THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

The Rapid-Link Transfer Protocol: Maximizing Data Exchange in Highly Mobile Networks

by
Avi Gadish

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

April 26, 2018

Professor Sam Keene, Advisor

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

_____

Dean Richard Stock, School of Engineering Date

_____

Professor Sam Keene, Advisor Date

Contents

List of Figures

List of Tables

Acknowledgements

This work is made possible by my advisor Professor Sam Keene, Professor Toby Cumberbatch, Professor Fred Fontaine of the Electrical Engineering Department at The Cooper Union

Thank you Peter Cooper, founder of The Cooper Union for your vision and for endowing this incredible institution that's "open and free to all who felt a want of scientific knowledge, as applicable to any of the useful purposes of life."

To my family and friends, you are an endless supply of motivation and support and to whom I cannot fully express my gratitude.

Thank you Maria Alejandra Dale Figeman, Jonathan Korn, David Russ, Michael Wodnicki, Eli Soffer and countless others for allowing me to engage you in countless discussions and thought exercises, and for enabling me to always move forward.

A special thanks to Professor Bhaskar Krishnamachari, Pradipta Ghosh, Kwame Wright and all of the helpful researchers at Autonomous Networks Research Group at the University of Southern California where the concept of the Rapid-Link Transfer Protocol was born. I have been so fortunate to collaborate with you over the past several years and your assistance has been instrumental towards my success.

Finally, for everlasting patience and invaluable assistance, thank you to my friends and fellow engineers in the Electrical Engineering Class of 2015 including Justin Alexander, Neema Aggarwal, Spencer Chan, Chris Curro, Dave Katz, Dennis Liang, Kelvin Lin, Caroline Yu, Caleb Zulawski and many more. You all have made learning at Cooper a stimulating, collaborative and unforgettable experience.

Abstract

Two vehicles, people, or objects (nodes) that encounter each other while in transit have an opportunity to share information. If the nodes are equipped with short-range communication radios, they may exchange data without the use of preexisting network infrastructure, provided that they are within communicable range. A wireless network comprised of such nodes is characterized by encounters between nodes that occur randomly and are sustained for only short duration. The brevity of an encounter poses unique challenges when designing a wireless protocol to facilitate data exchanges between highly mobile nodes. While the traditional transfer protocols, which support the Internet, are robust and reliable, they employ many additional services and safeguards that can be considered extraneous in networks that suffer from short encounter times. Furthermore, in vehicular networks, where two automobiles may pass one another on a freeway traveling at 60 mph, a transfer protocol like TCP may not succeed at establishing a connection before the encounter ends and the vehicles have moved out of communicable range. In the following thesis, the Rapid-Link Transfer Protocol is presented. The wireless protocol is designed for the "Internet of Moving Things" and enables connections to rapidly be established between two nodes in order to maximize the number of files that can be exchanged.

1 Introduction

1.1 Ad Hoc Networks

The Latin word “ad hoc” (literally “for this”) is used to describe things that are created or formed for a particular purpose only. When applied to the field of communication networks, an ad hoc network is any network topology in which two or more devices, called nodes communicate with each other without using any preexisting infrastructure such as the routers and access points used in conventional WiFi Hotspots and wired networks. Traditionally, these ad hoc networks are classified as point-to-point (P2P) networks and are comprised of two or more nodes that maintain a local connection across the channel by which they can exchange data. These networks require minimal configuration and the communication channel can take the form of a physical connection, such as a serial or Ethernet connection, or a wireless connection utilizing a wireless standard, such as WiFi (IEEE 802.11), Bluetooth (IEEE 802.15) or others [1,2]. Wireless ad hoc networks (WANETs) are prevalent among many multiplayer mobile phone gaming apps. The implementing of an ad hoc network is low-cost and efficient relative to creating a cellular network, which requires expensive equipment and leasing spectrum from the FCC; this makes WANETs a great network topology for use in locations where there is little or no preexisting infrastructure, when exploring or logging data in uninhabited areas such as deserts and oceans, and in emergency relief situations, where existing infrastructure may have been compromised. Ad hoc networks do not require connection to the Internet in order to exchange data so they are also deployed in locations where isolation from external networks is required while still preserving a local intranet such as in military environments.

1.2 Pedestrian Networks

If each node is associated with a single person as they walk around a specified location such as a building, a street, or even an entire city, a pedestrian network can be characterized as an ad hoc network that is formed by people that act as moving nodes. The network may be as small as a room in a building, or as large as a public plaza, or even a city and the pedestrians may repeatedly enter and exit the network at any time. For these reasons, when considering deploying a pedestrian

network, it is advantageous to think of the network as a *mobile ad hoc network*, or a MANET [3]. Considering a pedestrian network environment and a communication range of ~15 meters, two people walking past each other at 3 miles per hour may have less than 11 seconds with which to establish a link and exchange data.

One aspect in which a pedestrian network lends itself well to ad hoc networking is that people often leave or travel between areas of "network coverage." The fact that an ad hoc network does not rely on preexisting infrastructure makes it most suitable for a pedestrian network. If there is a large gathering of people at an event or if there are many people traveling through an undeveloped area, a network supported by people can be much more effective and much less costly than constructing a temporary network that would be capable of serving the large amount of users. Additionally, if one were to attempt to fit any other network topology to a pedestrian network, challenges would arise when attempting to determine which person (node) would carry the bulk of the burden and responsibility in regards to routing (e.g., determining the best path to route a given packet from its sender to its intended recipient) and network management (e.g., maintaining a list of nodes within the network). The problem is especially exacerbated if this "responsible" node happens to leave the network without notice. Therefore, it is reasonable to assert that an effective pedestrian network must be decentralized with each node holding equal responsibility within the network and each having equal opportunity to send and receive data as they travel in the network.

1.3 Vehicular Networks

Vehicular ad hoc networks (VANETs) are rapidly growing in popularity in the age of "connected cars" where modern vehicles are outfitted with various sensors to perform measurements that inform the vehicle of existing conditions on the road and surrounding conditions [4]. The motivation for communications between vehicles is clear and will be critical to achieving widespread adoption of autonomous (driverless) vehicles.

Vehicular and pedestrian networks are similarly characterized by random and opportunistic encounters between nodes in that one car does not know when it will encounter another car, nor does

it know how long the connection will be maintained before its neighboring node drives away along its course, potentially never to meet again. Ignoring the existential dread of never again encountering your newfound neighbor, the situation is worsened by the increased speeds typical to the nodes in VANETs, which, with freeway speed limit of 65 mph, can travel 10-25x faster than their pedestrian counterparts [5].

It is worth noting that minimizing power consumption is less critical in VANETs because vehicles have larger batteries than what could be achievable on a small mobile device in a MANET such as a cell phone. A vehicle outfitted with a 915 MHz Dedicated Short Range Communication (DSRC) radio has a maximum communication range of 300 feet [6]. Under these conditions, 2 vehicles traveling in opposite directions at 30 miles per hour have between 2 and 5 seconds of viable communication time.

In the following, the design of the Rapid-Link Transfer Protocol is laid out in detail. In Section 2, background topics are discussed to provide a reader with project scope, key concepts in communication theory, related work in order to convey the motivation for RLTP. In Section 3, the core design of RLTP is presented, followed by a discussion of simulations and deployment of physical nodes, which can be found in Section 4. Section 5 concludes with analyzing results and an evaluation of RLTP's feasibility in low-power environments, and further improvements to improve on RLTP in the future.

2 Background

2.1 Communication Challenges in Ad Hoc Networks

As discussed in Section 1, there is no central node in an ad hoc network topology. All nodes in the ad hoc network traditionally hold equal status and each node is free to associate and communicate with any other node provided they are within transmittable range. Any single node however, may not have access to, or be in range of, every other node within the network and as a result, some nodes may be required to act as a relay for messages addressed to other nodes within the network. These messages hop from one node to another, propagating through the network, until their intended recipient receives the message. To introduce some terminology: A *path* is the route that messages take between the initial transmitting node and its intended recipient and the path can be comprised of one or more *hops* [7]. A single-hop transmission is a message sent directly to its intended recipient, whereas a message that is relayed from one node to another before reaching its destination would be called a multi-hop transmission [8].

Algorithms that determine an optimal path for which nodes can route messages are called routing protocols. Many routing protocols have been developed that allow for messages to be transmitted to nodes that are not within its communicable range; these protocols can vary in complexity and robustness. For example, *flooding* is a simple routing scheme in which each node acts as a repeater, broadcasting a message it has received to all of its neighbors [6]. These neighbors likewise rebroadcast the same message to their own neighbors; this process continues until the message's recipient receives the message and sends an acknowledgment message (ACK) back down the path with which it arrived. While flooding is neither robust nor optimal, messages transmitted in this way will eventually arrive at their intended recipient with high probability, provided the intended recipient is present in the network. A key issue with flooding is that the network may become highly congested from duplicate messages, which could circulate within the network long after its intended recipient received the message [6]. Multi-hop routing schemes usually specify a *maximum hop count*, termed a “time to live” that ensures a packet cannot be rebroadcasted indefinitely. When the locations of nodes within the network are known, routers may incorporate *distance vector routing* schemes [7]. The distance vector

stores the best route for packets measured in terms of the numbers of hops a packet has to pass in order for it to reach its destination. Some routing protocols, including distance vector protocols can take other network information into account when determining the best route for a packet, such as network traffic and network latency.

Another challenge inherent to ad hoc networks is that nodes frequently, and without notice, disconnect from the network. The loss of a node can break the path established between two nodes if the lost node was one of the hops used to deliver packets between the nodes. The successful routing protocol for these ad hoc networks must be robust enough to handle the disappearance of a node while also being able to successfully reintegrate the node back into the network, should it reappear in at a later time even if it rejoins at a different location. Section 1 described some examples of mobile ad hoc networks where nodes often disappear and reappear in the network. For example, if we consider pedestrians as nodes moving freely within a pedestrian network, these people might leave and reenter the network each time they travel on the subway. Similarly, vehicles in a VANET inevitably disappear from the network whenever they are parked and the key is removed from the ignition. These examples show how the loss of signal or power can compromise a networks ability to function, but there are cases where it may be advantageous for a node to remove itself temporarily from the network. Nodes in Wireless Sensor Networks (WSNs), for instance, may be designed to conserve power by shutting down communication radios and only "wake up" for short intervals to send a message [8]. Ultimately, the unpredictability of highly mobile nodes makes it difficult to guarantee the reliable and timely delivery of messages, making these types of networks unsuitable for most applications involving life-safety or time-sensitive information.

Any ad hoc network requires a node to exist within communicable range of other nodes. In a mobile ad hoc network, nodes move freely and rely on encountering other nodes as they traverse the network independently. In the case of highly mobile nodes, communication is opportunistic in nature due to the encounter's stochastic occurrence and unknown duration. That is to say, if an encounter is to happen, it will occur at an unknown or random time, with an unknown or random node and that the communication may only be sustained for an unknown and random (but often very short) duration. Moreover, this connection between nodes can be supported by a link, or channel, with unknown and

variable quality that degrades without notice as the two nodes move apart [6]. Typical encounter times may range from as little as a couple of seconds (as in the case of two vehicles passing each other on a freeway) to potentially an encounter of infinite duration (if the two vehicles are stuck in bumper-to-bumper traffic, as is common to commuters traveling on one of Los Angeles's highways). As described, VANETs in particular suffer from very short and unpredictable encounter times due to traffic patterns and high speeds at which the nodes are able to move throughout the network [6]. Consequently when designing a protocol for these network topologies, it is imperative that a connection be established between nodes as rapidly as possible. If a connection is established, any proceeding communications should be focused on actual data exchanges, as the short encounter times provide extremely low overhead for non-data messages such as control or acknowledgement messages. Therefore, it is clear that wireless protocols that were developed for traditional networks will be ill equipped to handle the challenges inherent to such networks with highly mobile nodes.

2.2 A Short Introduction to Wireless Protocols

At the core of the Internet Protocol (IP) suite, which powers the Internet we use everyday, is the Transmission Control Protocol (TCP) [7]. In fact, TCP is widely used as the Internet standard for communications and the entire suite is often called TCP/IP [15]. TCP is the wireless protocol that enables the reliable, ordered, and error-checked delivery of a stream of data between two nodes in a network. Before any messages can be exchanged between nodes, TCP requires a complex three-part handshake to establish the new connection. If one was to attempt to utilize TCP to allow nodes to communicate in a MANET, the link between nodes may hardly be sustainable for a long enough period to simply establish the connection [4]. Further, we have ignored the possibility of data integrity and it is possible that any one of the three handshake messages (termed: TCP SYN, SYN-ACK, ACK) can become corrupted or lost resulting in additional delays. In [4], it was shown that the loss of a TCP handshake packet can result in up to 2.5 seconds of additional latency. Especially in the case of a VANET, the entire encounter could be spent just establishing a TCP connection! Therefore, as will be discussed later, a wireless protocol designed for these types of ad hoc networks generally opt to eliminate the lengthy but reliable handshake procedure in favor of rapidly establishing a connection in order to maximize time spend on exchanging data.

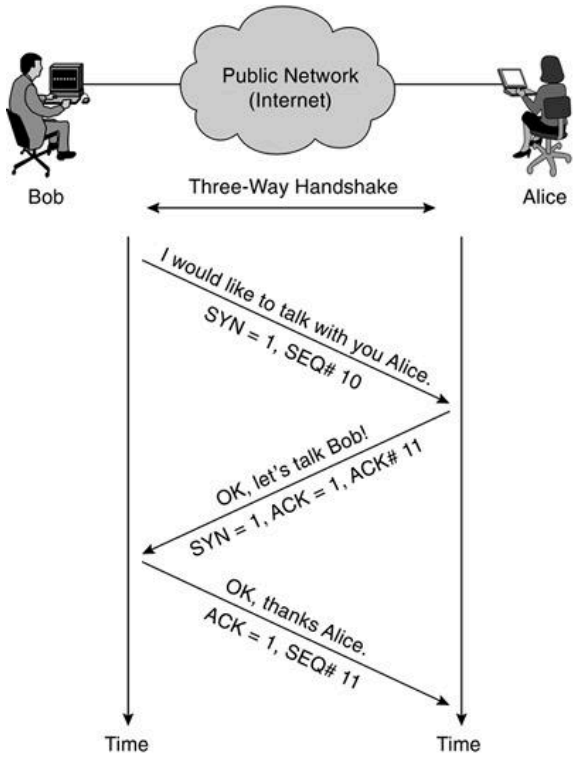


Figure 1: TCP's 3-way handshake: Bob can send data to Alice only after successfully establishing a TCP connection [9].

In addition to eliminating the handshake procedure, ensuring reliability may have to be sacrificed as well. Traditionally, control messages are used to relay information about the status of a node, the channel, or messages that have been received. For instance, an acknowledgment (ACK) message is often used to let a node know that the recipient of a message has successfully received the message [7]. If an acknowledgment message has not been received, the sending node may retransmit the message in an effort to successfully send the message. However, if the node has disappeared from the network, or in the case that it had received the original message successfully but the ACK message was lost, then retransmitting the original message is a waste of precious encounter time and should therefore be avoided. For MANETs with highly mobile nodes, such as VANETs, control messages should be minimized, modified, or eliminated in order to maximize data exchange messages by avoiding potentially unnecessary control transmissions.

If a node wishes to declare itself as part of the ad hoc network, it broadcasts a beacon packet to all nearby nodes [7]. Traditionally, a beacon packet may contain identification information about the node including, but not limited to: device identification information (such as a device ID or SSID), a time stamp used for synchronization, the beacon transmission interval, routing and recipient information such as the Traffic Indication Map (as specified in section 7.3.2.6 of the IEEE 802.11-1999 standard), as well as other parameters and hardware information such as supported data rates and any peripherals to which the node may be attached [1]. When a node receives a beacon from a neighboring node, it typically responds with an acknowledgment message that may contain its own device information. In this way the beaconing node can acquire information about the network it has just joined and all of the nearby nodes in that network.

Beaconing can present the same issues as extraneous control or acknowledgment messages and result in potentially wasteful use of encounter time unless utilized in effective ways. Traditionally, a request for information would be transmitted in a unique message frame separate from the beacon, which is solely used for node discovery [9]. Then, if a node receives a request for information that it is able to fill, a reply containing an acknowledgment message is sent back to inform the requesting node that the request will be filled. Finally, after sending an acknowledgment, the requested data is then packetized and transmitted. Although a beacon frame (format) is often standardized in the specifications of a wireless protocol, there may be fields within the beacon frame that can be *stuffed*, or combined, with other data in a way that allows additional information to be delivered in addition the standard beacon [10].

In RLTP *beacon stuffing* is employed to allow a node to share a great amount of information about itself within a single transmission that is sent at the beginning of an encounter. This is done in order to minimize transmissions that do not explicitly contain file data such as data requests or ACKs. In the same way, a node that received a beacon may reply with an acknowledgment message that has requested data appended to it. This technique may allow a node's request for data to be filled in potentially half of the transmissions when compared to other network protocols like TCP [9]. Ignoring the processing time required to process and fill each request, the time required to fill a node's requests will be cut in half as well.

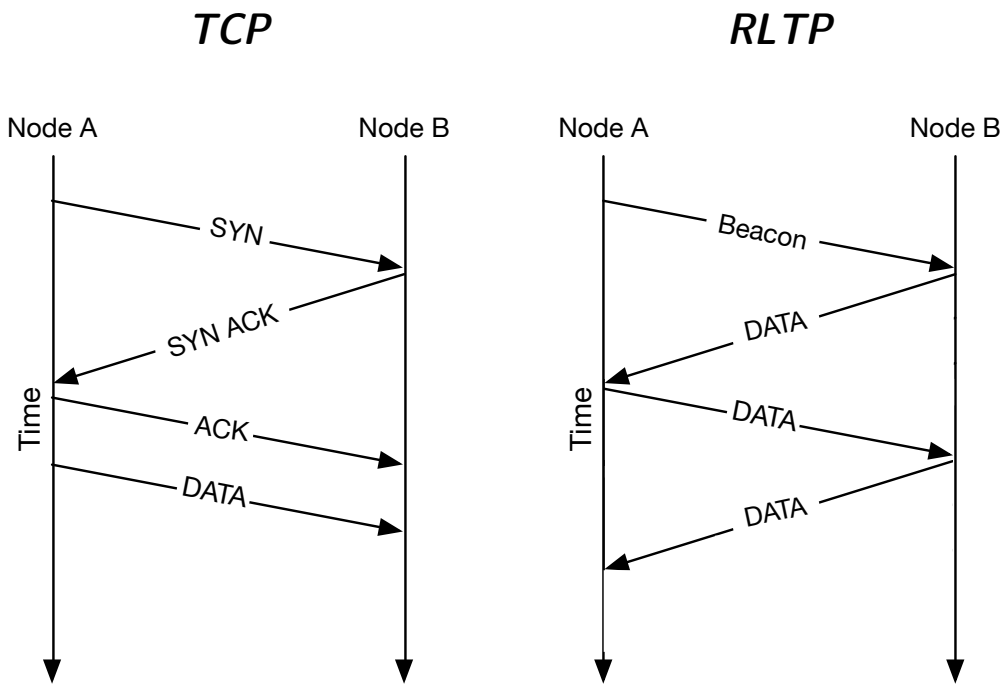


Figure 2: TCP vs. RLTP. With RLTP, file data can be transferred as early as the second transmission provided the Node B has a file requested in the beacon of Node A.

