Improving Tor using a TCP-over-DTLS Tunnel

by

Joel Reardon

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

Waterloo, Ontario, Canada, 2008

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.

Joel Reardon

Abstract

The Tor network gives anonymity to Internet users by relaying their traffic through the world over a variety of routers. This incurs latency, and this thesis first explores where this latency occurs. Experiments discount the latency induced by routing traffic and computational latency to determine there is a substantial component that is caused by delay in the communication path. We determine that congestion control is causing the delay.

Tor multiplexes multiple streams of data over a single TCP connection. This is not a wise use of TCP, and as such results in the unfair application of congestion control. We illustrate an example of this occurrence on a Tor node on the live network and also illustrate how packet dropping and reordering cause interference between the multiplexed streams.

Our solution is to use a TCP-over-DTLS (Datagram Transport Layer Security) transport between routers, and give each stream of data its own TCP connection. We give our design for our proposal, and details about its implementation. Finally, we perform experiments on our implemented version to illustrate that our proposal has in fact resolved the multiplexing issues discovered in our system performance analysis. The future work gives a number of steps towards optimizing and improving our work, along with some tangential ideas that were discovered during research.

Additionally, the open-source software projects latency_proxy and libspe, which were designed for our purposes but programmed for universal applicability, are discussed.

Acknowledgements

I would like to thank my supervisor Dr. Ian Goldberg for his assistance and guidance in developing this work. I would also like to thank my readers, Dr. U. Hengartner and Dr. S. Keshav, for their participation and feedback.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet privacy does not exist—at least not inherently in its design. It is erroneously believed to exist; searching for the exact phrase "the anonymity of the Internet" in Google returns 89,200 results in September 2008. However, without extra precautions, all actions on the Internet are linkable, and privacy disclosure is a growing concern. Chapter 2 presents definitions of privacy in the digital age, and explains the need for privacy on the Internet. Various nations have decided to heavily censor the Internet of any political content, and Internet privacy allows citizens of those nations the freedom to access an unfettered Internet.

Fortunately there already exists an outstanding tool to enable Internet privacy, which has seen widespread use and popularity throughout the world. This tool is called *Tor* [16], and consists of a network of thousands of routers whose operators have volunteered to relay Internet traffic around the world anonymously. Clients send their traffic into the Tor network and it exits at another location in the network. Any entity that cannot view the entire network is unable to associate the sender of any traffic with its ultimate destination. Tor aims to ensure that no such entity can view the entire network by geopolitically diversifying its routers. Thus, even government actors cannot quickly and discreetly compromise the Tor network. We describe Tor in more detail in Chapter 3 and also briefly cover the fundamental networking protocols of the Internet.

Despite its popularity, Tor has a problem that dissuades its use for all web traffic—it incurs greater latency on its users than they would experience without Tor. Additional latency is expected since users are sending their data to locations that are geopolitical diverse before dispatching. However, not all latency is accounted for by transport time. The goal of this thesis is an exploration to determine where and why this latency occurs, and how we can improve Tor. To begin our enquiry, Chapter 4 presents a suit of tools we have developed to help perform our evaluation. First we present the `latency_proxy`, a tool to simulate packet latency, packet dropping, and packet reordering that occur on the Internet. Next is the library `libspe`, a library to perform dynamic system performance evaluation, including monitoring data structures and performing timing results of code. This

library is designed to allow the monitored program to operate without interruption, while permitting the evaluator to access it remotely to perform analysis.

In Chapter 5 we perform our system performance analysis of Tor. We determine the cost of transport latency and performing computations, discovering that there exist other sources of latency outside those components. We begin exploring Tor's datapath thoroughly to determine where this latency occurs. We find that congestion control mechanisms cause data to become delayed inside buffers in Tor and the operating system. We discover that these mechanisms are being used improperly because of Tor's design and quantify the degradation it causes.

We present the solution to this problem in Chapter 6. Our proposal is a new transport layer for Tor that, importantly, is backwards compatible with the existing Tor network. Routers in Tor can gradually and independently upgrade, and our system provides immediate benefit to any pair of routers that choose to use our improvements. We discuss how it was implemented, and provide algorithms for our new components.

Chapter 7 presents the experiments to compare the existing Tor design with our new implementation. We compare latency and throughput, and perform timing analysis of our changes to ensure that they do not incur non-negligible computational latency. Our results are favourable: the computational overhead remains negligible and our solution is successful in addressing the improper use of congestion control.

This thesis concludes with a description of future work. We detail steps that must be taken before our contribution can be deployed. Ways in which our design can be improved are presented. An extension for our design is the use of end-to-end congestion control—we explore this in some detail. Our research also uncovered some tangential observations that might improve Tor; these are included as well.

Chapter 2

Privacy

Personal privacy and the privacy of correspondence are considered fundamental human rights, proposed in article 12 of the United Nation's Universal Declaration of Human Rights [72]. Countries with democratic tradition are the most respectful of privacy, with Germany and Canada leading the world in legislation aimed at preserving this right [54]. Private information, however, has never been more accessible than in our electronic age. Personal information is being stored digitally, aggregated, searched, shared, bought and sold, and kept indefinitely at an unprecedented rate.

A senior Central Intelligence Agency director has stated that it is time to rescind the notion that privacy is a right afforded to us: "Protecting anonymity isn't a fight that can be won. Anyone that's typed in their name on Google understands that. Instead, privacy, I would offer, is a system of laws, rules, and customs with an infrastructure of Inspectors General, oversight committees, and privacy boards...." [37]. Privacy Rights Clearinghouse chronicles near-daily data breaches that occur, including losses by government officials [55]. The common trend is the movement of data to laptops that are then stolen, or the improper disposal of sensitive information. A recent survey indicates that ten thousand laptops are lost each week in the largest thirty-six airports in the United States, with half containing either customer or private information [51]. The report added that two-thirds of travellers admit that they do not take any steps to protect or secure their laptops. Until legislation is enacted to ensure that those whose privacy has been violated are properly compensated by the offending firm, firms will be unlikely to choose to delete private data they collect, or put in place the proper safeguards to ensure the cessation of these data leaks.

Certainly the case is easily made that computer networks are prone to attack [50]. No firm can ensure that data is protected without incurring large costs. We suspect that any legislation that punishes data leaks will likely result in many firms no longer maintaining data records that are not directly necessary. For instance, Google was recently forced by a court to reveal customer usage data for their video viewing site *YouTube* to the private copyright holder Viacom [12]. While

Google bemoans the court decision as unjust, others criticize Google's insistence that IP addresses do not constitute personal information [77] and their decision to maintain comprehensive usage records years after their relevance, which Viacom has used to their advantage [12].

2.1 Perceptions of Privacy

Many people falsely believe that a physically unobserved act is private. As a colloquial example, one friend expressed happiness for our new radio-frequency identification (RFID) system for tracking library books, since they were no longer required to disclose their borrowed books for inspection when exiting the library. In reality, the new system may afford them less privacy as each book, without access control, broadcasts a unique identifier to whomever carries an inexpensive RFID reader. Online purchases are illustrative of this perception of privacy; bashfulness can result in purchases of embarrassing items over the Internet with a credit card in their name rather than paying with cash anonymously in person.

People generally want privacy, but seem only concerned about privacy when they are informed or reminded about it. A study of Canadians showed that nearly half refuse to offer private information, such as telephone numbers, to shopkeepers upon request [8]. Half also have asked retailers why such information was being collected. This is prudent behaviour: divulging such information is rarely in its subject's best interest. To explore the importance of privacy, John et al. [35] performed a study was done that exposed subjects to a website that prompted them for private information such as their inclinations towards disreputable acts. One website was designed to look professional and provided very strong privacy guarantees, and the other was designed to look childish and offered no such guarantee. Paradoxically, it was revealed that those who had been given privacy reassurances on the professional site were *less* likely to reveal private information than those who had been given no such assurances on the childish site. They concluded that the mention of privacy set off alarms in minds of the participants, who then chose to dismiss the assurances and protect their privacy themselves. The lack of any mention of privacy resulted in the participant giving no consideration to its loss.

The search engine *Ask* recently re-advertised itself as a privacy-enhancing search engine—promoting their AskEraser feature that allows users to erase their dossier of collected search results easily [5]. This feature, disabled by default, also warns that it may not delete search history, for instance, when a critical error occurs or they have been given a legal obligation in a local jurisdiction to collect search information for law enforcement. The paradoxical nature of privacy suggests that mentioning that search histories are being recorded and indexed by their identity will make people more concerned and suspicious about the service, even if the service deletes them in *bona fide*, and the service is a better choice for the privacy-concerned. Another search engine, *cuil*, made a wiser decision; it simply avoids collecting any identified search information altogether, and mentions this in the privacy policy

for those who are sufficiently concerned about privacy to investigate [14]. The fact that privacy concerns are strengthened when they are mentioned is likely why Google resisted placing a link to their controversial privacy policy on their front page [56]. Recently they have acquiescing for some countries where demand has been vocal [21].

2.2 The Nymity Slider

Different transactions disclose identity to different degrees. In Goldberg's Ph.D thesis [23] he presents the nymity slider, which clarifies the differences between levels of identity disclosure. It is a virtual one-dimensional slider that ranks the nymity of transactions along the axis of privacy. The highest position, *verinymity*, means true name. A verinymous transaction involves the disclosure of one's true identity, or information linked uniquely to their[1] true identity. Examples in real life include using one's credit card, or showing government identification to prove one's age. Below verinymity is *pseudonymity*, meaning false name. A pseudonym, or an alias, is a persistant front name behind which the true identity is hidden. In the real world, pseudonyms include author's pen names, or nicknames used by bloggers.

The important distinction between verinymity and pseudonymity is that one has control over the disclosure of their own identity—only the holder of the pseudonym can choose to reveal it. Verinymity can be obfuscated, such as a Social Insurance Number (SIN) that uniquely corresponds to an identity. However, the lack of personal ownership over the mapping from one's SIN to their identity makes it decidedly verinymous.

The remaining positions on the slider are two varieties of anonymity, meaning without names: linkable anonymity and unlinkable anonymity. Linkably anonymous transactions are transactions that can be linked together, however the linking does not correspond to a real identity or a pseudonym: they are simply correlated. An example of this is seen in retail outlets that use loyalty cards to offer a small discount. (Often these cards are trivial to acquire and offer immediate benefit.) Such cards require no registration and are not linked to any identity. They are valuable to retail outlets for collecting data about customer buying patterns so as to juxtapose items to encourage "impulse buying". Finally, the lowest level on the nymity slider is unlinkable anonymity. This is true anonymity, where individual transactions are unlinkable to an external party or to each other. Using cash for a purchase, or providing the police with an anonymous tip over the telephone, are examples of unlinkably anonymous transactions.

Importantly, the position on the nymity slider can move towards verinymity during a transaction much more readily than it can move towards anonymity. When

[1]This thesis will use pronouns for the third person plural as the singular third-person pronouns of indefinite gender when it is unambiguous with another plural object.

one chooses to disclose an aspect of their identity during a transaction, such as using ID to prove their age of majority, then that transaction becomes verinymous even if the purchase is then done anonymously with cash. Similarly, an unlinkably anonymous transaction becomes linkable when the decision is made to use a loyalty card for a discount.

2.3 Internet Privacy

The hidden user aspect of the Internet leads it to be widely considered anonymous; this is epitomized by the cartoon of a dog using a computer captioned by "On the Internet, no one knows you're a dog."[19] People believe that their transactions are unlinkable if they choose not to disclose their name. However, each Internet user has an IP address that is revealed whenever a web server is accessed. IP addresses are generally unique for each Internet user, however there are some caveats. Some users have their exact IP address hidden behind network address translation (NAT), which maps all computers on a local network to a single external IP address, which provides anonymity up to the set of users on their local network. Others are assigned IPs by their Internet Service Provider (ISP) temporarily, and so IP addresses are only unique for a particular user at a particular time. This addressing system is necessary for the backbone of the Internet to route traffic appropriately; however, it allows their transactions to be linkable by IP address.

An IP is a verinym, because the ISP is aware of the customer's true name that it associates with each IP. The user lacks self-determination of the correlation between their IP and their true identity, and ISPs are generally under no contractual obligation to maintain the privacy of this association. In fact, ISPs can be forced by courts to reveal the identity of their users when a copyright holder makes a complaint regarding intellectual property infringement [28].

2.4 Privacy Enhancing Technologies

A privacy enhancing technology (PET) is a technology whose goal is to enhance the privacy of the user. That is, a technology that prevents the position on the nymity slider from careening needlessly towards verinymity for its user's transactions. These technologies are designed for use in the online world, and are employed in a variety of contexts including instant messaging, web-browsing, and electronic publishing. Goldberg et al. [27, 24, 25] present a quintennial series that examines the progression of PETs. They contend that the most widely successful PET in history is Transport Layer Security (TLS), which is used to encrypt and secure Internet communication. It has become an invaluable tool on the Internet, and is used for nearly all authentication, e-commerce, and server remote control panels. Another successful PET is Off-the-Record Messaging [7], a plugin for popular instant messaging programs that provides authentication, security, confidentiality,

and deniability to conversations. Contributing to its success is both the widespread use of instant messaging for private conversations and the fact that it is provided as a standard component for Adium [1] and is used automatically when both parties support it—some users do not even realize that their conversations are being protected from eavesdroppers. This is the ideal way to give privacy to users; to be widely used it must be transparent to use and trivial to configure.

Anonymity in the real world depends on the number of users of a system. While post office boxes allow the owner to receive mail anonymously, if the owner was the only recipient of mail in the system then it would afford him very little privacy. Our pragmatic definition of privacy for in the electronic age is based on this reality of anonymity: we say to be anonymous is to be *indistinguishable from a set*; this set is called the *anonymity set*. The definition considers the existence of an adversary who has the goal of identifying an anonymous user. A successful PET would result in the adversary being unable to determine which user (from the anonymity set) is the target for whom they are searching. More strongly, it would prevent the adversary from pairing any user of the system with their identity. The goal in PETs is to have the anonymity set equal to the set of users. This definition creates a corollary goal for any PETs that we develop: it is imperative that such technologies are popular and well-used, as this will increase the anonymity set and thus strengthen the degree of anonymity afforded.

2.5 Modern Challenges

In the electronic age, privacy is often at odds with convenience. Lust of technology results in rapid adoption of fantastic technologies without considering the detrimental effect on privacy involved. Google's Street View is an example of this phenomenon. Their service conveniently shows street-level images in many American and world-wide cities, allowing virtual tourists to mill about town without leaving their computers. Unfortunately, it also shows cars and pedestrians nearby at the time of photography, going about their days, neither consenting nor aware of the photography that Google intends to preserve indefinitely. Figure 2.1 shows the resolution of the camera that has taken millions of such photographs. Privacy concerns have usually been dismissed with the statement that one who is in public has no right to expect privacy. Google, in a lawsuit against them over Street View, contends that "complete privacy doesn't exist [in the modern world]," while saying that Google "takes privacy very seriously." [63] The chairman of the US National Legal and Policy Centre, shocked by Google's apparent hypocrisy, puts forth that "in the real world individual privacy is fundamentally important and is being chipped away bit by bit every day by companies like Google." [63]

The view that one has no reasonable expectation to privacy outdoors is contentious, however, since minors or those in witness protection are also photographed without discrimination. Even the US Army is forced to opt out of Google's service, demanding that areas around sensitive sites are removed from their maps. Google

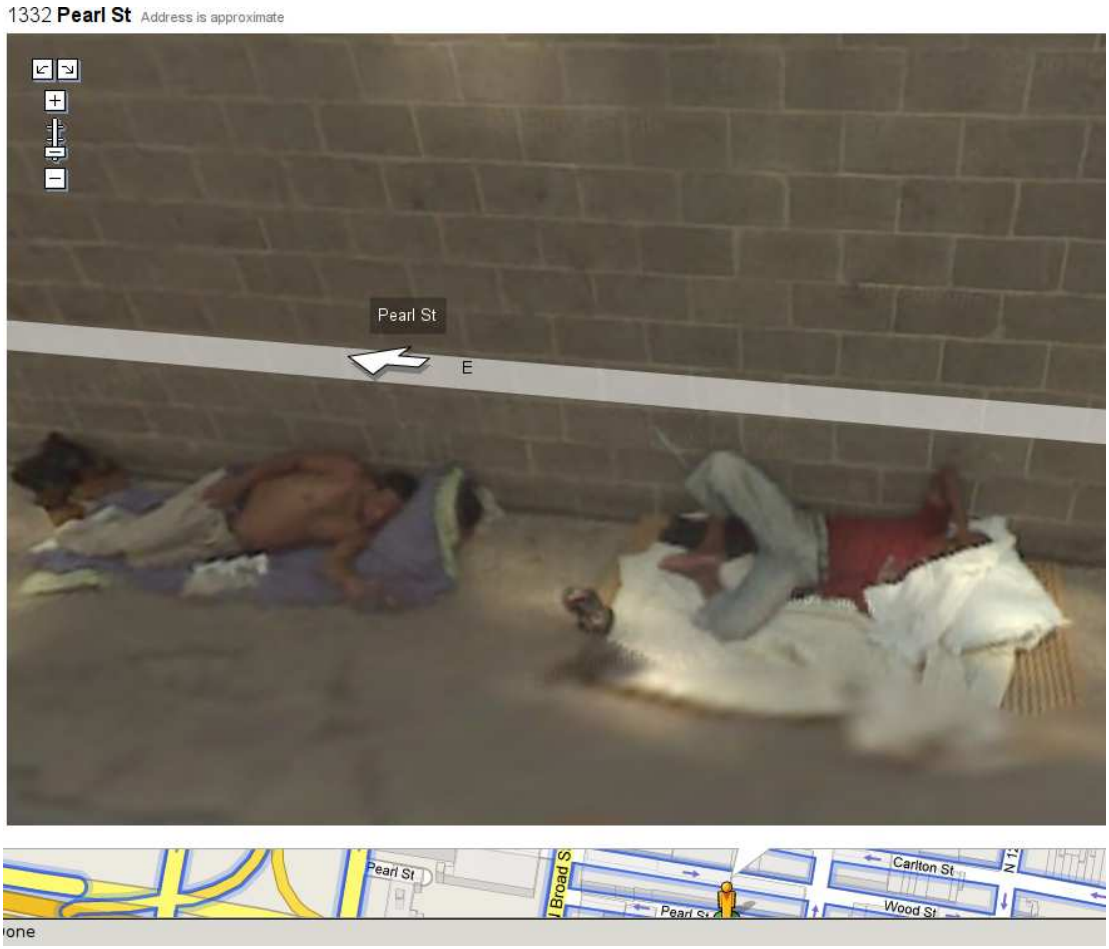

Figure 2.1: An example Google Street View image, taken in downtown Philadelphia, Pennsylvania.

chose to acquiesce to the demands of the US Army. The Canadian privacy commissioner has ruled that Street View is an egregious violation of privacy [67], and has reminded the content providers that under strong Canadian privacy laws, any person who may possibly be identified must provide express permission for their photograph to be used. Fortunately, the Canadian privacy commission has greater authority than American public opinion, and Google acquiesced to her concerns by ceasing their intentions to take street-level pictures throughout large Canadian cities. While such a service may indeed be useful for travellers who meticulously plan details of their vacation, including prefamiliarizing themselves with every turn *en route* to their hotel, it is far more important that the basic rights of humans beings are not rescinded by technological proliferation.

The Internet is perhaps the greatest challenge; we need an Internet that permits all users easy transparent access to information while still respecting fundamental human rights. In fact, the Internet has become intertwined with human rights and justice because of its use in underground publishing of political dissidence. While the United States used its economic strength to allow Radio Free Europe to broadcast jazz and unsavoury truths about the USSR into the iron curtain [49], the Internet has the power to make underground grassroots movements have massive impact, both locally and globally; flashmobs [70] and Pangea Day [47] are examples of Internet-based coordination of assembly. Moreover, a lack of privacy on the Internet permits some nations to persue active censorship of vast quantities of data on the Internet [69]. Finally, the Internet has resulted in the loss of personally identifiable information for many people who do not realize its worth. This is leading to a great rise in crimes related to identity theft as identity thieves find getting personal information a trivial endeavour, unlike the days of rummaging through rubbish for unshredded documents. In the remainder of this section we explore the importance of Internet privacy as it relates to political dissidence, censorship, and personal information.

2.5.1 Political Dissidence and Human Rights

Many regimes worldwide currently lack the same freedoms and access to justice as Canadians citizens enjoy, and consequently political activists have been arrested and victimized by these regimes. Being held as a political prisoner is truly a heinous fate, as regimes that consider political dissidence a criminal offence tend to dismiss human rights and often disregard the importance of *habeas corpus* and the Geneva convention.

Dissidents, whistleblowers, and journalists who believe that the Internet provides them anonymity make what can be a tragic mistake. The year 2007 witnessed the arrests of 36 bloggers—a record number—for their journalistic activities, with China and Egypt leading in the fight against free expression [29]. The Internet search and media giant Yahoo! has, at least four different times over its existence [57, 58, 60, 59], promptly aided the Chinese government arrest dissidents

by revealing the true identity for a Yahoo! email address. The cited reports are of journalists who had written dissident articles or sent materials to pro-democracy organizations. Yahoo!'s cavalier approach to user privacy has landed it in trouble in American courts. Yahoo! claims it did not question why the government wanted the identities, that there was no correlation with the divulging of information and that user summarily being arrested, and insist that they must adhere to local laws giving them no choice in the matter. A US congressional panel deemed Yahoo!'s testimony to be false and characterized its actions in preparing testimony as "inexcusably negligent behavior at best, and deliberately deceptive behavior at worst." [73]

While Yahoo! has perhaps the worst human rights record for an online firm, there is some competition. In May 2008, Google supplied information to the government of India to track the identity of a user of *Orkut*, Google's social networking site. The man used Orkut to express dissent against an Indian politician, stating "I hate Sonia Gandhi." [17] With Google's assistance he is now arrested and jailed by Indian authorities [61].

2.5.2 Internet Censorship

The proliferation of Internet censorship techniques in some nations that fail to see the value in having free and adversarial journalism is another reason for the need for Internet privacy. This is the dual problem to protecting a dissident author's identity; here we intend to protect the identity of someone retrieving censored data. If an external adversary is unable to determine that a client is making a request to a censored webpage then they will be unable to censor the data.

The People's Republic of China is the canonical example of an authoritarian regime that has opted to censor the Internet to avoid the promotion of free expression. They have managed to operate the most comprehensive censorship campaign in all of human history. Their endeavour is called the Golden Shield Project, but has been colloquially known as the "Great Firewall of China", as an homage to their *bona fide* world wonder.

