHET, the router responds to these dynamic links by initiating its peering protocol with the new neighbor. Bundles are routed based on a URI naming scheme. The host part of the URI determines to which DTN peer the bundle is routed, similar to how a hostname or IP determines the host in a web URL. The remainder of the URI acts much like the port in a TCP or UDP socket in specifying which process on the DTN host is the source or recipient of a given bundle. Future work may determine a scheme by which DTN users can be addressed by a handle, as used in various instant messaging environments, that is automatically mapped to a DTN name in a similar fashion to how DNS names are mapped to IP addresses. Until such ease-of-use exists, DTN users must manually approach the issue of naming. To determine the local DTN name, issue `route local_eid` from the DTN console.

As an introduction to bundling, follow the instructions below to set up a DTN testbed with two hosts. For the purposes of this example, the hosts will be called Alice and Charlie. To better understand the dynamic processes automated by neighbor discovery and PRoPHET, this testbed will be configured with static links and static routes. As a preparatory step, comment out any configuration that was added in previous steps.

To establish Bluetooth links between Alice and Charlie, first gather information about the adapters on each host. From the Linux command line, issue the following command on Alice, then issue the same command on Charlie, noting the output for each.

```
/usr/bin/hcitool dev
```

If no output is written to the terminal following the line "Devices:", then take some time to acquire and install a USB Bluetooth adapter. At least one record should be printed, beginning with the `hci0` adapter. The Bluetooth address (write this down for the next step) looks like an Ethernet address, and will follow the adapter name (as in, `hci0 01:89:de:ad:be:ef`). Once the Bluetooth adapter addresses are known for both hosts, proceed to the next step.

The next step is to set up listening interfaces on each host. To add the default Bluetooth adapter as an interface on Alice, issue the following command at the DTN console.

```
interface add bt0 bt
```

Issue the same command at the DTN console on Charlie. Check the status of each interface by issuing the command, `interface list`. The result of this command is that a Bluetooth convergence layer is now listening on RFCOMM channel 10 on the default Bluetooth adapter on host Alice and on host Charlie for inbound connection attempts from DTN peers.

Now that each host has a listening Interface on its Bluetooth adapter, the next step is to set up the path for outbound DTN connections, or Links. In the following instructions, substitute the Bluetooth address learned from the Linux console on Alice for the term, <bt-a>, and for Charlie for the term, <bt-c>. From the DTN console on host Alice, issue the following command.

```
link add bt0 <bt-c> ONDEMAND bt
```

From the DTN console on Charlie:

```
link add bt0 <bt-a> ONDEMAND bt
```

Links are unidirectional data paths between DTN hosts. The creation of a Link does not necessarily imply that a Link is opened by DTN. To check the status of the newly added link, type `link dump` at the DTN console.

To review what has been configured so far: inbound connections will be received by listening Interfaces, and outbound connections are configured via Links. At this point, DTN still lacks a decision plane, otherwise known as a router. The next step is to associate DTN names with Links so that the bundling system knows where to forward each Bundle. Bundles without an explicit route remain in storage until a route becomes available. (Or said another way, the Bundle's destination must be associated with a currently available Link before the Bundle can be forwarded.) To set up a route from Alice to Charlie, issue the following command at the DTN console on Alice.

```
route set type static
route add dtn://charlie.dtn/* bt0
```

All Bundles addressed to any URI that matches the pattern, dtn://charlie.dtn/*, are now forwarded over the Link named bt0. Set up the reciprocal route by entering the following at the DTN console on Charlie.

```
route set type static
route add dtn://alice.dtn/* bt0
```

Now that interfaces are listening, links are pointing, and routes are ready, the next step is to generate some Bundles. (N.B. All of the preceding commands, entered at the DTN console, can be preserved as bootstrap configuration by editing `/etc/dtn.conf` and restarting the DTN daemon.)

Interfaces, Links, and routes are now configured between the hosts Alice and Charlie. To send a message from user Bob at host Alice to user Don at host Charlie, type the following command on a console at host Alice: `dtnsend -s dtn://alice.dtn/bob -d dtn://charlie/don -t m -p "Hi Don, how are you today?"`. (User-space applications are still an active area of research and are only available as CLI.) To receive the message, type the following command on a console at host Charlie: `dtnrecv /don`. Limited instructions on further options available from the dtn user utilities are available by invoking the binary with `-h`.

As each command is processed by the DTN daemon, a log is emitted to `/var/dtn/dtnd.log`. This log can be monitored by `tail -f /var/dtn/dtnd.log`. To change the level of logging from its default (info), invoke the DTN process with `-l` by editing `/etc/init.d/dtn`. Add `-l debug` to the end of the quoted string ARGS to generate fully verbose logs into `/var/dtn/dtnd.log`. Restart dtn for the changes to take effect.

4.7 DTN with PRoPHET and Bluetooth

To further highlight the possibilities of neighbor discovery and dynamic routing, consider the following demonstration. Three DTN hosts, A, B, and C, are deployed such that A and C are both stationary and outside of Bluetooth range of each other. B is mobile and can move within Bluetooth range of either A or C. A has Internet connectivity, and is running a DTN-to-SMTP gateway application. C is completely isolated, but knows of A's SMTP service. Some user on C creates a message to be sent as email to the author, designates A as the gateway. By moving between A and C, B becomes a bridge that carries the email from C to A. The author checks his email to confirm its delivery.

To repeat this demonstration, refer to the configuration and script listings in Appendix C and follow the procedure below.

1. Ensure that DTN is started and initialized on all three hosts.
2. Log on to host C.
3. Run the script, `create_and_send_email.pl`
    1. Type in the email address (use a valid address that you have access to).
    2. Type in the host name (e.g., alice) of host A.
    3. Type in your famous quote of choice, and end your message with a blank line, followed by <ctrl>-d
4. Log on to host A.
5. Run the script, `dtn2mail.pl`.
6. Move host B within range of A.
    1. Check DTN console for results of `route dump`
    2. Ensure that non-zero route exists between B and A.
7. Move host B within range of C.
    1. Check DTN console for results of `route dump`
    2. Ensure that non-zero route exists between B and C.
    3. Ensure that `bundle list` shows the email bundle.
8. Move host B within range of A.
    1. Monitor route status with `route dump`.
9. Once `dtn2mail.pl` shows a message has been received, check your email.

The demonstration can be varied to show the effect of transitivity. Restart DTN on each of the hosts, to flush out previously created routes and bundles. Then, instead of starting host B near host A, prevent B from learning of a route to A until after B makes contact with C. Then verify that A does not relay its bundle for C through B. Next, move B into range of A so that it learns the route to A. Then move B into range of C and verify that the message for A gets relayed to B.

5 Developer Guide

In this chapter, we first present an index of our contributions to the DTN reference implementation. Since the work represented by the DTNRI project precedes our work, we next give a summary of the DTNRI architecture. Within the context of the existing architecture, we move through the details of our contributions. We close the chapter with suggestions on how to extend this work.

5.1 Contributions

The contributions listed in this section have been submitted and accepted as change requests against the CVS repository of the DTNRI. (Instructions for CVS checkout are presented later in this chapter.) All contributions are shown relative to the repository, where DTN refers to the base. For example, DTN/README refers to the README file in the base directory of the CVS repository.

In Figure 10 below, we list the original work that we contribute to the DTNRI. Each of these files can be identified in the CVS repository by searching for the name "Baylor University" in the copyright boilerplate. The list below includes 115 files for a total of 18,712 lines. (The total for all code in the DTNRI repository is 930 files, and 278,024 lines.)

Directory	Contributed Files
DTN/oasys/bluez	Bluetooth BluetoothClient BluetoothInquiry BluetoothSDP BluetoothServer BluetoothSocket RFCOMMClient RFCOMMServer (15 files; 2,328 lines)
DTN/oasys/test	bluez-inq-test.cc rfcomm-client-test.cc rfcomm-server-test.cc sdp-query-test.cc sdp-reg-test.cc (5 files; 424 lines)
DTN/servlib/cmd	DiscoveryCommand ProphetCommand (4 files; 482 lines)
DTN/servlib/conv_layers	BluetoothConvergenceLayer (2 files; 784 lines)
DTN/servlib/discovery	Announce BluetoothAnnounce BluetoothDiscovery Discovery DiscoveryTable IPAnnounce IPDiscovery (14 files; 1,734 lines)
DTN/servlib/prophet	Ack AckList Alarm BaseTLV Bundle BundleCore BundleImpl BundleList BundleOffer BundleTLV BundleTLVEntry BundleTLVEntryList Controller Decider Dictionary Encounter FwdStrategy

	HelloTLV Link Node OfferTLV Oracle Params PointerList ProphetTLV QueuePolicy Repository ResponseTLV RIBDTLV RIBTLV Stats Table TLVCreator Util (51 files; 9,798 lines)
DTN/servlib/routing	ProphetBundle ProphetBundleCore ProphetBundleList ProphetLink ProphetLinkList ProphetNode ProphetNodeList ProphetRouter ProphetTimer (15 files; 2,256 lines)
DTN/servlib/storage	ProphetStore (2 files; 190 lines)

Figure 10. Contributions of Original Work.

Other contributions in the form of change requests against existing code are listed in the CVS log with a comment, "From Jeff Wilson." Excluding the files listed as "Original Work" above, we contribute 48 modifications to existing DTNRI files. Figure 11 lists some of the highlights of these modifications.

Directory	Modified Files	Comment
DTN/oasys/io	IPSocket	add multicast and broadcast support to be used by Discovery
DTN/oasys/tclcmd	LogCommand	read log rules from arbitrary filespec
DTN/oasys/util	Options	add support to parameter parsing for Bluetooth, uint8, and uint16 types
DTN/servlib	DTNServer	integrate support for ProphetStore and Discovery process
DTN/servlib/bundling	BundleActions BundleDaemon	expand API for routers to delete arbitrary bundle
DTN/servlib/bundling	BundleEvent	add attribute to BundleReceivedEvent to track contact on which Bundle arrives
DTN/servlib/bundling	BundleProtocol	integrate discovery process with ANNOUNCE type code and handler
DTN/servlib/contacts	ContactManager Link	expand API to support dynamically discovered links
DTN/servlib/naming	DTNScheme EndpointID Scheme	improve support for computing routes to dynamically discovered peers
DTN/servlib/reg	AdminRegistration	integrate discovery process with ANNOUNCE type code and handler
DTN/servlib/routing	BundleRouter TableBasedRouter	migrate basic router functionality upstream so that non-table-based routers can inherit from it
DTN/servlib/storage	GlobalStore	integrate support for persistent storage of Prophet routes

Figure 11. Modifications Submitted against Existing Files.

5.2 DTN Source

We present two options for obtaining the source code to the DTNRI project. The first option acquires a snapshot of the source tree, taken at a release point by the project maintainers and made available via the Debian apt-get utility. The second option tracks daily changes more closely,

and is made available via CVS repository.

For the first option, as root change to the desired directory (e.g., /usr/src). Execute the command,
`apt-get source dtn.`
The apt-get utility downloads a tar-gzip file (e.g., dtn_2.5.0-1.tar.gz) to the current directory, then expands the archive to a new directory (e.g., dtn-2.5.0). For convenience, make a link:
`ln -s dtn-2.5.0 dtn.`
This source tree represents the most recent version released by the DTNRI project.

The second option gains access to the most current work via CVS, including bug fixes and experimental new features. First, as root, install the CVS utility by executing `apt-get install cvs`. Next, to obtain code from the CVS repository, follow the instructions posted at [4]. For convenience, the instructions are repeated here; however, any changes posted to the above site will supercede the following. With user privilege (not root), change to your home directory and create a src directory. Log in to the DTN repository (use "bundles" as the password) and check out the repository:

```
cvs -d :pserver:anonymous@code.dtnrg.org:/repository login
cvs -d :pserver:anonymous@code.dtnrg.org:/repository checkout DTN2
```

To keep current with daily development, the repository must be refreshed periodically:

```
cvs -q up -Pd
```

Configure and compile the source with the standard autoconf procedure. Invoke `./configure --help` for options available to the configure script. It is likely that this first attempt will fail. To correct the most common errors, log in as root and install the following packages with `apt-get install`.

- gcc
- g++
- debhelper
- tcl8.4-dev
- libdb4.4-dev
- libbluetooth-dev
- libxerces27-dev

5.3 Test Design

As mentioned earlier in the list of contributions, the DTNRI project has close to 1,000 source files with over 275,000 lines of code. The project is large enough that some testing automation is required. Although general purpose and open source testing frameworks are available for C++, the DTNRI project maintainer has created a unique framework using Tcl scripting for integration. Overall test results are gathered and displayed using [25]; however, we do not visit the details here.

The UnitTest framework integrates Tcl parsing with C++ class hierarchy. Precompiler macros hide much of the complexity of inheritance. A sample unit test is available at `DTN/oasys/test/sample-test.cc`. The DECLARE_TEST macro takes one argument, TestName, and is followed by a curly brace section of test code. Within this test code, other macros perform boolean checks to validate object functionality. Embedded in the macros are log commands that echo test status to stdout. If all the boolean checks pass, then UNIT_TEST_PASSED is returned before the end of the curly brace scope. If any of the boolean checks fail, the macro will return UNIT_TEST_FAILED. At the time of this writing, unit test integration with Tcl is not functional. Test results must be monitored manually. The design, however, is that Tcl parseable output is gathered by `DTN/oasys/test-utils/UnitTest.tcl` for an overall report of how many test modules passed and failed.

