# Weaving a Faster Tor: A Multi-Threaded Relay Architecture for Improved Throughput

by

Steven Engler

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

The Tor anonymity network has millions of daily users and thousands of volunteer-run relays, but growing it further has several research and deployment challenges. One such challenge is supporting the increase in bandwidth required by additional users joining the network. While simply adding more Tor relays to the network would increase the total available bandwidth, it requires that Tor's directory documents grow to accommodate these new relays, which in turn increases the burden on Tor clients who must download these large documents. These large directory documents are problematic for people using Tor on mobile devices or who have limited Internet access. Previous approaches to scale the Tor network require significant network-level architectural changes.

In order to increase the total available network bandwidth without needing to grow Tor's directory documents or change the network architecture, this work replaces Tor's existing relay architecture with a new multi-threaded architecture. This new architecture is designed to improve the throughput of individual relays that have available network capacity and access to a multi-core processor, and parallelizes Tor's network routing and circuit handling to offload these computationally expensive operations on additional threads. As Tor's current relay architecture is unsuitable for this type of multi-threading, we examine the obstacles in adapting relays to our new multi-threaded architecture. We built an implementation of a subset of this new design on top of the standard Tor code base to demonstrate the potential throughput improvements of this architecture. Under experimental conditions, we show that the multi-threaded implementation quadruples the relay's throughput compared to the standard Tor relay implementation when using four cores of an Intel Xeon server, and triples the relay's throughput when using a $50 Raspberry Pi single-board computer.

## Acknowledgements

I would like to thank my supervisor, Ian Goldberg, for his guidance and support throughout my program. His insight into privacy-enhancing technologies, such as Tor, was invaluable and I have learned much from it. I would also like to thank my committee members, Urs Hengartner and Ali Mashtizadeh, for their comments and feedback on this thesis. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

## Dedication

*To my parents, for their continued support, encouragement, and advice.*

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Tor, an overlay network designed for privacy, anonymity, and censorship resistance consists of over 6000 volunteer-run relays as of July 2020 [Tor20c] with a recent estimate of 8 million active daily users [MWBJ+18]. While the Tor network user base is large, there is potential for it to grow significantly. The demand for web privacy tools has been demonstrated by not only the popularity of Tor, but also the popularity of the consumer VPN industry and the inclusion of private browsing modes and anti-tracking features in major browsers [Moz20]. Public awareness about the extent of Internet surveillance has also grown with high-profile reporting on oppressive government surveillance programs and malicious advertising practices. Usability and performance shortcomings have limited the growth of Tor in the past, but significant improvements have been made in recent years such as better user documentation and support [Tor19a], usability enhancements in the Tor Browser and its release on Android [Tor19c], better relay congestion control and scheduling [Tor17], and improved censorship resistance techniques [Tor15]. There is also hope to bring native Tor support to additional applications in the future such as the private browsing mode in Firefox [Tor19b], which could introduce Tor to orders of magnitude more users.

Growing the Tor network and its number of users is important not only for helping more people protect their online presence, but also to help existing users by increasing their anonymity set and improving their privacy. If Tor's usage is to increase, the network must have the capacity to support the additional users. Scaling the Tor network to support more users presents several research and deployment challenges. Simply increasing the number of relays in the network is problematic for a number of reasons, one of which is the growing size of Tor's directory documents. While several approaches have been proposed to help the network scale, they often require network-level architectural changes. As a distributed, community-run network, these architectural changes are difficult to design, complicated to deploy, and require careful consideration to avoid partitioning the users in a way that could harm their privacy. While these network-

wide architectural changes will likely be needed sometime in the future, Tor would benefit from improved network performance and capacity in the short term.

We propose improving the network capacity by better utilizing existing network relays. By modifying only the internal relay architecture rather than the network architecture, the network can be improved through independent upgrades to existing relays without harming or partitioning users. While some relays are limited by network capacity, either from slow Internet or configuration limits set by the relay operator, other relays are not. These might be relays operated by academic or other large institutions with high-bandwidth Internet lines, or relays running on low-performance servers such as cloud-hosted virtual private servers or small single-board computers like the Raspberry Pi. If these relays are not network-limited, they will often be CPU-constrained. In both of these examples the servers typically have access to multiple CPU cores, most of which go unused due to Tor's mostly single-threaded relay architecture. The multi-threaded relay architecture described in this thesis better uses the CPU resources of congested relays with available network capacity, with the goal of providing more capacity to the network to support more users. As Tor's current single-threaded relay architecture does not trivially adapt to multi-threading, this thesis describes some of the challenges involved in shifting Tor relays into a multi-threaded model.

## 1.1 Contributions

This work intends to prove the following thesis statement:

*We can construct a Tor relay architecture with parallelized routing such that the relay's throughput increases due to better utilization of processing cores. This improved throughput allows relays to reduce client download times and support more users, while avoiding the network's directory scaling constraints and without changing Tor's network protocols.*

To substantiate this claim, the remainder of this thesis provides the following:

- A description of Tor's current relay architecture and the design attributes that make it unsuitable for parallelized routing.
- A multi-threaded relay architecture that preserves compatibility with the existing Tor network and parallelizes the end-to-end flow of network data through the relay.
- An experiment design to discover the maximum sustained throughput of a given relay implementation while experiencing heavy traffic.

- A multi-threaded relay implementation realizing a subset of our multi-threaded architecture, where the processing is better distributed across CPU cores and the maximum sustained throughput is much greater than Tor's current relay implementation.

This thesis is organized as follows: Chapter 2 describes the original ideas behind onion routing and how Tor builds upon them to create the Tor network. Chapter 3 lists existing improvements and research to support more users and improve the network's performance, some of which have been previously integrated into the live network. In preparation for our multi-threaded relay architecture, Chapter 4 explains components of Tor's current relay architecture that are relevant to the relay's routing. Chapter 5 describes some obstacles to designing and implementing a multi-threaded relay architecture, and presents our multi-threaded relay architecture that works around these obstacles. To demonstrate potential performance improvements of the architecture, Chapter 6 implements a subset of the multi-threaded architecture on top of Tor's current relay implementation and compares their throughput using an experimental Tor network. Chapter 7 outlines areas for future research and provides concluding remarks.

# Chapter 2

# Tor and Onion Routing

As Internet routing requires every packet to identify its sender and receiver, any router along the path of this packet can easily identify which Internet addresses are communicating. While these addresses are important from a technical standpoint, they have social implications for people who wish to keep their Internet communication protected from surveillance, censors, advertisers, or other malicious parties. Much of the earliest work on anonymizing network communication came from anonymous remailers, which accept users' emails and forward them to the destination in a way that prevents the destinations from learning the emails' senders. Email as an application-level service requires not only network-level defences, but also application-level ones (for example anonymizing email headers). Many of the ideas underlying these remailers have inspired anonymity networks today.

Earlier remailers offered weak privacy properties; they were vulnerable to traffic analysis attacks by observers monitoring the remailer's network and required the user to place all of their trust in the remailer provider [GWB97]. Later remailer designs improved upon this foundation using networks of mix nodes [Cha81], which mix and delay fixed-size messages (including fake messages as cover traffic) to hinder passive traffic analysis attacks. Clients send messages along multi-hop paths of mix nodes and protect them using a layered-encryption design that ensures each mix learns only the information it needs to forward the message to the next mix (or the destination) and send a validation response back towards the sender. Many of the design properties of remailers and mix networks do not apply well to the problem of low-latency, high-throughput, and bi-directional anonymous communication. Web browsing for example requires all three of these properties for communication between a client and web server. Mix networks are inherently high-latency if the frequency of inbound messages is not sufficiently large, and the fixed-size nested encryption design (as well as the required cover traffic) has a large bandwidth overhead.

## 2.1 Onion Routing

Providing efficient network-level privacy for bi-directional communication was a problem tackled by several network designs, one of which was onion routing [GRS96, RSG96, RSG98, GRS99]. Onion routing was designed to prevent linking of the sender and receiver at both the network and application layers. It uses the layered-encryption message design mentioned previously to create multi-hop network paths called *anonymous connections* (or *virtual circuits* as we will refer to them) in order to provide an anonymous bi-directional channel persisting for the life of the connection. This design meets the functional requirements of web browsing, where a small HTTP request by a client is followed by content responses from the web server.

In the original onion routing architecture a user application would connect to an application-specific proxy (for example an HTTP proxy), which would modify and anonymize application-specific data. It would then forward the destination address as well as the modified application data to the *onion proxy*. The onion proxy would build a multi-hop circuit through a network of *onion routers*, encrypting and sending the application data along this circuit. All data would be sent between nodes as fixed-size messages called *cells* with a header (a circuit identifier, message type, and payload length) and a payload (the type of data depending on the message type). Cells would use the type Create for circuit construction, Data for sending application data, Padding for adding link cover traffic, and Destroy for tearing down circuits when they are no longer needed.

After receiving a request from the application proxy, the circuit construction would begin with the onion proxy selecting a list of at most five onion routers and looking up their public keys. This onion routing architecture assumes that the network is static and that each onion proxy knows each onion router's public key. For each selected onion router $X$ and its public key $PK_X$, the proxy would choose key seed material $K_{0X}$, a cell expiration time $T_X$, and two cryptographic functions: $F_{fX}$ for data travelling on the circuit in the forward direction and $F_{bX}$ for the backward direction. The proxy would also generate a symmetric key $K_{1X}$ from the SHA hash of $K_{0X}$. Using this information for each onion router, the proxy would generate a multi-layered *onion* data structure. To help describe the structure of these onions, we use the notation $\{\alpha|\beta\}_{PK,K}$ for the first 1024 bits of the concatenation of $\alpha$ and $\beta$ encrypted under the 1024 bit public key $PK$, and the remaining bits encrypted under the symmetric key $K$. For example, the three-hop circuit $onion\_proxy \rightarrow A \rightarrow B \rightarrow C \rightarrow destination$ would result in the onion:[1]

$$\begin{aligned}&\{0|version|F_{bA}|F_{fA}|B|T_A|K_{0A}| \\ &\quad \{0|version|F_{bB}|F_{fB}|C|T_B|K_{0B}| \\ &\quad\quad \{0|version|F_{bC}|F_{fC}|\emptyset|T_C|K_{0C}|padding\}_{PK_C,K_{1C}}\}_{PK_B,K_{1B}}\}_{PK_A,K_{1A}}\end{aligned}$$

[1] The initial 0 is to keep the plaintext smaller numerically than the modulus for use with RSA.

As this onion is sent along the circuit, each onion router uses its private key to decrypt the onion and learn the information in its outermost layer. From the message header it learns a circuit identifier and from this decrypted onion layer it learns the destination of the next hop, the cell's expiration time, the key seed material, and both cryptographic functions. The onion router then derives three keys $K_{1X}$, $K_{2X}$, and $K_{3X}$ from the key seed material $K_{0X}$ using repeated SHA operations. Key $K_{1X}$ is used to decrypt the remainder of the onion, $K_{2X}$ is used with $F_{bX}$ to transform application data travelling towards the onion proxy, and $K_{3X}$ is used with $F_{fX}$ to transform application data travelling toward the destination. The onion router records the circuit identifier, the cryptographic function/key pairs $(F_{bX}, K_{2X})$ and $(F_{fX}, K_{3X})$, and the address of the next hop for future use, and temporarily stores a hash of the entire onion along with its expiration time $T_X$ to prevent replay attacks. The onion router can then add padding to the end of the remaining onion such that the size is the same as the original onion and send it to the next hop. This property of constant-sized onions is useful for limiting passive traffic analysis and for hiding the length of the circuit from the router. The end result of this circuit construction is that each onion router along the circuit has the cryptographic keys required for processing application data in both directions and a record of which onion router it should forward the data to.

After the onion proxy has sent the onion to build the circuit, it should send a Data cell with the payload containing a *standard structure*. This standard structure contains the onion routing protocol version, an application-level protocol identifier, a retry count, and the address of the final destination server (for example a web server). This cell payload, along with all future Data cell payloads are encrypted iteratively by the onion proxy for each node in the circuit starting with the final node (the innermost onion router in the original onion). Specifically, each iteration uses the inverse cryptographic operation that each individual onion router uses so that as the cell is sent along the circuit and decrypted once at each hop, the plaintext will be revealed at the final hop. The final hop, now knowing the final destination for the server, makes a connection through its outbound proxy and once successful, returns a positive status code back to the onion proxy. On receiving the positive status code, the onion proxy begins forwarding application data to the circuit following the iterative encryption described above. When either side of the connection closes the socket, a Destroy cell will be sent to the opposite end to close the circuit.

Onion routing as originally presented also offered a few additional components such as *reverse onions* for when the destination wants to send a message back to the proxy sometime in the distant future (for example when a server receives and later replies to an anonymous email), but since these features are not important for real-time onion routing, we do not discuss them here. While this onion routing architecture provided a framework for anonymous bi-directional communication, it had several problems such as outdated cryptography, no forward secrecy, unauthenticated link encryption, no support for dynamic networks or key distribution, the requirement of a new circuit for each application connection, and more.

## 2.2 Tor

Tor, originally described as the second-generation onion routing design, is a transport-layer overlay network that aims to prevent linking of the sender and receiver at the network layer [DMS04]. While Tor is based on the onion routing design, it solves many of the deployment challenges with onion routing (sharing node certificates, adding/removing nodes from the network, etc.), improves many of the privacy properties, and includes additional features such as anonymity for servers who wish to anonymously provide content to users. Specifically it offers anonymity against an adversary who can observe and modify some limited fraction of the network's traffic.

When a user application wishes to route a TCP connection through Tor, it will forward that request to a SOCKS proxy exposed by a Tor client running on the user's device. This Tor client builds multi-hop *circuits* through *relays* on the Tor network as visualized in Figure 2.1 and will assign the proxy request to one of these circuits. While the network supports circuits of up to eight hops [EDG09, Din12], the standard Tor implementation always uses three hops for circuits that provide anonymity. The client can request that any hop along this circuit make the TCP connection leaving the Tor network to the destination server that was initially requested by the

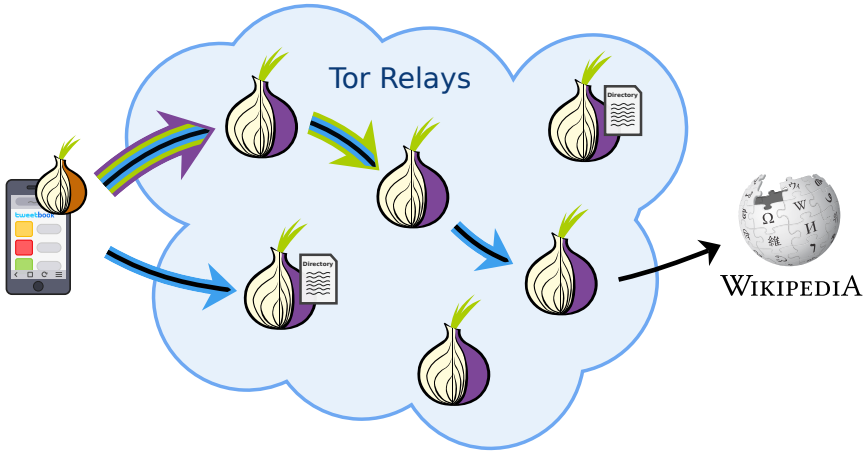



Figure 2.1: A diagram of the Tor network and how clients use it to anonymize their Internet traffic. A device running the Tor client proxy creates a three-hop anonymizing circuit through a network of Tor relays and tunnels an application stream through this circuit to a destination web server. The client also connects to a relay acting as a directory cache and downloads directory documents. The Wikipedia logo by the Wikimedia Foundation is licensed under CC BY-SA 3.0, and Tor logo by The Tor Project, Inc. is licensed under CC BY 3.0 US.

application (Tor's leaky-pipe circuit topology), although the standard Tor implementation always requests that the final hop in the circuit make this connection. While the circuit is the multi-hop path from the Tor client to the last hop (the *exit relay*), the end-to-end path from the user's application (the *entry connection*) through a circuit and to the final destination server (the *exit connection*) is known as a *stream*. Tor can multiplex multiple streams through a single circuit in order to improve performance, but only does so when it believes that anonymity will not be significantly impaired.

Clients construct circuits using a telescoping design that allows the circuit's origin to incrementally build a secure, confidential communication channel between it and each hop in the circuit. This circuit construction allows the client to communicate with a specific relay without any other relay learning the contents of that communication. Protecting this communication is important not only for protecting the contents of the application stream, but also to prevent any intermediate relay from learning the circuit's full path during circuit creation (each relay only learns the identity of its two adjacent nodes). As circuit building is an expensive operation, the client builds a pool of circuits ahead of time and assigns new streams to them as needed. Once a circuit has been used for a stream it is marked as "dirty", and ten minutes after will be considered "unusable" for new streams.

Since circuits typically use three hops, each hop is referred to by a name given its position in the circuit: the entry relay, middle relay, and exit relay. All relays that meet some basic requirements are eligible to be chosen by clients as *middle relays* for circuits. These are the simplest for relay operators to run as they only ever connect to other relays and never see any user application-level data. Relay operators might also opt-in to allowing their relay to be used as an *exit relay* which can be used to make connections leaving the Tor network. Relays that meet some uptime, bandwidth, and other requirements also become eligible to be selected by clients as *guard relays*. These guard relays acts as *entry relays* and are important as they are the relays that clients connect to directly and therefore learn the pseudo-identity (the IP address) of each Tor user who connects to them. While the Tor network generally[2] does not prevent a network observer from learning that a user is using Tor, these guard relays are in a privileged position as traffic confirmation and timing attacks against Tor users often depend on the adversary having the ability to modify, delay, or observe traffic at the user's guard relay [MD05, LLY+09]. In an attempt to limit the effectiveness of these attacks, Tor clients choose a small persistent set of guard relays for all circuits [EBA+12, DHKM14]. Clients use this guard set for several months as their entry relays.

Tor processes communicate using a custom application-level protocol, which consists of

[2]Tor supports optional unlisted bridge relays (which replace guards) and pluggable transports to prevent censors from identifying Tor users

sending *cells* across TLS-encrypted *relay connections*. Each pair of Tor processes will generally have at most a single relay connection between them, and any circuits on the network that pass along this edge will be multiplexed over this connection. There are different cell types for actions such as building or destroying circuits, relaying data, padding connections, negotiating protocol versions, authenticating relays, and more. The type of cell is specified by its *cell command*, and while most of these cells are designed only to be passed between two directly connected Tor processes (whether relays or clients), Relay cells and its various sub-types (such as Relay{Extend} and Relay{Data}) are designed to pass a message along a circuit and are how a circuit's origin communicates privately with each hop along the circuit's path. These cells are constructed using a layered-encryption design that allow only a specific hop to read the cell's payload. Tor uses entry-to-exit flow control for both circuits and streams using Relay{SendMe} cells.

Clients are responsible for choosing the path through the Tor network of each circuit it creates. In order to choose this path, clients must have some knowledge about the relays in the network. Every relay updates and signs its own *server descriptor* document containing information about the relay and uploads it every 18 hours (unless it changes significantly in the meantime) to each of the network's nine semi-trusted *directory authorities*. The authority saves this document and once per hour votes on the current status of the network by combining the status of each relay into a single *status vote* document. This document is signed by the authority and published to the others. Each authority receives the other votes and combines them into a single *consensus document*. Following the consensus algorithm, each authority should have arrived at the same consensus document and each provides their signature for the consensus. In addition, the authorities produce multiple variants of the consensus called *consensus flavours*. All of these *directory documents* are made public by the authorities, but relays and clients are only interested in the final consensus document and the server descriptors. Some relays act as *directory caches* and provide copies of the latest consensus and server descriptors (in its various flavours) to other relays and clients. The other relays and all clients download a flavour of this consensus once per hour (or if they have a recent consensus, they download a *consensus diff*) and any server descriptors that are out of date. While the directory authorities and directory caches (collectively called the *directory servers*) serve the directory documents over HTTP, they also allow clients to tunnel these HTTP requests through circuits (as shown with the one-hop circuit in Figure 2.1). Clients download directory documents from their *directory guard* relays.

### 2.2.1 Tor's Onion Routing

All communication in the Tor network happens over connections that are link authenticated and encrypted using TLS. During authentication between a client and relay only the relay must au-

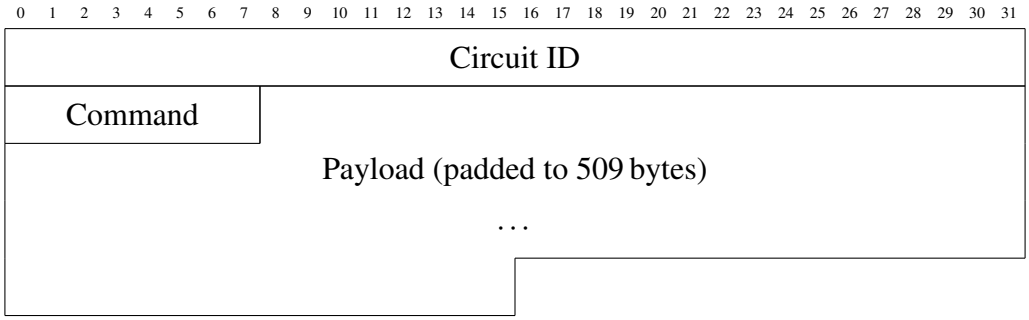



Figure 2.2: Structure of a fixed-length cell in Tor.

thenticate itself, but when establishing a connection between two relays they must mutually authenticate themselves. Older Tor link handshake protocols used TLS client certificates and renegotiation for this authentication, but for security and censorship-avoidance reasons the modern handshake protocol uses an "in-protocol" handshake where the responder authenticates itself during the TLS handshake and the initiator authenticates itself using the Tor protocol.

Much like the original onion routing design described in Section 2.1, messages are sent along these relay connections as cells. Cells are formatted as either 514-byte[3] fixed-length cells or variable-length cells. Each cell contains a circuit identifier, command, and payload as shown in Figure 2.2. Tor currently supports 17 core cell commands (and many more relay sub-commands), but several only exist for backwards compatibility. The most relevant cell commands for onion routing are the fixed-length Create2, Created2, Destroy, and Relay cells. Tor also supports a limited form of cover traffic using padding cells: Padding and VPadding[4] for link padding, and Relay{Drop} for circuit padding [PK20].

While the idea of persistent multi-hop circuits is similar to the original onion routing design, the circuit construction process differs in some important ways. Rather than its single-pass circuit creation technique using onions, Tor uses telescoping circuit creation. The client iteratively builds the circuit and performs a handshake with each hop. Each handshake requires a round trip to establish a shared secret and generate the session keys, which are used to tunnel circuit creation requests in future iterations. This telescoping approach allows for forward secrecy at the expense of much longer circuit creation times compared to the single-pass approach.

Tor uses Relay cells for end-to-end communication. The client can send Relay cells to any relay along the circuit and the payload of these cells are encrypted once for each hop using the session keys. Within this encrypted payload is data for the Relay cell's various subtypes,

[3] The first Version cell and any cells before that will be 512 bytes for backwards compatibility with the version 3 (and earlier) link protocol.

[4] Although it is in the specification document, Tor does not currently use its variable-length VPadding cells.

