BRUNET: DISRUPTION-TOLERANT TCP AND DECENTRALIZED WI-FI
FOR SMALL SYSTEMS OF VEHICLES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nicholas Brunet

December 2023
COMMITTEE MEMBERSHIP

TITLE: BRUNET: Disruption-Tolerant TCP and Decentralized Wi-Fi for Small Systems of Vehicles

AUTHOR: Nicholas Brunet

DATE SUBMITTED: December 2023

COMMITTEE CHAIR: John Bellardo, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Richard Murray
Lecturer, Computer Engineering

COMMITTEE MEMBER: Paul Hummel, Ph.D.
Lecturer, Computer Engineering
ABSTRACT

BRUNET: Disruption-Tolerant TCP and Decentralized Wi-Fi for Small Systems of Vehicles

Nicholas Brunet

Reliable wireless communication is essential for small systems of vehicles. However, for small-scale robotics projects where communication is not the primary goal, programmers frequently choose to use TCP with Wi-Fi because of their familiarity with the sockets API and the widespread availability of Wi-Fi hardware. However, neither of these technologies are suitable in their default configurations for highly mobile vehicles that experience frequent, extended disruptions. BRUNET (BRUNET Really Useful NETwork) provides a two-tier software solution that enhances the communication capabilities for Linux-based systems. An ad-hoc Wi-Fi network permits decentralized peer-to-peer and multi-hop connectivity without the need for dedicated network infrastructure. A background process adds disruption tolerance to specified TCP endpoints without any changes to existing software. This allows TCP connections to persist indefinitely over possibly multiple long network outages. Data sent by applications is automatically buffered and transmitted when network connectivity resumes.
ACKNOWLEDGMENTS

Thanks to:

- My parents, Joe and Dawn, for instilling in me the value of education and teaching me to pursue excellence in all my work. In particular, thanks to my mom, who spent countless hours homeschooling me and my five siblings all the way through high school.

- My fiancée, Genevieve, for her constant encouragement and patience this past year while I kept delaying deadlines for this project.

- My brother, Jesse, with whom I shared practically every activity for over 20 years. The friendly competition constantly pushed me to do better. You’re definitely one of the main reasons I’m in graduate school.

- My grandmother, Carol, for inspiring a good work ethic in me.

- Andy and Debbie, for generously giving me a place to live for the past several months while I worked on my thesis and extended deadlines.

- All of the wonderful faculty here at Cal Poly, for whom “learn by doing” is more than just a slogan. Professor Hummel, for never giving a dull lecture (was it just the hair?) or an easy project. Professor Murray, for inspiring this thesis and helping me narrow down a topic. Professor Siu (“Super Chris”), for giving the most lucid, precise lectures I’ve ever attended, and for all the help while I was teaching. And, of course, Professor Bellardo, for giving me a solid foundation in networks and especially for all the hours spent guiding me through every step of this project.

AD MAJOREM DEI GLORIAM
# TABLE OF CONTENTS

**LIST OF FIGURES** ............................................. ix

**CHAPTER**

1 Introduction .................................................. 1

1.1 Motivation .................................................... 1

1.1.1 Overview of BRUNET ........................................ 2

2 Related Work .................................................. 5

2.1 MANETs ....................................................... 5

2.1.1 MANET Routing Protocols ..................................... 5

2.1.2 Performance of MANET Routing Protocols ..................... 6

2.2 Disruption-Tolerant Connections ............................... 7

2.3 Store-Carry-Forward .......................................... 8

3 Design .......................................................... 10

3.1 Overview ...................................................... 10

3.1.1 Proxy Program Design ........................................ 12

3.1.2 System Scaling ............................................... 14

3.2 Underlying Ad-Hoc Network .................................... 15

3.3 Configuration File ............................................. 16

3.3.1 Design Concepts ............................................. 17

3.3.2 Implementation ............................................. 18

3.4 Connection Redirection Mechanism ............................. 19

3.4.1 Solutions Considered ........................................ 20

3.4.1.1 TPROXY .................................................. 21
3.4.1.2 NAT ................................. 21
3.4.2 Choosing a Netfilter Library ................................. 22
3.4.3 Netfilter Integration ................................. 23
3.5 Cache Files ......................................... 24
3.6 Managing Sockets ......................................... 26
3.7 Packet System ......................................... 27
 3.7.1 Creating a Logical Connection ................................. 29
 3.7.2 Sending and Receiving Data ................................. 30
 3.7.3 Acknowledging Data Delivery ................................. 31
 3.7.4 Closing a Logical Connection ................................. 31
 3.7.5 Buffering Output for Writing ................................. 33
 3.7.6 Receiving and Processing Packets on Input ................................. 36
3.8 Peer Connections ......................................... 37
 3.8.1 Establishing Inter-Proxy Connections ................................. 37
 3.8.2 Resuming Inter-Proxy Connections ................................. 38
3.9 Other Components ......................................... 39
 3.9.1 Dictionary for Logical Connections ................................. 39
 3.9.2 Error Handling ......................................... 40
 3.9.3 Logging ......................................... 41
4 Testing ................................................. 42
 4.1 General Information ................................................. 42
 4.2 BATMAN Configuration Verification ................................................. 42
 4.3 Unit Tests ................................................. 43
 4.4 End-to-End Tests ................................................. 43
 4.5 Performance Testing ................................................. 45
5 Concluding Remarks

5.1 Future Work

5.1.1 New Features

5.1.2 Performance Improvements

5.2 Conclusion

BIBLIOGRAPHY

APPENDICES

A Development Environment

A.1 Ad-Hoc Network

A.2 BATMAN

A.3 Compiling and Running BRUNET

A.4 Testing

B Complete Testing Log
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Components of a logical connection</td>
<td>3</td>
</tr>
<tr>
<td>3.1 Proxy program: bird’s-eye view</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Proxy program: interface with user programs</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Proxy program: interface with peer proxy programs</td>
<td>14</td>
</tr>
<tr>
<td>3.4 A complex configuration supported by BRUNET</td>
<td>15</td>
</tr>
<tr>
<td>3.5 BRUNET Packet Structure</td>
<td>27</td>
</tr>
<tr>
<td>3.6 Closing a logical connection</td>
<td>32</td>
</tr>
<tr>
<td>3.7 Pointers in a 12-byte circular output buffer</td>
<td>34</td>
</tr>
<tr>
<td>4.1 Average times for Test 1</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Average times for Test 2</td>
<td>47</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

1.1 Motivation

Reliable communication is essential for small systems where a number of vehicles cooperate to accomplish a common mission. Such systems frequently operate in remote regions where there is no general-purpose wireless communication infrastructure to provide networking among vehicles. Or, even if some infrastructure exists, it often is ill-equipped for highly mobile systems.

The *Perseverance* rover and *Ingenuity* copter are an example of a cooperative vehicle system on Mars. On Mars, communications infrastructure is limited to several satellites that provide limited Mars-to-Earth communications, and vehicles on the surface must precisely time communications with satellite orbits [1]. Thus, *Perseverance* and *Ingenuity* directly communicate with each other rather using a custom communication protocol and do not relay messages through a satellite [2].

While developing a specialized communication system optimized for a particular mission may be ideal, this is no small undertaking, especially with small-scale projects with a limited scope and budget. In practice, small-scale projects typically utilize TCP connections over Wi-Fi. These technologies are attractive due to being widely supported, inexpensive, and easy to use. While convenient for development in a lab environment, problems quickly arise when testing in the field.

In the standard configuration, Wi-Fi requires a central access point. In some systems, this is a device with a fixed location. In other systems, it is one of the vehicles. In
any case, utilizing a fixed access point or endowing a particular vehicle with this role forces other vehicles to remain nearby if they wish communicate with each other. This reduces the flexibility of the entire system.

A further issue is that TCP does not function well in environments with frequent, extended disruptions. In a system of vehicles, disruptions may occur if a pair of drones fly out of range of each other or a rover drives behind an obstacle. Data loss is likely if a connection times out during a transmission, and accounting for this can be a complex task.

1.1.1 Overview of BRUNET

BRUNET (BRUNET Really Useful NETwork) is a software system that improves wireless network connectivity and provides disruption tolerance to small systems of vehicles utilizing Wi-Fi and TCP on Linux. It is intentionally designed to support systems of 2–16 autonomous robots (such as rovers and drones). However, it is capable of running on any wireless network of Linux devices.

It does this through two independent subsystems. First, BRUNET utilizes a MANET (Mobile Ad-hoc NETwork) with the BATMAN (Better Approach To Mobile Ad-hoc Networking) routing protocol to provide decentralized, multi-hop wireless communications over Wi-Fi. Second, BRUNET provides disruption tolerance to arbitrary TCP connections via a proxy program. BRUNET uses standard Wi-Fi hardware and, it provides disruption tolerance in a completely transparent manner, so programmers can still rely on TCP sockets for their applications. In this way, BRUNET improves and does not replace Wi-Fi and TCP, allowing it to be integrated with very little effort.
The use of a MANET with BATMAN improves wireless communication of vehicles in the network in several ways without any additional hardware. Each node independently acts as access point, allowing arbitrary pairs to form direct peer-to-peer connections. There is no longer any need to move within range of a central access point for a pair of nodes to communicate. One or more nodes may also act as intermediaries, relaying frames between two nodes that are not within direct range of each other.

Disruption tolerance is implemented in the network stack at the application layer, and runs on each host as a user space program. BRUNET effectively allows TCP connections to persist over indefinite network outages. Large buffers (up to 1GB) allow applications to continue sending data even when connectivity is lost.

A TCP connection managed by BRUNET is called a logical connection. This is illustrated in Figure 1.1. Each logical connection externally appears to be a single TCP connection, (A), but actually consists of three distinct TCP connections, (B). At launch, each pair of BRUNET proxy programs establish (or attempt to establish) a connection over TCP. When a client connects to a server, that connection is redirected with Netfilter (NFT) rules to the local proxy program instead of to the remote server. On the server’s host, the proxy program establishes the complementary connection to the intended server. In this fashion, the wireless inter-proxy connection, managed internally by BRUNET, can suffer disruptions without affecting the connectivity state.