Topics of a political nature are most often censored: human rights, democracy and liberalism, any non-government sanctioned news, the situation in Tibet, and the Tienanmen Square massacre of 1989. The success of their censorship and chilling effects is apparent by the fact that the majority of Chinese citizens have not heard of the Tienanmen Square student protests and the resulting slaughter of hundreds to thousands of students [6]. The 2008 Olympics that are being hosted in Beijing, China have made this decade-old censorship suddenly a topic in western media, informing some citizens for the first time of its existence. One commented on the radio call-in program "Cross-Country Checkup" that he had heard a rumour, whose veracity he questioned, that China was going to somehow block Internet websites to journalists during the Olympics [13]. He was unaware of the fact that the Golden Shield project has existed for over a decade—well before China was

rewarded with the Olympics. Reporters who have travelled to China have been shocked to discover that the Internet is in fact heavily censored; this is in spite of the fact that China has eased its censorship for the Olympic period to the most permissive state to date. Initially it was stated that censorship would not be performed during the Olympics, however it was revealed later that the International Olympic Committee had quietly agreed to censorship for political topics [33]. Perhaps with greater recognition of these issues will come a greater impetus to change.

A controversial issue with regards to enabling this censorship campaign has been the eager support of Cisco systems to design and retail censorship solutions to China. Cisco's internal documents, leaked accidentally, revealed that they viewed the China's censorship as an opportunity to sell more routers [66]; however, they publically dismiss any proclivity towards censorship: "Cisco strongly supports freedom of expression on the Internet, and we respect the conviction of those who have brought these concerns forward." [74] A private shareholder initiative would have made Cisco, among other things, publish a yearly report detailing how Cisco ensures that it is not violating human rights; however, two-thirds of shareholders voted with the board of directors to avoid such a focus on human rights [38]. Their defence to their business decision is that they simply sell generic hardware, in a competitive market, to an international audience [74]. While China chooses to use their hardware for censorship, Cisco feels that they have done nothing, in the totality of their Chinese operations, for which they should feel ashamed [74]—their hardware is not customized explicitly for the purpose of censorship [74]. This is poetically similar to the reasoning that the 15-year-old Baltimore drug dealer DeAndre McCullough offered when asked by a journalist how he justifies selling heroin despite it having destroyed both his parents' lives: "[The addicts] are gonna buy it from somewhere, so it might as well be me." [64] Google, Yahoo!, Skype, and Microsoft agree with McCullough's reasoning; their duty to their shareholders outweigh any ethical considerations of censorship, and are complicit in content censorship so as to be able to operate in mainland China [30].

John Gilmore, the founder of the Electronic Frontier Foundation (which has a fundamental goal of fighting censorship) quips: "the Internet interprets censorship as damage and routes around it." [22]. This statement was made in regards to the Streisand effect [40], which is that any attempt to remove information from the Internet ironically results in the rampant proliferation and broadcast of something that would otherwise be relatively unknown. A classic example was the attempt to publically remove a secret code used in copy protection by threatening legal action against a website; a user community outraged over their website's compliance began a rebellion, prolifically repeating the secret code, ostensibly out of spite rather than legitmate dissemination, until the story was widely repeated by various news agencies [10]. Ideally, the Internet would be impossible to censor without abandoning its use entirely. However, the lack of privacy for senders and receivers of data makes censorship by Internet access providers possible. By ensuring no observer can determine both the source and destinations of Internet traffic we can prevent any targeted censorship of the Internet.

2.5.3 Identity Theft and the Dossier Effect

It may be difficult for the fortunate citizens of North America to relate to the need for a political dissident to have privacy; our British tradition makes political satire a pillar of our democracy. However identity theft motivates the need for privacy among citizens of enlightened democracies. Identity theft is an increasingly popular crime in the modern world, particularly in Canada [62]. The premise of identity theft is that a criminal uses someone else's identity to perform some transaction. Identity theft is often used to enable other crimes such as fraud. These transactions are often monetary in nature, such as withdrawing sums of money from another person's bank account, or using someone else's private information to obtain a credit card in their name. Identity theft can have devastating effects on people's credit history with consequences that are strenuous, if even possible, to resolve. Some victims of identity theft are detained by police serving warrants in their name [4].

The depth of the problem is only hypothesized in Canada, as Canadians banks intentionally tell victims not to report the crime to the police [45]. This business decision indicates that bad press against the bank is more fiscally harmful than simply restoring the funds without a police investigation. It is unsustainable for banks to continue passing the cost of identity theft evenly to its customers via service fees, since the identity theft will only continue to worsen until penalties against the perpetrators are exacted. The Royal Canadian Mounted Police are pleading for all victims to report the crime [62], and in a report on identity theft, proposed that banks be subjected to mandatory reporting laws for cases of identity theft [45].

Identity theft is made possible because criminals are able to access people's personal information [53]. The dossier effect is a means by which an eavesdropping adversary can collect a wealth of personal information about someone through linkable transactions involving minor privacy disclosures [23]. As established, all Internet traffic is easily linkable by the IP address of the source, and most webservers only use encryption for password authentication. A user might be prompted for their birthday to obtain a horoscope, their postal code to obtain a list of nearby retail outlets, and their name for a vanity search. This allows a passive eavesdropper, or the websites themselves, to collect a dossier of personal information by linking each mild disclosure to build a massive result. Once built, this dossier allows the identity thief to fill out credit card applications, answer identity challenges, etc.

Chapter 3

The Network Architecture of Tor

Providing anonymity to parties communicating over the Internet is an area of active research. The goal is to hinder attempts by an external observer to learn the association between pairs of communicators. Connecting the flow of data through a network uses techniques in traffic analysis, and so anonymity research includes both developing attacks using traffic analysis and then attempting to thwart their effectiveness.

Internet applications can be partitioned into two groups based on their demand for interactivity: those that demand low-latency and those that can tolerate high-latency. Low-latency applications require interactive communication and prompt responses: remote shells, instant messaging, and web browsing. High-latency applications permit far greater latency for communication without significantly impacting usability, as replies are not necessary or can be delayed: electronic mail, newsgroups, file downloads, low-bandwidth massively distributed computations, and electronic publishing.

High-latency anonymity has been widely studied [9, 15, 44]. Traffic analysis can be prevented by sending Internet traffic through a relay that is simultaneously used by many others; this relay is known as a *mix*. Its purpose is to group together a set of senders and their corresponding set of recipients. Each element of the set has an equivalent statistical likeliness as being the actual source or destination. Recall that this matches our definition of privacy; i.e., being indistinguishable from a set. Mixes use the dimension of time to achieve anonymity—they collect messages over time and relay the resulting set in batches. An adversary watching data enter and exit cannot correlate the two beyond the anonymity set (the batch of messages). As this problem already has reasonable solutions, we focus this thesis on the field of low-latency anonymity research.

Low-latency anonymity uses a relay to prevent a server from learning the identity of a client while still remaining useful for interactive purposes. The client sends their traffic to the relay, which dispatches it to the server on their behalf. The server links the transactions as having come from the perceived source; i.e., the relay. This system necessitates that other clients must use the relay, as we have

already established that a privacy enhancing system with one user affords little privacy.

This single relay is vulnerable to a number of attacks. Single-hop relays rely on the operator's unwillingness to disclose their users' identity. The relay itself is able to perform perfect traffic analysis, and so users must be able to trust the relay with their privacy. As we have seen with Yahoo!, for example, government actors are capable of coercing an entity entrusted with privacy rather easily. We present Tor in the next section, which uses relays as a building block in the design of a robust and powerful Internet anonymity PET.

3.1 Tor

Tor is the most successful PET that provides low-latency Internet anonymity. It provides anonymity based on relaying network traffic, but uses multiple relays in series to protect users' privacy. Each relay is aware of the relays adjacent in the series, but no entity (except for the client) is aware of both the client and the destination. By geographically and politically diversifying the selection of relays, Tor aims to render infeasible the legal and coercive attacks possible for single-hop relays.

Tor is the second generation of the Onion Routing project [68] that started at the US Naval Laboratories. The goal of Tor is to anonymize all Internet traffic—specifically traffic for low-latency or interactive applications. Its anonymity goal is to frustrate any attempts at traffic analysis; i.e., preventing attackers from either discovering pairs of communicating entities or linking multiple communications that share an entity [16]. Data is sent into the Tor network as the packets for one end of a TCP connection, which is relayed along a circuit built out of single-hop relays. It uses multiple layers of encryption, wrapped like an onion, which are removed at each relay along the path. This prevents any two messages from looking the same to an observer, and also prevents any relay from knowing the precise data being transported—except for the last relay, which actually dispatches the TCP request being made. The results are relayed back through the Tor network, and the original application making the connection receives the response as though it were dispatched locally. The Tor network consists of many computers whose owners have volunteered to run the Tor software, which relays and dispatches traffic. The relays in the network are called Onion Routers (ORs); we also refer to them as nodes. The anonymity requirements necessitate that it is difficult for an adversary to correlate the ORs where data enters and exits the Tor network. Tor accomplishes this by relaying traffic through the network with the goal of routing traffic through a segment that the adversary cannot observe.

Tor is currently the most advanced and successful provider of low-latency anonymity, having been launched in 2003 and, as of time of writing, has had five years of continuous up time despite major system and software updates. It is completely

free and open source, and relies on thousands [71] of volunteers to help route traffic. Tor has hundreds of thousands of users around the world; many thousands of people in China rely on Tor to provide them with anonymous and uncensored Internet access [41].

3.1.1 Threat Model

Tor's threat model assumes that the adversary is a non-global active adversary. Early work in anonymous systems often assumed the existence of a global passive adversary [9]—an entity who can observe all traffic in the network but does not interfere with any communication. A global adversary is confounded in most anonymous systems through the dimension of time; i.e., collecting messages over time and dispatching them in a batch. This model is too restrictive when designing a pragmatic low-latency anonymous system for use in web-browsing since it cannot be secure against a global passive adversary while still remaining useable for interactive tasks. Tor aims to provide a reasonable tradeoff between anonymity and performance in a low-latency setting. It assumes that its adversaries can only control small segments of the network: they can observe, modify, generate, delete, and delay traffic in that network segment, and can compromise a fraction of the computers in the Tor network. Tor uses a jurisdictional approach to protect client privacy, under the assumption that an adversary cannot effectively, or at least both rapidly and discreetly, operate in all jurisdictions.

3.1.2 Basic Operation

The ORs in the Tor network are all volunteers with minimal oversight to prevent malicious operators from participating. One need only run the free and open-source Tor program, and they become an OR. The main restriction is that only one OR can operate per unique IP address. To mitigate the risks inherent in such a volunteer system, Tor empowers clients to select their own ORs for their circuits and uses the notion of distributing trust: one cannot fully trust every node, so one diversifies their trust by making use of several different nodes. The client runs an Onion Proxy (OP) that builds a *circuit* through the Tor network; a circuit is an ordered list of routers through which their traffic will be routed. Each OR on the circuit knows only the entity directly previous and successive on the circuit. The first node in the circuit is called the entry node and is the only node aware of the original sender. The middle nodes are responsible for forwarding data they receive. The final node on the circuit is called the exit node; it is responsible for communicating with the client's intended destination and forwarding the replies back through the network.

Clearly there are abuse issues inherent in a system where a computer owner volunteers to perform arbitrary Internet transactions with the intention that they are not aware of the actual transaction being performed. Tor exit nodes can be used to dispatch spam or to perform network abuse reconnaissance such as port sniffing.

To encourage volunteers, Tor permits an expressive language for exit policies: the operator can block access over arbitrary ports and addresses to mitigate abuse issues.

Each node in the circuit is only aware of its direct neighbours. Anonymity is granted provided the nodes do not collude. The exit node is aware of the destination, and the entry node is aware of the client making the request. No node in the system is aware of both the client and the destination unless they share information. Circuit construction defaults to a length of three—this is to ensure that the exit node not only does not know the source of the request, but it is further unaware of which node knows this information as it is hidden behind a middle node. The question of the ideal circuit length, however, is an open question for research [16]. Tor's design documents emphasize that nodes on the path should be selected so that they are geographically and politically diversified. This makes it difficult for a single government entity to compromise every node along a circuit. Circuit selection also attempts to avoid multiple ORs under the same malicious operator by selecting ORs that come from distinct /16 networks.

The name onion routing is an analogy to the mechanism used to ensure that the data in a cell is revealed only to the last node: the payload is wrapped with multiple layers of encryption, and each node in the circuit peels away the topmost layer before relaying the cell. During circuit construction, the client negotiates a unique encryption key privately with each OR along the circuit. When it sends data along the circuit, it first encrypts the data for the exit node, then encrypts it again for each node backwards along the circuit. When the data is sent, each node applies the decryption function to remove its layer of encryption. The use of encryption at every node aims to ensure that nodes along the circuit are only aware of their direct neighbours. The final node on the circuit, after decrypting, will be left with plaintext data on which it can accordingly act. This procedure happens similarly in reverse, where each node encrypts the data when it passes, and the client applies the decryption function to remove each layer of encryption that has been added.

Tor packages all data into fixed-size cells for dispatching on the network. The design decision was to make cells 512 bytes in size, meaning that a little more than two cells can be dispatched in a single TCP packet over a typical network with a maximum transmission unit (MTU) of 1500 bytes. Streams are always cell-aligned, so the assumed size is the only delimiter for traffic in Tor. Each cell has a command that is used to control operations: creating new circuits, destroying old circuits, relaying data, and actually communicating with the destination.

3.1.3 Alternative Approaches

Tor is not the sole anonymity proxy available. For comparison, we outline the mechanisms used for alternative anonymity proxies to understand how they differ from Tor.

Anonymizer

Anonymizer [3] is a single-hop proxy. Clients open a connection to the anonymizing proxy, which strips their identifying information and forwards the result to the intended destination. This is a simple and effective design but the proxy is aware of the source and destination of messages—users must trust the single hop to protect their privacy. This creates a single point of failure; an entity that can observe traffic entering and exiting the proxy can break a user's privacy. Since there are no circuits, each user has separate connections for all their traffic.

Jondonym

The Jondonym project [2], formerly known as the Java Anon Proxy, is a multi-hop proxy similar to both Tor and Anonymizer. It amalgamates traffic into cascades—a fixed well-known circuit. Multiple hops are used to ensure that no entity knows the source and destination of messages. However, the circuit is public; any observer knows the nodes it must compromise to break a user's privacy. Users must trust the fact that the nodes on the fixed-circuit do not collude. Moreover, an entity that can observe traffic at each end of the cascade can perform effective traffic analysis [16].

Freedom

The Freedom network [26], which was a business endeavour to provide Internet anonymity, shares many similarities to Tor. It uses client-guided circuit construction, where circuits are built through a network of routers whose operators were paid to route traffic. UDP was used as the transport layer for all traffic between nodes. TCP traffic was captured from the client and sent over UDP, forwarded along the circuit, and reassembled into TCP at the exit node. This means that the client and the exit node share a TCP connection, with end-to-end reliability and congestion control. This has been proposed for Tor as well, which is discussed next.

UDP-OR

Another transport mechanism for Tor has been proposed by Viecco that shares similarities with the one that will be proposed in this thesis [75]. The proposed mechanism is to encapsulate TCP packets from the OP and send them over UDP until they reach the exit node. This means the reliability guarantees and congestion control are handled by the TCP stack on the client and the exit nodes, and the middle nodes only forward traffic. The benefits and concerns of this approach are discussed in Section 8.1.8.

Viecco's proposal requires a synchronized update of the Tor software for all users. This may be cumbersome given that Tor has thousands of routers and an unknown number of clients estimated in the hundreds of thousands. To manage

the system upgrade, it may be possible to first migrate the ORs to support the new method, but not actually use its functionality. Once upgraded, the exit nodes could inject content in their returned HTTP streams indicating to users that an upgrade is necessary before a certain date or their client will no longer function. Updated clients would also receive this warning message, but could remove the warning from the HTTP stream before forwarding it to the application. This means that all clients are warned in an obvious and disruptive manner until they upgrade, and the period of time for which this is done would be sufficiently long so as to warn all users. This is important, since some of Tor's users now need Tor simply to access Tor's homepage. An obvious concern is that if a client does not authenticate the received source code manually, a hostile exit node can provide a spurious link or simply deliver a modified Tor package that harms its user's privacy. The Tor network can audit for this attack by randomly downloading Tor from every exit node and confirming that it is the expected package.

3.2 Transport Protocols

To understand the sources of latency in the Tor network, one must first understand the fundamental transport protocols of the Internet: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). The content of this thesis relies heavily on the differences between these protocols so they are discussed in this section for completeness. The reader well-versed in networking can skip this section.

3.2.1 User Datagram Protocol (UDP)

UDP is a simple protocol for sending a packet of data over the Internet. It permits a computer to listen on a virtual port for a message, and reads the payload for packets of data that are sent. The two peers in communication are the connector and the listener. They communicate using virtual port numbers that identify their particular stream of communication. The listener opens a socket for the port on which they are interested in receiving data, and that socket indicates it is readable when a packet arrives destined for that port. UDP sockets can behave in a connectionless state: when reading from the socket, both the packet payload and the sender's address will be returned. This allows one socket to multiplex data from many other computers. Connectionless sockets require the destination's address for all outgoing datagrams. UDP sockets can also behave as connected sockets: they will reject, with an ICMP error message, all packets that were not sent from the intended peer. When sending data, the user need no longer specify the target address: it must be the peer to which the socket is connected.

However, the Internet is a chaotic and unpredictable asynchronous packet-switching network. Unlike circuit-switching networks (e.g. the telephone service)

that devote a number of fixed-speed and fixed-delay circuits, packets on the Internet are routed and traded around the world by dozens of routers cooperating to have the packets reach their destination. Unlike telephony, which provides a fixed bandwidth along established circuits, packet-switching networks are implemented such that they drop packets that exceed that current bandwidth due to congestion. Moreover, the distributed nature of the Internet tolerates malfunction through dynamic routing and this leads to reordered and duplicated packets.

UDP does not account for these realities, and simply sends buffers of data as datagrams to their intended destination; datagrams may arrive out of order, multiply, or not at all. The fundamental difference between UDP and TCP is that TCP is designed for robustness against the realities of asynchronous packet switching networks.

3.2.2 Transmission Control Protocol (TCP)

TCP is a protocol created to assure reliable communication through a chaotic network, accounting for packet loss and reordering [31]. It is simply and elegantly designed, and in the words of researcher and scholar Alan Kay, "a work of art"[36]. It operates in a connected context, allowing two computers to each send a stream of data to their peer; data cannot be sent over unconnected TCP sockets.

The overhead of TCP is much greater than UDP. Mitigating the realities of the Internet requires larger packet headers that communicate metadata for their stream and the state of their connection. Reliability is achieved by numbering and acknowledging all the data that has passed through the system; unacknowledged data is buffered and occasionally resent. Additional metrics are used for flow control, where receivers will advertise the size of data they can accept to encourage the sender to regulate their sending. This advertised value, called the window size, allows the receiver to indicate to the sender how much buffer space is available for new data. Senders also infer congestion that occurs through the network from the need to retransmit messages. Two algorithms, *slow start* and *congestion avoidance*, are implemented together to respectively recover from and avoid causing congestion.

The state of a TCP connection is managed by a state machine of a dozen states; transitions occur using control flags in each packet's header. The flags relevant to this thesis are the following:

- SYN (synchronization): The TCP header contains a synchronization number. This flag is used when connecting to inform the peer of the beginning of the numbered sequence of data that follows.
- ACK (acknowledgement): The TCP header contains a valid acknowledgement number. This is used to inform the peer how much data has been successfully received.

- RST (reset): The last message received was not expected. For instance, it might have been a data packet for an uninitialized connection, or contain acknowledgements for unsent data. After receiving a valid reset, the connection closes between the computers.
- FIN (finish): The sender has finished sending data. The receiver of a finish may have more data to send, so a connection can enter into a half-closed state before finally closing.

The remainder of this section details how TCP achieves reliability and congestion control.

Sequence Numbers and Acknowledgements

Reliability is achieved by having the peer acknowledge received data, and maintaining a local copy of all data that has not been acknowledged. The TCP protocol enumerates every octet (i.e., byte) of data sent over the connection; these enumerated values are called sequence numbers. When one peer sends a packet of octets, they also provide the sequence number of the first octet. Since data can arrive out of order, multiple times, or not at all, sequence numbers are used to order the received data, ignore existing data, and determine missing data. When a peer provides a socket with data to dispatch, the sequence of data is copied into a local buffer. This is used to generate retransmissions if data is lost in transit. When a retransmission timer expires, unacknowledged data is resent.

Acknowledgements are numbers sent between both peers that allow a receiver to indicate to its sender which data has been successfully received. Acknowledgements are cumulative, so a peer will not acknowledge any data that it has received while it is still waiting for data with a lower sequence number. When a sender receives an acknowledgement number, it will remove from their retransmission buffer all the data that has been acknowledged as successfully received.

Acknowledgements are included in the TCP header for all data messages sent between peers (except for the initial synchronization message). When data is being sent unidirectionally, this strategy of piggy-backing acknowledgements to data transfer is unsuccessful. For this case, TCP uses a delayed acknowledgement strategy; when a timer expires without having sent an acknowledgement then a packet consisting of only a TCP header is sent to the peer. This permits a bulk sender to empty its buffer of unacknowledged messages and avoid a congestion control mechanism that engages when a peer is unresponsive.

Congestion Control and Window Sizes

There are two strategies TCP uses to avoid packet dropping due to oversending data. Window sizes are used by the receiver to indicate how much data it can

accept, and congestion control algorithms are used by the sender to perceive network congestion.

Window sizes are used to prevent a sender from overwhelming a receiver with data beyond its capacity to handle. Overwhelming its peer with data inevitably causes the peer to drop packets, which results in poor network efficiency. Each TCP message includes a *window size* field in the header. The window size is used to specify how much more data the peer is ready to receive. The sender will only send data beyond the window size after receiving a new acknowledgement with an updated window size. Data that is written to the socket in excess of the window size will have its sending throttled. Thus, window sizes enable flow control for receivers to avoid overloading them.

A window of size zero indicates that the peer can no longer accept any more data, and the sender will stop sending data until it is told otherwise. Window size updates are sent as dataless packets, and so are not acknowledged. In case the window update is a lost packet, the sender begins a persist timer when it receives a window size of zero. When the persist timer expires, TCP sends a small packet to the peer, which triggers an acknowledgement message that includes the current window size.

While window sizes solve receiver congestion, TCP uses the congestion control mechanism to enable senders to perceive congestion in the network and throttle their traffic proactively. TCP implements a pair of algorithms: slow start and congestion avoidance. These are different algorithms with different goals, and function together to avoid packet drops. They are based on the principle that packet drops on the Internet occur almost solely due to congestion[1] and are best avoided for efficient networks. Data is sent to the peer, but the amount of data will exceed neither the receiver's window, nor the sender's current congestion window (CWND). The value of CWND represents the sender's perception of the current congestion on the network links between the peers based on recent performance. Specifically, CWND stores the number of bytes that it believes can be reliably dispatched before waiting for an acknowledgement to be returned. For the explanation in this section, we imagine CWND as storing the number of packets instead of bytes.