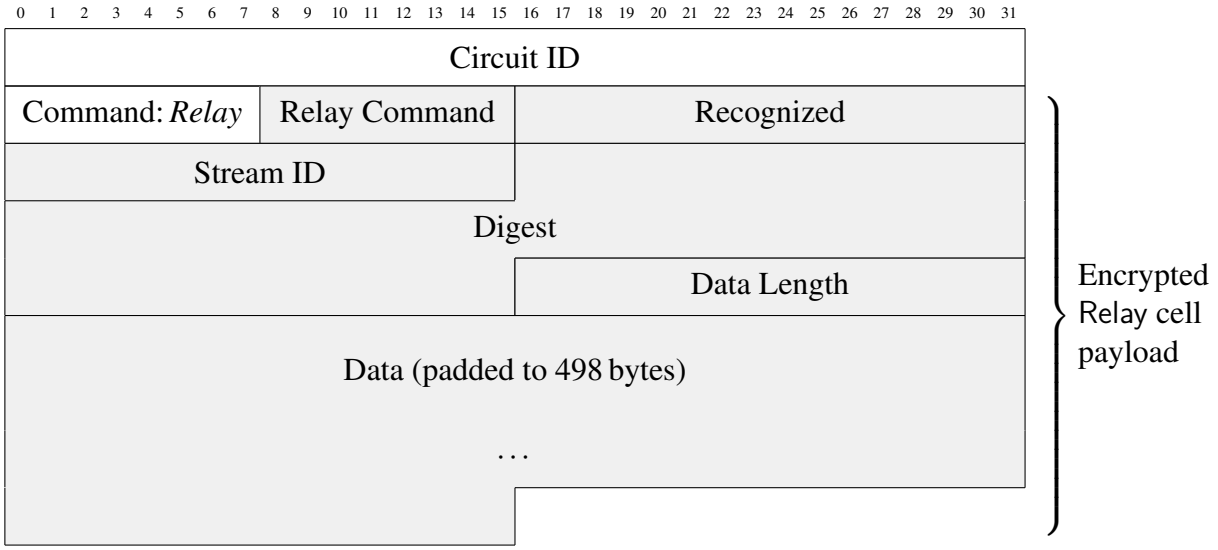



Figure 2.3: Structure of a Relay cell in Tor.

known as *relay commands* as shown in Figure 2.3. This layered-encryption design makes Relay cells useful not only for sending end-to-end application data using Relay{Data} cells, but also for circuit construction using Relay{Extend2} cells. To ensure the end-to-end integrity of Relay cells on a circuit, clients and relays maintain matching running digests of recognized Relay cell payloads. The client presents its view of the running digest by adding the first four bytes to the Relay cell, and relays compare its view of the running digest with the truncated value embedded in the cell. If a client wishes to send a Relay cell to the third hop in a circuit, it will build a Relay cell with the payload that it wants that hop to receive and iteratively encrypts that payload with the forward session keys of each relay in the path beginning with the farthest. The cell is sent to the first relay which decrypts the cell payload, checks if the "Recognized" field[5] is 0 and the running digest is correct, and if not forwards the decrypted cell payload in a new cell to the next hop. This process repeats until a relay (typically the final hop) decrypts the payload, finds the "Recognized" field is 0 and the running digest matches, and processes the Relay cell according to its relay command and data.

When building a circuit the client typically chooses three relays: one from its guard set as the entry relay, one non-exit relay as the middle relay, and one exit relay whose exit policy supports some given TCP port. The client performs an *ntor* handshake [GSU13, Mat13] with

[5]The "Recognized" field is set to 0 by the client and is simply an optimization to reduce the number of intermediate relays that have to calculate the running hash.

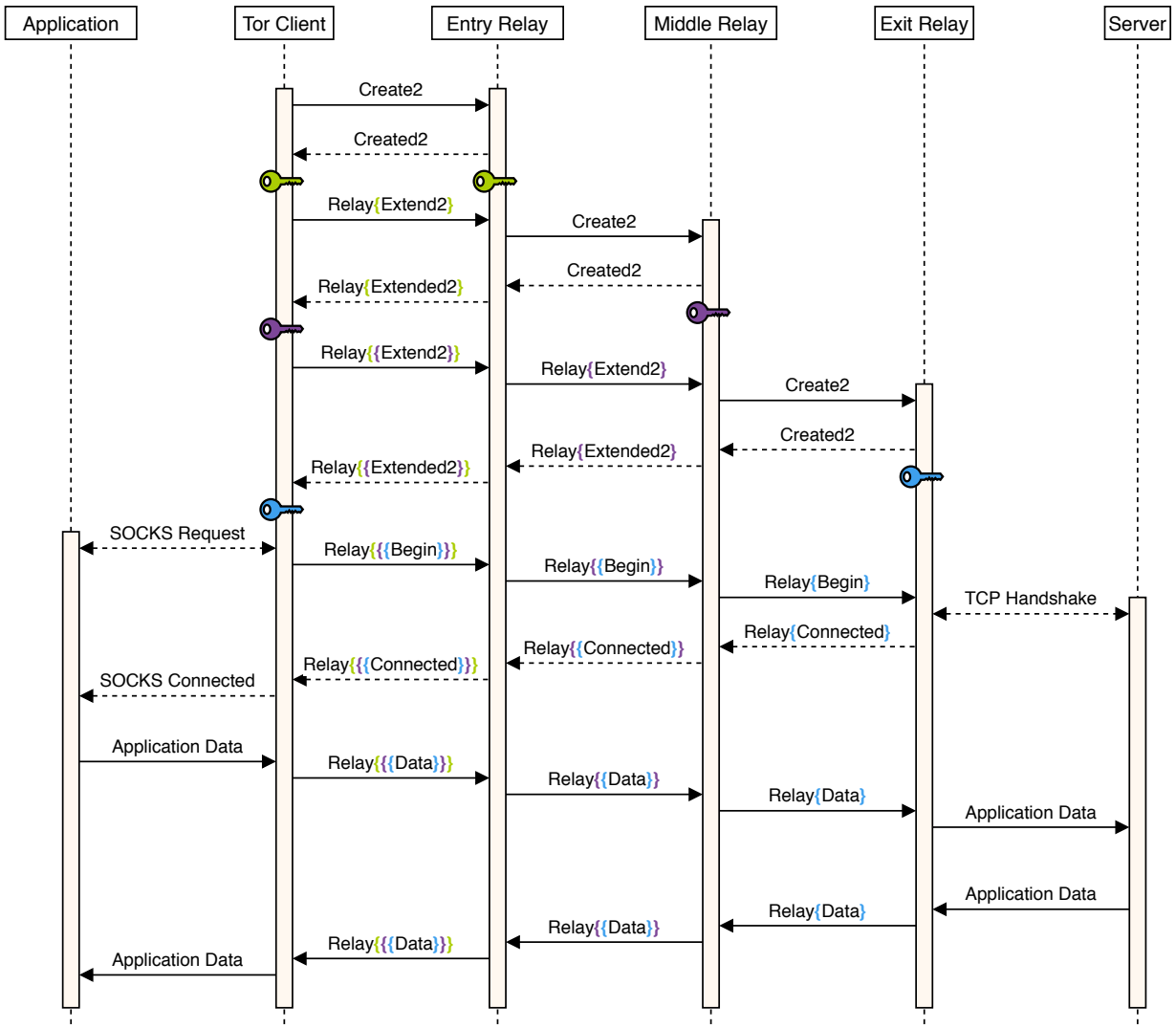



Figure 2.4: A visualization of circuit creation, stream initialization, and application data relaying. Each shared key represents a shared secret from which the relays derive the circuit's forward and backward session keys. Each relay is connected by a TLS connection, or makes a new connection before extending a circuit if they were not previously connected.

each relay in the circuit in order for the client and corresponding relay to derive shared session keys for encrypting Relay cells. For the first handshake, the client first sends a Create2 cell to the entry relay with a circuit handshake challenge and the relay replies with a Created2 cell with the handshake response. This creates a single-hop circuit that can then be extended to the remaining relays.[6] For each remaining relay, the client sends a Relay{Extend2} cell[7] through the in-progress circuit to its last relay. Eventually the entire circuit is built and becomes ready to handle application streams.

On receiving a new SOCKS proxy connection, the Tor client attaches this connection as an application stream to one of its pre-built circuits, or creates a new circuit if no existing circuits can handle the request. The client packages the SOCKS request's destination address as a Relay{Begin} cell and sends it to the exit relay. The exit relay then makes the TCP connection to this destination and once successful, sends a Relay{Connected} cell back to the client. The client informs the application that the SOCKS connection was successful and the application begins to send its application data. The client packages at most 498 bytes of this data at a time into Relay{Data} cells and sends them down the circuit. As the exit relay receives this stream data, it forwards it to the destination server. Simultaneously, as the exit relay receives application data from the destination server, it also packages it into Relay{Data} cells and sends them backwards along the circuit. When the client receives the stream data, it forwards it to the application behind the SOCKS proxy. Tor supports a non-standard SOCKS feature called optimistic data to allow clients to send application data before the exit makes the TCP connection to the destination in order to improve the time to first byte for HTTP connections, but this requires application-specific support and is usually not used today. The entire circuit construction and stream creation process is shown in Figure 2.4.

### 2.2.2 Experimentation Tools

As Tor is an anonymity network used by people whose personal safety is dependent on this anonymity, performing experiments and collecting data involving the live Tor network can be difficult to do in a privacy-preserving manner. Instead, various experiment environments and simulation tools have been developed for Tor research and development [SGD15]. Running a custom Tor network allows for fine control over the network topology and makes it possible to test network protocol changes. Chutney [Tor20a] is useful for initializing and bootstrapping

[6] Application streams can technically be attached to one-hop circuits, but relays explicitly disallow this in order to limit abuse of the network and to discourage unsafe usage.

[7] In practice Tor uses RelayEarly cells rather than Relay cells for circuit construction in order to limit the length of circuits, but for our purposes they are identical.

a local Tor network on a single server without any emulation. This can be useful for understanding performance characteristics of a single relay, but typically not the Tor network as a whole. It is limited to small network sizes and does not perform any network emulation, so it is unable to model the live Tor network. Chutney can be combined with other tools such as NetMirage [Ung20], a tool useful for emulating large virtual networks with configurable Internet routing latencies and congestion, but at the expense of added complexity and a one-time bootstrapping delay that scales quadratically with the size of the network. For experiments with much larger network topologies, the Shadow discrete-event network simulator [JH12] enables large-scale, efficient Tor network experiments. This comes with the drawback of long experiment times as large simulated networks run on a single computer, typically slower than real time. In addition, Shadow's system and network function emulation using symbol interposition can affect the accuracy of certain measurements.

# Chapter 3

# Tor Network Scaling and Performance

Growing the Tor network to support more users without compromising privacy and security has several challenges. Generally the network must not become more difficult to use, it should not introduce new attacks on the network or its users, and any network-level changes must be accompanied by a privacy-conscious migration plan as it is not feasible to upgrade the thousands of volunteer-run relays in unison. Scaling the Tor network has been a concern since its introduction and much work has been done to propose solutions to its scaling problems.

## 3.1 Directory Documents and Client Bandwidth

In order to prevent route capture [WMB10] and route fingerprinting attacks [DC06], each client downloads a consensus document that gives the client a global view of the network, as well as a descriptor document for each relay in the consensus. The size of Tor's consensus document and the bandwidth associated with delivering these documents to each client is one of the design limitations restricting the size of the Tor network. The total size of Tor's consensus-related documents that are downloaded by clients scales linearly with the number of relays in the network, and as clients must download these documents in order to bootstrap, the size can become problematic for clients with limited bandwidth such as mobile devices. In addition, assuming that the proportion of users to relays maintains constant as the network grows, the total bandwidth used by the network for delivering consensus-related documents scales quadratically in the size of the network. While this is manageable around its current network size, it is not scalable if the network grows in orders of magnitude.

### 3.1.1 Minimizing Consensus and Server Descriptor Documents

Clients in earlier versions of Tor downloaded directory documents like they do today, but over time the documents grew with information that was not directly relevant to these clients. It became clear that in order to conserve bandwidth, smaller documents were needed for both clients and relays. Tor's directory documents have undergone several changes, the most important for reducing bandwidth being the removal of relay descriptors from the directory document (relay descriptors are now downloaded individually as needed), the creation of the consensus document (a single document rather than requiring the download of several directory documents), microdescriptors (a smaller version of relay descriptors), and extra-info documents (containing information previously in directory documents that are not generally downloaded by clients or relays).

Modern clients now download only microdescriptors and a microdescriptor-flavoured consensus, which is a variation of the consensus document that references microdescriptors rather than the full relay descriptors. These microdescriptors contain only the server descriptor information that clients need to operate. They are downloaded by clients as compressed text documents, and when possible downloaded as "diffs" to further reduce bandwidth. While the amount of data required to be downloaded by each client has been reduced, it continues to scale linearly with the number of relays in the network.

### 3.1.2 Network Paths, Membership, and Views

Low-latency anonymity network designs often fall into common categories, for example source-routed vs. hop-by-hop path construction, centralized vs. decentralized network membership, global vs. partial network views, and client-server vs. peer-to-peer topologies. These categories often have trade-offs between scalability and security/privacy, and Tor tends to lean towards the latter in order to limit attacks that are possible in some of the more scalable designs. For example Tor's source-routed path construction (where clients have exclusive control over the paths of their circuits) prevents route capture attacks that are more common in networks with path construction based on random walks such as the peer-to-peer Crowds [RR98], MorphMix [RP02, TB06], and ShadowWalker [MB09, SDH+10] designs. Malicious nodes that can influence the path can perform route capture attacks by selecting only other colluding nodes for the remaining path, effectively limiting the path length. If the malicious node in this case can also learn that it is the first hop either by the predecessor attack [WALS04] or some other means, it can trivially deanonymize users. While Tor protects against this route capture attack, its current circuit construction process does not scale to the network sizes supported by these peer-to-peer networks.

Tor's use of directory authorities to provide a centralized membership directory help prevent Sybil [Dou02] and partitioning attacks [DDM03, DS08]. On the other hand, Tor's use of directory authorities is its own security concern as they provide a mostly central source of trust and are vulnerable to denial-of-service and other targeted attacks. The reliance on these directory authorities is also a scalability concern if the Tor network size grows in orders of magnitude. As Tor relays and clients download signed network consensus documents, each member has an identical global view of the network. This limits route fingerprinting attacks [DC06] where an attacker can infer information about a user by the paths that they select. The path selection made by users in anonymity networks such as Tor, Tarzan [FM02], and Torsk [MTHK09] are influenced by their view of the network, so different network views can create fingerprintable network paths. A downside to this arrangement is that distributing this global network view does not scale if the network grows to millions of nodes. While some peer-to-peer designs use structured topologies to provide distributed global network views, other peer-to-peer designs use unstructured topologies where nodes have only a limited network view. These peer-to-peer designs often use distributed hash tables (DHTs) to share and distribute their network views. Torsk, a structured peer-to-peer network, is designed to distribute Tor's directory service using a DHT while continuing to use a central authority and a client/relay distinction for traffic relaying.

### 3.1.3 Constant-sized and On-demand Directory Downloads for Clients

Rather than clients downloading information about every relay in the network as they do with the microdescriptor consensus, private information retrieval (PIR) can allow clients to retrieve information about specific relays from directory servers without the servers learning which relays were queried. Achieving this privacy property has computational and communication overhead, so PIR [CGKS95] generally comes in two flavours: information-theoretic PIR (ITPIR) and computational PIR (CPIR). ITPIR schemes require multiple non-colluding servers while CPIR schemes require only a single computationally bound server. PIR-Tor [MOT+11] proposes two different architectures based on each of these PIR flavours. The CPIR-based architecture uses Tor's directory servers as CPIR servers, allowing clients to privately request relay descriptors from these servers. A significant problem with this approach is the linear-scaling (in the size of the network) CPU overhead of CPIR on directory servers, requiring that clients limit the number of PIR requests that they make and instead re-use relays for multiple circuits. The ITPIR-based architecture uses a client's three directory guard relays as its ITPIR servers. The scheme is private as long as all of these guard relays do not collude. ITPIR does have a lower CPU overhead compared to CPIR. A common problem of both architectures is that Tor's circuit path selection algorithm must be restricted, particularly in the case of relay exit policies, relay families, and any other future path constraints. PIR-Tor proposes returning several descriptors per query and

allowing the client to choose its preferred relay, but it is not obvious that this approach is scalable. The CPU overhead of PIR also enables a denial-of-service attack against the PIR servers, so PIR-Tor proposes proof-of-work puzzles to limit the frequency of client requests. This has the potential to restrict low-performance mobile devices from the network. As these directory servers cannot influence which relays clients query nor can they learn which relays were queried, PIR-Tor also protects from route capture and route fingerprinting attacks.

Much like PIR-Tor, ConsenSGX [SG19] allows clients to request relay information from directory servers in a privacy-preserving manner. Rather than using PIR, ConsenSGX combines oblivious RAM (ORAM) [GO96] and trusted execution environments to provide more-efficient directory servers. Trusted execution environments such as those provided by Intel SGX allow programs to execute in secure containers called *enclaves*. These enclaves provide hardware-based confidentiality and integrity, preventing even privileged programs or users outside of the container from inspecting or modifying the container's state. Trusted execution environments typically do not protect against side channel attacks. ORAM is a cryptographic primitive designed specifically to hide memory access patterns, and can be combined with trusted execution environments to limit these side-channel attacks. While ORAM has large communication overhead, its *ORAM controller* can be securely run in a relay's trusted execution environment to significantly reduce the real-world effect of this communication overhead [SGF18]. The end result is an architecture that provides similar properties to PIR-Tor's CPIR architecture, but with much better performance at the expense of relying on trusted hardware to prevent route capture and route fingerprinting attacks.

Tor's circuit construction requires the client to have a copy of the microdescriptor for each relay in the path. These descriptors contain keys required for Tor's circuit handshake. While the previous approaches focus on privately downloading select relay descriptors before constructing circuits, Walking Onions [KMG20] allows clients to begin building circuits obliviously without any pre-existing relay keys or descriptors. Specifically the client allows the intermediate relays to select a random path during circuit construction, but in a way that allows the client to verify that the path was chosen randomly and not influenced by any intermediary. Keys required for the client to complete its half of the circuit handshake are delivered to the client only after extending the circuit. Walking Onions introduces new authenticated directory documents required to support the protocol, namely the *ENDIVE* network directory document, *SNIP* relay entry documents, and a small constant-sized *network parameters document*. The ENDIVE and network parameters documents play the role of Tor's consensus document, while the SNIPs act similar to Tor's current microdescriptor documents. Only relays download ENDIVE documents and the entire set of SNIPs, while both relays and clients download the network parameters document. Clients receive SNIPs during circuit extension and use them to complete the circuit handshake as well as to verify the correctness of the circuit construction. Walking Onions provides two circuit

extension designs, one a telescoping design similar to Tor's current iterative circuit extension, and the other a single-pass design that reduces communication to a single round trip. The telescoping design provides security, correctness, and privacy properties equivalent to Tor's current design, while the single-pass design reduces privacy slightly by relaxing the forward secrecy of circuit creation (but not of transported data). Like ConsenSGX, Walking Onions has a much lower CPU overhead for relays compared to PIR-Tor's CPIR architecture and additionally has no reliance on trusted hardware. It also does not significantly change Tor's security model. Walking Onions also requires restrictions on path selection and does not easily work with Tor's existing path selection algorithm.

While these three approaches provide better scaling for the network, they all require considerable architectural changes to the network. Walking Onions is currently in the process of being organized into a technical specification [Mat20], but there are still many remaining design questions to support all of Tor's existing features such as exit policies, relay families, onion services, consensus and voting protocols, migration plan to prevent network partitioning-based fingerprinting attacks, etc.

## 3.2 Routing Performance

The performance of Tor's volunteer-run relays is fundamental to the health of the network, and there has been plenty of work to monitor and improve this performance [AG16]. For interactive applications such as web browsing, the end-to-end latencies of connections over the Tor network have a large impact on usability. Prior work has shown that relays are the main contributors to delays in Tor, largely due to relay congestion [DSR+10]. These delays vary greatly across relays and over time making them unpredictable. While these observations were made before some significant priority-based congestion control mechanisms were implemented in Tor relays, it is still expected that relay congestion negatively affects latency, and therefore usability, for interactive applications.

### 3.2.1 Environment-aware Path Selection

As relays in the Tor network come with varying network and hardware capabilities, it is important to distribute network traffic across these relays in such a way that the congestion at any one relay is limited. To this end Tor clients use bandwidth-weighted random circuit path selection. It is important for each client to select relays following a common set of rules and data to prevent route capture and route fingerprinting attacks where an adversary could influence the path of

users' circuits, or infer information about the client's environment from the relays they choose. Tor has several different bandwidth measurements, most of which are used mainly for metrics and not path selection. Each relay monitors its own bandwidth usage in order to provide metrics to the directory authorities. These metrics come in the form of a single observed bandwidth value and separate network read/write histories. The *observed bandwidth* is the maximum sustained bandwidth over a ten-second period in the past five days and is published by the relay in its descriptor document. While directory authorities publish a capped version of each relay's observed bandwidth in their network status votes, it is only used as a backup bandwidth measurement since relays can provide false bandwidth information in an attempt to influence clients' circuit selection behaviour. Instead the network uses *bandwidth authorities* who attempt to measure the available bandwidth of relays. Accurately measuring this bandwidth for each relay from an external source is difficult in practice and is an active research area [SB09, JJH+17, TJJ20]. These measurements by the bandwidth authorities are also published in the directory authorities' network status votes, and if enough relays have been measured, the network consensus document uses these bandwidth measurements as *consensus weights*. These consensus weights directly influence clients' circuit path selection.

Circuit path selection is important for application performance, network load balancing, and anonymity. Tor's current path selection is not ideal from an application performance perspective, and several path selection algorithms have been proposed to improve performance. As relays are positioned around the world, the geographic path that connections take over Tor are usually much longer than the path the same connection would take without an overlay network. This increased path length adds inherent latency to all Tor connections as traffic is required to pass through these relays before being delivered to its destination. To reduce this latency, as well as to limit the likelihood of routing traffic through relays under the observation of the same ASes, LASTor presented a user-tunable path selection algorithm that attempts to choose relays along the geographic path from the client to destination server [AYM12]. It uses a clustered weighted shortest path algorithm that infers geographic locations of relays from their autonomous system. As this circuit path leaks information about the client's location, users are able to tune the path selection between privacy and utility depending on their needs. While in small-scale experiments this path selection algorithm improved latency, it was later found to perform poorly in more realistic conditions in both throughput and time to first byte [WTBS13].

Like LASTor, many earlier path selection algorithms focused on link latency, but as most of the delay comes from intra-relay delays caused by congestion, relay congestion based path selection has been found to provide better performance. The first approach to use relay congestion for path selection relied on active and opportunistic probing of a client's circuits [WBFG12]. Each relay would maintain a long-term history of measured congestion times and randomly choose relays for circuit using weights that prioritize relays with low congestion. One problem with this

approach is the high variability of relay congestion over time (especially over the course of a day). With thousands of relays on the network today and a high relay churn rate, a client who only uses Tor occasionally is unlikely to use enough circuits to gather a significant amount of relevant congestion measurements.

### 3.2.2 Congestion Control and Prioritization

While the majority of connections on the Tor network are for interactive web browsing, in 2008 a large fraction of data leaving the network was found to be for bulk data transfers such as file-sharing [MBG+08]. As this non-interactive traffic can harm the latency of interactive traffic, better congestion control mechanisms were needed for relays themselves. More recent measurements from 2016 have shown a decrease in non-web Tor traffic, about 24% down from about 51% in 2008, although the measurement method was significantly different in the two cases [JJ16]. In order to improve the performance and usability of interactive applications, priority-based circuit scheduling was introduced as a measure to improve latency [TG10]. Rather than Tor's round-robin scheduling, this approach prioritizes bursty circuits by assigning each circuit an exponentially weighted moving average (EWMA) of the number of sent cells. As cells are moved from their circuit queues to their corresponding relay connection's outbound buffer, circuits with lower EWMA counts are given higher priority. While the time to download small files (simulating interactive traffic) was improved, the improvement was limited due to the long queueing delay between scheduling and the sending of cells on the network. Later work also showed a significant reduction in throughput for bulk downloads when intermediate relays use the EWMA scheduling algorithm [JH12].