To fire the Unit Tests we contribute for Prophet components, change to `DTN/test` and executing the following.

```
ls unit_test/prophet*.cc | sed 's/\.cc//' | xargs make
```

Execute each test individually and note the results. Since Bluetooth does not have a loopback adapter, no Unit Tests are available for our Bluetooth contributions. However, in the following paragraphs, we describe our Bluetooth module validation using the DTNRI integration test framework.

The integration testing framework is built around Tcl script libraries that manage DTN configuration and instantiation. Individual DTN nodes are configured by IP address or hostname (fixed network access is required for test managment via ssh). Each node is referred to by a node ID, abstracting away the physical address. The abstraction is so well partitioned that any number of nodes can be instantiated by using localhost loopback, as long as each node has a distinct port for its command channel. For Bluetooth support, the default adapter is assumed to be the `hci0` interface. However, this value can be explicitly declared per node using the `-bluez_adapter` option. (Note that Bluetooth tests cannot be run against localhost loopback; instead, Bluetooth tests require two physical adapters installed in two distinct hosts.)

The first step in using the integration tests is to set up a network file, describing the nodes to be used in your testbed. Assume that your two test nodes are hosts Alfred and Baker. Your network file, `DTN/test/nets/mynet`, should resemble the following.

```
net::node 0 alfred 10001
net::node 1 baker 10001 { -blue_adapter hci2 }
```

Use the `-blue_adapter` option to override the default of "hci0." The next step is to set up ssh access between Alfred and Baker, preferably

using keys so that password authentication is not required. (Read your host system's man page on ssh_config and ssh-keygen to learn more.) Now test your nets file with an individual test. Change directory to `DTN`. Execute `./test-utils/run-dtn.tcl -d test/bluez-links.tcl --net mynet` to fire up a DTN testbed using static routes over Bluetooth links. We also contribute the following test scripts in `DTN/test`.

`bluez-inq.tcl`	compares results from DTN BlueZ Inquiry against Linux BlueZ hcitool
`bluez-rfcomm.tcl`	tests DTN RFCOMM transport between two Bluetooth-enabled hosts
`bluez-sdp.tcl`	compares results from DTN SDP query and Linux BlueZ sdptool's query, using DTN SDP registration
`discovery.tcl`	configures DTN testbed in linear topology to exercise IP discovery
`prophet.tcl`	configures DTN testbed in linear topology, simulates IP discovery, tests for successful Bundle delivery

Figure 12. Test Scripts in DTN/test.

Once these tests have run successfully, experiment with the other framework bootstrap script, `DTN/test-utils/run-tcl-tests.tcl`. Groups of integration tests are defined within this script. To execute the entire BlueZ validation suite (listed above) all at once, fire the script with these options: `./test-utils/run-tcl-tests.tcl bluez --net mynet`. Other DTN project maintainers have contributed other test suites. The "basic" suite tests other DTN core functionality, apart from our contributions.

5.4 DTNRI Architecture

To better communicate the context of our contributions, we now turn our attention to the architecture of the DTNRI. The well-designed, extensible object-oriented design of this system is anchored by the BundleDaemon, the event demultiplexer and dispatcher. A Tcl interpreter drives the DTNRI's event loop and is used for bootstrap configuration and runtime control. Objects such as Bundles, Links, and Contacts interact with one another and post events to the BundleDaemon to record the changes made. In turn, other handlers register with BundleDaemon to respond to changes made to objects of interest. The full list of events is defined in the source file, `DTN/servlib/bundling/BundleEvent.h`.

The decision plane is helmed by the BundleRouter interface. By subscribing to Link and Bundle events, the BundleRouter interface can implement an arbitrary routing protocol, and advise the BundleDaemon when to forward a given Bundle over a particular Link. Our ProphetRouter contribution inherits from BundleRouter.

A Link interacts with a Convergence Layer as its adapter to subscribe to user-accessible APIs for the underlying transport. At the time of this writing, only TCP and UDP Convergence Layers are fully operational (apart from our Bluetooth contribution). A Null Convergence Layer is available for the testing framework, and an Ethernet Convergence Layer has been mostly deprecated. Links represent communication opportunities over a specific transport to a remote DTN node. Links can be Opportunistic, OnDemand, or Scheduled. Together with Convergence Layer, Link provides a smaller interface to BundleRouter, simplifying the complexity of the transport to "open", "close", and "send_bundle". Interface represents the passive listening side of the Convergence Layer. Inbound connections become OpportunisiticLinks. Once the Convergence Layer has established its protocol with the peer, a Contact is formed to represent the real-time connection. (ProphetRouter uses the ContactUpEvent to implement its "New Neighbor" signal.)

The DTNRI project strives to achieve interoperatibility with a wide variety of platforms. To better abstract host-specific system calls, DTNRI wraps various system services (such as networking, thread support, or memory management) in object-oriented adapters, or "oasys". Common I/O methods form the object hierarchy for both the IP and the Bluetooth socket families. Persistent storage support maps a common API across support for files, BerkeleyDB, and MySQL, so that run-time configuration selects which implementation to use. Other system services, such as signals, threads, and debug logging, are similarly based on an object-oriented hierarchy. In the following discussion of the Bluetooth convergence layer, oasys Bluetooth sockets specifically adapted to the RFCOMM profile form the foundation for transport.

5.5 Bluetooth CL

The Convergence Layer adapts an underlying network technology to a simple interface, and compensates in software for any way the network protocol lacks, such as reliability or message boundaries. Our Bluetooth Convergence Layer contribution is based on Bluetooth's RFCOMM profile, which is used for serial byte streams, such as modem data and control channels (e.g., RS232). By selecting RFCOMM, with its connection-oriented stream semantics, our work closely parallels the existing TCP Convergence Layer. Both the TCP and the Bluetooth Convergence Layers inherit from the same hierarchy: ConvergenceLayer, ConnectionConvergenceLayer, and StreamConvergenceLayer. As a result of shared functionality, only connection setup and teardown and byte transport are directly implemented by the Bluetooth Convergence Layer. All other functions, such as message boundary and on-the-wire Convergence Layer protocol, are implemented by the base classes.

For the reader's convenience, part of the Convergence Layer interface is presented in the following table. See `DTN/servlib/conv_layers/ConvergenceLayer.h` for the full interface, which specifies the API common to all CLs.

ConvergenceLayer API

return	method name	parameters	description
bool	interface_up	Interface*, int argc, const char** argv	bring up a new interface
bool	interface_down	Interface *	close down the interface
bool	open_contact	ContactRef&	open a new contact (establish any CL specific connections)
bool	close_contact	ContactRef&	clean up any state associated with contact (in response to ContactDownEvent)
void	send_bundle	ContactRef&, Bundle*	attempt to send bundle on current link

The ConnectionConvergenceLayer interface tracks open links using a Connection object. When Contacts are opened, whether outbound (Link)

or inbound (Interface), a reference to the Connection is stored in the Contact's cl_info slot. The management of this Connection object is common to all connection-oriented protocols, and is implemented by ConnectionConvergenceLayer.

The StreamConvergenceLayer aims to share as much functionality as possible between protocols that have in-order, reliable, delivery semantics. Bundles are broken into segments that are sent sequentially. Bundles are not interleaved on-the-wire; only one is in flight at a time. Runtime options for segment size and segment acknowledgments can be set via configuration parameters. When acknowledgments are enabled, the receiver acknowledges each segment of the bundle as it is received. Keepalive messages are used to keep the underlying connection open. To optimize for flow control, ACKs are sent before bundle segments. State on inflight bundles is tracked; state is deleted upon receiving the final ACK.

The Bluetooth Convergence Layer completes the picture by implementing a concrete class for Listener and Connection using RFCOMM socket objects. The `interface_up` method instantiates and configures a Listener to correspond to the Interface. The Listener is an independent thread that blocks on `accept()`, waiting for incoming RFCOMM connections. The Connection object has passive and active mode. The passive Connection object is instantiated to handle incoming connections. The active object is used to initiate outbound Links whenever the `open` action is performed on a Link. The ConnectionConvergenceLayer and the StreamConvergenceLayer inherit controls from the base class interface to manipulate the Listener and Connection objects; the Bluetooth concrete classes implement all the technology-specific actions, such as `socket()`, `listen()`, `accept()`, `connect()`, `send()`, and `receive()`.

5.6 Neighbor Discovery

The goal of neighbor discovery is to automate the detection of neighbor proximity, and in response to a detected neighbor, to automate the configuration of an opportunistic link. Prior to our contribution of the design and implementation of neighbor discovery, no such solution was available to the DTNRI. Working with the principal architect of the DTNRI, we design and contribute the neighbor discovery framework, along with IP and Bluetooth implementations.

Our neighbor discovery framework abstracts the two goals of proximity detection and automatic link creation with the concepts of advertisement and response. Some technologies, such as IP, are better suited for a broadcast or multicast approach, whereas other link protocols, such as Bluetooth, are limited to a polling approach. The Discovery object represents the address family, and serves a dual role, both as a local registry of listening Convergence Layers, and as a listener or poller (depending on the implementation). To advertise an Interface (listening instance of a Convergence Layer), the user registers an Announce, which captures the transport-specific address of the Interface and sets the frequency of polling or beaconing. Until the first Announce is registered with the Discovery, no beaconing or polling takes place. Depending on the capabilities of the underlying technology, either beacons are sent out with rendezvous information advertising the registered Interfaces, or polling occurs at regular intervals. Individual beacons are sent out at the configured interval set by the respective Announce registration. Polling occurs at the minimal interval configured set by the registered Announce objects.

In addition to contributing the neighbor discovery design, we also contribute the implementation for Bluetooth and IP discovery. Bluetooth hardware has a discovery mechanism built in, a process called Inquiry. The Bluetooth Discovery adapter is just a thin wrapper around the Inquiry process. (See Chapter 4, Section 1 about setting discoverability for Bluetooth adapters.) However, IP has no direct knowledge of local proximity. The IP Discovery adapter makes up for this lack by sending out a beacon periodically. The beacon destination can be configured as a unicast, broadcast, or multicast address to approximate link layer discovery by distributing the beacon as far as possible. Any listening Discovery adapters within range receive the beacon, which has rendezvous information on how to contact the registered Convergence Layer.

To complement our Discovery implementation, we also contribute the Announce implementation for Bluetooth and IP. An Announce serves as a registration for the corresponding Discovery object to advertise available Convergence Layers. For Bluetooth, this registration also configures the polling frequency for the Bluetooth Discovery's Inquiry. Each adapter discovered by Inquiry is then queried by SDP to query the availability of DTN. For IP (whether TCP or UDP), the announce parameters establish how often the associated beacon is sent out. Each UDP beacon contains the IP address and port of the registered Convergence Layer, and a byte code to distinguish whether the Convergence Layer is TCP or UDP. Other researchers contributed the Bonjour discovery adapter, which does not use the announce convention, but combines the CL registration into the discovery parameters. (More information is available in the source code, `DTN/servlib/discovery`.) For each neighbor discovered, the discovery module creates a new opportunistic link and generates a link state change request. The end result is that a ContactUpEvent is generated by DTN (the PRoPHET router uses this as its "New Neighbor" signal).

With this overview in mind, we now turn to the specifics of how we implement our Bluetooth Discovery solution. Bluetooth Inquiry is made available to the Linux programmer via the BlueZ API using `hci_inquiry`. The parameter signature is listed below.[26]

```
int hci_inquiry(int dev_id, int len, int max_resp, const uint8_t *lap,
                inquiry_info **ii, long flags)
```

The device id `dev_id` represents a local Bluetooth adapter. To determine the device id of the first available Bluetooth adapter on the local system, invoke `hci_get_route(NULL)` and store its return value. Open the device by passing the device id in to a call to `hci_open_dev(int dev_id)`. Once the device id is located, and opened, create an array of `inquiry_info`. The size of this array is to be passed in to `hci_inquiry` as the parameter, `max_resp`. The flags should be set to IREQ_CACHE_FLUSH to request a "clean" Inquiry each time, else set to 0 to possibly include stale entries from previous Inquiries. Once the call to `hci_inquiry` returns the number of neighbors found (or -1 on error), the `ii` parameter points to an array describing the devices found nearby. The first field in each struct is the important one, `bdaddr`, the remote adapter's Bluetooth address. Walk each record in the array to learn the Bluetooth address of each adapter in range at the time of Inquiry. (This process is demonstrated in `DTN/oasys/bluez/BluetoothInquiry.cc`.)