Slow start is the algorithm used to determine the appropriate value for CWND, and to recover after a packet drops. It uses exponential growth, where the CWND begins at one, and increases by one after receiving an acknowledgement. A single packet is sent first, and after receiving the first acknowledgement, the client increments CWND and determines the round-trip-time (RTT), which is the elapsed time for a message to be delivered to the receiver and have an acknowledgement. The sender will dispatch two packets during the next RTT period, receive two acknowledgements, and increment CWND by two (then four, eight, etc.). This exponential growth continues until a packet is dropped, at which point the CWND is halved and is stored as the slow start threshold. Slow start ends at this point, and

[1]Extensions to consider wireless and satellite networks have been since included as they suffer packet drops due to hand-offs and temporary interference.

the congestion avoidance algorithm is started. Subsequently, whenever slow start is invoked again, it will always terminate at the slow start threshold and begin congestion avoidance.

Congestion avoidance is designed to increment CWND slowly towards the optimum value. If the connection behaves properly during an RTT interval, then congestion avoidance increments CWND by one. If packet loss is detected then CWND is reset to one packet and slow start is invoked. Packet loss is detected when the retransmission timer expires [65], but extensions to TCP [32] allow a receiver to indicate missing packets to the sender. When three duplicate acknowledgements arrive that do not acknowledge outstanding data, the sender infers the next segment is missing. A fast retransmit of the missing data is performed, followed by halving the CWND and continuing congestion avoiding instead of triggering slow start.

Chapter 4

System Performance Analysis

An informal poll conducted during a rump session at the Privacy Enhancing Technologies Symposium in 2007 showed that the vast majority of attendees had previously used Tor for anonymity; however, not one of them used it for all their web traffic. The speaker put forward that this is because of the latency in Tor: the latency is sufficiently high so as to discourage casual usage. This claim was not a contentious point among the audience. The goal of this thesis is to improve the latency and throughput of the Tor network, and so an analysis of Tor's performance is the first step towards this end.

Our enquiry into sources of latency in Tor began with an examination of latency in the datapath. We instrumented the source code for Tor with timing functionality around various components of the datapath. As broad sections revealed particular sources of latency, the instrumentation was refined to measure key areas. Of particular interest are the time taken for cell processing, the time taken to dispatch messages, and the time data spent waiting in buffers. The current state of buffers and connections were also logged during execution for later examination. Finally, the packet sniffing tool `tcpdump` was used to report information about all internet traffic being generated by the running tor servers.

Experimentation was initially performed on a local machine to garner an understanding of the mechanics of Tor. However, the latency introduced by local network communications does not reflect the realities of the Internet, where hosts are separated by physical distances whose communication delay is lower bounded by the speed of light, and further slowed by routing computations. Moreover, packet switching causes reordering, packet loss, and variable delay. Our local experiments were executed through a latency proxy that simulates these realities.

To ensure that the results would generalize to Tor routers on the live Tor network, we added an instrumented Tor node to the Tor network and took measurements. The pragmatic approach of observing results, changing the instrumentation, and rerunning the program would not longer work in this setting because the operator could not conjure legitimate Tor clients immediately after restarting. Moreover, Tor nodes that maintain long uptimes are more attractive and thus encourage more

clients to make use of it. We designed and implemented a library for performing run-time system performance evaluation, called `libspe`, to achieve the goal of performing varying experiments while perserving long uptimes. After collecting data, the results were then visualized to explore the behaviour of running Tor nodes.

This chapter describes the software that was developed for system performance analysis. The `latency_proxy` is used to simulate network latency and packet dropping on a local network. The library `libspe` is used to collect timing results for variables and monitor data structures during execution. Finally, the timing server and clients are used to measure throughput and latency.

4.1 latency_proxy: The Internet on a Loopback Device

We intend on simulating a Tor network by performing experiments on a local network. For our results to have predictive powers about the online world, we need to simulate the negative realities of the Internet. We design and implement the `latency_proxy`: a small network tool to simulate latency, reordering and dropped packets on a local network.

While sounding like a trivial network program, a simple proxy that accepted packets on a TCP socket, occasionally dropped them, and otherwise delayed them for a fixed duration before relaying them would be inadequate: the operating system would acknowledge data sent to the proxy as soon as it was read. This means that the TCP stack on the original sender would assume that the packet was promptly delivered when the proxy summarily returned an acknowledgement. If such a proxy intends to simulate packet dropping by failing to forward the packet, then the sender's TCP stack would see no reason to ever retransmit; its view was that the data was acknowledged by the intended destination. Moreover, such a proxy would interfere with congestion control mechanisms by having the sender believe the RTT was significantly shorter than its intended simulated value.

Therefore we designed a proxy that would actually capture all local traffic directly off the loopback device; such a mechanism is known as a packet sniffer. Packet sniffing allows one to see all traffic, including those belonging to other users, and so requires root privileges to function. The sender, whose data is passing through the proxy, would dispatch traffic to a spurious address on which no socket was directly listening. The packet sniffer, listening to all loopback traffic, would capture traffic destined to this address, and relay it to the intended address after some delay; packet dropping would occur if the sniffer chose not to relay it. Packet reordering happens naturally when we allow variance in delay intervals.

The source code for the latency proxy will be released as a free and open-source project under the BSD license.

4.1.1 Rewriting and Relaying Packets

To relay the packet we need to infer the intended destination from the false destination. For our purposes, we use a spurious IP address for the false address, and relay the packet, without changing the TCP port, to the loopback device. The proxy rewrites the TCP/IP header to reflect this destination, and dispatches the result into the system using a raw socket. (Opening a raw socket, which allows writing arbitrary packets with preformed headers, also requires root privileges.) We observe here that the sender's TCP stack will be expecting replies and acknowledgements from the false destination it provided. We must be able to capture replies intended for the connection. Figure 4.1 illustrates an example send and reply message sequence for a new TCP connection between two peers, which we explain in detail in the next two paragraphs.

Reply addresses are managed by creating a fake address space; we call these facade addresses. When relaying messages from the connector we rewrite the packet header's destination to the intended destination and the source to a unique facade address that we generate for each incoming TCP connection (i.e., upon observing a new SYN packet). Each TCP connection that is established through our proxy has the connector's address and port maintained in a lookup table, mutually mapped in a one-to-one relationship with its corresponding facade address. In effect, this is just an implementation of network address translation (NAT).

As TCP connections use unique random ports on their connection-side address, no two connections will share a port on the same IP. (The accepting end of the TCP connection, however, uses the same port for every socket). This allows us to use the address and port of a sender to associate uniquely to both a sender's TCP stream and its corresponding facade address. The destination will make their replies to the facade address, where their peer is retrieved from the lookup table and the packet is rewritten to the form expected by the sender's TCP stack. In our experiments, we use `192.168.0.1` as our spurious address for capturing outgoing TCP packets and the address space `10.1.*.*` for generating facade addresses. All packets sent to the spurious address are forwarded to the local host (`127.0.0.1`) on the same port, and all data send to a facade address are forwarded to the peer in the managed lookup table.

The proxy was later augmented to relay UDP packets as well. This change was nearly trivial to make, basically adding a clause in the packet capturing function to consider the layout of UDP header. The proxy would observe a UDP packet destined to the spurious IP address, and rewrite the IP of the sender to be the spurious sender before transmitting. Figure 4.2 illustrates an example message send and reply sequence diagram for a UDP communication between two peers.

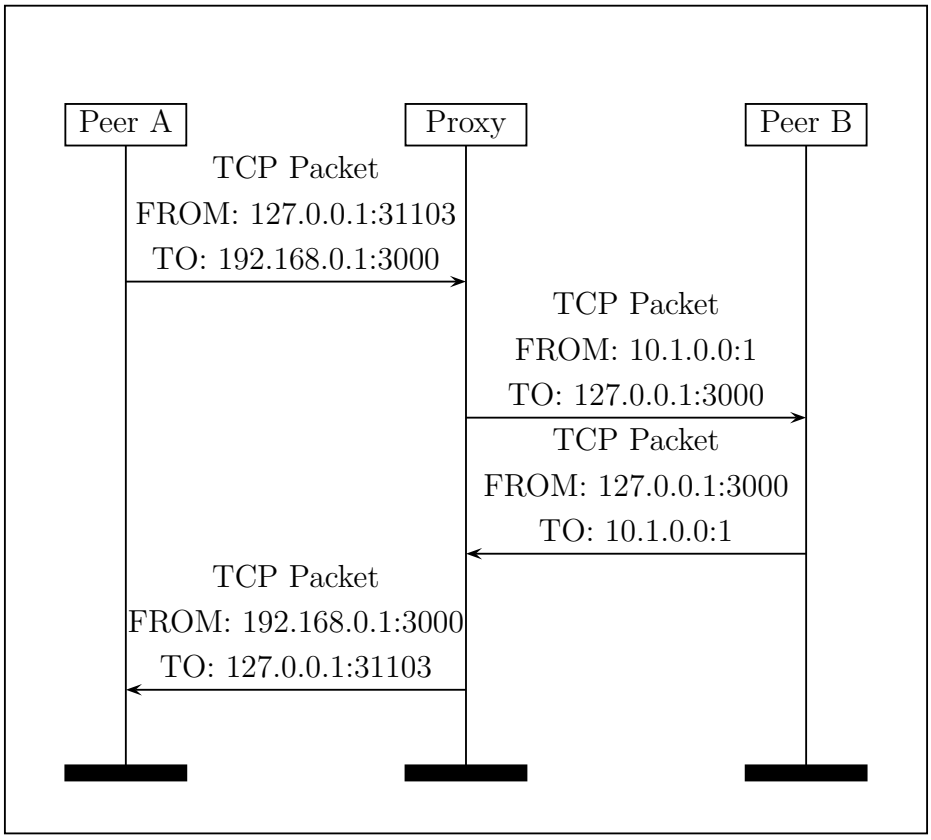

Figure 4.1: Message sequence diagram for a TCP connection over the latency_proxy.

4.1.2 Experimentation

Originally, the proxy was configured to admit a fixed latency and uniform packet dropping. However, a fixed latency ensures that all packets arrive in-order, although delayed, which is not true on the Internet. The proxy was later augmented to take delays and packet dropping from an exponential distribution about some variable mean. This allows a packet drop to occur both with occasional bursts of brief period at some times and sparsely with great period at other times.

Experiment 1 Determining RTT and packet drop rates for Tor ORs.

1: A script selected a dozen ORs whose throughput capacity exceeds 100 KB/s. This list was refined for geographic diversity.
2: A ping flood was sent to each server, using `ping -f`, for 30 seconds.
3: The program reported the minimum, average, and maximum round-trip times (RTT) and the packet drop rate.

Similarly, we use an exponential distribution for packet delay. However, the exponential distribution discretized to the positive integers poorly models packet delay directly: the probability of a delay value being selected is monotonically decreasing, so a zero delay will be the most frequent value. We compute packet

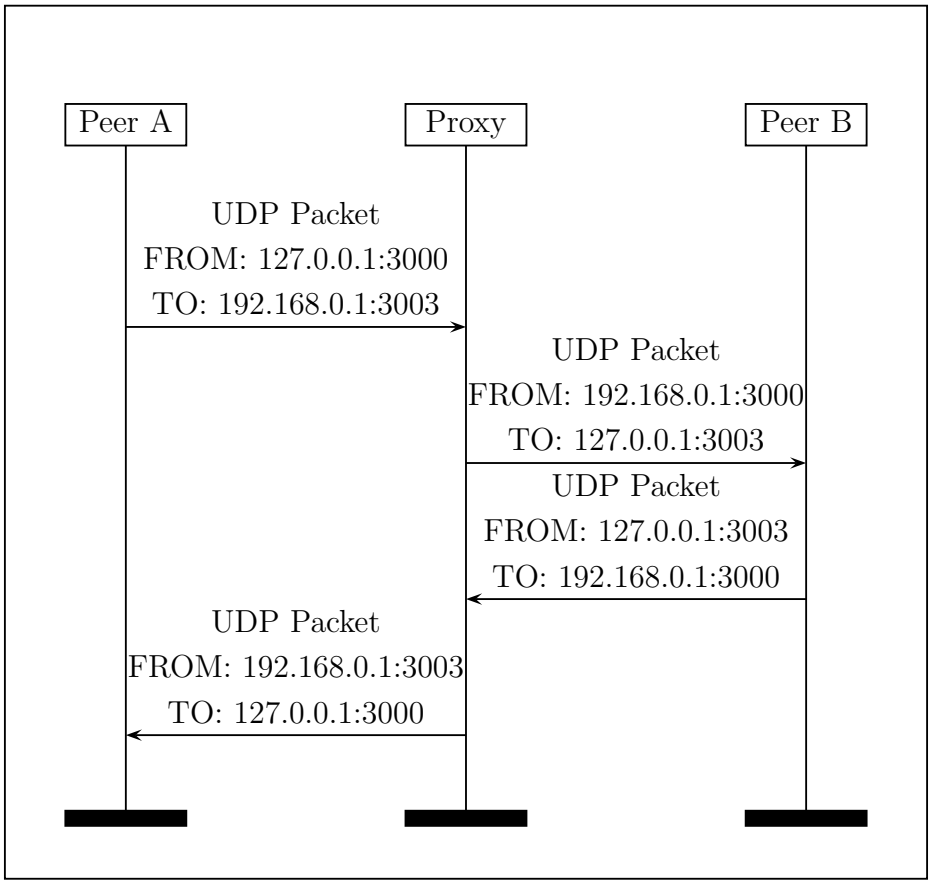

Figure 4.2: Message sequence diagram for a UDP connection over the latency_proxy.

delay using two components: a minimum delay imposed on all traffic, and a variable delay taken from an exponential distribution. This ensures that all traffic will be delayed based on the physical distance and routing complexity, with a random component that reflects congestion.

To determine suitable latency among Tor ORs, we performed Experiment 1. The results from Experiment 1 are recorded in Table 4.1. We emphasize that using a ping flood to the server results in worst-case performance for packet dropping, and conclude that the observed 1-2% packet drop rate forms an upper bound for the worst case drop rate among ORs. (In later experiments, we use a drop rate of 0.1% to explore the effect of packet dropping on Tor). The average RTT is approximately 10% greater than the minimum. This behaviour is expected: a minimum time is required by the asynchronous packet switching network between ORs, and an additional latency is caused by congestion along the router path towards the destination.

We observe that 80 to 130 ms is reasonable for RTTs, and so for our experiments we will use a unidirectional latency of 50 ms plus an exponentially distributed random latency whose mean is 10% of the minimum latency. The long tail of the exponential distribution also models the large maximum latencies that are observed,

Top-level Domain	Packets Dropped	Minimum RTT	Average RTT	Maximum RTT
.uk	2 %	112 ms	132 ms	255 ms
.de	1 %	116 ms	121 ms	256 ms
.de	2 %	122 ms	128 ms	261 ms
.at	2 %	122 ms	127 ms	190 ms
.com	1 %	45 ms	46 ms	100 ms
.ca	1 %	74 ms	81 ms	148 ms
.nl	0 %	100 ms	113 ms	414 ms
.se	1 %	133 ms	141 ms	368 ms
.es	7 %	237 ms	407 ms	550 ms
.at	2 %	120 ms	122 ms	183 ms
.com	1 %	72 ms	80 ms	238 ms
.bl	2 %	168 ms	177 ms	327 ms

Table 4.1: RTT and packet dropping for selected Tor routers.

which are generally double the minimal latency. We disregard the results for the Spanish router, as its seven percent packet dropping rate suggests some serious connectivity problems that are either temporary or will result in an OR that is rarely used.

4.2 libspe: A Dynamic System Performance Analysis Library

Our experiments began with a simple set of functions to perform timing of critical sections in Tor and help manage results. These experiments were done offline using a local Tor network for experimentation. When changes in the experiment were desired, they were made to Tor's source code and a new version was compiled and executed. However, this approach was undesirable for performing online experiments with our public Tor node `gurgle[.cs.uwaterloo.ca]`. To ensure that our Tor node provides meaningful results, we had to ensure that it need never be restarted. A long-lived Tor node is a valuable node in the network, and therefore would be selected by Tor's circuit building algorithm more frequently. Our offline experimental procedure would have necessitated restarting the router each time a change in the experiment was required, which was deemed infeasible.

The goal is to allow the programmer to specify all the data structures that they wanted to monitor before compiling. For instance, in Tor we want to inspect connection objects and buffer objects, and so we want to notify our library of the creation and deletion of these objects. At important parts of the program, such as reading from a buffer or writing to a connection, we add a call to our monitoring code that will call a particular callback function depending on the data structure

type. We need to be able to control whether the callback is invoked (i.e., perform an experiment), and control the actual callback function (i.e., change the experiment): these must both be able to be done dynamically.

The solution was to design and implement `libspe`: a dynamic system performance analysis library. It allows for a program to register any number of *observees*: a reference to some object that is being monitored. Each observee belongs to a *family*; for our purposes each data structure type is a family and observees in that family are instantiations of that type. Each family has a method that is invoked for observations, and so when an observation is made on that object, `libspe` invokes the corresponding method passing the observee. To allow for dynamic changes in the observation methods, we maintain a dictionary that maps families to their observation methods, and allow this to be easily modified at runtime.

The source code for `libspe`, written in C, will be released as a free and open-source project under the BSD license.

4.2.1 Static Data Collection

A set of static data collection variables are defined at compile time. These correspond to regions of the source code under timing scrutiny, a particular line of code whose execution period is being computed, or any particular variable whose value the operator has decided to record. Each variable is collected under both a family name and a subindex. For example, each buffer has a unique pointer value in memory, and so buffer sizes were recorded under the buffer size family subindexed by the buffer's memory address. The subindex value of zero is used to store either generic, amalgamated, or nonindexed variables.

The memory allocated to store the results during execution is currently configured at compile time. Each variable can either be a linked list of unbounded size, or a fixed-size array. If a variable uses a fixed-size array, then observations are collected by accepting new measurements with a decreasing probability in order to collect a random set of data distributed uniformly on a stream of unknown length. Our Tor node used a fixed-sized array of 1024 observations per non-zero subindex and 4096 observations for the zero index.

4.2.2 Interaction Socket

When initialized, `libspe` is provided the port number for a local interaction socket. A thread is spawned that accepts localhost connections on that port. All runtime aspects of `libspe` are controlled through this socket. Foremost, static data collection can be toggled, and the cumulative distribution functions for all the static timing variables previously collected can be output to a file. Since observations on data structures are initially disabled, the interaction socket allows the listing of all the observees currently being managed and the enabling or disabling of observation for specific observees or entire families, along with the filenames used for

results. Finally, the callback library used to collect data during an experiment can be changed with a command specifying the new callback library.

4.2.3 Observers

Registration of candidate data structures for observation is made at compile time. Each object registers itself with the `spe` instance when in use, and deregisters itself from the `spe` instance when it is no longer being used. Tor was modified to register objects in their constructors and deregister in their destructors. While generic observations are made in discrete intervals, `spe` can also be forced to make observations at particular places in the program (e.g., observing on a socket before writing or observing on a buffer before an insertion). Each observee is assumed to be in an *off* state when the program begins execution; `libspe` will not generate any data for observers that are *off*, even if they force an observeration. At runtime, through the interaction socket, the operator can turn observees *on* to generate data, either individually or by family. Each enabled observer is given an open file for outputting data, and an observation time period.

Static state is maintained between observations. Each observee has an associated state object that can be used in the observation method; the state is passed alongside the observee when making observations. When an observee is registered, a special invocation of the observer method indicating initialization is performed so that state data can be initialized. Similarly, when an observer is disabled, another special cleanup invocation is performed to signal that the observer must free allocated memory. Thread safety is assured by having each observee acquire a lock before before calling the observation method and release it after the observations method returns.

4.2.4 Dynamic Callbacks

Dynamic callbacks are used to collect data from data structures at runtime. Since the data the operator may wish to collect may change as the program executes and collected data is examined, `libspe` allows dynamically loaded libraries of observation routines to obviate restarting the instrumented program due to a change in experiment.

The set of callback families must be known prior to execution. Each observee that registers specifies its family (i.e. data structure type) upon registration. Each family is associated with a specific observation method that is invoked in the library. Initially, there is no library and so all families are associated with a `null` function. When the interaction socket loads a new library, each family's associated function symbol is loaded from the new library. Henceforth, all observations will now invoke this new method.

As an example, suppose the operator wants to report the size of a buffer over time. They compile a library with an observation function that takes a buffer as

a parameter, and writes its size to a file. The operator connects to the interaction socket, informs `libspe` of its observation library, and enables observations on all buffers. Suppose after determining the sizes of every buffer, they find one buffer which they wish to explore in more detail. The operator then writes a new method to report the contents of the buffer, changes in sizes over time, the memory allocated for the buffer, etc. This new method is compiled into a new libray, and `libspe` is told (via the interaction socket) to use this new library for experimental callbacks henceforth. The operator then enables observations on the single buffer of interest, and `libspe` will use their new method to report more information.

4.2.5 System Interface

The system interface for `libspe` contains two components: the static API used during the instrumentation of the program, and the runtime interface used to control experimentation while executing.

The instrumented program initializes `libspe` with `initialize()`, which configures data set sizes and the local port used for the interaction socket. Data structures that are to be inspected are registered with the `register()` method. The functions `start_timing()` and `stop_timing()` are placed around code to respectively start and stop the timer. The elapsed time is computed when stopping the clock, and is stored locally using the `data_point()` function. To store a single piece of data, such as the current size of the buffer, one can call `data_point()` directly. Finally, `period()` is used in lieu of `start_timing()` and `stop_timing()` to measure the time elapsed before the program counter returns to the same line of code. It computes the difference in time between now and the time it stores locally. It replaces the stored time with the current time, and adds the computed difference using `data_point()`. Since there is no initially stored value, the first call is ignored but all subsequent calls compute the period properly. Like all the other methods, `period()` is indexed by both a family name and a subindex, allowing for expressive period calculations based on the current system logic.

At runtime, an operator can connect to `libspe` through the interaction socket to control its behaviour. The listening thread will spawn a new thread upon accepting a connection, and the new thread will respond to operator demands sequentially. As mentioned, these include changing the callback libraries, enabling observation, and outputting the static data that has been collected. Resources are shared between the `libspe` static program interface and its dynamic socket interface, and all data access and function calls are threadsafe.

4.3 Timing Client/Server

Our enquiry into improving Tor will measure the end-to-end latency and throughput of traffic through circuits. To reliably measure these statistics, we wrote a

simple timing client/server application to execute through Tor. The server runs two threads: the listening thread and the writing thread. The listening thread accepts new connections and add them to a linked list. The writing thread will repeat sending a timestamp message to all accepted sockets that are writable. We `select()` (an operating system call that takes as input a set of file descriptors (FDs) and returns the set of FDs that are currently ready for an I/O operation) for writeability on the list of sockets to ensure that our server application will never block while trying to write. If no socket is writable then we sleep until one is ready. In our experiments, the timing server always sleeps while sockets are unwritable and so the timing server's computations do not interfere with our results.