Further investigation found that cells spend most of their time in kernel socket outbound queues. To reduce the queueing delay between scheduling and the sending of cells on the network, Jansen et al. proposed a smarter connection scheduler [JGW+14]. This kernel-informed socket transport (KIST) connection scheduling algorithm moves the queueing delay from the kernel's outbound queues to Tor's internal cell queues, giving Tor better circuit and connection scheduling control. This approach is complementary to the EWMA circuit scheduling algorithm. For each relay connection, the KIST connection scheduler queries the kernel to learn how much data the socket can actually send (calculated using TCP-specific information such as the sending congestion window, maximum segment size, amount of unACKed data, and amount of buffered data) and writes at most that much data. This keeps kernel outbound queues small and maximizes Tor's control over data priority. Both EWMA and KIST are in use in Tor relays today.

To limit congestion, Tor uses a simple window-based flow control algorithm that limits the number of in-flight cells on circuits and streams. Since relays communicate at the application

layer, the Tor network breaks the end-to-end congestion control principle and instead uses an entry-to-exit flow control for both circuits and streams. This flow control is in addition to the link-based TCP congestion control between relays. Tor's flow control uses Relay{SendMe} cells to notify the opposite end of the stream when it is allowed to send more cells. Each of these cells refill 10% of the corresponding window. To examine the performance of Tor's flow control, AlSabah et al. studied the effect of smaller window sizes and dynamic window sizes on download times and time to first byte [ABG+11]. While these new window sizes provided better performance in some cases, they also significantly worsened performance in others. They also replaced Tor's fixed-window entry-to-exit flow control with the N23 per-link flow control scheme [KBC94], finding that web client performance improved noticeably with a small overall reduction in performance for bulk downloads.

## 3.3 Onion Services

Tor allows users to run *onion services* (originally called *hidden services*), which are services accessible only over the Tor network. These onion services have similar anonymity properties to Tor clients: the IP address of the server is hidden from users accessing the service. Tor's onion service infrastructure has undergone several changes to improve the privacy and reliability of onion services. Onion services are accessed by users through Tor by making requests to the service's Tor-specific ".onion" domain. Connections to onion services generally occur in three stages: a lookup stage where the client asks the hidden service directory servers for the service's *hidden service descriptor*, which contains a list of its *introduction points*; an introduction stage where the client presents itself to one of the service's introduction points; and a rendezvous stage that creates the communication channel between the client and service. The end result is that the client and service each make outbound anonymous circuits to a relay designated as the rendezvous point, which links the two circuits together allowing them to communicate [Tor20d].

The server associated with an onion service connects to the network using circuits from a Tor proxy much like clients do. This creates a bottleneck for onion services where data from all users accessing the onion service must pass through this single proxy. This bottleneck is also a popular denial-of-service target for malicious parties who wish to limit access to an onion service [Gou19]. *OnionBalance* [CK20] takes advantage of Tor's onion service protocol to publish a hidden service descriptor that combines introduction points from several proxies. This allows onion services to load balance their traffic across several Tor proxies. The proxies are not required to run on the same server hardware, which increases both performance and reliability. The methods used by OnionBalance add some implementation and protocol complexity, and OnionBalance has limitations such as a maximum of 60 proxies.

While our work focuses on the architecture of relays rather than proxies, the architecture of Tor's proxies are very similar (they use the same code base) and would also benefit from a multi-threaded design. We discuss this further in Section 7.2.1.

## 3.4 Scalable Network Service Designs

Tor relays use an event-driven design as we will discuss later in Chapter 4, but there are other ways to build network services, all with different performance, scalability, and development tradeoffs. Event-driven designs typically rely on an eventloop to wait for events and perform non-blocking actions when those events occur, which is important for network applications that must process many concurrent high-latency connections. These event-driven network applications use a non-blocking network API to prevent the application from blocking when it could otherwise be handling other connections. While this design tends to be highly performant on single-core hardware, it is difficult to scale across multiple cores (or multiple servers) and obscures the application's control flow from developers. These problems have led to architectures such as SEDA [WCB01], which divides the application into several stages running asynchronously, each with their own eventloop. Stages are connected by event queues and pass events along them. This resembles a mix of the actor and communicating sequential processes (CSP) programming models, where a fixed number of stages communicate only over buffered event queues. While SEDA does offer useful properties such as improved modularity and adaptive resource usage, it does not solve the fundamental control flow problems of event-based designs. Thread-driven designs are another approach to designing network service as they significantly improve the control flow problem of event-driven designs, but in some cases can only support a small number of simultaneous connections due to overhead from scheduling and large stack sizes when many threads are used. Thread-based designs hide the need for asynchronous I/O by hiding the eventloop and scheduling behind a custom blocking network API.

Capriccio [vBCZ+03] takes this thread-driven approach but uses user-level threads and cooperative threading to significantly decrease the overhead of initializing and scheduling threads, enabling many more threads to be run concurrently. As these are user-level threads, they do not run in parallel. The Tasks language model [FMM07] provides another approach to cooperative threading by translating annotated task-based code to equivalent event-driven code. This translation uses continuation-passing style (CPS) to preserve control flow while allowing the scheduler to interleave tasks. While both SEDA and Capriccio are not ideal for all network services, these techniques are supported by languages designed for network I/O concurrency such as the Go programming language. Go supports goroutines, which are light-weight user-level threads that act similar to coroutines. Unlike with system threads, applications can have millions of gorou-

tines at a time and execute them over a smaller number of system threads. These goroutines can also be used as logical stages linked by channels, but unlike SEDA, these stages will all use the same underlying eventloop/scheduler. Recent work has re-visited the performance of user-level threads by developing a custom application runtime that allows applications to use the thread-driven model with competitive performance to event-driven applications [KB20]. The production-grade high-performance Memcached key-value store, which uses the libevent library as Tor does, was converted to a thread-per-session model and evaluated using this new application runtime. The thread-based version performed better than the event-based version in many cases.

## 3.5 Summary

Tor has incorporated several network scaling and performance designs from the research community such as the KIST connection scheduler, the EWMA circuit prioritization algorithm, and the ntor circuit handshake protocol. Other research designs such as Walking Onions are currently being drafted as technical documents in preparation for eventual implementation. Research designs that do not get implemented in Tor sometimes influence eventual designs that do. For example, the latest proposal for round-trip time congestion control [Per20] builds upon the dynamic window size ideas by AlSabah et al. mentioned earlier, as well as other Tor and networking research.

Our work on improving the throughput of CPU-limited relays using multi-threading is largely orthogonal and complementary to the network scaling and performance improvements described in this chapter. A multi-threaded relay architecture enables an increase in the network capacity *without* increasing the network size or requiring any network architectural changes. Distributing the work across multiple threads can also reduce processing and queueing delays, improving latency. Network architecture changes will be required sometime in the future to scale beyond the improvement provided by a multi-threaded relay architecture, but these network-level scalability solutions are much more difficult to deploy. Better congestion control and circuit selection algorithms are also important for improving application latency and throughput due to Tor's fixed-window flow control, but do not replace the need for increased total bandwidth capacity available on the network.

# Chapter 4

# Current Tor Relay Architecture

Before presenting a multi-threaded relay architecture, it is important to discuss Tor's current relay architecture. The multi-threaded architecture is heavily inspired by this current architecture and they share many similarities. Since we are interested in parallelizing the end-to-end processing of most network data through the relay, this chapter focuses on Tor's networking, its processing of different cell types, and how data is communicated between objects.

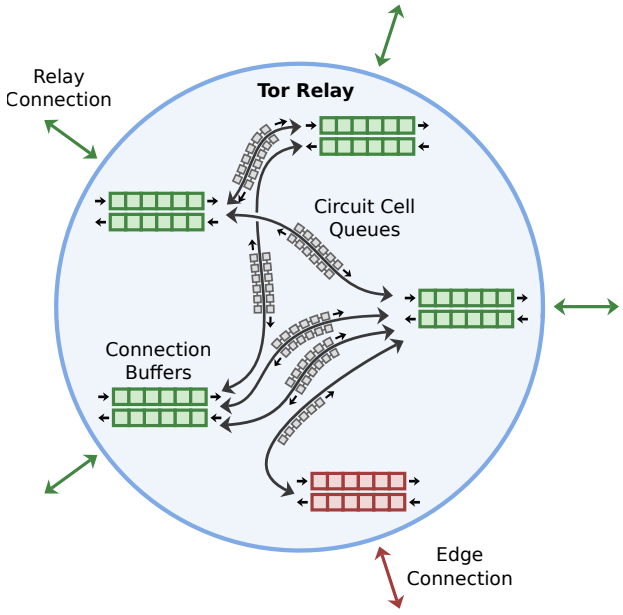



Figure 4.1: A basic overview of the flow of relayed data within a Tor relay.

As Tor is an overlay network, multiple streams can be multiplexed over a single circuit and multiple circuits can be multiplexed over a single connection. Due to this multiplexing, data received on one connection is distributed to many other connections as visualized in Figure 4.1. Relays must receive data on one connection, determine which circuit (and possibly which stream) it belongs to, process it, and likely send it out on another connection. This means that within a relay, connections are intrinsically linked to other connections with circuits being the bridges between any two connections.

Relays classify their TCP connections into several types depending on their purpose. The important connection types for this discussion are *relay connections* (connections between Tor processes) and *edge connections* (connections entering or leaving the Tor network). For each connection Tor maintains an inbound and outbound buffer for reading/writing to the connection's socket. Circuit objects hold state for each circuit that passes through the relay (for example its forward and backward session keys) and keep references to any associated relay or edge connections. Circuit objects also keep outgoing *cell queues*, which are FIFO linked lists of cells that are to be sent on a relay connection. As circuits are bi-directional, circuit objects have two cell queues for each of their two relay connections: one for the forward direction (away from the circuit's origin) and one for the backward direction (towards the circuit's origin). It is important to note that Tor also has channel objects which are a cell handling abstraction, but they have a 1:1 relationship with relay connections and in practice there is not a clear distinction between the two, so we subsume the functionality of these channels into the relay connections for simplicity.

Relays use an event-driven design that runs an eventloop to wait for non-blocking network, timer, and signal events and run their corresponding event callbacks. For example, each connection has socket read/write events, and there are periodic events that run to check memory usage, perform scheduling, and more. This eventloop uses the high-performance libevent library which on modern Linux systems uses the epoll API as its backend event notification system. At the start of an iteration of the eventloop, libevent checks the trigger conditions for its pending events (socket state changes, timer, etc), and if any have triggered, adds them to a queue of active events. Active events are then processed sequentially until the queue is empty, at which point the next eventloop iteration begins and libevent re-checks its pending event trigger conditions. Event callbacks can manually add events to the active queue regardless of their trigger conditions (or even events with no trigger conditions), and these events will be processed within the same eventloop iteration before re-checking event triggers. An unfortunate side effect is that manually activating events will prioritize them over events waiting on their event triggers such as socket events. As this can delay the reading and writing of network data, or possibly cause infinite event chains that cause the eventloop iteration to never complete, Tor uses its own *post-loop events*, which when processed will cause any events activated after that time to be run in the next eventloop iteration rather than the current iteration. Tor uses these post-loop events for a small number of events.

## 4.1 Circuits, Cells, and Connections

The core responsibilities of relay connections are to receive, process, and send cells. When data is received on a socket corresponding to a relay connection, the eventloop runs the connection's read event callback. This callback reads and decrypts the data from the socket using the OpenSSL library and stores the decrypted data in the corresponding connection's inbound buffer. The relay then immediately unpacks the data from this buffer into cell objects and acts on them individually according to their cell command. If this results in new cells being prepared to send on a relay connection (for example forwarding cells, extending circuits, or performing the link handshake), these new cells are either added to a circuit's outbound cell queue, stored in the next relay connection's circuit multiplexer, or written directly to the relay connection's outbound buffer. If they have not already, these new cells will eventually make their way to a connection's outbound buffer. When a relay connection has data to send, the eventloop waits for this connection's socket to become writeable and runs its write event callback. This encrypts and writes the data from its outbound buffer to the socket. Tor may also perform this encryption and socket writing outside of the connection's write event callback, for example during the KIST scheduling loop.

### 4.1.1 Connection and Circuit Scheduling

Relays use a scheduler to improve prioritization and reduce queueing delays on outbound relay connections (as discussed in Section 3.2.2). It is the scheduler's responsibility to inform relay connections when they should send cells and how many they can send. Those connections then choose their highest-priority circuits and copy cells from those circuits' cell queues to the connection's outbound buffer as shown in Figure 4.2. Each relay connection maintains its own circuit multiplexer (CMUX) in order to prioritize circuits based on their past usage. These circuit multiplexers follow a CMUX policy. The only currently implemented CMUX policy is the EWMA policy, which prioritizes bursty circuits and circuits that have not been used recently. As a CMUX policy might need to store data for each circuit (for example the circuit's EWMA value), a global hash table stores circuit-specific data for the CMUX policy. Schedulers may also do additional processing on top of simply informing relay connections when they should send cells. For example, Tor's KIST scheduler will also tell each connection to write its outbound buffer to the socket immediately within the scheduling loop (as opposed to waiting for a socket write event). Tor's scheduler only acts on circuits' cell queues, meaning it plays no role in cells that are added directly to a relay connection's outbound buffer. This also means that the scheduler is not involved in writing to edge connections, which do not use cell queues.

The scheduler and each circuit multiplexer must be notified of a variety of events. The scheduler must be informed when relay connections want to send more cells, and when they have cells waiting to send. The former happens after connections write to their socket, and the latter happens when cells are added to a circuit queue. The scheduler itself also needs to accept global updates from reloaded configuration files and new network consensus parameters. Circuit multiplexers must also be notified when circuits are attached or detached from connections, when cells are moved from circuits' cell queues to the connection's outbound buffer, and when there are more cells waiting on a circuit's cell queue. The scheduler also accesses these circuit multiplexers directly in order to compare and prioritize connections globally, and maintain a global priority queue of all pending connections.

### 4.1.2 Extending Circuits and Relaying Data

When a relay receives a request on a relay connection to create a circuit, it must register a new circuit, process the circuit handshake (also known as an *onionskin*), and send a reply on the same relay connection. As described in Section 2.2.1 and visualized in Figure 2.4, the new circuit

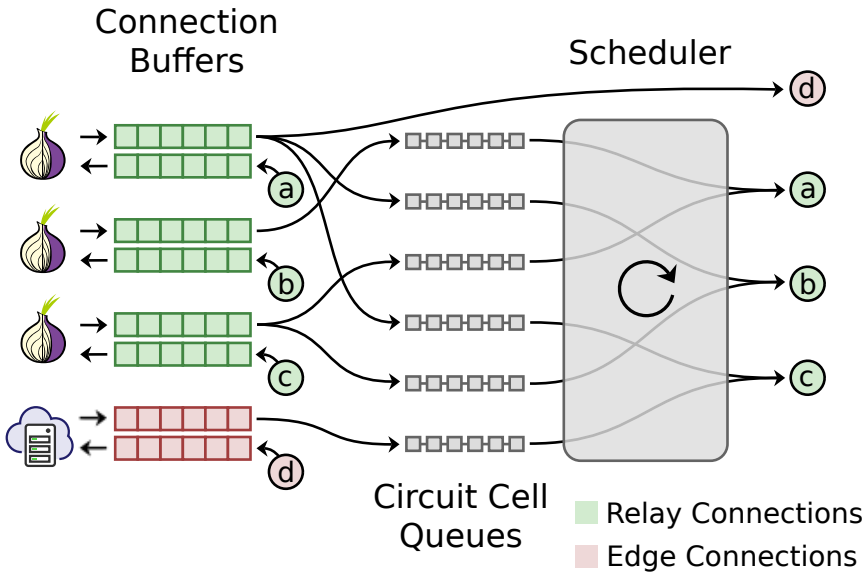



Figure 4.2: Tor's connection scheduler sits between the cell queues and relay connections' outbound buffers, and requires global knowledge about each connection and cell queue. Each onion represents a relay. The Tor logo by The Tor Project, Inc. is licensed under CC BY 3.0 US.

request is received as a Create2 cell. A new circuit object is created with the provided circuit identifier and stored in a global hash table so that when future cells are received on a relay connection with a given circuit identifier, the corresponding circuit object can be found quickly. This table is also used to prevent duplicate circuit identifiers for a given relay connection. The new circuit object is also stored in a global list of all circuits. Processing the circuit handshake is left to a threadpool (see Section 4.3), which when complete moves the result back to the main thread and is used to add a new Created2 cell to the circuit's backward cell queue. At this point the circuit has not been configured for any streams or extended to any other relays, so the circuit is initially linked only to this relay connection on which it was created.

The new circuit can now receive Relay cells. After looking up the circuit object for the given circuit identifier in the global hash table and decrypting these cells using a key from the circuit handshake, the received cells will either be *recognized* (by the process described in Section 2.2.1) if they are destined for this relay, or *unrecognized* if they are to be forwarded on to another relay. If the cell is recognized, the relay unpacks the cell contents, looks for an edge connection attached to this circuit with a matching stream identifier, and processes the relay command in the context of the edge connection. If this circuit has not yet been extended to other relays, any incoming Relay cells must be recognized. In addition if the circuit does not have any streams, any incoming Relay cells must also be *control cells*. These control cells are simply Relay cells with a stream identifier of 0, meaning they do not belong to any stream.

If the relay recognizes a Relay{Extend2} cell, it parses the cell to obtain information about which relay it should extend the circuit to. It searches for the best connection to this relay in a global hash table of existing relay connections, and if an existing relay connection was found, the circuit is linked to this connection. A new unique circuit identifier is obtained from this other connection and is packaged into a new Create2 cell. This new cell is then added to the circuit's forward cell queue so that the scheduler will later send the cell on its circuit's linked connection. If an existing connection was not found, a new relay connection is launched asynchronously. This involves adding the connection to a variety of global lists and hash tables. The Create2 cell in this case is saved and later sent once the new relay connection is established and the link handshake complete. Regardless of how the Create2 cell was sent, sometime in the future a Created2 cell will be received on this other connection once the circuit has successfully been extended. Like with other cells, the corresponding circuit object is retrieved from the global hash table and a Relay{Extended2} cell is appended to the circuit's backward cell queue to be eventually received by the circuit's origin. Now that this relay is no longer the last hop in the circuit, it is possible for the relay to receive unrecognized Relay cells on this circuit. The cell's circuit identifier is simply replaced with the circuit identifier for the next hop in the circuit and the Relay cell is appended to one of the circuit's cell queues.

While these steps establish a circuit, the client may still wish to open streams through this

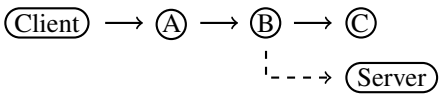
circuit using Relay{Begin} cells. After recognizing one of these cells on a multi-hop non-rendezvous circuit, the resolver resolves the destination address and makes a new edge connection to this server. The circuit and edge connection are mutually linked, and later during another event callback if the connection was successful, a Relay{Connected} cell is appended to the circuit's backward cell queue. Now that the relay has a working circuit with an attached stream, it is possible for the relay to receive recognized Relay{Data} cells. In this case the circuit and stream delivery windows for flow control are decremented and the data from the cell is copied directly to the stream edge connection's outward buffer. If a Relay{SendMe} flow control cell is to be sent, it is added to the circuit's backward cell queue. On the other hand, when an edge connection receives stream data, it is packaged into Relay{Data} cells and appended to the circuit's backward cell queue.

### 4.1.3 Passing Data Between Connections

Looking at some of the cells that are fundamental to a relay's onion routing in Figure 4.3, we can see that relays move data from one connection to another using four primary mechanisms:

- Adding cells to a circuit's cell queue so that they will be later sent on another relay connection by the scheduler.
- Copying data directly to another connection's outward buffer.
- Storing cells within a circuit object, but not within the circuit's cell queues, so that they will later be sent by another relay connection.
- Adding cells to a queue within another connection's circuit multiplexer so that they will be later sent on the other relay connection by the scheduler.

These four mechanisms highlight only the flow of data between connection objects, but data may also flow between connections and other objects. For example, the relay's connection padding subsystem runs in the eventloop with timer events and can add Padding and PaddingNegotiate cells directly to relay connections' outward buffers. Connections may not always need to pass data on to other objects and instead may write data immediately to their own outward buffers. For example the Versions, NetInfo, Certs, AuthChallenge, and Authenticate cells used for link handshaking do not require interaction with other connections.




(a) An example 3-hop circuit with a stream exiting from the 2nd hop. While streams typically exit from the last hop, this configuration must be supported by all relays in order to be compatible with the Tor network. This configuration also simplifies the presentation of different events that a relay must consider, allowing us to describe both circuit and stream events in a single figure.

| From Conn. | Incoming Event/Cell | Outgoing Event/Cell | To Conn. | How Information is Shared Between Connections |
|---|---|---|---|---|
| Ⓐ | Create2 | Created2 | Ⓐ | N/A: Same connection. |
| Ⓐ | Relay{Extend2} | Create2 | Ⓒ | Added to circuit cell queue. |
| | | *Open new relay connection* | Ⓒ | Saved in circuit object. |
| Ⓒ | *Relay connection established* | Create2 | Ⓒ | N/A: Same connection. |
| Ⓒ | Created2 | Relay{Extended2} | Ⓐ | Added to circuit cell queue. |
| Ⓐ | Relay{Begin} | *Open new edge connection* | Server | N/A: Nothing to share. |
| Server | *Edge connection established* | Relay{Connected} | Ⓐ | Added to circuit cell queue. |
| Ⓐ | Relay{Data} | *Stream/TCP data* | Server | Data copied directly to edge connection's outward buffer. |
| | | Relay{SendMe} | Ⓐ | Added to circuit cell queue. |
| Server | *Stream/TCP data* | Relay{Data} | Ⓐ | Added to circuit cell queue. |
| Ⓐ | Destroy | Destroy | Ⓒ | Stored in connection's circuit multiplexer. |
| Ⓒ | Destroy | Relay{Truncated} | Ⓐ | Added to circuit cell queue. |

(b) Examples of cells that a relay can receive on different connections (from the perspective of relay Ⓑ), what cells it will typically send in response, and how this data is transferred between the internal connection objects within the relay.

Figure 4.3: Information sharing between connection objects.

## 4.2 Other Components

There are many other many important components of a relay to consider when designing a multi-threaded architecture. While our architecture does not consider every feature that Tor implements, there are a few other relevant components that are described in this section.

### 4.2.1 Bandwidth Accounting

Relays need to keep track of how many bytes they read and write to the network. These records are used to limit the bandwidth of the relay to a limit set by the relay operator, as well as to publish bandwidth statistics in its *server descriptor* and *extra-info document*. Tor uses a few types of bandwidth limits, some of which are global across all connections and others individual to each relay connection. Global bandwidth limits fall into two categories: a token bucket-based limit and an accounting-based limit. The former limits the bandwidth to an average rate (and an optional burst rate) while the accounting-based limit prevents the relay from exceeding some total bandwidth for the day/week/month. These token buckets and counters are updated when a connection reads or writes to the network, and the token buckets are also updated periodically by a timer. If a token bucket is empty when a connection attempts to read or write, the connection will become inactive for approximately 100 ms. It is also possible for relay connections to cause a token bucket to become slightly negative since the number of bytes read or written to the socket in a TLS record may not match the number of bytes read or written to the TLS stream abstraction.

### 4.2.2 Subsystems and Publish-Subscribe Messaging

Tor's subsystem interface provides a simple, standard method for initializing and shutting down a global component of the relay. The *subsystem manager* is designed to simplify and centralize the management of these subsystems. This subsystem manager is useful for allocating and freeing global memory in a single location and for initializing components in a well-defined order. To provide another abstraction for subsystems, the *publish-subscribe* module allows subsystems to publish messages to other subscribed subsystems, and is designed to reduce and make explicit the dependence of submodules on one another. This publish-subscribe module is a subsystem-specific high-level wrapper around the *dispatch* module.