Figure 1.1: Components of a logical connection
of the user’s client and server applications. All of this takes place in a manner completely transparent to the client and server applications.
Chapter 2

RELATED WORK

2.1 MANETs

MANETs (Mobile Ad-hoc NETworks) are a class of wireless networks designed for mobile devices. They are characterized by the lack of a central access point or supporting infrastructure and have a variable topology. Nodes may be added or removed with ease, and disruptions are frequent [3]. Each node possesses its own wireless communication capabilities and forms direct connections with other nodes in the network.

With the addition of a specialized MANET routing protocol, multi-hop communications become possible, allowing nodes otherwise out of range of each other to form an end-to-end connection via one or more intermediate nodes. Specializations of MANETs include VANETs (Vehicular Ad-hoc NETworks) and FANETs (Flying Ad-hoc NETworks), although many other types exist.

2.1.1 MANET Routing Protocols

MANET routing protocols have been the subject of extensive research [4], [5], [6], [7]. In their survey of MANET routing protocols, Ramphull et al. classify protocols into four general categories: proactive, reactive, hybrid, and nature-based [3]. With proactive protocols, each node maintains a complete routing table for the entire network. When a new node appears, every node immediately computes a path to it. Proactive routing tends to consume both memory and bandwidth in keeping routing tables updated, and scales poorly because of this. However, routes are consistent and
predictable. A reactive routing protocol takes the opposite approach, determining routes on demand, caching them for later use. Reactive routing experiences high latency in sending packets to a new destination but memory and bandwidth overhead is much lower. Hybrid and nature-based protocols attempt to use the best features of proactive and reactive protocols. For example, some hybrid protocols perform proactive routing locally within designated regions, but use reactive routing between regions. Nature-based protocols are typically modeled after the group behavior of some species (e.g. ants, bees, or birds). Routing roles in the network may be heterogeneous with different nodes taking on certain responsibilities.

Several proactive routing protocols were considered for use in BRUNET: Babel, BATMAN, and OLSR [7]. Babel was originally proposed in RFC 8966, and is based on distance-vector routing [8], [5]. An implementation is available in BIRD (BIRD Internet Routing Daemon) [9]. BATMAN (Better Approach To Mobile Ad-hoc Networking) is developed by Open-Mesh and is included as a module in the Linux kernel [10]. Murray et al. consider it to be a biologically-inspired protocol rather than a strictly proactive protocol (according to the nomenclature given in [3]) [5]. OLSR (Optimized Link State Routing) was at one point very popular [7], [5]. However, a current implementation could not be found (this includes the one used by Murray et al., described in [11]).

2.1.2 Performance of MANET Routing Protocols

Several authors have conducted performance comparisons of Babel, BATMAN, and OLSR, with mixed results. Hernandez et al. compared Babel and BATMAN on Raspberry Pis and found that Babel had higher throughput, though neither was good for large networks [4]. Murray et al. tested all three on embedded PCs and likewise found that Babel had higher throughput; otherwise, the results were inconclusive [5].
Abolhasan et al. found that Babel had the best route repair time, while OLSR had the worst. They also determined that BATMAN provided the highest stability and packet delivery [7]. Finally, Guillen-Perez et al. measured the protocols’ real-world performance on a FANET. They found that Babel outperformed the other two and that BATMAN performed worst due to its slow update frequency [6].

2.2 Disruption-Tolerant Connections

TCP is optimized for networks with brief disruptions in end-to-end paths [12]. Buffer sizes are small and prolonged disruptions lead to connection timeouts. Once ended, a TCP connection cannot be resumed and there is a potential for data loss.

One particular type of disruption occurs when a device changes networks. When a network changes, the IP address of a host typically changes as well, automatically terminating existing TCP connections. *IP mobility* is the ability for mobile devices that frequently switch networks to maintain a connection to a server. Solutions generally fall into two categories: kernel space implementations at the transport layer, and user space implementations at the application layer [13].

Each approach has drawbacks [13]. New transport-layer protocols typically replace TCP. This is often impractical because software built on TCP must be modified, sometimes extensively. On the other hand, user space applications incur overhead due to copying data to and from the kernel. However, these can be less intrusive because they leverage the existing network stack and can even operate in a manner transparent to existing software.

One such solution is SMSL (Session-based Mobile Socket Layer), developed by Kimura et al., which has much in common with BRUNET [13]. SMSL is a user-space frame-
work. It encapsulates one or more sequential TCP connections to an online server as a session, seamlessly detecting disruptions and reestablishing connections. It can function completely transparently to application software by intercepting socket system calls. However, unlike BRUNET, SMSL is primarily aimed at providing IP mobility and requires the support of a centralized online node for synchronization between client and server nodes.

Another similar project is Cedos, developed by Moon et al. [14]. Cedos implements delay-tolerant Wi-Fi offloading. Wi-Fi offloading is a feature whereby mobile apps opt to use Wi-Fi instead of cellular data for large downloads. Like SMSL, Cedos handles changes in a client’s IP address, network disconnections, and long delays between reconnections. However, it does this by implementing a custom transport-layer protocol which is accessed via a sockets-like API. Because delay-tolerant downloading would require significant changes to existing server-side infrastructure, the authors also describe an online proxy server that fetches the download from the real server and forwards it in a delay-tolerant fashion to client programs when they connect.

2.3 Store-Carry-Forward

*Store-carry-forward* (SCF, also called store-and-forward) is a technique described by Khabbaz et al. in their comprehensive survey of literature on disruption-tolerant networking [12]. In a MANET, a given source-destination pair may only rarely have a contemporaneous end-to-end connection, or this connection may be constantly disrupted. With SCF, data is relayed to intermediate nodes instead of to the destination node direction. The intermediate nodes store the data and forward it to the destination node or other intermediate nodes.
Tsuru et al. explore scenarios where SCF is applied to VANETs [15]. Vehicles act as relay nodes, and data is exchanged between them when they are nearby. Data can also be exchanged with roadside stations and existing infrastructure. The authors point to use cases such as coordination among fleets of self-driving cars to increase safety and efficiency, or offloading data for intensive processing that cannot be done on a particular vehicle.
3.1 Overview

BRUNET can be characterized as a MANET with disruption tolerance for contemporaneous end-to-end connections. It may be contrasted with a store-carry-forward implementation in that the latter allows for data transfer between hosts that do not experience contemporaneous end-to-end connectivity.

Transparency is the central design goal for BRUNET’s disruption tolerance on TCP sockets. While this accounted for a large portion of the development time and often limited design flexibility. However, the tradeoff is worthwhile due to the following benefits:

- Applications are able to use the standard sockets API for creating, managing, and closing TCP connections.
- Aside from disruption tolerance, sockets operate normally. The precise meaning of this is discussed in Section 3.4.
- Proxy programs are configurable with simple configuration files that specify the connections to be managed. They do not require extensive command-line parameters or an API.
- Connections not marked to be managed completely bypass BRUNET and operate normally.

BRUNET has the following limitations:
• It supports, by default, up to 16 nodes on the network.

• It supports, by default, up to 16 simultaneous client applications and 16 simultaneous server applications on each node.

• It is designed for low-throughput applications, although nothing inherently prevents large file transfers or video streaming.

• It is not optimized for time-sensitive communications.

• As a first-attempt project, efficient designs are used when they don’t add excessive complexity.

BRUNET is designed to run on a Raspberry Pi 4 (or equivalent) running Raspberry Pi OS (32-bit) based on Debian 11, “Bullseye” (released November 2021). It makes use of many Linux-specific features and is not intended to be portable. There is no inherent reason why it cannot run on other processor architectures or non-Debian Linux flavors, but it has not been tested on those.

**Terminology**  A running instance of BRUNET on a particular node is called a *proxy program*. The term *user* applies to things relating to client or server applications whose connections managed by BRUNET. Thus, these are called user programs or applications. A user socket is a file descriptor belonging to a BRUNET proxy program that connects to a user program. BRUNET manages connections conforming to a many-to-one client-server architecture. Unqualified, the terms *client* and *server* refer to the user applications at the endpoints of the TCP connection.
3.1.1 Proxy Program Design

Figure 3.1 shows a bird’s-eye view of the proxy program architecture. Arrows show flow of data. One or more user applications establish TCP connections to the proxy program. Data read from connections is cached. This constitutes what can be termed the “user-facing” interface of a proxy program. At some point, data from the caches is encoded in a custom packet format. These packets are sent to the appropriate peer proxy program on another host. This is called the “peer-facing” interface. Note that the proxy programs on the far right of the diagram reside on different hosts. Each has the same architecture as the proxy program detailed in this figure.

Figure 3.2: Proxy program: interface with user programs
The user-facing interface is shown in Figure 3.2. Each proxy program has a single socket, \texttt{USER LSOCK}, which listens for connections from user client applications (redirected by NAT rules in Netfilter). (In this diagram and others, a socket is designated with a box with two arrows, indicating the two channels for reading and writing.) Each new client connection causes a new logical connection to be created. In addition, the proxy program on the host where the server resides is notified, and that proxy program attempts to connect to the server. In both cases, several resources are allocated for each logical connection. A socket, \texttt{USER SOCK}, is used to write and read data to and from the user program (which manages the complementary socket). Each logical connection manages a pair of cache files, which store data for the read and write channels of \texttt{USER SOCK}.

Data originating at the local user application is read from a cache and eventually encoded in packets, to be sent to peer proxy programs. Data intended for the local user application (from its counterpart on another host) arrives in packet form from other proxy programs, is written to a cache and eventually written to \texttt{USER SOCK}.

A stream of data flowing from the client to the server is said to be flowing in the \textit{forward} direction. The other stream flows in the \textit{backward} direction. This convention applies to both caches and packets. This serves as an host-independent means of identifying the destination for data in a logical connection.

Additional logical connections each have their own independent user socket and cache pair. Note that a user application may use more than one TCP connection (common when it is a server). In that case, each TCP connection corresponds to a single logical connection.

The peer-facing interface is shown in Figure 3.3. Each proxy program has a single socket, \texttt{PEER LSOCK}, which listens for connections from proxy programs on other
hosts. It maintains a single connection to each other proxy program via the socket PEER SOCK (there is one peer socket for each other proxy program). This may be accepted via PEER LSOCK, or it may come from connecting to a peer (Sections 3.8.1 and 3.8.2).

Outgoing packets associated with the various logical connections tracked by the proxy program are all appended to an output buffer (OBUF) associated with that peer socket. This includes the data packets indicated in Figure 3.2, but also additional command packets used to communicate information about a logical connection with another proxy program. Bytes in the OBUF are eventually written to the peer socket. Bytes read from the peer socket are copied into the input buffer (IBUF), which holds at most one packet at a time. With each incoming packet, the appropriate action for the associated logical connection is taken before the next packet is read.