Sending a constant stream of data is used to measure throughput, so the timing server also runs another listener to measure latency. When it receives a connection, it also provides the connector its timestamp, and closes the connection immediately. The difference in the timestamp and the time it was received is the directional latency of the circuit.

The client will connect to the server, which then starts continually sending timestamps. The client reads as much data possible, computes the difference between the current time and the timestamp, and then discards the read data. We use latency from the server to determine when the system has reached steady state; the delay for the stream of timestamps will continue to increase until it reaches its natural value—a function of capacity and throughput.

We begin to collect data when the system reaches steady state. The client computes throughput by counting the bytes received from the socket. The client also measures the delay by connecting to the server that only returns a single timestamp instead of a bulk stream. This gives the delays incurred by the network along the return direction when the network has reached a steady state with the data from the bulk streams. Average latency and total throughput are computed each second, logged for later processing, and then reset for the next second interval.

Chapter 5

Latency in Tor's Datapath

In Section 3.1 we described the operation of the PET Tor. Briefly, it is a set of onion routers (ORs) through which circuits are built to relay Internet traffic; the movement of data through the network aims to obfuscate the source and destination from observers, affording the user with anonymity. The datapath for Tor refers to the sequence of locations where data resides while travelling between the source and destination. Figure 5.1 shows the simplified datapath; data goes from the OP, through each OR on the circuit, to the server; replies travel back through the circuit to the OP. This chapter closely examines the datapath for Tor, searching for sources of latency. In particular, we explore the datapath inside each OR.

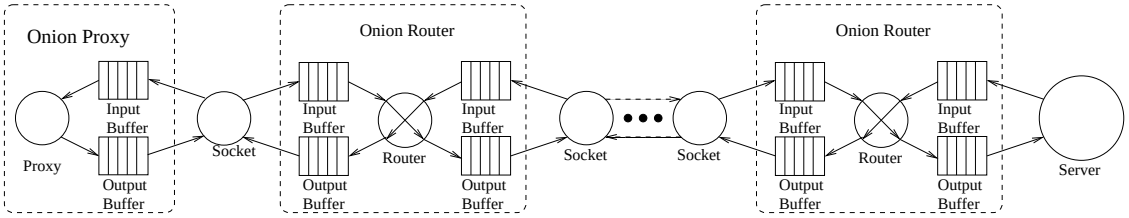

Figure 5.1: A circuit datapath in Tor.

In decisions pertaining to the system, the authors of Tor seem to be concerned that the computations involved in the operation of a Tor node results in significant sources of latency. For instance, they omit integrity checks at each hop [16], opting to check the integrity of the packet only at the end (ostensibly to increase the routing speed). They also omit one unnecessary encryption operation at the client side [16], because the additional TLS encryption at the transport layer obviates Tor's outermost layer of encryption. While these improvements certainly decrease Tor's processing time, reduce its power consumption, and improve its efficiency, it remains to be determined if these optimizations have any significant impact on the router's execution time or if a different bottleneck exists that limits the throughput of the system.

Profiling results using the tool `gprof` are referenced in Tor's original design papers [16]. Profilers such as `gprof` allow the program to execute normally but will

pause its execution at frequent and even intervals to probe the program counter; this observes in which function the program is currently executing. Probing provides a sample of execution time that it uses to build a report informing how much time is spent executing each function. It also reports the call graph, which traces function calls by their parents and records how often a function is called. Profiling is used to diagnose slow programs and fix bugs (unexpected results often indicate a logic error). Tor's profiling results indicate that the majority of its execution is spent performing encryption operations, an unsurprising result that was confirmed with our own experiments. However, profiling metrics have limitations in system performance evaluation, as they do not consider latency caused by inter-thread communication and condition waiting. Examples include data queued in buffers waiting to be processed and the communication of data between multiple programs over a network. This limitation of profiling forms a natural entry point into our enquiry into Tor's latency. Instead of examining latency from the program's perspective as it executes, we will examine it from the data's perspective as it travels through the datapath.

This chapter begins with experiments to investigate the following two hypotheses:

- There is significant latency beyond transport latency in Tor.
- Cell processing, including encryption, cannot represent the largest source of (non-transport) latency, because there are significant sources of latency inside the datapath that are not uncovered by profiling.

Our first hypothesis is tested by contrasting the overall latency to the expected transport latency. We use this ratio to make an estimate of the communication latency overhead. Our second hypothesis is tested by computing the time required for cell processing. This forms an upper bound on its processing latency. We show that this processing latency does not represent a significant amount of the overall latency.

The remainder of the chapter is devoted to isolating where latency occurs inside Tor. We determine that congestion control mechanisms are responsible for latency that is not uncovered by profiling. While we cannot circumvent congestion control to achieve better performance, Tor uses TCP unwisely by multiplexing circuits over a single TCP stream. This permits congestion control, packet dropping, and packet reordering to cause undesirable cross-circuit interference.

Two machines were used for our experiments. An Intel Pentium 4 3.00 GHz dual core with 512 MB of RAM was used for experiments performed on `gurgle` (our live Tor node), running modified version of Tor 0.1.2.18. A commodity Thinkpad R60—1.66 GHz dual core with 1 GB of RAM—was used for our local experiments. Care was taken during experimentation to ensure that the system was never under load sufficiently significant to influence the results. Experiments on our local machine used a modified version of Tor 0.2.0.25-rc. The difference in versions is because our

local experiments occurred a year after we deployed our public Tor node, and we wanted to be able to deploy our changes to the real Tor source code.

5.1 The Impact of Transport Latency

Transport latency is a necessary and so unavoidable component of the latency in Tor's datapath. Upon first consideration of the mechanics of Tor—routing one's traffic about the world and back—the observed latency can be speciously attributed to the delays incurred by the additional Internet exchanges added to the client's datapath. Indeed, the greatest latency is mandated by our current understanding of the physical world. We perform an experiment to determine an upper bound on the transport latency, and contrast this to an observed latency so as to determine the overhead latency. Experiment 2 lists the steps we followed to determine what proportion of total latency is transport latency.

Experiment 2 Determining the overhead latency for Tor.

1: An `apache` webserver was initialized on `gurgle` to return `HTTP error 404` to requests for the root directory. This is done to ensure that HTTP replies require only one cell in Tor to transport.
2: A public Tor node (OR) was run on `gurgle` for a couple days to accrue traffic.
3: A Tor client (OP) was run on `gurgle`. It used a circuit of length three. The first and last nodes were selected randomly but the middle node was forced to be the `gurgle` OR.
4: The client was instructed to retrieve the webpage from apache on `gurgle` (which will return error 404) and the wall clock time was recorded.
5: The RTTs from `gurgle` to both the randomly chosen ORs in the circuit were computed by pinging the ORs and averaging a dozen results.
6: The time to dispatch, service, and interpret a web request was computed by retrieving the webpage from `gurgle` without Tor. All network latency was removed by performing this step locally on `gurgle`. We call `time wget 127.0.0.1` to precisely record the time required to satisfy a request. The times for a dozen measurements were averaged.
7: The experiment was performed six times using randomly selected ORs for the first and last hops each time.

Figure 5.2 shows the datapath for Experiment 2 and illustrates the times we are measuring. Our experiment sends traffic to OR A, then to `gurgle` as the middle node in our circuit, then to OR B, and returns to `gurgle` as the intended destination. The reply follows the same path in reverse. The transport time for the first two messages, which travel from gurgle to OR A and back, will be approximately the RTT for OR A. Similarly, the network time for the next two messages will be equal to the RTT for OR B. Since the reply will follow the same path in reverse, we have that the expected total time spent transporting the request and reply cells

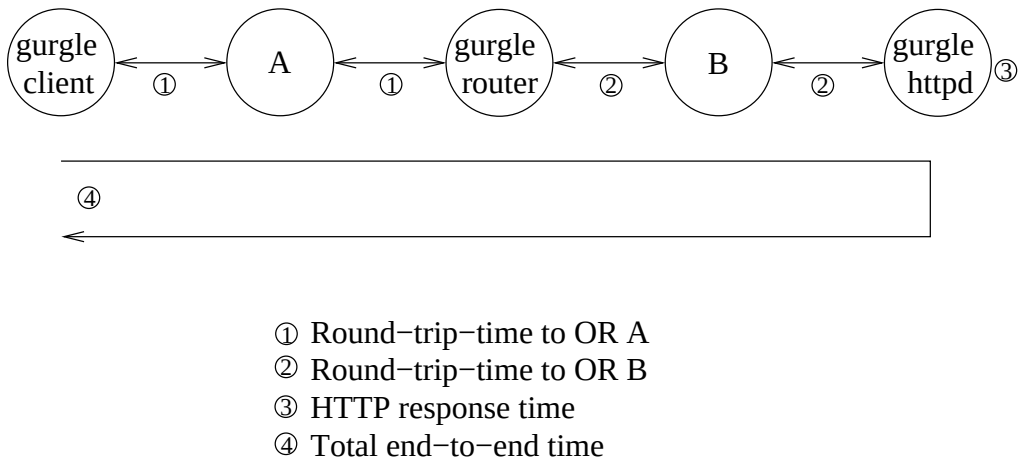

Figure 5.2: The datapath for Experiment 2.

between the machines is lower bounded by twice the round trip times for the first and last nodes plus the overhead of fetching the webpage. Our hypothesis is that the total time required to fetch the webpage over Tor will be greater than twice the round-trip times for each OR and the service time, and that the difference represents the additional latency introduced along the Tor datapath.

Before presenting the results, we first explain the motivation for the web-server returning an error, along with a misconception about Tor that the above experimental design exposed. The goal was to measure, as strictly as possible, the transport latency of various Tor circuits without concern for throughput. Both an `HTTP` request and the `HTTP error 404` reply can fit in a single Tor cell, which is an atomic form of data transfer in the Tor network. By ensuring only the time required to transmit a single cell in each direction is measured, we protect our experiment from measuring additional latency necessitated by the transfer and processing of multiple cells. However, it was quickly apparent that network latency would never account for a majority of the wall clock time, a counterintuitive result that suggested an experimental error. By having the `gurgle` OR report its traffic over our circuit, it was determined that each request required five cells of data. Two correspond to the upstream and downstream flow of data, but the remaining three are used by Tor to control circuits.

When a user makes a new TCP connection to a web server over Tor, they begin by signalling to the exit-node on their circuit that they wish to begin their connection. The exit-node executes a TCP handshake with the server (which introduces a single RTT of latency) and replies with the result to the client. If successful, the client proceeds with the data request. So first a new TCP stream is initialized, then the request is made, then the reply is sent, and a final cell indicates that the http server has closed the connection. (The last two messages travel from server to client in parallel.) The message sequence diagram for our experiment is presented in Figure 5.3. We observe that if we combine the first two messages, BEGIN and DATA, from the client to the exit node then we have an effective way to halve

1st Node RTT	2nd Node RTT	Transport Time	Total Time	Overhead Time	Overhead Proportion
126 ms	44 ms	684 ms	750 ms	67.5 ms	9.0 %
126 ms	116 ms	972 ms	1100 ms	130 ms	12 %
111 ms	109 ms	884 ms	1280 ms	398 ms	31 %
119 ms	77.6 ms	790 ms	845 ms	56.5 ms	6.6 %
180 ms	186 ms	1468 ms	2100 ms	634 ms	30 %
116 ms	236 ms	1412 ms	1690 ms	280 ms	17 %

Table 5.1: Transport and overhead latency in Tor. Each row uses a different circuit.

latency whenever a new TCP stream is established. TCP streams proliferate in casual webbrowsing since hypertext anchors are used to navigate quickly between servers, and images and other content are often referenced from separate servers.[1] Each of these operations requires a new TCP connection, which incurs a needless doubling of latency before beginning to receive the HTTP content. We propose an improvement based on this observation in the future work section.

Table 5.1 summarizes the results of experiment 2 to three significant digits, and also presents the time spent in overhead both in total and as a percentage of the total time. The webpage retrieval, a constant 1.5 ms, was included in the transport time but omitted from the data. We note that the overhead columns are corrected for the serialized BEGIN and DATA flows illustrated by figure 5.3.

This experiment shows us that while the transport latency caused by the sluggish speed of light is the most significant source of latency in Tor, we can still improve the user's experience by examining latency caused inside Tor routers. Our experiment shows that the overhead is subject to high variance, a result that has been observed by Wendolsky et al. [76]. We observe that a significant amount of latency in Tor occurs outside of network transmission. A possibility is that the cost of performing encryption operations and other cell processing overhead incurs significant latency; this is refuted formally in the next section. (One can fathom that if encryption were the source of latency then there would be less variance in overhead since each run of the experiment executes the same number of encryption operations on the same amount of data.) The remaining sections of this chapter will investigate the sources of this additional latency.

[1] As an example, many websites include a script to have the webbrowser dutifully connect to Google and register their access as part of their analytics project. The privacy-aware disable this inside their webbrowser, and perhaps it would be wise for exit-nodes to prevent this needless loss of privacy. Indeed, it might be mutually beneficial as usage patterns from Tor nodes add noise to their analytical metrics.

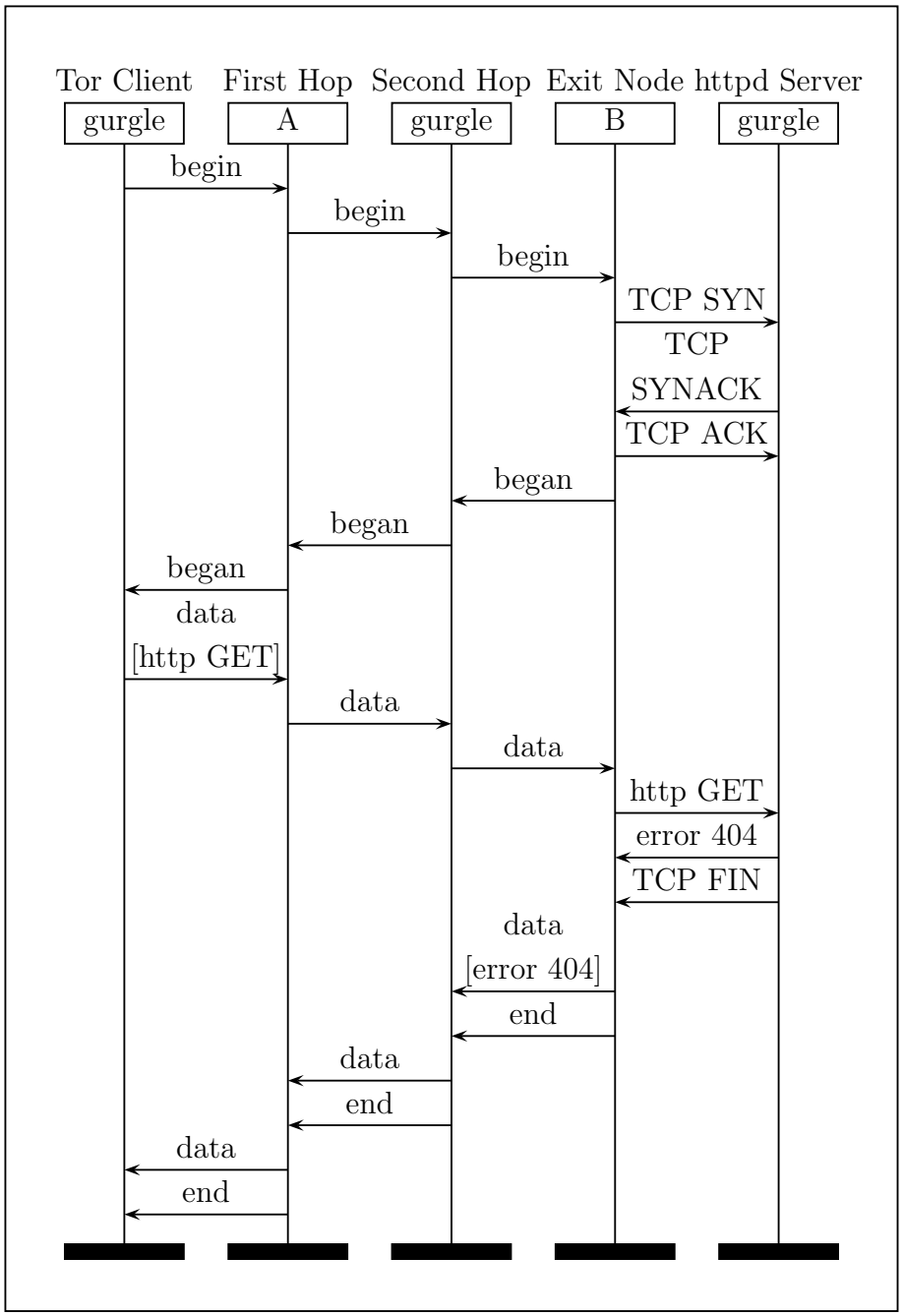

Figure 5.3: Message sequence diagram for Experiment 2.

5.2 Latency along the Computational Datapath

We have shown that Tor introduces significant latency along the datapath beyond the expected network latency. This section will consider the computations that occur at each Tor router along the circuit, and determine the latency that is introduced by their execution.

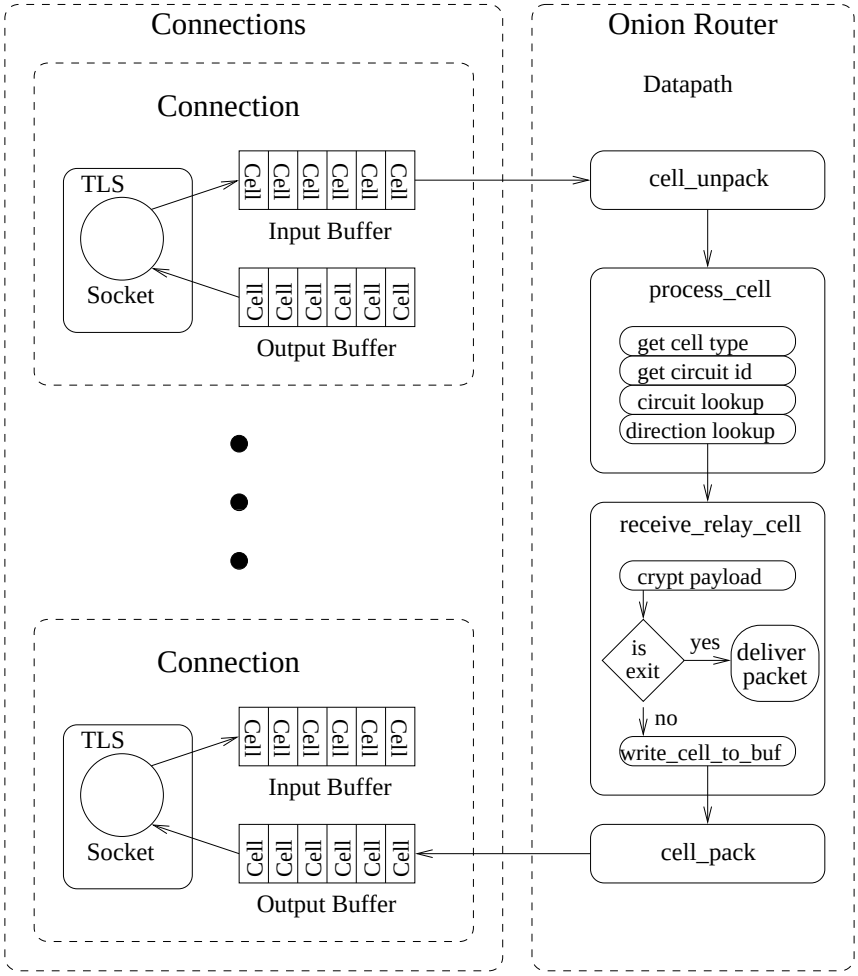

Figure 5.4: Simplified datapath for cell processing.

Tor uses fixed-sized cells as the basic object of communication. Figure 5.4 shows the processing datapath that occurs at each node, which corresponds to the "router" step in Figure 5.1. Data is read from a socket and placed on an input buffer. From there, cells are unmarshalled, processed, and marshalled onto an output buffer where they are later dispatched to a socket. Our enquiry is focused into the cell processing time for a single node. Circuit construction, which requires expensive public-key cryptography operations, is not considered for latency since Tor preemptively builds circuits so as to avoid impacting user experience.

Cell processing occurs entirely in a single flow of execution. First, it calls the method `cell_unpack()` to unmarshall a single cell from the input buffer into a

`cell_t` structure. The `cell_t` is then processed; the cell's circuit and direction are retrieved, and the payload is encrypted or decrypted accordingly. Finally, the cell is marshalled onto an output buffer using the `cell_pack()` method. A different execution flow removes it from the output buffer and dispatches to the appropriate socket. All the responsibility of the router on processing incoming data occurs between the `cell_unpack()` and `cell_pack()` functions, particularly the circuit lookup and adding or removing a layer of encryption. Another thread removes data from the output buffer and writes it out to the socket over TLS.

We know from profiling that Tor spends most of its time executing AES operations during cell processing. Experiment 3 was performed to investigate the actual time required to process a cell.

Experiment 3 Determining the processing latency for Tor.

1: The source code of Tor was instrumented for measurements with `libspe` and the following modifications were made to time cell processing precisely:
   - Cell structures were expanded to include a time value field.
   - When a cell is unmarshalled, the current system time is stored in the cell's time value field.
   - When a cell is marshalled, the difference between the current system time and the stored system time is recorded.

2: The `gurgle` Tor node was set to run for a couple of days to build traffic.

3: Data for the `processing_time` variable was collected for a couple hours for all circuits.

Figure 5.5 shows the graph of the cumulative distribution function for cell processing timing collected from Experiment 3. We emphasize that the time scale is in microseconds, i.e. millionths of seconds. The time Tor takes to process 90% of cells is between 8 and 10 microseconds. For a circuit of length three, we would see six cell processing sequences for a query/response in transit and a further six AES operations performed client-side.

The additional use of TLS to encrypt outgoing traffic between nodes will increase the overhead. Timing results inside OpenSSL's read and write methods indicate that reads require 30 microseconds and writes requires 40 microseconds. These operations, plus cell processing, yield a potential throughput for our test machine of over 6 MB/s, more network bandwidth than the average OR is willing (or able) to donate. Since there are three TLS links in our circuit, the use of each requires both an encryption and decryption operation, we expect 70 microseconds of computation per TLS link. There are six TLS links in the path, yielding 420 microseconds of TLS latency for a cell to travel up the path and another cell to be returned in reply.

Therefore the expected computational latency along a circuit is 540 microseconds for a full trip. A generous upper bound for the computation time required to process the five cells in our experiment would be two milliseconds. This is an order

of magnitude briefer than either the duration we have observed or any duration that is humanly perceptible. Experiment 3 tells us that the lion's share of latency exists elsewhere.

5.3 Queueing Latency along the Datapath

In the previous sections we have established that Tor introduces latency into the datapath. We have discounted the obvious transport latency from the sum, and eliminated computational latency as a cause. In this section we explore the Tor's datapath as a queueing model to determine sources of latency.