Using the Bluetooth adapter address discovered by Bluetooth Inquiry, the next step is to determine whether DTN is a service offered by the remote host. Bluetooth defines the Service Discovery Protocol for registering local services and querying remote registries. Our Bluetooth Discovery implementation initiates an SDP query against each discovered Bluetooth adapter. (The full source listing of the SDP query is found in `oasys/bluez/BluetoothSDP.cc` on lines 46 through 188.) Before initiating an SDP query against a discovered Bluetooth device, the

UUID of the desired application must be known. The informal UUID used by the author's contribution to the DTNRI is listed below.

```
DCA38352BF6011DAA23B0003931B7960
```

(This arbitrary 128-bit string was generated by a Macintosh using the `uuidgen` utility. Future work should include a formal push to register DTN as a known application with a reserved UUID within the Bluetooth SDP protocol. It is likely that the Bluetooth SIG will issue a different number in response to a formal request.) Create an array of `unsigned int` filled with the aforementioned 128 bit UUID. Pack this array into a `uuid_t` using `sdp_uuid128_create` as demonstrated on line 72 of `BluetoothSDP.cc`. The next step is to create a couple of request records using `sdp_list_append` to limit the scope of the SDP query. Open a session handle to SDP with `sdp_connect`. Once the parts have been assembled and prepared, execute the query with a call `sdp_service_search_attr_req` (line 85). A linked list is returned via the `seq` parameter of all SDP records matching the query (recall that the query scope is limited to the DTN UUID). Details of parsing through the linked list are available in the commented source code listing. Each record is parsed for the DTN's route id (configured with "route local_eid" at the DTN console) and RFCOMM channel. This information is used to create a link to the discovered peer.

The complement to querying a remote SDP registry is to register a service with the local SDP registry. This registration makes information available to answer to incoming SDP queries. To register with the local SDP daemon, the application fills in a service registration with details on the nature of the service. (See source code listing for `oasys/bluez/BluetoothSDP.cc`, lines 207 through 278 for details.) The DTN service is described by the aforementioned 128-bit UUID. Since RFCOMM's byte stream is built on top of a packet stream (L2CAP), both UUIDs are packed into the registration. The local DTN route ID is published into this record, associating it with the RFCOMM channel, the RFCOMM UUID, the L2CAP UUID, and the DTN UUID. The BlueZ implementation of the SDP daemon and library functions uses Unix sockets to hold open this registration as long as the process that creates it. The file descriptor is opened by the constructor of the Bluetooth Announce, and closed by the destructor.

The IP discovery module opens a UDP listener, binding to the socket specified by the discovery command's parameters. The default behavior is to bind to INADDR_ANY, but the `local_addr` parameter overrides this with a specific adapter's interface. No outbound beacons are sent until the first announce is registered. All beacons are sent to the broadcast address unless the discover parameter `addr` specifies a multicast address (any IPv4 address in the CIDR of 224.0.0.0/4). An alternate broadcast address can be specified. To send to a unicast address, specify the destination with `addr` and set the `unicast` flag to true. The announce registration requires the `interval` parameter and `cl_type` (UDP or TCP), but other parameters can be left to the default behavior. If no Convergence Layer address is specified via the `cl_addr` parameter, then the beacon advertises INADDR_ANY, which causes the beacon receiver (the remote discovery listener) to use the IP from the beacon's packet header. The default CL port is 4556 but can be overridden with the `cl_port` parameter. Beacons are sent according to the interval set by the announce parameter `interval`.

5.7 PRoPHET

We contribute a dynamic router implementation based on the PRoPHET protocol[9], designed to operate in DTNs of mixed mobile and stationary devices. The core of this protocol centers on tracking the interaction of participating relays. Routes are ranked by how often relays are encountered by their peers: routes encountered more often should reflect a higher probability of future encounter, and conversely, older routes should decay as time passes. Frequent interaction results in a higher ranked route.

We design our PRoPHET implementation to be decoupled from the host DTNRI implementation using the Façade design pattern, which is useful for limiting the exposure of an API between modules of a system. This has the desirable side-effect of maximizing code portability. The interface to our implementation is defined in neutral terms, independent of the host system's implementation, so that integrating the module into the host system is reduced to the process of implementing the host interface as a proxy or shim layer.

On the Façade side, factoring out components and unit testing are simplified by reducing or removing the interaction with the host system. Our PRoPHET implementation is designed as an object-oriented finite state machine. Information flowing from the DTN host into the Façade interface goes through the Controller interface, which is loosely based on the DTNRI's list of event handlers. We implement PRoPHET in a distinct namespace, apart from the host implementation, and define our interfaces with respect to neutral objects. Any system calls made from our Façade objects are mapped through our BundleCore interface. Integration between our Façade implementation and a particular DTN host environment is achieved through mapping our neutral Façade objects through the DTN host-specific wrappers for Bundles and Links, and through the implementation of the BundleCore interface.

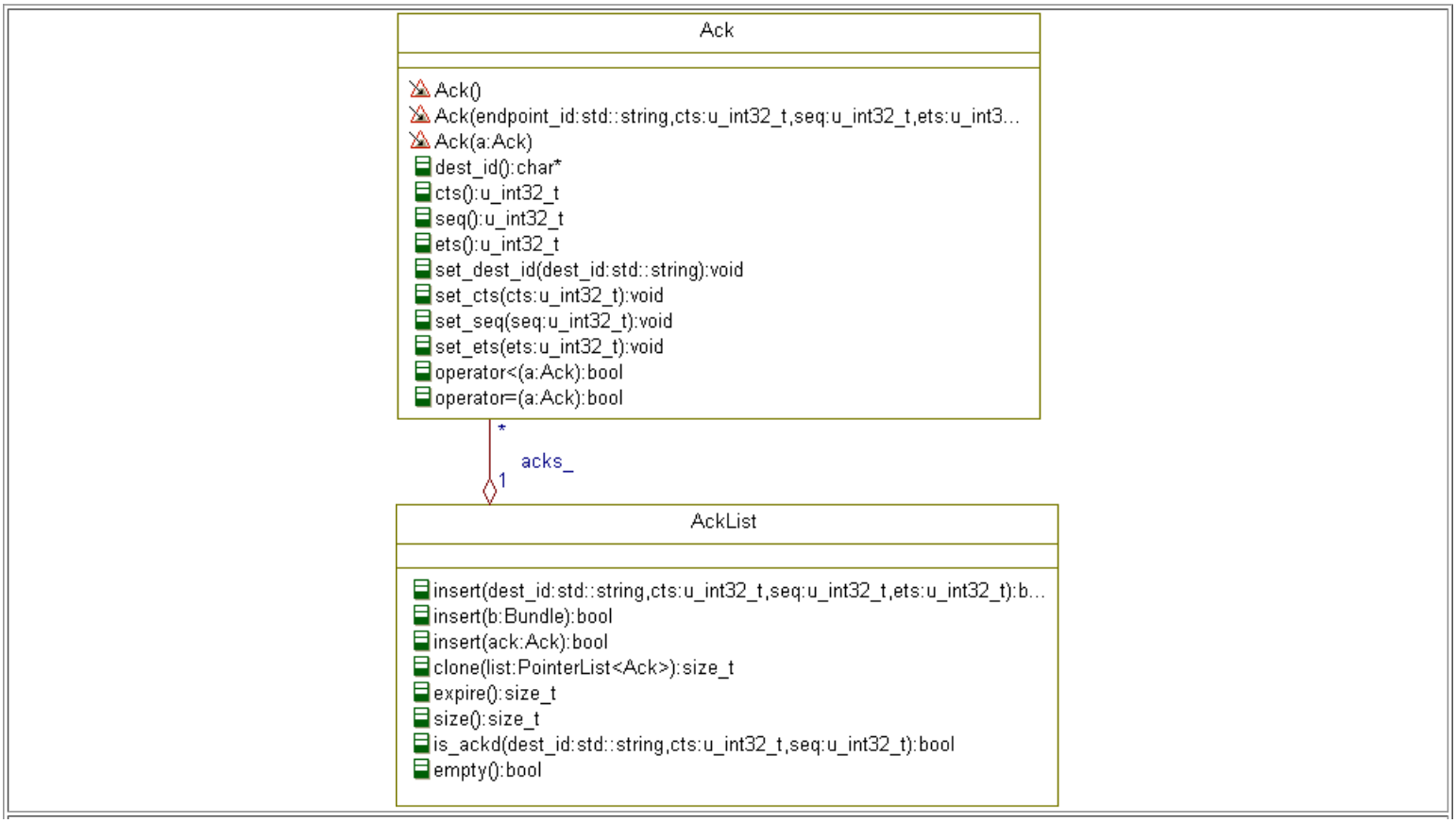
Our Façade implementation is built on the following component design. A Node represents a current route, storing its name, predictability value, and age (as well as some flags). Nodes are aggregated by Table, which also answers what a route's current predictability value is, updates a route based on direct or transitive algorithm, and updates all routes with age algorithm. Repository monitors the DTN host to enforce PRoPHET's queuing policy on the local Bundle store, and keeps Bundles in eviction order. Our Bundle wrapper is the Façade interface into the DTN host's Bundle object. Similarly, our Link wrapper is the Façade interface into DTN host's Link object. The BundleCore interface maps DTN host services for use by Façade objects, including bundle operations (send, receive, delete), routing utilities (whether a name matches a given route), and persistent storage facilities (to serialize routes to/from disk). Dictionary manages name indexing for routing information in an Encounter's protocol exchange. BaseTLV is the base class for each TLV used in protocol exchange, and is specialized to HelloTLV, RIBTLV, RIBDTLV, BundleOfferTLV, and BundleResponseTLV. ProphetTLV is the outermost wrapper TLV for aggregating any number of BaseTLV-derived objects used in protocol exchanges. FwdStrategy forms a base class for forwarding strategies. QueueComp is the base class for queue policy comparators, used by Repository to achieve eviction ordering. Decider is a base class for the forwarding decider which determines whether to relay a bundle to a given peer, based on the active FwdStrategy. Encounter represents the protocol state machine that manages protocol exchanges with a given peer. And bringing all of the above together, Controller forms the Façade interface controller that manages Encounters, mapping bundle events to appropriate Encounter instances.

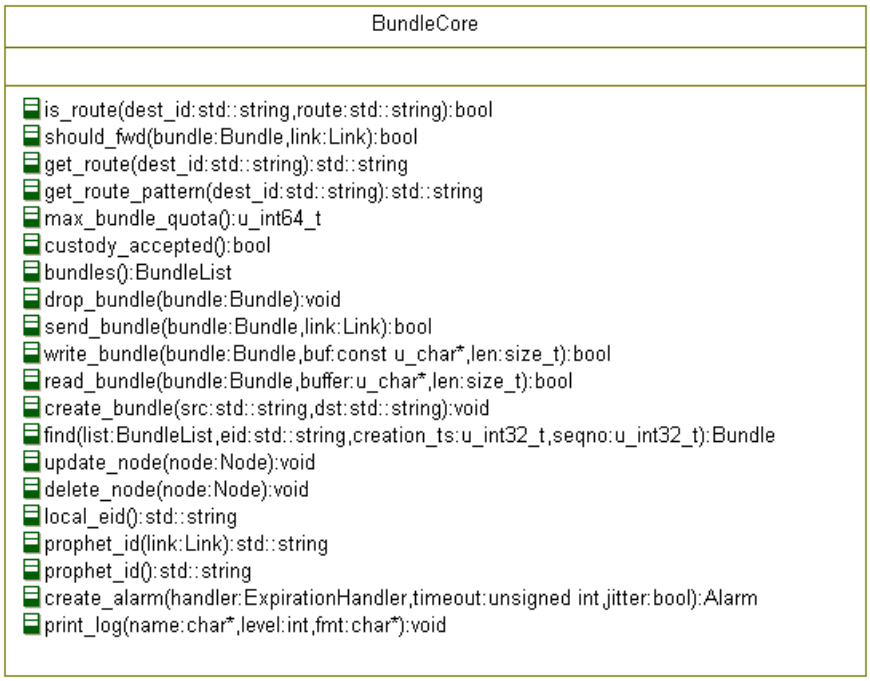
As of this writing, our DTNRI implementation is the only implementation of our Façade interface. To integrate Façade with DTNRI, our implementation passes events into PRoPHET via the ProphetRouter interface, which inherits from the DTNRI BundleRouter interface. The following table (Figure 13) matches the dtn::ProphetRouter event handler method with its matching prophet::Controller interface. Figure 14

