BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

BACHELORS THESIS

Disruption Tolerant Networks for Underwater Communications

Author:
Arnav DHAMIJA
(arnav.dhamija@gmail.com)

Supervisors:
Prof. Mandar CHITRE
&
Dr. Suvadip BATABYAL

A thesis submitted in partial fulfillment of the requirements of BITS F421T

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
May 10, 2019

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

Abstract

Bachelor of Engineering (Hons.) Computer Science

Disruption Tolerant Networks for Underwater Communications

by Arnav DHAMIJA

Disruption Tolerant Networks (DTNs) are employed in applications where the network is likely to be disrupted due to environmental conditions or where the network topology makes it impossible to find a direct route from the sender to the receiver. Underwater networks typically use acoustic waves for transmitting data. However, these waves are susceptible to interference from sources of noise such as the wake from ships, sounds from snapping shrimp, and collisions from acoustic waves generated by other nodes.

DTNs are good candidates for situations where successfully delivering the message is more important than low delivery times and high network throughput. This is true for certain applications of underwater networks. DTNs can also create new options for network topologies, such as opening up the possibility of using "data muling" nodes if the network is resilient to delays.

The Acoustic Research Laboratory (ARL) at NUS has developed their own Groovy-based underwater network simulator called *UnetStack*, in which network protocols can be designed and tested in a simulator. These protocols can later be directly deployed on physical hardware, such as Subnero's underwater modems. Hence, this project revolves around creating a new UnetStack protocol called `DtnLink` for enabling disruption tolerant networking in various use cases of the ARL.

Contents

List of Figures

List of Tables

Abbreviations

ARL	**A**coustic **R**esearch **L**aboratory
AUV	**A**utonomous **U**nderwater **V**ehicle
DTN	**D**isruption **T**olerant **N**etwork
EM	**E**lectro**m**agnetic
JVM	**J**ava **V**irtual **M**achine
MTU	**M**aximum **T**ransmission **U**nit
PDU	**P**rotocol **D**ata **U**nit
RFC	**R**equest **F**or **C**omments
SCAF	**S**tore **C**arry **A**nd **F**orward
TTL	**T**ime **T**o **L**ive

Chapter 1

Introduction

1.1 Overview

1.1.1 Disruption Tolerant Networks

Disruption Tolerant Networks (DTNs) are used in a number of applications where conventional communication schemes are inadequate due to erratic network conditions, lack of network infrastructure, or long propagation delays in the communication medium. Unlike conventional network protocols which rely on end-to-end connectivity at a given instant of time, DTNs do *not* require a complete path from the source to the destination when transmitting the message.

Furthermore, some DTN protocols [1] create multiple copies of the messages, expecting at least one of these copies to opportunistically reach the destination node. All types of DTN protocols employ a type of Store-Carry-And-Forward (SCAF) mechanism to store the message until it can be sent to the destination. The message can either be sent directly to the destination or via another node in multi-hop routing.

This makes DTNs very useful for sending data when the network used inherently unreliable due to environmental conditions and when delivery is prioritised over network throughput. For example, NASA used their own implementation of a DTN to communicate with the ISS from Earth [2].

1.1.2 Underwater Acoustic Networks

Underwater wireless communication is a developing field [3], which presents several issues which are not typically encountered in terrestrial wireless networks. For one, electromagnetic waves do not propagate through water due its high dielectric constant, so conventional RF wireless protocols can not be used. Instead, acoustic waves are used for encoding and transmitting

information. However, being based on sound waves, this is much more susceptible to interference from sources of noise such as the wake from ships, sounds from animals, and collisions from acoustic waves generated by other nodes. In particular, Singapore is a challenging environment for deploying underwater acoustic networks due to the noise created from its busy shipping industry. Singapore also is the natural habitat of snapping shrimp, which produce a distinct sound wave which interferes with these acoustic waves [4]. Due to all these issues, there is a higher probability of transmitted messages being dropped due to the lossy channel medium than there is typical RF networks. Hence, the network is more likely to be *disrupted*.

Acoustic waves travel at the speed of sound in water (around $1500\,\mathrm{m/s}$), which is several orders of magnitude slower than EM waves which travel at the speed of light. This can result in high propagation delays (d_{prop}). The bitrates of acoustic networks is low, usually in the order of 5 KB/sec. This leads to longer transmission delays (d_{trans}). Processing of acoustic waves can involve significant error correction and signal processing which contributes to a higher processing delay as well (d_{proc}). Putting all this together, we get the following delay for sending a single message:

$$d_{end-to-end} = d_{prop} + d_{trans} + d_{proc}$$

Therefore, delays can be significant in underwater networks and protocols need to be designed which take these delays into account.

We can see that underwater networks can be affected by both *disruptions* and *delays* in the channel medium. DTN protocols are meant to alleviate the affect of both of these issues, making them a good fit for underwater networks.

1.1.2.1 UnetStack

The *Unet* (Underwater Networks) project is jointly developed by the Acoustics Research Laboratory (ARL) and its commercial partner, Subnero[1]. UnetStack [5] is an agent-based network simulator which is used for testing the protocols that are used in real-world deployments of underwater networks. UnetStack uses a *software-in-the-loop* network stack based on the Java Virtual Machine (JVM) which allows protocols developed in UnetStack to be directly deployed on hardware. It has APIs for Groovy, Java, Python, and C.

UnetStack does not use the conventional layered network model. Instead, the network stack is divided into "agents" which handle different concerns of the network. For example, the `ROUTING` agent handles the management of routes of messages and the `PHYSICAL` agent can be used as a driver for an underwater modem. Messages can be directly passed from one agent to another.

[1] https://subnero.com/

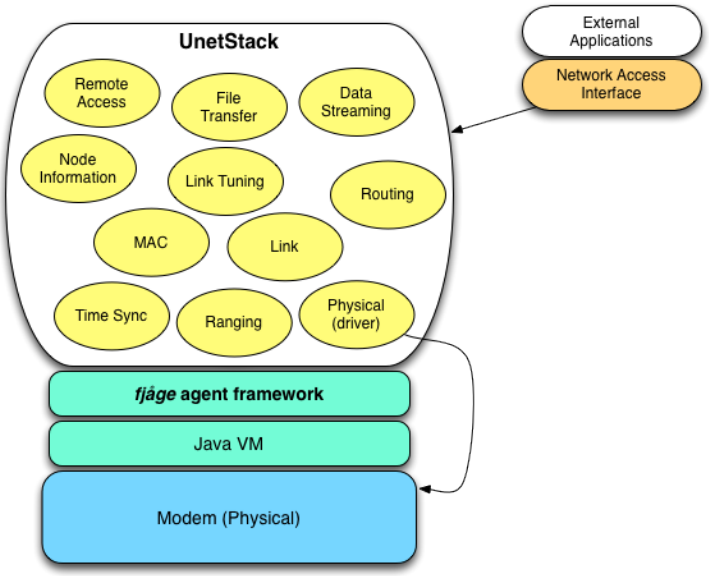

FIGURE 1.1: UnetStack Agent Architecture

This flexibility is particularly important in underwater networks where network bandwidth is at a premium.

1.2 Use Cases

This project is about developing a new `LINK` agent which will implement a DTN protocol. This agent will be called `DtnLink` throughout this report. It is designed for the use cases of some of ARL's projects.