2.3 The OSI Network Model

The Open Systems Interconnection network model (OSI Model) is a conceptual model of a network that characterizes internal functions of a communication system by partitioning them into seven abstract layers [12]. In general, each layer receives control of a packet form its preceding layer and manipulates a portion of that packet before delivering it to the proceeding layer. The lower layers of the OSI model manage electrical signals, manipulate chunks of binary data, and route data across networks, while higher levels perform network requests and responses, determine how to represent data, and deliver the data to the requesting application. The standardization's origins date back to the late 1970's as part of a project at the International Organization for Standardization (ISO/IEC 7498-1) [11].

The seven layers in the "stack" of the OSI Model along with a brief description of their functions are as follows [12]:

1. Physical Layer – Transmission and reception of raw bit streams over a physical medium. The physical medium can be a wired or wireless link.
2. Data Link Layer – Reliable transmission of data frames between two nodes connected by a physical layer.
3. Network Layer – Structuring and managing a multi-node network, including addressing, routing and traffic control.
4. Transport Layer – Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing.
5. Session Layer – Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes.
6. Presentation Layer – Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption.
7. Application Layer – High-level APIs, including resource sharing, remote file access, directory services and virtual terminals.

When two OSI-compatible nodes communicate the data that is to be transmitted is packetized into a fixed-length payload called a protocol data unit (PDU) [11]. This occurs at the uppermost layer (Application Layer) of the OSI model and is then passed down to the level below. The packet that is received from a previous layer is called a service data unit (SDU) and at the lower layers, the SDU is appended with header and/or footer data that relevant to that particular layer [11]. The SDU along with the header/footer data becomes the PDU for that layer, which is then passed along to the next layer down where the process is repeated until it reaches the lowermost level from which the packet is transmitted to the receiving node. Upon receipt of a packet, the receiver node passes the packet to the lowermost level of the stack (Physical Layer). Each layer at the receiver parses and strips off its pertinent header/footer data before passing the packet as an SDU to the next higher layer. This stripping process continues on each subsequent layer until reaching the Application Layer of the receiving node, where at last the data can be consumed, as it was initially intended. If executed correctly, the entire process should be "transparent" in that any data transmitted should be received

at the Application Layer without any degradation in quality regardless of the path by which a packet traveled and irrespective of the type of network.

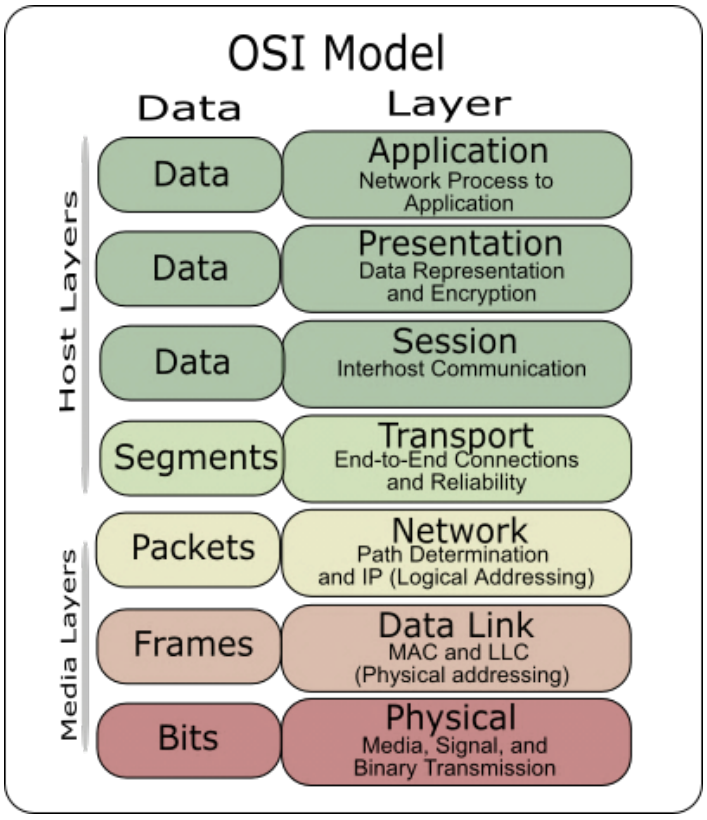


Figure 3: The seven layers of the OSI Network Model and their functions [13].

As a general rule, each layer of the stack is isolated from the rest such that the only information passed between layers are in the form of PDUs and SDUs. However, in practice this is not always the case, as some services require coordination across multiple layers. For example, certain security services (as defined by ITU-T X.800 recommendation) are present across all layers [14]. Another example of a cross-layer service is the Address Resolution Protocol (ARP), which is used to translate IPv4 addresses, which exist in OSI layer 3 (Network Layer), into Ethernet MAC addresses that are present in OSI layer 2 (Data Link Layer) [7].

It is worth mentioning that the TCP/IP model, which powers the Internet, does not strictly adhere to the seven-layer OSI Model. Instead, the Internet Protocol is structured in four broad layers. This structure is often compared to the OSI Model in that TCP/IP's Application Layer contains the OSI

Application Layer, Presentation Layer, and most of the Session Layer. The TCP/IP Transport Layer fulfills the remainder of the functions of the OSI Session Layer in addition to the OSI Transport Layer. The TCP/IP third layer, called the Internetworking Layer is a subset of the OSI Network Layer and the fourth TCP/IP layer, the Link Layer, includes the OSI Data Link and Physical Layers, as well as parts of OSI's Network Layer [15].

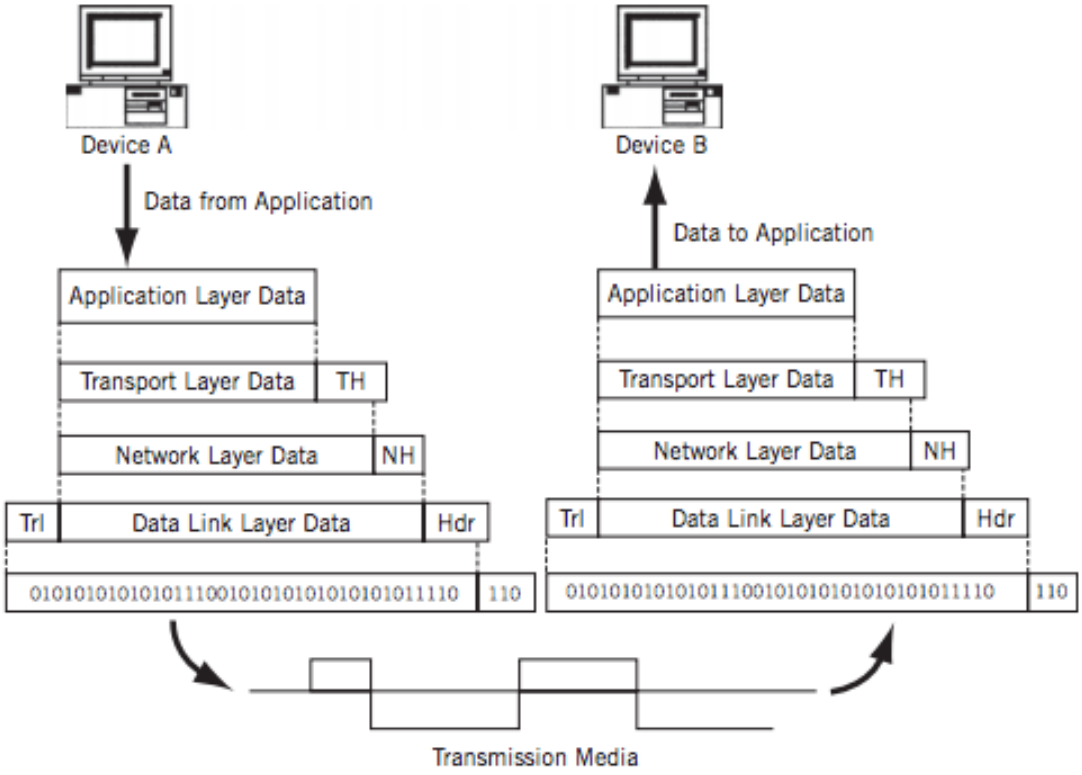


Figure 4: The TCP/IP stack. Each layer appends a header and/or footer to the SDU before passing the packet, as a PDU, to the next layer [15].

2.4 Related Work

Development of the Rapid-Link Transfer Protocol was partially motivated by previous studies aimed to determine the feasibility of exchanging data between fast-moving vehicles on the road. In their 2009 paper titled "Feasibility of Content Dissemination Between Devices in Moving Vehicles," Zahn, et al investigated different wireless protocols including TCP, and a proprietary protocol named Broadside in order to determine if those communication protocols were able to support data transfers

in networks where “connections are often short-lived and have variable link quality” [16]. Their design included frequently transmitting beacons to declare a node’s presence in the network, using Bloom filters for negotiating file transfers and a bit-vector approach to acknowledgement messages.

The problem that Zahn et al were trying to solve was sharing map data and Point of Interest (POI) files, which could be stored on Personal Navigation Devices (PNDs) like in-dash navigation systems. In their work, when two moving vehicles encounter each other on the road, they must quickly establish a connection by exchanging beacons. Next, a representation of files that each PND currently has must be transmitted between the two vehicles in order to determine which node should transmit files that the other node requires.

In order to maximize the utility of each transmission, the data structure that is used to represent the information each PND currently has must be as efficient as possible. Efficiency here means that each message exchanged should be designed to avoid lengthy negotiations between devices in order to determine which files should be exchanged. Therefore an efficient representation scheme for the files each node currently has must maximize the amount of information shared in the smallest size. The representation scheme implemented by Zahn et al utilized a Bloom filter to determine common files between the two PNDs. A Bloom filter is a space-efficient probabilistic mapping conceived by Burton Howard Bloom in 1970 for memory and time constrained applications, which are also error-tolerant [17]. Bloom filters are used to test whether a given element is a member of a predefined set. The Bloom filter consists of a large vector of bits initially set to 0 and several defined hash (mapping) functions. A hash function maps, or hashes, some set element, such as a numbered file in a dataset, to one of the positions in the bit vector with a uniform random distribution. When initializing the Bloom filter for the first time, each element is hashed to a position in the bit vector and this bit is set to 1. Since there are multiple hash functions, each element must get passed through every hash function, which results in several 1’s for each element in the Bloom filter. After the filter is initialized, if one wants to determine if an element is a member of the set, it is simply passed through each of the hash functions. If the element is mapped to any position in the bit vector that is still set to 0, then the element in question is definitely not in the set because that bit would have been set to 1 during the initialization of the Bloom filter. Due to how the Bloom filter is initialized, different elements

could be mapped to the same bit position by any of the hash functions. Thus, after initialization, there may be ambiguity in which element set the value of a bit to 1 in the bit vector. This ambiguity is the primary caveat of Bloom filters as false positive matches are possible. A false positive match could occur if an element that is not a member of the set hashes to positions that have all been set to 1 as a result of a combination of other elements, which hashed to those same positions. Notably, false negative matches are not possible when representing files with Bloom filters. So in the context of file negotiation between two PNDs, one PND transmits its initialized Bloom filter and the receiving PND is able to determine that a file it can share is either "possibly in the set" of files that its correspondent has shared or it is "definitely not in set" of files the other PND has shared. The probability of false positives is directly proportional to the number of elements represented by the Bloom filter. This can become particularly problematic as datasets can become increasingly large, and the number of elements in the dataset grows proportionally longer when dividing the dataset into a smaller number of elemental blocks. In the simulations run by Zahn et al, the expected rate of false positive matches is less than 1% for a packet size of ~1,000 bytes (~1 MB). Yet, it is unlikely that a packet size of this size would be suitable for such short encounters. Sending larger packets carries a higher likelihood that the receiving an incomplete packet when the encounter ends, rendering the partial packet unusable. Let us consider a short duration encounter where a PND was only able to transmit 750 bytes of data: with 1,000-byte packets, the received data is thrown away because it is an incomplete packet. Now, if the data were instead segmented 500 byte files then the first 500-byte packet would have been received successfully and the encounter would have terminated during the transmission of the second 500-byte file, which would be thrown away. So while higher packet sizes suffer from packet loss complications, it is likely that Zahn et al opted to use larger packets to reduce the length of the Bloom filter and thereby minimizing false positives.

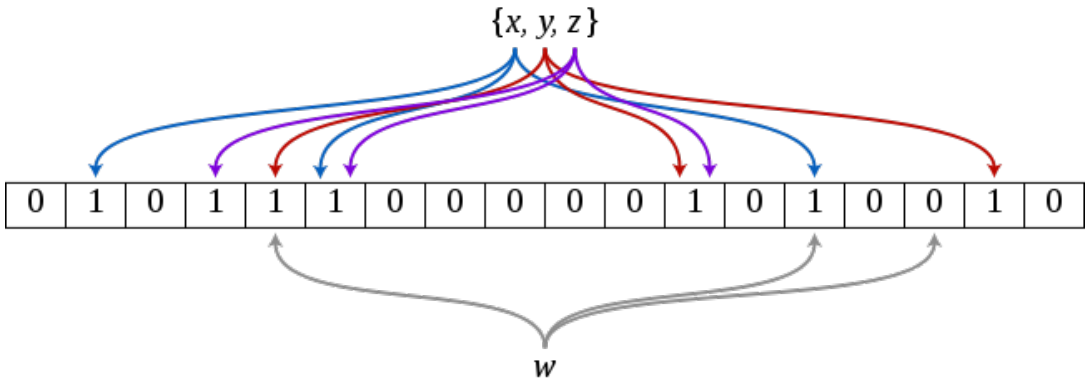


Figure 5: An example of a Bloom filter, representing the set {x,y,z}. The colored arrows show the positions in the bit vector that each set element is mapped to. Element w cannot be in the set because it hashes to at least one position that contains a 0 [18].

The study conducted by Zahn et al tested multiple wireless protocols including Broadside-IP, which implements TCP, and Broadside-Raw, which is a bare-bones protocol similar to Contiki's RIME stack, which will be discussed in the following section. Over several vehicular experiments both in residential and motorway conditions, Broadside-Raw outperformed the TCP implementation by an average of 44%. Yu and Bai, in their 2011 paper titled, "ETP: Encounter Transfer Protocol for Opportunistic Vehicle Communication," showed that ETP was able to double the throughput of TCP when both vehicles are in motion [19]. When only a single vehicle is moving (resulting in longer encounter times), [19] reports a performance improvement of 20-50% relative to TCP. These studies highlight TCP's inability to perform under vehicular network conditions and demonstrate the motivation for designing protocols specifically for use in applications for what we may term "the Internet of *Moving* Things."

Another piece of related work comes from the Autonomous Networks Research Group (ANRG) at the University of Southern California. In their 2014 paper "Optimizing Downloads over Random Duration Links in Mobile Networks," Bhargava et al present MERLIN (Maximum Expected download over Random LINks), a single-phase transfer protocol with many similarities to RLTP [20]. The term *single-phase* refers to the server-client relationship between two MERLIN nodes; in each encounter, one node sends requests for data (client), which the other node (server) attempts to provide all of the files it can share from the request before the encounter ends.

A brief overview of how the MERLIN protocol's design is as follows: the server and client nodes initially each have some (different) subset of a dataset, which is divided into N equal-sized and numbered blocks. The server periodically broadcasts its discovery beacon, which also contains the number of files the server currently has of the dataset. Upon receipt of a beacon, the client has information about its own subset of the dataset, the server's number of files, as well as statistical information about the encounter such as the distribution of the encounter duration. While [20] leaves the estimation of this distribution to future work, the MERLIN client relies on this information to determine the optimal amount of data requests to transmit. Requests for data come in 10 byte packets with the structure of {Starting File Number; Ending File Number}, which represent the largest contiguous range of files that the client requires. MERLIN's key contribution is a model that determines the mathematically optimal number of these request packets to send as a function of each node's current percentage of the dataset and the statistical distribution of the encounter duration. Due to the high computation requirements of MERLIN, much of the optimization occurs on a remote processor, which is able to solve the optimization problem more efficiently. This remote processor could be realized in the form of a vehicle's onboard computer, a user's mobile device, or even on a remote server. The remote processor provides the client node the optimal number of ranges to request. When the server receives the optimal requests, it immediately begins to send all its available files within the requests without waiting for any acknowledgement. The client then receives the sent files until the encounter ends and the protocols performance is measured by how many files could be received by the client.

MERLIN's motivation lies in the critical need to maximize the time spent on actual data transfer during in each encounter. The chart below shows the percentage of time spent on different tasks as the number of requests is varied. These results assume an Exponential encounter distribution and it can be seen that the number of request packets increases with the number of requests, while the amount of idle time decreases. The data transfer time, shaded in dark blue, initially increases then decreases indicating the presence of an optimal number of requests to send [20]. Because each of MERLIN's range requests require their own request packet, the packet time increases linearly with each additional requested range and as expected, it is not advisable, or even feasible, to request every file within a single encounter.

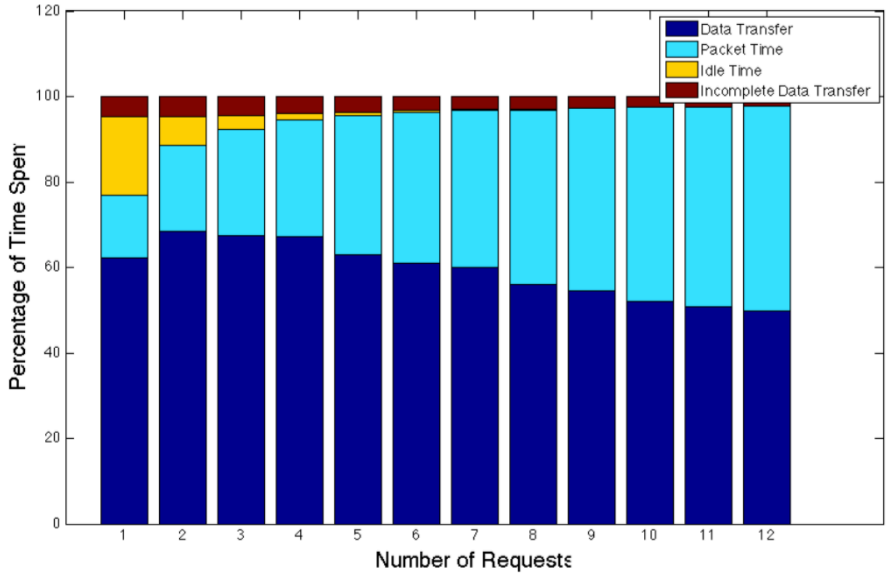


Figure 6: MERLIN's percentage of time spent on different tasks for exponential encounters [20].

The results obtained in previous and related work were essential in designing RLTP. It has been shown that the structured stack like the OSI Model or the stack used for TCP/IP connections is problematic in highly mobile networks and was therefore avoided in favor of a more efficient and barebones stack like Broadside-Raw [12,19]. In RLTP, the lightweight RIME stack was used, which will be discussed below. Also, in order to mitigate packet loss, packet sizes in RLTP were reduced as much as possible. To this end, the packet size of an RLTP data file is 102 bytes, approximately 10% of the packet size of Broadside-Raw. RLTP negotiates file transfers using a range vector representation as opposed to a Bloom filter, which would degrade in fidelity as the size of the dataset

increases. The range vector representation is not only smaller in size, but will never yield a false positive or false negative result, which ensures that nodes will not share a file that the other node already has. MERLIN was developed using a similar range vector approach to RLTP, but is implemented differently, with the primary distinction being that all range requests in RLTP are grouped together in a single packet.