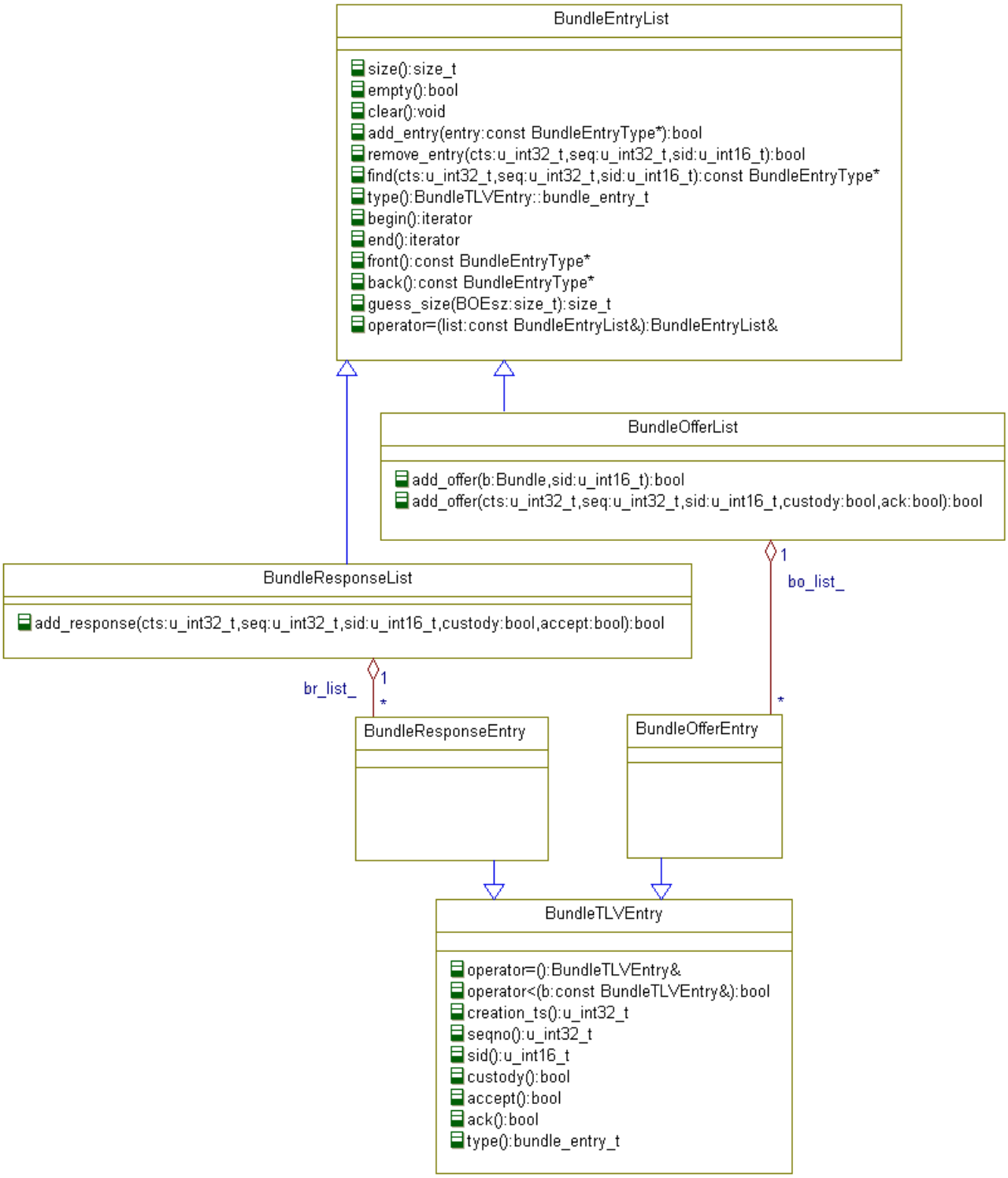
shows a graphical representation of the object hierarchy in our Façade implementation. (The Rhapsody UML document is available for download[27].)

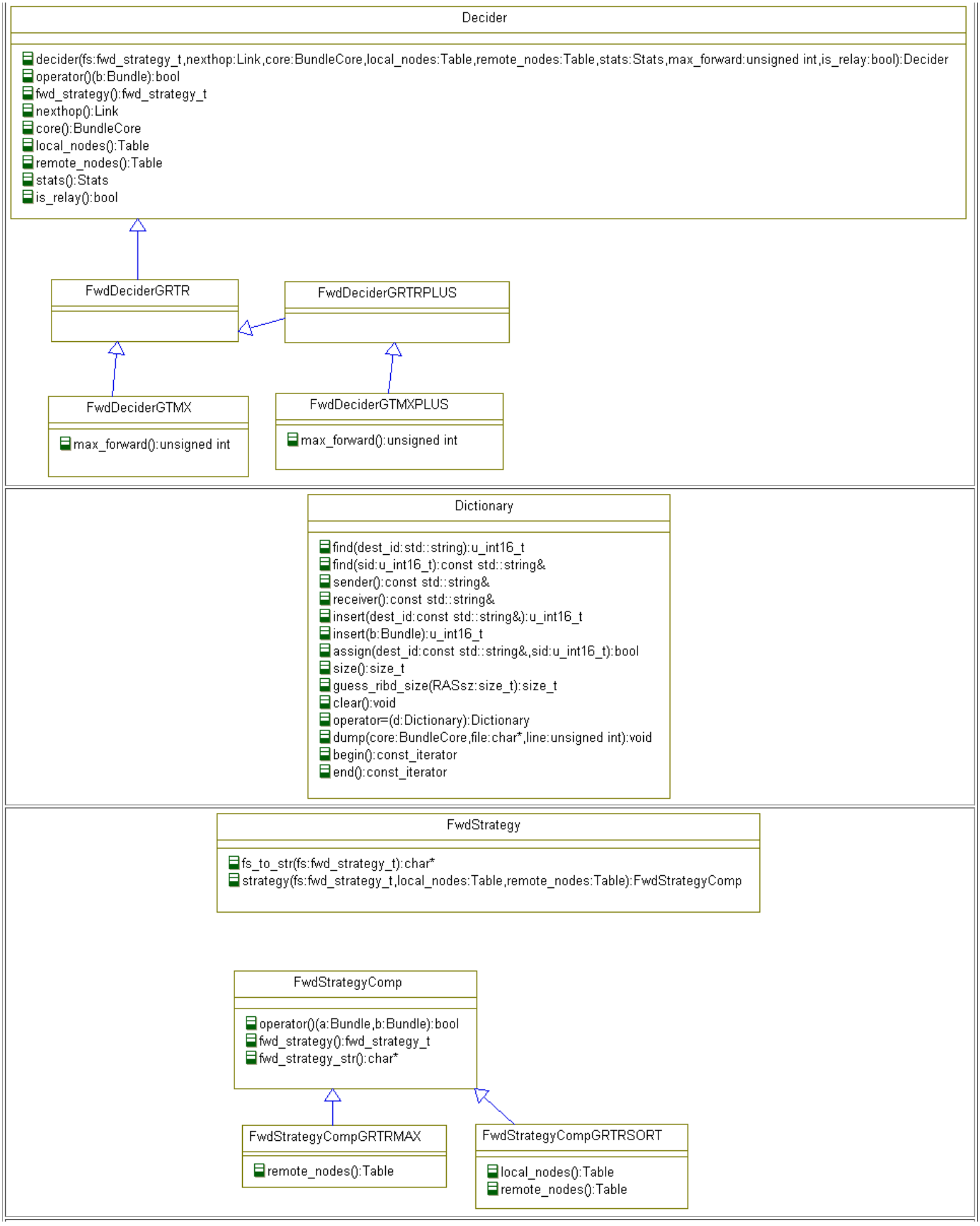
dtn::ProphetRouter method	prophet::Controller method	Description
handle_bundle_received	handle_bundle_received	Depending on the destination address, a Bundle can be intended for a Prophet control session (Encounter) or some application
handle_bundle_delivered	ack	When a Bundle is delivered to its final destination, a prophet ACK gets flooded out so that the Bundle can be dequeued across the network
handle_contact_up	new_neighbor	Begin a new Prophet control session (Encounter) to exchange routes and bundles with new peer
handle_contact_down	neighbor_gone	Give up on Prophet peer since connectivity is lost

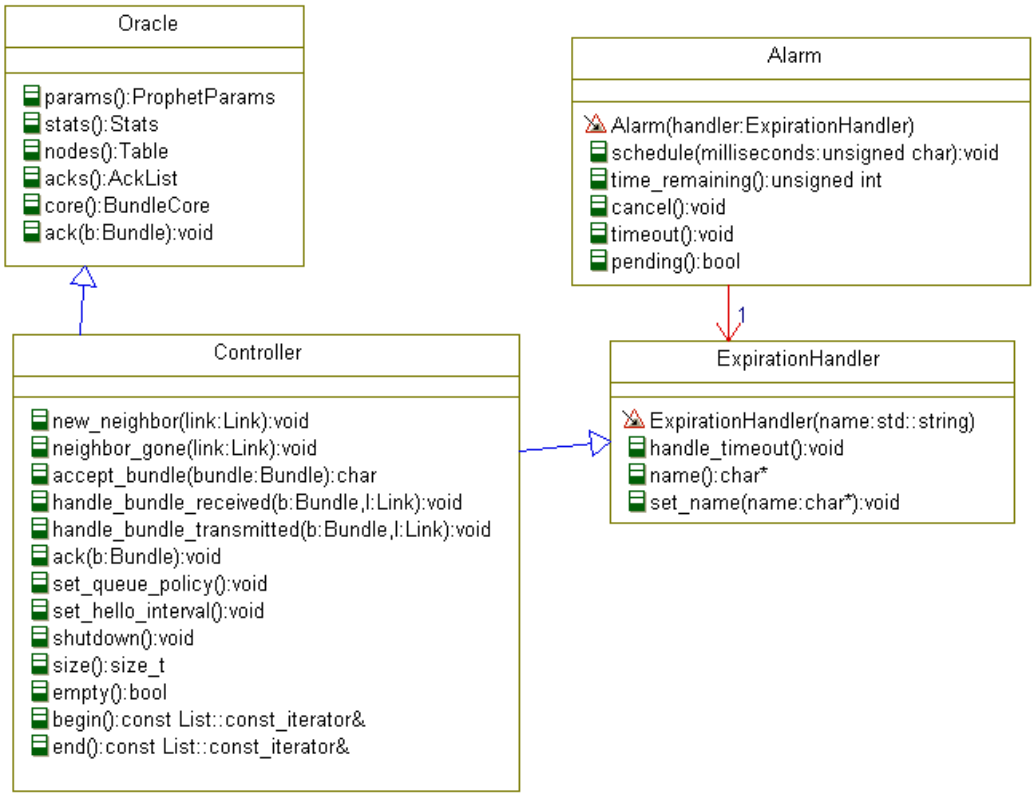
Figure 13. DTNRI and PRoPHET Façade integration

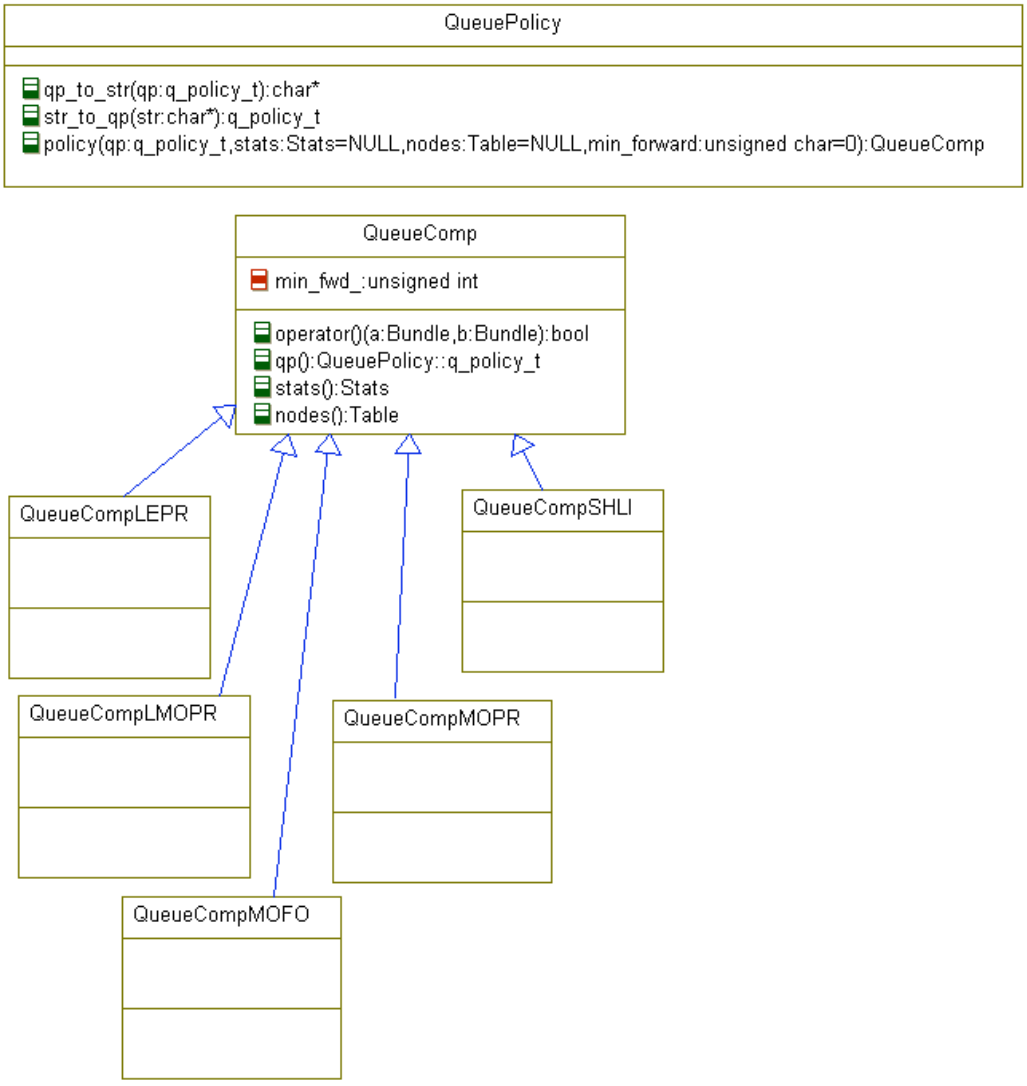


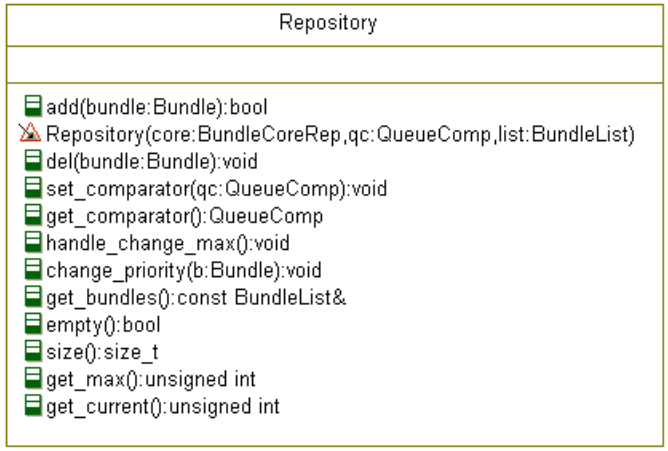

BundleCore
is_route(dest_id:std::string,route:std::string):bool
should_fwd(bundle:Bundle,link:Link):bool
get_route(dest_id:std::string):std::string
get_route_pattern(dest_id:std::string):std::string
max_bundle_quota():u_int64_t
custody_accepted():bool
bundles():BundleList
drop_bundle(bundle:Bundle):void
send_bundle(bundle:Bundle,link:Link):bool
write_bundle(bundle:Bundle,buf:const u_char*,len:size_t):bool
read_bundle(bundle:Bundle,buffer:u_char*,len:size_t):bool
create_bundle(src:std::string,dst:std::string):void
find(list:BundleList,eid:std::string,creation_ts:u_int32_t,seqno:u_int32_t):Bundle
update_node(node:Node):void
delete_node(node:Node):void
local_eid():std::string
prophet_id(link:Link):std::string
prophet_id():std::string
create_alarm(handler:ExpirationHandler,timeout:unsigned int,jitter:bool):Alarm
print_log(name:char*,level:int,fmt:char*):void