Some of these are as follows:

- **Data Muling**: UnetStack is used on sensor nodes for collecting sensor measurements from parts of the ocean. The sensor stores the data until a diver can retrieve the sensor. This is a labour intensive procedure. To supplant this, `DtnLink` can be used for sending the sensor's data to an AUV [6] when it comes in range of the sensor. Unet AUV's have sophisticated algorithms for navigating towards a sensor for establishing a link for communication [7].
- **Time Varying Links**: A concern in underwater networks is that certain links are only available under certain conditions. For example, high bandwidth optical links are short-ranged and require a Line of Sight to the destination for communication. Ideally, `DtnLink` should be able to choose the most optimal link depending on the link's availability and bitrate.
- **USB Link**: `DtnLink` will maintain a list of pending messages in the node's non-volatile storage. As a potential alternative to sending these messages wirelessly, a USB Link agent

could work in conjunction with `DtnLink` for automatically copying these messages to an external storage device.

FIGURE 1.2: The NUSwan Robot

- **NUSwan**: The *NUSwan* [8] in Figure 1.2 is a water surface dwelling robot which autonomously collects data about the water quality in Singapore's reservoirs with its sensors. This data is relayed to the cloud using an LTE connection. However, in large reservoirs, the LTE connection may be temporarily unavailable due to lack of coverage. `DtnLink` can store pending messages and then send these messages when the LTE link is available.

1.3 Modelling a DTN Protocol for Underwater Networks

`DtnLink` aims to be a drop-in addition to UnetStack for adding disruption tolerant communication support. Hence, it is essential to define some of the features which are required for disruption tolerant communication in underwater networks:

1.3.1 Node Advertisement Messages

As previously mentioned, underwater communication is adversely affected by packet collisions. Hence, a pending message should only be sent when the sender is within communication range of another node to avoid flooding the network with messages which cannot reach the destination. To accomplish this, `DtnLink` SHOULD periodically send a message without any data at a set interval to advertise its existence to nearby nodes (a so-called "Beacon" message). On receiving this Beacon message, a node can start sending datagrams to the Beacon's sender.

UnetStack nodes also have the capability to *snoop* on the messages destined for other nodes sharing the same physical medium for communication. This capability is used for discovery of other nodes without having to send an explicit Beacon message. Additionally, `DtnLink` SHOULD NOT send an additional Beacon message if it has sent a datagram on a particular link in that time period.

`DtnLink` MUST support the capability to store datagrams on the non-volatile storage of nodes until it can be sent to the destination. It MUST also delete datagrams whose TTL has expired.

Note that the working of this Beacon functionality is under the assumption that the connectivity of the links is symmetric. That is, if Node A can receive a transmission from Node B, Node B is also able to receive a transmission from Node A. However, this assumption may not be valid for certain underwater applications.

1.3.2 Required Features

- **Storage**: `DtnLink` MUST store datagrams on the node's non-volatile storage until the datagram can be sent to the destination.
- **TTL**: Datagrams saved to the node's non-volatile storage MUST be deleted when the TTL of the datagram is exceeded. TTL information for a datagram MUST be propagated through the network. `DtnLink` SHOULD do so by encapsulating a datagram in its own PDU as described in Section 2.1. As each node may not have its clock in sync with other nodes, TTL SHOULD be stored as the time left till the message expires instead of an expiry time for a particular node.
- **Reliability**: `DtnLink` only uses Link agents supporting reliability for sending messages. Hence, we are guaranteed to know if a datagram has failed or has been successfully delivered. `DtnLink` SHOULD forward a `DatagramDeliveryNtf` to the requesting application. On the other hand, if a datagram times out, `DtnLink` SHOULD send a `DatagramFailureNtf` to the application.
- **Node Advertisement**: As explained in Section 1.3.1, `DtnLink` SHOULD periodically send "Beacons" on all its underlying Link agents for alerting other nodes about its presence. On receiving a Beacon, a node can start sending messages residing in its non-volatile storage to that node.
- **Multiple Links**: A particular node may have multiple available Link agents. `DtnLink` SHOULD populate a list of all the Link agents which support reliability. `DtnLink` MAY also automatically switching between links depending on whether they have a connection to the next hop for a message.
- **Power Failure Recovery**: Power failure on a node will typically cause the node to drop all pending messages in its buffers. `DtnLink` MAY implement a mechanism of gracefully recovering from power loss by resending messages which are pending in the node's non-volatile storage provided their TTLs have not expired.
- **Fragmentation**: Messages which exceed the MTU of the underlying links MAY be split by `DtnLink` into smaller fragments which are sent like regular message. If the implementation

supports fragmentation, the receiving instance of `DtnLink` MUST wait for the reception of all of these fragments before reassembling the original message. Fragments MUST be encoded in the `DtnLink's` PDU format.

- **Randomised Sending**: While a very rare issue in real-world deployments, message sent at exactly the same time can result in collisions in simulations. Hence, `DtnLink` MAY delay sending message by a random amount of time.
- **Stop-And-Wait Sending**: To avoid congesting the channel medium, `DtnLink` MAY adopt the strategy of only sending one message at a time and waiting for a notification about its receipt before sending the next one.
- **Short-circuit Sending**: As implemented by the newer UnetStack3 agents, `DtnLink` MAY support short-circuiting messages on single-hop routes by sending the message without its PDU headers. This reduces the message size. Regardless, messages exceeding the MTU will still need to be encoded in PDUs to be reassembled at the receiver's instance of `DtnLink`. However, messages sent through short-circuiting may be duplicated at the receiver.
- **Single-copy Routing**: Some DTN routing algorithms use packet replication to send messages to the destination. This approach might be sub-optimal for underwater networks which are constrained by transmission power limitations and suffer from packet collisions when the network is flooded with messages. Therefore, an implementation of DTNs for UnetStack MAY NOT use packet replication.

From this set of requirements, we can identify which ones can be included in our protocol. The `DtnLink` agent supports all of theses features, including the optional ones as illustrated in Chapter 2.

Chapter 2

Design

`DtnLink` is written in Apache Groovy, the lingua franca of the *fjåge* and the *Unet* project [9]. Groovy runs on the JVM and can be used either statically and dynamically. This allows it to be fully compatible with all Java code and its associated libraries.

2.1 Message Sending

2.1.1 The DtnLink PDU

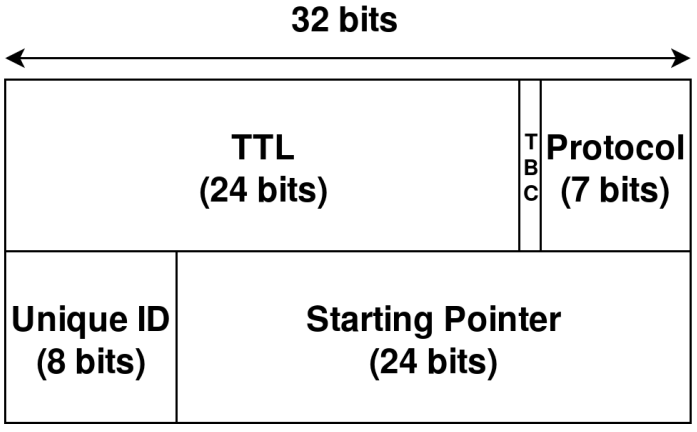

FIGURE 2.1: The DtnLink Protocol Data Unit (PDU)

Before sending messages, `DtnLink` encodes the data in a PDU (Protocol Data Unit) which is encapsulated in a `DatagramReq` before sending on a link.