2.5 Contiki: The Open-Source OS for the Internet of Things

The Internet of Things (IoT) is a term coined to represent the network of physical objects or "things" that can be embedded with electronics, software and sensors that enable these objects to exchange data with other similarly equipped devices [21]. Each "thing" is uniquely identifiable through its embedded computing system but is able to interoperate within the existing Internet infrastructure. These devices are the building blocks of Wireless Sensor Networks (WSNs) in which devices use embedded sensors to keep logs of various conditions local to the device.

When looking for a platform on which to implement RLTP, Contiki was a prime candidate because it is free, open-source and has active developers who support the project and are moderately active on its community forums. Contiki is an operating system designed for the Internet of Things and is designed to connect tiny low-cost, low-power microcontrollers to the Internet [22]. Applications and functions in Contiki are written in the standard C programming language. Because C is so well documented and has been tested over time, it lends itself well towards simple and rapid development. Also, Contiki is designed for physically small devices with computing power constraints and provides excellent handling of memory allocation, IP networking and runs on a wide range of affordable devices.

The Contiki operating system package also provides numerous example programs that teach new programmers the OS's basics, as well as provide insight in how to use Contiki's protothreads which are a mixture of event-driven and multi-threaded programming mechanisms [22].

Another useful tool that Contiki provides is The Cooja Simulator. Cooja is a Contiki network simulator that provides an environment that allows developers to test and debug their applications as they run in large-scale networks. Almost all of the simulations run for RLTP were done in Cooja. One of Cooja's most important features is the ability to fully emulate specific hardware devices to test applications on. The simulation will provide hardware specific details for the specific device it is emulating. The hardware that RLTP was deployed on was the Tmote Sky, which is one of the devices that Cooja emulates. By emulating an actual hardware device in Cooja, the challenges of realizing an application on that same hardware are greatly minimized, as device specific bugs can be resolved early on in the testing phase.

Cooja, as packaged with Contiki, does not have the ability to simulate mobile nodes within a network. Due to the nature of the network that RLTP is intended for, a means for simulating *mobile* nodes was imperative. Luckily, an open-source mobility plugin has been developed that allows the user to input coordinates for a node to travel through at a given time step [23]. With the mobility plugin, RLTP was tested in numerous network environments including a simulation where two nodes pass by each other from opposite directions, and a simulation where a single node completes a circuit and communicates with several nodes as it travels. These simulations model how a vehicle might encounter another vehicle on a freeway and a vehicle driving around in a residential area and encountering several other vehicles.

2.6 The Tmote Sky

As mentioned above, the hardware device that RLTP was tested on was the Tmote Sky module. The Tmote Sky, or affectingly, "mote," is an ultra low power wireless module designed for use in sensor networks, monitoring applications, and rapid application prototyping for the Internet of Things. The Tmote Sky conforms to the IEEE 802.15.4 wireless standard to interoperate seamlessly with other devices and has many embedded sensors including humidity, temperature, and light sensors as well as providing flexible interconnection with various peripherals such as other sensors, antennae, and integrated circuits (ICs). The Tmote Sky can function as a data gatherer in isolation or in conjunction with other motes to form a mesh network. The Tmote Sky has a bandwidth of 250 kbps

over a 2.4GHz channel. That means that the motes are able to exchange data at a rate of 250,000 bytes of data per second, which is a bit lower the maximum speed of a DSL connection when it was first introduced in the early-2000s [24]. The Tmote Sky is equipped with an embedded 8MHz Texas Instruments MSP430 microcontroller with 10 KB RAM and, 48 KB of Flash memory. The mote is optimized for ultra-low current consumption as well as quick wakeup from sleep, which allows it to run for extended periods of time (on the order of months) on just 2 AA batteries; it can also be powered by USB [22].

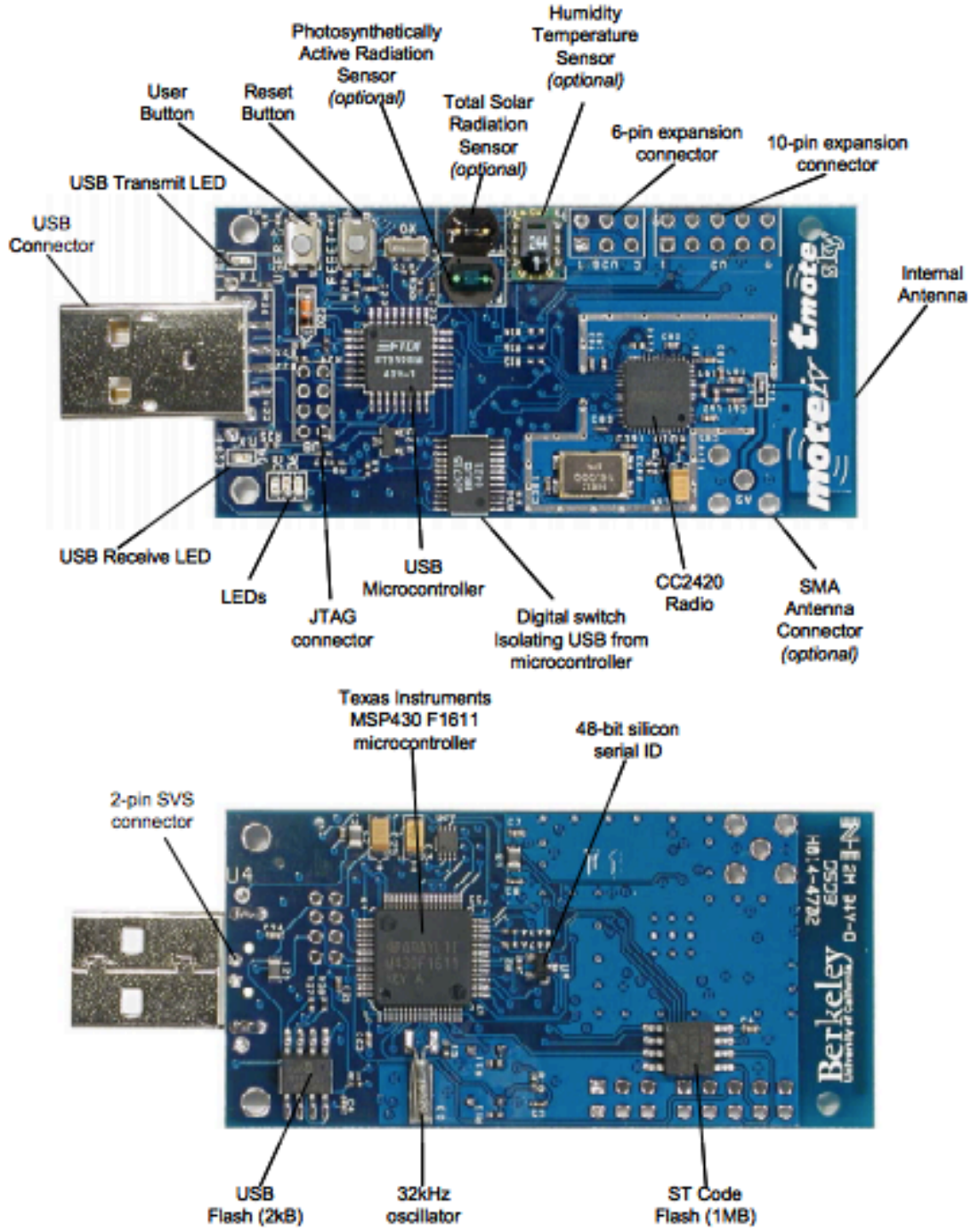


Figure 7: Front and back of the Tmote Sky module.

3 Design

3.1 Motivation

As discussed, a protocol that is suited for communications in pedestrian networks must not abide by the stringent and structured rules of traditional communication protocols such as TCP/IP [4]. While there are more lax protocols that are less robust, such as UDP, a protocol designed for a MANET must break away from single-purposed, and compartmentalized messages and embrace a hybrid model where a single frame can contain information used for discovery, control, and negotiation along with transmitted data. In addition, any routing protocol that attempts to model the complex and erratic mobility in a pedestrian or vehicular network may be too mathematically and computationally intensive for small and primitive devices [20]. Due to the brevity of encounters that exist within these MANETs, these routing protocols should be discarded in favor of a "best effort" or heuristic-based routing scheme.

In order to be deployed on devices with limited resources, the algorithms designed should be memoryless. Memoryless operation refers to the independence between present encounters and any previous encounters. An algorithm might use memory in order to store information about a previously encountered node in an effort to streamline future interactions with that same node. Since a node might encounter many other nodes, keeping records of recent encounters may consume significant memory.

Finally, because nodes may frequently disappear and reappear throughout the network, an encounter may abruptly end with another encounter beginning shortly after. In order to mitigate the challenge of a node having to keep track of whom it was communicating with, it may be advantageous to incorporate a stateless algorithm. Statelessness refers to an algorithm treats the current transmission in context to previously received messages in a given encounter. An algorithm is stateless if each message exchange is an independent transaction that is unrelated to any previous of the previous messages that were sent or received. In this way, the communication consists of independent messages, which can be properly received without any context of what messages may have been sent

before. A stateless protocol does not require the node to retain any information about the status of an encounter. On the other hand, a stateful algorithm requires storing and making decisions based on a node's internal state throughout the encounter. In a stateful algorithm it is possible for a node to become stuck in one state if it does not receive the appropriate message that may be required in order to proceed to the next state in the algorithm. For example, a server node in MERLIN has two states. In state 1, the server periodically broadcasts its discovery beacon, which contains the number of files it has to share. When it receives a request for files from the client, the server moves to state 2, where it stops broadcasting its beacon and instead shares files and waits for more requests. If the client node disappears, the server node can get stuck waiting for a request, potentially missing the opportunity broadcast a beacon to initiate another encounter. This can become a dire issue in a MANET where packets are often dropped, lost or corrupted as communicating nodes travel away from one another. In order to mitigate the wasted encounter time, MERLIN uses a timeout process (countdown timer) to reset the server's state if no requests are received for a preset duration; using timeout events to trigger a reset is a stopgap solution, however, as it still requires the protocol to idle during the encounter.

3.2 Developing Environment

The Rapid-Link Transfer Protocol is designed to operate within the Data Link Layer of the Open Systems Interconnection (OSI) model. Recall, OSI is the conceptual model that characterizes and standardizes the internal functions of a communication link by partitioning it into abstraction layers. The Data Link Layer provides functional and procedural processes that transfer data between nodes. However, as is common when developing new algorithms and protocols, RLTP was designed and tested on the Application Layer. The advantage to developing at the Application Layer is that one does not have to consider the ancillary functions provided by the Data Link Layer when designing an algorithm that is meant to operate on a lower layer of the stack. Some of these considerations include carrier sensing (e.g., CSMA), which is required if nodes are to share the available bandwidth equally, as well as error-delectating code on all received packets (e.g., CRC), which ensure that packets have not been distorted as a result of poor channel conditions, interference, or noise [7].

RLTP was developed on a no-frills IP stack called the RIME stack, which is included in Contiki as an alternative to the standard TCP/IP [25]. The RIME stack lightweight set of custom networking protocols designed specifically for low-power wireless networks. RIME was created as an alternative for use in cases where the overhead and bloat of the TCP/IP stack are prohibitive. The RIME stack provides a set of communication primitives including: single-hop unicast, single-hop broadcast, multi-hop unicast, network flooding, and address-free data collection [25]. These primitives can be used on their own or combined together to form more complex protocols. In the context of RLTP, all transmissions are "single-hop" meaning that each message is sent directly to its intended recipient and that nodes are never responsible for routing packets. A unicast transmission is a message addressed to a specific node. If a node receives a unicast message, it checks the header of the message for the destination address and, if it is the intended recipient, the message is ingested (copied to memory) and parsed. If it is not the recipient, the message is discarded, or dropped; in a routing scheme that supports multiple hops, the node may act as a relay and the message will be retransmitted to another nearby node. In contrast to unicasts, broadcast transmissions have no destination address and a received broadcast can be ingested and parsed by any nodes. For completion, a message addressed to multiple nodes is called a single-hop multicast or multi-hop multicast. Because RLTP does not employ any routing algorithms, multi-hop transmissions are not supported.

RLTP uses combinations of unicast and broadcast communication primitives for beacons and sending data. Designed for networks with highly mobile nodes, RLTP utilizes single-hop communications to transmit data between nodes within communicable range. Unicast transmissions are used for control messages and specific data requests while discovery beacons utilize broadcast transmissions. In RLTP, when filling a request for data, the block of data is packetized and sent as a broadcast transmission. The advantage to sending blocks as broadcast over unicasts is that broadcasted blocks may be received by other nearby nodes, in addition to that block's intended recipient. In this way, multiple nodes can benefit from a single transmission. Whenever a node receives a packet containing a block, it reads the block ID of the block from the packet header to if it already has the block contained in the packet. If it already has that block, the packet is dropped, but if the block is new, the packet is parsed and the block data is received.

3.3 Project Scope

Some brief comments on project scope are necessary in order to place this work in context with current advancements in IoT development, as well as address significant security concerns.

RLTP facilitates communication between two nodes within a network of many mobile nodes. While a third node may benefit from messages exchanged between other nodes, such as receiving a broadcasted block, these benefits are secondary, and not considered when evaluating the effectiveness of the protocol. Additionally, it is worthwhile to mention the current limitations of RLTP. As designed, RLTP is used to disseminate a single, sequenced dataset throughout a MANET. The dataset must be partitioned into N equally sized, numbered blocks. Feasibly, RLTP allow for more than one datasets, provided that they are also sequenced (numbered) and the number of datasets is predetermined.

It is important to note that RLTP is designed without any security considerations or constraints. This is not the ideal approach to designing a networking protocol, although it is a common practice. Naturally there are many security factors to consider when dealing with broadcast communications. Among these concerns are anonymity, authenticity, and preventing malicious activity such as attempts to alter or hijack messages and data mining. All protocols designed for MANETs face these same security concerns and a means for securing communications in these network topologies is imperative before the proliferation of IoT devices. However, IoT security is a vast topic and the subject of a large amount of research and as such, is outside of the project scope and reserved as a topic of future research.

3.4 Data Representation

As a node traverses the network, it periodically broadcasts a beacon message to alert all nearby nodes of its presence within the network. In order to facilitate the rapid transfer of data between encountering nodes, this discovery beacon in RLTP is stuffed, or combined, with information about that node's data and what data the node currently requires. When a nearby node receives the beacon,

it parses that information to determine what, if any, data can be sent back as a response. When requesting data in a network, there are many ways that a node can represent the contents of its own dataset. The most straightforward way that a node can represent the blocks of the dataset it currently has is via a bitmap. A bitmap is a one-to-one mapping of every sequenced block of the dataset. A binary 1 can indicate that the node currently has that block, while a binary 0 indicates that is does not currently have that block and must acquire that block from some other node. Given unlimited encounter time, a node could feasibly share its entire bitmap to all nearby nodes with the hopes that a nearby node has a block to share (i.e., they have 1 in their bitmap where I have a 0). The nearby node will then know which blocks are being requested and can share them accordingly. Upon receipt of a new piece of the dataset, the bitmap is updated to reflect that the block as been received by setting the appropriate 0 to a 1. Since the bitmap is a one-to-one mapping, a bitmap of every block could be tens of thousands of bytes long for a large dataset. It follows that sharing a node's entire bitmap is inefficient at best, and at worst impossible, given the short duration of encounters typical to MANETs. Representing the data as a direct bitmapping is inefficient because the time required to transmit the full bitmap could consume the entire encounter.

When designing RLTP, it was vital to find an efficient representation of the data each node requires. An effective and more common version of the bitmap representation is known as bit vector representation. In a bit vector representation, a node shares a portion of its bitmap with neighbors. The length of the bit vector is predetermined for all nodes and a subset of the bitmap is transmitted along with the sequence number (address) of the first block represented by the bitmap. An internal shift variable stores the sequence number of the most recently shared bit vector and the variable is incremented with each transmission of the bit vector to ensure that all portions of the dataset have equal opportunity to be represented. An advantage of this method is that the bit representation is of fixed length for any dataset but bit vector representation suffers from the fact that only small portions of the dataset can be represented at a time. Additionally, using a bit vector for requesting data is inherently inefficient because you are sharing information about data you already have (1's in the bit vector), which is not as useful to a node who is receiving your beacon for the first time.

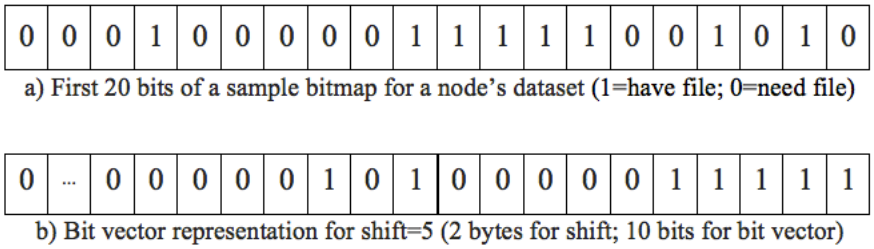


Figure 8: Bit vector representation of a bitmap: a) Sample bitmap b) Bit vector representation: 2 bytes are allocated for representing the shift variable, which is used to determine what subset of the dataset the bit vector is referencing (total length of bit vector = 26 bits ≈ 2.25 bytes).

An alternative to the bit vector representation is the range vector representation. A range vector contains tuples of addresses and/or ranges that can represent a group of consecutive blocks. For instance, an ordered paid (x,y) can represent a range of blocks from address x to address y. Alternatively, the start address and length of the group can be used in a tuple (x,y) where x is the starting address of the group of blocks and y is the number of sequential blocks that come after x. For maximal efficiency, RLTP utilizes the latter scheme because in large datasets, the address number may become increasingly large, while the lengths of the ranges tend to be smaller.

In RLTP, each node has access to its own private bitmap. From this bitmap, ranges that represent contiguous blocks of data are calculated and these ranges are grouped together as a range vector. There are two possible range vectors to compute and both were built into early designs of RLTP. The first possible range vector is made up of contiguous blocks of data that the node already has. More explicitly, this range vector is comprised of ordered pairs that include the start address of a block labeled with a binary 1 in the bitmap and the number of proceeding blocks that also are labeled with a 1. In simple English, the range vector could read as: "*I already have block X and also the next Y blocks after X.*" The second type of range vector is the dual of the first and is a vector containing ordered pairs that represent contiguous gaps of data present in the bitmap. These tuples include the start address of a block labeled with a binary 0 in the bitmap and the number of proceeding blocks that also are labeled with a 0. In simple English, this range vector of gaps would read as: "*I do not have block X nor do I have the next Y blocks after X.*" The two range vectors are considered duals of each other because if you had an exhaustive list of a node's contiguous gaps, you could generate the exhaustive list of that same node's contiguous blocks of data. Because of this duality, only one type of

range vector is needed and in order to facilitate quickly filling requests for data, the range vector containing a node's gaps is used in RLTP. That is to say, when requesting data it is more valuable to state, "*I require block X and the next Y blocks after X.*" over "*I already have…*" Finally, when constructing a range vector it is helpful to sort the gaps by the size (length) of each gap in descending order.

Start	Length
5	5
1	3
15	2
18	1
20	1

a) Largest Gaps

5	5	1	3	15	2	18	1	20	1

b) Range vector (5 Largest Gaps; [start, length])

Figure 9: Range vector representation of a bitmap. a) Tabulation of gaps from the bitmap in figure 7a, sorted by gap size in descending order. b) The range vector generated from (a). Each ordered pair in the range vector is 4 bytes long (total length of range vector = 20 bytes).