### 3.1.2 System Scaling

Figure 3.4 illustrates several important behaviors of BRUNET when scaled past a single client and server, alluded to earlier. It shows a snapshot in time of a network with three hosts, thirteen clients, and four servers.\(^1\) Each line represents a single

---

\(^1\)This configuration is taken from the [limit3] test in Appendix B.
TCP connection. First, each client connection to a proxy program (nominally to the remote server) corresponds to a single logical connection. Complementing this, each server has one connection to the local proxy program per client. Each pair of proxy programs has exactly one connection (a bold line), through which all data flows.

### 3.2 Underlying Ad-Hoc Network

To create an ad-hoc network, BRUNET utilizes IBSS (Independent Basic Service Set) support in the Linux kernel’s *mac80211* subsystem (official documentation can be found at [16]). IBSS networks are defined by the IEEE 802.11 standard ([17] ch. 6). It offers a basic feature set that enables decentralized peer-to-peer communications. However, an IBSS network does not include the capability for an intermediate node to forward frames to an out of range device (i.e. multi-hop functionality). Appendix Section A.1 describes the configuration process for the ad-hoc network in detail.

Multi-hop routing for the ad-hoc network is provided by BATMAN. The primary alternative considered was Babel, via the BIRD tool (a modern implementation of OLSR could not be found). Despite inferior performance in some areas, BATMAN
was chosen over Babel because BIRD’s Babel support was at a very early stage and it only supported IPv6.\textsuperscript{2} The BATMAN Linux kernel module, \textit{batman-adv} is installed by default on Raspberry Pi OS. As part of the kernel, the present author believes it has better prospects than BIRD for long-term support. Additionally, BATMAN’s documentation is more extensive than BIRD’s Babel implementation. Appendix A.2 describes the configuration process for BATMAN in detail.

### 3.3 Configuration File

When executed, proxy program is passed the path to a configuration file as a command-line argument. This is the primary interface to BATMAN and is in keeping with the overarching goal of reducing intrusiveness. Several alternative interfaces were considered: an API, command-line arguments, and user input (whether delivered interactively via \textit{stdin} or otherwise). However, configuration files have several advantages. Files are extremely common for configuring services in the Linux networking environment. They also persist across program runs and system reboots. Command-line arguments quickly become cumbersome when providing the volume of information necessary for configuration. User input makes little sense since BRUNET is intended to be run as a background process. Providing an API as the primary interface is inconsistent with the goal of completely transparent operation. Of course, with future work, any of these could serve as a useful secondary or auxiliary interface for BRUNET.

\textsuperscript{2}This is noted in version 1.6 of BIRD’s documentation, on which the decision was based. [18]. At the time of this writing, BIRD’s Babel support has developed a great deal and is likely now a viable alternative [9].
3.3.1 Design Concepts

The configuration file primarily specifies which connection endpoints are to be managed by BRUNET. These are identified in the file as triplets: a client IPv4 address, a server IPv4 address, and a server port. All TCP connections matching all three characteristics in a triplet are managed by the proxy program. A single triplet, therefore, may describe simultaneous and/or sequential client connections to a given server. It also allows for multiple server programs to be executed in sequence, provided they all bind to the same address and port. Connections not matching any triplet in the file are ignored by BRUNET and operate normally. Non-TCP communications are ignored. Thus, a single user program may utilize some connections managed by BRUNET while operating others normally.

The configuration file is designed so that each proxy program uses the same file, with only minor changes. In particular, each file specifies the IPv4 address of “this device.” This, by extension, indicates which network interface should be used by the proxy program. Because it is referenced statically in the file, the IP address of the desired interface must be statically assigned.

Other than the IP address for “this device,” the other content in the configuration file must remain identical across all nodes in the network. This critical assumption permits certain optimizations to be made in the implementation. This is because each configuration file describes not just the local host but all hosts relevant to BRUNET. (The small number of nodes in the network prevents the file from becoming unwieldy.) Therefore, there is no need for any sort of initial peer discovery process.

Because all relevant IP addresses are included as endpoints of at least one connection, each proxy program can maintain a list of all peers in the system. Because the peers
are stored in a list, an index into this list is sufficient to identify a peer. Accordingly, this index is known as a peer ID.

### 3.3.2 Implementation

BRUNET uses YAML for its configuration file [19]. This was chosen because, in his experience, the present author considers it to be intuitive, flexible, and widely used. Using the libfyaml library eliminated the need to write a custom file parser. This is one of the recommended C language parsers on the official YAML website, and it has satisfactory documentation [20], [20].

On launch, a function parses the file once and copies values into a C structure of configuration parameters, which mirrors the YAML file structure, for convenient processing. Values from this structure are copied into various data structures that are used throughout the program’s life. The triplets identifying managed connections are used to construct Netfilter rules (Section 3.4.3). In particular, each triplet is analyzed and unique IP addresses are inserted into an array. In order to ensure that all nodes reconstruct the same array, the current node’s IP address is included. The list is then sorted. Thus, an index into this array globally identifies a particular node and can be used to look up its address or any other state information. The index of a peer can be determined efficiently from its address using a binary search. This has the added benefit of removing any dependency on the order of triplets listed in the configuration file.
3.4 Connection Redirection Mechanism

The fundamental problem with connection redirection is how to do it transparently. That is, how a client application on machine (address $A$, system-provided port $p$) can connect to a server application (address $B$, well-known port $P$) in such a way that the actual inter-machine connection from $A$ to $B$ is handled by the proxy programs on the respective machines. Yet, the connection must appear completely normal from the perspective of the client/server endpoints. Any solution to this problem must satisfy the following constraints:

- The client should be able to connect to the server by passing the tuple $(B, P)$ to connect(2) on a socket created with socket(2). The server should simply passes $(B, P)$ to bind(2) on its listening socket and call accept(2) to complete the connection. These should be the only steps required to establish a connection on the part of a user program.

- Calling getsockname(2) on the client side should return $(A, p)$. Calling getpeername(2) on the server side should return $(A, q)$, where $q$ is a system-assigned port number not necessarily equal to $p$.

- Calling getpeername(2) on the client side should return $(B, P)$. Calling getsockname(2) on the server side should return $(B, P)$.

- If either the client or server closes its end of the connection, the other side should receive the normal notifications of that (e.g. EOF, poll/signal events, etc.).

\footnote{That the port number changes does not violate the transparency requirement in any meaningful way. It is assumed that the server does not rely on a client binding to any particular port number.}
• The actual inter-machine TCP connection should be able to be shut down and reestablished without the client or server seeing any change in the state of their sockets.

### 3.4.1 Solutions Considered

Several potential solutions (or components thereof) that were considered but ultimately discarded are discussed here. Raw sockets are a mechanism to read and write raw IP datagrams [21]. The Linux kernel’s TUN/TAP device driver offers a way to send and receive Ethernet frames or IP datagrams [22]. Both of these have the advantage of allowing the programmer to directly modify packet contents. However, the programmer must re-implement certain network operations normally performed by the kernel, a non-trivial task. Another alternative considered was to use two IP addresses, one for proxy program communications and another for user application communications. Routing rules could be set based on the “proxy” IP address. This was also discarded since it increases BRUNET’s intrusiveness by placing special requirements on user programs.

The remaining attempted solutions consisted in leveraging the Linux kernel’s Netfilter subsystem (also called nftables), a framework for packet filtering and routing [23]. The organization responsible for Netfilter, the Netfilter Project, originally developed the iptables family of tools, used to configure Netfilter. While iptables is a mature project, it has been superseded by the nft(8) command-line tool. Like its predecessor, nft(8) has extensive, excellent documentation and a powerful instruction syntax for easy testing of various rules [24]. Two approaches were attempted: a transparent proxy (TPROXY) and network address translation (NAT).
3.4.1.1 TPROXY

The term “transparent proxy” is used to refer to two related mechanisms enabled by the same socket option: IP_TRANSPARENT at the IPPROTO_IP (equivalently, SOL_IP) level [25]. This option enables transparent proxying on the socket, which permits binding to a nonlocal address. This is useful on the server side, because the proxy program connecting to a server could masquerade as the original client. The second use of IP_TRANSPARENT is to allow the socket to work in conjunction with the TPROXY Netfilter rule.

The TPROXY Netfilter rule redirects packets to a different port and/or address without modifying packet headers [24]. Because no packet data is changed, a call to getsockname(2) reveals the original port and address. This rule, in theory, satisfies the transparency constraint on the client side. However, through testing it was discovered that this rule can only apply to packets entering a system (the prerouting hook) and not to those originating with local programs (the output hook). Oddly, this limitation is not explicitly mentioned in the documentation (i.e. [24]). Regardless, this precluded any further use of transparent proxying.

3.4.1.2 NAT

Network Address Translation (NAT) is typically used to allow hosts on a local area network to interface with the global internet. Typically run as a router service, NAT translates local addresses to the router’s globally unique address. Packets returning from a machine on the internet are sent to the router, which then forwards it to the

4Specifically, attempting to use the output hook with TPROXY would fill the screen with a large number of syslogd messages that were generally indecipherable.
proper host on the local network. Further details on this common use of NAT can be found in [26], pp. 394-397. However, NAT is also useful for BRUNET.

NAT has two forms: DNAT (Destination NAT) and SNAT (Source NAT). A DNAT rule has a similar effect to a TPROXY rule, except that a packet’s destination address and/or port are actually modified so that the original destination address and port are not recoverable from the packet data alone. However, because Netfilter tracks connections redirected with NAT, the original destination address is accessible via `getsockopt(2)` with SOORIGINAL_DST at the level IPPROTO_IP. Thus, DNAT can be used to intercept a new client connection and allow it to be accepted by the local proxy program. An SNAT rule modifies the source address and/or port of packets before delivering them to the destination, masking the actual source. Netfilter’s connection tracking ensures that packets sent back to the apparent source are delivered to the actual source. Thus, SNAT allows a proxy program to masquerade as a remote client to a server on the same host.

### 3.4.2 Choosing a Netfilter Library

A number of different interfaces and libraries were considered with the goal of configuring Netfilter.