The structure of this PDU is shown in Figure 2.1. The following is the data represented in this PDU:

- 24-bit TTL, representing the lifetime of the message in seconds.
- 1-bit To Be Continued (TBC) bit, for informing the receiver if more fragments are expected for large messages which do not fit in the `LINK`'s MTU. A value of 0 indicates the transmission is complete for that payload.
- 7-bit Protocol number of the original message. This is used by UnetAgents for identifying which `DatagramNtfs` are intended for them.
- 8-bit Unique ID, for uniquely identifying messages and distinguishing payloads by the tuple of their sender and the Payload ID.
- 24-bit Starting pointer, for informing the receiver about where to insert the contents of a fragment into its payload file.

A `DatagramReq` is the data structure used for sending messages between agents in UnetStack. This PDU is generated when the `DtnLink` receives a `DatagramReq` containing the the message from another agent. Before sending to the destination node, the message's TTL is updated. Therefore the `DtnLink` PDU helps in tracking the message's TTL, identifying duplicate messages, and managing the sending of large messages (*payloads*).

The following examples illustrate the different cases handled by the `DtnLink` agent.

2.1.2 Single-Hop Message Delivery

In these examples, we can see how the `DtnLink` sends messages to the destination by encoding the information in its PDU format, described in Section 2.1.1.

In the below figures, a UnetAgent application (App/1) on Node 1 wants to send a message via its `DtnLink` agent (DTNL/1). `DtnLink` encodes the message in its PDU and then it uses an underlying `ReliableLink` (RL/1). The blue part of the figure indicates the message being transmitted physically underwater. After reception of the message at the modem of Node 2, the ReliableLink (RL/2) will pass the message upto the node's `DtnLink` (DTNL/2). Here, the message will be decoded, the message information will be extracted and passed onto the application of Node 2 (App/2).

In Figure 2.2 we can see how the `DtnLink` sends messages when the destination is the next hop in the network. In this case, `DtnLink` waits until the destination node is online by receiving its Beacon message and then sends the datagram. On successful delivery acknowledgement from

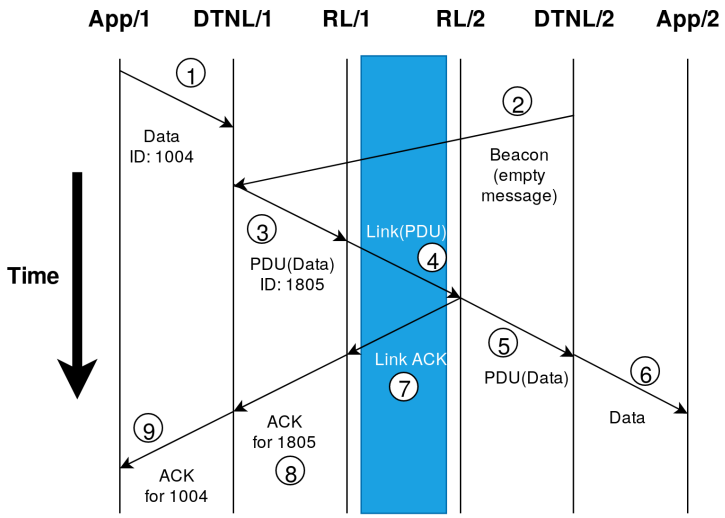

FIGURE 2.2: Single-Hop Delivery

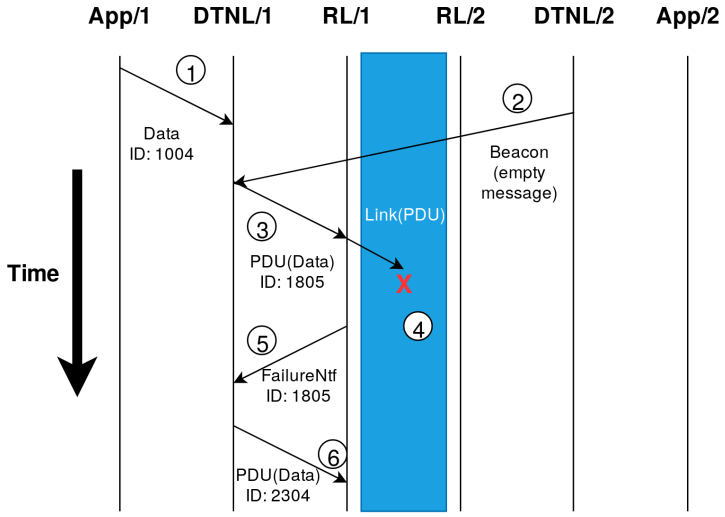

FIGURE 2.3: Single-Hop Failure

the underlying link, the ACK, called a `DatagramDeliveryNtf` in UnetStack terminology, is passed up to the application.

Figure 2.3 illustrates an example of failure of sending a datagram. In case if the sender does not receive a `DatagramDeliveryNtf` (ACK) before its timeout period, the underlying link on the sender will send a `DatagramFailureNtf` which is received by the `DtnLink`. As failing to send a message at a particular point of time is *not* necessarily failure in DTNs, the `DtnLink` will attempt to send the message at a later point of time when the destination node is online.

TTL expiry as shown in Figure 2.4 can occur when the destination node is not online during the lifetime of the message. In these cases, the message is deleted from the sender and it can no longer be sent in any circumstances. The `DtnLink` informs the requesting application about this failure with a `DatagramFailureNtf`.