The dispatch module is a static message-passing system. The dispatcher is created from a dispatch configuration, which defines message types, channels, and receivers. The dispatcher can have multiple channels each with its own channel identifier. Channels are implemented as

a queue of messages. Messages sent on a channel are added to the channel's queue, and later removed and processed when the channel is flushed. Each message has a message identifier, which is used to connect messages to receivers. Receivers can register to accept messages with a specific message identifier and provide a callback function to handle those messages. Receivers do not belong to any particular channel and will therefore receive messages with the specified identifier from any channel. There can be many receivers registered for a given message identifier. On the other hand, each message identifier is mapped to a single channel, so messages with a given identifier can only be sent over a single channel. When a channel is flushed it runs the callbacks of receivers for each pending message in the channel. Channels have an alert function that is run when a channel becomes non-empty, which determines how and when the channel is flushed. The publish-subscribe module provides three different alert functions: flush the channel immediately (on-stack), make active a libevent event to run in Tor's eventloop, or do nothing so that the channel flush can be performed elsewhere, such as periodically with a timer event. Once the dispatcher is created, it cannot be updated with new message types, receivers, or channels. In addition to this restriction, the publish-subscribe module requires that subsystems declare the types of messages that they publish and subscribe to at compile time. This makes the publish-subscribe module less susceptible to runtime errors, but at the expense of being very inflexible.

### 4.2.3 Onion Service Rendezvous Points

The Tor network supports *onion services*, which are anonymous services that Tor clients can connect to using a known onion service name. The onion service architecture has many complex components, but for relaying data the most important is the rendezvous component. In order to maintain anonymity for both the client and service, each build their own circuits to a rendezvous relay selected by the client. The rendezvous relay links the two circuits such that cells arriving from one circuit are redirected to its paired/spliced circuit. This means that unrecognized Relay cells arriving at the last hop of a rendezvous circuit (for example the client's rendezvous circuit) will be redirected and sent backward down its spliced circuit (the service's circuit). Unlike regular circuits where unrecognized Relay cells are processed by a single circuit object, unrecognized Relay cells arriving on a rendezvous circuit are processed by two different circuit objects as they move from one relay connection to another. This processing involves encrypting/decrypting the cells twice — once for each circuit.

### 4.2.4 Out-of-memory Handling

Tor relays require buffers for reading/writing connection socket data, as well as cell queues for circuit scheduling. As the relay accepts more connections and circuits, its memory usage typically increases. Tor relays are designed to limit the size of their connection buffers and the Tor protocol itself is designed to limit the length of each circuit's cell queues (the protocol allows for a maximum of 1000 in-flight cells per circuit). To prevent a relay from using too much memory, either due to limited resources or denial-of-service attacks [JTJS14], Tor will destroy some of its circuits and connections if its memory usage is nearing its limit. For every cell that Tor adds to a circuit's cell queue, it computes the total memory usage of all buffers, cell queues, half-closed streams, cached hidden service descriptors, cached DNS entries, etc. If this memory usage is greater than 75% of the system's memory (or 40% if the system has at least 8 GiB of system memory), the relay begins to destroy cached data, circuits, and directory connections until the new memory usage is under 90% of this limit. Circuits and directory connections with the oldest data (the oldest cells in queues for circuits and the oldest memory chunks within a buffer for connections) are destroyed first. This requires knowledge of the relay's total memory usage, as well as a global view of all circuit objects.

## 4.3 Threadpool and Work Queues

Relays support multi-threading for a few specific tasks using a threadpool and work queues. Each relay initializes a threadpool with a thread count configurable by the relay operator. The threadpool contains three work queues of different priorities for its *CPU worker threads* to process, and a reply queue for sending the results back to the relay's main eventloop thread. While each thread processes work entries from high-priority queues first, half of the threads may also choose lower priority work with a small probability. To have both types of threads, Tor requires the threadpool to use at least two threads. Each work entry is analogous to a continuation containing the state of some work and a function to process the work. This threadpool serves two purposes: it allows the relay to move small isolated tasks outside of the main eventloop to improve performance, and reduces the likelihood that these long-running tasks can create significant throughput fluctuations that could potentially aid timing attacks. These CPU worker threads do not use libevent as Tor's main thread does. Instead they simply wait on a condition variable to synchronize access to the work queues. The specific tasks that are processed in CPU worker threads are the following:

- Circuit handshakes and related cryptography (known as processing onionskins).
- Compressing network consensus documents for caching.

- Computing the difference between two cached consensus documents.

The first task listed is performed by all relays whenever they receive requests to create new circuits (from Create2 cells). The latter two tasks are performed specifically on relays configured to act as directory caches, and as these tasks are tied to the consensus period, they tend to be performed about once per hour. While parallelizing these tasks does offer some performance improvement to the relay, the primary task of forwarding cells along circuits is not parallelized and does not scale over multiple threads. This means that in practice relays get little benefit from using more than two CPU worker threads, and these threads are often not used efficiently.

## 4.4 Historical Notes

Tor's eventloop has existed since its original fork from a first-generation onion routing implementation in 2002. At the time this eventloop was very simple, using only a `poll` system call within a loop to check for socket events on a global list of connection objects. While Tor's network and relay architectures changed significantly from that of first-generation onion routing, the general eventloop design of the earlier onion routing implementation shares many similarities with Tor's code today. This onion routing code was developed by an undergraduate student for a final-year project [Syv05], so it is conceivable that scalability was not of high conern. When Tor forked from this code in 2002, multi-core parallelism was not yet commonplace and the network was not large enough to require that relays distribute their processing across several servers, so performance typically favoured event-driven architectures at these smaller scales. In 2005 Tor transitioned to the libevent library for better performing event-notification APIs such as `epoll` and `kpoll`, and better cross-platform support. Parallel processing has also existed in Tor since CPU workers were first added in 2003 to process circuit handshakes.[1] Tor's original circuit handshake cryptography was much slower than the ntor handshake [GSU13, Mat13] used today, so it was even more important then to be able to offload these expensive blocking tasks to other parallel workers. On Linux, Tor originally ran these CPU workers in separate processes rather than threads, but pthreads support was added a couple of years later, and thread-safe workqueue/threadpool support many years after that.

[1] Tor also supported DNS workers, but this functionality was later offloaded to libevent.

## 4.5 Summary

This chapter reviewed the high-level routing architecture of a Tor relay and some of its fundamental objects such as connections and circuits. These basic ideas are important for any relay architecture compatible with the Tor network. While relaying cells is one of the primary tasks of a relay, the relay also has other tasks such as extending circuits, processing directory documents, and many more not described in this chapter. A multi-threaded architecture should aim to parallelize the data-relaying aspect of the relay, but it must do so in a way that is compatible with the relay's other tasks, many of which require some shared state to manage connections. In the next chapter, we present a design of such a multi-threaded architecture.

# Chapter 5

# A Multi-threaded Tor Relay Architecture

In this chapter we propose a multi-threaded Tor relay architecture that parallelizes the end-to-end processing of relayed data as it passes through the relay. We have three primary goals:

- Preserve compatibility with the existing live Tor network.
- Parallelize the end-to-end flow of unrecognized Relay cells throughout the relay.
- Maintain some similarity with Tor's existing relay architecture in order to increase the amount of code that can be re-used in its implementation.

The maximum throughput of a relay, assuming it is not limited by network bottlenecks or configuration options, is largely limited by the rate at which a relay can receive, process, and send cells. The path through the relay that these cells take is often referred to as the fast (or sometimes critical) path. This is the path from reading cells on TLS-encrypted relay connections, parsing the cells, processing them, scheduling their circuits, and writing them on a different TLS connection (or edge connection). This path is largely CPU-limited and the Tor developers have put effort into improving its performance [Mat18, Gou18]. This is the path that we parallelize across threads.

If the total throughput of a relay increases, it can either handle more users at a given per-circuit throughput, or provide higher throughput per circuit for the same number of users. If the relay serves as a bottleneck for the circuit, this higher relay throughput could improve the circuit's overall throughput. For a CPU-bound relay multi-threading has the potential to improve not just circuit throughput, but also latency. Little's law states that for a queue where units are added at a rate of $\lambda$ and remain in the queue for a time $W$, the average number of units in the queue is

$L = \lambda W$ [Lit61]. In the context of a Tor circuit, which acts as a limited-length queue of cells in the network, $\lambda$ represents the average throughput of the circuit, $W$ represents the average latency of the circuit, and $L$ represents the average number of in-flight cells in the circuit. Assuming the average number of in-flight cells $L$ remains constant and the per-circuit throughput $\lambda$ increases, the latency $W$ of the circuit must decrease. This leads to better responsiveness and a better user experience for interactive applications such as web browsing.

## 5.1 Obstacles to Multi-threading

Tor's current relay architecture uses several design patterns that make it unsuitable for multi-threading. Any multi-threaded architecture that is designed to maintain similarity with Tor's current architecture must work around these obstacles. While we could have instead avoided these by issues by redesigning Tor's relay architecture from the ground up (preferably taking advantage of more featureful and safer programming languages), this approach would have a much lower potential of being deployed on the live Tor network. That being said, we also expect that ideas and observations from our architecture may be useful if new relay implementations are built in the future.

### 5.1.1 Global State

Tor makes heavy use of global state and singletons. While this often simplifies the writing of new code that requires access to this global state, these design patterns are well-known obstacles to high-performance parallel programming. Tor version 0.4.2.6 has over 650 global mutable variables (non-constant variables stored in the program's initialized or uninitialized data segments), not including duplicates from C macro expansions or globals used for logging rate limits. Many of these are lists, hash tables, counters, and boolean flags. While parallel accesses to global memory can simply be wrapped with mutexes to provide memory-safety in many cases, they alone do not protect against logical race conditions, possibly introduce deadlocks, and will likely have a non-negligible performance impact. Furthermore, correctly identifying which existing global state will be shared among threads and protecting it adds complexity and significant technical debt. For example, if some particular code is to be run in parallel, it and any lower-level code that it calls must be audited to ensure that any global state (which now has become shared state) is properly protected, and any future changes made to any of this code must be careful to not unsafely introduce global state. To prevent developers from inadvertently sharing global state among threads, code that accesses global state (including code that accesses global state

indirectly through lower-level functions) must be clearly separated from code that does not. In practice this is often difficult and error-prone, and instead global state should be limited.

Problems caused by global state are not exclusive to parallel programs. Global state can introduce unexpected side effects in non-parallel programs as well and can make it difficult for developers to reason about the program's state as a whole. To reduce unexpected side effects, it is often useful to limit the scope of global state to the top-most components of the program, or remove it entirely. The remaining global state can be encapsulated in local data structures and passed to lower-level code as needed through arguments or using methods such as dependency injection. This allows developers to easily identify state that may change in lower-level components of the program and reduce the chance of unexpected side effects. For Tor, this generally means moving many of these global objects to local objects in the mainloop that must be passed explicitly to other components. While this does not reduce the amount of shared state that a multi-threaded Tor might need, and not all shared state is global state, limiting global state would be a large step toward making it easier to translate Tor's existing relay architecture to a multi-threaded architecture without introducing race conditions or other bugs. While our multi-threaded architecture introduces very little shared state, any implementation of it using code from Tor's existing implementation (facilitating our goal of code reuse) must be careful to prevent Tor's large amount of existing global state from becoming shared state.

Below are some examples of Tor's global state related to connections and circuits. These structures are often accessed in many locations throughout the relay. As mentioned in Chapter 4, we ignore the existence of Tor's *channel* objects (different from the channels we discuss later in this chapter) and combine their functionality into connection objects for simplicity.

- Map of connection identifiers to connection objects.
- Map of relay identity digests to connection objects.
- Lists of all connection objects, connection objects that are in the process of closing, and connection objects that are directly linked to other connection objects.
- Priority queue of connection objects for the scheduler.
- Map of connection objects and circuit identifiers to circuit objects.
- Map from connection identifiers and circuit identifiers to circuit multiplexer data and circuit objects.
- List of all circuits.

- Connection token buckets.
- Relay's persistent state and configuration options.

### 5.1.2 Object Ownership and Roles

From the list of a few of Tor's global structures in the previous section, it is clear that references to connection and circuit objects are stored in several different global structures. For example references to connection objects are stored in both a global list of connections as well as a map of connection identifiers to connection objects. Having several mutable references to objects in different data structures is often a sign of a lack of clear ownership of the data. It can be useful in parallel applications for data to have a clear owner who is responsible for creating, accessing, lending, and destroying/freeing this data and its memory. This ownership can help developers prevent data from being inadvertently shared across threads. Rather than sharing memory, it is often better to transfer ownership of it. When the ownership is transferred between threads rather than shared across threads, the memory can be safely used in either thread without locking since it is only owned and accessible from one thread at a time. Our multi-threaded architecture

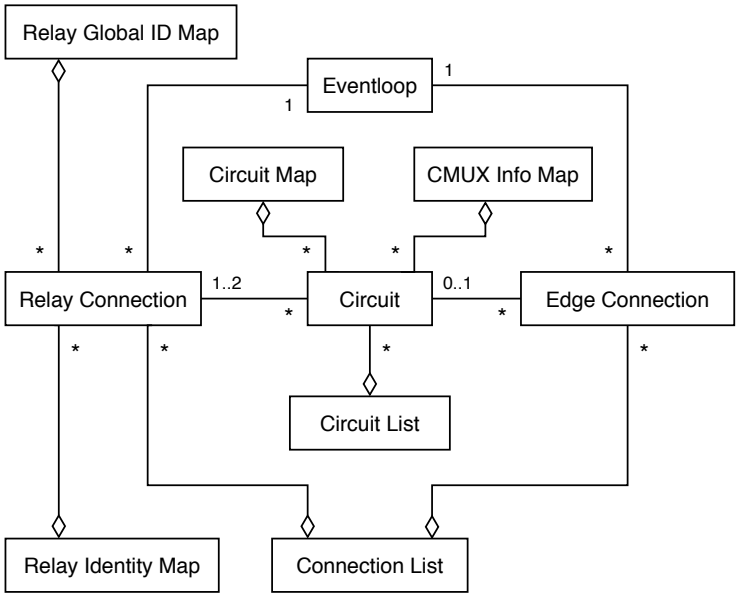



Figure 5.1: Association relationships between select types in Tor. Integers show the multiplicity of associations and diamonds represent shared aggregation. Navigability is generally bi-directional, either due to references or global state, but is not shown.

transfers ownership of connection objects to other threads, so well-defined ownership of this data is important. Alongside clear data ownership, limiting the number of circular references is also important. As discussed in Chapter 4, circuit objects link connections together. Each circuit object is shared by two connections using circular references, so a given circuit object stores references to its two adjoining connection objects and those two connection objects both store references to the circuit object. This means that through the circuit object, each connection can access the other. As connections may have many circuits, each connection object can access many other connection objects. If connections are to be processed in different threads, they cannot have these circular references. The associations between a few of Tor's object types are demonstrated in Figure 5.1, and this web of associations/references makes it difficult to separate connection objects.

As Tor's relays are implemented in the C language, their design is restricted by the limited language features. For example Tor's different connection types all inherit from a common base connection type. As C does not support object oriented programming, Tor's primitive version of inheritance is implemented using structs that encapsulate a common base struct, with some C macros to support downcasting. Much of Tor's networking code uses downcasting in condition blocks, meaning that networking code for the different connection types is not easily separable and changing the implementation of one connection type requires changes to code that handles other connection types as well. This design could be improved significantly with a language that supports object oriented features in a more natural way. In addition, objects in Tor often perform actions outside of what their expected role might be. This is largely due to unexpected side effects caused by Tor's global state. For example, when an edge connection at a circuit's origin receives a Relay{Extended2} cell, the circuit has successfully extended to a new relay. If the circuit construction has finished and there are no more relays to extend to, the circuit is now ready to accept new streams. The circuit then calls a function to attach any pending streams to circuits, which indirectly has access to all pending streams and circuits due to global state. These accesses modify the states of these connections and circuits, and can also result in the generation of new connections being made to other relays. A simple function call acting on a circuit (`circuit_build_no_more_hops(circ)`) is now directly accessing other edge connections and circuits, objects that are not within its scope. These types of side effects are very difficult to work with in a multi-threaded program.

## 5.2 Rejected Designs

In this section we discuss two alternative designs for distributing relay data processing across CPU cores. While likely simpler to implement, they would not be expected to scale well across

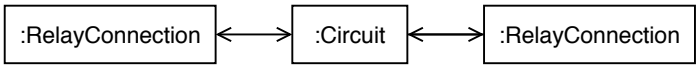
cores and have unnecessary overhead.

### 5.2.1 Threadpool and Work Queues for Expensive Operations

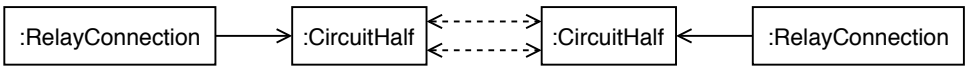
Rather than designing a new architecture for the end-to-end processing of Relay{Data} cells, an alternative design could offload expensive parts of this cell processing to a work queue and threadpool. Much like the circuit handshake processing is offloaded to the relay's threadpool, expensive operations such as decrypting the payload of Relay cells or TLS encryption could be performed by the threadpool while the non-expensive operations remain in the main thread. There are a few problems with this approach. The first is that while cryptographic operations such as TLS reading/writing and cell AES encryption/decryption were found during benchmarking to take a significant amount of time, the majority of the end-to-end processing time could not be blamed on any specific operations. Other expensive operations such as connection/circuit scheduling do not translate easily to a work queue design. If only a fraction of the end-to-end processing can be parallelized, the relay will experience weak scaling across multiple CPU cores and the throughput will be limited by the scaling of the serialized processing in the main event-loop. In addition, using a work queue can introduce additional queueing latency. For example, Tor's ntor circuit handshake processing, which occurs in cpuworker threads, will incur some inter-thread communication latency overhead. Using Tor's statistics dump feature with an exit relay on the live Tor network, this communication overhead (shown as a fraction of the computation time) was measured at approximately 42% (160 µs) in January 2020, and 550% (1.7 ms) in December 2018 (a time when the relay was receiving significantly more traffic). While these numbers are small, moving multiple separate operations to the threadpool using work queues could add significant queueing delays to Tor's fast path, especially when the relay is congested.

### 5.2.2 Running Multiple Relays on the Same Server

As relays are mostly single-threaded, an obvious approach to providing more bandwidth to the Tor network might be to run multiple Tor relays on the same server (possibly one per CPU core). In practice there are a few problems with this approach. The first is that it results in additional relays in Tor's consensus documents. As discussed in Section 3.1, growing the consensus increases the burden on clients and increases the fraction of the network traffic dedicated to delivering directory documents. Another problem is the additional overhead of running multiple relays. Running additional relays on the same server increases the load on directory authorities and directory caches, which must provide additional documents to these relays, and leads to




(a) Tor's architecture: A single circuit object is shared by both connections. Through the circuit, each connection can directly access the other.




(b) Multi-threaded architecture: Neither connection (or circuit half) can access their counterpart directly. Instead they can only communicate through channels.

Figure 5.2: Comparison of the navigability of Tor's circuit objects compared to the multi-threaded architecture's circuits. Dashed lines represent channels.

unnecessary overhead on the server itself. It also limits the relay's ability to load-balance connections across CPU cores. A third problem is that Tor limits each IP address to two relays in order to limit Sybil attacks on the network. While it is possible for servers to have multiple IP addresses, requiring multiple addresses in order to use more than two CPU cores is not a scalable design. Today there is at least one relay operator [Tor20e] who uses eight IP addresses in order to run 16 relays on a single server.

## 5.3 The Multi-threaded Relay Architecture

Our multi-threaded architecture parallelizes the end-to-end processing of relayed data by distributing its connections across additional threads with their own eventloops. These threads are responsible for reading, processing, and writing network data for each of their connections. Communication between threads uses asynchronous messages passed along thread-safe buffered channels that integrate into the eventloop. This architecture focuses on the main onion-routing tasks of a Tor relay such as creating and extending circuits, relaying data, and connection/circuit scheduling. It aims to require little shared state, remove the dependence of connection objects on one another, and provide well-defined ownership of connection and circuit objects. This leads to few required locks and safer memory management. This multi-threaded architecture is complementary to Tor's existing work queue and threadpool. As expensive synchronous operations will block the current thread for a short time and prevent it from processing the thread's other circuits and connections, offloading occasional expensive operations to a threadpool would continue to improve the responsiveness of the relay.

Unlike Tor's current architecture where one connection object can directly access another

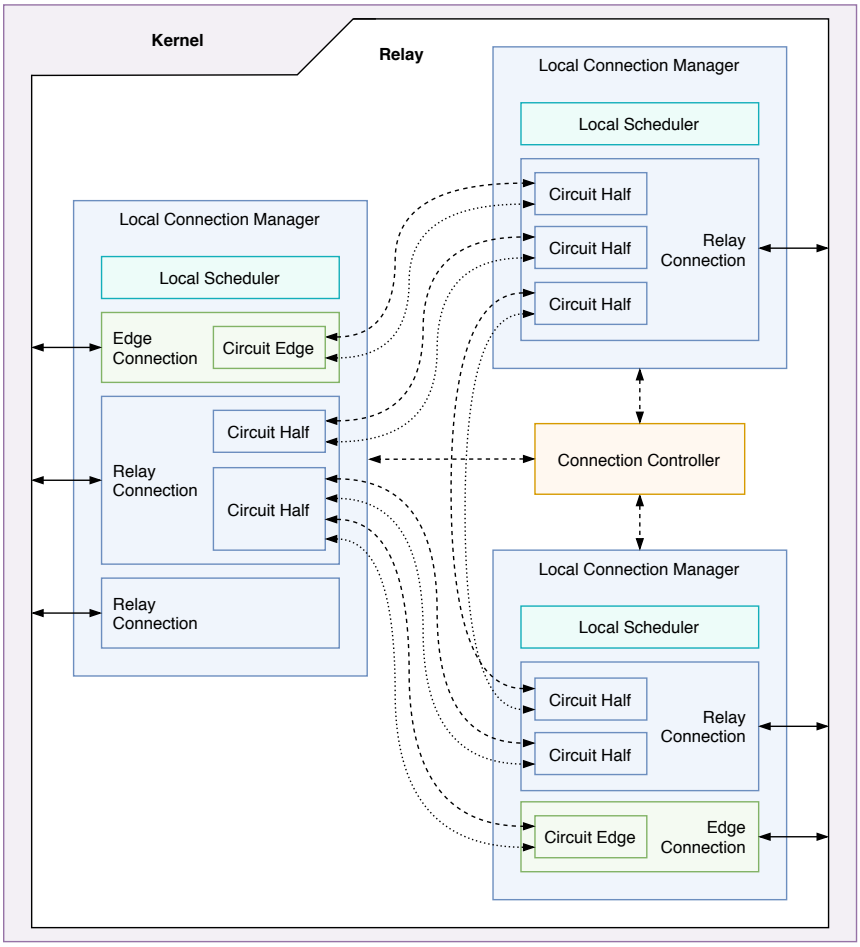
through a shared circuit object, this type of multi-threaded architecture requires clear separation between connections. Since connections should not share a single circuit object, circuits are instead split into two circuit halves, one for each connection. These two circuit objects have no reference to each other and can only communicate over a thread-safe channel as shown in Figure 5.2. This independence allows two connections to be owned by different threads while still allowing their circuits to communicate with limited and well-defined shared state (the channel) and without potentially introducing unsafe memory conditions. Channels are also used for communication between each additional thread and the main thread, which acts as a global controller for the relay.