- `netlink(7)` is a Linux protocol for communication between kernel and user space. The NETLINK_NETFILTER family provides access to the kernel’s Netfilter subsystem. The control Netlink offered was too fine-grained for the purposes of

---

5The SOORIGINAL_DST socket option appears nowhere in official documentation. It is accessible after importing `linux/netfilter_ipv4.h`, although that file does not document the purpose of the socket option. It is referenced, however, by a couple of Stack Overflow posts from 2011: [https://stackoverflow.com/a/6554625/10710003](https://stackoverflow.com/a/6554625/10710003) and [https://stackoverflow.com/a/5814636/10710003](https://stackoverflow.com/a/5814636/10710003).
this project. The man page refers users to a high-level library such as libnetlink [27].

- **libnetlink(3)** is a library used to access kernel routing tables and interfaces with rtnetlink(7). The man page indicates that it exists for internal use and new programs ought to use another library, such as libmnl [28], a minimalistic library aimed at simplifying generic Netlink programming [29]. It is not targeted at Netfilter.

- **libnftnl** is a library built on libmnl for accessing the kernel Netfilter subsystem. It is used by the nftables framework. This library would have been ideal for a lightweight, efficient implementation, but it would have required substantial time to implement due to sparse documentation [30].

- **libnftables(3)** is the high-level library provided by the Netfilter Project for application integration with nftables. It provides a simple parser that can interpret commands very similar to the those specified in the man page for the nft command-line tool. This mean it was almost trivial to write the C code for configuring Netfilter rules based on nft commands. The library’s JSON parsing features could also be of use in future work [31]. Its non-development version was installed by default.

### 3.4.3 Netfilter Integration

On launch, a proxy program reads its configuration file (Section 3.3), and parses the list of triplets specifying the endpoints of each connection to be tracked. This list is used to directly generate the NAT rules. If the current host is a client in a connection, a DNAT rule is added to redirect packets to the proxy program. This rule matches only IP packets originating on the same host and having the destination
address and port of the server. Packets matching this rule are redirected to the local proxy program, which is listening on a known port. To ensure that only local programs’ connections are redirected, the proxy program binds its listening socket to the loopback address, 127.0.0.1. If the current host is a server in a connection, an SNAT rule is added to allow the proxy program to appear as a remote client. This rule matches only IP packets originating on the same host and having the destination address and port of the server. Packets matching this rule are modified so that their source address is that of the client in the connection.

Upon a clean program termination, which normally is triggered by the SIGINT signal (Ctrl-C), any rules added by BRUNET are deleted by the signal handler. The program is also be terminated cleanly with SIGTERM (the default signal generated by kill(1)), SIGABRT (from assert(3)), or SIGHUP (from a terminating parent terminal).

### 3.5 Cache Files

BRUNET requires large caches that can be grown or shrunk and allow for random accesses. Several general implementation approaches were considered: allocating a buffer from the heap, using a file with read(2)/write(2)/lseek(2), and memory mapping a file with mmap(2). Memory mapping was chosen due primarily to its ease of use and flexibility, particularly with random reads and writes. Though this is not currently used in BRUNET, a memory map can be used to refer to a portion of a file, which is useful if the region exceeds the available RAM and swap space. Otherwise, performance benefits are unclear without profiling (see [32] and [33] p. 315).

A single cache stores a sequence of bytes for a single direction or stream of a logical connection. Thus, two caches are needed for each full-duplex logical connection. When a user program writes data to a socket, the proxy program writes the data
into the corresponding cache at its write offset (or head). In order to forward the data to the destination socket, bytes are read, beginning at the read offset. (Multiple independent read offsets are supported; this is described below.) When the bytes are delivered and acknowledged by the receiving end, the bytes in the cache are freed, beginning from the ack (acknowledge) offset. Offsets are incremented automatically during their respective operations.

To allow reuse of acked memory for acknowledged bytes, a block structure is imposed on the cache file. Blocks are equal in size and aligned to a memory page for efficiency. The first block in the cache is used to store metadata about the cache, including read/write/ack offsets and pointers to two linked lists of blocks: a free list and an active list. The first eight bytes of a block are a header, used to store the offset of the next block in the list. While this adds some complexity to the read/write/ack operations, since headers must be jumped over, it prevents having to separately allocate space for the linked lists. All blocks are initially in the free list. When a write occurs, a block is removed from the free list and appended to the active list. When the current block in the active list fills up, a new block is appended and the write operation continues in that block. When all the bytes in the oldest active block are acked, that block is prepended to the free list. Read operations do not cause any blocks to be reorganized.

In order to support BRUNET’s future development into a store-forward network, each cache supports multiple independent read offsets, identified by a peer ID. In an store-and-forward network, data may be written to a peer that is not the endpoint of a logical connection. There is only one ack offset, indicating data acknowledged by the final destination. An ack operation that advances the offset past a read offset also advances the read offset. This prevents reading data from the cache that has already been delivered.
To aid the cache abstraction, logical read, write, and ack offsets are maintained. The offsets used internally by the cache are raw offsets from the base memory-mapped address. Neighboring blocks in a linked list are not necessarily contiguous in the actual file, and there is overhead due to block headers. Thus, the $n^{\text{th}}$ byte in the stream is not at raw offset $n$ in the file. However, it is extremely useful as an abstraction (and for some internal calculations as well) to have a logical write offset that corresponds to the number of bytes written to the cache (and likewise for the read and ack offsets).

### 3.6 Managing Sockets

The BRUNET proxy program uses an event-driven architecture. Linux provides several independent system calls to support event-driven programming: `select(2)`, `poll(2)`, and `epoll(2)`. All of these accept some list of file descriptors to monitor for data to read or write or the presence of an error on the socket. `poll(2)` was chosen for its simplicity, although there is no reason why the other two could not have been used. `poll(2)` requires an array of `pollfd` structures, each of which contains a file descriptor to be monitored, a field of events to include, and a field of events actually present on the descriptor. `poll(2)` returns several events, the most important of which are POLLIN, POLLOUT, POLLHUP, POLLRDHUP, and POLLERR. These represent for a given socket, respectively, that data is available to be read, data may be written, the peer closed its end of the connection (hangup), the peer closed its end of the connection or only the writing half of the connection, the connection experienced an error (typically indicating simply that the connection was closed). Internally, the proxy program is structured around handling these events for each socket.

All sockets in the proxy program are monitored by a single `poll(2)` invocation:

1. The *user listening socket*, for user application connections redirected by Netfilter
2. The *peer listening socket*, for connections from other proxy programs

3. Up to 16 *peer sockets* (via 2 above, and `connect(2)`)

4. Up to 16 *user client sockets* (via 1 above)

5. Up to 16 *user server sockets* (via `connect(2)`).

For maximum efficiency, nonblocking I/O is used for all operations. To prevent spurious POLLOUT events, POLLOUT is enabled on-demand for each socket, that is, only when data is available to write.

### 3.7 Packet System

All data exchanged over an inter-proxy connection, with the exception of the synchronization bytes (Section 3.8.2), is encapsulated in a packet. Each packet belongs to a logical connection. A NEW packet is sent when a new logical connection is created, and informs a peer of this connection, so it can prepare to accept data. A DATA packet carries data to a peer. An ACK packet is sent in response to a DATA packet, acknowledging it. Two packets relate to closing a logical connection: an EOD packet indicates the end of data in a particular stream (i.e. all data has been read from one user socket), while a NO_WR packet indicates that no more data can be written to the destination socket for a given stream.

![BRUNET Packet Structure](image)

**Figure 3.5: BRUNET Packet Structure**
All packets follow the format given in Figure 3.5: a 16-byte header with a variable-length payload. All multi-byte values are in host byte order. The 32-bit logical connection ID, $LC_ID$, is globally unique and indicates which logical connection in the system the packet belongs to (Section 3.7.1 describes how this is created). The 8-bit direction field ($Dir$) indicates an absolute direction the packet travels. A packet traveling from the client to the server is traveling forwards. A packet traveling from the server to the client is traveling backwards. The 64-bit $Offset$ field indicates an absolute byte position position in the associated stream. Note that the direction a packet travels (the $Dir$ field) does not necessarily match the direction of the stream the packet is for. For example, an ACK packet acknowledging data sent from the forward stream (client to server) travels backwards (server to client). The 16-bit $Length$ field specifies the length of the payload in bytes, which may be anywhere from 0 to 1008 bytes. Finally, the 8-bit $Type$ field is an integer indicating the packet type.

Each of the packets corresponds to various stages in the life cycle of a logical connection. When a logical connection is created, a NEW packet is generated. The DATA and ACK packets manage the transfer of data during its lifetime. And a combination of EOD and NO_WR packets closes it.

Packets are not sent or received directly from the peer socket. Instead, two buffers are used as intermediaries. Outgoing packets are “staged”, that is, appended to a circular output buffer, assuming it has enough space. Care is taken that if multiple packets are pending, they are not sent out of order (e.g. a DATA packet before the NEW packet for that logical connection). This is discussed in Section 3.7.5. Incoming packets are handled one at a time in an input buffer, described in Section 3.7.6.
3.7.1 Creating a Logical Connection

A new logical connection is created when the user program listening (Section 3.6) socket receives a POLLIN event. A new user socket is created by `accept(2)` and begins to be monitored by `poll(2)`. A logical connection *entry* is stored on that host (Section 3.9.1). Each logical connection entry consists of several fields. The 32-bit *logical connection ID* is really the composite of two values: the 4-bit peer ID of the current host, and the 28-bit *instance* number. The instance number is a host-specific number initialized to zero on launch and incremented each time a new client connection is formed.\(^6\) The combination of these two pieces of information is sufficient to uniquely identify any logical connection the network. Due to NAT redirection, the original destination (IP address and port) of the user connection must be retrieved from the kernel using `getsockopt(2)`, as discussed in Section 3.4.1.2. Then, the peer IDs of both the client (the local host) and server, as well as the server port are stored in the logical connection entry.

The NEW packet’s LC_ID field is that of the newly created logical connection. The offset is set to zero, the direction is forward. The payload length is set to 12 bytes. The payload consists of three 32-bit values, the corresponding triplet originally found in the BRUNET configuration file: the client’s peer ID, the server’s peer ID, and the connection port. Once constructed, the packet is staged for sending to the server in the packet output buffer (Section 3.7.5). Once this occurs, DATA packets may be sent.