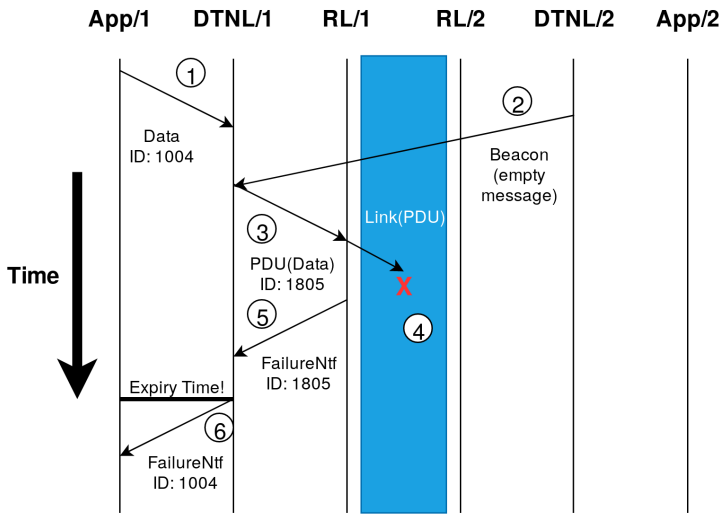

FIGURE 2.4: Single-Hop TTL Expiry

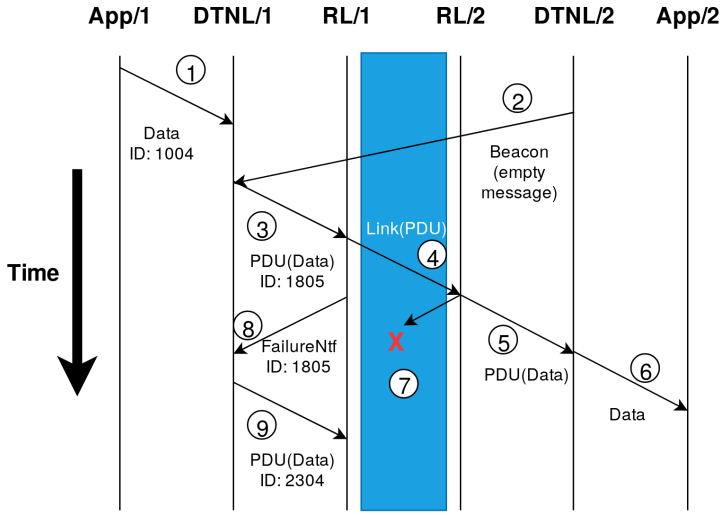

FIGURE 2.5: Single-Hop ACK Failure

In the final case, Figure 2.5 shows a problematic scenario in which the `DatagramDeliveryNtf` (ACK) message is lost due to a lossy channel medium. When this happens, the link on the sender's side will timeout and will generate a `DatagramFailureNtf` for `DtnLink`. This will make the `DtnLink` resend the message, resulting in the receiver receiving duplicate messages. Clearly, this problem must be avoided by having `DtnLink` check for duplicate messages.

2.1.3 Duplicate Message Detection

As shown in Figure 2.5, a dropped ACK can cause a message to be sent repeatedly until a `LINK` level ACK is received. This can cause duplicate messages at the receiver.

`DtnLink` solves this by encoding a random, 8-bit Unique ID in the PDU (Section 2.1.1) for each message. When a receiver receives a message, it computes the `hashCode` of the entire message after excluding the TTL field of the PDU. This value is stored in a `Set` in the receiver's instance of `DtnLink`. If the generated `hashCode` does not exist in this `Set`, the message is sent to the application, else the message is discarded.

2.1.4 Short Circuit Message Sending

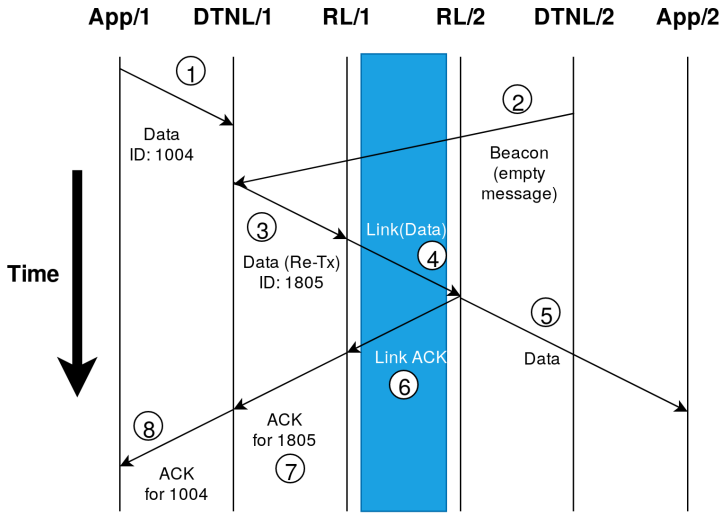

FIGURE 2.6: Short Circuiting Messages

In some cases, we might choose to send the message directly without encoding the `DtnLink` PDU headers to reduce message size. In this case, we can *short circuit* the message, as briefly mentioned in Section 1.3.2. Figure 2.6 shows how a message is transmitted straight to the desired agent without encoding it in the `DtnLink` PDU format.

However, the trade-off of short circuiting is that it can only work for single-hop messages and it eschews the duplication message detection mechanism which was explained in Section 2.1.3.

2.2 Power Failure Recovery

Disruption tolerant networks can also be affected by disruptions in the network infrastructure. For instance, it's possible that a battery powered node runs out of charge in the middle of a mission or a solar powered buoy loses power on a cloudy day. Ideally, we would want our protocol to be able to *gracefully* recover from such disruptions and keep the pending messages intact for sending in the future.

`DtnLink` is capable of recovering from the situation in which a node is unexpectedly shutdown and the `DtnLink` agent is terminated. This is implemented by saving the next hop of a message

and its expiry time along with its PDU on the node's non-volatile storage. On startup `DtnLink` scans its directory for pending messages which have not yet expired. The next hop and expiry time of a message is used in rebuilding the `metadataMap` which is used by `DtnStorage` for tracking pending messages. Once this is done, `DtnLink` can send the messages via the strategies discussed in Section 2.1.2.

2.3 Components

`DtnLink` has been designed with modularity in mind. The following classes were created to implement `DtnLink`.

- `DtnLink`: `DtnLink` extends `UnetAgent` and handles the sending and receiving of messages. As explained in Section 1.3.1 and Section 2.1.3 it also sends Beacon messages and checks for duplicate messages. It also responds to `DatagramDeliveryNtf` and `DatagramFailureNtf` messages and sends messages according to the priority set by the user.

  `DtnLink` is configured to only send datagrams on `LINK` agents which support the `RELIABILITY` capability. Supporting `RELIABILITY` does not mean that the `LINK` will always be able to successfully send the message. Instead, it means that the `LINK` agent is able to generate acknowledgements for every message sent.

  When the `DtnLink` finds a new node (either through a probe or a snooped message), it will query this data structure for the PDUs destined for the node. Once this is done, the TTLs are checked for expiry and sent by a `LINK` agent.

  As we are exclusively using `LINK` agents with `RELIABILITY` we are *guaranteed* to get a acknowledgement about the result of the delivery. If `DtnLink` is notified of a successful transmission, the entry is deleted from the tracking `Hashmap` in `DtnStorage` and the corresponding PDU file is deleted along with it. If the `DtnLink` receives a notification about delivery failure, it attempts to send the message at a later time when the node is within the transmission range.

- `DtnStorage`: This will handle the storage mechanism. It will track outbound PDUs, fragment and reassemble payload messages, and will delete expired PDUs. Expired messages are deleted periodically at an interval which can be set by the user. It also encodes and decodes into a format which can be used by `DtnLink`. It can also restore the state of `DtnLink` after restarting from power failure.

- `DtnLinkManager`: `DtnLink` is expected to handle a variety of `LINK` agents. A node can have a number of communication media, such as an underwater acoustic modem, optical link, and Ethernet tether. Some nodes may only support one of these communication media. The `DtnLinkManager` class maintains data structures which store information about the

properties of each `LINK` agent that a node supports. It also maintains lists of the links supported by the node's neighbouring nodes. These data structures are updated on every message received by the node. Furthermore, a user can also set the priority of links used for communicating between two nodes which share more than one common `LINK`.

- `DtnPduMetadata`: This class allows `DtnStorage` to track the messages which it has sent to other nodes. Each message ID has a corresponding `DtnPduMetadata` object which records the next hop destination of the message, its expiry time, and the number of bytes of the message successfully transmitted for payloads.

2.4 Capabilities

This agent will support the `LINK` and `DATAGRAM` service. Other agents can forward messages with a valid TTL value to the `DtnLink` for disruption tolerant delivery. Messages without a valid TTL will be refused outright.