### 5.3.1 Building Blocks

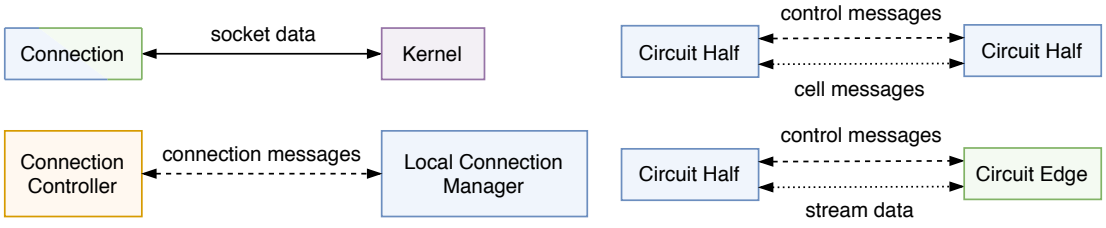
The multi-threaded architecture is made up of several components. Figure 5.3 presents an overview of how these components fit together, and Figure 5.4 describes the relationships between a few of them. This subsection will discuss most of these components and how they overcome some of the obstacles in Section 5.1. The *local scheduler* component is discussed in Section 5.3.3.

**Channels.** Rather than sharing state across threads, channels are used to send messages and pass ownership of data between threads. Each channel should provide the ability for two threads to asynchronously send and receive messages. Channels are bi-directional, buffered, and should integrate into the eventloop such that each end of the channel is notified of messages added at the opposite end. These channels should also support dynamic initialization and allow either end of the channel to be closed asynchronously, in which case the opposite end should be notified of the closure. Tor's current relay implementation already includes the dispatch module (described in Section 4.2.2), which provides a channel construction. While it may be possible to adapt it to the channels we use in our multi-threaded architecture, they are not trivially compatible and are designed for a different purpose.

**Circuit half and circuit edge.** Each connection object stores general data such as its file descriptor, incoming/outgoing buffers, and a local token bucket for rate limiting. Specific connection types might store additional data. For example, relay connections store information about the TLS state, link protocol state, and each of its circuits. Circuit information is stored in circuit objects: a *circuit half* for relay connections and a *circuit edge* for edge connections. Each circuit half has its own cell queue for cells exiting the relay from its half. A circuit half can be linked by channels to at most one other circuit half, but can also be linked to many circuit edges (Tor's leaky-pipe topology). For each other circuit object they are linked to, circuit objects hold ends of two different channels, one for general data/cells and the other for control messages. Distribut-




(a) Overview of the message-passing architecture using three connection threads with several connections and circuits. The connection controller runs in the main thread's eventloop, while the local connection managers each run in their own connection thread.




(b) Types of messages/data that each channel uses.

Figure 5.3: Overview of the message-passing architecture. Channels are represented by dashed/dotted lines.

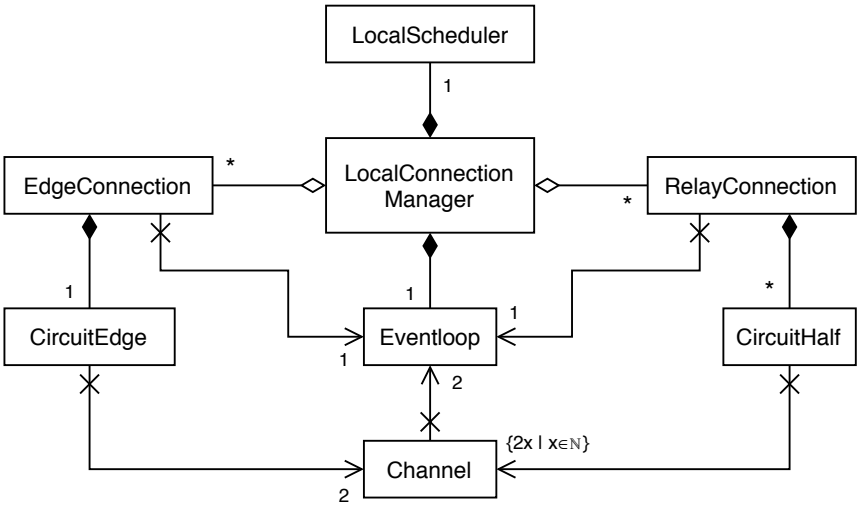



Figure 5.4: Association relationships between select types in the multi-threaded architecture. Integers show the multiplicity of associations, hollow diamonds represent shared aggregation, and filled diamonds represent composite aggregation. Explicit navigability is shown by arrows and crosses to enforce thread-safety.

ing message types across two channels is useful for prioritizing control messages such as circuit destroy messages.

**Local connection manager and connection controller.** As each connection thread must manage many connections, each thread uses a *local connection manager*. This stores each connection that it owns in a hash table (keyed by a globally unique connection identifier), and is responsible for maintaining those connections by attaching them to the local eventloop, refilling their rate-limiting token buckets, adding connection padding, etc. While these tasks do not require a global view of the relay, other tasks required by the relay such as extending circuits or performing denial-of-service prevention do. These tasks that require a global view of the relay are instead performed by the *connection controller* running in the main thread. This connection controller holds the information it requires about all of the relay's connections and circuits, but importantly not references to the connection and circuit objects themselves as these are owned by the local connection managers. For example, this connection controller will maintain a hash table mapping the public identity digests for all connected relays to their corresponding unique connection identifiers (the same connection identifiers stored by its local connection manager). Each local connection manager has a channel between itself and the connection controller for use when it needs to perform tasks that require some global knowledge.

**Eventloop.** We do not define the structure of the eventloop in our architecture, but we assume a libevent-like library where an eventloop object has many individual event objects. These event objects may have a trigger that they are waiting on (for example a socket becoming readable), a callback function, and possibly data to pass to the callback if the language does not support closures. Connections should be able to modify their events and enable/disable them when needed; for example, disabling a socket read event if a token bucket does not allow any more bytes to be read. To avoid circular references, these event objects should not store a reference to the connection objects directly as Tor's existing architecture does. Instead they should store the connections' numeric identifiers so that the local connection manager (the owner of the connection) can look up the identifiers' corresponding connection objects as needed. This reduces the number of mutable references to the connection object but has a performance penalty from the lookup. If in practice this performance penalty is too great, a weak reference to the connection can be stored in the event object instead of its identifier.

### 5.3.2 New Connections

New connections can be created in two cases: the relay requires a connection to an external relay or server, or one of the relay's listening sockets accepts a new connection. Both of these cases are handled by the connection controller. After opening or accepting a connection, the controller chooses a globally unique identifier for the connection. It then transfers this chosen identifier and the ownership of the connection object to one of the connection threads' local connection managers by sending a *connection_register_relay* message along its channel. On receiving the message the local connection manager registers socket readable/writable events for the connection with the thread's eventloop. The event structures should store the connection's numeric identifier as its callback data. The local connection manager then stores the connection object itself and its two event structures in a hash table with the connection identifier as the key. Whenever one of the events trigger, the local connection manager can look up the corresponding connection object in the hash table using the identifier.

### 5.3.3 Connection and Circuit Scheduling

Tor's scheduling loop is performance-critical and requires a global view of all connections in the relay. This is problematic in a multi-threaded architecture since the scheduler would either need to acquire locks on each connection, or use message passing to inform connections about how much data they can send. The large amount of locking would harm the relay's performance, and message passing would break some of the assumptions of Tor's primary scheduler, the KIST

scheduler, which relies on querying TCP information from the socket and immediately writing data to it before the TCP information becomes stale. Rather than using a global scheduler, each local connection manager uses its own local scheduler. This local scheduler only knows about connections owned by the local connection manager and can therefore perform its scheduling operations without locking or needing to synchronize with other connection managers. As the local schedulers have direct access to the connection objects, they are able to write the connection's buffers to the socket directly within the scheduling loop and therefore do not break the functionality of KIST. These local schedulers would work very similar to Tor's existing schedulers, moving cells from circuit queues to connection outward buffers. In practice scheduling is not an operation that must provide perfect prioritization, and we do not expect that using several smaller schedulers rather than one global scheduler would significantly harm the performance of the relay or its circuits as long as connections are reasonably load-balanced across connection managers (and consequently across threads).

This architecture does not provide a definitive solution for load-balancing connections as this would depend strongly on the implementation and performance benchmarks. For each new connection, a trivial method of load-balancing connections would be for the connection controller to assign them to local connection managers in a round-robin manner. In practice this may perform badly as some connections will be more popular than others. Instead it might be useful for connections to occasionally report recent usage metrics (such as an exponentially weighted moving average of the bytes sent/received) back to the connection controller so that it can better load-balance new connections. The controller may also wish to occasionally move connections between local connection managers in order to better distribute connections. Channels can be used to move connections between connection managers, but this requires the connections to be detached from their old eventloops during this communication and would lead to a short delay where no traffic on those connections would be processed. This delay could be minimized if the channel and eventloop implementations support prioritization.

### 5.3.4 Extending Circuits and Relaying Data

When a relay connection receives a Create2 cell, it must check with the denial-of-service prevention subsystem, create a new circuit half object with the provided circuit identifier, and process the circuit handshake. This handshake processing can be performed immediately or by a thread-pool as it is in Tor's current design. As the denial-of-service prevention requires a global view of the relay and runs as part of the connection controller, the query to the denial-of-service subsystem is sent over the local connection manager's channel using a *query_dos_sys* message. Some time in the future a *dos_defense* reply is received and the in-progress circuit is either destroyed or allowed to continue (see Figure 5.5). The final Created2 cell can be added to the circuit half's cell

queue to later be sent by the scheduler. Rather than using a global hash table mapping all (connection, circuit identifier) pairs to circuit objects like Tor's current design, each relay connection has its own hash table mapping circuit identifiers to circuit half objects.

Extending circuits requires communication between connections owned by possibly two different local connection managers. The connection controller acts as an intermediary as it can communicate with all connection managers and has a global view of all connections (see Figure 5.6). The circuit extension process involves parsing the Relay{Extend2} cell, finding (or opening) a connection to the desired relay, creating a new circuit half on this relay connection, linking the two circuit halves using channels, and passing a message along one of these new channels. The two circuit halves are linked when a connection controller generates two new channels and sends opposite ends of each to both circuit halves. This allows the original circuit half to send messages (such as the payload of the Create2 cell) to the new circuit half directly. New streams can be created in the same manner, using the controller to create the new edge connection and linking the original circuit half with the new circuit edge (see Figure 5.7).

Building circuits and streams has some additional complexity compared to Tor's current architecture due to its message passing, but it allows for better separation of connections and circuits from the relay's global components. In addition, dividing circuit objects into two indepen-

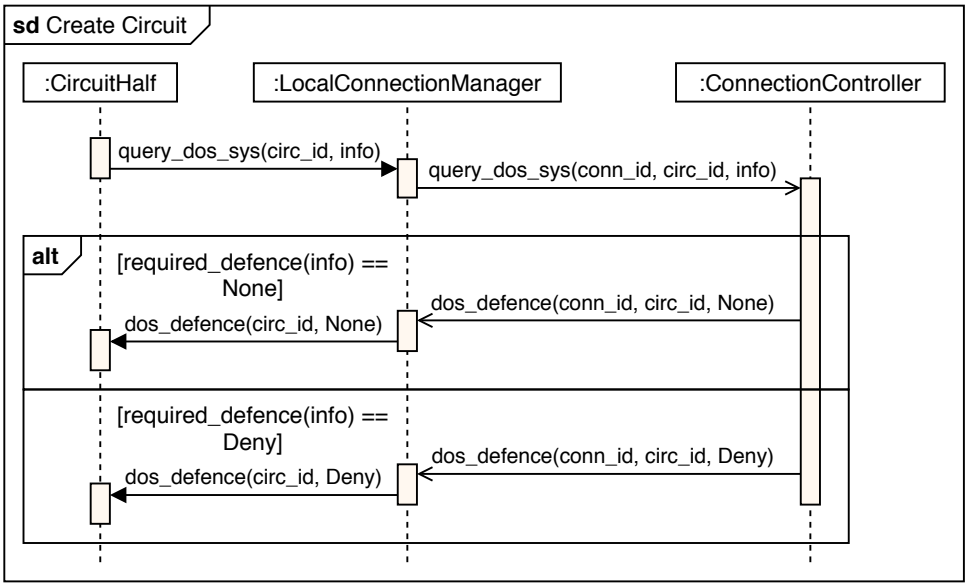



Figure 5.5: Overview of the messages involved in creating a circuit. All asynchronous messages are passed along channels. The relay's denial-of-service subsystem is queried to check if any defences should be applied.

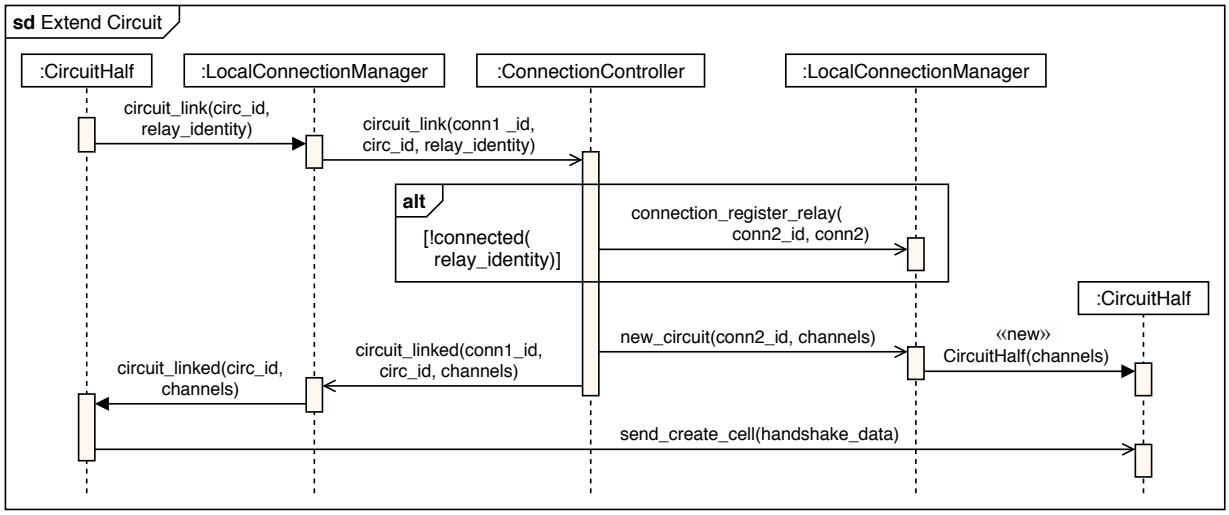



Figure 5.6: Overview of the messages involved in extending a circuit. All asynchronous messages are passed along channels. A new circuit half is created on the relay connection specified by the Relay{Extend2} cell and both circuit halves are given opposite ends of two communication channels. From this point on, these two circuit halves can exchange cells and control messages directly over these channels.

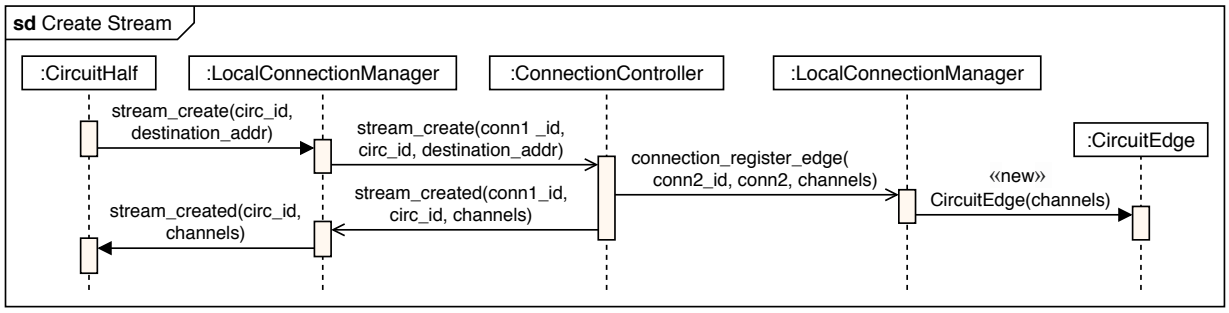


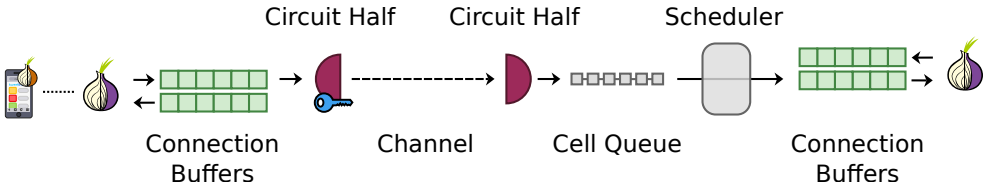

Figure 5.7: Overview of the messages involved in creating an exit stream. All asynchronous messages are passed along channels. A new circuit edge is created on the edge connection created by the connection controller and the circuit half and circuit edge are given opposite ends of two communication channels.

dent halves allows for better separation of connections and allows each circuit half to process cells independently from the other. From a relay's perspective, all circuits are directional and asymmetric; Relay cells travelling forward down the circuit (away from the circuit's origin) may be recognized by this relay, but Relay cells travelling backward can never be. This asymmetry means that a circuit's session keys must always be stored in the circuit half closest to the circuit's origin (generally the one which received the Create2 cell; see Figure 5.8). For general-purpose circuits this means that only one circuit half performs cell encryption or decryption and the other will simply pass the Relay cell on. These circuit halves run independently and are not concerned with processing done by the other. This independence translates naturally to circuits at rendezvous points, which link circuits together as described in Section 4.2.3. In this case, both circuit halves are closest to their own circuit's origin and will therefore each have their own session keys. These circuit halves can be linked with a channel just like general-purpose circuits, but instead both circuit halves will perform their own independent cell encryption or decryption.

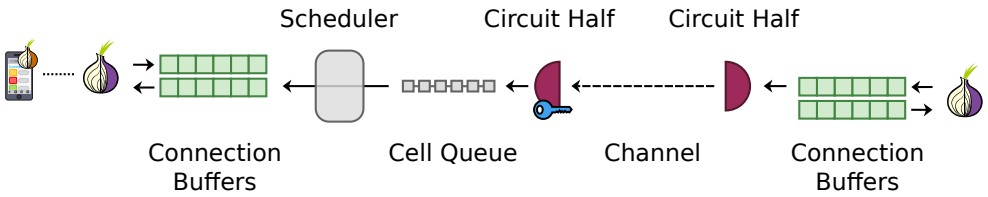
Passing cells between circuit halves is straightforward. If one half wishes to send a cell to the other, it writes the cell payload and its cell command to the cell channel. The other circuit half will be notified of the new cell and can read it from the channel. If it is a Relay cell and the circuit half has a session key, the circuit half may encrypt or decrypt the cell payload. The cell payload and command are then combined with the circuit identifier to form a packed cell, which can then be added to the circuit half's cell queue and later sent by the scheduler like Tor's current architecture. Circuit halves have another channel joining them: the control channel. This channel is designed for messages that should be prioritized over the cell channel. For example, Destroy cells should be sent as soon as possible as there is no reason to process any other waiting cells if the circuit is to be closed. This channel should never be used for Relay cells since these cells must always be processed in order due to their running digest. While a circuit half can be linked to at most one other circuit half, it may be linked to many circuit edges (one for each stream). When a circuit half is linked to a circuit edge, these communicate over a data channel rather than a cell channel. This channel transports raw application/stream data rather than cells. The circuit half is responsible for converting between this application data and Relay{Data} cells, and performing any required cryptography. All information that is shared between connections, including the examples described in Figure 4.3, travel through these cell, data, and control channels.

### 5.3.5 Bandwidth Accounting and Out-of-memory Handling

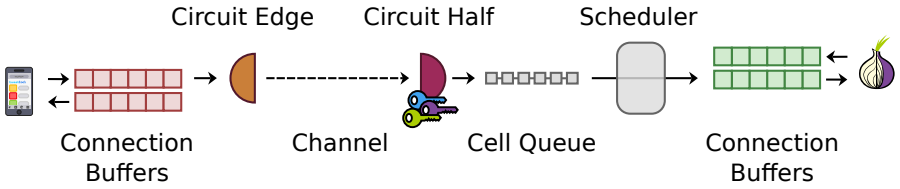
While much of the relay's global state can be handled asynchronously by the main thread's connection controller, connections may need to access some global state synchronously. For example, relays are often configured with global bandwidth limits as discussed in Section 4.2.1. Each thread must make sure that they combined do not exceed any of these limits. Before reading



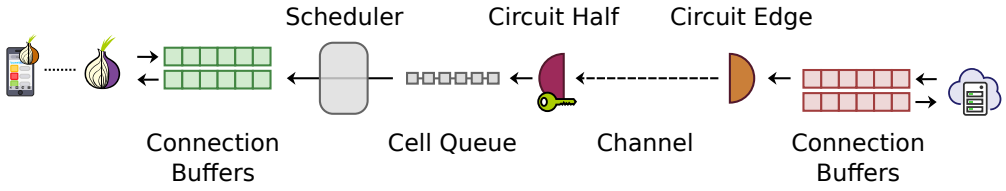

(a) Relay cells moving forward along the circuit at a relay.



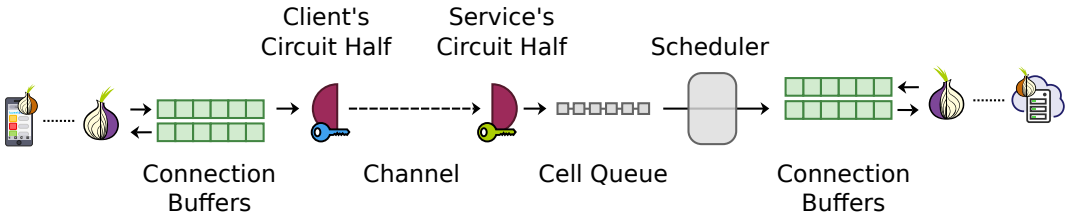

(b) Relay cells moving backward along the circuit at a relay.




(c) Relay cells moving forward along the circuit at the circuit's origin.




(d) Relay cells moving backward along the circuit at the stream's exit.




(e) Relay cells (sent by the client) switch circuits at a rendezvous point.

Figure 5.8: Examples of the circuit objects and key locations in various circumstances. Each purple onion represents a relay. The Tor logo by The Tor Project, Inc. is licensed under CC BY 3.0 US.

or writing network data, Tor's current design will check the remaining capacity of the token buckets, attempt to read or write at most that much data, and then drain the buckets by only how much data was actually read or written. In this multi-threaded architecture if these token buckets were simply made atomic, it would lead to a time-of-check to time-of-use race condition. One approach using atomics could be to temporarily protect some amount of the token bucket and later commit only the amount used, although it is possible that connections would over-protect the token buckets and prevent other connections from reading or writing for a short time. Another approach could check the amount of data available to read using the `ioctl` system call, atomically drain the token bucket by this amount, and then read at most that many bytes from the socket, but this does not translate to the setting of writing network data. A simpler approach could give each thread its own token bucket with a fraction of the total capacity. For example a relay supporting 100 Mbps with 4 connection threads could give each thread a token bucket with a re-fill rate of 25 Mbps, but this requires good load-balancing across threads to make full use of the 100 Mbps.

Memory management is another component that requires a global view of the relay. Tor re-calculates its memory usage and runs its out-of-memory handler if needed for every relayed cell. This is not possible in our multi-threaded architecture, but we can still periodically check the memory usage. Instead, each connection manager could periodically re-calculate the memory usage of each circuit and send them in a message to the connection controller. Each time the controller receives updated memory usage information, it can store the updated data and choose bad circuits from its stored set of all circuits. The identifiers for each circuit can then be sent in a message to their connection managers instructing them to close the circuits.