Intuitively, the range vector seems to outperform the bit vector simply by virtue of being able to represent potentially very large group of requested files in a single range (4 bytes). The bit vector, on the other hand requires 2 bytes to represent the start address (shift value) and then 1 bit to represent the status of every file afterwards. Since a byte is composed of 8 bits, a 4 byte long bit vector is capable of depicting the status of 16 blocks in the bitmap. In smaller datasets, the bit vector may outperform the range vector if the length of each gap (range) is smaller than 16 blocks. However, the range vector maintains several added advantages in that all of the data represented in a range is requested data, while the bit vector has a 50% chance of sharing information about a file that the requesting node already has. Also, since the range vector is sorted, additional gains in efficiency are achieved by prioritizing the largest of a node's gaps. To illustrate these advantages, let us study the bit vector in Figure 8. The bit vector has a size of only 26 bits, but is only able to represent 5 files that the node requires; in this case that translates to less than 20% efficiency because every 1

transmitted within the bit vector indicates that the node already has that block, which is not useful information for the purpose of file requests.

As mentioned above, all values in the RLTP range vector are stored as unsigned 16-bit integers (UInt16). 16 bits, or 2 bytes, of memory (4 bytes per range) are used so that the range vector is able to represent any address in a potentially very large bitmap. For instance, with 16 bits allocated for the start address of a gap, 2^{16}-1 = 65,535 different bitmap addresses can be represented; as such, any dataset that is to be disseminated by RLTP must be segmented into at most 65,535 equal sized blocks. Larger datasets could be shared if data addresses were represented with more bytes; UInt32 is a data structure that uses 4 bytes to store an unsigned integer and it can represent 2^{32}-1= 4,294,967,295 unique data addresses! Storing addresses and lengths as UInt32's would greatly increase the size of datasets that can be shared using RLTP, but at the cost of doubling the size of the range vector.

In RLTP, nodes declare their presence in the network by periodically broadcasting a discovery beacon. In order to "stuff" the beacon with a data request, the range vector of all gaps in the dataset is calculated and sorted in descending order as in Figure 9(a). In order to reduce the size of the beacon, only the first *K largest gaps* in the dataset are included in the beacon. The value of K is one of the variable parameters that can be optimized to increase the performance of RLTP. In the simulations conducted, a value of *K = 10 Largest Gaps* was used for a dataset divided into N = 100 sequenced blocks. RLTP transmits the largest gaps first because they contain the most information in the least amount of memory; when requesting files, the more information that can be transmitted to nearby nodes the higher the probability that a receiving node will have a file contained within the gaps to send back to the requesting node.

After determining that nodes should prioritize sharing larger gaps first, the question of how to prioritize filling requests remained. It was determined heuristically that upon receiving a nearby node's K largest gaps, a receiving node should first attempt to fill requests from the smallest of the gaps in an effort to preserve larger gaps for future encounters with other nodes. For example, upon receipt of a beacon, if a node has a single block to share from a gap of length 20 and also a single

block to share from a gap of length 5, the node should send the block from the gap of length 5 so as to avoid partitioning the gap of length 20 into two smaller sized gaps. In this way, the broadcasting node's larger gaps will remain intact for future encounters, thereby increasing the likelihood that it will have successful encounters with other nodes in the network once the current encounter has ended.

3.5 RLTP Overview

Before diving into the intricacies of RLTP's design, it is helpful to first understand the protocol's procedure at a high level. While there are many scenarios and initial conditions that define each encounter, there is a general communication flow that occurs when nodes meet under more favorable conditions. What follows is one example of a typical RLTP encounter:

- Node A transmits its discovery beacon, which contains the *K largest gaps* that are stored in its range vector.
- Node B receives the discovery beacon and checks the K gaps for blocks it can share.
  - The block numbers of these blocks are stored in a list called the Push Queue.
- Node B broadcasts the first block on its Push Queue along with the *M largest gaps* (M<L) stored in its range vector.
- Node A receives the block from Node B.
  - Node A's bitmap is updated to reflect the new file.
  - Node B's *M largest gaps* are parsed and available blocks are added to the front of Node A's Push Queue.
- Node A transmits the first block on its Push Queue it in a packet that also contains the *M next largest gaps* contained in Node A's Push Queue.
- Node B receives the block from Node A.
  - Node B's bitmap is updated to reflect the new file.
  - Any blocks it has in the received *M next largest gaps* are appended to the top of the push queue. Because these M gaps are the *next largest gaps*, they must be smaller gaps than the initial *K largest gaps* that were initially sent in the beacon. Node A's largest gaps can be preserved for future encounters by sharing blocks from smaller gaps with higher priority.
- Node B transmits the next block on the Push Queue along with its *M next largest gaps*.

This process can continue until the encounter ends due to the nodes falling out of range or due to one of the node's Push Queues becoming empty. In the latter case, the node unicasts a data request to indicate that it has no blocks to share. The data request is similar in structure to the discovery

beacon and contains the node's *K next largest gaps*. Given a long enough encounter, a node ultimately share all of the gaps contained in its range vector. In these cases, it may be in the node's best interest to terminate the encounter. Below is a possible continuation of the above example encounter. Recall, Node B has just transmitted a block packet containing a requested file and Node B's *M next largest* gaps. Let us also assume that Node A's Push Queue is currently empty:

- Node A receives the block from Node B, and Node A's Push Queue is empty.
    - Node A's bitmap is updated to reflect the new file.
    - Node A parses the *M next largest gaps*, which have just arrived with the most recent block, but has no blocks contained in the request.
- Node A has no files to share and sends a data request to Node B containing its *K next largest gaps*.
    - Node A has now shared all of its gaps with Node B.
- Node B receives the data request from Node A.
    - Node B appends any files it can share from Node A's *K next largest gaps* to its push Queue
- Node B transmits the next block on the Push Queue along with its *M next largest gaps*.
- Node A receives the block from Node B
    - Node A's bitmap is updated to reflect the new file.
    - Node A parses the *M next largest gaps* that arrived with the most recent block, but still has no blocks contained in the request.
- Node A sends an empty request to Node B because it has no files to share, and it has already shared the entire range vector with Node B.

At this point, the duration of the encounter has likely far surpassed the expected duration, but Node B may still have files that it can share with Node A. With each shared block, Node B is also sharing its next *M largest gaps*, and as a result Node A may be able to share more files with Node B. Whenever Node A does not have blocks to share, Node A will continue to send empty requests in response to blocks from node B, which act as an ACK message. This may continue until both nodes have shared all of their gaps and both nodes' Push Queue are empty at which time the nodes will terminate the encounter.

The state diagram below shows RLTP's logic flow. Then, the following subsections detail various design choices, and the RLTP beacon and block packet structures are laid out.

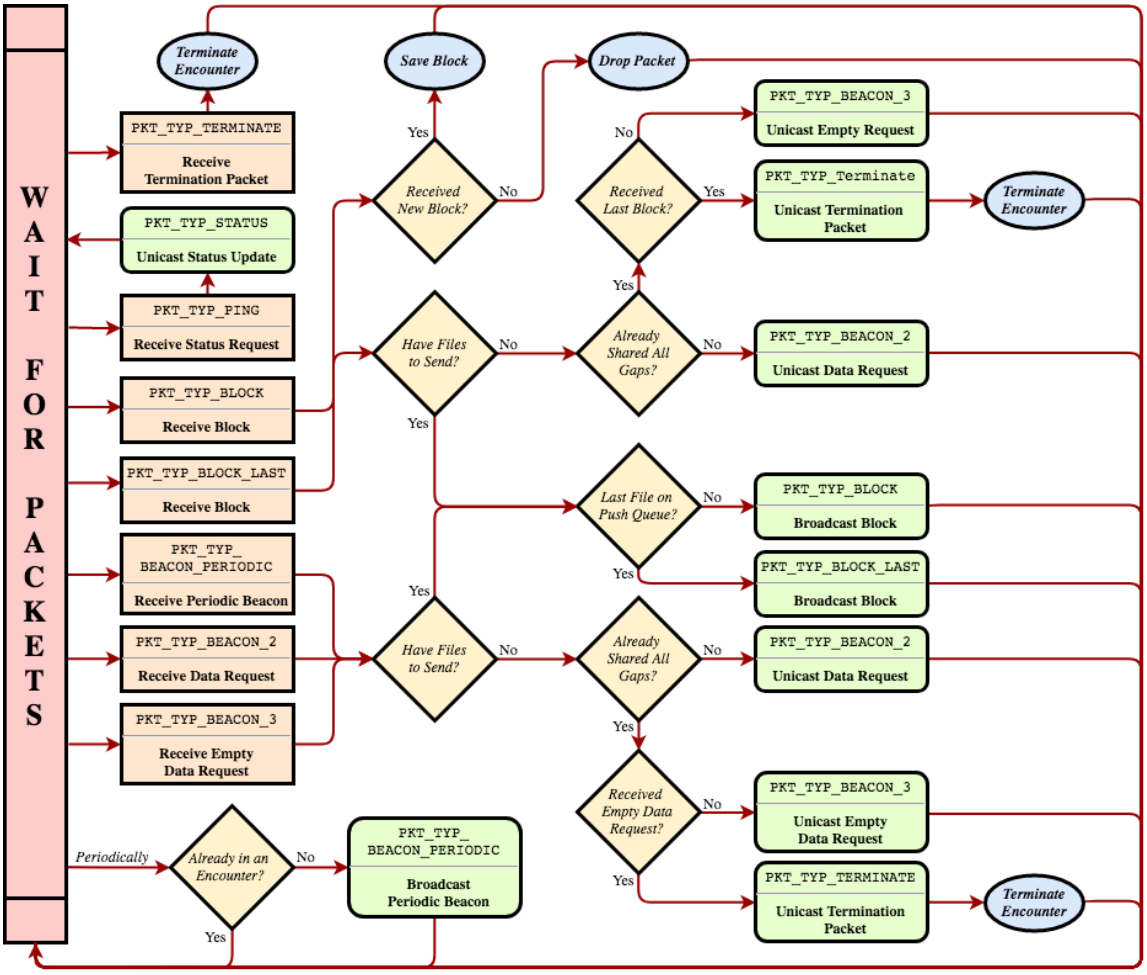


Figure 10: RLTP state transition diagram.

3.6 Beacon Packet Structure

Once the range vector of gaps is computed and sorted the K largest gaps are selected and inserted into the beacon. The frame begins with a header, which is required by the RIME stack and includes channel information, device identification, etc. This header is not directly a part of the RLTP beacon but is appended to the beacon frame at a lower level of the RIME stack. The RLTP beacon frame is structured as follows:

The first component of the RLTP beacon is the packet type. Nodes use this field in order to differentiate between a packet containing data (e.g., `PKT_TYP_BLOCK`) and a beacon packet, which contain requests for data. Beacon packets can have one of three distinct packet types:

PKT_TYP_BEACON_PERIODIC, PKT_TYP_BEACON_2, and PKT_TYP_BEACON_3. Each type of beacon is transmitted under specific circumstances such that the receiving node is able to make certain assumptions about the sender based on the type of beacon it receives. RLTP uses the packet type in order to maintain a stateless and memoryless algorithm.

PKT_TYP_BEACON_PERIODIC is the name of the packet type for a standard discovery beacon, which is broadcasted periodically at a predefined interval. When simulating RLTP, a beacon interval of 1-2 seconds was used. Before attempting to broadcast a discovery beacon, a node must first check an internal flag to determine if it is currently in an encounter. Recall, the memoryless property of RLTP prevents the node from "remembering" what messages it has previously sent. RLTP uses flags stored as Boolean variables, which can be raised (flag=true) or lowered (flag=false), in order to allow the node to keep track of certain occurrences without having to keep a log (memory) of every message that is sent and received. If the encounter flag is not raised, the beacon is immediately broadcasted. If the flag is raised the node infers that it must have previously been in communication with another node and now must determine if it is *still* in the encounter by comparing the number of blocks it currently has to the number of blocks it had before its last attempt to transmit its periodic beacon (1-2 seconds prior). If it has not received any new blocks since the last beacon attempt, then the encounter is assumed to have terminated and the periodic beacon is transmitted. This ensures that at most only one discovery beacon will be "missed" when an encounter inevitably ends.

The PKT_TYP_BEACON_2 packet type has the same structure as the periodic discovery beacon but is used for requesting blocks once an encounter has already started. The PKT_TYP_BEACON_2 packet can only be sent in the event that a node's Push Queue is empty. This reply primarily serves as a data request, but also operates as an implicit acknowledgement (ACK) of the previously received packet. The PKT_TYP_BEACON_2 packet can be sent in response to any type of packet (including a PKT_TYP_BEACON_PERIODIC packet if the receiving node has nothing in contained in the gaps of a node's discovery beacon) and allows the receiving node to confidently infer that, given the current gaps shared, the broadcaster has nothing to currently offer the receiver. Again, this functionality serves to maintain the stateless algorithm, as the receiver of a PKT_TYP_BEACON_2 has no memory of

any previous encounters with the other node and would otherwise "not remember" that it had previously communicated with the other node.

The third, and final type of beacon, `PKT_TYP_BEACON_3` is sent for a similar reason as a `PKT_TYP_BEACON_2` with the added constraint that the `PKT_TYP_BEACON_3` can only be sent if a node has already shared all of the gaps in its range vector. The receiver of a `PKT_TYP_BEACON_3` can infer that the broadcaster of the `PKT_TYP_BEACON_3` has an empty Push Queue and has already shared all of its gaps. Upon receipt of this type of packet, a node may continue to share blocks in its Push Queue along with its remaining gaps. Eventually however, both nodes' Push Queues will be depleted and when a Node attempts to send a `PKT_TYP_BEACON_3` packet in response to a `PKT_TYP_BEACON_3` packet, a `PKT_TYP TERMINATE` packet is transmitted instead, which ends the encounter.

Following the packet type is the node's current percent of the dataset. In initial implementations of RLTP this metric was used to determine which node currently has a higher percentage of the dataset. One purpose of transmitting the percent was to achieve maximal dissemination throughout the network *as a whole*. That is to say, by prioritizing the sharing of data with nodes of a lower percentage, a higher overall average percentage can be achieved (with respect to all nodes in the network) with a minimized variance. To that end, RLTP attempted to send files to the node with the lower percent. This was achieved by having a node reply to a discovery beacon with a `PKT_TYP_BEACON_1` (an antiquated packet type) if it had a lower percent than the broadcasting node. This "re-beaconing" would reset the encounter in a way that the receiver of a `PKT_TYP_BEACON_1` is the node with the least percent. However, this design choice was abandoned early on as it added unnecessary control transmissions.

The final part of the beacon contains the K largest gaps. As described previously, the gaps are ordered from largest to smallest and have 2 bytes allocated for the start address and 2 bytes allocated for the length of the gap (offset). If a node has less than K new gaps to share, the gap address and offset are both assigned the value -1, indicating that these values are null. Additionally, all gap values in a `PKT_TYP_BEACON_3` are set to -1 by definition.

With a single byte allocated to packet type, and another byte for percentage, and at 4 bytes per gap, the beacon packet (regardless of beacon type) will be 2+4*K bytes. For K = 10, the packet size will be a mere 42 bytes. The size of the RLTP beacon frame is about half of the size of the WiFi (IEEE 802.11) discovery beacon, which is broadcasted periodically by every wireless router. The WiFi beacon frame is 61-83 bytes long depending on the size of the network's name, or Service Set Identifier (SSID), which can range from 1 bye to 32 bytes in size [4].

Packet Type	% Complete	Gap 1 Start Address	Gap 1 Offset	Gap 2 Start Address	Gap 2 Offset	•••	Gap *K* Start Address	Gap *K* Offset	**Total** ***(K = 10)***
1 byte	1 byte	2 bytes	2 bytes	2 bytes	2 bytes	(28 bytes)	2 bytes	2 bytes	**42 bytes**

Figure 11: RLTP beacon packet structure (K = 10).

3.7 Block Packet Structure

As described, block packets not only contain block data, but also contain additional information about a node's gaps. This additional information is critical to providing the maximum likelihood that a node's Push Queue will not be empty when there is still opportunity to share data. The RLTP block frame begins with the same RIME header as the RLTP beacon frame described in the previous subsection and the remainder of the RLTP block frame is structured as follows:

Similar to the beacon frame, the block frame begins with the packet type. A single byte of memory is allocated for the packet type, which has two possible values: `PKT_TYP_BLOCK`, which is the default packet type for all blocks, and `PKT_TYP_BLOCK_LAST`. As the name suggests, `PKT_TYP_BLOCK_LAST` is the packet type RLTP uses whenever it transmits the last block in its Push Queue. Under most circumstances, RLTP treats blocks `PKT_TYP_BLOCK` and `PKT_TYP_BLOCK_LAST` in the same manner. As described in the precious subsection, if a node receives a block when their Push Queue is empty and also has shared all of the gaps in its range vector, then the would transmit an empty data request (`PKT_TYP_BEACON_3`), indicating that it has no blocks and no new gap data to share. However, if the block received is of the `PKT_TYP_BLOCK_LAST` type, then the node can correctly assume that the Push Queue of the block's sender is also empty (and will remain empty because there is no new gap information to be provided). In this case, instead of transmitting an empty data request, it is most

efficient to terminate the encounter, as the pair of nodes have exhausted efficient communication and would be better served finding a new node to exchange data with. In order to terminate the encounter, the receiving node responds to the `PKT_TYP_BLOCK_LAST` with a `PKT_TYP_TERMINATE`, which tells its neighbor that it should end the encounter and both nodes lower their encounter flags and begin to broadcast their discovery beacons.

Following the packet type is the node's percent of the dataset, which is used primarily for diagnostic and logging purposes. While this may seem like an inefficient use of a byte, due to the way the C Programming Language handles memory addresses, the byte in question would have been wasted regardless. In C, data types do not normally start at arbitrary byte addresses in memory. Rather, each type (except char) has an alignment requirement; 2 byte shorts must start on an even address, 4 byte integers or floats must start on an address divisible by 4, and 8 byte longs or doubles must start on an address divisible by 8 [26].

The next part of the RLTP block frame consists of the block number, stored as a 2 byte unsigned integer (uint8_t), followed by the block's payload (data). In order to keep packet sizes to a minimum, the payload is a mere 80 bytes in size, which means that the dataset must be divided into N 80-byte sized blocks.

The final part of the block frame contains the *M next largest gaps*. As described previously, the gaps are ordered from largest to smallest and have 2 bytes allocated for the start address and 2 bytes allocated for the length of the gap (offset). If a node has less than M new gaps left to share, the gap address and offset are both assigned the value -1, indicating that these values are null. For simulations of RLTP, a value of M = 5 was used. At 4 bytes per gap, the size of the M gaps will be 5*M bytes, or 20 bytes.

With a single byte allocated to packet type, another byte for percentage, 2 more bytes for the block number and 80 bytes for the block's payload, and 20 bytes for the M gaps, the packet size for a block will be 104 bytes.

Packet Type	% Complete	Block Number	Block Data	Gap 1 Start Address	Gap 1 Offset	•••	Gap *M* Start Address	Gap *M* Offset	Total *(M = 5)*
1 byte	1 byte	2 bytes	80 bytes	2 bytes	2 bytes	(12 bytes)	2 bytes	2 bytes	**104 bytes**

Figure 12: RLTP block packet structure ($M = 5$).