Figure 5.4 shows that there are two types of buffers associated with each connection: the input buffer and the output buffer. Each connection between a pair of routers has an input and an output buffer on both ends. When data arrives via a connection from another router, the data is placed on the input buffer. The `cell_unpack` function removes the data from the buffer, and the `cell_pack` function places the processed cell on the appropriate output buffer. Reading and writing operations are executed on cells in a round-robin manner across input and output sockets, with a scheduler that controls execution. When processing cells on an input buffer, Tor processes all available cells before processing another connection, but it does not perform any socket writes in the same flow—it places the processed cells on the appropriate output buffers and indicates to the scheduler that data is ready to write to a socket.

Writing to sockets is scheduled alongside reading from sockets, and so is performed in a round-robin manner when data is available. Tor removes the data from the output buffer, encrypts it with TLS, and dispatches it over the appropriate socket. The time that data waits in an output buffer before being dispatched is a potential source of latency, and in particular, a source of latency that will not appear during profiling.

To investigate this latency, we observe each buffer's size whenever data is added to or removed from a buffer, along with the current time. Knowing the actual buffer sizes over time, and that Tor buffers operate with first-in-first-out (FIFO) semantics, we can simulate data queueing and travelling through the buffers after execution to determine how long each byte of data spent waiting. Experiment 4 was performed to investigate the time data spends waiting in buffers.

We plot the results of this experiment—the time data spent waiting in buffers—as a CDF and as a graph over time for every buffer. We present our results first for input buffers, then for output buffers.

Experiment 4 Determining the latency of Tor's buffers.

1: Tor was instrumented to report both the time and the size of a buffer whenever data was either added or removed from a buffer. This was done for every buffer allocated in memory.
2: A public Tor node was run on `gurgle` for a couple of days to accrue traffic.
3: Tor was instructed to begin logging buffer sizes every time data was added or removed.
4: A script processed the changes in buffer sizes to simulate the buffer's queueing model. Whenever data was added into a buffer, it was tagged with the entrance time. When it was removed, the elapsed time was recorded.

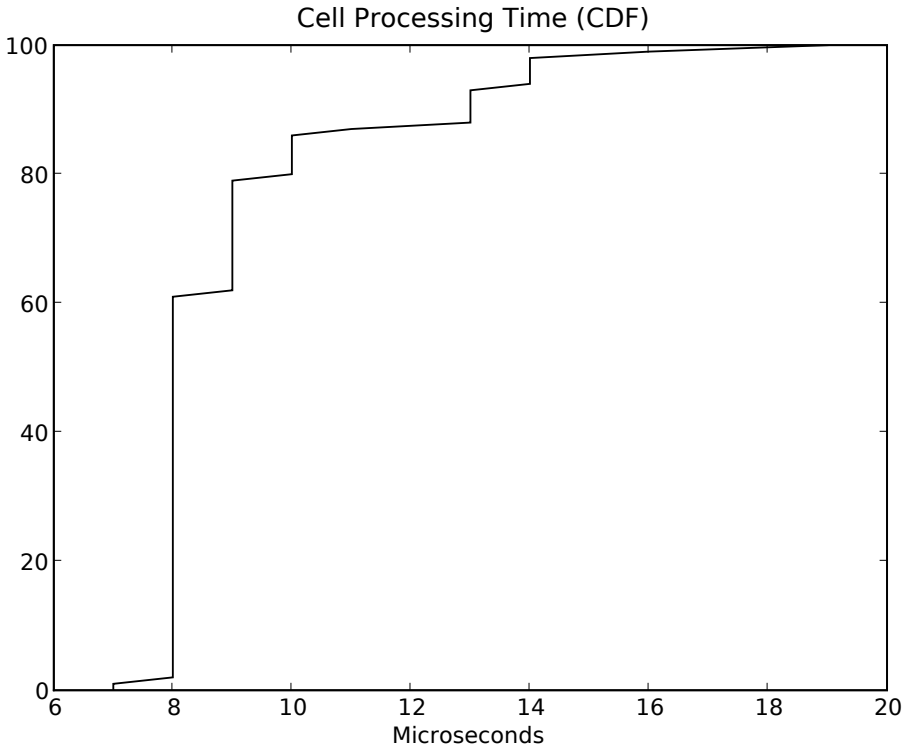

Figure 5.5: Cell processing time reported from Experiment 3.

5.3.1 Input Buffers

Input buffers are where Tor places cells it has read from the socket before they are processed. It does this for cell-alignment purposes; the TCP stream does not send packets aligned to cells and so partial cells will reside on the buffer until the remainder arrives. Figure 5.6 presents the results from one particular buffer, and we show in Figures 5.7 and 5.8 that these results are representative of all input buffers. Figure 5.6(a) shows the latency for data in an input buffer, graphed over time. Figure 5.6(b) shows this data as a cumulative distribution function. The waiting time is effectively negligible for all data that entered the buffer, approximately 5 microseconds. This is easily accounted for by the time it actually takes to copy data into and remove it from the buffer, and so we hypothesize that input buffers are not the cause of the observed latency in Tor. Infrequently, we see spikes in the waiting time; however, the longest delay is still less than a hundred microseconds. Figure 5.6(c) shows the input buffer size graphed over time, and Figure 5.6(d) shows this data as a cumulative distribution function. The occasional spikes in buffer size occur simultaneously to spikes in Figure 5.6(a), which shows the latency introduced in the buffer. Since Tor reads all available packets from a socket onto the input buffer, the input buffer will surge in size but will then be emptied as the data is processed. Despite the surges in buffer size, the associated delay is insignificant.

To prove the representative nature of the input buffer discussed above, we present a partition plot of the distributions of over a hundred input buffers in Figure 5.7. The X-axis corresponds to increasing percentiles; the data on the left side of the chart are the minimum measurements and the right side of the chart are the maximum measurements. The Y-axis corresponds to the set of all the buffers being graphed. A vertical line segment whose length is 10% of the total height would express that 10% of the buffers have their measurements represented in that line. The actual data is conveyed by the partitioning lines that separate segments in the body of the plot. The partition lines correspond to a variable crossing specific values, such as contours lines on a relief map. The height of the span between partition lines indicate what percent of buffers have their variables's percentile measurement fall into the partition range. For input buffers, we see that effectively all buffers for all percentiles fall into the range of waits less than 10 microseconds. Figure 5.8 shows the data in Figure 5.7 zoomed into the high percentiles. At very high percentiles we can see some delay on input buffers, but the size of the delay is still negligible and their occurrence is rare since they only occur in the highest percentiles.

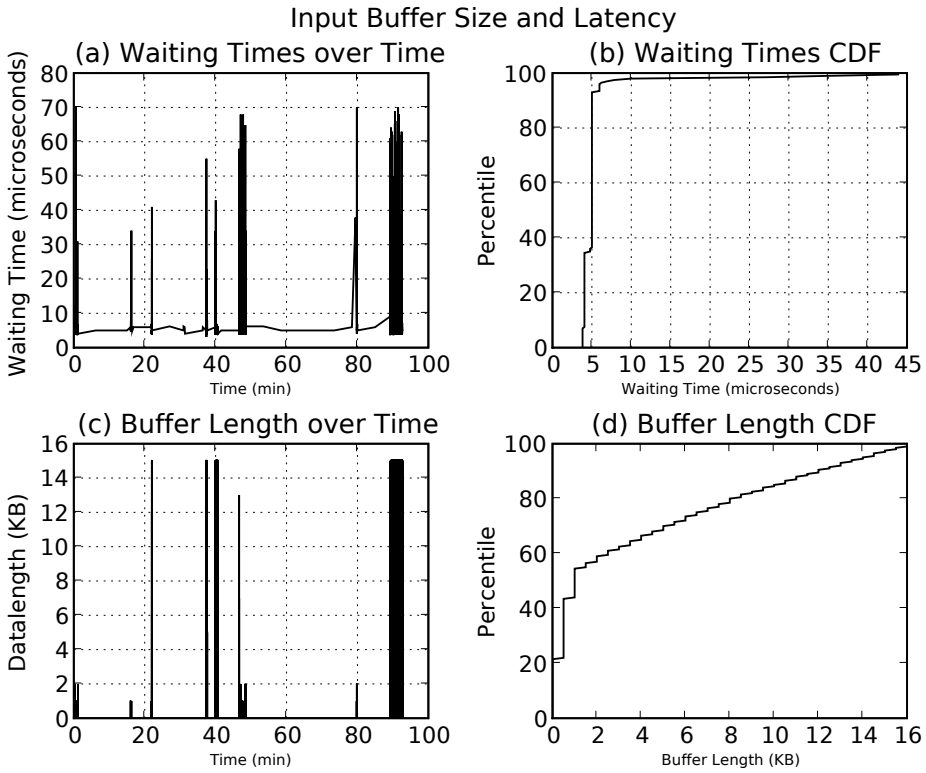

Figure 5.6: Waiting times and data sizes for a representative input buffer. (a) shows the time that data spends waiting in the buffer over time. (b) shows the time data spends waiting in the buffer as a cumulative distribution function. (c) shows the buffer size over time. (d) shows the buffer size as a cumulative distribution function.

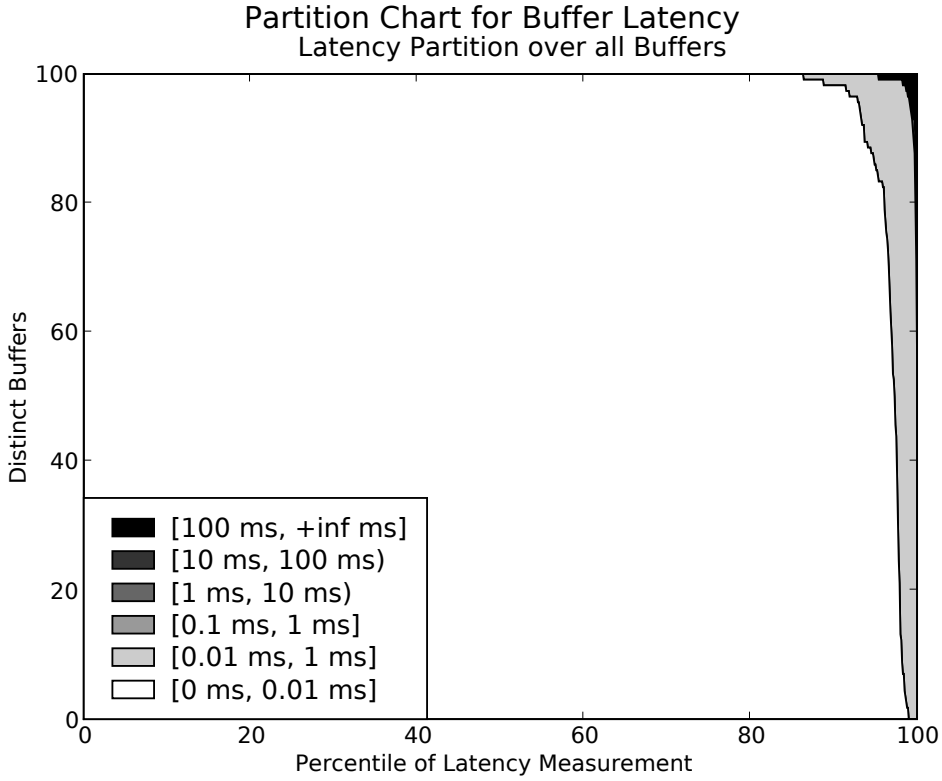

Figure 5.7: Partition diagram for input buffer latency across many buffers. White corresponds to buffer latencies less than 0.01 ms. Black corresponds to buffer latencies greater than 100 ms, and each darker shade represents a ten-fold increase in latency.

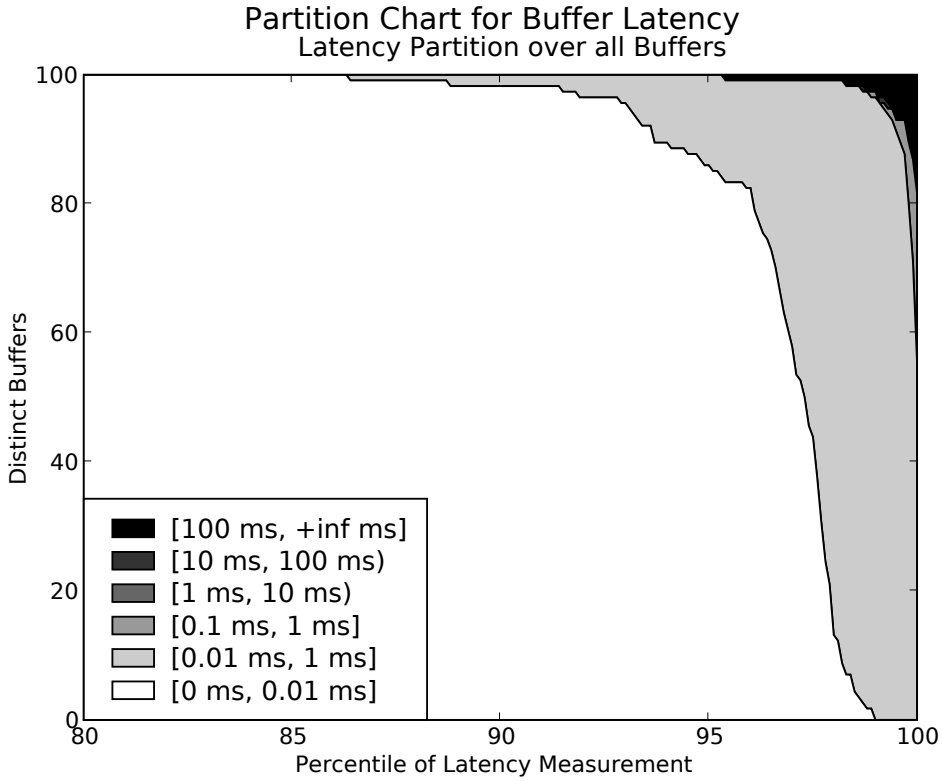

Figure 5.8: Zoomed area of interest for Figure 5.7.

5.3.2 Output Buffers

Output buffers are used in Tor for data that is ready to write to the socket; i.e., cells that have been processed and are waiting to be dispatched. Output buffers occasionally swell with data, resulting in longer wait times before being dispatched.

In our examination of hundreds of buffers, nearly every output buffer had a period of time where it grew significantly, however it was usually for a very brief time. The buffer would suddenly surge in size and hence, albeit briefly, the latency before dispatching the buffered data increased; Figure 5.9 is an example.

Other categories of buffers have other characteristics: some have systemic connection problems that result in enormous sizes and high latency, while others serve relatively little data and are not a source of latency. These are likely due to vastly different attributes of the connections between the two peers on either side of the circuit. However, the buffer in Figure 5.9 is illustrative of the standard case we are addressing—a buffer that serves a lot of data and occasionally introduces significant latency. The periods of latency occur with varying frequency for each buffer.

Thus, we imagine an efficacy slider for buffers. At one end we have buffers whose data is removed and dispatched immediately with arbitrarily small delays, and on the other end we have buffers whose data must wait arbitrarily long before being processed. Figure 5.10 visualizes the distribution of buffers based on the time spent waiting. Each partition represents a discrete range of buffer delay, the X-axis corresponds to increasing percentile of the buffer delay, and the Y-axis corresponds to the space of all buffers we are examining. We emphasize that each increasing partition represents a ten-fold increase in expected latency—partitions are distributed logarithmically.

By observing the two ends of the graph, we see that the expected performance of every buffer varies throughout execution. The right end of the chart (which corresponds with the highest percentiles of wait times) indicates that all buffers encounter some period of increased latency. The great majority of buffers experience a data latency greater than 100 milliseconds at some point during their execution. Similarly, most buffers have negligible delay for another period of execution, as indicated by the leftmost part of the graph. This would be periods when data enters the buffer and is promptly removed and dispatched to the appropriate socket.

More importantly, there are two ranges of latencies in which most buffers fall: the range of 0.1 to 1 milliseconds and the range of 100 to 1000 milliseconds. These two ranges, three orders of magnitude apart, account for nearly all the buffer delays observed. Clearly the range of 0.1 to 1 millisecond is of no concern; these buffers introduce negligible latency into the datapath. The other group of buffers that introduce up to one second of latency may account for a great quantity of the observed latency in Tor. Since the total expected RTT for delivering traffic through the network is less than a second, a comparable latency in these buffers is of necessary concern.

One may be tempted to deduce that the two categories of buffers are related to their relative throughput: buffers that fall into the rapid category simply transport less traffic and so are less liable to become burdened, while the buffers that introduce latency do so simply because they are overburdened with traffic. In reality, while increased throughput results in a slight tendency towards slower buffers (as would be expected) any conclusive partitioning of buffer latency with regards to throughput is specious; Figure 5.11 substantiates this argument. Figure 5.11(a) is a partitioning of all buffers, and the remaining subplots correspond to the latency-based partition of buffers only for particular average throughput ranges. Specifically, Figure 5.11(b) is the partitioning for buffers whose average throughput is less than 1 KB/s, Figure 5.11(c) is the same for average throughput between 1 and 5 KB/s, and Figure 5.11(c) for buffers whose throughput exceeds 5 KB/s. Observe that all four graphs reveal that the composite cumulative distributions have the same shape since the partitioning contour lines all have the same form. Thus we can confirm that despite the variance in how much data travels through the buffers, the categories of slow and fast buffers remain three orders of magnitude apart. As expected, buffers that see less data are slightly faster than the average, i.e., the contour curve that separate fast and slow buffers is slightly above the average, and this partition falls below average (meaning slightly slower buffers) as the throughput increases. When buffers transport more data, the number of buffers that respond exceptionally fast (i.e., less than 100 microseconds), is subsumed by the category of faster buffers. These buffers are likely never empty, even if they are being continually emptied. This trend, however, is slight and the most important observation is that the existence of slow buffers is a systemic problem that occurs regardless of their utilization to transport data.

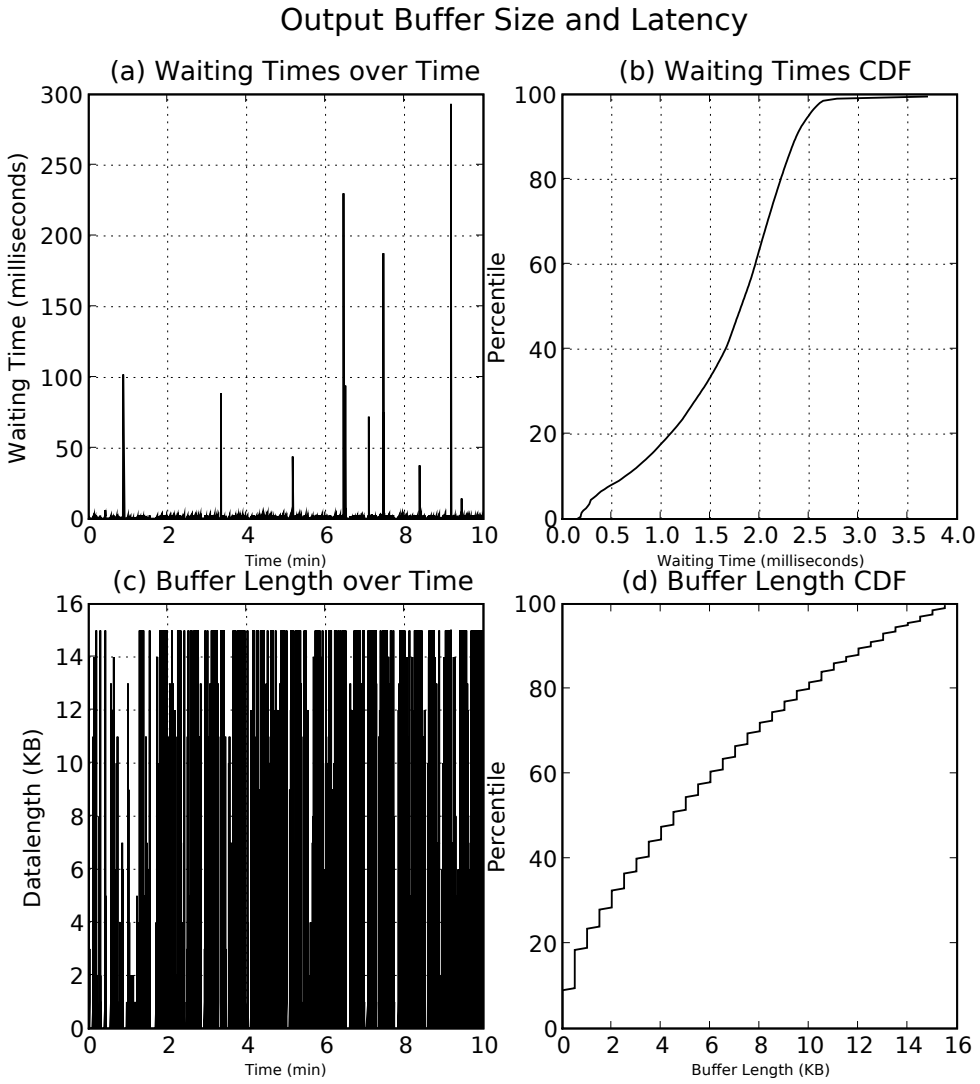

Figure 5.9: Waiting times and data sizes for a representative output buffer. (a) shows the time that data spends waiting in the buffer over time. (b) shows the time data spends waiting in the buffer as a cumulative distribution function. (c) shows the buffer size over time. (d) shows the buffer size as a cumulative distribution function.

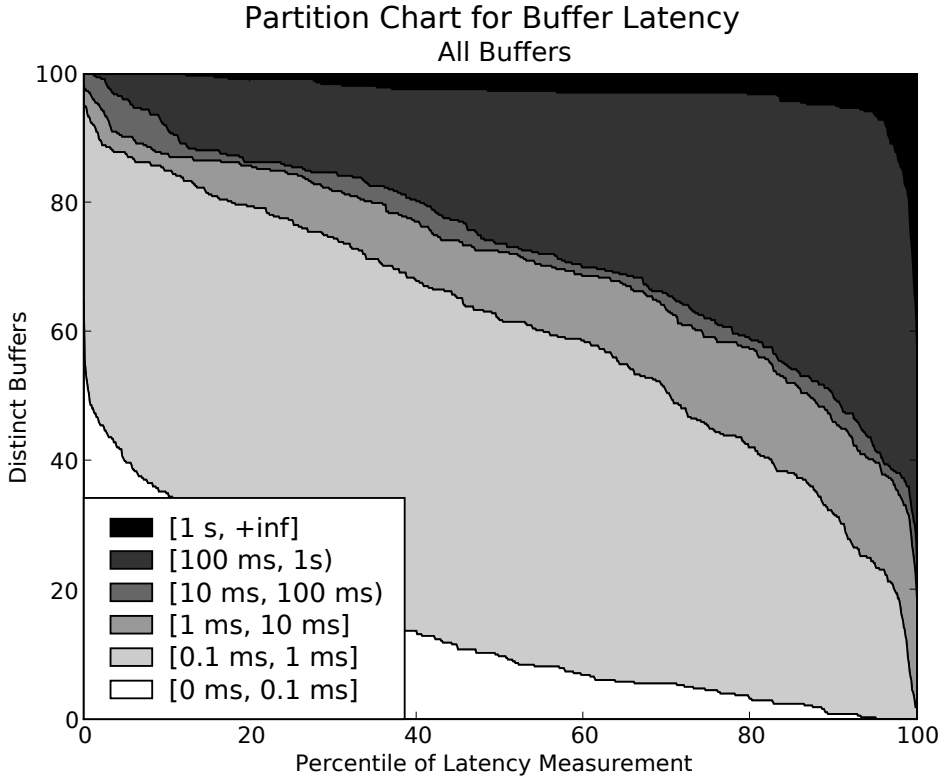

Figure 5.10: Partition diagram for output buffer latency across many buffers. White corresponds to buffer latencies less than 0.1 ms. Black corresponds to buffer latencies greater than 1 s, and each darker shade represents a ten-fold increase in latency.