In its current iteration, `DtnLink` only support single-copy and single-hop routing. If required, `ROUTING` agents can be used in conjunction with `DtnLink` for multi-hop purposes.

2.5 Configurable Options

`DtnLink` is highly configurable and the following Parameters can be adjusted depending to the use-case:

- **Short circuit**: As explained in Section 2.1.4, short circuiting messages can reduce message size. However, this parameter is turned off by default as short circuiting messages makes it impossible to check for duplicate receptions of a particular message.
- **Periodic Functions**: The `beaconTimeout` (maximum time of the link being idle before sending a Beacon message), `GCPeriod` (time period of deleting expired and delivered datagrams from non-volatile storage), `datagramResetPeriod` (time period of sending datagrams), and `linkExpiryTime` (time for which a link can remain idle without removing it from the active links list) parameters can be set at runtime.
- **Datagram Priority**: Messages can be sent according to their order of `ARRIVAL`, ascending order of `EXPIRY` times, and in a `RANDOM` manner. These options are exposed in the `datagramPriority` parameter.
- **Link Priority**: The order in which underlying links are used by `DtnLink` can be changed by sending a list of the AgentIDs to `linkPriority`. If these AgentIDs are `null` or not registered by the `DtnLink`, the request will be ignored.

2.5.1 Automated Regression Testing

`DtnLink` can be tested reproducibly. As new features are added to the agent, it is imperative that a basic subset of its functionality remains intact. These tests check that the key features of `DtnLink` are working correctly.

As the `DtnLink` will work in conjunction with several other agents, it is more useful to see the output of the agent on certain inputs rather than diving into the implementation of how each function performs. This is formally called "Black-box" testing.

FIGURE 2.7: Black-box testing

The above figure is a simple example of the key concept of the black-box. The internals can be totally abstracted for the tests as we only wish to see the outputs of the black-box on certain inputs. In these tests, the `DtnLink` is the black-box and the specially developed `TestApp` and `TestLink` agents test the behaviour of the `DtnLink`. More specifically, the `TestApp` prepares `DatagramReqs` for sending to the `DtnLink` and the `TestLink` checks the receipt of these datagrams, and send the corresponding `Ntfs` to the `DtnLink`.

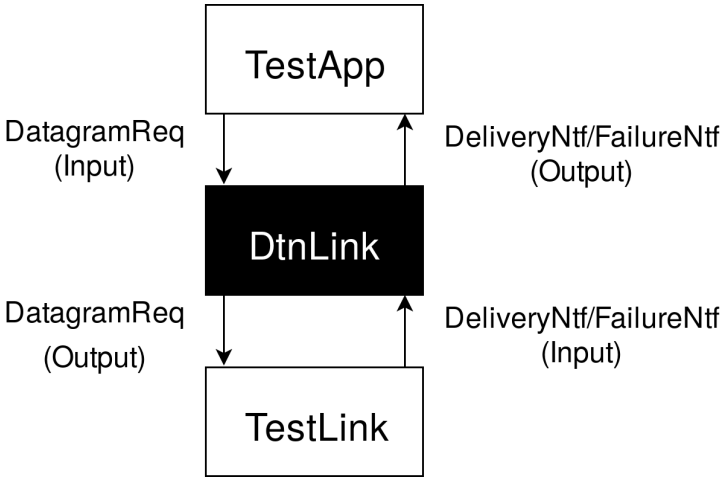

FIGURE 2.8: The DtnLink Test Harness

Figure 2.8 shows what the test harness of `DtnLink` looks like. In this, we have created a new `TestLink` and `TestApp` class which sends `DatagramReq` messages (input) to `DtnLink`. We can then check if `DtnLink` produces the correct messages (output) at the App and Link parts of the test.

By these means, we can "trick" the DtnLink into behaving as it would in a multi node simulation. The following tests are conducted with this test suite, using JUnit[1]:

- TRIVIAL_MESSAGE: This test sends an empty DatagramReq to DtnLink to check if the agent correctly accepts messages with TTLs encoded.
- SUCCESSFUL_DELIVERY: This test sends a DatagramReq with the USER protocol number to check if the message sent to the underlying link is sent without the DtnLink headers and can be short circuited. It also checks whether the DatagramReq is formatted correctly and has the original Protocol number.
- ROUTER_MESSAGE: This test sends a DatagramReq with the ROUTING protocol number to check if the message sent to the underlying link is encoded correctly with the DtnLink PDU scheme and has its TTL adjusted accordingly.
- BAD_MESSAGE: This test checks if the DtnLink responds with a Performative.REFUSE when it receives a DatagramReq without a set TTL value.
- EXPIRY_PRIORITY: This test checks if the messages sent to DtnLink in EXPIRY_PRIORITY mode from TestApp are forwarded to the TestLink in order ascending order of their TTL values.
- ARRIVAL_PRIORITY: This test checks if the messages sent to DtnLink in ARRIVAL_PRIORITY mode from TestApp are forwarded to the TestLink in order ascending order of their arrival times.
- RANDOM_PRIORITY: This test checks if the messages sent to DtnLink in RANDOM_PRIORITY mode from TestApp are forwarded to the TestLink in random order without regards to the TTL values or arrival time.
- LINK_TIMEOUT: This test checks if DtnLink correctly disables sending messages on links which have not sent a message for a certain period of time.
- MULTI_LINK: This test checks if DtnLink correctly uses the priority of Links set through a ParameterReq to change the priority of the links used to communicate with other nodes.
- PAYLOAD_MESSAGE: This test checks if the DtnLink is capable of correctly fragmenting a large message into smaller fragments to fit in the underlying link's MTU. These fragments are sent to another instance of DtnLink to check if they fragments can be successfully reassembled to form the original datagram.
- REBOOT: This test simulates the behaviour of DtnLink in event of a power failure. It runs two instances of DtnLink, one after the other. In the first instance, the test sends messages to DtnLink for storage and fails all its attempts to transmit the message successfully. The

[1] https://junit.org

result of this is that the `DtnLink`'s directory will be populated with unsent messages. After this, another instance of `DtnLink` is created. This test checks that `DtnLink` successfully rebuilds its `metadataMap` and transmits the messages residing in its internal storage.

Chapter 3

Simulations & Results

The `DtnLink` agent aims to improve the reliability of sending messages for real-world applications of UnetStack. To better understand how well underwater network protocols work, UnetStack includes a simulator in which underwater nodes running Unet protocols can be simulated. As the communication media is often lossy, UnetStack supports multiple underwater communication models such as the Protocol Channel Model, Basic Acoustic Model, Mission 2013a Model [5], and Urick Acoustic Model [10].

In the Protocol Channel Model, we can adjust the values of `pDetection` which allows us to simulate varying levels of disruption. `pDetection` is the probability a node will be able to detect a signal which is within the node's detection range. Lossy channels have a low value of `pDetection`.

By simulating various scenarios with `DtnLink`, we can better understand how messages can be sent in a disruption tolerant manner.