4 Results & Discussion

4.1 Range Vector vs. Bit Vector

A main component of RLTP, which sets it apart from other transfer protocols, is the highly efficient representation of data contained in each beacon (and block). The range vector approach utilizes a node's contiguous gaps of data to quickly share a large portion of that node's data needs in order to facilitate a rapid transfer of data. Unlike the bit vector, the range vector only represents data that a node is missing and currently requires. However, the memory required to share ranges is much more than the memory required by the bit vector. When attempting to compare the two representations, one must also consider the range vector's ability to share a potentially large portion of the node's bitmap (or possibly all of it) in a single transmission, while still maintaining a beacon size akin to those of other wireless protocols.

The range vector requires 4 bytes of memory for each gap, totaling to 40 bytes (for K = 10), or 320 bits. Immediately, it follows that for datasets of size N $\leq$ 320 it can be more memory efficient to simply share a node's entire bitmap. An equal sized byte bit vector would require 2 bytes for the bit vector's start address, and 38 bytes, or 304 bits, used to share a portion of the node's bitmap. Thus, under some conditions with small enough datasets, the bit vector may outperform the range vector. Therefore, an analysis of which method performs better was required in order to determine if the efficiency of the range vector outweighed its larger memory requirements.

To illustrate the advantages of the range vector representation two nodes named Alice and Bob were simulated in MATLAB. Alice was initialized with 70% of a dataset and Bob was initialized with 40% of the dataset. Since there may be some overlap in Alice's and Bob's subsets of the dataset the maximum amount of available data is the union of Alice and Bob and is expressed as the maximum combined data of the two nodes. In one simulation (depicted in Figure 12), the maximum percent that the nodes can achieve with the available information is approximately 82% and is represented by the purple dotted line. To clearly demonstrate the difference between the two representations, a modified version of RLTP was used, which prioritized sending data to the node with the lower

percent of the file (Bob). This modification makes RLTP work similarly to MERLIN by creating a "catch up" portion of the encounter where one node acts as a server, which fills requests by the client until the two have the same percent of the file. Both the bit vector and range vector representations were simulated with the modified RLTP and the "catch up" period can be used to compare the ability of each method to rapidly fill requests but minimizing the number of data requests transmitted.

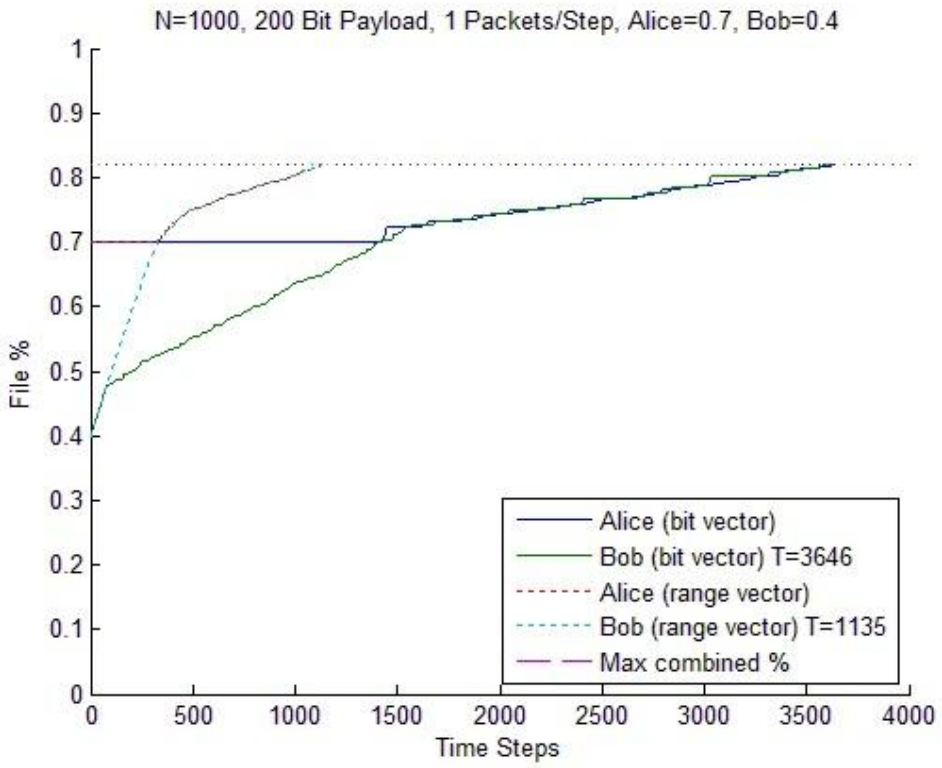


Figure 13: Performance of bit vector and range vector representations. The range vector implementation converges to the maximum combined percent almost 2,000 time steps faster than the bit vector approach.

In the simulation shown in figure 12, each time step corresponds to the transmission of a single packet, which could be either a beacon packet (data request) or a block packet (transmission of data). The encounter duration was assumed to be infinite to allow both nodes to attain the maximum combined percent of the dataset. The simulation can be reasonably divided into two distinct phases: First, the "catch up" period, where Bob must get to the same percent as Alice, followed by the "exchange" phase, in which Alice and Bob go back and forth requesting and sharing data until both

have reached the combined maximum percent. Using a range vector for requests, Bob starts with 40% of the dataset and catches up to Alice's 70% in approximately 300 time steps (dashed blue line). For a dataset divided into N = 1000 sequenced blocks, this 30% increase corresponds to Bob receiving 300 blocks. This shows that the initial range vector with Bob's largest gaps provided Alice all of the information she required to help Bob catch up to 70%. The same could not be stated, however, for the bit vector simulation. It takes Bob almost 5 times longer to reach 70% when using the bit vector (solid green line) because there is a lower probability of Alice having files contained within a given subset of Bob's bitmap. In the second phase of this simulation, both nodes act as server and client, prioritizing the node with the lesser percent. At this point the lines representing Alice and Bob becomes hard to distinguish as both lines rise in lockstep towards 82%. The performance of both representation methods will vary significantly depending on multiple initial conditions such as the number of blocks of the dataset, Alice and Bob's starting percentages, and the maximum combined percent of the two nodes. Overall however, the range vector outperformed the bit vector representation in a simulation environment, often converging to the combined maximum percent over 2,000 steps faster.

4.2 Selection of K & M

All RLTP data requests are sent in the form of gap. The request from a node with 0% of the dataset would come in the form of a single gap, with the starting address of 0, and a length of N. Thus, the minimum number of gaps required to share the entirety of a node's dataset is 1. As a nodes percentage increases, the number of gaps required to share the entirety of its bitmap first increases, and then eventually decreases as the node attains 100% of the dataset. The maximum number of gaps that would ever be required is expressed as $N \div 2$, and only occurs if a node has 50% of the file such that the 1's and 0's in the bitmap alternate (i.e., if the node has just all of the odd (or even) numbered blocks). For a dataset divided into N = 1,000 blocks, this would equate to a maximum of 500 gaps.

When transmitting a data request it is beneficial to send as many gaps as is feasible in order to maximize the likelihood that a node will be able to fill the data request, provided that the packet size

does not become increasingly big. For a beacon, K = 10 was used to create a beacon of size 42 bytes and M = 5 was used to create a block packet of size 104 bytes.

Simulations in MATLAB were conducted in order to assess the performance of each additional gap for incremental values of K. The likelihood that a node was able to provide at least 1 file from the requested ranges (i.e., the Push Queue of the receiving node was not empty) is plotted on the y-axis against each value of K, which is plotted on the x-axis. The likelihood is computed as the percent of iterations that a node's Push Queue was not empty. For this simulation, 10,000 iterations (5,000 per node) were tested for each value of K (ranging form K = 1 to K = 20). In each simulation, the two nodes are initialized with a random subset of a dataset which is divided into N = 1,000 blocks and then the node shares K gaps and the other node generates a Push Queue based on the files it can share.

Figure 13 shows the results of the simulation. Across all iterations, when only the largest gap was transmitted, the receiving node was able to provide at least 1 file in over 85% of encounters! This number jumps to over 90% when the two largest gaps are sent.

There are diminishing returns, however, when adding more and more gaps to a data request. The primary limiting factor in MANETs is short encounter duration and nodes must be careful not to over-request information. Indeed, two nodes rarely will maintain connection long enough to transmit every file in their respective Push Queues. As shown in Figure 6, MERLIN demonstrated these diminishing returns as well, showing increased portions of encounters being spent requesting data rather than sharing it [20]. Notably, an asymptotic behavior is observed as initial conditions can be quite unfavorable (e.g., one node may have a very small portion, or none of the dataset). As such, there is no value of K that will yield a success rate of 100%. Another run of this simulation, for K values between 1 and 50, validates this asymptotic behavior and shows that no significant increase in performance is seen between K = 20 and K = 50. A plot of these results can be found in the Appendix.

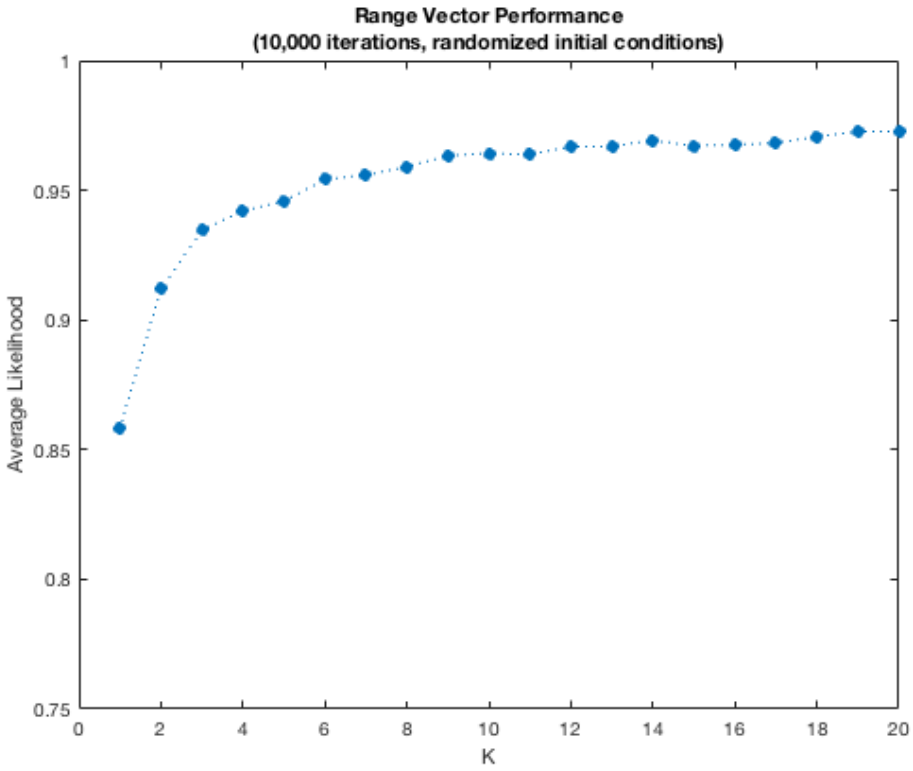


Figure 14: Range vector performance as a function of K.

K	Likelihood	K	Likelihood
1	*85.8%*	11	*96.4%*
2	*91.2%*	12	*96.7%*
3	*93.5%*	13	*96.7%*
4	*94.2%*	14	*96.9%*
5	*94.6%*	15	*96.7%*
6	*95.4%*	16	*96.8%*
7	*95.6%*	17	*96.8%*
8	*95.9%*	18	*97.1%*
9	*96.3%*	19	*97.3%*
10	*96.4%*	20	*97.3%*

Table 1: Range vector performance as a function of K,

Initially, a value of K = 20 was selected for the number of gaps to include in the discovery beacons. However, as the performance increase from K = 10 to K = 20 is only ~1%, the value of K was reduced to K = 10, effectively halving the size of the discovery beacon from 82 bytes to 42 bytes. Once the beacon is received, the encounter begins; the blocks that are to be exchanged must not be bloated with extraneous information because packet transmission time increases linearly with packet

size and additional computation time is required with each additional gap a node must parse. All of these factors were considered when selecting a value for M, the gaps sent with each block packet.

Until now the focus of data requests have been focused on a node's receipt of a discovery beacon containing a request in the form of the beaconing node's *K largest gaps*. In this scenario, the receiver learns a lot of about the beaconing node and is usually able to respond with both a block and its own data request in the form of its *M largest gaps*. Ensuring that the beaconing node receives enough information in the *M Largest Gaps* is imperative because it this node must populate its own Push Queue for an efficient encounter. In order for exchange data at peak efficiency, both nodes must have at least one file to share at each phase of the encounter. With similar performance to K = 10 gaps, sharing M = 5 gaps has a ~95% likelihood that the beaconing node will be able to share a block of its own in response to the block it receives.

4.3 Cooja Simulations

The Cooja simulator was used to test RLTP in various situations. Cooja is a network simulator package provided with Contiki that "allows large and small networks of Contiki motes to be simulated. Motes can be emulated at the hardware level, which is slower but allows precise inspection of the system behavior, or at a less detailed level, which is faster and allows simulation of larger networks" [22]. In tests, Cooja's hardware emulation was used to simulate an RLTP deployment on Tmote Sky motes. Two such topologies are discussed below:

In the first simulation, a typical RLTP encounter is depicted. Two nodes encounter each other on antiparallel paths (i.e., traveling in opposite directions) as might occur when two vehicles drive past each other on the road. The nodes periodically broadcast their discovery beacons until the nodes are within communicable range; eventually, one node receives a discovery beacon and the encounter is initiated. The simulated nodes travel at 15 meters/second (~34 mph) and have a communication range of 50 meters (~165 ft.). The speed is what might be common for vehicles traveling on a local road and 50 meters is a conservative value for the communicable range of the Tmote Sky [27,28]. For all RLTP simulations, the dataset is divided into N = 100 blocks and nodes are initialized with a randomized dataset: Node A was initialized with 61% of the dataset and Node B was initialized with

30% of the dataset. The beacon interval was set to 1.5 seconds and the nodes pass each other at a distance of 4 meters (~13 feet) apart away, which is approximately the width of a lane in the U.S. Interstate Highway System [29].

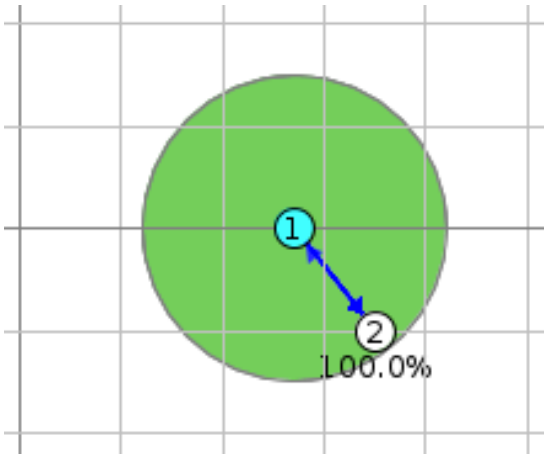


Figure 15: Cooja simulation of a typical RLTP encounter between two mobile nodes: a green circle marks the communicable range of a node.

On average, this simulation resulted in each node being able to share ~3.5 blocks, culminating to an average of ~7 files being shared during the encounter. Results are tabulated for 20 runs of the simulation below in Table 2 and extended simulation data can be found in the Appendix.

Run	Encounter Time *(sec)*	Files Received by Node A	Files Received by Node B	Total Files Exchanged	Total Messages Exchanged	Efficiency
1	1.7	3	3	6	7	*85.7%*
2	1.9	4	4	8	9	*88.9%*
3	1.6	3	2	5	6	*83.3%*
4	0.6	0	1	1	2	*50.0%*
5	1.6	3	4	7	8	*87.5%*
6	1.7	4	4	8	9	*88.9%*
7	1.7	3	4	7	8	*87.5%*
8	1.8	4	4	8	9	*88.9%*
9	2.0	4	4	8	9	*88.9%*
10	1.6	3	4	7	8	*87.5%*
11	2.0	4	4	8	9	*88.9%*
12	1.6	4	3	7	8	*87.5%*
13	1.7	4	3	7	8	*87.5%*
14	1.9	4	4	8	9	*88.9%*
15	2.3	4	5	9	10	*90.0%*
16	0.5	1	0	1	2	*50.0%*
17	2.1	4	4	8	9	*88.9%*
18	1.7	3	4	7	8	*87.5%*
19	1.8	3	4	7	8	*87.5%*
20	1.5	3	3	6	7	*85.7%*
Average	**1.7**	**3.4**	**3.6**	**6.7**	**7.6**	***84.0%***

Table 2: Simulation results for a typical RLTP encounter between slow vehicles.

If TCP was used in a similar environment the first 3 transmissions in the encounter would have been standard handshake messages, and a data request would only be sent on the fourth transmission. Since ~8 messages were exchanged during the encounter, assuming similar conditions to the RLTP simulation, TCP would have required 3 messages to establish a connection and 1 message for a data request resulting in just 4 messages to share blocks. Compared to RLTP's 7 shared files, this is a 175% performance improvement over TCP and the result demonstrates RLTP's ability to outperform traditional transfer protocols and facilitate rapid transfers of data in networks that suffer from limited encounter duration.

The simulation was then repeated for vehicles moving at the faster speed of 30 meters/second (~67 mph) to model automobiles driving on a freeway and is otherwise identical to the previous simulation [5]. The results are tabulated below and extended simulation data can be found in the Appendix:

Run	Encounter Time *(sec)*	Files Received by Node A	Files Received by Node B	Total Files Exchanged	Total Messages Exchanged	Efficiency
1	*0.80*	2	1	3	4	*75.0%*
2	*1.10*	2	2	4	5	*80.0%*
3	*1.10*	2	2	4	5	*80.0%*
4	*0.40*	1	0	1	2	*50.0%*
5	*0.50*	1	1	2	3	*66.7%*
6	*0.40*	1	0	1	2	*50.0%*
7	*0.80*	1	1	2	3	*66.7%*
8	*1.00*	2	2	4	5	*80.0%*
9	*0.80*	2	1	3	4	*75.0%*
10	*N/A*	0	0	0	0	*N/A*
11	*1.20*	2	3	5	6	*83.3%*
12	*1.20*	2	2	4	5	*80.0%*
13	*N/A*	0	0	0	0	*N/A*
14	*0.70*	0	1	1	2	*50.0%*
15	*0.70*	2	1	3	4	*75.0%*
16	*0.90*	2	1	3	4	*75.0%*
17	*1.20*	3	2	5	6	*83.3%*
18	*0.90*	1	2	3	4	*75.0%*
19	*1.00*	2	2	4	5	*80.0%*
20	*0.50*	0	1	1	2	*50.0%*
Average	*0.8*	***1.8***	***1.6***	***2.9***	***3.7***	***70.8%***

Table 3: Simulation results for a typical RLTP encounter between fast vehicles.

As expected, the average encounter time for vehicles traveling 30 meters/second is roughly half of the average encounter time measured for vehicles traveling at 15 meters/second. Under these harsher conditions, RLTP is still able to facilitate data exchange with an efficiency of ~70%. Here, efficiency is defined as the number of transmissions that actually include shared data (i.e., a message that is not just a beacon or data request). Note that 100% efficiency cannot be achieved because the first transmission is always a discovery beacon.