Upon reception of a NEW packet, the logical connection entry present on the originating host is reconstructed and stored. (This entry is retrieved when processing

\(^6\)This is implemented simply as a static variable inside the function that handles new user client connections.
additional packets for this logical connection.) The user server socket is created and a connection to the server is attempted. If the connection fails, the close sequence (Section 3.7.4) is initiated. Otherwise, data is can be sent or received from that socket.

### 3.7.2 Sending and Receiving Data

Once a logical connection entry is established at an endpoint (client or server), data transmission is possible. A POLLIN event on the user socket allows at most 1008 bytes of data to be read from the socket and stored in the appropriate cache for that logical connection — forward, if the data is read from the client socket, and backward, if the data is read from the server socket. When the peer socket becomes writable, a DATA packet is formed. The direction of the data packet is the same as the cache. The payload length is maximized with several constraints: the maximum packet length (1024 bytes), the amount of data actually available to be read from the cache, and the space in the output buffer (Section 3.7.5). The minimum payload length is 1 byte. Once constructed, the packet is staged for sending in the packet output buffer.

When a DATA packet is received, the logical connection entry for that packet is retrieved, and the payload is written to the cache.\(^7\) When the user socket becomes writable, data is read from the cache and copied into the user socket’s circular output buffer, the contents of which are then written to the socket. This intermediate buffer is necessary because the amount of data that the \texttt{write(2)} system call will actually

\(^7\)At the time of this writing, the offset is not actually checked. The design of the packet buffering system and the connection recovery/syncing mechanism eliminates the need for this field in a DATA packet. However, it would become useful if BRUNET were expanded to be a store-carry-forward network, where a node could receive a DATA packet with a payload that had already been received in whole or in part from another peer.
accept cannot be determined. The amount of data written is limited by both the empty space available in the user socket’s output buffer, as well as the amount of data actually available for reading in the cache. The user output buffer is discussed further in Section 3.7.5.

### 3.7.3 Acknowledging Data Delivery

An ACK packet is generated after data has been copied from the cache to the user output buffer, at which point the data is considered “delivered” to the user. This is sufficient because the output buffer is guaranteed to write all its contents to the socket unless the socket is closed first. In this case, the ACK packet is simply irrelevant and does not cause any issues. The ACK packet’s direction is opposite that of the cache that was read from. Its offset field indicates the total count of bytes delivered (according the definition above). The payload is empty for an ACK packet and the length is set to zero.

Upon the reception of an ACK packet, the offset is passed to the appropriate cache to update its ack head. This allows the cache to free acknowledged bytes. Additional actions may be taken at this point if these are the last bytes in the stream (see Section 3.7.4).

### 3.7.4 Closing a Logical Connection

Closing a logical connection is defined as destroying all resources related to a logical connection — the user socket and the logical connection entry, including its caches — such that any future packets for that logical connection received at that host cannot be processed and are ignored. It is desirable to close a logical connection only after two flags have been set: (1) FIN RD, reading has finished, and (2) FIN WR writing
Figure 3.6: Closing a logical connection

has finished. ("Reading" and "writing" are with respect to the local user socket, rather than the peer across the logical connection.)

These events are triggered by the user program closing its side of the connection. There are a number of ways this can occur, as shown in Figure 3.6. Because TCP is a full-duplex connection, the user may choose to only shut down the reading or the writing end, but not both, as with the `shutdown(2)` system call. Or, they may choose to shut down both, as with the `close(2)` system call. From the perspective of the user socket monitored by `poll(2)`, these actions are signaled by three events: POLLERR, POLLHUP, and POLLIN with EOF (`read(2)` returning zero). According the `poll(2)` manual page, these are not necessarily mutually exclusive [34]. POLLHUP indicates that the other end of the connection was closed, but that data may still be read from
the socket buffer (until reaching EOF). POLLERR indicates an error condition, and may occur after writing data to a channel when the counterpart has stopped reading.

The FIN_RD and FIN_WR flags are set when their respective packets, EOD and NO_WR, are actually staged in the output buffer. Both POLLERR and POLLIN with EOF set a flag indicating that an EOD packet may be sent. While no new data will be added to the cache after this point, the data in the cache must be emptied. Thus, the EOD packet is only sent once all data sent to the peer has been acknowledged. Only at this point is FIN_RD set. Both POLLERR and POLLHUP indicate that writing has finished, namely, no additional data can be written to the user socket. These cause a NO_WR packet to be sent to the peer immediately, also setting FIN_WR.

The reception of these packets also causes the flags to be set. When an EOD packet is received, FIN_WR is set, but only after all data in the cache has been delivered. Also, the write end of the user socket is shut down. This causes the user program to receive EOF. This allows the user program to handle this appropriately, which typically involves closing its end of the connection after it has finished sending data. When a NO_WR packet is received, FIN_RD is immediately set.

On either host, once both FIN_RD and FIN_WR are set, then the logical connection is destroyed.

### 3.7.5 Buffering Output for Writing

When `write(2)` is invoked with a socket, the number of bytes it accepts cannot be guaranteed. Therefore, a packet, once generated, cannot be guaranteed to send

---

8 An apparent solution to this is using the SO_SNDBUF option (see [36]). This prevents `poll(2)` from indicating a socket is writable until a certain number of bytes are available in its output.
in a single call to `write(2)`. But because packets are variable length and this length is specified in the header, a packet must be transmitted in its entirety so the receiving end can delineate packet boundaries. The most straightforward solution to this is to pair an intermediate output buffer (OBUF) with each peer socket.

The OBUF is circular and internally uses three pointers: \( R \) (read), \( W \) (write), and \( A \) (acknowledge). Each points to the next position that will be operated on. Further, operations are relative to the buffer, not relative to the socket. Thus, if \( W = 2 \), the byte at position 2 in the buffer is empty and will be the next to be written to. Likewise, \( R \) refers to the next byte that can be read, and \( A \) refers to the oldest unacknowledged byte. This is illustrated in Figure 3.7. The buffer is considered empty when \( A = W \) (Figure 3.7(C)). The buffer is considered full when \( A \) is at the position immediately prior to \( W \), accounting for wraparound (Figure 3.7(B)). This comes with the cost that a single byte in the buffer can never be written to, but it removes the need for additional logic to disambiguate the empty and full cases. \( R \) lies between \( A \) and \( W \), and can be equal to either. Because this is a circular buffer, wraparound must be accounted for when incrementing values or calculating differences between pointers.

![Figure 3.7: Pointers in a 12-byte circular output buffer](image)

The OBUF is circular and internally uses three pointers: \( R \) (read), \( W \) (write), and \( A \) (acknowledge). Each points to the next position that will be operated on. Further, operations are relative to the buffer, not relative to the socket. Thus, if \( W = 2 \), the byte at position 2 in the buffer is empty and will be the next to be written to. Likewise, \( R \) refers to the next byte that can be read, and \( A \) refers to the oldest unacknowledged byte. This is illustrated in Figure 3.7. The buffer is considered empty when \( A = W \) (Figure 3.7(C)). The buffer is considered full when \( A \) is at the position immediately prior to \( W \), accounting for wraparound (Figure 3.7(B)). This comes with the cost that a single byte in the buffer can never be written to, but it removes the need for additional logic to disambiguate the empty and full cases. \( R \) lies between \( A \) and \( W \), and can be equal to either. Because this is a circular buffer, wraparound must be accounted for when incrementing values or calculating differences between pointers.

buffer. Unfortunately, the man page says that this socket option is set by default to 1 byte and cannot be changed on Linux.
When incrementing, the remainder operator is used. For example,

\[ W_{\text{new}} = [W + \text{(increment)}] \%(\text{buffer length}) \]

The following piecewise equation is used to calculate the number of unread bytes, the difference between \( W \) and \( R \).

\[
\text{(unread bytes)} = \begin{cases} 
  W - R & \text{if } R \leq W \\
  W - R + \text{(buffer length)} & \text{if } R > W 
\end{cases}
\]

Similar logic is used to calculate other differences.

When POLLOUT occurs on the associated peer socket, packets are constructed and written to the OBUF. All logical connection entries on that host are looped through in the order that they are returned by the dictionary iterator (Section 3.9.1). Any pending packets for that logical connection are constructed and appended before moving on to the next logical connection. DATA packet payloads are sized according to the constraints in Section 3.7.2. After all logical connections have been processed or the OBUF is filled, its contents are read from it and written to the peer socket.

When \( W < R \), as in Figure 3.7(A), two noncontiguous memory regions in the buffer constitute the bytes to be written. For this reason, the \texttt{writev(2)} system call is used to write both regions atomically, avoiding two separate \texttt{write(2)} calls. Instead of a buffer, \texttt{writev(2)} is passed an array of \texttt{iovec} ("IO vector") structures, each of which consists of a base pointer and a length; the vectors are written in the order they appear in the array [37]. When there is only one memory region (\( W > R \)), \texttt{writev(2)} is called using one vector.

Bytes in the buffer are freed when they are acknowledged by the other side of the TCP connection. This information is derived from the Linux TCP\_INFO socket op-
getsockopt(2) returns a structure of internal TCP values, one of which is `tcpi_bytes_acked`, the number of bytes acknowledged over the lifetime of the connection. A is updated before copying values into the buffer.

The update mechanism is more complex than simply incrementing the A pointer. It first calculates the `ack increment`, the number of bytes acknowledged since the last update. When this value is calculated, it ignores the byte sent for establishing the TCP connection as well as any sync bytes (see [26] p. 239 and Section 3.8.2). The new A value is calculated from `ack increment`. Because an OBUF is designed to persist across multiple peer connections, the `total ack count` maintains the total number of bytes acknowledged on all associated sockets over the lifetime of the program. This value is also incremented using `ack increment`.

### 3.7.6 Receiving and Processing Packets on Input

Unlike the output buffer, only one packet is received at a time on the input buffer (IBUF). Reading a packet into the IBUF has two stages: reading the header and reading the payload. `read(2)` is called at least once for each POLLIN event. During the first stage, only the bytes necessary to read the header are requested. Once the header has been read, the length field is retrieved and that many bytes are requested from `read`. If at any point `read(2)` returns fewer than the bytes requested, the function returns and leaves the remainder to be read on subsequent POLLIN events.