### 5.3.6 Limitations

This architecture does not attempt to describe every aspect of a Tor relay. For example, we do not touch on a relay's optional directory server or hidden service directory, introduction points, control connections, and more. Instead we focus on the routing tasks of a relay. That said, we do describe how some additional components fit into the architecture such as denial-of-service prevention, out-of-memory handling, and bandwidth accounting. In addition, due to the prevalence of global state in Tor and the widespread use of connection and circuit objects, implementing the architecture may require significant development effort. Asynchronous message passing can also significantly increase the program's complexity when one thread sending a message must maintain some state in anticipation of a reply. For example, when a circuit half receives a Relay{Extend2} cell, it sends a message to the connection controller (as shown in Figure 5.6) and later receives a reply. This circuit half must remember some state about the context of the message (in this case the circuit handshake data) and be able to resume where it last left off. This

requires additional code to store and load the state, and makes it more difficult for developers to follow the program's control flow. This is a design problem that is simpler in languages that support continuations, or coroutines in general.

## 5.4 Summary

Our multi-threaded relay architecture fulfills our three goals outlined at the start of the chapter:

- Preserve compatibility with the existing live Tor network — we do not require changes to any of Tor's network protocols.
- Parallelize the end-to-end flow of unrecognized Relay cells throughout the relay — we use connection threads with circuit halves and channels for communication across threads.
- Maintain some similarity with Tor's existing relay architecture in order to increase the amount of code that can be re-used in its implementation — we keep many of the same design concepts such as the scheduler, circuit cell queues, and the eventloop.

As the processing of Relay{Data} cells and unrecognized Relay cells does not require interaction with the main thread, we expect this architecture to scale well across threads when relaying large amounts of data. Tasks such as circuit extension, accepting incoming connections, processing consensus documents, and uploading the relay descriptor do rely on the main thread, which may become a bottleneck for the relay, but as the end-to-end processing of Relay{Data} and unrecognized Relay cells is completely independent, the throughput of existing connections and circuits should not be limited by this bottleneck. The bottleneck will affect the rate at which the relay can accept new connections and circuits, but as these main thread tasks are less CPU intensive than the task of relaying cells, we expect that the relay will be able to scale over many connection threads before becoming limited by the main thread. There may also be opportunity to further move some of these main thread tasks to other threads. To understand how the relay scales in practice, we experiment with an implementation of key portions of our multi-threaded architecture in the next chapter.

# Chapter 6

# Implementation and Evaluation

To understand the performance of our multi-threaded architecture, we developed a relay implementation based on the existing Tor implementation (which we refer to as *vanilla Tor*). This chapter describes our relay implementation, how it compares to the multi-threaded architecure in Chapter 5, and how it compares to vanilla Tor. While profiling a Tor relay running on an experimental network, we found that the relay was spending much of its time performing TLS encryption and decryption. For this reason we focused on parallelizing the socket reading/writing, TLS encryption/decryption, and unpacking of cells. This covers a subset of the design from Chapter 5 and demonstrates a lower bound for the performance improvement that could be obtained from implementing the entire multi-threaded architecture. We compare the throughput of this new multi-threaded relay implementation to the vanilla Tor implementation using experiments designed to saturate a relay with network traffic.

Our experiments evaluate the performance of the multi-threaded relay on two different systems: a high-performance server with plenty of memory and Intel CPU cores, as well as a low-performance but inexpensive Raspberry Pi 3B+ single-board computer with four ARM CPU cores. While high-performance relays will typically be running on powerful hardware, multi-threading has the potential to improve low-performance relays as well. Relays with low throughput can harm the experience for users due to the resulting circuits with lower throughput and more unpredictable latency, so improving the performance of these relays is important for providing a more stable experience for users. In addition, due to their relatively low cost at around $50, these single-board computers reduce the barrier of entry for people wanting to contribute their own bandwidth to the network. Inexpensive hardware is also useful for bridge relay operators who may want to provide bandwidth to a small number of friends with censored Internet, but who do not have the means to purchase more traditional server or computer equipment.

## 6.1 Multi-threaded Relay Implementation

The multi-threaded relay implementation described in this section was based on Tor 0.4.2.6 (released January 2020). Due to many of the challenges described in Section 5.1, it was not feasible to implement the entire multi-threaded architecture. Instead a subset of the architecture was implemented with much of the relay's cell processing remaining in the main thread. Even with a smaller scope, most of the code that was moved to the connection threads had to be re-written to make it thread-safe and to avoid use of global state. Rather than passing cells directly between connection threads, cells are passed from a connection thread to the main thread for processing, and later from the main thread to another connection thread. The end-to-end cell processing is shown in Figure 6.1. This design required changes to several parts of Tor to support this multi-threading.

### 6.1.1 Connection Threads and Per-thread Eventloops

While the libevent library does not support running a single eventloop across multiple threads, it does support individual per-thread eventloops. The event base that manages the state of its eventloop is also thread-safe, meaning events can be added to an eventloop running in one thread from other threads. We modified Tor's threading and eventloop system to support multiple eventloops in different threads. This involved improving Tor's threading support to make threads joinable rather than detached, and properly cleaning up threads during the relay's shutdown process. We also improved Tor's threadpool and work queue system to handle tasks originating in any thread. As described in Section 5.3, the relay's main thread starts multiple connection threads, each with its own eventloop. The number of connection threads launched by the relay can be modified by the user in the relay's configuration options.

### 6.1.2 Channels and Inter-thread Message Passing

To support message passing between threads, we built a directional channel based on a message queue design. This message queue channel uses an event subscription model where an event listener subscribes to events published by an event source. Published messages are transferred to the event listener, added to the listener's message queue (a linked list), and later processed during the listener's corresponding eventloop. Listeners are attached to their owning thread's eventloop and can signal to the eventloop when it receives new messages. On receiving this signal, the eventloop allows the listener to iteratively process each message in its queue. Listeners use an eventfd file descriptor provided by libevent's eventloop to wake up the eventloop in another

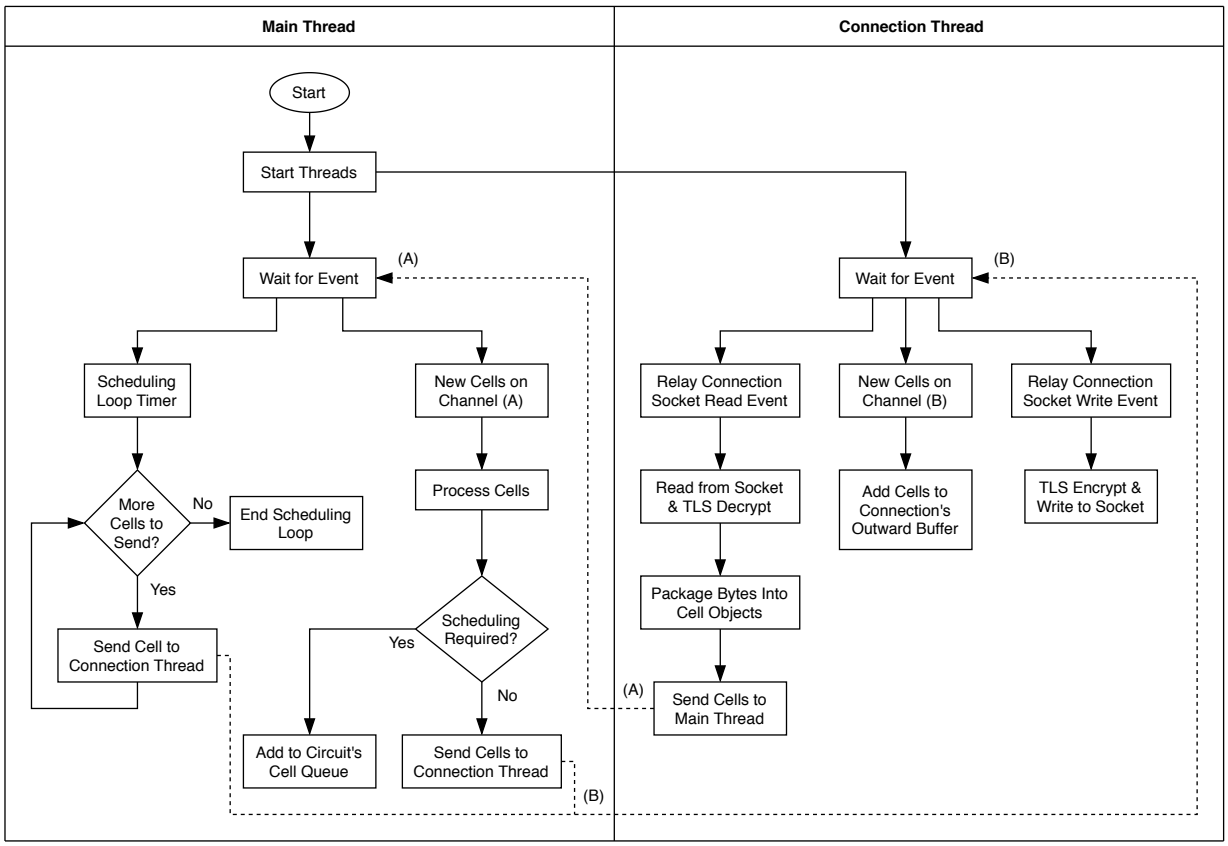



Figure 6.1: Overview of the relay connection cell processing and message passing in the multi-threaded relay implementation. There can be multiple connection threads, each handling a number of relay connections. Each relay connection has its own set of channels (A) and (B). Leaf nodes implicitly return to the eventloop.

thread. As Tor relays require many TCP connections, they are often constrained by the number of file descriptors allowed by the system. Using a per-eventloop file descriptor for notifications allows it to scale to many event listeners without using additional file descriptors. The ownership of any data contained within a message is transferred from the event source to the listener. In order to improve performance, messages can be published to an event listener silently, meaning that the event listener adds the message to its queue but does not notify its eventloop. This is useful when an event source wishes to publish several messages and does not want the listener to notify its eventloop for each one (an action that requires locking and writing to the file descriptor). Instead it can notify the eventloop only after publishing the last message in a batch. It also supports updateable messages, where if a listener receives consecutive messages of the same type, the listener can merge them into a single message. This can improve performance when a specific message type acts as a signal rather than a distinct message.

### 6.1.3 Memory Allocators

The most common messages passed between threads in our multi-threaded relay implementation are cell-related messages. This requires each cell to be stored on the heap as it is passed between threads and leads to a large number of memory allocations every second in each thread. While the standard Glibc memory allocator is designed to handle simultaneous memory allocations in different threads using arenas and thread-local caches, we still found that the allocator hindered performance in our multi-threaded relay due to lock contention. Instead we also build Tor with gperftools' tcmalloc and the jemalloc memory allocators which are designed for performance in multi-threaded applications.[1] These libraries require no changes to the Tor code and can be either linked during the build process or dynamically loaded using symbol interposition. We do not aim to understand the performance differences between these allocators, but rather to illustrate any possible performance differences to aid in future development. We compare the difference in relay throughput using Glibc's memory allocator and these two other allocators in Section 6.3.2. While implementing object pools may be helpful for reducing the number of required memory allocations, we instead leave the management of memory pools up to the memory allocator for simplicity.

[1]The gperftools tcmalloc library is a community maintained version of Google's original TCMalloc library, which is different from Google's current public TCMalloc library.

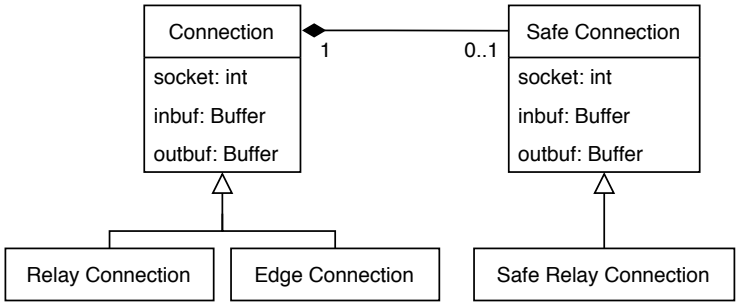



Figure 6.2: The thread-safe connection type and its relationship to regular connections. If the connection type has a corresponding "safe connection" object, data such as the socket and connection buffers are stored in this safe connection object instead. As not every connection type has a corresponding thread-safe connection type yet, this data is stored in one of two locations depending on the connection type.

### 6.1.4 Parallelizing Relay Connections

Each TCP connection in Tor is represented by a connection object. To move the processing of network data to the new connection threads, the connection objects must be thread-safe and must not access any other objects or global state. Tor has different specialized connection types (for example relay, edge, directory, and controller connections) which all inherit[2] properties of a common base connection type. This base connection type holds general connection data such as the socket, connection buffers, and pointers to linked connections. If we were to support thread-safety for a specific connection type, such as the relay connection type, it would require the base connection type to also be made thread-safe. Modifying the base connection type would then require us to modify all of Tor's other connection types. This would force significant changes to many other parts of Tor and would have a cascading effect throughout the Tor code. Instead, we take the approach of moving small logical pieces out of the connection objects and into new thread-safe connection objects. For lack of a better name, we call these thread-safe types "safe" types. The "safe connection" type has a 1:1 relationship with the connection type. Just like the relay connection type inherits from the connection type, our "safe relay connection" type inherits from the "safe connection" type as shown in Figure 6.2. While this approach is not ideal from a design perspective and leads to some code duplication, the goal would be for these thread-safe types to slowly absorb the functionality of the original connection types, until finally the original connection objects could be removed entirely.

Using this approach of moving small logical pieces out of the connection objects, we moved

[2]The term "inherits" here is used in a loosely object-oriented sense.

most of the core networking functionality of the relay connection into the new thread-safe objects. Much of this code was re-written to avoid accessing other objects or global state, but not all of the functionality was moved over to the thread-safe objects. For example, the safe relay connection object supports connection-specific token buckets, but not global token buckets. When a new relay connection object is created, the mainloop attaches its safe connection object's socket read/write events to a connection thread's eventloop. The connection thread is chosen in a round-robin manner to evenly distribute these safe connection objects across connection threads. When there is data to read or write on a relay connection's socket, these safe relay connection objects process the data. Specifically, these safe relay connections read from the socket, TLS decrypt the data, parse the data into cells, and add them to a channel, which sends them to the main thread's eventloop as shown previously in Figure 6.1. The mainloop processes these cells, schedules them, and instead of writing the cells directly to the socket in the scheduling loop like the current Tor architecture, it adds the cells to a channel. These cells are received by the safe relay connection objects in one of the connection threads and when the socket becomes writeable, TLS encrypts the data and sends it. An important note is that this breaks the primary purpose of the KIST scheduler, which is to write data immediately to the socket. Since cells are scheduled in the main thread and then sent over a channel to another thread before being written to the socket, there is an additional queuing delay between when the cells pass through the scheduler and when they are written to the socket. This limitation exists only in our implementation and not in the multi-threaded architecture.

## 6.2 Experimental Design

Our experiments were designed to evaluate the throughput of our multi-threaded relay as a middle relay (the most common type of relay on the network) by measuring the maximum sustained throughput of the relay while under heavy CPU load. These experiments were not designed to model real-world Tor network traffic, but rather to provide a simple model for comparing specific aspects of the relay performance. In order to saturate the relay, Tor clients on an experimental Tor network build hundreds of circuits through this target relay and simultaneously send data through them to a server. We measure the throughput and CPU usage of this relay, as well as timing information about the data that passes through these circuits. By changing the configuration and version of the target relay, we measure the performance of a middle relay with different relay modifications and compare it to the standard relay implementation. Each client and relay in these experiments are based on Tor 0.4.2.6, with custom patches as described later in this section.

We used Chutney to initialize and bootstrap a Tor network for the experiment. We extended

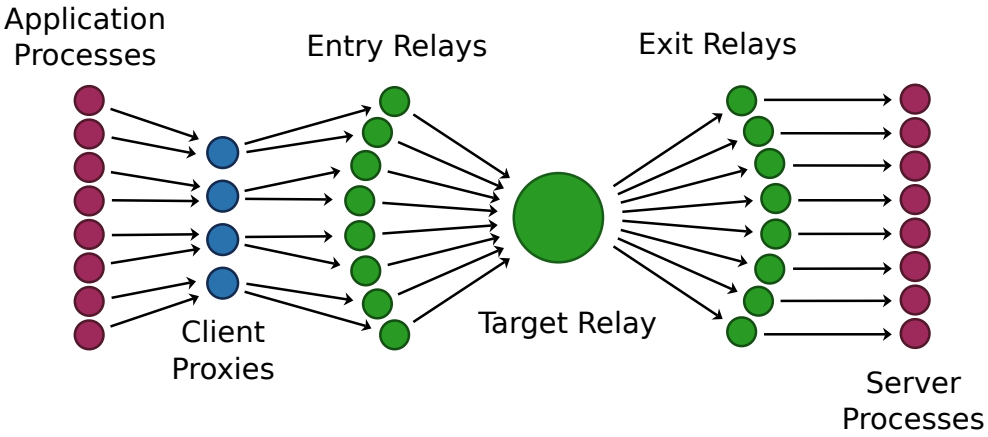



Figure 6.3: Overview of the experiment network and the paths of circuits. Each circuit is routed through the target relay.

Chutney to support additional debugging features such as profiling relays with Gperftools, debugging memory and threading issues with Valgrind and Helgrind, setting NUMA configurations, and more. We also extended Chutney to launch Tor relays on multiple servers. We chose Chutney for our experiments as it was easy to extend to support multiple servers and allowed us to profile and benchmark the relay's CPU performance without any emulation. The network consists of five types of Tor nodes: clients, directory authorities, non-exit relays, exit relays, and a single target relay (the relay we wish to test the performance of). Each circuit we create uses a non-exit relay as the first hop, the target relay as the middle hop, and an exit relay as the final hop. This effectively partitions the network into groups: entry relays, exit relays, and the target relay, as shown in Figure 6.3.

We run all of the relay, proxy, client, and server processes on a single computer, which we will call the *control server*, except for the target relay, which runs on a separate computer. In order to test the relay performance on a variety of hardware, we run experiments with the target relay running on six cores of a fast Intel Xeon server, which we will call the *Intel server*, and also on a slower but much less expensive quad-core Raspberry Pi 3B+ single-board computer, which we will call the *Raspberry Pi server*. Table 6.1 shows the configuration of each system. The control server uses 64 virtual CPU cores, which is significantly more cores than are used on either of the target servers. The control server and Intel server are connected directly by a 10 Gbps connection. The control server and Raspberry Pi server are situated in different buildings and connected by a 1 Gbps connection.[3] As proxies and relays save network documents and other data, all Tor

[3]The Raspberry Pi's Ethernet adapter is attached over USB 2.0, limiting the effective bi-directional network bandwidth to ~150 Mbps.

Table 6.1: System information about the three servers used in our experiments. For Intel CPUs, core counts represent virtual cores (including hyperthreading).

| Name | CPU | RAM | NIC |
|---|---|---|---|
| Intel Server | 6 cores of a 16-core 2.40 GHz Intel Xeon | 128 GiB | 10 Gbps |
| Raspberry Pi 3B+ | Quad-core 1.4 GHz ARM Cortex-A53 | 1 GiB | 1 Gbps over USB 2.0 |
| Control Server | 4×16-core 2.40 GHz Intel Xeon | 512 GiB | 10 Gbps and 1 Gbps |

processes running on the control server write their data to a temporary filesystem in memory.

We run the target relay in a few different configurations for performance comparisons. As a baseline we use the standard Tor implementation (vanilla Tor) with a small patch to log its throughput. Specifically, it records the per-thread inbound and outbound bytes on relay connections every 500 ms. The data is recorded in memory and written to the disk only when shutting down the relay in order to limit any effect of this logging on performance. We also use our multi-threaded relay implementation with the above patch as the target relay and run it with varying numbers of relay connection threads. Finally for each configuration described above, we run them with the jemalloc memory allocator library, gperftools's tcmalloc library and the default GNU memory allocator of glibc.[4]

We configured the target relays for all experiments to use three CPU worker threads (see Section 4.3).[5] These threads are different from the connection threads used by our multi-threaded Tor code. When running on the Intel server, we assign the target relay process a NUMA policy with 6 virtual CPU cores (3 physical cores and their paired hyperthreading cores) and memory on the same NUMA node. We also disabled the directory cache functionality on the target relays for both servers. In order to bootstrap our experimental Tor network quickly, we use a consensus voting interval of 40 seconds compared to 1 hour on the real Tor network. A side effect of this configuration is that relays acting as directory caches need to process new consensus documents every 40 seconds, which is much more often than on the real Tor network. As this put a small but unrealistic burden on our target relay, we disabled this directory cache behaviour. To monitor the target relay performance, we log CPU usage data from the kernel every 200 ms using the various status files in the Linux proc pseudo-filesystem. For each thread of the relay, we calculate its

[4]We use gperftools version 2.7 and jemalloc version 5.2.1 on both servers, and glibc versions 2.19 on the Intel server and 2.28 on the Raspberry Pi server.

[5]Tor uses one more CPU worker thread than what is specified in its configuration file, so we configured Tor to use 2 CPU worker threads (using the '`NumCPUs 2`' option) so that it would actually use 3 CPU worker threads.

Table 6.2: Experiment configurations for the two servers.

| Name | Client Proxies | Entry Relays | Exit Relays | Streams per Client | Data per Stream | Total Circuits |
|---|---|---|---|---|---|---|
| Intel Server | 150 | 300 | 300 | 10 | 10 MiB | 1500 |
| Raspberry Pi | 100 | 300 | 300 | 6 | 5 MiB | 600 |

CPU usage as the fraction of scheduler clock ticks that the thread ran in user or kernel mode, and the total clock ticks of the CPU core(s) that the thread ran on. If the thread switched to a different CPU core at some point within a time step, we calculate the CPU usage for both cores independently and use the mean.

Under standard operation when applications connect to Tor's SOCKS proxy, Tor assigns these application streams to existing circuits if available. In order to control the circuit paths that are used for our application streams, we use the Python Stem library [Joh20] to build circuits with specific paths and attach our streams directly to those circuits. Stem allows us to interact with the control port of each Tor client in order to issue it commands and listen for events. We make each Tor client build some number of circuits (for example 10), and we choose the circuit's entry and exit relays in a round-robin manner in order to balance the load among these relays (see Figure 6.3). Table 6.2 shows the various network parameters for each server. To prevent Tor from managing the circuits we create (for example closing circuits that are unused but that we plan to use in the future), we set the "controller" purpose for these circuits. Due to a Tor bug we found with these "controller" circuits, all of the clients and relays in our network (other than the target relays) have a small patch to fix support for these circuits. Specifically, we correct the destination address that is sent in the Relay{Begin} cell.

Once the circuits have been built on all clients, we start application processes, which connect to the SOCKS port on our clients. Each of these application streams is assigned to a different circuit so that there is at most one stream per circuit. These streams connect through the Tor network to a server, which forks and handles each stream in its own process. Each client/server pair performs a custom handshake so that the client can identify itself and inform the server how much data it will send (see "Data per Stream" in Table 6.2), and then waits for a signal. Once all of the application streams have been attached to circuits, a signal is sent to all client processes and they begin simultaneously sending data through the Tor network to the server. All of these circuits pass through the target relay, and since the control server uses significantly more CPU cores than the target relay to process the same amount of network data, the target relay becomes the bottleneck limiting the application stream throughput. As the server processes receive data they record the time and number of bytes read from the socket. Eventually all server processes receive their data and the experiment ends. While most of the experiment code is written in

Python, the task of sending data from the application process and receiving data in the server process is run from C using the Python-C API in order to maximize the performance.

## 6.3 Results

Following the experimental setup in Section 6.2, we repeated each experiment configuration 10 times. Tor's current relay implementation is identified by "vanilla Tor", and the multi-threaded relay implementation is often shortened to "MT Tor". All of the results in this section ignore Tor's CPU-worker threads, which have an insignificant effect on the CPU performance and throughput of the relay in these experiments. For results with the multi-threaded relay implementation where there are no connection threads, all code is run in the main thread much like vanilla Tor. All results are shown using the jemalloc allocator due to its all-around good performance, but we compare memory allocators in Section 6.3.2.