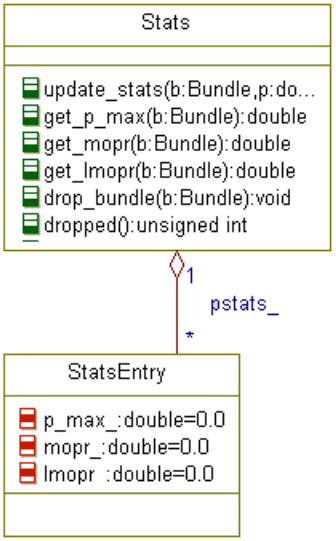

BundleEntryList
size():size_t
empty():bool
clear():void
add_entry(entry:const BundleEntryType*):bool
remove_entry(cts:u_int32_t,seq:u_int32_t,sid:u_int16_t):bool
find(cts:u_int32_t,seq:u_int32_t,sid:u_int16_t):const BundleEntryType*
type():BundleTLVEntry::bundle_entry_t
begin():iterator
end():iterator
front():const BundleEntryType*
back():const BundleEntryType*
guess_size(BOEsz:size_t):size_t
operator=(list:const BundleEntryList&):BundleEntryList&
BundleOfferList
add_offer(b:Bundle,sid:u_int16_t):bool
add_offer(cts:u_int32_t,seq:u_int32_t,sid:u_int16_t,custody:bool,ack:bool):bool
1
bo_list_
*
BundleResponseList
add_response(cts:u_int32_t,seq:u_int32_t,sid:u_int16_t,custody:bool,accept:bool):bool
br_list_
1
*
BundleResponseEntry
BundleOfferEntry
BundleTLVEntry
operator=():BundleTLVEntry&
operator<(b:const BundleTLVEntry&):bool
creation_ts():u_int32_t
seqno():u_int32_t
sid():u_int16_t
custody():bool
accept():bool
ack():bool
type():bundle_entry_t


Decider
decider(fs:fwd_strategy_t,nexthop:Link,core:BundleCore,local_nodes:Table,remote_nodes:Table,stats:Stats,max_forward:unsigned int,is_relay:bool):Decider
operator()(b:Bundle):bool
fwd_strategy():fwd_strategy_t
nexthop():Link
core():BundleCore
local_nodes():Table
remote_nodes():Table
stats():Stats
is_relay():bool
FwdDeciderGRTR
FwdDeciderGRTRPLUS
FwdDeciderGTMX
max_forward():unsigned int
FwdDeciderGTMXPLUS
max_forward():unsigned int
Dictionary
find(dest_id:std::string):u_int16_t
find(sid:u_int16_t):const std::string&
sender():const std::string&
receiver():const std::string&
insert(dest_id:const std::string&):u_int16_t
insert(b:Bundle):u_int16_t
assign(dest_id:const std::string&,sid:u_int16_t):bool
size():size_t
guess_ribd_size(RASsz:size_t):size_t
clear():void
operator=(d:Dictionary):Dictionary
dump(core:BundleCore,file:char*,line:unsigned int):void
begin():const_iterator
end():const_iterator
FwdStrategy
fs_to_str(fs:fwd_strategy_t):char*
strategy(fs:fwd_strategy_t,local_nodes:Table,remote_nodes:Table):FwdStrategyComp
FwdStrategyComp
operator()(a:Bundle,b:Bundle):bool
fwd_strategy():fwd_strategy_t
fwd_strategy_str():char*
FwdStrategyCompGRTRMAX
remote_nodes():Table
FwdStrategyCompGRTRSORT
local_nodes():Table
remote_nodes():Table


Oracle
params():ProphetParams
stats():Stats
nodes():Table
acks():AckList
core():BundleCore
ack(b:Bundle):void
Alarm
Alarm(handler:ExpirationHandler)
schedule(milliseconds:unsigned char):void
time_remaining():unsigned int
cancel():void
timeout():void
pending():bool
1
Controller
new_neighbor(link:Link):void
neighbor_gone(link:Link):void
accept_bundle(bundle:Bundle):char
handle_bundle_received(b:Bundle,l:Link):void
handle_bundle_transmitted(b:Bundle,l:Link):void
ack(b:Bundle):void
set_queue_policy():void
set_hello_interval():void
shutdown():void
size():size_t
empty():bool
begin():const List::const_iterator&
end():const List::const_iterator&
ExpirationHandler
ExpirationHandler(name:std::string)
handle_timeout():void
name():char*
set_name(name:char*):void


QueuePolicy
qp_to_str(qp:q_policy_t):char*
str_to_qp(str:char*):q_policy_t
policy(qp:q_policy_t,stats:Stats=NULL,nodes:Table=NULL,min_forward:unsigned char=0):QueueComp
QueueComp
min_fwd_:unsigned int
operator()(a:Bundle,b:Bundle):bool
qp():QueuePolicy::q_policy_t
stats():Stats
nodes():Table
QueueCompLEPR
QueueCompSHLI
QueueCompLMOPR
QueueCompMOPR
QueueCompMOFO


Repository
add(bundle:Bundle):bool
Repository(core:BundleCoreRep,qc:QueueComp,list:BundleList)
del(bundle:Bundle):void
set_comparator(qc:QueueComp):void
get_comparator():QueueComp
handle_change_max():void
change_priority(b:Bundle):void
get_bundles():const BundleList&
empty():bool
size():size_t
get_max():unsigned int
get_current():unsigned int


Stats
update_stats(b:Bundle,p:do...
get_p_max(b:Bundle):double
get_mopr(b:Bundle):double
get_lmopr(b:Bundle):double
drop_bundle(b:Bundle):void
dropped():unsigned int
1
pstats_
*
StatsEntry
p_max_:double=0.0
mopr_:double=0.0
lmopr_:double=0.0

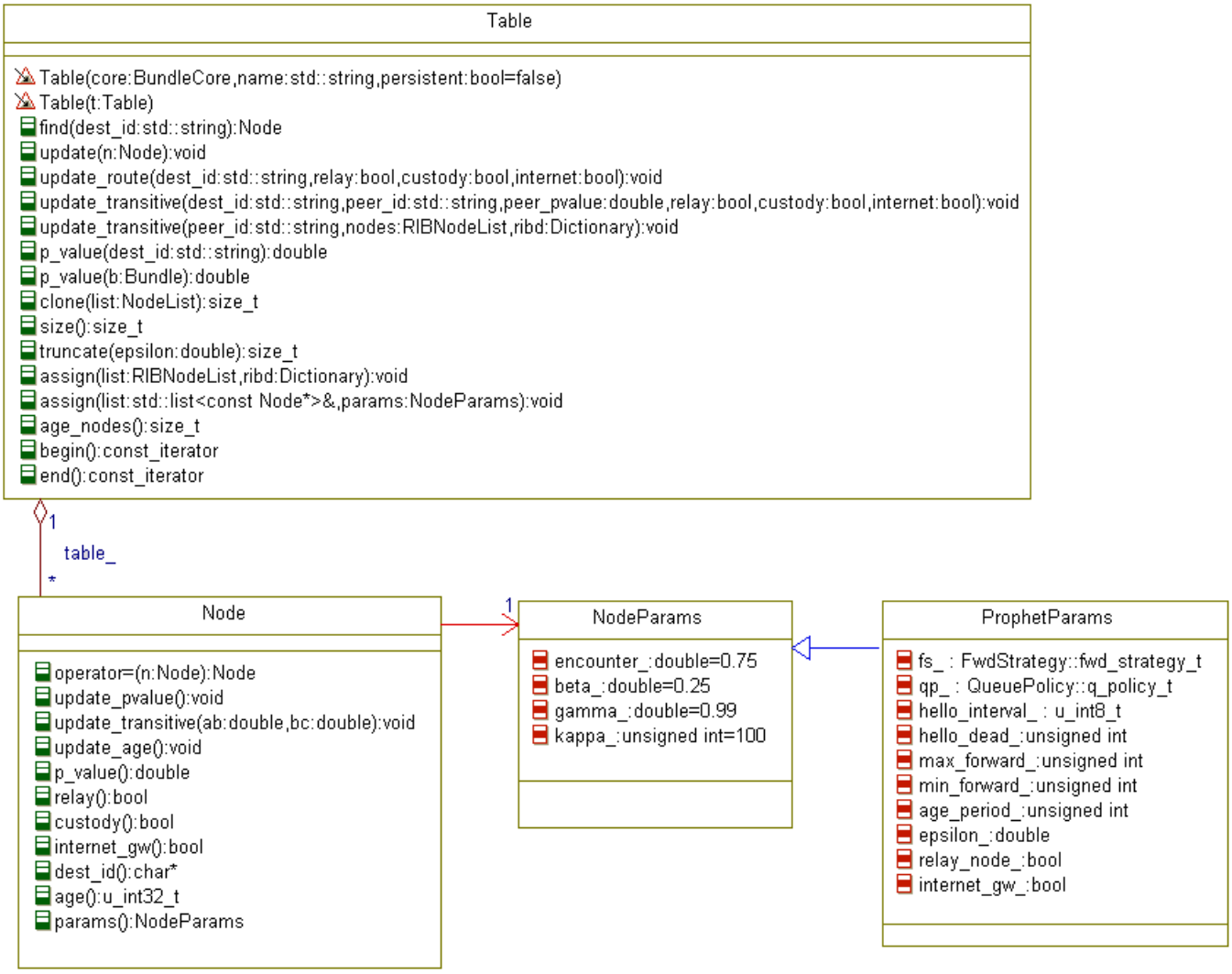


Table
Table(core:BundleCore,name:std::string,persistent:bool=false)
Table(t:Table)
find(dest_id:std::string):Node
update(n:Node):void
update_route(dest_id:std::string,relay:bool,custody:bool,internet:bool):void
update_transitive(dest_id:std::string,peer_id:std::string,peer_pvalue:double,relay:bool,custody:bool,internet:bool):void
update_transitive(peer_id:std::string,nodes:RIBNodeList,ribd:Dictionary):void
p_value(dest_id:std::string):double
p_value(b:Bundle):double
clone(list:NodeList):size_t
size():size_t
truncate(epsilon:double):size_t
assign(list:RIBNodeList,ribd:Dictionary):void
assign(list:std::list<const Node*>&,params:NodeParams):void
age_nodes():size_t
begin():const_iterator
end():const_iterator
1
table_
*
Node
operator=(n:Node):Node
update_pvalue():void
update_transitive(ab:double,bc:double):void
update_age():void
p_value():double
relay():bool
custody():bool
internet_gw():bool
dest_id():char*
age():u_int32_t
params():NodeParams
1
NodeParams
encounter_:double=0.75
beta_:double=0.25
gamma_:double=0.99
kappa_:unsigned int=100
ProphetParams
fs_ : FwdStrategy::fwd_strategy_t
qp_ : QueuePolicy::q_policy_t
hello_interval_ : u_int8_t
hello_dead_:unsigned int
max_forward_:unsigned int
min_forward_:unsigned int
age_period_:unsigned int
epsilon_:double
relay_node_:bool
internet_gw_:bool