---

9Discovering this initially was rather difficult. It is defined in `/usr/include/linux/tcp.h`, and `#import <linux/tcp.h>` must be used. The field is defined in RFC 4898 as `tcpESTatsAppHC-ThruOctetsAcked` [39]. A discussion of the Linux implementation can be found on the Netdev Archive mailing list [40].
As soon as an entire packet is in the IBUF, it is passed to various functions that perform the appropriate actions for that packet (Section 3.7). The next packet is read and processed on the next POLLIN event.

3.8 Peer Connections

Each proxy program maintains a single connection to each other proxy program. Connections are established on launch and reestablished whenever a disruption long enough to terminate the TCP connection occurs.

3.8.1 Establishing Inter-Proxy Connections

Each proxy program establishes one connection to its peer proxy programs. On launch, each attempts to connect to all of the peers listed in the configuration file, while simultaneously listening for these connections and accepting them as they come in. However, only one connection can exist between peers at a time.

When a socket is set as nonblocking, immediately after creation with \texttt{socket(2)}, the behavior of \texttt{connect(2)} changes [41]. Normally, it returns only after a connection succeeds or fails. In the nonblocking case, it returns immediately. When the connection succeeds or fails, POLLOUT on \texttt{poll(2)} is triggered, and \texttt{getsockopt(2)} is used to check \texttt{SO\_ERROR} to determine whether the connection succeeded. If it succeeded, that socket becomes the peer socket.

When a connection fails, a one-shot timer is set for reconnection. The length is approximately 5 seconds (see discussion below). The \texttt{timerfd\_create(2)} family of system calls is used [42]. These create and manage a timer tied to a file descriptor. The file descriptor is made readable when the timer expires. This enables easy integration...
with poll(2), because the file descriptor uses the slot of the peer socket in the array passed to poll(2). When the timer completes, a new connection attempt to that peer is made. If that fails, a new timer file descriptor is created and the process repeats.

The accept system call either returns immediately with a valid socket or returns an error. If a nonnegative file descriptor already exists for that peer (via connect(2) or timerfd_create(2)), it is always closed. If the file descriptor refers to a non-established connection (i.e. POLLOUT has not been triggered), close(2) ends the attempt. If the file descriptor is for a timer, the timer is canceled.

An infinite reconnect loop occurs when two peers attempt to connect simultaneously. Each peer will see the incoming connection, accept it, and close the previous file descriptor, from connect(2). But, if both proxy programs do this simultaneously, both connections will be closed. Each side will then receive notification of the termination, and set a timer to reconnect, repeating the whole process. To prevent this, the reconnection intervals are lengthened by 0–9 tenths of a second, selected randomly. Thus, it is unlikely that two peers choose the same interval. If they do, the situation is eventually resolved in subsequent connection attempts. An alternative solution considered was to have the peer with the lower IP address attempt to connect while the other wait. However, this would have required a redesign of several other components. Timer randomization provided a simple, if imperfect solution.

After a connection is established to a peer, a final synchronization step is needed. This is discussed in Section 3.8.2.

### 3.8.2 Resuming Inter-Proxy Connections

When a peer connection fails, the socket is closed and the reconnection sequence described in Section 3.8.1 commences. However, the OBUF and IBUF are maintained
in their current state. When the connection is reestablished, each stream must be *synchronized* before any packets may flow in that direction.

The synchronization process consists in each side sending to the other the number of bytes actually received on that end over all previous connections. (For an initial connection, this value is 0.) This value is used to update the *total ack count* in the OBUF. This is necessary because, when a connection is disrupted, an acknowledgment may be lost or simply not sent, due to TCP’s delayed acknowledgments ([26] p. 247).

The actual number of bytes received on one end is often greater than the number of bytes acknowledged at the sending side. Once the synchronization value is received, transmissions resume from the OBUF and data may be received at the IBUF.

### 3.9 Other Components

#### 3.9.1 Dictionary for Logical Connections

Logical connections are stored in a *dictionary* data structure, implemented as a hash table. The dictionary uses 32-bit unsigned integers as keys and stores opaque pointers.

The dictionary supports insert, delete, and lookup functionality, as well as iterators. Logical connection IDs (Section 3.7.1) are used as keys. Logical connection entries are allocated from the heap and the pointers are stored in the dictionary.

The dictionary is implemented as a hash table with a fixed number of buckets. Collisions are handled using chaining with a linked list. Since the number of logical connections is not large, the number of buckets is 64. The hash function is derived from Knuth’s discussion of hash functions ([43] pp. 516-519). A cyclic shift is performed on each byte of the key and they are XORed together. The resulting value
is multiplied by a large prime integer. That value modulo the number of buckets produces the bucket index.

Each item in the dictionary is a member of a single linked list. This list is independent of the chaining lists used to avoid collisions. Items are appended to the list when they are added to the dictionary, and deleted from it when they are removed from the dictionary. An iterator is simply an opaque pointer to a dictionary item. This may be incremented by using provided functions that traverse the linked list and return the new iterator, a pointer to the next item. An iterator may also be used to retrieve the value stored in the dictionary. Iterators are stored outside of the dictionary, so an unlimited number of iterators may exist. However, iterator behavior is undefined when an item is deleted from the dictionary.

3.9.2 Error Handling

Error handling is standardized across most of the code base. Functions that can fail are passed an ErrorStatus structure. This structure simply contains a string, an error message. Various functions are provided to set the error message using printf-like string formatting, with options to prepend or append additional text to an existing error message. For nearly all functions, a negative return value indicates an error, and a nonnegative value indicates success. This permits a deeply nested function to set an error message and return a negative, leaving its parent to handle the error. Its parent could ignore the error or further propagate it up the call stack by returning a negative, optionally prepending or appending a value to the error message first. System calls that set errno(3) can be handled easily as well. A function is provided that leverages strerror(3) to generate an error message based off the error number (e.g. “connection refused”). The function also makes use of a freely available library, errnoname, that converts an error number to its symbolic name as a string.
(e.g. “ECONNREFUSED”) [44]. (While the `strerrorname_np(3)` function nominally provides the latter functionality, during testing it appeared to be nonfunctional [45].)

Finally, functions are provided to display and clear (and free) the error message.

Generally, each file descriptor used by `poll(2)` has an associated error structure. This allows errors to be handled conveniently at the connection level. Additionally, there is a top-level error structure that is used for all other errors.

### 3.9.3 Logging

A custom log message function replaces most uses of `printf(3)`. The logging function uses `printf`-like string formatting, but offers two advantages. First, it allows for the redirection of all log messages to one or more files as alternatives to `stdout(3)` or `stderr(3)` [46]. Second, an argument specifies one of several levels, which in ascending order are `debug`, `info`, `error`, and `critical`. Generally, the `error` and `critical` levels are both used for displaying error messages (Section 3.9.2), with messages directed to `stdout` and `stderr`, respectively. The `info` level is used for messages that should be present in a release configuration. The `debug` level is used for the vast majority of messages. An initialization function sets a global minimum level such that only log messages at that level or higher are displayed while the program runs. When printing, each message is prefixed with a string indicating its level (e.g. “[INFO]” or “[ERROR]”), although this feature can be disabled.
4.1 General Information

BRUNET was tested using three Raspberry Pis, each with a statically assigned address on wlan0.

- Pi A: Raspberry Pi 400, 10.0.0.1
- Pi B: Raspberry Pi 4, 10.0.0.2
- Pi C: Raspberry Pi 4, 10.0.0.3

Each Pi ran a console-only version of Raspberry Pi OS 11, derived from Debian 11, “Bullseye.” The Pis were typically accessed via SSH from a MacBook Pro running macOS 13 Ventura. The MacBook Pro and Pis were all connected to an Ethernet switch, and the MacBook Pro used internet sharing to provide internet access to the Pis without interfering with the ad-hoc network.

4.2 BATMAN Configuration Verification

An experiment was conducted to verify that multi-hop functionality was properly configured in BATMAN. The three Raspberry Pis are referred to as Pi A, Pi B, and Pi C. A cardboard box (approx. 12 × 8 × 6 inches) covered in aluminum foil served as Faraday cage. Pi B, powered by a battery, was placed inside the box and the lid was closed. Pi A was set next to the box with Pi B, such that Pi A and Pi B were in
range of each other. Pi C was moved to the opposite end of the house. The BATMAN
traceroute tool (batctl traceroute, man page at batctl(8)) displays all devices in a
path to a given destination, much like the standard traceroute(1) tool. The following
routes were tested with this setup:

1. Route from B to C: B → A → C
2. Route from C to B: C → A → B
3. Route from B to A: B → A

These results confirmed that multi-hop functionality was active.

4.3 Unit Tests

Unit tests were written to test several components independently (dictionary, cache,
and error system). Tests are compiled into an independent executable by running
make test. All testing code is surrounded by conditional macro blocks so that it is
not included in the normal executable.

4.4 End-to-End Tests

46 tests were performed to verify various program features. The complete test log may
be found in Appendix B. Unless otherwise noted, each test completed successfully.
Scripts used are found in the repository. Most tests fall in one of the three following
categories.
Manual Tests  *nc*(1) was used extensively as client and server to verify basic operations: creating logical connections, closing logical connections, sending a few bytes at a time in each direction.

Sending Files  *nc*(1) was used to send (or receive) files by redirecting *stdin* and *stdout* using the shell. Typically, a known file would be sent and the output would be checked against the same file with *diff*(1). See note on files used below.

Multi-Client Tests  The *tcpserver*(1) program (Appendix A.4) executes the supplied program for each client. It supports multiple simultaneous clients. Running *tcpserver*(1) with *cat*(1) creates an echo server. This made verification easy for clients: the input file would be redirected to *stdin*, the output to *stdout*, and the input and output files would be checked with *diff*(1).