While file transfers were not completed in some runs of the simulation, there were runs where RLTP was able to share a file in encounters that lasted less than half a second, which demonstrates that RLTP is a viable transfer protocol under extremely short encounter times characteristic of vehicular networks. Also, if one were to simulate this scenario with TCP, it is reasonable to expect that very few runs would result in successfully shared files because on average only 4 messages were exchanged

between nodes. TCP would consume the entire encounter with its handshake procedure and a data request making RLTP perform infinitely better than TCP under these conditions.

A third iteration of this simulation was conducted to model a pedestrian network. In this simulation, the nodes passed each other on antiparallel paths spaced 3 meters apart (~10 feet) at a speed of 1 meter/second (2.2 mph), which is close to the average walking speed of 1.4 meters per second [30]. The communicable range was reduced to just 15 meters to account for stricter constraints on power consumption characteristic to small and power-constrained devices that might be used in pedestrian networks [3]. With a significantly longer encounter duration, nodes in a typical pedestrian encounter were able to exchange ~41 blocks using RLTP. The pedestrian simulation was also repeated 20 times, and the simulation data is tabulated below along with a comparison to the vehicle simulation:

Run	**Encounter Time *(sec)***	**Files Received by Node A**	**Files Received by Node B**	**Total Files Exchanged**	**Total Messages Exchanged**	**Efficiency**
1	12.8	12	27	39	56	*69.6%*
2	12.5	12	30	42	60	*70.0%*
3	12.6	12	21	33	44	*75.0%*
4	12.3	12	27	39	54	*72.2%*
5	12.7	12	32	44	65	*67.7%*
6	12.4	12	27	39	55	*70.9%*
7	10.8	12	27	39	55	*70.9%*
8	12.1	12	25	37	52	*71.2%*
9	12.3	12	33	45	69	*65.2%*
10	12.2	12	26	38	54	*70.4%*
11	12.7	12	34	46	70	*65.7%*
12	12.5	12	27	39	55	*70.9%*
13	12.2	12	32	44	65	*67.7%*
14	11.9	12	27	39	55	*70.9%*
15	12.3	12	32	44	66	*66.7%*
16	12.0	12	26	38	53	*71.7%*
17	12.7	12	30	42	62	*67.7%*
18	12.1	12	28	40	56	*71.4%*
19	12.5	12	32	44	64	*68.8%*
20	12.3	12	29	41	59	*69.5%*
Average	**12.3**	**12.0**	**28.6**	**40.6**	**58.4**	***69.7%***

Table 4: Simulation results for a typical RLTP encounter between pedestrians.

While the average encounter time for the pedestrian network simulation is over 15 times longer than the vehicular network simulation at 30 meters/second, their efficiencies are comparable. The reason that RLTP appears to perform worse in a pedestrian network is partially the result of the encounter being *too long.* As is depicted in Table 4, Node A always received exactly 12 blocks from Node B. As noted above, Node A and Node B were initialized to 61% and 30% respectively. When the encounter is sustained for an extended period, as is the case in a pedestrian network where nodes move more slowly, Node A is able to receive all of the files in Node B's Push Queue. After Node B has shared all of the files it can, it proceeds to send data requests to Node A in lieu of blocks. Figure 10 illustrates that this behavior is by design and the result is a lower efficiency (i.e., a higher fraction of messages exchanged are control messages). Importantly, the tabulated efficiencies are highly dependent on the percent of the dataset each node's initialized with. That is to say: if Node B had more files to share with Node A, then it wouldn't have sent as many control messages and the efficiency would have been improved.

As described previously, if one were to attempt to use TCP in a pedestrian network, the 3 additional control messages from the handshake would further degrade the efficiency. Under identical conditions to this simulation, the number of files exchanged would as much as 3 less than the simulation average of 40.6 files. For the above simulation, the number would be less than 3 because Node B did not always have a file to send and therefore the additional control messages required by TCP would not have made as much of an impact. Thus, we can compute the maximum performance increase of RLTP relative to TCP assuming that both nodes always have a file to share. Under these conditions the TCP encounter would only have succeeded in sharing 37.6 files compared to RLTP's 40.6 (due to the handshake replacing 3 of the files shared by RLTP), which equates to a 108% performance improvement over TCP.

Simulation Type	**Node Speed *(m/s)***	**Communicable Range *(m)***	**Encounter Time *(sec)***	**Blocks Exchanged**	**Efficiency**	**Performance Increase from TCP**
Vehicles	15	50	1.7	6.7	*84.0%*	*175%*
Vehicles	30	50	0.8	2.9	*70.8%*	∞
Pedestrians	1	15	12.3	40.6	*69.7%*	*108%*

Table 5: Comparison of simulation results.

Finally, while RLTP was designed to optimize an encounter between two nodes, it is important to see how the protocol functions when more than two nodes encounter each other. A final simulation was conducted to demonstrate how RLTP functions when more than 2 nodes are in range of each other. Because RLTP transmits all data blocks as broadcast messages, a third node may be able to benefit from an encounter occurring between two other nodes. In figure 15, Node 3 is able to capitalize on the encounter occurring between Node 1 and Node 4; Node 1 fulfills a data request for Node 4 and Node 3 is able to receive that same block. Provided that Node 3 does not already have a copy of that data block, it saves the block and the node's bitmap and largest gaps are updated to reflect the newly received data.

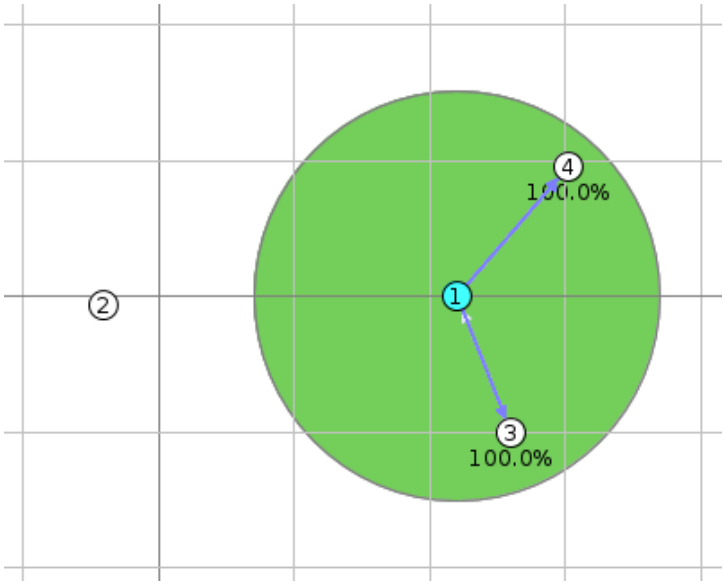


Figure 16: Cooja simulation involving four mobile nodes: RLTP enables Node 4 to benefit from the data being shared between Node 1 and Node 3.

4.4 Deploying The Tmote Sky

In order to determine the feasibility of deploying RLTP in the real world, The Rapid-Link Transfer Protocol was flashed on to two physical Tmote Sky motes, which were deployed in a pedestrian network environment.

The motes successfully were able to exchange data in situations where two pedestrians walked past each other, and also when a single pedestrian ran past a stationary mote. Due to the fact that the motes only display output when connected to a computer, real-time viewing of encounter data is limited. To gain insight from live encounter data, a third mote, connected to a computer, was used as a status node. The status node was programmed to ping nearby motes with a status update request (`PKT_TYP_PING`) to which nodes would respond with their current percent of the dataset (`PKT_TYP_STATUS`). In this way the progress of the hardware motes running RLTP could be determined.

The real world deployment performed similarly to the Cooja simulations in environments with limited obstacles and in locations with controlled interference. However, in real world environments, good channel conditions cannot be guaranteed and packet loss was prevalent in areas where there was a high volume of other wireless electronics, and in locations where there was not a direct line of sight between the two communicating motes.

Finally, a real world network topology was created where two communicating nodes would fall out of range and lose connection and one of those nodes moved on to encounter a new, third node. The goal was to determine if the second encounter would be initiated successfully following the dropped connection with the first node. As designed, the second encounter was initiated following a missed discovery beacon attempt, demonstrating that RLTP functions as a stateless algorithm that is able to recover from encounters when they are abruptly disconnected.

4.5 Applications of RLTP

The Rapid-Link Transfer Protocol is, at its core, a method of disseminating a common dataset among nodes that move freely in an environment prone to encounters of unknown duration. In the context of vehicular and pedestrian networks, RLTP may be used as a support to current methods of data dissemination including peer-to-peer (P2P) file sharing, (e.g., BitTorrent), or a more simple single client download of a file from a server. That is to say, RLTP can function as a means of data offloading, wherein applications may obtain available data from nearby nodes in lieu of sending

requests to remote servers. A simple example of such an application would be distributing a software package or update throughout an office space by utilizing frequent encounters between employees' devices as they move about the office space. Server nodes could be placed in an area of the office with high traffic such as the lobby and kitchen to distribute different parts of the software to each employee's device as they pass by. As employees move throughout the office, they can exchange data until every employee has received the needed files.

RLTP can be implemented as a solution for a variety of mobile systems within the Internet of Things. Vehicular networks suffer from even shorter encounter times than pedestrian networks and RLTP can be used to communicate map, traffic, and Point of Interest (POI) information between vehicles as they travel on a freeway. Additional applications can be found in Wireless Sensor Networks (WSNs) in which sensors are connected to mobile units or even animals, as described below.

There are also far-reaching applications of RLTP to be found in tracking animal and marine life. Communication devices called tags are routinely attached to wildlife in order to unobtrusively collect various data on the animals such as their migration patterns. Wildlife research, and especially studies that involve tagging and tracking marine animal are increasing every year. While there were just 40 studies in which ocean life was tagged and tracked in 2002, that number increased to 140 by 2014 and is still growing [31]. While the number of studies has increased dramatically, tagging technology has remained relatively unchanged over the past 3 decades. GPS and radio wave transmissions work well for tracking land animals, but these signals do not travel far under water and therefore have limited use. Thus, trackers equipped with such transmitters can only communicate through satellite links when the animal swims up to the surface of the water. Aside from this strict limitation, satellite communications are very costly and their area of coverage may be limited. Another common challenge in animal tagging is that researchers are rarely able to catch the same animal twice, which can prevent them from obtaining important data stored locally on the tag. Consider how RLTP might be deployed on tags used to track the travel patterns of sharks: Each tagged shark is a node, and the dataset to be disseminated throughout the network is a library containing each shark's tracking data. Every tag generates and records its own data as the shark travels, and when encountering another tagged shark, the trackers can exchange information resulting in both sharks carrying information

about themselves and their neighbor. Other nodes can be placed in fixed locations, connected to buoys, or attached to boats, which act as sink nodes whose sole responsibility it to collect data from any shark that happens to pass by. In this way, a single passing shark can provide researchers with information about an entire shiver (school of sharks).

Swarm Intelligence and swarm robotics are emerging areas of research that attempt to model the collective behavior of animals or organisms called agents. Locally, the members of a swarm act as individual agents and follow simple rules. These agents generally exhibit a collective behavior, such as their motion, or act towards a common goal. Examples of swarms found in nature include ants working together to build a colony, schools of fish traveling in coordinated movements to ward off predators, and the collaborative creation of a beehive [32]. Researchers inspired by these behaviors developed the concept of swarm robotics. Swarm robotics is a new approach to the coordination of multi-robot systems that consist of large numbers of simple robots. Thee robots are programed to act in a collective behavior as decentralized, self-organized system [33]. Swarms of small robots are advantageous over larger robots because they are cheaper, simple to design, and easily replaceable. RLTP can be used to provide swarms of independent and autonomous devices an efficient communication scheme by which information can be quickly disseminated. Abstractly, every RLTP node is a member of a swarm where the common goal is to attain and share every block of the dataset. The Rapid-Link Transfer Protocol is a simple set of rules that govern communications towards achieving a global goal through simple individualized local encounters.

5 Conclusions

5.1 RLTP, for the Internet of Moving Things

In this thesis, the design of a transfer protocol capable of disseminating data in a highly mobile and decentralized network was proposed. The protocol was shown to achieve a 100% efficiency increase compared to TCP. Additionally, the protocol is suitable to IoT applications employing a memoryless algorithm with simple states for easy deployment on low-cost, low-power electronics.

The Rapid-Link Transfer Protocol is applicable in many network environments and results show RLTP's ability to maximize the utility of even the shortest encounters. This is achieved by avoiding excess messages, such as those imposed by traditional communication protocols, in order to minimize the number of transmissions required before useful data can be shared.

The primarily limitation of RLTP is that it requires the a priori knowledge of an existing dataset that is divided into a known number of blocks. This requirement restricts RLTP from sharing data that may grow in unpredictable ways. Other limitations include being reliant on random encounters and unreliable data transmissions, which make RLTP unsuitable for mission-critical or life-safety workloads.

RLTP is best suited for wireless sensor networks, where data is highly organized and node movements can be modeled and in networks with highly mobile nodes such as autonomous vehicles and wearable devices. RLTP can be deployed on these sensors and devices to enable efficient, delay-tolerant communications during random duration encounters.

5.2 Future Work

There is considerable room to improve both RLTP's algorithm and optimize its implementation. One aspect of Contiki that proved problematic was creating multi-threading processes on the mote. Multi-threading allows for multiple processes or actions to be active simultaneously. While the processor can

only process a single instruction at a time, multi-threading allows the processor to switch between two active processes, periodically switching to work on a second task while the first task is temporarily paused. Currently, RLTP utilizes separate threads for sending blocks, computing and parsing gaps, and maintaining the bitmap. Ideally, a separate process thread would also be written to handle the processing of incoming blocks. This thread would allow a node to process received blocks during the downtime between encounters, further reducing the amount of encounter time spent on computations rather than exchanging data.

One important limitation in the current design of RLTP is handling packet loss. Figure 10 illustrates that during an encounter, RLTP's resting state is to wait for an incoming packet. If a packet is lost due to interference or data corruption, there is no mechanism for the sending node to retransmit the packet as in TCP. If a packet is dropped, for whatever reason, RLTP will move forward assuming that the encounter has idled because the nodes have moved out of range and the encounter will terminate after a missed discovery beacon attempt. If the nodes are still in range of each other after the encounter is terminated, they can pick up where they left off upon receiving another discovery beacon. While incorporating a mechanism for retransmitting undelivered packets would require explicit ACK messages, RLTP may be improved in the future by retransmitting the last message if a response has not been received within a specified interval. If this were to be developed, the retransmission would likely be send as a unicast message and the interval to wait should be smaller than the discovery beacon interval to ensure that the encounter is not terminated prematurely.

Other improvements to RLTP may be realized by implementing some techniques described in previous works. Performance might be improved if Bloom filters were used instead of RLTP's range vector. While Bloom filters may yield false positives, more research is required to determine if the probability of false positives could be made low enough that it would outperform the range vector.

Also, it was shown that under some conditions, a bit vector representation is more efficient than the range vector. A hybrid approach could be implemented where the data request is made in the most efficient way and new packet types are introduced differentiate between the different representations.

6 References

[1] “IEEE 802.11TM WIRELESS LOCAL AREA NETWORKS.” IEEE 802.11, The Working Group Setting the Standards for Wireless LANs, www.ieee802.org/11/.

[2] I. Gifford. “IEEE 802.15 Working Group For.” IEEE 802.15 Working Group for Wireless Personal Area Networks (WPANs), www.ieee802.org/15/.

[3] J. Hoebeke, I Moerman, B Dhoedt, P Demeester. An Overview of Mobile Ad Hoc Networks: Applications and Challenges. Department of Information Technology, Ghent University – IMEC vzw, Sint Pietersnieuwstraat 41, B-9000 Ghent, Belgium. 2005.

[4] J. Ott, D. Kutscher. Drive-thru Internet: IEEE 802.11b for Automobile Users. In Proc. of IEEE Infocom 2004.

[5] Speed Limits. Insurance Institute for Highway Safety, www.iihs.org/iihs/topics/laws/speedlimits.

[6] J. Guo, N. Balon, "Vehicular Ad Hoc Networks and Dedicated Short-Range Communication", Technical Report, June 2006.

[7] J. Kurose, K. Ross. (2004). Computer Networking, Third Ed. Benjamin/Cummings. ISBN 0-321-22735-2.

[8] M. Mohammadi Zanjireh, H. Larijani. (2015). A Survey on Centralised and Distributed Clustering Routing Algorithms for WSNs. IEEE Vehicular Technology Conference. 2015. 10.1109/VTCSpring.2015.7145650.

[9] “TCP Three-Way Handshake.” Koreanatccit, 11 Oct. 2011, koreanatccit.wordpress.com/.

[10] R. Chandra, J. Padhye, L. Ravindranath, A. Wolman. “Beacon-Stuffing: Wi-Fi without Associations.” Eighth IEEE Workshop on Mobile Computing Systems and Applications, 2007.

[11] “ISO - International Organization for Standardization.” ISO/IEC 7498-1:1994 - Information Technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model, 1 June 1996, www.iso.org/standard/20269.html.

[12] “The OSI Model's Seven Layers Defined and Functions Explained". Microsoft Support. Retrieved 2015-04-15.

[13] Hewitt, JB. OSI Reference Model for Data Networks. Wikipedia, 16 Mar. 2005, commons.wikimedia.org/wiki/File%3AOsi-model-jb.png.

[14] X.800 : Security Architecture for Open Systems Interconnection for CCITT Applications, www.itu.int/rec/T-REC-X.800-199103-I/en.

[15] Goralski, Walter. The Illustrated Network How TCP/IP Works in a Modern Network. Morgan Kaufmann, 2009.

[16] T. Zahn, G. O’Shea, A. Rowstron. Feasibility of Content Dissemination Between Devices in Moving Vehicles. In Proc. of ACM CoNEXT 2009.

[17] Bloom, B. Space/time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, 13 (7). 422-426.

[18] D. Eppstein and M. T. Goodrich, "Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters," in IEEE Transactions on Knowledge and Data Engineering, vol. 23, no. 2, pp. 297-306, Feb. 2011

[19] B. Yu and F. Bai. ETP: Encounter Transfer Protocol for Opportunistic Vehicle Communication. In Proc. of the IEEE INFOCOM, 2011.

[20] A. Bhargava, S. Congero, T. Ferrell, A. Jones, L. Linsky, J. Mohan, B. Krishnamachari. “Optimizing Downloads over Random Duration Links in Mobile Networks“, 25th International Conference on Computer Communication and Networks, August 2016.

[21] Internet of Things Global Standards Initiative, ITU, 20 July 2015, www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx.

[22] Dunkels, Adam. Contiki: The Open Source OS for the Internet of Things. www.contiki-os.org/.

[23] F. Osterlind. “Contiki Projects Community.” Cooja Mobility Plugin, Mar. 2010, sourceforge.net/p/contikiprojects/code/HEAD/tree/sics.se/mobility/.

[24] “Internet Speeds Explained.” Dealer Information Systems, 13 Nov. 2013, www.dis-corp.com/content/index.php/blog/internet-speeds-explained/.

[25] A. Dunkels, F. Österlind, Z. He. "An Adaptive Communication Architecture for Wireless Sensor Networks, "*Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems*" (2007). Sydney, AU.

[26] Raymond, Eric S. The Lost Art of C Structure Packing. 1 Jan. 2014, www.catb.org/esr/structure-packing/.

[27] Florida Speed Limits. Florida Drivers Association, www.123driving.com/dmv/drivers-handbook-speed-limits.

[28] FCC. "Tmote Sky User Manual Tmote-Sky-Datasheet-102 Sentilla Corporation." FCCID.io, fccid.io/TOQTMOTESKY/User-Manual/Users-Manual-Revised-613136.