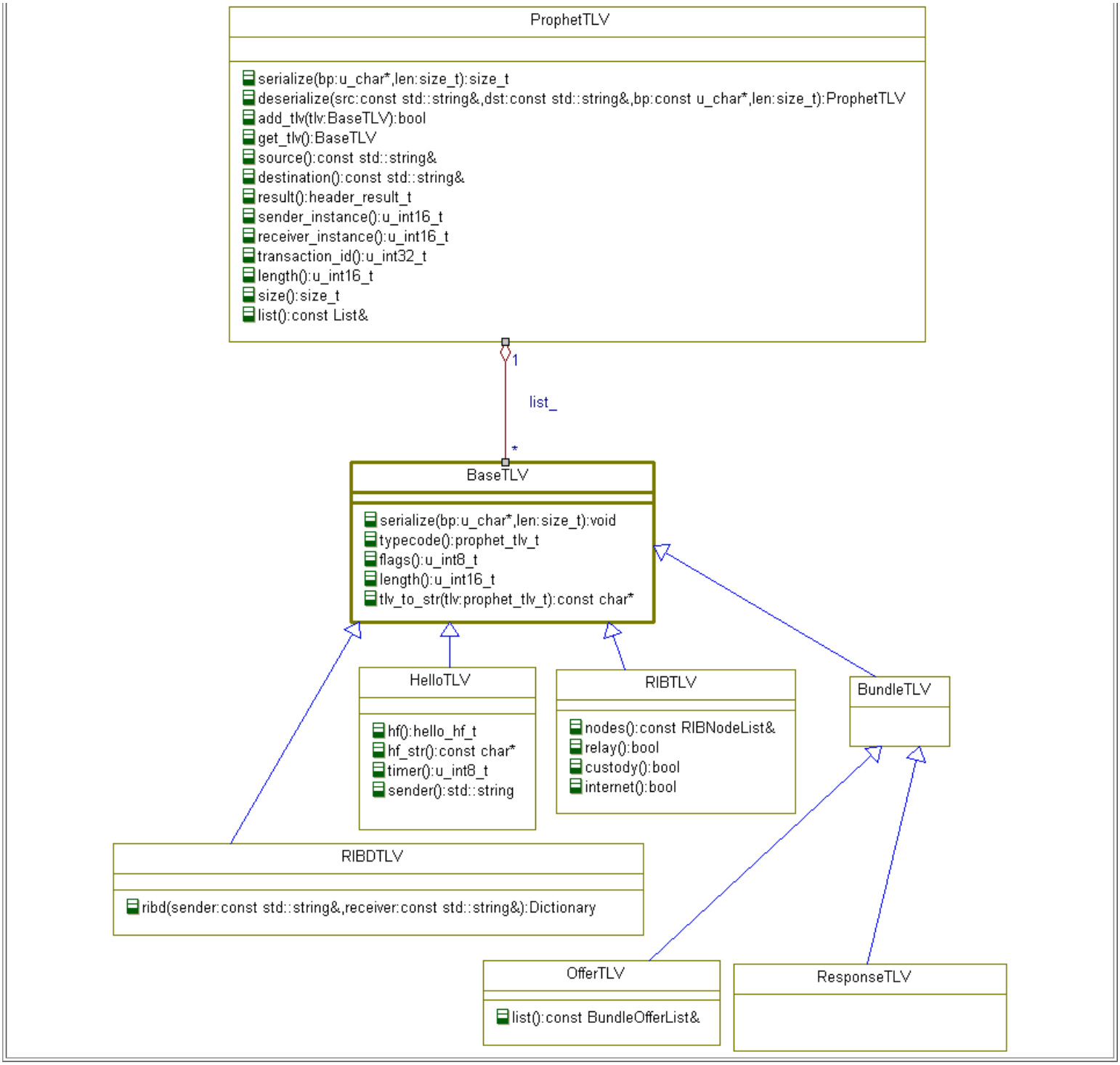


Figure 14. PRoPHET Façade Object Diagrams

In addition to the prophet::Controller interface, ProphetRouter also interacts with the DTN host's BundleCore implementation, in this case dtn::ProphetBundleCore. As Links and Bundles come through the DTN bundling engine (i.e., the DTN namespace), ProphetRouter listens for the relevant events and creates or destroys the matching Façade elements in the prophet namespace. Within the Façade, all interfaces are described in terms of prophet::Link and prophet::Bundle.

To get a better idea of how events flow through DTNRI and translate into Façade actions, consider the following scenario. Hosts Alice and Charlie are DTN relays with neighbor discovery enabled and running the PRoPHET router. A user on Alice sends a message to a user on Charlie. The Bundle is submitted to the DTN daemon on host Alice using the DTN API; this event generates the BundleReceivedEvent message. ProphetRouter receives the event and adds a Façade handle for the new Bundle into the BundleCore. At this point, any amount of time may pass; the Bundle is committed to persistent storage and will remain there until it expires. Some time before expiration, host Charlie moves into range of the discovery process on host Alice. The discovery mechanism on Alice, whether by polling or beacons, discovers Charlie's presence and initiates a new Opportunistic Link to Charlie. Next, the discovery mechanism on Alice requests that the newly created Link be made available. ProphetRouter steps in and attempts to open the Link. The convergence layer carries out the attempt to open the Link by initiating Convergence Layer protocol over the Link. A successful Link open results in a ContactUpEvent. ProphetRouter implements a handler for the ContactUpEvent, adding a wrapper to the Façade for the associated Link and calling prophet::Controller::new_neighbor. Before we

conclude the scenario between Alice and Charlie, we stop to explain our implementation of the PRoPHET protocol in greater detail.

The Controller method, new_neighbor, searches through its list of current Encounters to determine whether a session to the peer is already open. If none exists, a new Encounter is created to handle protocol messages exchanged with the discovered neighbor. All PRoPHET protocol messages are passed via Bundles in the data plane. In the ProphetRouter method, handle_bundle_received, each Bundle is first wrapped in a Façade wrapper, then inspected to determine whether it is destined for an Encounter. If so, it is passed to the Encounter that is associated with the Link over which the Bundle arrived. Otherwise, the Bundle waits in storage to be relayed to peers or delivered to local applications as appropriate. The Encounter hands off the Bundle to ProphetTLV, which deserializes the aggregated TLVs used by the protocol. For each BaseTLV-derived message that is extracted from the ProphetTLV, the state machine logic determines the response. One weakness in the PRoPHET protocol is its lack of error handling. If messages are received out of order, or if a timeout occurs while waiting for an expected response, the protocol only dictates the local state transition without specifying how to signal the peer of the error condition.

The first phase of the protocol is the Hello exchange, in which hello_interval is negotiated and protocol version is confirmed (see Appendix B). Once the session is established, each participant alternates through complementary phases of the Information Exchange, both as Initiator and as Listener (see Appendix B). The protocol specifies that the first peer to send the Hello SYN message is the implicit position 0 of the index, and the other peer becomes the implicit position 1. In practice, this tie-breaker is not effective due to the way that neighbor discovery works (at least in the case of the DTNRI), such that the discovery event is simultaneous for both parties. To overcome this, a new tie-breaker is used by this implementation; whichever peer's DTN name is lexographically sorted lower becomes the SYN sender. This SYN sender designation also has significance beyond implicit indeces -- the SYN sender is the first to assume Initiator. The other peer takes the Listener phase first. After completing the first half of the Information Exchange, the peers swap roles to complete the other half. The peering session is broken whenever link state is lost, or the peer becomes unresponsive (i.e., does not respond after HELLO_INTERVAL elapses HELLO_DEAD times).

To conclude our scenario, we return to the situation where a bundle destined for Charlie is spooled on Alice, awaiting an opportunity for relay. Fortunately, Alice and Charlie directly encounter each other, and the resulting ContactUpEvent causes the subsequent protocol peering, which is managed by the Encounter object. Although the Information Exchange phase would eventually offer to forward the bundle from Alice to Charlie, the PRoPHET Internet Draft[9] explicitly allows fast-path forwarding to short-circuit the routing protocol in the event of a direct encounter. Because Charlie is the destination of the bundle, Alice immediately forwards it across the opportunistic link as soon as it becomes available.

6 Appendix A. Bluetooth

This appendix presents a more detailed overview of the Bluetooth specification. The full specification is available from the Bluetooth Special Interest Group[5].

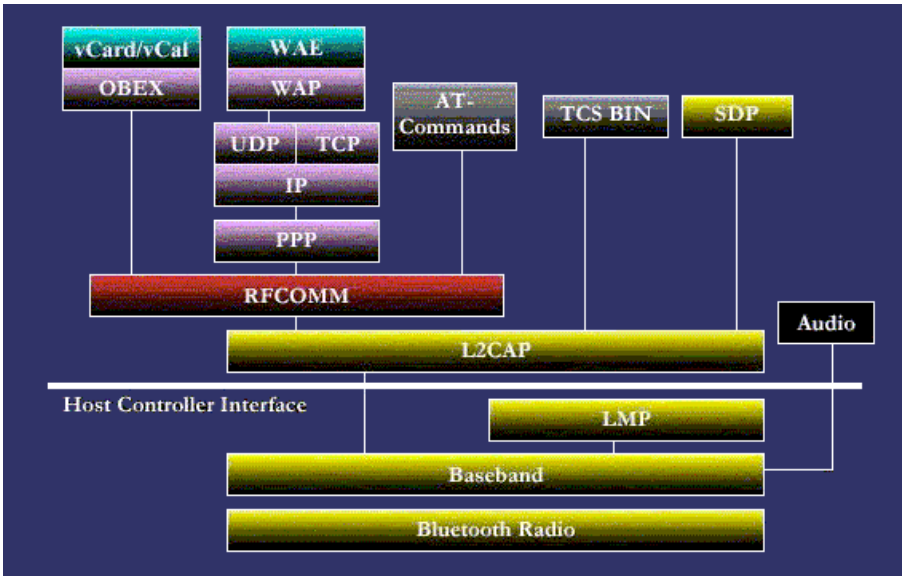


Figure 15. Bluetooth Stack[28]

Information is passed over the Bluetooth radio as packets in time-division-multiplexed (TDM) slots. Packets are managed by L2CAP (logical link control and adaption protocol). To avoid interference with other wireless technologies, Bluetooth employs a frequency hopping technique across a known range[29].

This frequency hopping adds complexity to Bluetooth's discovery mechanism, Inquiry. Inquiry moves rapidly through the known set of frequencies at a rate of 625 microseconds per frequency to beacon its timing information (clock offset to the beginning of its frequency hopping schedule). The passive side of Inquiry, any device configured as "discoverable" constantly moves through the range of known frequencies at the rate of 1.28 seconds per frequency. Eight such cycles, or 10.24 seconds, is the recommended minimum inquiry[30]. The technology is designed for a user to discover all her devices in local range. This becomes a liability in creating an interpersonal network. Further testing and field deployment have yet to determine whether this latency will adversely affect our neighbor discovery process.

Bluetooth defines the RFCOMM profile, a serial emulation similar to RS232 but implemented over a packet stream, like TCP over IP. It is a byte stream with hardware flow control (out of band) and software flow control (in-band, such as XON/XOFF). Automatic repeat request (ARQ) is similar to TCP's ACK for reliability accounting. Similar to TCP's window size, RFCOMM tracks windows using a credit scheme negotiated when

L2CAP initiates packet stream. Once the sender consumes the last credit, no further packets are sent until more credits are received. A unique credit accounting is performed per data link connection, per direction. Additionally, flow control is also performed by the software component of the Bluetooth stack[30]

7 Appendix B. PRoPHET

The following details are condensed from the PRoPHET Internet Draft[9] and presented here for the reader's convenience.

7.1 Replication Optimizations

PRoPHET offers the following forwarding strategies. Each strategy answers the question, per bundle, whether to forward to the encountered peer. Since no single forwarding strategy will work well for every scenario, a variety of strategies is offered. The default strategy is GRTR: relay this bundle to this peer only if this peer has greater likelihood of delivering this bundle. Another strategy is GTMX: observe GRTR, but do not forward this bundle more than a user-specified parameter, NF_max. The strategy GRTR+ is also like GRTR, but further restricted so that it only forwards the bundle to the peer if the peer's delivery metric is greater than the previously highest encountered metric (P_max). Another strategy, GTMX+, combines GRTR+ with GTMX, so that the bundle is only forwarded if the remote's delivery metric is greater than P_max, and the bundle has been forwarded less than NF_max times. The strategy GRTRSort is like GRTR, but forwards bundles in order of descending order of the difference between the remote and local delivery metric; this means that the first to be forwarded has the greatest improvement in delivery probability. The strategy GRTRMax also sorts, but in descending order of the remote's delivery metric, so that the bundle with the greatest absolute probability of delivery gets forwarded first.

To optimize the use of storage resources, PRoPHET's queuing policy describes which bundles are evicted first, possibly before notice of successful delivery. (N.B.: bundles accepted as custody transfer MUST NOT be dropped before the TTL expires.) One such policy, FIFO, is a simple first-in-first-out policy; however, this takes no account of PRoPHET's knowledge domain. Using DTN statistics per bundle, the MOFO policy dictates that the most forwarded bundle will be the first evicted. Combining DTN statistics with PRoPHET's delivery probabilities, the MOPR policy evicts the most favorably forwarded bundle first, using a bounded exponential growth algorithm. Similarly, the Linear MOPR policy is like MOPR, except that the algorithm uses linear increase. The SHLI policy evicts bundles with shortest remaining TTL. The LEPR policy evicts bundles with the lowest delivery probability first, but only if the bundle has been forwarded more than MF times (where MF is a user-configurable parameter).