Several files were used during testing:

1. short.txt: a file with the letter “a”. 2B.
2. lorem*.txt: files with “lorem ipsum” text. 446B, 16KB, 156KB.
3. bible.txt: the complete King James Bible, from the Canterbury Corpus (https://corpus.canterbury.ac.nz). 4.1MB
4. test*.in: base64-encoded random text, generated with *openssl*(1)
5. *.in: various random binary files, generated on-the-fly with *openssl*(1)

Testing natural network disruptions was difficult (see paragraph below on a natural disconnect test). However, network disruptions were simulated by adding and removing a filter to drop all incoming frames on *wlan0*. A script was written to toggle this filter at various frequencies (typically with a period of 1–5 seconds).
**Natural Disconnect Test**  
For the last test in Appendix B, BRUNET was tested with SSH. In the configuration file, Pi B Port 22 was set as the server, and Pi A as the client. Then, BRUNET was started using `tmux(1)` so the programs would not receive SIGHUP when disconnecting Ethernet cables from the MacBook Pro. Two SSH sessions were started: one from Pi A (managed by BRUNET) and one from Pi C. Basic functionality was confirmed by typing in the terminal on Pi A, which caused bytes to be sent in both directions. Also the logical connection closed successfully when the `exit(1)` command was issued on the session on Pi A. Next, `top(1)` was executed in each session and left to run. Because `top(1)` updates the screen every few seconds, this ensured transmissions. Pi B (now battery powered) was disconnected from Ethernet, placed in the Faraday cage, and placed approximately 40 feet away in a car. This ensured that it was wirelessly disconnected. After the disconnection was confirmed (Pi A began its repeated reconnect attempts), the keystrokes for quitting `top(1)` and printing “hello” were entered on the now-frozen SSH sessions (`q, echo hello, Enter`). After letting 30 minutes elapse, Pi B was brought back into proximity with the other hosts. The Ethernet cable was attached and the `tmux(1)` session restored to view proxy program log messages to verify that the proxy programs had reconnected. On Pi A (managed by BRUNET), the session resumed and the keystrokes typed 30 minutes earlier were executed. Further commands could be issued without an issue. On the other hand, the session from Pi C had terminated earlier with a “broken pipe” message.

4.5 **Performance Testing**

Basic performance testing was done to determine the amount of overhead induced by BRUNET. `time(1)` was used to record real (“real”), user (“user”), and system (“sys”)
time. Each test was run three times with BRUNET and three times using a normal configuration (without BRUNET), and the results were averaged.

**Test 1** 5 simultaneous clients with an echo server, 10MB input files with random bytes and sending them as fast as possible. Figure 4.1 shows the measured times.

![Figure 4.1: Average times for Test 1](image)

Ratios: 
- real = 0.65
- user = 0.98
- sys = 0.70
**Test 2** 5 simultaneous clients with an echo server, 100KB input files with random bytes and sending them with randomized packet sizes (1 – 512 bytes) and randomized intervals between writes (1 – 10 ms). Figure 4.2 shows the measured times.

![Graph](image)

**Figure 4.2: Average times for Test 2**

Ratios: \((\text{time without BRUNET}) / (\text{time with BRUNET})\)

- real = 0.89
- user = 0.97
- sys = 0.85

Although a third test was performed, sending 1 byte and immediately exiting, this provided unreliable results which are not reported here.
5.1 Future Work

5.1.1 New Features

The most important improvement to BRUNET would be the implementation of store-forward capabilities. Because BRUNET is designed to be compatible with a future store-forward network implementation, most critical features take this into account. In particular, logical connections, packets, caching, and configuration are all compatible with a store-forward network, with no major architectural changes.

The user interface could be improved in several ways. More parameters should be user-configurable (with sensible defaults) from the configuration file. Command-line options could be added for adjusting the log message level without having to recompile.

5.1.2 Performance Improvements

When staging packets, some logical connections may not handled in a timely manner if some considered earlier in the loop are sending large amounts of data. Refer to Section 3.7.5 for more information.

Another improvement would be the refinement of the peer connection recovery mechanism. While the mechanism is functional, it could be more responsive, perhaps
leveraging layer 2 information to determine when a new node appears on the network.

The caching mechanism requires that the entire file be memory-mapped simultaneously. This means cache size is limited by available swap space. Switching to a traditional file I/O approach could be a simpler alternative. A performance comparison should be performed.

5.2 Conclusion

Through its two-tier solution, BRUNET improves the communication capabilities for systems of autonomous vehicles. Decentralizing the network allows nearby vehicles to communicate without needing proximity to a central access point. Adding disruption tolerance to communications can be incredibly useful, but frequently isn’t worth the time investment on small-scale projects. BRUNET does all of this in a non-intrusive manner, allowing developers to focus on the essential tasks for their system.


A.1 Ad-Hoc Network

1. Enable IBSS (ad-hoc) mode on the network interface with the following commands. Although they can be run manually, they should be added to /etc/rc.local so that they run automatically after rebooting. Place them before the exit 0 line (if it is present) so the script doesn’t exit before reaching them. In this example, “PiFi” is the ESSID of the ad-hoc network. “2432” is the frequency (2.432 GHz), which can also be changed. (View more documentation by running iw wlan0 help.)

```
sudo iw wlan0 set type ibss
sudo ip link set wlan0 up
sudo iw wlan0 ibss join "PiFi" 2432
```

2. Configure the interface to use a static IP address (10.0.0.4 in this example) add the following two lines to /etc/dhcpcd.conf.

```
interface wlan0
static ip_address=10.0.0.4/24
```

3. Move (or delete) wpa_supplicant.conf to prevent auto-joining an existing (normal) Wi-Fi network:

```
cd /etc/wpa_supplicant
sudo mv wpa_supplicant.conf wpa_supplicant_backup.conf
```
NOTE: When these commands take effect after a reboot, the Pi will be unable to access the internet unless some other method has been enabled. If you are accessing the Pi wirelessly on wlan0 over SSH, it will not join whatever network you were using. I used macOS’s Internet Sharing feature (Settings > General > Sharing > Internet Sharing) to share my laptop’s WiFi connection over Ethernet. Because I had multiple devices, I plugged my laptop and all the Pis into a dumb switch. In addition to providing internet to all the Pis, this also allowed me to control the Pis over SSH (note that this is not enabled by default) and bring the wireless network up and down as needed for testing. As regards connecting to the ad-hoc network, Apple devices (iPad, Mac) seem to have no issues joining it as though it were a regular network. This may be useful for debugging purposes. I was unable to get my Windows/Ubuntu laptop to join (note that Raspberry Pi OS is derived from Debian, so perhaps Debian would work better than Ubuntu in this regard).

4. Reboot.

   `sudo reboot`

5. Verify that the network is configured properly. In particular, check that the mode is listed as “Ad-Hoc” and that the ESSID matches the one specified above (“PiFi”).

   `iwconfig wlan0`

6. Verify that the IP address was set successfully. The second line of the output should list the address: “inet 10.0.0.4”.

   `ifconfig wlan0`

7. Finally, if this is not the first device to be set up, ping another Pi’s static IP address (e.g. 10.0.0.1) and verify that it responds.

   `ping 10.0.0.1`
A.2 BATMAN

The following steps are sufficient to install and configure BATMAN. If these fail on your device, or you wish to further configure it, the official installation guide is available at https://openwrt.org/docs/guide-user/network/wifi/mesh/batman.

1. Install the BATMAN control tool, batctl.

   
   ```bash
   sudo apt install batctl
   ```

2. Add the following lines to `/etc/rc.local` to run on startup. These initialize the BATMAN network interface, `bat0`. The first line indicates which physical interface should be used, and the next two activate the interface.

   ```bash
   sudo batctl if add wlan0
   sudo ifconfig wlan0 up
   sudo ifconfig bat0 up # interface bat0 created by batctl
   ```

3. Create a file `bat0` in `/etc/network/interfaces.d`. Add the following lines to the file.

   ```bash
   auto bat0
   iface bat0 inet auto
      pre-up /usr/sbin/batctl if add wlan0
   ```

4. Open `/etc/modules` (kernel modules to load on startup) and add the following line.

   ```bash
   batman-adv
   ```

5. Reboot.

   ```bash
   sudo reboot
   ```

6. Verify that BATMAN is using the correct physical interface with the following command. You should see “wlan0: active”.

   ```bash
   ```
sudo batctl if

7. If this is not the first host to be configured, run the following command to check whether the MAC addresses of the other devices running BATMAN appear.

sudo batctl n

8. Finally, run the following command and verify that the bat0 interface appears (there may be a message saying “no wireless extensions”, which is OK).

iwconfig

A.3 Compiling and Running BRUNET

1. The libnftables library is installed by default on Linux. However, to compile BRUNET, the development version must be installed.

sudo apt install libnftables-dev

2. The libfyaml library can be found on GitHub at https://github.com/pantoniou/libfyaml. The following steps are adapted from the README on GitHub.

(a) Install prerequisites. The README lists additional prerequisites for various extra features, but these are not necessary for BRUNET.

    sudo apt install gcc autoconf automake libtool git make \ libltdl-dev pkg-config

(b) Navigate to the Releases page and download version 0.8 (libfyaml-0.8.tar.gz). A newer version probably will work, but this project has only been tested with 0.8.

(c) Navigate to the download folder, and extract its contents.

tar -xzf libfyaml-0.8.tar.gz
(d) Navigate to the new folder, `libfyaml-0.8`, and run the following commands.

```bash
./bootstrap.sh
./configure
make
sudo make install
```

(e) Finally, run the following command to add the newly installed shared library to the system's cache, so the runtime linker can find it. Without this command, BRUNET will compile but not run.

```bash
sudo ldconfig
```

3. Download the BRUNET source code from the GitHub repository: [https://github.com/clausworks/BRUNET](https://github.com/clausworks/BRUNET). Navigate to the Releases page and download the latest release.

4. Navigate to the source code folder and run the following command.

```bash
make
```

5. Verify that the executable launches. Edit `sample_setup.yaml` to include the appropriate IP addresses and ports.

```bash
sudo ./brunet sample_setup.yaml
```

A.4 Testing

1. Install `ucspi-tcp`, which includes `tcpserver(1)`, used in testing.

```bash
sudo apt install ucspi-tcp
```

2. Various testing scripts are included with the source code.
Appendix B

COMPLETE TESTING LOG

PROXY PROGRAM CONNECTIONS

General comment:
Manually verify operations succeed by reading output messages.

Log level: LOG_DEBUG
Config file: 1.yaml

[proxy1AB]
Given: comms up; A up; B down
Show: reconnect repeat starts on A
Repeat: swap A/B

[proxy2AB]
Given: comms up; A up; wait for reconn repeat; B up
Show: reconnect repeat starts on A, then connects to B; sync succeeds
Repeat: swap A/B

[proxy3AB]
Given: comms up; A up; B up quickly
Show: B connects to A before reconnect sequence starts
Repeat: swap A/B

CONNECTION ESTABLISHMENT/RECOVERY

General comment:
Test that user program connections can connect and disconnect properly.