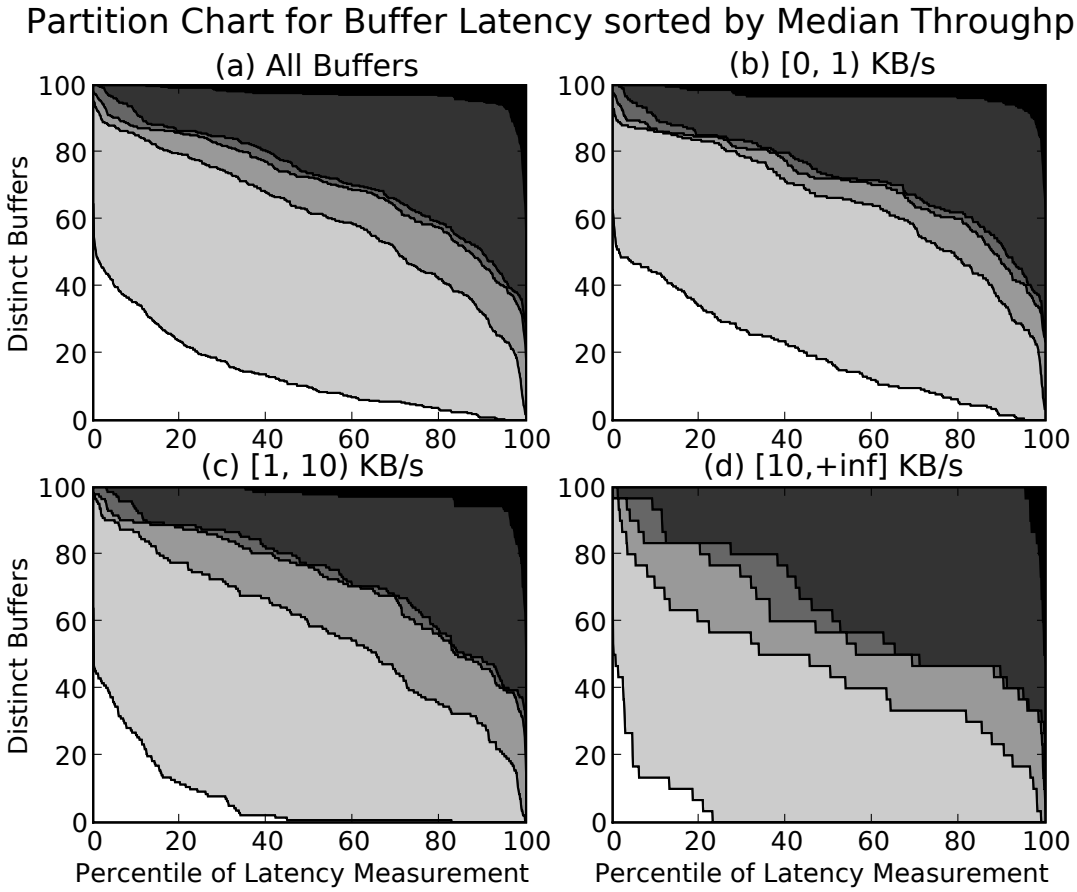

Figure 5.11: Partition diagram for output buffer latency sorted by throughput. (a) is the partitioning of all buffers. (b) is the partitions of buffers whose throughput is less than 1 KB/s. (c) is the partitioning of buffers whose throughput is between 1 and 10 KB/s. (d) is the partitioning of buffers whose throughput is greater than 10 KB/s. White corresponds to buffer latencies less than 0.1 ms. Black corresponds to buffer latencies greater than 1 s, and each darker shade represents a ten-fold increase in latency.

5.4 Thread Model for Reading and Writing

The observation of the previous section is that data is often available to dispatch but the function to actually perform the write is not being invoked. Our enquiry informs us to examine the component responsible for the invocation of writing. The threading model of Tor uses a single thread that handles all the reads, writes, and processing for every connection. Reading data from the socket into an input buffer, then processing available cells from the input buffer and placing the result on the output buffer is done as atomic operation. Taking data from the output buffer and dispatching to the network is also done as atomic operation. The single thread uses a round-robin scheduling semantics to perform work: reading when data is available on the socket, processing when data is available on a buffer, and dispatching when the socket is writable. All work is done serially.

The asynchronous callback library `libevent` manages this thread of execution. This library is designed to manage many file descriptors (FDs) concurrently and invoke callback functions when the file descriptor is ready for input/output (I/O); i.e., when the operation will not block. Tor uses `libevent` to manage all the socket connections to other routers, and so `libevent` is responsible for bridging the connection between socket file descriptions and the input and output buffers used by Tor. This execution model ensures that Tor will not block due to performing I/O operations on an unready file descriptor.

Tor informs `libevent` of every socket used for interrouter communication. `Libevent` is told to always manage read events when a socket has incoming data; write events are enabled only when there is data waiting on the buffer ready to be sent. `Libevent` wraps over a `select()` call to poll all FDs for I/O readiness and round-robins between the FDs with a single execution thread. This thread calls the appropriate callback method and only executes other I/O operations when it returns. This behaviour was confirmed by registering a callback method with an infinite loop; once the loop was called it ceased the invocation of all other callbacks, including timers, for the remainder of execution. Thus, the duration of one callback method can affect the latency of other callbacks. In particular, if a write to one socket stalls, then all other output buffers used by Tor will be unable to send data.

Since `libevent` is responsible for invoking the method that removes data from the buffer, and we witnessed that it is failing to do so in some circumstances, there are three hypotheses. First, `libevent` could be thrashing—when it returns from one callback it promply invokes another. As a queueing model, this would mean that Tor's service time (output) is unable to keep up with arrival rate (input), which results in an unsteady state. Second, callbacks could consume sufficient time so as to introduce unnecessary delays in other threads. Finally, the socket could be unwritable and `libevent` is behaving properly. Experiment 5 was performed to test the first two hypotheses, which were shown to be false.

Figure 5.12 shows the cumulative distribution function for the time that `libevent` spends waiting between callback invocations. It shows that for half of the

Experiment 5 Determing if `libevent` is thrashing or consuming great time in callbacks.

1: `Libevent` was instrumented to report the period between invocations of a callback, the name of the callback, and the time elapsed in the callback.
2: Tor was linked with the instrumented `libevent`.
3: A public Tor node `gurgle` was left running for a couple of days to accrue traffic.
4: `Libevent` was instructed to begin logging data.

time it waits for a few microseconds (likely consumed by the instrumentation code and the polling function instead of an actual wait). The other half of the time it sleeps for dozens of milliseconds without work. Tor uses the `libevent` callback to perform all the core functionality of routing traffic, and so this delay indicates that our node has no routing work to do during this time.

The lack of any pause between the majority of invocations of `libevent` may, at first glance, indicate that is thrashing the majority of the time, the following thought exercise produces a more likely hypothesis. The expected well-functioning behaviour of Tor is the following: `libevent` triggers the read callback, a cell is read into the system from a socket, this cell is promply processed by Tor in a matter of microseconds, put on the output queue, and control returns to `libevent`. Upon successful polling of the output socket, `libevent` will promptly invoke the write callback without any delay. If this is the case, then the lower 50 percentiles represent the read/write behaviour of Tor, and the upper 50 percentiles indicate the delays while waiting for data to arrive. To prove our hypothesis, we reperform Experiment 5 to annotate each delay measurement with the respective callback function that it executed after waiting and inspect the results to observe that longer delays occur before read events, collated with brief delays before write events. Figure 5.13 shows that `libevent` is not thrashing; it plots the percent of time (over intervals of a second) that `libevent` spends sleeping while waiting for work. It is clear that `libevent`, and therefore Tor, spends approximately 90% of its execution time in an idle state.

Figure 5.14 shows the cumulative distribution function for the time that `libevent` spends inside callback methods. It shows that nearly 80% of callbacks execute in fewer than 100 microseconds, which represents all relay and cell processing operations performed by Tor. Moreover, the maximum time reported was less than 7 milliseconds. This negligible time suggests that `libevent` functions properly, i.e. it only executes reads and writes callbacks when it knows that they will not block.

The long tail end occurs because some events triggered by `libevent` are for housekeeping and other tasks. A special timer callback expires every second and is used to multiplex a variety of timers that expire in multiples of second intervals. Figure 5.15 shows the CDF for the duration of the second elapsed callback function. We see that this function is also rapid for the most part, but the long tail appears at a smaller percentile. The maximum time reported was still a mere 9 milliseconds, and so we conclude that the housekeeping callback is not a significant source of

latency.

Our first two hypotheses are both false. Libevent is neither thrashing nor are cells ready for service being subjected to large delays while waiting to be serviced by the single I/O thread. The logical conclusion is that `libevent` is functioning properly, and data is being buffered because the socket is simply not writable. We explore socket writability in the next section.

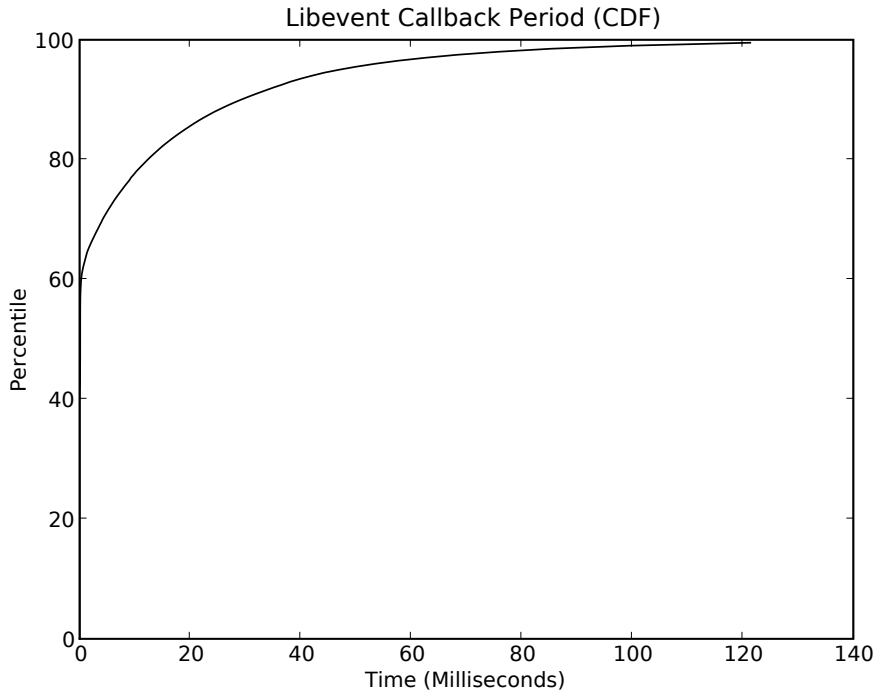

Figure 5.12: Distribution of waiting intervals in `libevent` (CDF).

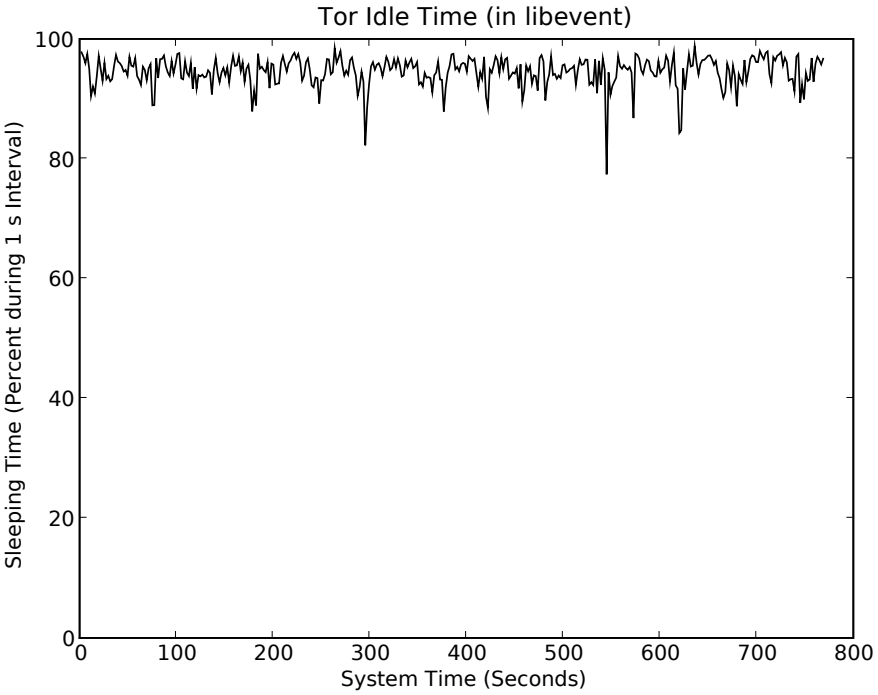

Figure 5.13: Idle time as a percent of 1 s intervals in Tor.

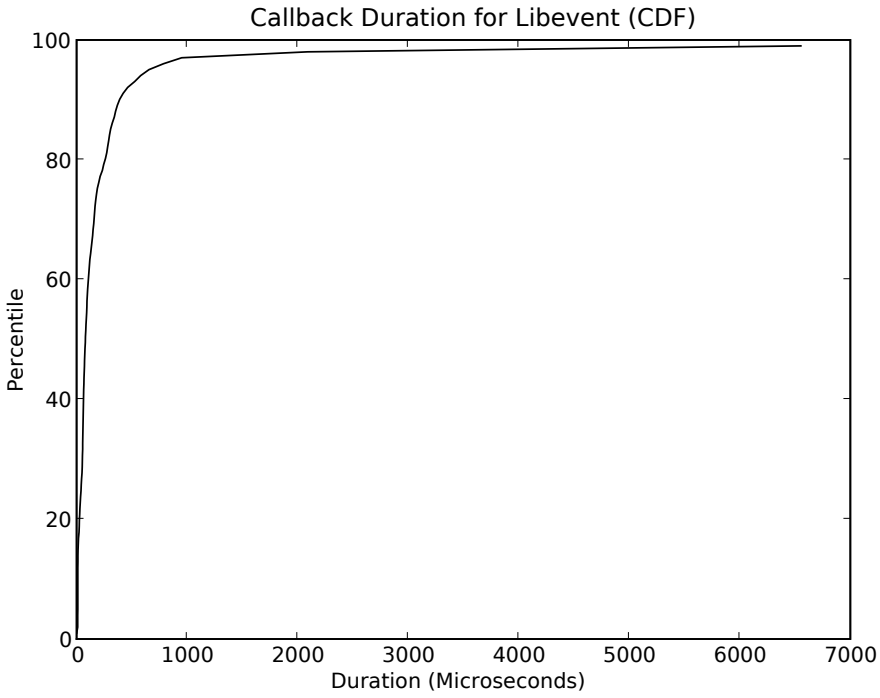

Figure 5.14: Duration for the execution of `libevent`'s event callbacks.

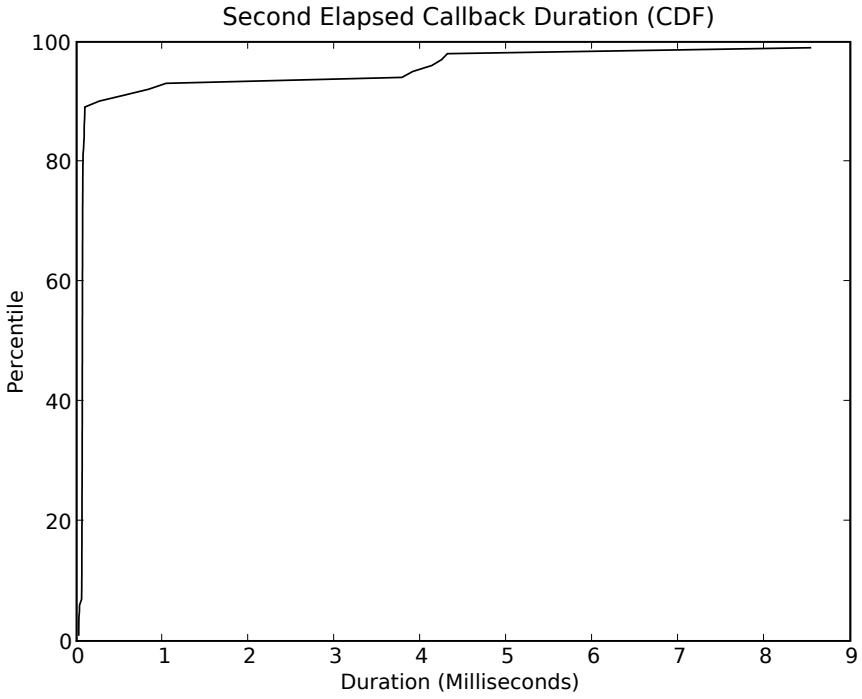

Figure 5.15: Duration for the `second elapsed` callback.

5.5 Unwritable Connections

As earlier described, buffers often swell when the write callback is not being invoked. We have ruled out a poorly functioning `libevent` as the reason for this behaviour—`libevent` often sleeps waiting for work. The problem is that the socket itself is unwritable. There are a couple of intuitive reasons for this to be the case. The other end of the TCP connection could return a window of size zero, indicating that they are currently unwilling to accept any more data. Alternatively, the other end of the connection may not have acknowledged much of the recently sent data, and the TCP buffer that maintains all sent-but-unacknowledged data is full. As we will show, the latter theory holds, as unwritable sockets occur precisely when the TCP buffer is nearly full.

5.5.1 TCP Window Sizes

The TCP window is a value included in the header of every TCP packet used for congestion control, which is discussed in section 3.2.2. The TCP specifications state that when a peer receives a TCP window size of zero it must immediately stop sending data. Since Tor occasionally fails to dispatch data, we perform Experiment 6 to determine if the window size ever drops to 0. This would result in the invocation of flow control based on the receiver's inability to read more data.

Experiment 6 Determining TCP window sizes over time.

1: A public Tor node `gurgle` was left running for a couple of days to accrue traffic.
2: `gurgle` was instructed to begin logging data about connection destination and writability over time.
3: `tcpdump` was used to capture packet header data; in particular, the destination and window size over time.
4: Data was collected for ten minutes.

Parsing logs from `tcpdump` show that the window size for all connections never drops to zero. Figure 5.16 is an example that is representative of all connections. The Y-axis corresponds to the window size in KB, and the X-axis corresponds to time in seconds. The series of vertical bars indicate periods of time when the socket is unwritable. We conclude that the window sizes vary between 65 KB and that value minus a segment size, depending on when the acknowledgement is sent. Window size is independent of sporadic periods of when the socket is unwritable. Occasionally the window size drops slightly; one such occurrence was before a retransmission was sent. This corresponds to out-of-order data that must sit on the TCP input buffer, consuming buffer space formerly available for new data, until the appropriate missing data arrives. We will explore this as a potential source of latency in Section 5.6.2.

5.5.2 TCP Output Buffer Sizes

The TCP output buffer is where TCP maintains a perfect record of all unacknowledged data. This record is used to generate retransmission messages if a packet is dropped. If the peer acknowledges data slowly, even if the data was sent properly, then the TCP output buffer will swell until acknowledgements are received. We hypothesize that the buffer size grows to a point where more memory cannot be allocated, and consequently the operating system reports the socket as unwritable. We perform Experiment 7 to explore the TCP output buffer size.

Experiment 7 Exploring the relationship between TCP output buffer size, writability, and unacknowledged packets.

1: A public Tor node `gurgle` was left running for a couple of days to accrue traffic.
2: `gurgle` was instructed to begin logging data about connection destination, writability, TCP output buffer size, the number of unacknowledged packets, and Tor's output buffer size, all over time.
3: Data was collected for ten minutes.

Figure 5.17 shows the output buffer size for a socket in Tor. The X-axis corresponds to time and the Y-axis corresponds to the size of data on the output buffer. We annotate this chart by using noughts for observations made when the socket is writable and crosses when the socket is unwritable. It shows that the socket becomes unwritable when its size grows beyond what its buffer can hold. Any writes at this point must cause the socket to block since there is no room to put more data. This causes the socket to report that it is unwritable.

TCP records the data that has been sent but not acknowledged to be able to retransmit lost packets. When data is acknowledged the duplicate copy is removed from the buffer. Figure 5.18 shows the relationship between unacknowledged messages and the output buffer size. The background line plot graphs the size of the output buffer over time. Crosses at the top of the chart occur when the socket reports it is unwritable. The horizontal line at the top indicates the socket's advertised output buffer capacity. Circles specify the number of unacknowledged packets over time, i.e. packets in flight. These values are scaled by the maximum segment size to get an approximate size of the outstanding data in bytes.

Figure 5.18 illustrates a number of things concisely. First, unwritable sockets occur when the remaining capacity in an output buffer is too small to accept new data. This in turn occurs because there is already too much data in the buffer, which is because there is too much unacknowledged data in flight and throttled data waiting to be sent. Data is throttled when the congestion window is reached; this is approximately a dozen or so packets for our example. Once congestion throttles sending, the data queues up until either packets are acknowledged or the buffer is full. Interestingly, despite having often a dozen packets in flight at any point in time, the socket reported only a single packet drop during the observation period. It occurred during the valley around 400 seconds when the number of unacknowledged

packets dropped; this behaviour suggests the fast retransmit function activated by halving the congestion window and starting congestion avoidance, which grows the congestion window over time to a capacious 18 packets. The RTT for this connection was 260 ms, and so plenty of data is sent during a RTT to account for its latency.

We have identified that latency in buffers, both in Tor and in the operating system, contribute to the observed latency beyond the round trip times between hosts. While data is delayed because of congestion control (but not flow control), it is foolhardy to attempt to circumvent congestion control as a means of improving Tor's latency. There is hope, however, as the next section details a problem in Tor's use of TCP that may contribute unnecessarily to latency.