[29] Urban Street Design Guide. National Association of City Transportation Officials, nacto.org/publication/urban-street-design-guide/street-design-elements/lane-width/.

[30] "Study Compares Older and Younger Pedestrian Walking Speeds." USRoads.com, TranSafety, Inc., 1 Oct. 1997, www.usroads.com/journals/p/rej/9710/re971001.htm.

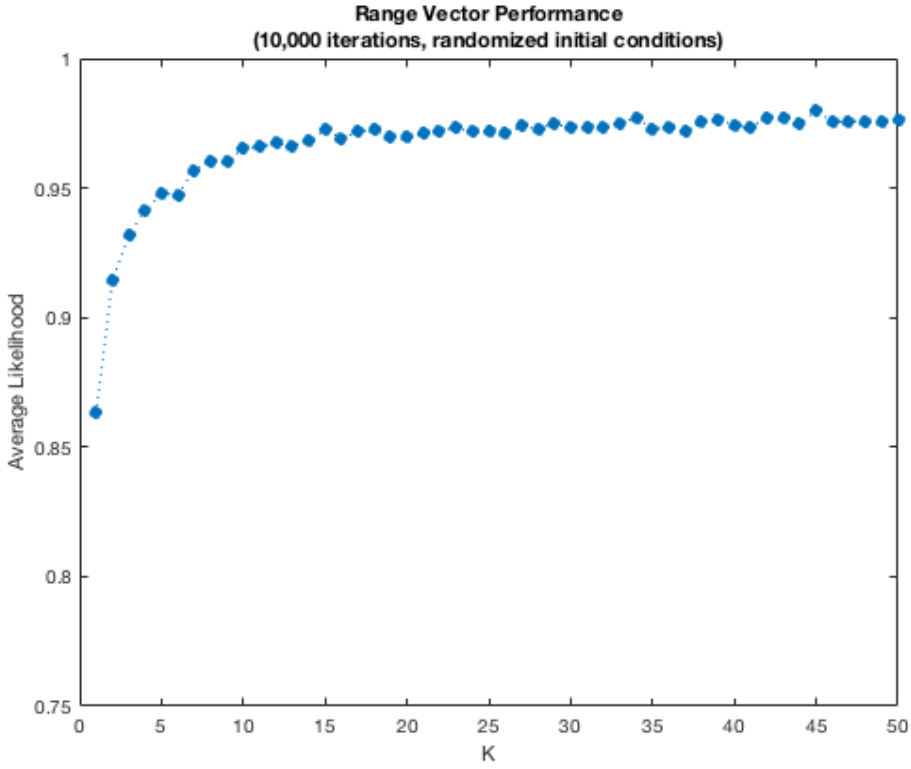
[31] N. Hussey, S. Kessel, K. Aarestrup, S. Cooke, P. Cowley, A. Fisk, R. Harcourt, K. Holland, S. Iverson, J. Kocik, J. Mills, F. Whoriskey. (2015). ECOLOGY. Aquatic animal telemetry: A panoramic window into the underwater world. Science. 348. 1221. 10.1126/science.1255642.

[32] Sangita Roy, Samir Biswas, Sheli Sinha Chaudhuri, "Nature-Inspired Swarm Intelligence and Its Applications", IJMECS, vol.6, no.12, pp.55-65, 2014.DOI: 10.5815/ijmecs.2014.12.08

[33] H. Hamann, Swarm Robotics: A Formal Approach, Springer, New York, 2018.

7 Appendix

7.1 Range Vector Performance (K = 1 thru 50)



K	Likelihood	K	Likelihood	K	Likelihood	K	Likelihood	K	Likelihood
1	86.3%	11	96.6%	21	97.2%	31	97.3%	41	97.4%
2	91.4%	12	96.7%	22	97.2%	32	97.4%	42	97.7%
3	93.2%	13	96.6%	23	97.3%	33	97.5%	43	97.7%
4	94.1%	14	96.8%	24	97.2%	34	97.7%	44	97.5%
5	94.8%	15	97.2%	25	97.2%	35	97.3%	45	98.0%
6	94.8%	16	96.9%	26	97.1%	36	97.4%	46	97.5%
7	95.7%	17	97.2%	27	97.4%	37	97.2%	47	97.6%
8	96.0%	18	97.3%	28	97.3%	38	97.5%	48	97.6%
9	96.0%	19	97.0%	29	97.5%	39	97.7%	49	97.5%
10	96.5%	20	96.9%	30	97.4%	40	97.4%	50	97.6%

7.2 Additional Simulation Data

Simulation #1: RLTP Encounter in a Vehicular Network (Speed = 15 m/s)

Run	Time of Encounter Initiation	Time Final Message Received	Time of Encounter Termination	Total Encounter Time	Termination Delay
1	13.2	14.9	17.6	1.7	2.7
2	13.1	15.0	17.6	1.9	2.6
3	13.3	14.9	17.8	1.6	2.9
4	14.2	14.8	17.2	0.6	2.4
5	13.3	14.9	17.8	1.6	2.9
6	13.3	15.0	17.8	1.7	2.8
7	13.2	14.9	17.7	1.7	2.8
8	13.2	15.0	17.8	1.8	2.8
9	13.0	15.0	17.6	2.0	2.6
10	13.3	14.9	17.8	1.6	2.9
11	13.0	15.0	17.5	2.0	2.5
12	13.2	14.8	17.8	1.6	3.0
13	13.3	15.0	17.8	1.7	2.8
14	12.9	14.8	17.5	1.9	2.7
15	12.7	15.0	17.2	2.3	2.2
16	14.3	14.8	17.3	0.5	2.5
17	12.7	14.8	17.2	2.1	2.4
18	13.2	14.9	17.9	1.7	3.0
19	13.2	15.0	17.6	1.8	2.6
20	13.3	14.8	17.7	1.5	2.9
Average	**13.2**	**14.9**	**17.6**	**1.7**	**2.7**

Simulation #2: RLTP Encounter in a Vehicular Network (Speed = 30 m/s)

Run	Time of Encounter Initiation	Time Final Message Received	Time of Encounter Termination	Total Encounter Time	Termination Delay
1	10.1	10.9	13.1	0.8	2.2
2	9.8	10.9	12.4	1.1	1.5
3	9.9	11.0	12.6	1.1	1.6
4	10.3	10.7	11.2	0.4	0.5
5	10.3	10.8	11.2	0.5	0.4
6	10.3	10.7	11.6	0.4	0.9
7	10.1	10.9	11.5	0.8	0.6
8	9.9	10.9	11.8	1.0	0.9
9	10.0	10.8	12.5	0.8	1.7
10	N/A	N/A	N/A	N/A	N/A
11	9.7	10.9	12.5	1.2	1.6
12	9.8	11.0	12.7	1.2	1.7
13	10.9	N/A	12.4	N/A	N/A
14	10.2	10.9	11.7	0.7	0.8
15	10.2	10.9	12.6	0.7	1.7
16	9.9	10.8	12.7	0.9	1.9
17	9.6	10.8	12.5	1.2	1.7
18	10.0	10.9	12.9	0.9	2.0
19	9.9	10.9	12.6	1.0	1.7
20	10.3	10.8	11.6	0.5	0.8
Average	**10.1**	**10.9**	**12.2**	**0.8**	**1.3**

Simulation #3: RLTP Encounter in a Pedestrian Network (Speed = 1 m/s)

Run	Time of Encounter Initiation	Time Final Message Received	Time of Encounter Termination	Total Encounter Time	Termination Delay
1	44.1	56.9	59.1	12.8	2.2
2	44.5	57.0	59.7	12.5	2.7
3	44.3	56.9	59.5	12.6	2.6
4	44.5	56.8	59.8	12.3	3.0
5	44.3	57.0	59.5	12.7	2.5
6	44.5	56.9	59.0	12.4	2.1
7	46.1	56.9	58.4	10.8	1.5
8	44.9	57.0	58.4	12.1	1.4
9	44.6	56.9	59.6	12.3	2.7
10	44.4	56.6	58.1	12.2	1.5
11	44.2	56.9	59.0	12.7	2.1
12	44.4	56.9	59.1	12.5	2.2
13	44.8	57.0	58.6	12.2	1.6
14	44.9	56.8	59.5	11.9	2.7
15	44.6	56.9	59.2	12.3	2.3
16	44.9	56.9	59.6	12.0	2.7
17	44.1	56.8	59.4	12.7	2.6
18	44.6	56.7	59.6	12.1	2.9
19	44.4	56.9	59.9	12.5	3.0
20	44.6	56.9	59.7	12.3	2.8
Average	**44.6**	**56.9**	**59.2**	**12.3**	**2.4**

7.3 Source Code: “RLTP.c”

RLPT.c Page 1 of 8

```
#include "contiki.h"
#include "sys/etimer.h"
#include "sys/stimer.h"
#include "sys/clock.h"
#include "net/rime/rime.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#include "lib/random.h"
#include "calcs.h"

// flag for if node should broadcast its periodic beacon
static int inEncounter = 0;
int previous_block_sent = -1;
int prev_files_rcvd = 0;

// event to send when a block is ready to be sent
static process_event_t block_ready;

// naming processes
PROCESS(bcn_broadcast, "Beacon Broadcast");
PROCESS(send_blocks, "Send Blocks");

// starting processes
AUTOSTART_PROCESSES(&bcn_broadcast, &send_blocks);

// naming broadcasts and unicasts
static struct broadcast_conn bcn_broadcastC;
static struct broadcast_conn blk_broadcastC;
static struct unicast_conn unicastC;

// Create PushQ
pushQ_struct pushQreal;
pushQ_struct* pushQ = &pushQreal;

// callback for receiving BCN
static void BCN_rcv_Call(struct broadcast_conn *c, const linkaddr_t *from) {
  inEncounter = 1; // Prevent Peiodic beacons during encounter
  uint8_t pkt_typ = ((char*)packetbuf_dataptr())[0];
  uint8_t my_percent = calcPercent();
  if (pkt_typ==PKT_TYP_PERIODIC)  {
    bcn_struct* bcn = (bcn_struct*)(packetbuf_dataptr());
    printf("Broadcast received from %d.%d.\n", from->u8[0], from->u8[1]);
    printf("  Message Type: %d \n  My Percent: %d \n  His Percent: %d
```

```
\n",(uint8_t)pkt_typ,my_percent,(uint8_t)bcn->percent);
    contig_struct* his_gaps = (contig_struct*)bcn->gaps;
    //printGaps(his_gaps,NUMGAPS);
    genPushQ(his_gaps, pushQ, NUMGAPS);
    printPushQ(pushQ);

    if (pushQ[0].length == 0) { //nothing to send (regardless of %)
      /* SEND BEACON_TYP_2 (BCN*) */
      printf("My pushQ is empty. Sending BCN*...\n");
      initBeacon();
      ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_2; //overwrite PKT_TYP
      unicast_send(&unicastC, from);
    }

    else { // I have files in his largest gaps.
      int blk_num = pushQ->Q[pushQ->length-1];
      printf("Preparing Block %d...\n", blk_num);
      block_struct blk = makeBlock(blk_num, pkt_typ, pushQ->length);
      if (blk.block_num != -1) {
        // allocating event
        block_ready = process_alloc_event();

        // send event and pushQ to send_blocks process
        process_post(&send_blocks, block_ready, &blk);

        //  broadcast_send(&blk_broadcastC);
        previous_block_sent = blk_num;
        printf(" Sent!\n\n");
      }
      else {
            printf(" Failed to send!\n\n");
      }
         pushQ->length--;
      }
   }

  else if (pkt_typ==PKT_TYP_PING) { // if Status Update Request
    printf("\n\nReceived Status Update Request.\n");
    initBeacon();
    ((char*)packetbuf_dataptr())[0] = PKT_TYP_STATUS; //overwrite PKT_TYP
    unicast_send(&unicastC, from);
    printf("  Sent Status Update.\n\n");
  }
}

/*---------------------------------------------------------------------------*/
```

```
// callback for receiving Block
static void BLK_rcv_Call(struct broadcast_conn *c, const linkaddr_t *from) {
  inEncounter = 1; // Prevent Peiodic beacons during encounter
  printf("Block received from %d.%d. (BLK bc)\n", from->u8[0], from->u8[1]);
  block_struct* block = (block_struct*)(packetbuf_dataptr());
  uint8_t pkt_typ = block->pktTyp;
  if (updateBitmap(block->block_num)) {
    printf("  Block: %d Size: %d Data: ", block->block_num, sizeof(block->data));
    int i;
    for (i = 0; i < sizeof(block->data); ++i) { //should be 80
      printf("%c,", block->data[i]);
    }
    printf("\n\n");
  }
  //printGaps(block->gaps,ACK_GAPS);
  genPushQ(block->gaps, pushQ, ACK_GAPS);
  printPushQ(pushQ);

  if (pushQ->length == 0) { //nothing to send (regardless of %)
    initBeacon();
    if (sentAllGaps()) {
      if (pkt_typ == PKT_TYP_BLOCK_LAST) {
        /* TERMINATE ENCOUNTER */
        printf("My pushQ is also empty. I have already sent all my gaps and he has
sent all Blocks.\nTerminating...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_TERMINATE; //overwrite PKT_TYP
      }
      else {
        /* SEND BEACON_TYP_3 (BCN**) */
        printf("My pushQ is empty. I have already sent all my gaps.\nSending
BCN**...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_3; //overwrite PKT_TYP
      }
    }
    else {
      /* SEND BEACON_TYP_2 (BCN*) */
      printf("My pushQ is empty. I have more gaps to share.\nSending BCN*...\n");
      ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_2; //overwrite PKT_TYP
    }
    unicast_send(&unicastC, from);
  }

  else { // I have files in his largest gaps.
    int blk_num = pushQ->Q[pushQ->length-1];
    printf("Preparing Block %d...\n", blk_num);
    block_struct blk = makeBlock(blk_num, pkt_typ, pushQ->length);
    if (blk.block_num != -1) {
```

```
     // allocating event
      block_ready = process_alloc_event();
      // send event and pushQ to send_blocks process
      process_post(&send_blocks, block_ready, &blk);
      previous_block_sent = blk_num;
      printf(" Sent!\n\n");
    }
    else {
      printf(" Failed to send!\n\n");
    }
    pushQ->length--;
  }
}

/*---------------------------------------------------------------------------*/

// callback for receiving unicast
static void uni_rcv_Call(struct unicast_conn *c, const linkaddr_t *from) {
  inEncounter = 1;
  printf("Unicast received from %d.%d.\n", from->u8[0], from->u8[1]);
  uint8_t pkt_typ = ((char*)packetbuf_dataptr())[0];
  int my_percent = calcPercent();
  if (pkt_typ==PKT_TYP_BEACON_2) {
    bcn_struct* bcn = (bcn_struct*)(packetbuf_dataptr());
    printf("    Message  Type:  %d  \n    My  Percent:  %d  \n    His  Percent:  %d
\n",pkt_typ,my_percent,bcn->percent);
    printf("BCN* received: Sender has empty PushQ.\n");
    contig_struct* his_gaps = (contig_struct*)bcn->gaps;
    //printGaps(his_gaps,NUMGAPS);
    genPushQ(his_gaps, pushQ, NUMGAPS);
    printPushQ(pushQ);

    if (pushQ->length == 0) { //nothing to send (regardless of %)
      initBeacon();
      if (sentAllGaps()) {
        /* SEND BEACON_TYP_3 (BCN**) */
        printf("My pushQ is also empty. I have already sent all my gaps.\nSending
BCN**...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_3; //overwrite PKT_TYP
      }
      else {
        /* SEND BEACON_TYP_2 (BCN*) */
        printf("My  pushQ  is  also  empty.  I  have  more  gaps  to  share.\nSending
BCN*...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_2; //overwrite PKT_TYP
      }
      unicast_send(&unicastC, from);
```

```
   }

    else { // I have files in his largest gaps.
      int blk_num = pushQ->Q[pushQ->length-1];
      printf("Preparing Block %d from uni*...\n", blk_num);
      block_struct blk = makeBlock(blk_num, pkt_typ, pushQ->length);
      if (blk.block_num != -1) {
        // allocating event
        block_ready = process_alloc_event();
        // send event and pushQ to send_blocks process
        process_post(&send_blocks, block_ready, &blk);
        previous_block_sent = blk_num;
      }
      else {
        printf(" Failed to send!\n\n");
      }
      pushQ->length--;
    }

  if (pkt_typ==PKT_TYP_BEACON_3) { //He has sent all of his gaps and both his and
my PushQ are currently empty
    bcn_struct* bcn = (bcn_struct*)(packetbuf_dataptr());
    printf("    Message Type: %d \n    My Percent: %d \n    His Percent: %d
\n",pkt_typ,my_percent,bcn->percent);
    printf("BCN** received. Sender has sent all his gaps and has an empty
PushQ.\n");
    contig_struct* his_gaps = (contig_struct*)bcn->gaps;
    //printGaps(his_gaps,NUMGAPS);
    genPushQ(his_gaps, pushQ, NUMGAPS);
    printPushQ(pushQ);

    if (pushQ->length == 0) { //nothing to send (regardless of %)
      initBeacon();
      if (sentAllGaps()) {
        /* END ENCOUNTER. */
        printf("My pushQ is also empty. I have also sent all my gaps.\nTerminating
Encounter...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_TERMINATE; //overwrite PKT_TYP
        inEncounter = terminate(pushQ);
      }
      else {
        /* SEND BEACON_TYP_2 (BCN*) */
        printf("My pushQ is also empty, but I have more gaps to share.\nSending
BCN*...\n");
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_BEACON_2; //overwrite PKT_TYP
      }
      unicast_send(&unicastC, from);
```

```
   }

    else { // I have files in his largest gaps.
      int blk_num = pushQ->Q[pushQ->length-1];
      printf("Preparing Block from uni**%d...\n", blk_num);
      block_struct blk = makeBlock(blk_num, pkt_typ, pushQ->length);
      if (blk.block_num != -1) {
        // allocating event
        block_ready = process_alloc_event();
        // send event and pushQ to send_blocks process
        process_post(&send_blocks, block_ready, &blk);
        previous_block_sent = blk_num;
        printf(" Sent!\n\n");
      }
      else {
        printf(" Failed to send!\n\n");
      }
      pushQ->length--;
    }
 }

  if (pkt_typ==PKT_TYP_TERMINATE) { //All Gaps have been sent ant both his and my
PushQ are empty
    bcn_struct* bcn = (bcn_struct*)(packetbuf_dataptr());
    printf("Received Termination Packet. Terminating encounter.\n");
    printf("    Message  Type:  %d  \n    My  Percent:  %d  \n    His  Percent:  %d
\n",pkt_typ,my_percent,bcn->percent);
    inEncounter = terminate(pushQ);
  }

  if (pkt_typ==PKT_TYP_STATUS) { //All Gaps have been sent ant both his and my
PushQ are empty
    bcn_struct* bcn = (bcn_struct*)(packetbuf_dataptr());
    printf("Status Message received from %d.%d.\n", from->u8[0], from->u8[1]);
    printf("    Message  Type:  %d  \n    My  Percent:  %d  \n    His  Percent:  %d
\n",pkt_typ,my_percent,bcn->percent);
  }
}

/*---------------------------------------------------------------------------*/

// naming broadcast callbacks
static const struct broadcast_callbacks beacon_call = {BCN_rcv_Call};
static const struct broadcast_callbacks blocks_call = {BLK_rcv_Call};
static const struct unicast_callbacks unicast_call = {uni_rcv_Call};

/*---------------------------------------------------------------------------*/
```

```
// bcn_broadcast to send periodic beacons
PROCESS_THREAD(bcn_broadcast, ev, data) {
   // exit handler
   PROCESS_EXITHANDLER(broadcast_close(&bcn_broadcastC);)
  PROCESS_EXITHANDLER(unicast_close(&unicastC);)

   // begin process
   PROCESS_BEGIN();

   // open broadcast
   broadcast_open(&bcn_broadcastC, 128, &beacon_call);
  unicast_open(&unicastC, 146, &unicast_call);

   // Initialize node
   static struct etimer et;
   genBitmap(-1); //argument for Percent, use -1 for random %

  while(1) {
    /* Every BCN_INTERVAL */
    etimer_set(&et, CLOCK_SECOND*BCN_INTERVAL); //0-2s of randomness
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    if (!inEncounter) {
       //printf("Initialize Beacon...\n");
      create_gaps(); //creates my_gaps for the current encounter
      initBeacon();
       //printf("Beacon Initialized!\n");
       ((char*)packetbuf_dataptr())[0] = PKT_TYP_PERIODIC; //overwrite PKT_TYP
       broadcast_send(&bcn_broadcastC);
       printf("Periodic Beacon sent.\n");
     }
     else {
      if (prev_files_rcvd == getFilesRcvd()) {
        inEncounter=0;
        create_gaps(); //creates my_gaps for the current encounter
        initBeacon();
        ((char*)packetbuf_dataptr())[0] = PKT_TYP_PERIODIC; //overwrite PKT_TYP
        broadcast_send(&bcn_broadcastC);
        printf("Encounter has timed out. Periodic Beacon sent.\n");
        printf("Files Received: %d\n", getFilesRcvd());
      }
      else {
              printf("In Encounter. Didnt Send Periodic Beacon.\n");
      }
      prev_files_rcvd = getFilesRcvd();
     }
  }
```

```
  PROCESS_END();
}

/*---------------------------------------------------------------------------*/

// send_blocks to broadcast blocks from PushQ
PROCESS_THREAD(send_blocks, ev, data) {
   // exit handler
   PROCESS_EXITHANDLER(broadcast_close(&blk_broadcastC);)

   // begin process
   PROCESS_BEGIN();

   // open broadcast
   broadcast_open(&blk_broadcastC, 129, &blocks_call);

   while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == block_ready);
      //printf("Send Process: Sent Block %d\n", blkk->block_num);
      broadcast_send(&blk_broadcastC);
      //printf("SENT! blk %d with typ %d\n",(int)blkk->block_num,(uint8_t) blkk-
>pktTyp);
  }
  PROCESS_END();
}
```