Log level: LOG_DEBUG
User programs:
- clnt1 (interactive): nc ADDR PORT
- clnt2 (send file): nc -q 0 ADDR PORT < INFILE > OUTFILE
- serv1 (interactive, single client): nc -l PORT
- serv2 (echo, multi client): tcpserver ADDR PORT cat

ESTABLISHMENT
-------------
Config: 1.yaml

[estab1]
Given: comms up; A & B up; serv1 down; clnt1 up
Show: A sends LC_NEW, B recvs & registers LC; B sends LC_EOD, LC_CLOSED_WR;
      A recvs & closes conn to clnt1 immediately

[estab2]
Given: comms up; A & B up; serv1 up; clnt1 up; kill clnt1
Show: A sends LC_NEW, B recvs; A sends LC_EOD; B recvs, closes
      conn to serv1; B sends LC_EOD & LC_CLOSED_WR; A recvs LC_EOD, closes
      conn to clnt1; A recvs LC_CLOSED_WR, ignores

[estab3A]
Given: comms up; A & B up; serv1 up; clnt1 up; A sends 1B; B sends 1B;
      kill clnt1
Show: A sends LC_NEW, B recvs; A sends LC_DATA, B recvs & sends LC_ACK;
      B sends LC_DATA, A recvs & sends LC_ACK; A sends LC_EOD; B recvs, closes
      conn to serv1; B sends LC_EOD & LC_CLOSED_WR; A recvs LC_EOD & closes
      conn to clnt1; A ignores LC_CLOSED_WR

[estab3B]
Given: comms up; A & B up; serv1 up; clnt1 up; B sends 1B; A sends 1B;
      kill serv1; A sends 1B
Show: A sends LC_NEW, B recvs; B sends LC_DATA, A recvs & sends LC_ACK;
      A sends LC_DATA, B recvs & sends LC_ACK; B sends LC_EOD; A recvs, does
      not close conn to clnt1; after 1B sent, B sends LC_EOD & LC_CLOSED_WR,
      closes conn to serv1; A recvs, closes conn to clnt1

[estab4A]
Given: comms up; A & B up; serv2 up; clnt2 up, short.txt
Show: A sends LC_NEW, LC_DATA, LC_EOD; B sends LC_ACK, LC_EOD, LC_CLOSED_WR;
      diff output & input

[estab4B]
Given: same as [estab4A], but lorem.txt  
Show: same as [estab4A]; ensure only one LC_DATA ea dir

[estab4C]  
Given: same as [estab4A], but bible.txt  
Show: same as [estab4A]; ensure many LC_DATA ea dir

RECOVERY  
-------
Config: 1.yaml

[recov1A]  
Given: comms down; A & B up; serv1 down; clnt1 up; wait for reconn repeat; comms up  
Show: A & B connect after comms up; packets exchanged, same as in [estab1]

[recov1B]  
Given: A & B up; comms down; serv1 down; clnt1 up; wait for reconn repeat; comms up  
Show: A & B connect; sync occurs after reconnect; packets same as in [estab1]

[recov2A]  
Given: comms up; A & B up; comms down; serv1 up; clnt1 up; kill clnt1; comms up  
Show: A & B connect; packets same as in [estab2]

[recov2B]  
Given: comms down; A & B up; serv1 up; clnt1 up; kill clnt1; comms up  
Show: A & B connect after comms up; packets same as in [estab2]

[recov3A]  
Given: comms up; A & B up; serv2 up; clnt2 up, bible.txt; comms down during transfer; comms up  
Show: A & B connect; data transfer starts, stops, resumes, completes; close succeeds; diff output & input

[recov3B]  
Given: comms down; A & B up; serv2 up; clnt2 up, bible.txt; comms up after transfer  
Show: A & B connect; all data from clnt2 cached; clnt2 hits EOF but does not close; after comms up, transmission resumes and all data is echoed back to clnt2 successfully; diff output & input
Given: comms up; A & B up; serv2 up; clnt2 up, bible.txt; until complete, run toggle_wlan0, up 1s, down 3s
Show: A & B connect; A & B disconnect/reconnect several times during data transmission; data transmission complete successfully

MULTIPLE CLIENTS/SERVERS

MULTIPLE CLIENTS

Config: 1.yaml

[mclnt1ABC]
Given: comms up; A & B up; serv2 up; seq_clnt.sh, short.txt, 2 clnts
Show: First LC closes before second starts; diff input/output successful
Repeat: bible.txt, 5 clnts; 10 clnts, lorem_long.txt

[mclnt2AB]
Given: comms up; A & B up; serv2 up; simul_clnt.sh, bible.txt, 2 clnts
Show: Second begins transmitting before first ends; diff input/output
Repeat: 10 clnts

MULTIPLE SERVERS

Config: 2_p.yaml (two servers on same host; different ports)
Given: serv1/clnt1 port 1234, serv2/clnt2 port 2345; use nc; serv1/2 up;
  connect clnt1/serv1; send text clnt1->serv1, & reverse; kill clnt1;
  repeat w/ serv2/clnt2
Show: data delivered to correct program; serv/clnt program closes as appropriate
Repeat: same but clnt2/serv2 first; clnt1/serv1 first, but kill serv1 first;
  clnt2/serv2 first, but kill serv2 first
Note: keep proxy programs open; implicitly test sequential LCs

[mserv1ABCD]
Config: 2_p.yaml (two servers on same host; different ports)
Given: serv1/clnt1 port 1234, serv2/clnt2 port 2345; use nc; serv1/2 up;
  connect clnt1/serv1; send text clnt1->serv1, & reverse; kill clnt1;
  repeat w/ serv2/clnt2
Show: data delivered to correct program; serv/clnt program closes as appropriate
Repeat: same but clnt2/serv2 first; clnt1/serv1 first, but kill serv1 first;
  clnt2/serv2 first, but kill serv2 first
Note: keep proxy programs open; implicitly test sequential LCs
connect clnt1/serv1; connect clnt2/serv2; send text clnt1->serv1, & reverse; send text clnt2->serv2, & reverse; kill serv1; kill serv2;
Show: data delivered to correct program; clnts close appropriately

[mserv1F]
Config: 2_p.yaml (two servers on same host; different ports)
Given: 5 simultaneous clnts per serv (serv1: 1234, serv2: 2345); use simul_clnt.sh (test[1-5]_5000000.in) for clnts, tcpserver for servs
Show: each clnt recvs correct data, diffs match

[mserv2]
Config: 2_cs.yaml (A as clnt & serv; B as serv & clnt)
Given: 5 simultaneous clnts per serv; user simul_clnt.sh (test[1-5]_5000000.in) for clnts, tcpserver for servs
Show: each clnt recvs correct data, diffs match

THREE DEVICES
------------
Given: For each of the following config files, use adv_simul_clnt.sh w/ 5MB (n_bytes = 5000000) to generate 5 clients per server.
Show: diffs all match

[three1]
Config: 2_c.yaml (B as serv; A and C as clnts)

[three2]
Config: 2_cp.yaml (B as serv, 2 ports; A and C as clnts)

[three3]
Config: 2_s.yaml (B and C as servs, same port; A as clnt)

[three4]
Config: 2_sp.yaml (B and C as servs, diff ports; A as clnt)

[three5]
Config: 3_circ.yaml (A-B clnt-serv; B-C as clnt-serv; C-A as clnt-serv)

======
LIMITS
======
(6 tests)
Config: 1.yaml
Given: Use adv_simul_clnt.sh (5MB) to generate 16 clients. Use LOG_INFO.
Show: Program functions with maximum number of user clients.
Repeat: with 500KB inputs & toggle_wlan0.sh
Notes: When running this using toggle_wlan0, I ran into two bugs that I could not reproduce easily. First, tcpserver reported "connection reset" for one of the clients midway through. Second (testing using LOG_DEBUG & 1MB files), the server proxy program aborted at assert in receive_peer_sync:
   assert(peer_total_received >= peer->obuf.total_acked);

Config: 16_cp.yaml (16 servers on B, different ports)
Given: Use simul_serv.sh (500KB) to generate 16 servers. Generate one client per server with adv_simul_server_p.sh.
Show: Program functions normally; diffs match
Repeat: with toggle_wlan0.sh

Config: complex.yaml
Commands (in order):
   A$ bash toggle_wlan0.sh 5 2
   C$ bash toggle_wlan0.sh 1 1
   A$ tcpserver 10.0.0.1 13000 cat
   B$ tcpserver 10.0.0.2 10000 cat
   C$ tcpserver 10.0.0.3 10000 cat
   C$ nc -l 12000
   B$ nc 10.0.0.3 12000
   A$ bash sporadic_simul_clnt.sh 10.0.0.2 10000 100000 5 -3
   A$ bash sporadic_simul_clnt.sh 10.0.0.3 10000 100000 5 -3
   A$ bash adv_simul_clnt.sh 10.0.0.3 10000 10000000 1
   B$ bash adv_simul_clnt.sh 10.0.0.1 13000 10000000 1
   (after completion, send 1 line each direction on nc servers, close clnt)

Config: ssh.yaml
Given:
Launch proxy programs in tmux. This prevents the connections from being shut down when the terminal disconnects. Open SSH connection from A to B and another from C to A. Confirm the connections are live and that only the one from A is managed by the proxy programs. Launch top on both terminals. Place B in "Faraday
cage" and move approximately 40 ft. away. Monitor A to ensure it begins
reconnect sequence. On both SSH terminals, type `echo hello` and press Enter.
Wait 30 minutes. Bring B back and reconnect Ethernet cable. Relaunch tmux
session and verify that the proxy program is still running.

Show:
Proxies reconnect successfully. SSH session ends on C, but remains alive on A.
Hello echo message appears on A, and further commands can be issued. Exiting
session closes LC.

==========
PERFORMANCE
==========
(3 tests)

[ perf1 ]
Setup: 1.yaml, adv_simul_clnt.sh w/ 10MB inputs, 5 clients. Use time(1) and
echo server.
Measure: real/user/sys time. Repeat 3 times with BRUNET, 3 times without.

[ perf2 ]
Setup: Same as [ perf1 ], but with sporadic_simul_clnt.sh. Use 10KB, 5 clnts,
1ms delay
Measure: Same as [ perf1 ]

[ perf3 ]
Setup: Use nc to send one byte to the echo server (short.txt).
Measure: Same as [ perf1 ]