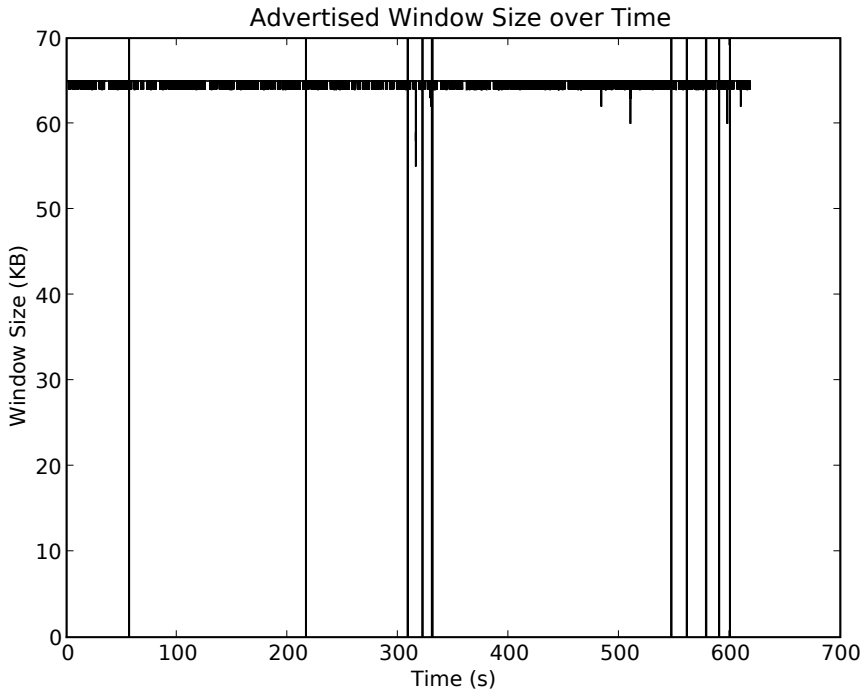

Figure 5.16: TCP window size over time.

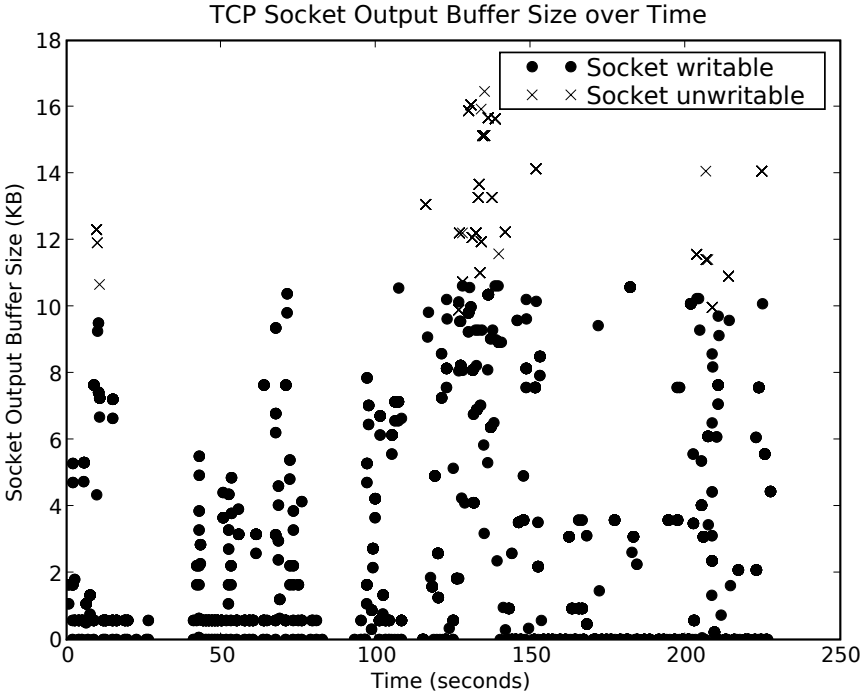

Figure 5.17: TCP socket output buffer size over time.

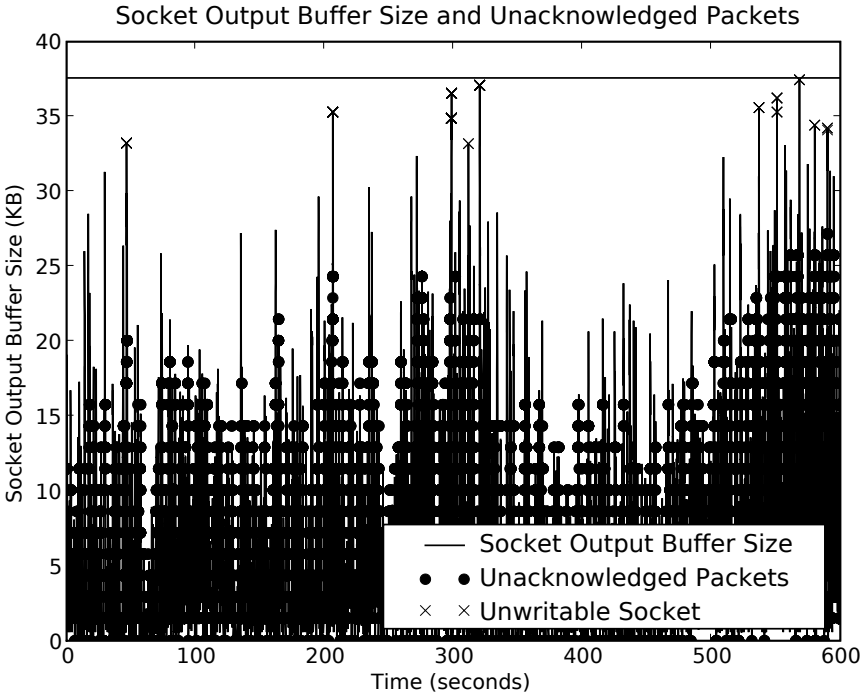

Figure 5.18: TCP socket output buffer size, writability, and unacknowledged packets over time.

5.6 TCP Multiplexing Problem

In this chapter we have first shown that computational complexity is not a significant source of latency, and have then shown that network conditions can result in significant latency while data waits in buffers. The reason data remains in buffers is due to congestion control mechanisms in TCP that reduce throughput. We must not impede congestion control to improve throughput—TCP congestion control is thoroughly researched to maximize throughput while avoiding packet drops. Despite this, we have a hypothesis about how TCP is used in Tor in a way that is degrading performance.

Tor's circuits are multiplexed over TCP connections; i.e., a single TCP connection between two routers is used for multiple circuits. When a circuit is built through a pair of unconnected routers, then a new TCP connection is established. However, if a circuit is built through already connected routers, then the existing TCP stream will carry both the existing circuits and the new circuit. This is true for all circuits built in either direction between the ORs.

In this section we explore how congestion control affects multiplexed circuits and how packet dropping and reordering can cause interference across circuits. We show that TCP was not intended to multiplex circuits in this manner, and briefly propose an alternate transport to which we devote the next chapter.

5.6.1 Unfair Congestion Control

Motivated by results we observed in the data generated by our Tor node, we believe that multiplexing TCP streams over a single TCP connection results in the improper application of TCP's congestion control protocol. In particular, the nature of TCP's congestion control mechanism results in multiple data streams competing to send data over a TCP stream that gives priority to circuits that send more data (i.e., it gives each byte of data the same priority regardless of its source). A busy stream that triggers congestion control will cause low-bandwidth streams to struggle to have their data sent. Figure 5.19 shows a demultiplexed buffer throughput graph for a real connection over Tor. Time increases along the X-axis, and data increases along the Y-axis. The two series correspond to two different streams which show when data is sent over time. The periods that swell indicate that Tor's internal output buffer has swelled; the left edge grows when data enters the buffer, and the right edge grows when data leaves the buffer. This results in the appearance of a line when the buffer is well-functioning, and a triangular or parallelogram shape when data arrives too rapidly or the connection is troubled. Additionally, we strike a vertical line across the graph whenever a packet is dropped.

What we learn from this buffer is that it serves two circuits. One circuit serves one MB over ten minutes, and sends cells evenly. The other circuit is inactive for the most part, but three times over execution it suddenly serves 200 KB of cells very quickly. We can see that each time this happens, the buffer swells with data

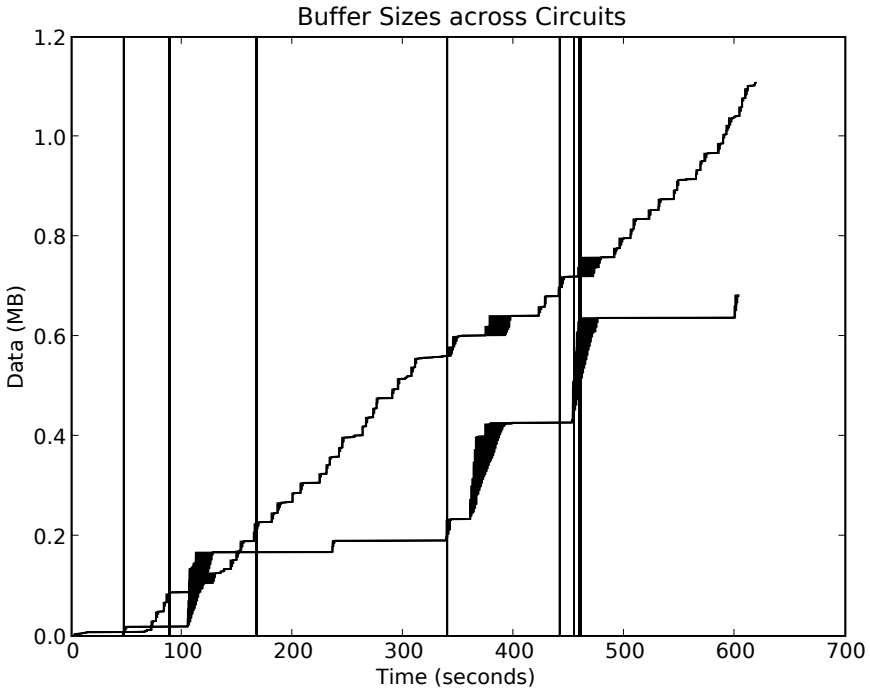

Figure 5.19: Example of congestion on multiple streams.

incurring a significant delay. Importantly, the other circuit is affected despite the fact that it did not change its behaviour. Congestion control mechanisms that throttle the TCP connection will give preference to the burst of writes because it simply provides more data, while the latency for low-bandwidth application such as `ssh` increases unfairly.

Since the main goal of Tor is interactive low-latency web-browsing, it is naturally of interest to consider whether large transfers are apt to interfere with this goal. Research has indicated that while only 3% of Tor circuits are used for BitTorrent (a protocol that indulges in large bulk transfers), it accounts for 40% of the transported data in Tor [41]. Clients who attempt to transport their interactive sessions alongside BitTorrent transports that become heavily congested will suffer increased latency as a consequence. In the next chapter, we will implement a system that separates the TCP streams for each circuit to avoid this problem, and our experimentation will show that it improves latency. For the remainder of this section, we will perform experiments that quantify the effects of packet dropping and packet reordering on multiplexed circuits using a local Tor network.

5.6.2 Cross-Circuit Interference

Multiplexing multiple streams of data over a single TCP stream ensures that the received data will appear in the precise order in which the component streams were

multiplexed. When packets are dropped or reordered, the TCP stack will buffer available data on input buffers until the missing in-order component is available. We hypothesize that when active circuits are multiplexed over a single TCP connection, Tor suffers an unreasonable performance reduction when either packet dropping or packet reordering occur. Cells may be available in-order for one particular circuit but are being delayed due to missing cells for another circuit. Ideally, multiplexed streams in TCP would be sent over the same connection using different channels, where in-order guarantees were only made for data sent over the same channel. However TCP sends all data over the same channel and so it is only readable in the order it was dispatched. Packet loss or reordering will cause the socket to indicate that no data is available to read even if other circuits have their sequential cells available in buffers. In the next chapter, we will present a UDP-based solution to demultiplex circuits from TCP streams and assuage the interference of packet dropping and reordering across circuits.

Figure 5.20 illustrates the dropped-packet hypothesis; cells for distinct circuits are represented by shades and a missing packet is represented with a cross. In the remainder of this section we perform experiments to determine how packet dropping and reordering affect a local Tor network, and emphasize that whenever this delay occurs it is unnecessary.

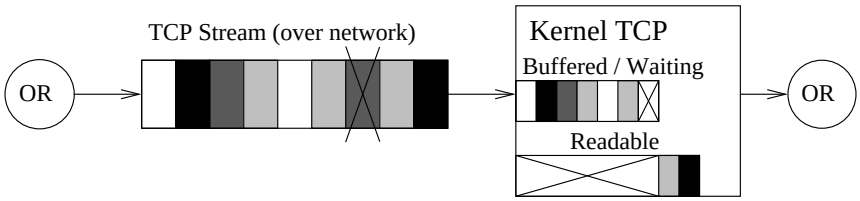

Figure 5.20: TCP correlated streams. Shades correspond to cells for different circuits.

Packet Dropping

The hypothesis on the cross-circuit interference of packet dropping is that missing packets from one circuit will result in delays for other circuits multiplexed over the same stream.

We will verify our hypothesis in two parts: First, in this section we show that packet drops on a shared link degrades throughput much worse than drops over unshared links. Second, in Chapter 7, we will show that this is not the case with our UDP implementation.

Experiment 8 was performed to investigate the effect of packet dropping on circuit multiplexing.

There is an important caveat in step 3: there is a difference between the frequency of traffic for the two sets of connections in our experiment (and all other

Experiment 8 Determining the effect of packet dropping on circuit multiplexing.

1: A Tor network of six routers on a single host was configured to communicate through the *latency_proxy* to control packet dropping. The latency was set to 50 milliseconds for all experiments. Packet reordering was disabled.
2: Eight clients built circuits that were fixed so that the second and third ORs were the same for each client, but the first hop was evenly distributed among the remaining routers. Figure 5.21 illustrates this setup.
3: There were three runs of the experiment. The first did not drop any packets. The second dropped 0.1% of packets on the shared link, and the third dropped 0.1% of packets on the remaining links.
4: The ORs were initialized and then the clients were run until circuits were established.
5: Timing clients running on the local machine connected to each OP and used Tor to tunnel their connection to a timing server running on the local machine.
6: Data was collected for over a minute.

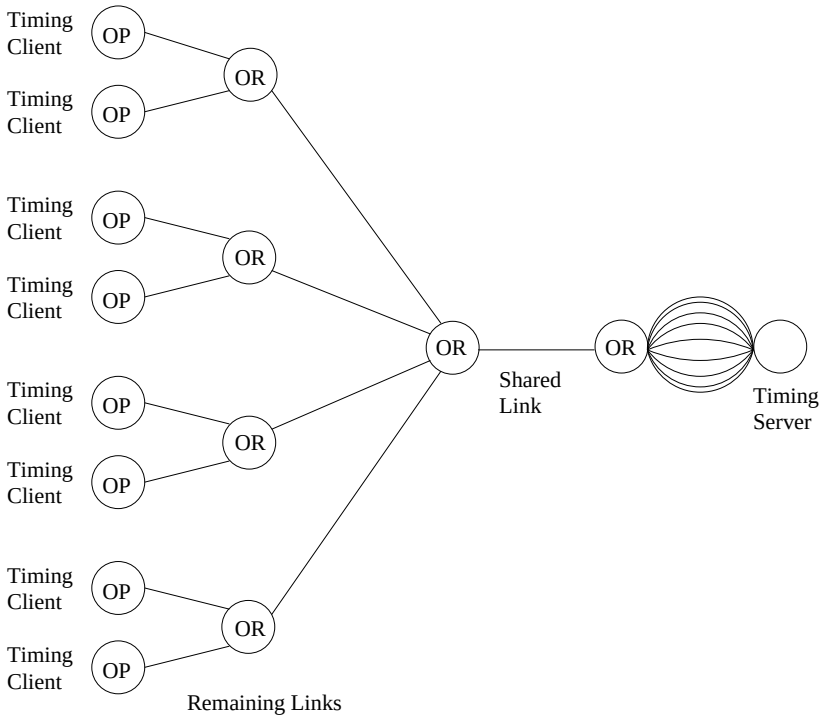

Figure 5.21: Setup for Experiments 8, 9, and 12. The first two columns of links constitute the remaining links, in contrast to the shared linked. The splay of links between the final OR and the timing server are each a separate TCP connection per circuit, and are not implemented through the `latency_proxy`.

Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation	Effective Drop Rate
No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
0.1 % (remaining)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 5.2: Throughput for different dropping configurations. Network throughput is the total data sent along all the circuits.

Configuration	Average Latency	Latency Degradation	Effective Drop Rate
No dropping	933 ± 260 ms	0 %	0 %
0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 5.3: Latency for different dropping configurations.

experiments set up like Figure 5.21). Traffic for separate circuits that are multiplexed over a single connection can be sent together in a single TCP packet. Hence, the shared link may transport one packet that will scaffold multiple packets to send to the next router. Moreover, the remaining links span two hops along each circuit whereas the shared link spans only one. This means that each packet will be eligible to be dropped twice as often in the remaining links than the shared link. We elucidate this disparity by explicitly providing the effective drop rate, i.e. the ratio of packets dropped to the total number of packets as reported by the `latency_proxy`.

The results of Experiment 8 are shown in Tables 5.2 and 5.3. The average results for throughput and delay were accumulated over half a dozen executions of the experiment, and the mean intervals for the variates are computed using Student's T distribution to 95% confidence.

These results confirm our hypothesis. The throughput degrades nearly threefold when packets are dropped on the shared link instead of the remaining links. This is despite a significantly lower overall drop rate. The behaviour of one TCP connection can adversely affect all correlated circuits, even if those circuits are used to transport less data.

Table 5.3 informs us that latency increases when packet dropping occurs. Latency is measured by the time required for a single cell to travel alongside a congested circuit, and so we average a few dozen such probes. It is unlikely that any of these probes are specifically dropped, so the increasing trend reflects the added delay on a TCP connection after a packet drop occurs. Again we see that dropping on the shared link more adversely affects the observed delay despite a reduced drop rate.

Configuration	Network Throughput (KB/s)	Circuit Throughput (KB/s)	Throughput Degradation
No reordering	221 ± 6.6	36.9 ± 1.1	0 %
Reorder (remaining)	219 ± 19	36.6 ± 3.2	0.8 %
Reorder (shared)	175 ± 8.4	29.3 ± 1.4	21 %

Table 5.4: Throughput for different reordering configurations.

Cross-Circuit Interference of Packet Reordering

The hypothesis on the cross-circuit interference of packet reordering is that (as with packet dropping) packet reordering causes unnecessary delays for other circuits multiplexed over the same stream. We perform Experiment 9 to validate the hypothesis.

Experiment 9 Determining the effect of packet reordering on circuit multiplexing.

1: A local Tor network of six routers on a single host was configured to communicate through the `latency_proxy` to control packet reordering. The latency was set to 50 milliseconds with a variable reordering component taken from an exponential distribution with a parameterized mean. Packet dropping was disabled.
2: Eight clients built circuits that were fixed so that the second and third ORs were the same for each client, but the first hop was evenly distributed among the remaining routers. Figure 5.21 illustrates this setup.
3: There were three runs of the experiment. The first used a reordering mean of 0 ms, meaning that it did not reorder any packets. The second used a mean of 5 ms, and only reordered of packets on the shared link. The third used a ordering mean of 5 ms, and reordered packets on the remaining links.
4: The ORs were initialized and then the clients were run until circuits were established.
5: Timing clients running on the local machine connected to each OP and used Tor to tunnel their connection to a timing server running on the local machine.
6: Data was collected for over a minute.

The results of experiment 9 are shown in Tables 5.4 and 5.5. The average results for throughput and delay were output over a half dozen executions of the experiment, and the mean intervals for the variates are computed using Student's T distribution to 95% confidence.

These results confirm our hypothesis. We see that throughput is nearly unaffected when reordering packets on the unshared connection. This is due to the fact that reordering packets on connections whose data is all the same circuit means that buffered data is available for reading when the missing packet arrives. All the available data will then be read for the same stream, and so the one second average throughput remains relatively unaffected. However, reordering packets on

Configuration	Average Latency	Latency Degradation
No reordering	933 ± 260 ms	0 %
Reorder (remaining)	1002 ± 450 ms	7 %
Reorder (shared)	1122 ± 351 ms	20 %

Table 5.5: Latency for different reordering configurations.

the shared link means that data for separate streams will simply stall waiting for data that does not contribute to that stream's throughput.

Latency increases for reordering packets. Foremost, we are increasing the latency by a random exponential around a mean of 5 ms, and so expect to see an increase in minimum latency. The larger confidence interval for reordering on remaining links reflects the fact that we are adding random latency for two connections along the circuit. This means that the remaining links see an expected 10 ms of additional latency versus 5 ms of additional latency for the experiment where reordering was performed on the shared link. Despite the reduced additional latency, we see that reordering on the shared link incurs a much greater cost on latency than reordering on the single shared link.

5.7 Summary

In this chapter we have performed a system performance evaluation to determine sources of latency in Tor. We have placed a bound on transport latency and found that there still exists significant latency beyond that. We have determined that computational latency is insignificant. There exists significant latency for data waiting inside Tor's output buffers, and we have isolated TCP congestion control as the reason for this latency.

Multiplexing circuits over a single connection is a potential source of unnecessary latency since it causes TCP's congestion control mechanism to operate unfairly towards connections with smaller demands on throughput. High bandwidth streams that trigger congestion control result in low-bandwidth streams having their congestion window reduced unfairly. Packet dropping and reordering also causes available data for multiplexed circuits to wait needlessly in socket buffers. These effects degrade both latency and throughput, which we have shown in experiments.

The next chapter presents a solution to this problem. Ideally, we wish to open a separate TCP connection for every circuit, as this would be a more appropriate use of TCP between ORs. However, exposed TCP headers will leak information about data being sent between ORs for each circuit, which opens the traffic to attack. Our solution is to tunnel multiple TCP streams traffic over DTLS, a UDP protocol that provides for the confidentiality of the traffic it transports. By tunnelling TCP over a secure protocol, we can afford both the TCP payload and the TCP headers with the security properties of DTLS.

Chapter 6

Proposed Transport Layer

This chapter presents a new transport layer for Tor with the goal of relieving some of the problems observed with the existing TCP transport layer. Our proposal is for a TCP-over-DTLS tunnelling transport; such a tunnel will transport TCP/IP packets between peers using DTLS—a secure datagram (UDP-based) transport [43]. User-level TCP stacks running inside the program will be responsible for generating and parsing the TCP/IP packets sent over DTLS. Our solution will use a single unconnected UDP socket to communicate with all other ORs externally, and have a separate TCP connection for each circuit internally. This decorrelates circuits from the same TCP stream which we have shown to be a source of unnecessary latency, while at the same time protecting the TCP headers for separate streams from being sent in plaintext. Moreover, it reduces the number of sockets needed in kernel space, which is known to be a problem that prevents some Windows computers from volunteering as ORs. Figure 6.1 shows the design of our proposed transport layer.