### 6.3.1 Throughput and CPU Usage

Each experiment logs the number bytes sent and received on relay connections every 500 ms. From this data we find the maximum sustained throughput over a 30 second period, which we calculate as the maximum of a 30 second moving average. This represents the maximum throughput

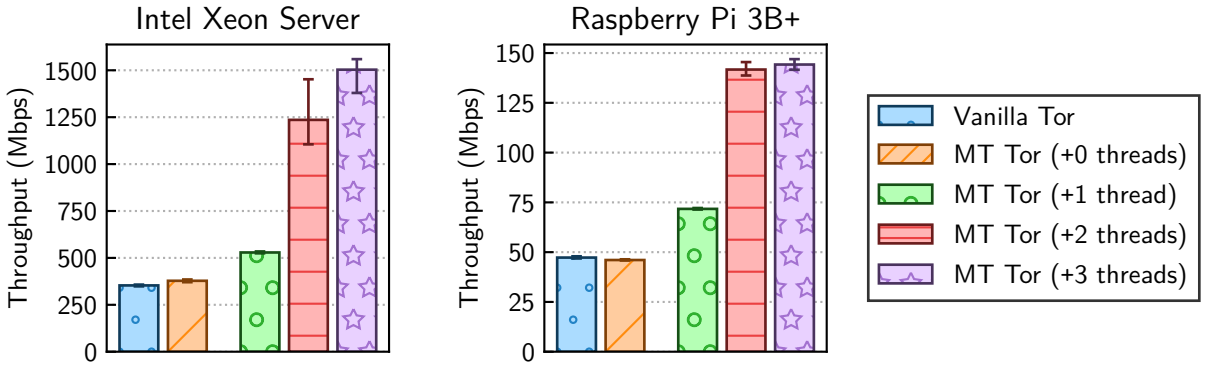



Figure 6.4: Maximum sustained throughput over 30 seconds of an original Tor relay and a multi-threaded Tor relay with varying numbers of connection threads. Error bars show the minimum and maximum values measured over 10 repetitions. All relay versions use the jemalloc memory allocator.

that the relay was able to sustain for at least 30 seconds and is similar to the "observed bandwidth" metric advertised by relays in their server descriptors [Tor20b]. This calculation discards the first 30 seconds of throughput data to allow the relay's performance to reach a steady state. As shown in Figure 6.4, we see a significant throughput improvement with our multi-threaded relay when using several threads. When running with three connection threads, the maximum sustained throughput is about 4.3 times greater than vanilla Tor on the Intel server, and about 3.1 times greater on the Raspberry Pi server. As much of the parallelized operations are TLS reading/writing, which use AES in CBC mode, it might be expected that the Raspberry Pi would scale better across CPU cores due to its lack of hardware-accelerated AES support (ARMv8's Cryptography Extension), but this does not appear to be the case. On the other hand, the throughput of the multi-threaded relay on the Raspberry Pi is very close to the limit of its USB-attached network adapter (about 150 Mbps). When running with a single thread, the multi-threaded relay implementation performs very similar to vanilla Tor even with the multi-threaded relay's additional overhead from message passing.

When comparing the performance of our relay implementation with varying numbers of threads, the single-threaded performance is not directly comparable to the multi-threaded performance since the single-threaded case is handled differently. As mentioned earlier in this section, when running our relay implementation with only a single thread (and no connection threads), the connection processing code is run in the main thread and shares the CPU core's capacity with other processing performed by the relay in the same thread. This is useful for comparing this special case to vanilla Tor. In all of the other cases where at least one connection thread is used, the connection processing code is run entirely in these connection threads and not the main thread. In addition, the single-threaded case does not have the overhead of writing to another thread's eventfd file descriptor to wake up the thread for each message message sent (or for groups of messages using the silent delivery feature described in Section 6.1.2). On the Intel server, we can also see that the throughput more than doubled when increasing from one connection thread to two. In the case of two connection threads, each thread was able to process data at a higher throughput than when only a single conection thread was used. The reason for this is not clear, but may have been due to several factors including the assignment of threads to different CPU cores (including hyperthreading cores) or thermal throttling. We also found that the relay was buffering more network data when using multiple connection threads, which, along with the observation of the main thread being the bottleneck as shown in Figure 6.5, suggests bufferbloat along the channels. While bufferbloat can negatively affect latency, larger buffers may allow connection threads to read and write larger amounts of data to the socket at a time, reducing the number of system calls used and increasing throughput.

If we look at the relay's performance over the first 60 seconds of the experiment in Figures 6.5 and 6.6, we get a much more detailed picture of how the throughput relates to the CPU usage.

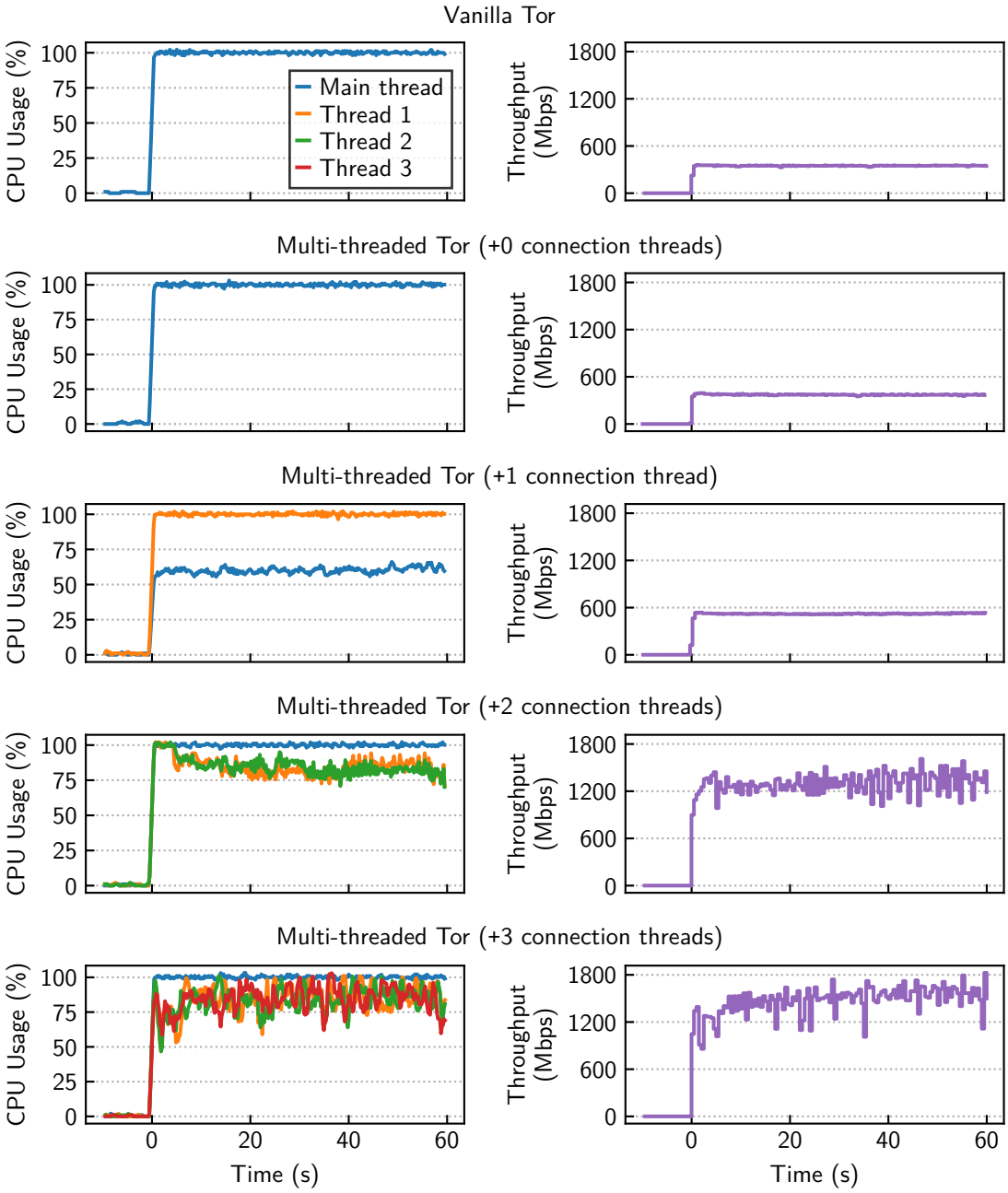



Figure 6.5: The CPU performance of each connection thread and the relay's throughput with varying numbers of connection threads for a single run of the experiment on the **Intel server** using the jemalloc memory allocator. The clients begin sending data through the network at time 0. CPU usage is averaged using a 1-second sliding window.

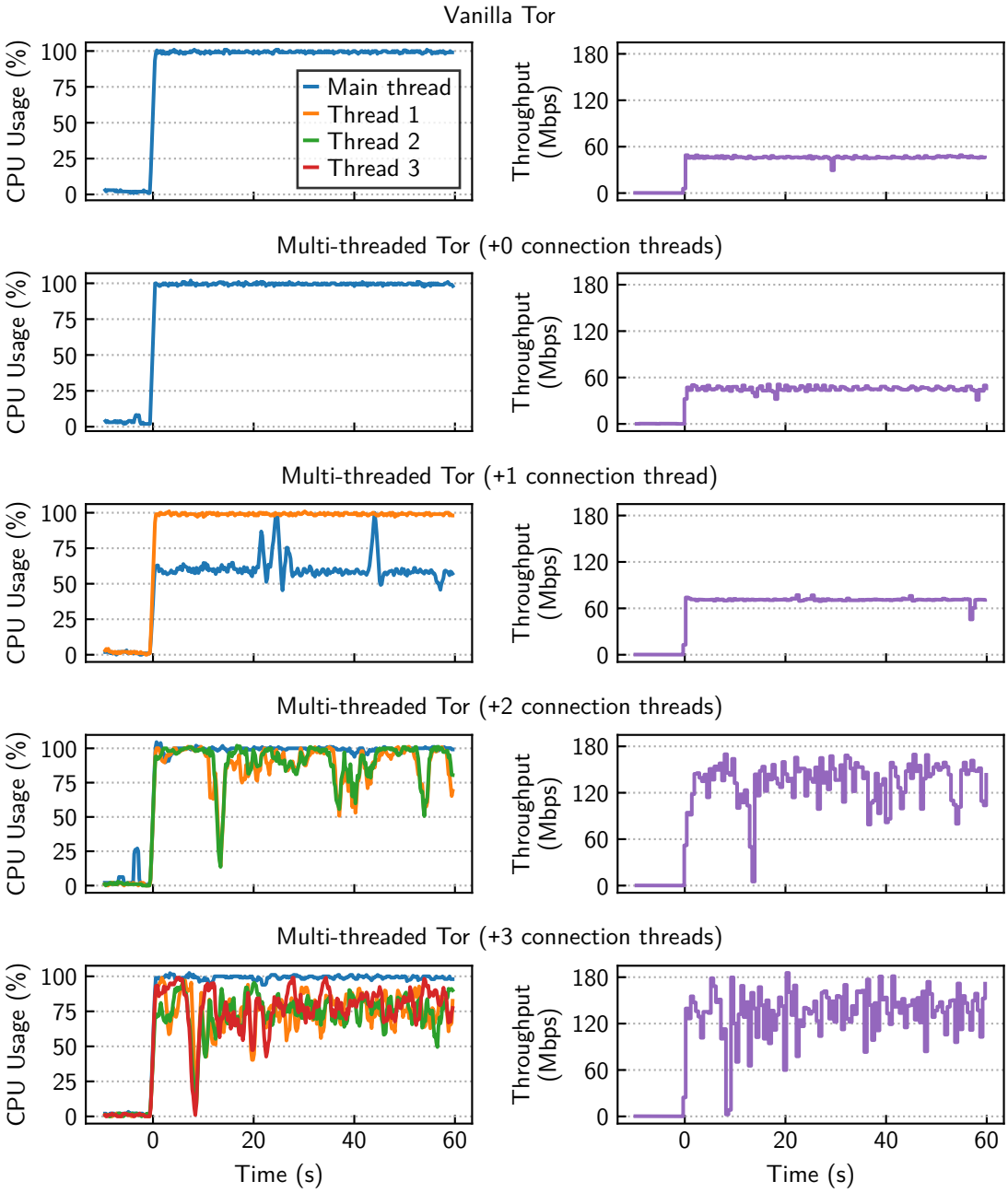



Figure 6.6: The CPU performance of each connection thread and the relay's throughput with varying numbers of connection threads for a single run of the experiment on the **Raspberry Pi** using the jemalloc memory allocator. The clients begin sending data through the network at time 0. CPU usage is averaged using a 1-second sliding window.

Each graph shows the relay's throughput, as well as the CPU usage of the relay's main thread and connection threads. The relay's threadpool CPU-worker threads are not shown as they are remain at 0% usage. Time 0 represents the start of the experiment (when streams begin sending data) and before that is when the experiment is being initialized (creating circuits and connecting streams). Once the experiment starts, the CPU usage climbs quickly, with one of the threads becoming a performance bottleneck. When using more than one connection thread the main thread becomes the bottleneck, limiting the relay's scaling performance when using more than two connection threads. The relay's throughput closely follows the CPU usage of the relay's connection threads, but this is expected since the throughput is calculated from the number of bytes sent and received by these connection threads. The relay's throughput remains mostly stable throughout the experiment, but has some variability when using several connection threads.

### 6.3.2 Memory Allocator

We repeated each experiment configuration with three memory allocators: the standard glibc allocator, the jemalloc allocator, and gperftools' tcmalloc library. The three allocators performed similarly when the relay was run with either 0 or 1 connection threads, but performed quite differently when run with more connection threads as shown in Figure 6.7. Since the multi-threaded relay implementation uses frequent memory allocations, memory allocators designed for multi-threaded applications like tcmalloc and jemalloc allow the relays to perform with significantly

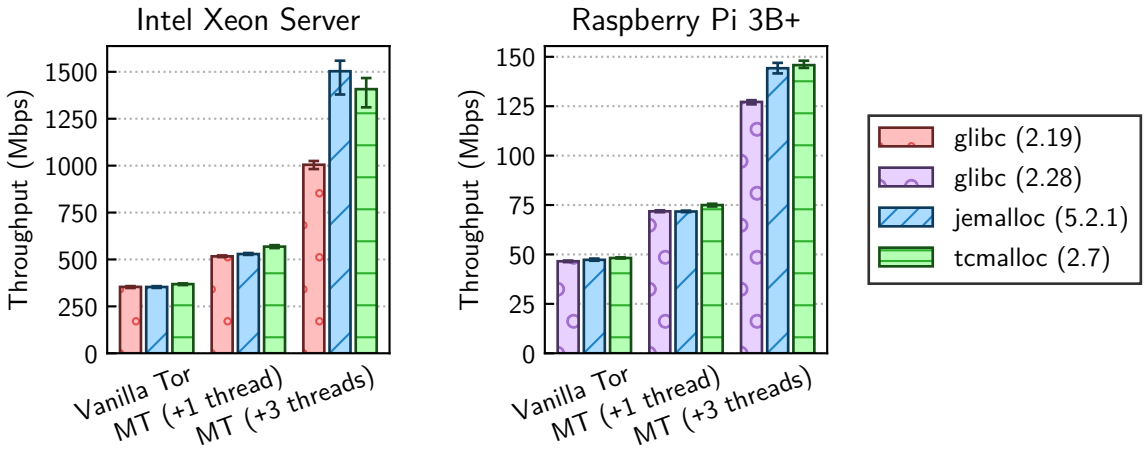


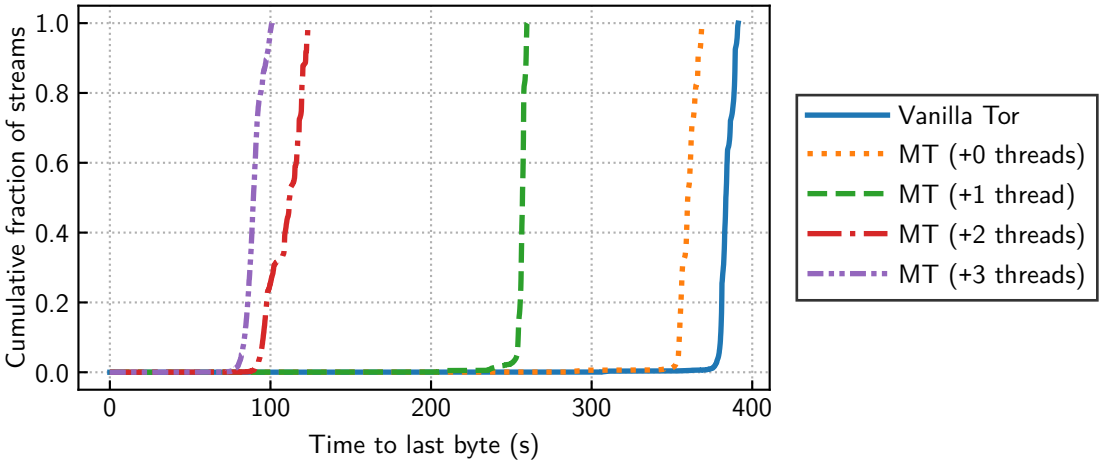

Figure 6.7: Maximum sustained throughput over 30 seconds of Tor with the three tested memory allocators. Error bars show the minimum and maximum values measured over 10 runs.

higher throughput when using more threads. This performance difference was more significant on the Intel server than the Raspberry Pi. From these results its not clear which of the tcmalloc or jemalloc libraries perform better, but both perform much better than glibc. It is worth noting that while we use recent versions of tcmalloc and jemalloc, the Intel server runs a much older version of glibc. This version of glibc does include features designed for better multi-threaded allocator performance such as arenas and thread-local caches, but does not have all of the latest improvements. Unlike tcmalloc and jemalloc which can be easily added to a program using symbol interposition, glibc does not provide a simple method of replacing the program's memory allocator with a newer version, and instead requires the program and all of its dependencies to be rebuilt with this new glibc version (and its corresponding loader). While the throughput performance difference is clear, we do not measure other metrics that could be important for choosing the best allocator for our implementation. For example, we do not measure the memory usage of the process, which may be greater for allocators that perform more aggressive caching. Understanding the performance of these memory allocators is not a goal of our work, but demonstrates a significant performance consideration when developing a multi-threaded relay.

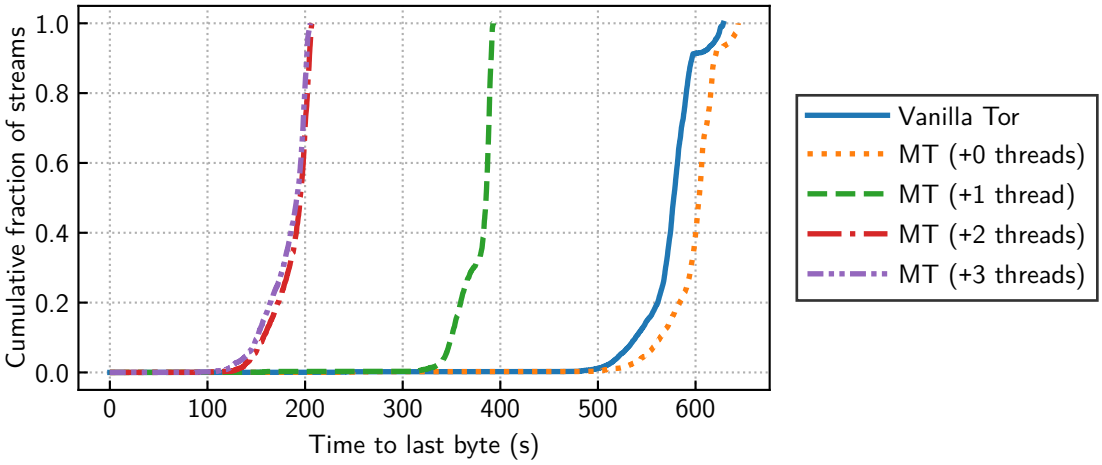
### 6.3.3 Stream Performance

The previous results have looked at the performance of the relay, but the performance of the individual application streams is also important as it influences the end user's experience. Two metrics that we measure are the streams' upload times (time to last byte) and their initial upload latencies (time to first byte). Each stream transfers some amount of data (documented in Table 6.2) from the client to the server, so the time to last byte represents the time from when the first byte was sent by the client to the time when the client's last byte was received by the server. Similarly, the time to first byte represents the time from when the first byte was sent by the client to the time that same byte was received by the server. These graphs each show the cumulative distribution for each stream over all 10 repetitions of the experiment.

As all streams are the same, the relay should not prioritize any stream, connection, or circuit over another. In this case it is expected that most streams will complete around the same time. Figure 6.8 shows that this is the case for both the vanilla Tor relay and the multi-threaded relay, demonstrating that our multi-threaded relay does not seem to negatively affect the relay's circuit prioritization mechanisms in our experiments. When using no connection threads, the multi-threaded relay performed similar to vanilla Tor. The streams completed slightly earlier when running on the Intel server, but slightly later on the Raspberry Pi. These upload times agree with the relative relay throughputs in Figure 6.4, where the multi-threaded relay (with no connection threads) had slightly greater throughput than vanilla Tor on the Intel server, but slightly worse throughput on the Raspberry Pi. These results show that the multi-threaded relay does not have

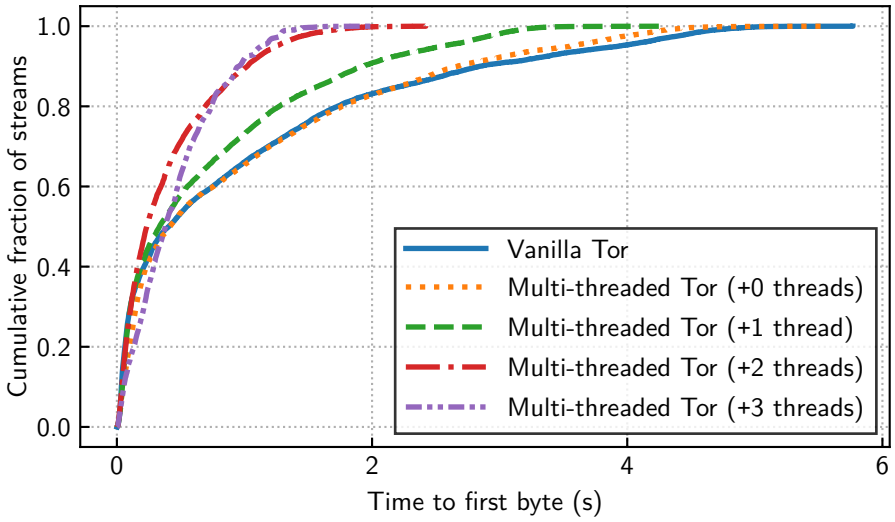



(a) Intel server using the jemalloc memory allocator.

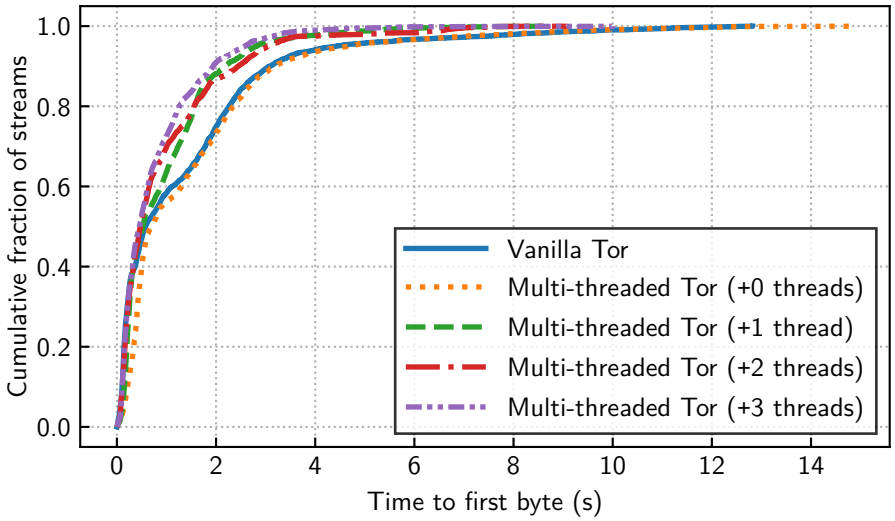



(b) Raspberry Pi using the jemalloc memory allocator.