3.1 Scenarios

3.1.1 DTN Multihop

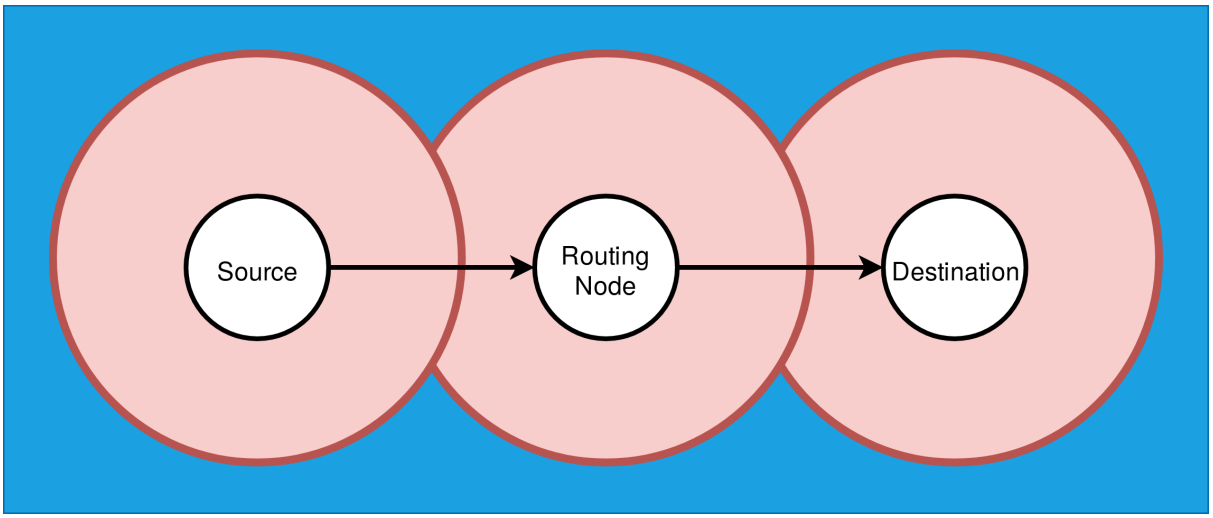

FIGURE 3.1: Multihop scenario

In this scenario (illustrated in Figure 3.1), we have a set of three nodes which are placed in such a way that the sender and receiver are out each other's communication range (shaded in red). Hence, in this scenario, we require an intermediate node which relay messages. In this simulation, we can compare the performance of `DtnLink` to the typically used `ReliableLink` to see how effective it is in lossy networks. The details of the parameters used in this simulation are given in Table 3.1 and Table 3.2.

Parameter	**Value**
Simulation Time	10 800 s
Communication Range	1500 m
Message Size	40 bytes
Message Frequency	10 s
Message TTL	10 800 s
Total Messages sent from Source	200

TABLE 3.1: Multihop simulation parameters

Node	**X**	**Y**	**Z**
Source	0 m	0 m	−50 m
Routing Node	1500 m	0 m	−50 m
Destination	3000 m	0 m	−50 m

TABLE 3.2: Node co-ordinates for the multihop scenario

The entire simulation is run for different values of `pDetection` for both `DtnLink` and `ReliableLink`.

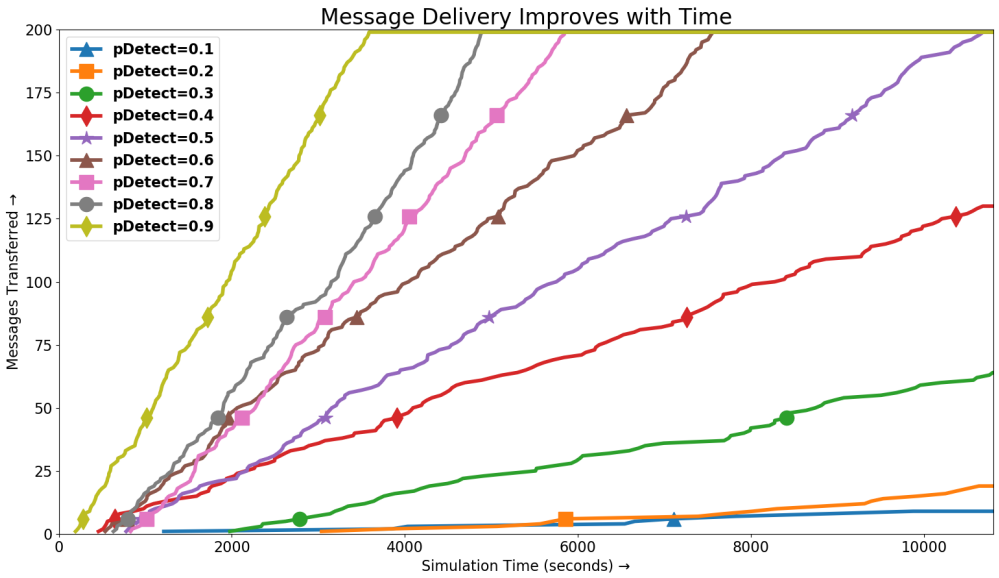

FIGURE 3.2: Effect of pDetection on message delivery versus time

In Figure 3.2, we can see that the number of messages delivered increases significantly as the simulation time increases. At lower levels of `pDetection` (0.1–0.4), the number of messages transferred is much lower as the reception of several messages fail due to the lossy channel medium. However, given that the simulation runs for enough time and the message's TTL does not expire, `DtnLink` will be able to eventually transfer all the messages sent by the sender.

When `pDetection` varies from 0.5–0.9, we can see that all 200 messages sent by the sender reach the destination node within the time frame of this simulation. Here, we can see that the time taken to transfer all the messages is affected by `pDetection`. This is because a message needs to be retried more times when `pDetection` is low.

For seeing if `DtnLink` is beneficial when we are using lossy networks with a low value of `pDetection`, it is useful if we can compare it to the performance of a commonly used `LINK` agent which does not have explicit support for disruption tolerance. In the following figures, we can see how `DtnLink` compares[1] with using the popular `ReliableLink` agent for sending messages.

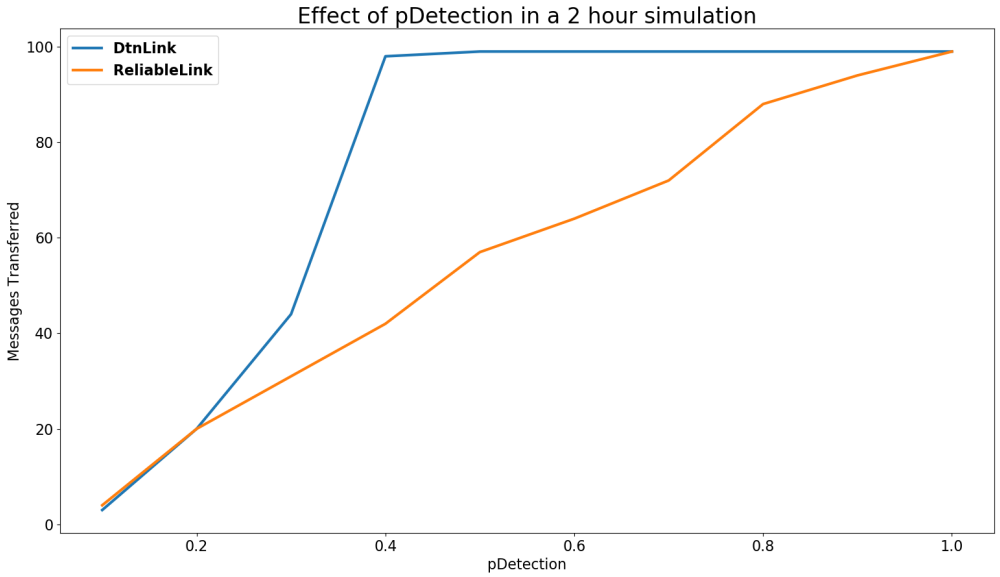

FIGURE 3.3: Comparison of ReliableLink and DtnLink for different values of pDetection

In Figure 3.3, we can see that for a set of 100 messages, `DtnLink` outperforms `ReliableLink` over a simulation time of 7200 s. This is due to `DtnLink`'s capability of being able to retry the message until it is successfully delivered. `ReliableLink`'s probability of successfully delivering the message is directly a function of `pDetection` as it will only retry a failed message until its `maxRetries` (default = 3) limit is exceeded.