7.4 Source Code: "RLTP-core.h"


```
#ifndef __CALCS_H__
#define __CALCS_H__

#define NUM_BLOCKS 1000
#define NUMGAPS 10 //hard set maximal amount of gaps to transmit
#define ACK_GAPS 5 //number of gaps to send with each block
#define MAXGAPS NUM_BLOCKS/2 //The most gaps you can have
#define BEACON_SIZE 1 + 1 + 4*NUMGAPS // 42. [pktType (1), %_Complete (1),
Largest_Gaps(=80)]
#define BCN_INTERVAL 1.5 // BCN every BCN_INTERVAL seconds
#define BLOCK_DATA_SIZE 80 // bytes
#define BLOCK_SIZE 1 + 1 + 2 + BLOCK_DATA_SIZE + 4*ACK_GAPS // 104 bytes/block [1b
pktTyp +1b percent + 2b block_num + 80b data + 4b*ACK_GAPS]

typedef struct {
    int start_addr;
    int offset;
} contig_struct;

typedef struct {
    uint8_t pktTyp;
    uint8_t percent;
    contig_struct gaps[NUMGAPS];
} bcn_struct;

typedef struct {
    uint8_t pktTyp;
    uint8_t percent;
    int block_num;
    char data[BLOCK_DATA_SIZE];
    contig_struct gaps[ACK_GAPS];
} block_struct;

typedef struct {
    int length;
    int Q[NUM_BLOCKS];
    uint8_t init;
} pushQ_struct;

/*************************************************************/
void genBitmap(int P);

void set_FLAG(int flag);
```

```
int sentAllGaps(void);

uint8_t calcPercent(void);

int getFilesRcvd();

block_struct makeBlock(int bitmap_index, uint8_t pktTyp, int pushQ_length);

uint8_t updateBitmap(int bitmap_index);

void sort(contig_struct *ptr_to_contigs);

void printGaps(contig_struct *ptr_to_contigs, int amt);

int find_gaps(contig_struct *ptr_to_contigs, int *bits);

void create_gaps(void);

int genGapBuf(contig_struct *gapBuf, int max_gaps2send);

void genPushQ(contig_struct *rcv_gaps, pushQ_struct *pushQ, int amt);

void printPushQ(pushQ_struct *pushQ);int terminate(pushQ_struct *pq);

void initBeacon(void);

// enumeration of PKT_TYPs
enum {
   PKT_TYP_PERIODIC,   //Periodic BCN
   PKT_TYP_BEACON_1,    //Beacon (BCN)
   PKT_TYP_BEACON_2,   //Reply to a Beacon, with a Beacon. (BCN*)
   PKT_TYP_BEACON_3,    //reply to BCN* (BCN**)
   PKT_TYP_BLOCK_FIRST,//Send a Block (reply from BCN_PERIODIC)
   PKT_TYP_BLOCK,      //Send a Block
   PKT_TYP_BLOCK_LAST,   //Send a Block (last block in PushQ)
   PKT_TYP_PING,      //Ping node for Status
   PKT_TYP_STATUS,      //Status PKT (basically a BCN)
   PKT_TYP_ACK,      //ACK BCN
   PKT_TYP_TERMINATE   //Empty packet.
};

#endif /* __CALCS_H__ */ //
```

7.5 Source Code: "RLTP-core.c"


```c
#include <stdio.h>
#include <string.h> // memset, memcpy
#include "packetbuf.h"
#include "calcs.h"
#include "random.h"

/* NODE'S INTERNAL VARIABLES */
static contig_struct my_gaps[MAXGAPS]; //NUM_BLOCKS/2 = 50
static int mygap_ptr = 0; // initialized to point to first gap
static int mygap_amt = 0;
static block_struct block_buf;
static bcn_struct bcn_buf;
int bitmap[NUM_BLOCKS];
char sampleDataBlock[BLOCK_DATA_SIZE] =
"SampleDataSampleDataSampleDataSampleDataSampleDataSampleDataSampleDataSampleData";

/* Flags & Counters */
int files_recvd = 0; //counts how many files have been received in the encounter
int FLAG_sent_all_gaps = 0;

/*********************************************************/

void genBitmap(int P){
   /* INITIALIZE BITMAP */
   memset(&bitmap, 0, sizeof(bitmap));
   /* Random init with node-specific seed */
   //overwrites the seed at contiki-sky-main.c:307
   random_init(*(uint16_t*)(linkaddr_node_addr.u8[0])*3+5);

   /* HARDCODE PERCENT OF NODE 1 & 2 */
   if (linkaddr_node_addr.u8[0] == 1) { // Node 1
      P = -1;
   }
   if (linkaddr_node_addr.u8[0] == 2) { // Node 2
      P = -1;
   }

 /* GENERATE BITMAP & GAPS */
   if (P<0){                  // Use P<0 for random Percentage
      P = random_rand() % 101; // Random number between 0-100
   }
   int i, count = 0;
   printf("Bitmap (N=%d) generated from seed P=%d:\n",NUM_BLOCKS, P);
```

```
for (i = 0; i < NUM_BLOCKS; i++) {
      if (random_rand() % 101 <= P) {
         bitmap[i] = 1;
         count++;
      }
      printf("%d,",bitmap[i]);
      if (i>0 && (i+1) % 25 == 0) {printf("\n");} // Punctuation.
   }
   find_gaps(my_gaps, bitmap);
   //printGaps(my_gaps,MAXGAPS);
   printf("%d Percent. %d gaps in bitmap.\n",100*count/NUM_BLOCKS, mygap_amt);
}

block_struct makeBlock(int bitmap_index, uint8_t pkt_typ, int pushQ_length){
   block_buf.block_num = bitmap_index;
   block_buf.percent = calcPercent();
   if (bitmap_index <0 || bitmap_index > NUM_BLOCKS) {
      printf("BLOCK NUM NOT IN SCOPE! \n");
      block_buf.block_num = -1;
      return block_buf;
   }
   block_buf.pktTyp = PKT_TYP_BLOCK; // If this block is a reply to a block
   if (pkt_typ == PKT_TYP_PERIODIC) {      // If this is the first Block I am
sending (in response to a BCN)
          block_buf.pktTyp = PKT_TYP_BLOCK_FIRST;
          mygap_ptr = 0;   //Reset Ptr to resend NUMGAPS previously sent in
unanswered beacon.
    }
    if (pushQ_length == 1) {      // If last Block I have in PushQ
       printf("Sending Last Block in PushQ!\n");
       block_buf.pktTyp = PKT_TYP_BLOCK_LAST;
    }
   memcpy(block_buf.data, sampleDataBlock,sizeof(block_buf.data)); //Would
actually read block from memory
   genGapBuf(block_buf.gaps,ACK_GAPS);

   /* CREATE PACKET FROM block_buf */
   packetbuf_copyfrom(&block_buf,sizeof(block_buf));
   packetbuf_set_datalen(BLOCK_SIZE); //100b Pkt: [PKT_TYP (1b +1b buffer),
Block_Num (2b), Block Data (80b), 4 Gaps (16b)]
   //printf("Made block %d of type %d\n", block_buf.block_num,block_buf.pktTyp);
   //printGaps(block_buf.gaps,ACK_GAPS);
   return block_buf;
}

uint8_t updateBitmap(int bitmap_index){
   if (bitmap_index < 0 || bitmap_index >= NUM_BLOCKS) {
```

```
      printf("Block Number out of scope!\n");
      return 0;
   }
   else if (bitmap[bitmap_index] == 1) {
      printf("Block already in Bitmap\n");
      return 0;
   }
   else { //update bitmap
      bitmap[bitmap_index] = 1;
      printf(" Updated bitmap with block %d.\n",bitmap_index);
      files_recvd++;
      return 1;
   }
}

uint8_t calcPercent() {
   int i, count = 0;
   for (i = 0; i < NUM_BLOCKS; ++i) {
      count = count + bitmap[i];
   //   printf("i %d, count: %d, bitmap_i
= %d\n",random_rand()%101,count,bitmap[i]);
   }
   //printf("count: %d, NUM_BLOCKS = %d\n",count,NUM_BLOCKS );
   uint8_t percent = (100*count)/NUM_BLOCKS;
   return percent;
}

int getFilesRcvd() {
   return files_recvd;
}

void sort (contig_struct *ptr_to_contigs) {
    contig_struct temp;
    int i;
    uint8_t didSwap;
    // Bubble Sort by Offset (descending)
    didSwap = 1;
    while (didSwap) {
        didSwap = 0;
        for (i = 0; i < MAXGAPS - 1; i++) {
            if (ptr_to_contigs[i].offset < ptr_to_contigs[i+1].offset) {
                temp = ptr_to_contigs[i];
                ptr_to_contigs[i] = ptr_to_contigs[i+1];
                ptr_to_contigs[i+1] = temp;
                didSwap = 1;
            }
        }
```

```
    }
}

int find_gaps(contig_struct *ptr_to_contigs, int *bits){
   memset(ptr_to_contigs, -1, MAXGAPS*sizeof(contig_struct));
   int i, j, offset = 0, g = 0;
   for (i=0; i<NUM_BLOCKS; ++i){
      if (bits[i] == 0) {
         int s_addr = i;
         if (i==NUM_BLOCKS-1) { //end case check 1: last block is start/end of a
gap
            ptr_to_contigs[g].offset = offset; // should alwaysbe 0
            ptr_to_contigs[g].start_addr = s_addr; //should always be NUMBLOCKS-1
            g++;
         }

         for (j = i+1; j < NUM_BLOCKS; ++j){
            if (bits[j] == 0){
               offset++;
               if (j==NUM_BLOCKS-1) { //end case check 2: last block = end of gap
                  ptr_to_contigs[g].offset = offset;
                  ptr_to_contigs[g].start_addr = s_addr;
                  i=j;
                  g++;
               }
            }

            else { // if 1 (end of gap)
               ptr_to_contigs[g].offset = offset;
               ptr_to_contigs[g].start_addr = s_addr;
               g++;
               i=j;
               offset=0;
               break;
            }
         }
      }
   }
   sort(ptr_to_contigs); //sort in descending order
   mygap_amt = g; // number of gaps that arent empty
   mygap_ptr = 0;
   return mygap_amt;
}

void create_gaps(void){
   find_gaps(my_gaps, bitmap);
   //printGaps(my_gaps,NUMGAPS);
```

```
}

void printGaps(contig_struct *ptr_to_contigs, int amt) {
   int i;
   printf("%d Gaps:\n",amt);
   for (i = 0; i < amt; i++) {
      printf("Start address: %d Offset: %d \n", ptr_to_contigs[i].start_addr,
ptr_to_contigs[i].offset);
   }
   printf("\n");
}

void genPushQ(contig_struct *rcv_gaps, pushQ_struct *pushQ, int gaps_to_search){
   int i, off;
   if (!pushQ->init) { // if PushQ has only just been initialized
       //printf("Initializing PushQ for the first time!\n");
       memset(pushQ->Q, -1, sizeof(pushQ->Q));
      pushQ[0].length = 0;
      pushQ->init = 1;
   }
   // start with smallest gap to allow nodes to retain larger contiguous gaps
   for (i = 0; i < gaps_to_search; i++) {
      for (off = 0; off <= rcv_gaps[i].offset; off++) { //ignores empty gap
         if (bitmap[rcv_gaps[i].start_addr + off] == 1) {
            pushQ->Q[pushQ->length] = (int) rcv_gaps[i].start_addr + off;
            pushQ->length++;
         }
      }
   }
}

void printPushQ(pushQ_struct *pushQ){
   int i;
   printf("Push Queue:\n");
   for (i = 0; i < pushQ[0].length; ++i) {
      printf("%d, ", pushQ[0].Q[i]);
   }
   printf("\n\n");
}

int sentAllGaps(void){
   return FLAG_sent_all_gaps;
}

void initBeacon(void) {
   bcn_buf.pktTyp = PKT_TYP_BEACON_1;
   bcn_buf.percent = calcPercent();
```

```
   genGapBuf(bcn_buf.gaps,NUMGAPS);
   //printGaps(bcn_buf.gaps,NUMGAPS);
   /* CREATE PACKET FROM BCN_BUF */
memcpy((void*)&(((char*)packetbuf_dataptr())[0]),(void*)&bcn_buf,sizeof(bcn_buf));
   packetbuf_set_datalen(BEACON_SIZE); // 42b Pkt: [PKT_TYP (1b), my_% (1b), 10
Gaps (40b)]
}

int genGapBuf(contig_struct *gapBuf, int max_gaps2send){
   // Raise FLAG if all gaps have been sent
   if (mygap_ptr >= mygap_amt) {
      printf("All gaps have previously been sent! Raising Flag.\n");
      //printf("gap_ptr=%d, gap_amt=%d\n",mygap_ptr,mygap_amt);
      FLAG_sent_all_gaps = 1;
   }
   if (FLAG_sent_all_gaps) {
      //printf(" (Already Sent All Gaps)\n");
      memset(&gapBuf[0], -1, max_gaps2send*sizeof(contig_struct));
      return 0;
   }
   else {
      // Determine amount of Gaps to send
      int gap2send;
      if (max_gaps2send > mygap_amt - mygap_ptr) {// Prevent sending >MAXGAPS
         gap2send = mygap_amt - mygap_ptr;
      }
      else {
         gap2send = max_gaps2send;
      }

      // Populate Gap Buffer
      memcpy(&gapBuf[0],(my_gaps)+mygap_ptr,gap2send*sizeof(contig_struct));
      mygap_ptr = mygap_ptr + gap2send;
      memset(&gapBuf[gap2send], -1, (max_gaps2send -
gap2send)*sizeof(contig_struct));
      return gap2send;
   }
}

int terminate(pushQ_struct *push_q) {
   printf("Received %d Blocks. Encounter Terminated.\n",files_recvd);
   FLAG_sent_all_gaps = 0;
   files_recvd = 0;
   push_q->init = 0;
   return 0;
}
```

7.6 Sample Code: Cooja Node Mobility

2_cars_passing_15.dat

#node	time(s)	x	y
0	0	-200	2
1	0	200	-2
0	0.5	-192.5	2
1	0.5	192.5	-2
0	1	-185	2
1	1	185	-2
0	1.5	-177.5	2
1	1.5	177.5	-2
0	2	-170	2
1	2	170	-2
0	2.5	-162.5	2
1	2.5	162.5	-2
0	3	-155	2
1	3	155	-2
0	3.5	-147.5	2
1	3.5	147.5	-2
0	4	-140	2
1	4	140	-2
0	4.5	-132.5	2
1	4.5	132.5	-2
0	5	-125	2
1	5	125	-2
0	5.5	-117.5	2
1	5.5	117.5	-2
0	6	-110	2
1	6	110	-2
0	6.5	-102.5	2
1	6.5	102.5	-2
0	7	-95	2
1	7	95	-2
0	7.5	-87.5	2
1	7.5	87.5	-2
0	8	-80	2
1	8	80	-2
0	8.5	-72.5	2
1	8.5	72.5	-2
0	9	-65	2
1	9	65	-2
0	9.5	-57.5	2
1	9.5	57.5	-2
0	10	-50	2
1	10	50	-2

```
0               10.5            -42.5           2
1               10.5            42.5            -2
0               11              -35             2
1               11              35              -2
0               11.5            -27.5           2
1               11.5            27.5            -2
0               12              -20             2
1               12              20              -2
0               12.5            -12.5           2
1               12.5            12.5            -2
0               13              -5              2
1               13              5               -2
0               13.5            2.5             2
1               13.5            -2.5            -2
0               14              10              2
1               14              -10             -2
0               14.5            17.5            2
1               14.5            -17.5           -2
0               15              25              2
1               15              -25             -2
0               15.5            32.5            2
1               15.5            -32.5           -2
0               16              40              2
1               16              -40             -2
0               16.5            47.5            2
1               16.5            -47.5           -2
0               17              55              2
1               17              -55             -2
0               17.5            62.5            2
1               17.5            -62.5           -2
0               18              70              2
1               18              -70             -2
0               18.5            77.5            2
1               18.5            -77.5           -2
0               19              85              2
1               19              -85             -2
0               19.5            92.5            2
1               19.5            -92.5           -2
0               20              100             2
1               20              -100            -2
0               20.5            107.5           2
1               20.5            -107.5          -2
0               21              115             2
1               21              -115            -2
0               21.5            122.5           2
1               21.5            -122.5          -2
0               22              130             2
```

1	22	-130	-2
0	22.5	137.5	2
1	22.5	-137.5	-2
0	23	145	2
1	23	-145	-2
0	23.5	152.5	2
1	23.5	-152.5	-2
0	24	160	2
1	24	-160	-2
0	24.5	167.5	2
1	24.5	-167.5	-2
0	25	175	2
1	25	-175	-2
0	25.5	182.5	2
1	25.5	-182.5	-2
0	26	190	2
1	26	-190	-2
0	26.5	197.5	2
1	26.5	-197.5	-2
0	27	205	2
1	27	-205	-2