Figure 6.8: The time to last byte (upload time) for each stream combined for all 10 repetitions.




(a) Intel server using the jemalloc memory allocator.




(b) Raspberry Pi using the jemalloc memory allocator.

Figure 6.9: The time to first byte for each stream combined for all 10 repetitions.

a large performance overhead compared to vanilla Tor when using the same number of threads. When scaling the multi-threaded relay to additional connection threads, streams completed much quicker due to the higher relay bandwidth, which corresponds to much faster data transfer times for clients.

The multi-threaded relay implementation uses more buffers than vanilla Tor due to its channels, leading to a possibility of more bufferbloat and longer circuit latencies. Since all streams are started at the same time, the relay quickly becomes saturated as shown by the CPU load in Figures 6.5 and 6.6. The initial upload latency of each stream corresponds to the latency during this time. The lower the latencies, the better the relay is handling the sudden load increase. Figure 6.9 shows that the multi-threaded relays generally had lower latencies than vanilla Tor. Just like their throughput was similar, vanilla Tor and the multi-threaded relay with no connection threads performed extremely similarly.

## 6.4 Discussion

The multi-threaded relay implementation parallelizes only the relay connection networking component of the architecture described in Chapter 5, but already demonstrates a significant performance improvement of a multi-threaded relay on both limited and high-performance hardware. It is likely that implementing more components of the architecture would lead to better scaling over more CPU cores. As our implementation does not cover our entire multi-threaded architecture, the implementation is a fusion of our architecture and Tor's current architecture. This means that our implementation does not take full advantage of either architecture. It does not achieve the fully parallel end-to-end processing of cells from the multi-threaded architecture, and we lose the performance characteristics of the KIST scheduler from both architectures. Our implementation also has much more cell queuing than either architecture alone requires. Our relay implementation should not be viewed as strictly better than Tor's current implementation, but rather as a demonstration of the performance improvements possible with a relay implementation based soley on our architecure.

The experimental design differs from the real-world Tor network in several ways, such as the network size, client application streams, network topology, and relay workload. Unlike typical web browsing where a single circuit multiplexes multiple HTTP streams, clients in our experiments perform a bulk data transfer in a single stream per circuit. While these streams may not be representative of typical Tor usage, the target relay cannot see streams within a circuit and handles all unrecognized Relay cells in the same manner regardless of the circuit's composition. Another difference is the direction of the flow of data. Typical web browsing downloads much more data than it uploads (data mostly flows backward along circuits), but our experiments

have clients uploading data (data flows forward along circuits). This difference in data direction should not negatively affect the results since when relaying unrecognized Relay cells, the target relay performs the same cell processing regardless of the direction of cells. Similar to the stream direction is the overall flow of data over the network. From Figure 6.3 it is clear that data flows over relay connections almost completely in a single direction (from left to right in the diagram). In reality relay connections may have data flowing in both directions along a relay connection due to circuits with origins on both sides of the connection. Network size, another difference, is commonly difficult to model realistically due to the large number of relays and Tor users. For example a typical relay (as measured by an exit relay on the live Tor network) might have thousands of relay connections and tens of thousands of circuits. Our experiments are much smaller with only around 600 relay connections and 1500 circuits (with the Intel server). Another difference is the workload that the target relay performs. On the real-world Tor network, relays continuously accept and make new connections, and extend circuits. For example, using Tor's debug logging feature with an exit relay, the relay reported receiving around one incoming relay connection per second. The experiments performed in this chapter perform these actions ahead of time. While these are not very expensive operations, they do add some CPU load to the relay that is not modelled in these experiments.

The experiments performed in this chapter do not attempt to model the real-world Tor network, and it is possible that in practice the relay may not scale as well as it does in these experiments. While using more realistic models might provide more accurate results, the purpose was not to propose that this implementation should be used on the Tor network. Rather the implementation is a step toward a more thorough implementation based on the architecture in Chapter 5, and the goal of these experiments were to show that a multi-threaded relay can be implemented without significant performance issues. There are also other performance metrics that may be useful to measure, but that we did not record, such as relay memory usage and queueing delays.

The most significant technical challenges in developing our implementation were due to the global state and ownership obstacles described in Section 5.1. We made several attempts to decouple different object types (such as relay connection and circuit objects), but this proved to require significant changes throughout large parts of the Tor code. Without clear interfaces for these objects and little distinction between public and private members, we instead focused on moving functionality from these object types into new types that were designed to be thread-safe. For Tor experimentation and development, another challenge was the limited number of libraries developed for interacting or experimenting with Tor. There are several tools designed for Tor experimentation, but they rarely provide a library interface that allows developers or researchers to extend them or add features. An important exception to this is the Stem Python library [Joh20] for interacting with Tor, which is a well-documented, full-featured, and modular library that demonstrates design characteristics that research tools should aim to imitate.

## 6.5 Summary

Our relay implementation successfully used multiple CPU cores to improve the relay's throughput. The maximum sustained throughput tipled on the Raspberry Pi and quadrupled on the Intel server when running with three connection threads. The initial latency measurements also improved when using our implementation. Comparisons of different memory allocators showed a significant relay performance difference between memory allocators, and demonstrates the need for consideration of various memory allocators when implementing a multi-threaded relay. With a throughput of nearly 150 Mbps, the Raspberry Pi running the multi-threaded implementation is a much more viable system for running an inexpensive Tor relay.

# Chapter 7

# Future Work and Conclusion

Providing better CPU utilization using multi-threading has the potential to improve both high- and low-performance Tor relays as shown in Section 6.3. While not all relays today are CPU-limited, many more might become so as the demand on the network grows. More users, greater webpage download sizes, and faster average residential internet bandwidth all contribute to a larger load on the network and its relays. Congested relays have been shown to cause a large negative impact on stream latencies and throughput, so reducing the opportunity for congestion to occur will help the network better handle more traffic and users. Our multi-threading approach to supporting more users and scaling the network avoids the downsides of other network scaling designs described in Section 3.1. Multi-core CPUs are commonplace today and while clock speeds of desktop and server x86 CPUs see marginal yearly gains, core counts have grown significantly in recent years. The network should be capable of fully utilizing the resources donated by the community.

## 7.1 Security and Privacy

With any changes to the network or its relays, it is important to discuss possible harm to users' security or privacy. We do not provide a detailed security or privacy analysis in this work, but believe that the multi-threaded architecture does not introduce any new attacks on users or the network.[1] Tor provides limited protection against timing analysis attacks where an observer can

[1] The security of our architecture is not to be conflated with the security of our implementation that was built for experiments in Chapter 6. The implementation is certainly not production-ready and we make no claims about the security of the code itself.

monitor traffic at both entry and exit points of the network. Increasing the maximum throughput of relays may make this timing analysis easier if it leads to lower-latency circuits, but this may be counteracted by higher Tor usage and more users due to the faster relays and a faster network.

One important consideration is the effect on attacks that rely on distinguishable relay throughputs. For example, an adversary can probabilistically identify the relay that bottlenecks a circuit's throughput by correlating the circuit's throughput with individual relay throughput measurements taken using active probing [MKJ+11]. The probing is stealthy as it is indistinguishable from regular client traffic, and the adversary only needs to know the user's circuit bandwidth at the exit relay or web server. If they can observe several of a user's circuits, the user's guard relay can be identified as it will appear as the bottleneck more often than other relays. This attack depends on a wide range of relays with different throughput characteristics, and the faster throughput enabled by multi-threading may widen the gap between the slowest and fastest relays, making it easier for attackers to distinguish relays by their throughput. A related attack is one where the adversary takes advantage of performance-enhancing features in Tor to identify a user's guard relay. The adversary probes relays to observe their bandwidth while simultaneously inducing throttling on a specific user's high-bandwidth circuit [GJH13]. The intuition is that repeatedly throttling the user's circuit will cause a reduction in load on the relays that the circuit passes through. This repeated reduction in load will be observed by the probes as an increase in bandwidth, allowing the adversary to learn the user's guard relay. While this attack takes advantage of performance-enhancing features in Tor, namely its congestion control mechanisms, the multi-threading approach may reduce the effectiveness of this attack since a circuit's performance will be affected by at most two threads. If the adversary's probes end up on different threads than the user's circuit, the probes may not detect throughput changes caused by that circuit.

## 7.2 Future Work

Our multi-threaded relay implementation in Section 6.1 realizes only a portion of the multi-threaded architecture in Section 5.3. When scaling to several CPU cores, the relay's main thread becomes the bottleneck limiting the relay's throughput. It is likely that the entire multi-threaded architecture would scale much better as it performs the end-to-end processing of Relay cells outside of the main thread, unlike our implementation where most of the cell processing occurs in the main thread. Implementing the remainder of the architecture and performing throughput experiments would better show the scaling of the architecture across CPU cores. The multi-threaded relay implementation also needed to make design choices that led to over-complicated code in order to work around Tor's global state and circular references. Implementing the entire architecture would simplify the design of the code, reduce the global state, and eliminate the

circular references between connection objects. This is likely to be required if Tor is ever to move to a memory-safe language such as Rust.

The multi-threaded architecture divides the relay's global scheduler into several thread-local schedulers. These schedulers run independently under the assumption that if connections are being distributed and balanced across threads, then it is not important that the thread-local schedulers prioritize circuits globally. Once the entire architecture is implemented, experiments could be performed to see if this assumption holds in practice. If not, is there a scheduler design that works across threads and would be useful for a relay, while taking into account the relay's congestion and queueing latency concerns? Investigating different load balancing techniques when the relay is experiencing real-world traffic patterns would also be useful for making sure that each thread is being used efficiently.

The experiments in Section 6.2 studied only the relay performance and not the performance of the network. It would be useful to simulate a much larger and more realistic network to understand the effects on the network as a whole and its individual users. These additional experiments would benefit from a more complete implementation of the multi-threaded architecture to better understand the effects of changes such as per-thread connection schedulers and parallelized edge connections that were not implemented in our experiments. The experiments in this thesis focused mostly on relay throughput and stability, but additional latency measurements would also be useful for understanding the relay's congestion and users' experience.

### 7.2.1 Onion Services

Like relays, Tor proxies for both clients and onion services are single-threaded. As they are part of the same code base, the fundamental design of the proxy is very similar to the design of relay. As discussed in Section 3.3, the proxy is a bottleneck for an onion service since all traffic must pass through it. Onion services would benefit greatly from a multi-threaded proxy design, and as the proxy implementation uses the same code base as the relay, onion services may benefit directly from the implementation of a multi-threaded relay. The onion service scaling provided by multi-threaded proxies would be complementary to OnionBalance. Whereas OnionBalance could be used for scaling proxies across multiple servers and improve reliability, the multi-threading would make it easier to scale across many CPU cores on a single server. Since OnionBalance requires additional configuration by the onion service operator, a multi-threaded proxy would provide better performance out-of-the-box for operators who do not want to configure OnionBalance.

## 7.3 Conclusion

With the objective of scaling the Tor network by better utilizing existing relays, this thesis constructed a multi-threaded relay architecture that considers many aspects of a relay's routing, most importantly the processing of different cell types as they arrive on a connection. By implementing a subset of this architecture and monitoring its throughput and CPU performance when experiencing heavy traffic, we showed that the architecture does indeed better use processing cores. The performance of this restricted implementation is likely much lower than what the performance of the entire architecture would be, but the implementation gives a lower bound on what performance improvements could be expected from implementing the entire architecture. The maximum sustained throughput was more than four times greater when running with four threads on the Intel Xeon server, and more than three times greater when running with four threads on the Raspberry Pi. As we do not change any aspects of the network's architecture or protocols, these multi-threaded relays could be deployed by relay operators easily and without any coordination with other relays. CPU-limited relays with excess bandwidth capacity would be able to contribute more bandwidth to the network, allowing it to support more users.

# References


[ABG+11] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey M. Voelker. DefenestraTor: Throwing out Windows in Tor. In *11th Privacy Enhancing Technologies Symposium*, pages 134–154. Springer, 2011.

[AG16] Mashael AlSabah and Ian Goldberg. Performance and Security Improvements for Tor: A Survey. *ACM Computing Surveys (CSUR)*, 49(2):1–36, 2016.

[AYM12] Masoud Akhoondi, Curtis Yu, and Harsha V. Madhyastha. LASTor: A Low-Latency AS-Aware Tor Client. In *IEEE Symposium on Security and Privacy*, pages 476–490. IEEE, 2012.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *36th IEEE Symposium on Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[Cha81] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[CK20] Donncha Ó Cearbhaill and George Kadianakis. OnionBalance, 2020. https://onionbalance.readthedocs.io/en/latest/.

[DC06] George Danezis and Richard Clayton. Route Fingerprinting in Anonymous Communications. In *6th IEEE International Conference on Peer-to-Peer Computing*, pages 69–72. IEEE, 2006.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *IEEE Symposium on Security and Privacy*, pages 2–15. IEEE, 2003.

[DHKM14] Roger Dingledine, Nicholas Hopper, George Kadianakis, and Nick Mathewson. One Fast Guard for Life (or 9 months). In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2014.

[Din12] Roger Dingledine. Avoiding infinite length circuits, 2012. https://gitweb.torproject.org/torspec.git/tree/proposals/110-avoid-infinite-circuits.txt.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. *13th USENIX Security Symposium*, 2004.

[Dou02] John R. Douceur. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[DS08] George Danezis and Paul Syverson. Bridging and Fingerprinting: Epistemic Attacks on Route Selection. In *8th Privacy Enhancing Technologies Symposium*, pages 151–166. Springer, 2008.

[DSR+10] Prithula Dhungel, Moritz Steiner, Ivinko Rimac, Volker Hilt, and Keith W. Ross. Waiting for Anonymity: Understanding Delays in the Tor Overlay. In *10th IEEE Conference on Peer-to-Peer Computing (P2P)*, pages 1–4. IEEE, 2010.

[EBA+12] Tariq Elahi, Kevin Bauer, Mashael AlSabah, Roger Dingledine, and Ian Goldberg. Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor. In *ACM Workshop on Privacy in the Electronic Society*, pages 43–54, 2012.

[EDG09] Nathan S. Evans, Roger Dingledine, and Christian Grothoff. A Practical Congestion Attack on Tor Using Long Paths. In *USENIX Security Symposium*, pages 33–50, 2009.

[FM02] Michael J. Freedman and Robert Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. In *9th ACM Conference on Computer and Communications Security*, pages 193–206, 2002.

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 134–143, 2007.

[GJH13] John Geddes, Rob Jansen, and Nicholas Hopper. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *13th Privacy Enhancing Technologies Symposium*, pages 164–184. Springer, 2013.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gou18] David Goulet. cmux: Refactor, test and improve performance of the circuitmux subsystem, 2018. https://gitlab.torproject.org/legacy/trac/-/issues/25328.

[Gou19] David Goulet. Onion Service - Intropoint DoS Defenses, 2019. https://lists.torproject.org/pipermail/tor-dev/2019-May/013837.html.

[GRS96] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding Routing Information. In *International Workshop on Information Hiding*, pages 137–150. Springer, 1996.

[GRS99] David Goldschlag, Michael Reed, and Paul Syverson. Onion Routing. *Communications of the ACM*, 42(2):39–41, 1999.

[GSU13] Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and one-way authentication in key exchange protocols. *Designs, Codes and Cryptography*, 67(2):245–269, 2013.

[GWB97] Ian Goldberg, David Wagner, and Eric Brewer. Privacy-enhancing technologies for the Internet. In *IEEE COMPCON 97*, pages 103–109. IEEE, 1997.

[JGW+14] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *23rd USENIX Security Symposium*, pages 127–142, 2014.

[JH12] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society, February 2012.

[JJ16] Rob Jansen and Aaron Johnson. Safely Measuring Tor. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1553–1567, 2016.

[JJH+17] Aaron Johnson, Rob Jansen, Nicholas Hopper, Aaron Segal, and Paul Syverson. PeerFlow: Secure Load Balancing in Tor. *Proceedings on Privacy Enhancing Technologies*, 2017(2):74–94, 2017.

[Joh20] Damian Johnson. Stem, 2020. https://stem.torproject.org/.

[JTJS14] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. *Network and Distributed System Security Symposium (NDSS)*, 2014.

[KB20] Martin Karsten and Saman Barghi. User-level Threading: Have Your Cake and Eat It Too. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–30, 2020.

[KBC94] H. T. Kung, Trevor Blackwell, and Alan Chapman. Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing. In *Conference on Communications Architectures, Protocols and Applications*, pages 101–114, 1994.

[KMG20] Chelsea Komlo, Nick Mathewson, and Ian Goldberg. Walking Onions: Scaling Anonymity Networks while Protecting Users. *29th USENIX Security Symposium*, 2020.

[Lit61] John D. C. Little. A Proof for the Queuing Formula: L=$\lambda$W. *Operations Research*, 9(3):383–387, 1961.

[LLY+09] Zhen Ling, Junzhou Luo, Wei Yu, Xinwen Fu, Dong Xuan, and Weijia Jia. A New Cell Counter Based Attack Against Tor. In *16th ACM Conference on Computer and Communications Security*, pages 578–589, 2009.

[Mat13] Nick Mathewson. Improved circuit-creation key exchange, 2013. https://gitweb.torproject.org/torspec.git/tree/proposals/216-ntor-handshake.txt.

[Mat18] Nick Mathewson. See if we can allocate less for HMAC in Tor relays, 2018. https://gitlab.torproject.org/legacy/trac/-/issues/25007.

[Mat20] Nick Mathewson. Specification for Walking Onions, 2020. https://spec.torproject.org/walking-onions.

[MB09] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer Anonymous Communication Using Redundant Structured Topologies. In *16th ACM Conference on Computer and Communications Security*, pages 161–172, 2009.

[MBG+08] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining Light in Dark Places: Understanding the Tor Network. In *8th Privacy Enhancing Technologies Symposium*, pages 63–76. Springer, 2008.

[MD05] Steven J. Murdoch and George Danezis. Low-Cost Traffic Analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195. IEEE, 2005.

[MKJ+11] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *18th ACM Conference on Computer and Communications Security*, pages 215–226, 2011.

[MOT+11] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security Symposium*, 2011.

[Moz20] Mozilla. Firefox privacy, by the product, 2020. https://www.mozilla.org/en-CA/firefox/privacy/products/.

[MTHK09] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable Onion Routing with Torsk. In *16th ACM Conference on Computer and Communications Security*, pages 590–599, 2009.

[MWBJ+18] Akshaya Mani, T. Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. Understanding Tor Usage with Privacy-Preserving Measurement. In *Internet Measurement Conference*, pages 175–187, 2018.

[Per20] Mike Perry. RTT-based Congestion Control for Tor, 2020. https://gitweb.torproject.org/torspec.git/tree/proposals/324-rtt-congestion-control.txt.

[PK20] Mike Perry and George Kadianakis. Tor Padding Specification, 2020. https://gitweb.torproject.org/torspec.git/tree/padding-spec.txt.

[RP02] Marc Rennhard and Bernhard Plattner. Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection. In *ACM Workshop on Privacy in the Electronic Society*, pages 91–102, 2002.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.

[RSG96] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Proxies for Anonymous Routing. In *12th Annual Computer Security Applications Conference*, pages 95–104. IEEE, 1996.

[RSG98] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, 1998.

[SB09] Robin Snader and Nikita Borisov. EigenSpeed: Secure Peer-to-peer Bandwidth Evaluation. In *8th International Conference on Peer-to-Peer Systems*, page 9, 2009.

[SDH+10] Max Schuchard, Alexander W. Dean, Victor Heorhiadi, Nicholas Hopper, and Yongdae Kim. Balancing the Shadows. In *9th Annual ACM Workshop on Privacy in the Electronic Society*, pages 1–10, 2010.

[SG19] Sajin Sasy and Ian Goldberg. ConsenSGX: Scaling Anonymous Communications Networks with Trusted Execution Environments. *Proceedings on Privacy Enhancing Technologies*, 2019(3):331–349, 2019.

[SGD15] Fatemeh Shirazi, Matthias Goehring, and Claudia Diaz. Tor Experimentation Tools. In *International Workshop on Privacy Engineering (IWPE'15)*, pages 206–213. IEEE, 2015.

[SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *Network and Distributed System Security Symposium (NDSS)*, 2018.

[Syv05] Paul Syverson. Onion Routing: Brief Selected History, 2005. https://www.onion-router.net/History.html.

[TB06] Parisa Tabriz and Nikita Borisov. Breaking the Collusion Detection Mechanism of MorphMix. In *International Workshop on Privacy Enhancing Technologies*, pages 368–383. Springer, 2006.

[TG10] Can Tang and Ian Goldberg. An Improved Algorithm for Tor Circuit Scheduling. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 329–339, 2010.

[TJJ20] Matthew Traudt, Rob Jansen, and Aaron Johnson. FlashFlow: A Secure Speed Test for Tor. *arXiv preprint arXiv:2004.09583*, 2020.

[Tor15] The Tor Project. Tor Browser 4.5 is released. 2015. https://blog.torproject.org/tor-browser-45-released.

[Tor17] The Tor Project. KIST and Tell: Tor's New Traffic Scheduling Feature. 2017. https://blog.torproject.org/kist-and-tell-tors-new-traffic-scheduling-feature.

[Tor19a] The Tor Project. Meet The New TorProject.org. 2019. https://blog.torproject.org/meet-new-torprojectorg.

[Tor19b] The Tor Project. Mozilla Research Call: Tune up Tor for Integration and Scale. 2019. https://blog.torproject.org/mozilla-research-call-tune-tor-integration-and-scale.

[Tor19c] The Tor Project. New Release: Tor Browser 8.5. 2019. https://blog.torproject.org/new-release-tor-browser-85.

[Tor20a] The Tor Project. Chutney, 2020. https://gitweb.torproject.org/chutney.git.

[Tor20b] The Tor Project. Tor directory protocol, version 3, 2020. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt.

[Tor20c] The Tor Project. Tor Metrics — Servers, 2020. https://metrics.torproject.org/networksize.html?start=2020-01-01&end=2020-07-31.

[Tor20d] The Tor Project. Tor Rendezvous Specification — Version 3, 2020. https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt.

[Tor20e] The Tor Project. Tor Metrics Search, June 2020. https://metrics.torproject.org/rs.html#search/185.220.100.

[Ung20] Nik Unger. NetMirage, 2020. https://crysp.uwaterloo.ca/software/netmirage/.

[vBCZ+03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. *ACM SIGOPS Operating Systems Review*, 37(5):268–281, 2003.

[WALS04] Matthew K. Wright, Micah Adler, Brian Neil Levine, and Clay Shields. The Predecessor Attack: An Analysis of a Threat to Anonymous Communications Systems. *ACM Transactions on Information and System Security (TISSEC)*, 7(4):489–522, 2004.

[WBFG12] Tao Wang, Kevin Bauer, Clara Forero, and Ian Goldberg. Congestion-Aware Path Selection for Tor. In *International Conference on Financial Cryptography and Data Security*, pages 98–113. Springer, 2012.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[WMB10] Qiyan Wang, Prateek Mittal, and Nikita Borisov. In Search of an Anonymous and Secure Lookup: Attacks on Structured Peer-to-Peer Anonymous Communication Systems. In *17th ACM Conference on Computer and Communications Security*, pages 308–318, 2010.

[WTBS13] Chris Wacek, Henry Tan, Kevin S. Bauer, and Micah Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2013.