However, at `pDetection` from 0.1–0.2, we can see that `DtnLink` does not offer much advantage over `ReliableLink`. The reason behind this is `DtnLink`'s Stop-And-Wait protocol of sending messages which limits the number of messages it can send in a given amount of time. If the channel is very lossy, `DtnLink` will spend a long amount of time waiting for the result of a message being delivered. However, as shown in Figure 3.2, the message will be transferred given more time. This effect is more apparent in Figure 3.4.

[1] `DtnLink` uses `ReliableLink` as the underlying link for actually transmitting messages over the channel medium. Hence, it can be expected that at the very least, that `DtnLink` should not perform worse than `ReliableLink`. Nevertheless, these simulations are illustrative of in which scenarios using `ReliableLink` as the underlying agent of `DtnLink` can be beneficial compared to using it without `DtnLink`

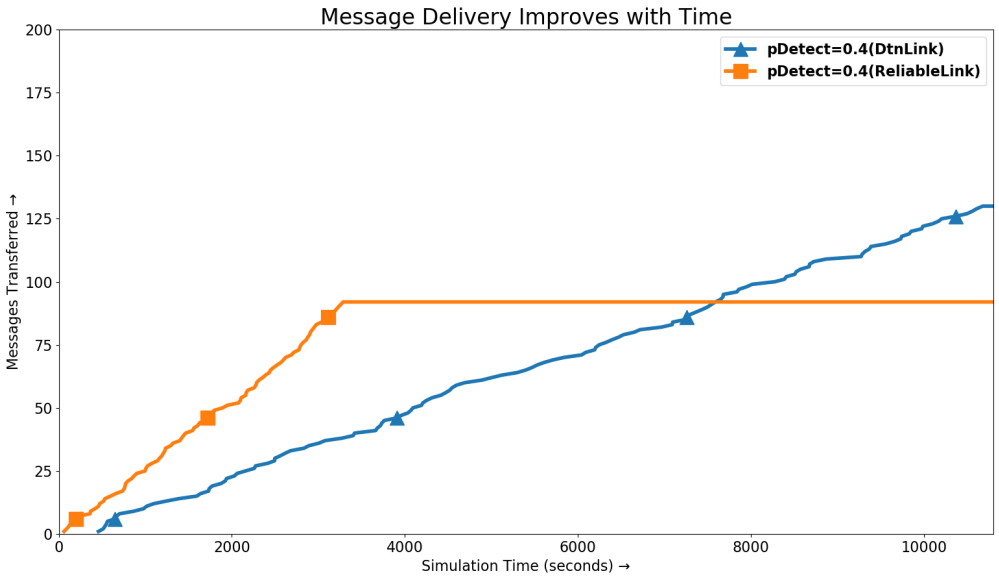

FIGURE 3.4: Comparison of ReliableLink and DtnLink for fixed value of pDetection

In Figure 3.4, we can see that the messages delivered by `DtnLink` only exceeds that of `ReliableLink` after a certain amount of time. This shows that the Stop-And-Wait sending method of `DtnLink` can negatively impact delivery times. Hence, `DtnLink` is more useful when used in an application which can tolerate long delays.

3.1.2 AUV Data Muling

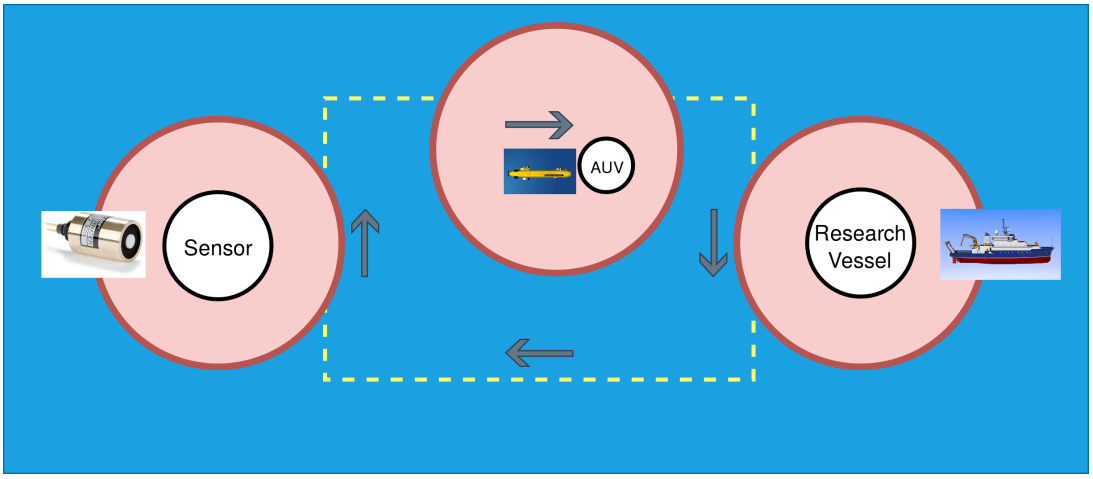

FIGURE 3.5: AUV Data Muling scenario

In Figure 3.5 we can see an example of an AUV carrying messages between two nodes - that is the sensor and the research vessel. This use case, described in Section 1.2 is a variant of the multihop demonstration shown above. In this case, the sensor and the vessel may be too far apart to transmit data to each other. An AUV with the `DtnLink` agent can help in relaying messages

between these two vessels. The parameters for the same are given below in Table 3.3 and Table 3.4.

Parameter	**Value**
Simulation Time	8800 s
Communication Range	600 m
Message Size	50 bytes
Message Frequency	10 s
Message TTL	8800 s
Total Messages sent from Source	200

TABLE 3.3: Data Muling simulation parameters

Node	**X**	**Y**	**Z**
Sensor (Source)	0 m	0 m	−50 m
AUV (Data Mule)	900 m	0 m	−50 m
Ship (Destination)	1800 m	0 m	−50 m

TABLE 3.4: Initial node co-ordinates for the Data Muling scenario

In this particular simulation, the AUV starts near the sensor and makes it way to the research vessel. It makes two rounds in the trajectory shown in Figure 3.5, with each round having a period of 4400 s. In this particular simulation, the research vessel and the sensor are kept apart at a distance of 1800 m. The detection range of the nodes is set to 600 m. The source node generates 100 messages with a TTL of 8800 s containing 50 bytes of randomly generated data which is sent out every 10 s for 2000 s. The entire simulation is run for 8800 s each for different values of `pDetection`.