7.2 Interface Requirements

The PRoPHET Internet Draft gives some guidance on how to interface with other layers. For example, some basic services are expected from the Bundle Agent (the DTN host environment). Also, some prerequisites are expected from the underlying link technology. In the following lists, PRoPHET is considered the routing agent.

The Routing Agent expects the following primitives from the Bundle Agent. The first is "Get Bundle List": the Bundle Agent should enumerate the current list of bundles in its inventory. Next, "Send Bundle" allows the Routing Agent to cause the Bundle Agent to transmit a given bundle to a given peer. When the Routing Agent requests "Accept Bundle," the Bundle Agent commits the given bundle to persistent storage. The Routing Agent will signal "Bundle Delivered" to the Bundle Agent to indicate that the given bundle has reached its final destination. The Routing Agent signals "Drop Bundle" to the Bundle Agent to drop a given bundle from persistent storage.

By subscribing to the above-mentioned primitives from the DTN host, PRoPHET can be adapted to any number of underlying link-layer technologies. PRoPHET specifies that certain primitive operations and signals are necessary from the link layer. PRoPHET must be alerted whenever neighbors pass in and out of range, as follows. The link must deliver the "New Neighbor" signal to indicate that a peer has come into range. The link must deliver the "Neighbor Gone" signal to indicate that a peer has passed out of range. With each signal, the link must deliver "Local Address" as a unique identifier to describe underlying link interface is affected.

7.3 PRoPHET Protocol

Participants in a PRoPHET network learn routes by encountering peers and exchanging routing information. The protocol used during this encounter consists of two phases: the Hello Procedure and the Information Exchange. PRoPHET responds to the New Neighbor signal by initiating the Hello Procedure, similar to TCP's three-way handshake in establishing a peering session. Once a session is established, peers enter Information Exchange to exchange routing tables and negotiate which bundles to exchange.

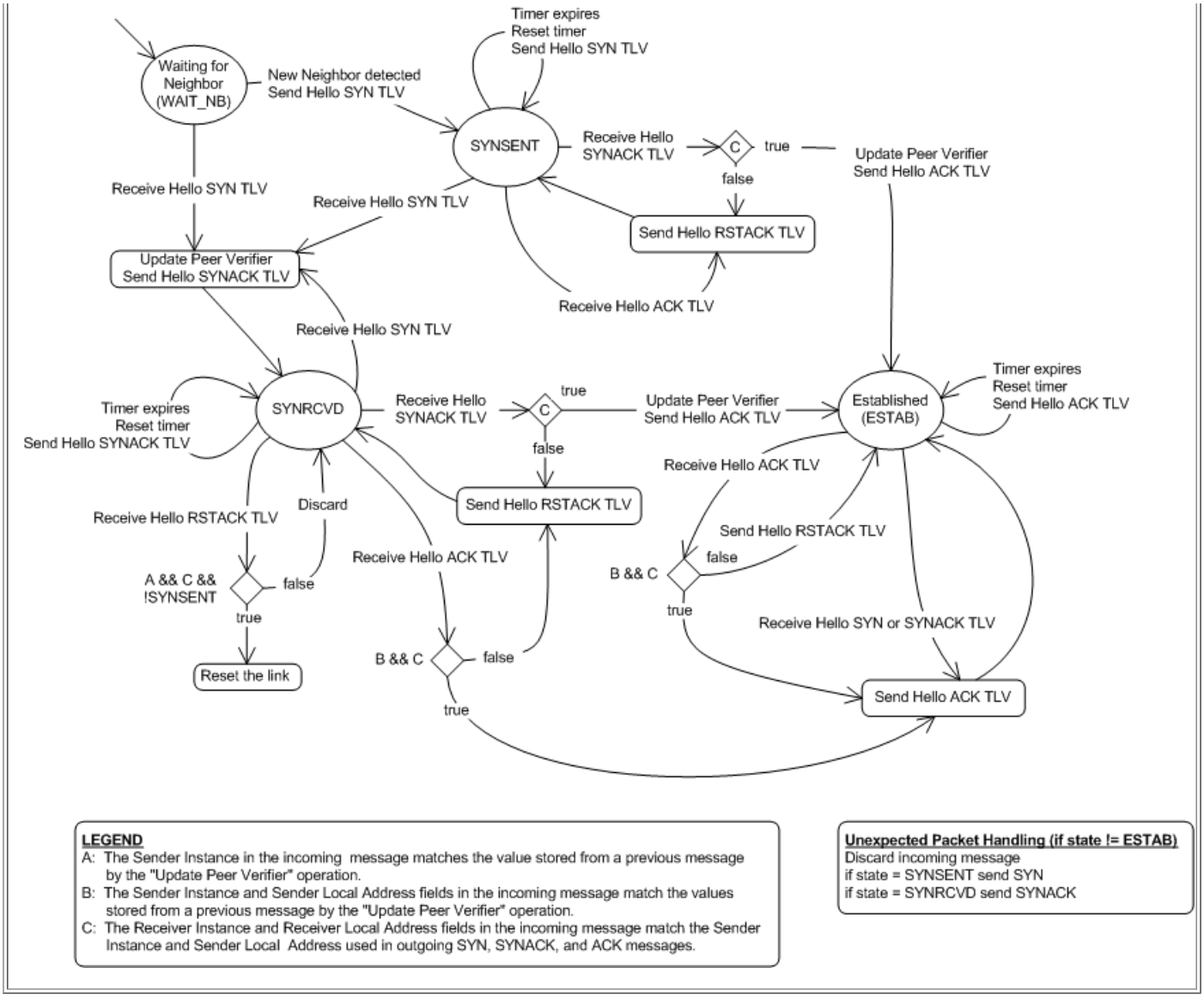


Figure 16. PRoPHET Hello Procedure.

The Hello procedure (see Figure 16) allows PRoPHET nodes to negotiate protocol parameters, such as protocol version and message exchange timeout. The first peer to initiate contact, by SYN message, becomes the SYN sender. This SYN sender designation carries significance in later parts of the protocol. SYNACK is sent in response to SYN, then ACK is sent in response to SYNACK. Once the session is established, the next phase of protocol is the Information Exchange.

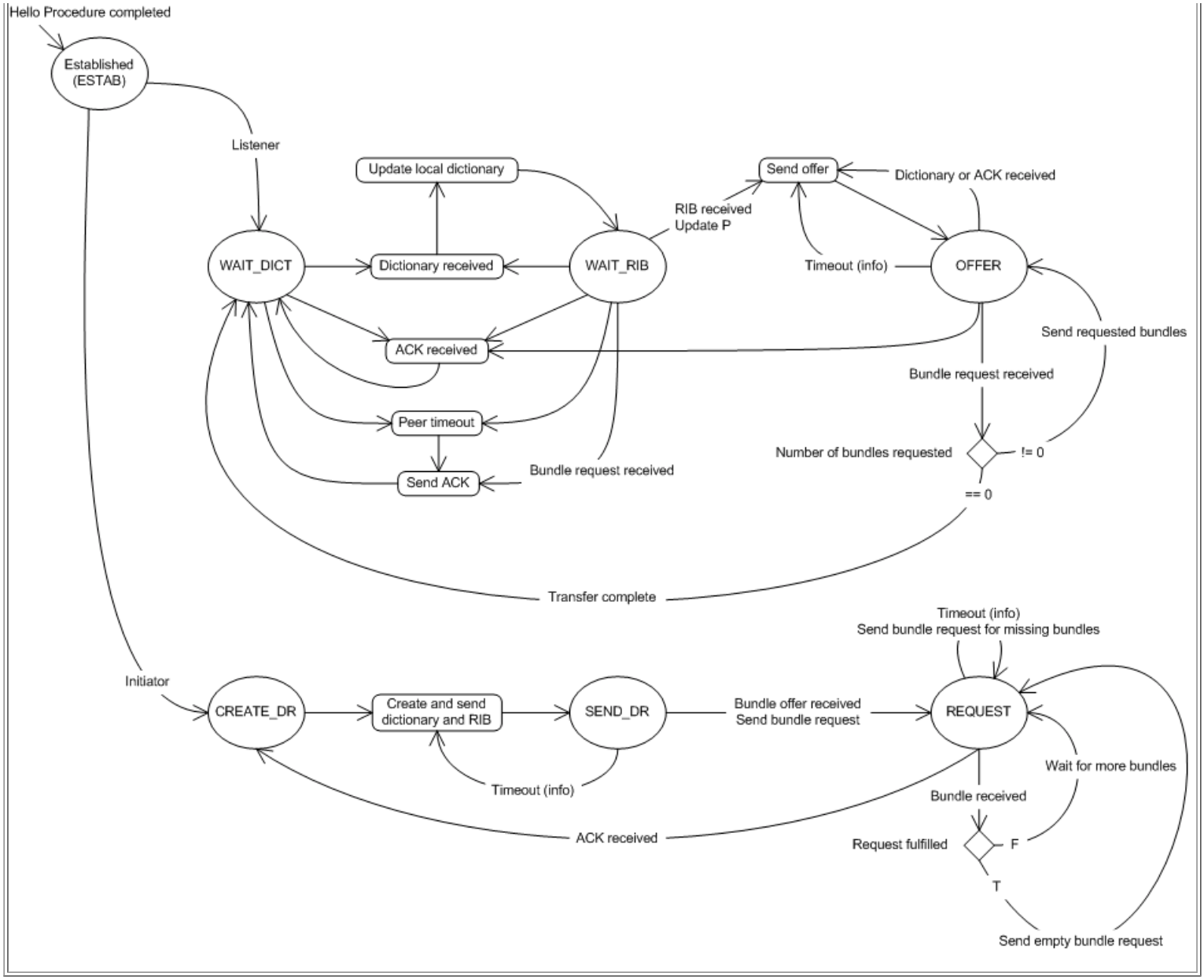


Figure 17. PRoPHET Hello Procedure.

In the Information exchange (see Figure 17), each peer has two phases, Initiator and Listener. The peer with the SYN sender designation first becomes Initiator, then alternates to Listener. Conversely, the SYN receiver first becomes Listener, then alternates to Initiator. The Initiator begins the half-phase by creating and sending a Dictionary. A Dictionary enumerates all the destination addresses of the Bundles in the Initiator's inventory, creating an ephemeral index to be used for the duration of this half-phase. Using the Dictionary's index, the Initiator next lists all delivery probabilities for known destinations: a routing information base. Once the Listener receives the Dictionary and RIB, the Listener prepares a Bundle Offer based on its local forwarding strategy. The Initiator receives the Offer and decides which Bundles to request for relay. The Listener sends each requested Bundle until Initiator sends an empty Request, signalling the end of the half-phase. At this point, the peers switch roles and proceed through the other half-phase to complete route learning and bundle exchange. During the protocol, if link state is lost, the peering session is broken ("Neighbor Gone" signal), and the Hello Phase must be repeated to regain a session. Otherwise, if link state remains unchanged, Information Exchange is periodically repeated until peering session is broken.

8 Appendix C. Example Configuration.

This section presents DTN configuration and scripts we use to demonstrate the routing of a bundle across a DTN then through an SMTP gateway on the edge of the DTN. Our original demonstration is designed for two stationary DTN relays and one mobile DTN relay. The procedure we use for our demonstration is detailed in Chapter 4.

To support the use of Perl modules referred to in the following script listings, log on to each node and perform the following configuration steps.

1. Change to your DTN source directory.
2. cd applib
3. make perlapi
4. sudo make perlapi_install

5. sudo cpan
6. install Mail::Sendmail

On each of the three relays, create a DTN config file using the listing shown in Figure 18.

```
log /dtnd info "dtnd parsing configuration..."
console set addr 127.0.0.1
console set port 5050
storage set type berkeleydb
storage set payloaddir bundles
storage set dbdir         db
route set type prophet
route local_eid "dtn://[info hostname].dtn"
prophet set hello_dead 6
interface add bt0 bt
discovery add btdisc0
discovery announce bt0 btdisc0 bt interval=9
log /daemon info "dtnd configuration parsing complete"
```

Figure 18. Example configuration.

Create a startup script using the listing in Figure 19. Change the source directory to match the root of your DTN source tree. Change the config directory to match the location of the config file created using Figure 18. Start up DTN on each of the demo nodes.

```
#!/bin/bash
cd ~/dtn/src/DTN2
rm dtn.log
./daemon/dtnd -dtc ~/dtn/prod/demo.conf -o ./dtn.log
sleep 5
telnet localhost 5050
```

Figure 19. Example DTN startup script.

Create a perl script using the listing in Figure 20 to create and send email through a DTN tunnel (embed an email in a DTN Bundle).

```
#!/usr/bin/perl
#
#     Copyright 2007 Baylor University
#
#     Licensed under the Apache License, Version 2.0 (the "License");
#     you may not use this file except in compliance with the License.
#     You may obtain a copy of the License at
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
#     Unless required by applicable law or agreed to in writing, software
#     distributed under the License is distributed on an "AS IS" BASIS,
#     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#     See the License for the specific language governing permissions and
#     limitations under the License.
#

use dtnapi;
use strict;

my $maxInputLines = 100;   # no more than 100 lines per input
my $bundleLifetime = 3600; # one hour
my ($dest_set,$gwy_set,$body_set);

# Config for email
my %smtp = (
    # originating email address
    'source' => 'Jeff Wilson <Jeff_Wilson@baylor.edu>',
    'subj'   => "Message from DTN",
);

# Config for DTN
my %dtn  = (
    'dest'    => 'dtn://%s.dtn/smtp', # template
    'domain'  => 'baylor.edu',        # default dtn domain
    'timeout' => $bundleLifetime,     # one hour lifetime
);

# temp debug setup
#$dest_set++; $smtp{'dest'} = 'Jeff Wilson <jeff_wilson@baylor.edu>';
#$gwy_set++;  $dtn{'dest'}  = 'dtn://alice.baylor.edu.dtn/smtp';
#$body_set++; $smtp{'msg'}  =

#########################
#    SUBs               #
#########################

sub prompt_for_info {
    my ($prompt,$numlines,$confirm) = @_;
    unless (defined $prompt) { return undef; }
    unless (defined $numlines) { $numlines = 1; }
    unless (defined $confirm) { $confirm="Is \"%s\" correct? "; }
```

```
    if ($numlines == 0 || $numlines > $maxInputLines)
    {
        # impose a limit
        $numlines = $maxInputLines;
    }
    my $ok;
    my $line;
    do {
        print $prompt;
        my @lines = ();
        while (<>) {
            chomp;
            push @lines,$_;
            last unless (--$numlines);
        }
        $line = join("\n",@lines);
        printf $confirm,$line;
        $_ = <>;
        if ($_ =~ m/[^Yy]/) { $ok++; }
    } while (! $ok);
    return $line;
}

########################
#    MAIN              #
########################

# Read email address of destination
unless ($dest_set) {
$smtp{'dest'} = &prompt_for_info("\nEnter the destination email address:\n",1);
}

# Read hostname of DTN to SMTP gateway
unless ($gwy_set) {
my $gwy = &prompt_for_info("\nEnter the DTN to SMTP gateway:\n",1);
unless ($gwy =~ m/\./) { $gwy .= ".".$dtn{'domain'}; }
$dtn{'dest'} = sprintf($dtn{'dest'},$gwy);
}

# Read in message body
unless ($body_set) {
$smtp{'msg'} = &prompt_for_info("\nType message, end with <ctrl-d>\n",0,
                                "\nIs above message OK to send?\n");
}

# Encode a Bundle payload for the SMTP gateway to use
# to send an email to the specified destination
$dtn{'payload'} = "From: $smtp{'source'}\n\r\n".
                  "To: $smtp{'dest'}\n\r\n".
                  "Subject: $smtp{'subj'}\n\r\n".
                  "Body: $smtp{'msg'}\n\r\n";

# try to contact local DTN
print "Attempting to open RPC to local DTN\n";
my $dtn_h;
eval { $dtn_h = dtnapi::dtn_open };
if ($@) {
    print "Error opening handle to DTN\n";
    exit;
}

# Build up endpoint ID for this originating process
$dtn{'src'} = dtnapi::dtn_build_local_eid($dtn_h, "mailapp");

# Using DTN config, put together a Bundle and ship'er off
my $id = dtnapi::dtn_send(
            $dtn_h,                     # handle
            $dtn{'src'},                # source
            $dtn{'dest'},               # destination
            'dtn:none',                 # replyto
            $dtnapi::COS_NORMAL,        # priority
            0,                          # dopts
            $dtn{'timeout'},            # expiration
            $dtnapi::DTN_PAYLOAD_MEM,   # payload_location
            $dtn{'payload'}
        );

# clean up SWIG memory
$id->DESTROY;

# close RPC
print "Closing RPC to local DTN\n";
dtnapi::dtn_close($dtn_h);
```

Figure 20. Perl script used to create and embed email in DTN bundle.

Receive the Bundle bearing the tunneled email and forward it to an SMTP gateway using a perl script created from the listing shown in Figure 21.

```
#!/usr/bin/perl
#
```

```
#     Copyright 2007 Baylor University
#
#     Licensed under the Apache License, Version 2.0 (the "License");
#     you may not use this file except in compliance with the License.
#     You may obtain a copy of the License at
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
#     Unless required by applicable law or agreed to in writing, software
#     distributed under the License is distributed on an "AS IS" BASIS,
#     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#     See the License for the specific language governing permissions and
#     limitations under the License.
#

use Mail::Sendmail;
use dtnapi;
use strict;

############################
#           CONFIG         #
############################

my $reg_timeout = 3600; # one hour
my $smtp_relay = "fs-exchange1.baylor.edu";

############################
#           MAIN           #
############################

$SIG{INT} = \&CatchSigInt;

my $dtn_h;

print "Attempting to open RPC to local DTN\n";
eval { $dtn_h = dtnapi::dtn_open; }; warn $@ if $@;
unless (defined $dtn_h || $dtn_h)
{
    die "Unable to open handle to DTN - fatal error, quitting\n";
}

my $mailapp = dtnapi::dtn_build_local_eid($dtn_h, "smtp");

print "Registering \"$mailapp\" with DTN\n";
my $regid = dtnapi::dtn_find_registration($dtn_h, $mailapp);

if ($regid != -1) {
    # found existing registration, so call bind()
    dtnapi::dtn_bind($dtn_h, $regid);
} else {
    # otherwise create new registration
    $regid = dtnapi::dtn_register(
                        $dtn_h,                # handle
                        $mailapp,              # endpoint
                        $dtnapi::DTN_REG_DROP, # action
                        $reg_timeout,          # expiration
                        0,                     # init_passive
                        ""                     # script
            );
}

print "Listening for incoming Bundles, press <ctrl-c> to quit\n";
while (1) {

    # block until bundle arrives
    my $bundle = dtnapi::dtn_recv(
                $dtn_h,                  # handle
                $dtnapi::DTN_PAYLOAD_MEM, # copy from memory
                -1                       # infinite wait
                );

    # silent error
    next unless (defined $bundle && $bundle);

    print "Received bundle from ".$bundle->{source}."\n";

    my @arr = split /\n\r\n/, $bundle->{payload};

    my ($to,$from,$subj,$body);
    for my $i (@arr) {
        if ($i =~ m/^From\: (.*)$/) {
            $from = $1;
        }
        elsif ($i =~ m/^To\: (.*)$/) {
            $to = $1;
        }
        elsif ($i =~ m/^Subject\: (.*)$/) {
            $subj = $1;
        }
        elsif ($i =~ m/^Body\: /) {
            $body = substr($i,length("Body: "));
        } else { next; }
    }
    print "
```

```
Payload received:
[$bundle->{payload}]
[To:] $to
[From:] $from
[Subject:] $subj
[Body:] ".substr($body,0,50)." ...
";

    if ($to && $from && $subj && $body ) {
        &forward_mail($to,$from,$subj,$body);
    }

    # dispose of bundle, now that everything's in local memory
    $bundle->DESTROY;
}

##########################
#           SUB's        #
##########################

sub CatchSigInt {
    print "Closing RPC handle to DTN\n";
    dtnapi::dtn_close($dtn_h);
    exit;
}

sub forward_mail {
    my ($to,$from,$subj,$body) = @_;

    my %mail = (
        To => $to,
        From => $from,
        Subject => $subj,
        Body => $body,
        smtp => $smtp_relay,
        );

    if (sendmail %mail ) {
        my $len = length $body;
        print "Mail sent to $to OK, $len byte payload\n";
    } else {
        print "Error: $Mail::Sendmail::error \n";
    }
}
```

Figure 21. Perl script used to receive DTN bundle and forward embedded email.

9 References

1. ↑ 1.0 1.1 1.2 1.3 1.4 1.5 1.6 Fall, Kevin. "A Delay-Tolerant Network Architecture for Challenged Internets." *SIGCOMM 2003*. http://cs.baylor.edu/~wilsonj/research/kfall-dtn-arch.pdf
2. ↑ 2.0 2.1 Baker, F. "An outsider's view of MANET." http://w3.antd.nist.gov/wctg/manet/draft-baker-manet-review-01.txt 2007
3. ↑ Delay Tolerant Networking Research Group. http://www.dtnrg.org/ 2007
4. ↑ 4.0 4.1 4.2 4.3 4.4 4.5 Delay Tolerant Networking Reference Implementation. http://www.dtnrg.org/wiki/Code 2007
5. ↑ 5.0 5.1 5.2 Bluetooth Special Interest Group. http://bluetooth.com/Bluetooth/SIG/ 2007
6. ↑ 6.0 6.1 6.2 BlueZ, Official Linux Bluetooth protocol stack. http://www.bluez.org 2007
7. ↑ C. Perkins, E. Royer. "Ad hoc On-Demand Distance Vector Routing," *IEEE Workshop on Mobile Computing Systems and Applications,* 1999 http://cs.baylor.edu/~wilson/research/aodv.pdf
8. ↑ V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss. *Delay-Tolerant Networking Architecture.* http://www.ietf.org/rfc/rfc4838.txt 2007
9. ↑ 9.0 9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 A. Lindgren, A. Doria. "Probabilistic Routing Protocol for Intermittently Connected Networks." DTN Research Group, Internet Draft, March 2006. http://www.dtnrg.org/docs/specs/draft-lindgren-dtnrg-prophet-02.txt
10. ↑ W. Zhao, M. Ammar. "Message Ferrying: Proactive Routing in Highly-Partitioned Wireless Ad Hoc Networks." *Proceedings of the IEEE Workshop on Future Trends in Distributed Computing Systems*, pp. 308-314, Puerto Rico, May 2003. http://cs.baylor.edu/~wilsonj/research/FerryFTDCS.pdf
11. ↑ B. Burns, O. Brock, B. Levine. "MV Routing and Capacity Building in Disruption Tolerant Networks." *IEEE Infocom*, Miami, FL, 2005. http://cs.baylor.edu/~wilsonj/research/Infocom2005.pdf
12. ↑ M. Demmer, K. Fall. "DTLSR: Delay Tolerant Routing for Developing Regions." *NSDR 2007*, August 27, 2007, Kyoto, Japan. http://cs.baylor.edu/~wilsonj/research/dtn-routing-dr.pdf
13. ↑ 13.0 13.1 A. Vahdat, D. Becker. "Epidemic Routing for Partially-Connected Ad Hoc Networks." Duke University, Department of Computer Science, May 2000. http://cs.baylor.edu/~wilsonj/research/epidemic.pdf
14. ↑ 14.0 14.1 14.2 14.3 A. Lindgren, A. Doria, O. Schelen. "Probabilistic Routing in Intermittently Connected Networks." *MobiHoc 2003.* http://cs.baylor.edu/~wilsonj/research/icr_lindgren.pdf
15. ↑ 15.0 15.1 Palo Wireless Bluetooth Resource Center. "RFCOMM Protocol," http://www.palowireless.com/infotooth/tutorial/rfcomm.asp 2007
16. ↑ Bluetooth SIG. "RFCOMM with TS 07.10." June 2003. http://cs.baylor.edu/~wilsonj/research/pdf/rfcomm.pdf
17. ↑ 17.0 17.1 Bluetooth SIG. "Specification of the Bluetooth System, 2.0." November 2004. http://cs.baylor.edu/~wilsonj/research/pdf/BT_Core_v2.0_EDR.pdf
18. ↑ E. Vergetis, R. Guérin, S. Sarkar, J. Rank. "Can Bluetooth Succeed as a Large-Scale Ad Hoc Networking Technology?" *IEEE Journal on Selected Areas in Communications,* Vol. 23, No. 3, March 2005. http://cs.baylor.edu/~wilsonj/research/pdf/Vergetis_JSAC05.pdf

19. ↑ DTN Interest Mailing List http://mailman.dtnrg.org/mailman/listinfo/dtn-interest/ 2007
20. ↑ T. Clausen, C. Dearlove, J. Dean, "MANET Neighborhood Discovery Protocol." http://www.ietf.org/internet-drafts/draft-ietf-manet-nhdp-04.txt 2007
21. ↑ J. Ott, D. Kutscher, C. Dwertmann. "Integrating DTN and MANET Routing," *SIGCOMM '06.* http://www.netlab.tkk.fi/~jo/papers/2006-chants-dtn-aodv.pdf
22. ↑ M. Musolesi, S. Hailes, C. Mascolo. "Adaptive Routing for Intermittently Connected Mobile Ad Hoc Networks" *IEEE WoWMoM 2005*.
23. ↑ http://www.debian.org/CD/ 2007
24. ↑ http://www.dtnrg.org/wiki/ProphetRouter 2007
25. ↑ http://flamebox.sourceforge.net/ 2007
26. ↑ A. Huang, "An Introduction to Bluetooth Programming" http://people.csail.mit.edu/albert/bluez-intro/index.html 2007
27. ↑ PRoPHET Façade UML Documentation, http://cs.baylor.edu/~wilsonj/uml.tgz 2007
28. ↑ http://www.palowireless.com/infotooth/images/tutorial_images/spec_stack%2egif 2007
29. ↑ T. Salonidis, P. Bhagwat, L. Tassiulas. "Proximity Awareness and Fact Connection Establishment in Bluetooth," *Proceedings of the First Annual ACM Workshop on Mobile and Ad Hoc Networking and Computing,* 2000. http://cs.baylor.edu/~wilsonj/research/mobihoc00e.pdf
30. ↑ [30.0] [30.1] BlueZ mail list archives, http://www.bluez.org/lists.html 2007