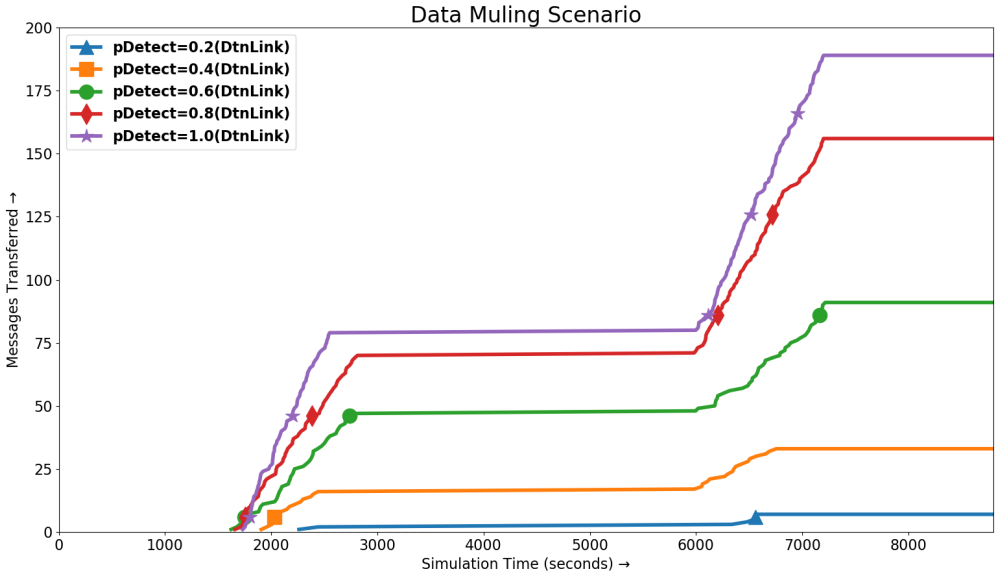

FIGURE 3.6: One Sender and one Receiver in the Data Muling scenario

Figure 3.6 illustrates this situation. At all values of `pDetection` we can see distinct trends in how messages are transferred. In the curve of messages delivered successfully at the destination, we can see that there are no messages transferred till $t = 1700\,\mathrm{s}$, a spike in message delivery till $t = 2500\,\mathrm{s}$, a period of dormancy till $t = 6000\,\mathrm{s}$, and a final spike of message delivery which lasts till $t = 7200\,\mathrm{s}$.

This can be explained by observing the periods of time in which the AUV comes in communication range of a node. At the beginning, the AUV collects data from the sensor while it is far away from the ship. This occurs till roughly $t = 900\,\mathrm{s}$, after which the AUV moves away from the sensor, with its internal storage populated with messages to be transferred to the ship.

At around $t = 1700\,\mathrm{s}$, the AUV passes by the ship and transfers its pending messages to it. This continues till the AUV has either moved out of the range of the ship or has finished sending whatever messages it picked up when it was in the range of the sensor earlier.

After this the AUV comes out of the communication range of the ship at around $t = 2500\,\mathrm{s}$. It makes another flyby of the sensor and receives more messages that were pending on the sensor's internal storage. These are carried over and transferred to the ship at around $t = 6000\,\mathrm{s}$ to transfer whatever new messages it received from the sensor between $t = 2500\,\mathrm{s}$ and $t = 6000\,\mathrm{s}$. The AUV then comes out of range of the ship at around $t = 7200\,\mathrm{s}$ and returns to its starting point next to the sensor.

These trends in message delivery can be observed consistently at all values of `pDetection`. Using a `LINK` which does not support disruption tolerance would have caused all the messages meant for the ship in this scenario to fail instantly as the sensor and the ship are never in the communication range of the AUV at the same time. Therefore, we can see that `DtnLnk` can open up new options for network topologies which were not previously possible with other `LINK` agents.

Chapter 4

Conclusions

Underwater communication is a rapidly developing field which is supplemented with acoustic communications, optical links, and AUVs. Due to various reasons such as interference due to the noise from ships, sounds from animals, and packet collisions in the channel medium, there are several challenges in successfully delivering a data underwater. UnetStack, the software stack of the Unet project allows one to deploy network protocols in software which can later be deployed on real hardware.

As shown in the scenario discussed in Section 3.1.1, `DtnLink` can significantly improve the success rate of message delivery without sending unnecessary transmissions, owing to the use of `Beacon` datagrams, timeouts for each link, and TTLs for each message. Furthermore, the data muling scenarios in Section 3.1.2 illustrate that `DtnLink` can open up new possibilities in network topologies which were not earlier possible with non-disruption tolerant `LINK` agents.

However, it is important to note that `DtnLink` may not be ideal in all use cases as demonstrated in Figure 3.4 where it can be seen that `ReliableLink` is better than `DtnLink` in time-constrained applications. Hence, it is upto the discretion of the user to configure and use `DtnLink` with parameters (Section 2.5) which best fit the environment where the protocol is to be deployed.

`DtnLink` uses an extensive `JUnit` test suite for each build for regression testing. Future work for the `DtnLink` includes expanding the concept to cover multi-hop acknowledgements with specialised underwater routing algorithms. It would also be useful to make `DtnLink` a smart protocol, which would be able to adapt to its environment according to its measurements of the channel's performance.

Appendix A

Appendix

[1] Source code for `DtnLink`

[2] RFC For Disruption Tolerant Protocols in UnetStack

[3] Final thesis presentation

Bibliography

[1] T. Spyropoulos, R. Naveed, and B. Rais, "Routing for disruption tolerant networks : taxonomy and design," pp. 2349–2370, 2010.

[2] N. Marshall and S. Flight, "DTN Implementation and Utilization Options on the International Space Station," no. April, pp. 1–13, 2010.

[3] M. Chitre, S. Shahabudeen, L. Freitag, and M. Stojanovic, "Recent advances in underwater acoustic communications & networking," *Oceans 2008*, pp. 1–10, 2008.

[4] M. Legg, "Snapping shrimp dominated natural soundscape in singapore waters," pp. 127–134, 2012.

[5] M. Chitre, R. Bhatnagar, and W. S. Soh, "UnetStack: An agent-based software stack and simulator for underwater networks," *2014 Oceans - St. John's, OCEANS 2014*, 2015.

[6] M. Chitre, "DSAAV - A distributed software architecture for autonomous vehicles," *Oceans 2008*, pp. 1–10, 2008.

[7] M. Doniec, I. Topor, M. Chitre, and D. Rus, "Autonomous, Localization-Free Underwater Data Muling Using Acoustic and Optical Communication," *Experimental Robotics*, pp. 841–857, 2013.

[8] T. Koay, A. Raste, Y. H. Tay, Y. Wu, A. Mahadevan, and S. Pieng, "Near Persistent Interactive Monitoring In Reservoirs Using NUSwan – Preliminary Field Results," 2015.

[9] A. Issac, S. A. Samad, and A. S. Jereesh, "Software tools for simulation and realization of underwater networks," in *2017 International Conference on Communication and Signal Processing (ICCSP)*, pp. 0457–0461, April 2017.

[10] R. J. Urick, "Principles of underwater sound, mcgraw-hill book co," *New York*, 1983.