Our design uses TCP at each hop between ORs. This contrasts with another proposal to use UDP as the transport layer and TCP at the end-points [75]. Middle nodes only forward UDP packets in their proposal. We explore this proposal further in Section 8.1.8, including its benefits, weaknesses, and how it could be added to our system. A key difference is that our proposal does not require a synchronized update of the entire Tor network.

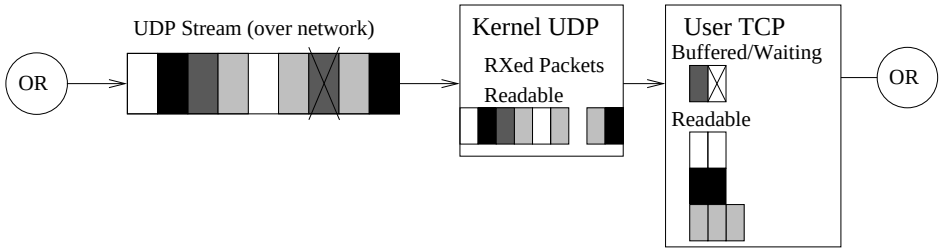

Figure 6.1: Proposed TCP-over-DTLS Transport showing decorrelated streams. Shades correspond to cells for different circuits (cf. Figure 5.20).

In this chapter, we first summarize the problems with TCP, propose our TCP-over-DTLS tunnel, and discuss our implementation design and details. This includes how our transport layer will work with the existing Tor network and how it was integrated into Tor.

6.1 Problems with TCP

Chapter 5 has shown that in the current TCP transport layer, multiplexing circuits over the same TCP connection contributes to observed latency:

- Congestion control mechanisms over multiplexed streams unfairly limit all streams instead of the most active ones.
- Packet dropping or reordering over multiplexed streams causes latency for all streams instead of the affected stream.

Unfairness in congestion control means that some streams that have large bulk traffic, such as BitTorrent, can negatively affect the remaining streams. We wish to assure fairness alongside privacy in anonymous routing—ensuring that interactive web browsing circuits are not subjected to unnecessary latency when multiplexed with bulk transfer circuits is a step towards this goal.

The interference that multiplexed circuits can have on each other during congestion, dropping, and reordering is a consequence of using a single TCP connection to transport data between each pair of ORs. We propose to use a separate TCP connection for each circuit, which ensures that congestion or drops in one circuit will not effect other circuits. In order to hide the details of which traffic belongs to which circuit, and provide the necessary confidentiality and security of the transported cells, we use DTLS—secure datagram transport.

6.2 TCP-over-DTLS Tunnel

DTLS [43] is the datagram equivalent to the ubiquitous TLS protocol that secures web traffic. We chose DTLS for our application due to its acceptance as a standard and its existing implementation in the OpenSSL library which is already being used in Tor. A TCP-over-DTLS transport, known as a tunnel in the networking literature [50], means that the DTLS protocol will be used to transport data for another protocol (in this case the TCP protocol). We would take a prepared TCP/IP packet that was meant to be sent by the system, remove the unnecessary IP header, and encapsulate the TCP packet inside a DTLS packet that is then sent in a UDP/IP datagram. The receiving system will remove the UDP/IP header when receiving data from the socket, decrypt the DTLS payload to obtain a TCP packet, translate it into a TCP/IP packet, which is then forwarded to the user-level TCP stack that

processes the packet. A second read from the user-level TCP stack will provide the packet data to our system.

In our system, the TCP sockets will reside in user space, and the UDP sockets will reside in kernel space. The use of TCP-over-DTLS affords us the great utility of TCP: guaranteed in-order delivery and congestion control. The user-level TCP stack controls provides the functionality of TCP, and the kernel-level UDP stack is used simply to transmit packets. The secured DTLS transport allows us to protect the TCP header from forgery and effect a reduced number of kernel-level sockets.

ORs require opening many sockets, and so our user-level TCP stack must be able to handle many concurrent sockets, instead of relying on the operating system's TCP implementation which varies from system to system. In particular, some discount versions of Windows artificially limit the number of sockets the user can open, and so we use Linux's free, open-source, and high-performance TCP implementation inside user space. Even Windows users will be able to benefit from an improved TCP implementation, and thus any user of an operating system supported by Tor will be able to volunteer their computer as an OR if they so choose. We state here (and explore further in the experimentation and future work sections) that our user-level implementation does not currently permit opening thousands of sockets, but it is straightforward to resolve before deployment.

UDP allows sockets to operate in an unconnected state. Each time a datagram is to be sent over the Internet, the destination for the packet is also provided. Only one socket is needed to send data to every OR in the Tor network. Similarly, when data is read from the socket, the sender's address is also provided alongside the data. This allows a single socket to be used for reading from all ORs; all connections and circuits will be multiplexed over the same socket. When reading, the sender's address can be used to demultiplex the packet to determine the appropriate connection for which it is bound. What follows is that a single UDP socket can be used to communicate with as many ORs as necessary; the number of kernel-level sockets is constant for arbitrarily many ORs with whom a connection is established.

Tor currently uses TLS to secure the payload of its TCP messages; we will

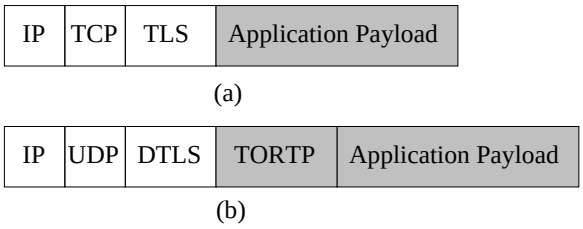

Figure 6.2: Packets for TCP Tor and our TCP-over-DTLS improved Tor. Encrypted and authenticated components of the packet are shaded in grey. (a) shows the packet format for TCP Tor. (b) shows the packet format for our TCP-over-DTLS Tor. TORTP is a compressed form of the IP and TCP headers, and will be discussed in Section 6.4.2.

use Datagram TLS (DTLS) to secure the payload of our UDP packets. Observe that the payload of our UDP packets includes the TCP header. Our TCP-over-DTLS allows each circuit to manage a unique TCP connection while providing confidentiality to the TCP header to prevent details of each circuit per connection from being revealed an observer. Protecting the header also prevents the TCP censorship attack that involves forging TCP headers, because the TCP header will now be secured inside the UDP payload. Figure 6.2(a) shows the packet format for TCP Tor, and Figure 6.2(b) shows the packet format for our TCP-over-DTLS Tor, which has expanded the encrypted payload to include the TCP/IP headers generated by the user-level TCP stack. The remainder of this chapter will discuss how we designed, implemented, and integrated these changes into Tor.

6.3 Backwards Compatibility

Our goal is to improve Tor to allow TCP communication using UDP in the transport layer. While the original ORs transported cells between themselves, our proposal is to transport, using UDP, both TCP headers and cells between ORs. The ORs will provide the TCP/IP packets to a TCP stack that will generate both the appropriate stream of cells and a TCP/IP packet containing a TCP acknowledgement to be returned.

The integration of this transport layer into Tor has two main objectives. First is that of interoperability; it is essential that the improved Tor is backwards compatible with the TCP version of Tor so as to be easily accepted into the existing codebase. Recall that Tor has thousands of ORs and has not experienced any downtime since it launched in 2003. We cannot insist that the userbase upgrades simultaneously to integrate our changes. A subset of nodes that upgrade and can take advantage of UDP, henceforth called "UDP nodes", which then provide evidence of TCP-over-DTLS's improvement for the user experience, is the preferred path to acceptance. Our second objective is to minimize the changes required into the existing Tor codebase. We must add UDP connections into the existing datapath by reusing as much existing code as possible. This permits future developers to continue to improve Tor's datapath without having to consider two classes of communication. Moreover, it encourages the changes to quickly be usurped into the main branch of the source code. While it will come with a performance cost for doing unnecessary operations, we will perform timing analysis to ensure that the resulting datapath latency remains negligible.

For interoperability we let older nodes communicate with the improved nodes using TCP. The listeners for TCP remain unchanged but newer nodes can advertise a UDP port for TCP-over-DTLS connections. When the OR is instructed to make a connection (for example, when forming a circuit to a previously unconnected node) the connecting OR first checks if the destination OR offers a UDP service and uses it if available. Thus, two UDP nodes will automatically communicate using UDP without disrupting the remaining nodes; their UDP connection will be

inconspicuous even among ORs involved in a circuit that uses that link. Traffic can switch between transport mechanisms in a node, since all traffic flows over a single connection between two nodes. All traffic that travels between a pair of UDP nodes will then travel over a UDP link.

Clients of Tor will not need to upgrade their software to obtain the benefits of UDP transport. In fact, it is important that they continue to choose their circuit randomly among the ORs: intentionally choosing circuits consisting of UDP nodes when there are only a few such nodes decreases the privacy afforded to the client by rendering their circuit choices predictable. However, when two neighbouring ORs are instructed to form a connection, and they both can communicate over UDP, then they will automatically use it to improve the user's experience.

6.4 User-level TCP Stack

If we simply replaced the TCP transport layer in Tor with a UDP transport layer, our inter-OR communication would then lack the critical features of TCP: guaranteed, in-order transmission of streams; and the most well-studied congestion control mechanism ever devised. We wish to remove some of the unnecessary guarantees of TCP for the sake of latency, i.e. we do not need cells from separate circuits over the same connection to arrive in the order they were dispatched. However, we must still be able to reconstruct the streams of each individual circuit at both ends of the connection. We use a TCP implemention in user-space (instead of inside the operating system) to accommodate us; a user-level TCP stack provides the implementation of the TCP protocols [31] as part of our program. User-level socket file descriptors and their associated data structures and buffers are accessible only in user-space and so are visible and relevant only to Tor. We use the UDP transport layer and DTLS to transport TCP/IP packets between the UDP peers. Only part of the TCP/IP packet is transmitted; the details will be discussed in section 6.4.2, but it serves our purposes now to conceptualize the two nodes as transporting full TCP/IP packets as the UDP datagram's payload. Upon receiving a UDP datagram, the kernel will remove the UDP header and provide Tor with a TCP/IP packet; Tor decrypts the DTLS payload and presents the result (a TCP/IP packet) to its user-level TCP stack. Similarly, when the user-level TCP stack presents a packet for transmission, the node will forward the packet to the kernel which then writes it to the intended destination over UDP. The stack also performs retransmission and acknowledgements of TCP data that is integral to TCP's reliability; these are forwarded over UDP in the same manner.

A user-level TCP stack provides an implementation of the suite of socket function calls, such as *socket()*, *send()*, and *recv()*. These reimplementations exist in harmony with the proper set of operating system commands, allowing both a user-level and kernel-level network layer. Thus, data structures and file descriptors created by calls to the user-level stack are visible and relevant only to the parent process; the operating system manages its sockets separately. The user-level stack

responds to socket calls by generating packets interally for dispatching as dictated by the TCP protocol [31].

It may seem cumbersome to include an entire TCP implementation as a core component of Tor. In particular, patching the kernel's implementation of TCP to support our features would take significantly less effort. However, Tor relies on volunteers to route traffic; complicated installation procedures are an immediate roadblock towards the ubiquitous use of Tor. The diverse operating systems Tor aims to support and the diverse skill level of its users prevent its installation from requiring external procedures or super-user privileges.

6.4.1 Daytona: A User-Level TCP Stack

Daytona [52] is a user-level TCP stack that we chose for our purposes. It was created by researchers studying network analysis, and consists of the implementation of Linux's TCP stack and the reimplementations of user-level socket functions. Additionally, it uses `libpcap` (as we did for the `latency_proxy`) to sniff packets straight from the ethernet device and a raw socket to write generated packets, including headers, onto the network. A system diagram of Daytona is shown in figure 6.3. Daytona was designed to operate over actual networks while still giving user-level access to the network implementation. In particular, it allowed the researchers to tune the implementation while performing intrusive measurements. A caveat—there are licensing issues for Daytona's use in Tor. As a result, the deployment of this transport layer into the real Tor network may use a different user-level TCP stack. Our design uses Daytona as a replaceable component and its selection as a user-level TCP stack was out of availability.

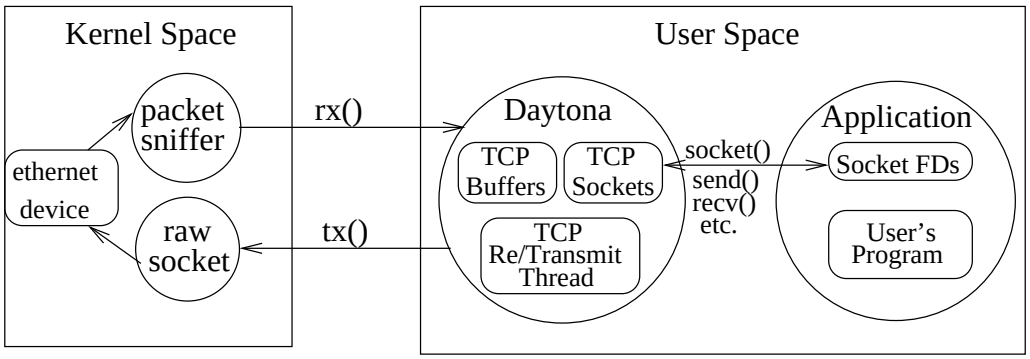

Figure 6.3: Daytona User-Level TCP Stack System Diagram.

However, Daytona's integration with a live network is more functionality than necessary for our purposes. We simply need the user-level stack to manage the multiplexing of many TCP streams over a single UDP connection. As Tor obtains packets over UDP, it forwards them to the user-level stack, which then adds it to the appropriate socket. If the packet contains data that is relevant for immediate reading, Tor will read from the appropriate socket to withdraw the fresh cells from

the user-level stream. Conversely, when Tor would previously send a cell over a socket, it instead uses the user-level send function. The user-level stack generates a packet to dispatch containing header information and the cell. It may also contain an acknowledgement for previous data and perhaps a retransmission of unsent data, as dictated by the TCP protocol. The generated packets need to be given to Tor, where it can route them to the appropriate OR over UDP, encrypted using the negotiated DTLS session.

6.4.2 UTCP: Our Tor-Daytona Interface

Our requirements for a user-level TCP stack are to create properly formatted packets, including TCP retransmissions, and to sort incoming TCP/IP packets into data streams: a black box that converts between streams and packets. For our purpose, all notions of routing, ethernet devices, and interactions with a live network are unnecessary. To access the receiving and transmitting of packets, we commandeer the `rx()` (receive) and `tx()` (transmit) methods of Daytona to instead interface directly with reading and writing to connections in Tor. The new system diagram for our user-level TCP stack, which we call the UTCP Interface, is shown in Figure 6.4.

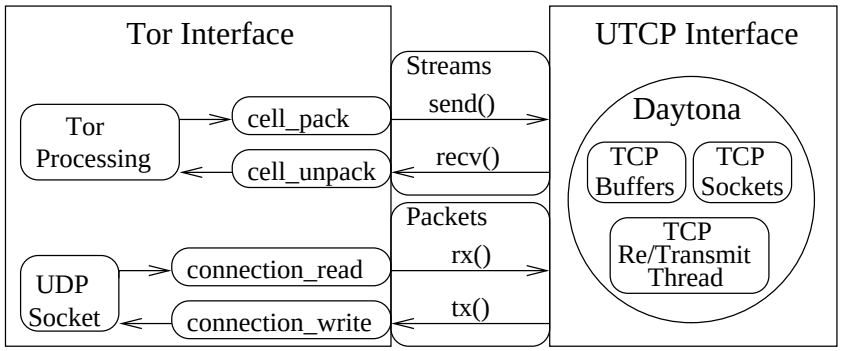

Figure 6.4: User-Level TCP Stack System Diagram.

UTCP is an abstraction layer for the Daytona TCP stack used as an interface for the stack by Tor. Each UDP connection between ORs has a UTCP-connection object that maintains information needed by our stack, such as the set of circuits between those peers and the socket that listens for new connections. Each circuit has a UTCP-circuit object for similar purposes, such as the local and remote port numbers that we have assigned this connection.

As mentioned earlier, only part of the TCP header is transmitted using Tor. We do this simply to optimize network traffic, as much of the information in the 40 byte TCP/IP header is provided by the necessary UDP/IP header. The source and destination addresses and ports can be replaced with a numerical identifier that uniquely identifies the circuit for the connection. Since a UDP/IP header is transmitted over the actual network, Tor is capable of performing a connection look up based on the address of the packet sender. (Recall that in connectionless socket

semantics, the sender's address is returned along with the packet data.) With the appropriate connection, and a circuit identifier, the interface to Daytona is capable of generating the TCP/IP headers such that Daytona will accept the packet to the appropriate TCP stream.

The following header data needs to be transmitted: sequence, acknowledgement, and window size numbers for reliability and congestion control, an identifier for demultiplexing the packet to the proper circuit, and some TCP flags to control connections. We send the sequence and acknowledgement numbers intact, the TCP flags we need, and replace the pair of IP addresses and TCP ports with a single port ID and a connector flag. The port ID is used in a map to retrieve the circuit, and is selected incrementally based on the number of opened circuits between the hosts. The connector flag indicates which peer selected the port ID, so each peer manages two maps of port IDs to circuits. Transmission of the header information is secured at the transport level by DTLS, so an external eavesdropper cannot learn the number of open circuits or precise distribution of data transfer among circuits by observing traffic. Figure 6.5 shows the layout of the TORTP header.

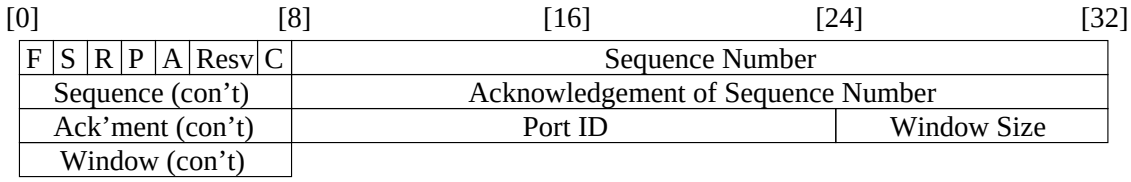

Figure 6.5: Layout of the TORTP header.

When the UTCP interface receives a new packet, it uses local data and the TORTP headers to create the corresponding TCP/IP header. The resulting packet is then injected into the TCP stack. When Daytona's TCP stack emits a new packet, a generic *tx()* method is invoked, passing only the packet and its length. We look up the corresponding UTCP circuit using the addresses and ports of the emitted TCP/IP header, and translate the TCP/IP header to our TORTP header and copy the TCP payload. This prepared TORTP packet is then sent to Tor, along with a reference to the appropriate circuit, and Tor sends the packet to the destination OR over the appropriate connection. The next section will describe both of these processes in more detail.

6.5 Integration of UDP Transport into Tor

In this section we describe how we integrated our TCP-over-DTLS transport layer as an optional transport layer for use in the Tor network.

Our goal is to replace TCP sockets with user-level TCP sockets when possible (i.e., whenever the peer is a UDP node), and replace all kernel-level socket calls with their equivalent user-level calls. Additionally, we also want to create a new socket for each circuit in Tor when the peer is a UDP node, and use the existing transport in Tor otherwise. Sending a CREATE cell was formerly sufficient to create a new circuit, but now we must perform user-level TCP handshaking, and then send the CREATE cell as the first data segment. This is further confounded by our goal of complete interoperability with the existing Tor network. It was necessary to make some larger changes to the operation of Tor itself: how UDP connections are established between ORs, and how circuits are established. This section describes the changes we made and describes our new algorithms for transmitting and receiving messages.

6.5.1 Establishing a Connection

Each pair of connected ORs in Tor manages a connection object at each endpoint. The connection object wraps a TLS object that secures all data sent over the connection against a variety of network-level attacks. The encryption used for TCP sockets in TLS is stream based, necessitating reliable, in-order delivery for the encrypted messages to be properly decrypted. Since we will use a UDP model of communication, we need a cipher that can decrypt datagrams without any guarantees of delivery or delivery order. The popular OpenSSL library provides an implementation of DTLS, a mechanism exactly for securing UDP socket connections. Where TCP connections wrapped a TLS object, our UDP connections will now each manage a DTLS object for securing their traffic. The existing datapath in Tor is preserved, and we change functions that operate on connections to use our UDP extensions when appropriate.

Limitations on the number of available file descriptors that prevent Windows computers from volunteering as ORs can be resolved with our UDP implementation: each OR opens a single UDP socket, operating in unconnected mode, which accepts and sends all traffic between ORs. Unfortunately, OpenSSL's DTLS implementation follows the stream paradigm of TCP. When a client begins a handshake, it forces the underlying UDP socket to enter into a connected state and hence will not accept traffic from any other destination. This restricts our ability to use a single UDP socket to accept data from all connections and use the peer's address for demultiplexing. We resolved this problem by providing our own layer of indirection between the socket and OpenSSL.

Before we explain the solution, it is useful to understand the mechanics of OpenSSL. The main data type in OpenSSL is the `ssl_t`. Socket level functions such

as `connect` and `close`, have equivalents in SSL, generally named by prepending an `SSL_` (i.e. `SSL_connect` and `SSL_close`) and passing an `ssl_t` instead of a socket as the first argument. These objects wrap over a buffered input/output (BIO) layer. BIOs themselves wrap over a file descriptor, such as a socket. The BIO layer is aware of the semantics for reading and writing to a file descriptor, and can be easily replaced with a customized BIO layer that wraps a different kind of file descriptor. Our custom BIO layer manages a unconnected socket by maintaining a reference to the peer with whom it shares a key and only sends and receives data from that peer. A single unconnected socket is shared by each instantiated BIO layer in our system. This socket is called the multiplexing socket because it exists in a one-to-many relationship with the ORs to which it has established DTLS connections. This permits each OR to use a constant number of kernel-level sockets independent of the number of communicating ORs. Handshaking still occurs in a connected context, but the socket is migrated to the shared UDP multiplexing socket immediately afterwards.

UDP-enabled ORs will now advertise both a UDP listening address and a UDP multiplexing address in their router description, and open a UDP socket for listening on both those address. The connecting OR will initiate a DTLS handshake from a new UDP socket and the listener will begin their communication by completing the DTLS handshake. After handshaking, both the connector and the acceptor will look up their peer's multiplexing address in their copy of the public directory of Tor nodes. They both silently migrate the underlying file descriptor for the SSL to the multiplexing address and set the peer address to the multiplexing address. Hence, when data is sent to the `ssl_t`, it will use the sendto method on the multiplexing socket using their peer's address. The multiplexing socket is in a one-to-many relationship with the BIO objects; each distinct BIO object maintains the peer's address for the `sendto` call on the multiplexing socket. After migrating to the multiplexing socket, the connector closes its connecting socket.

If the connector is an OP, then the acceptor will not be able to authenticate its peer or get its multiplexing address. In this case, the OP will use a separate UDP socket for each connection, and the OR will migrate its socket to the multiplexing socket without changing the peer's address. This is not a concern for proliferation of sockets, however, because OPs need only one socket for each of their circuits, which will be in the order of the number of sockets they intend to open and tunnel through Tor.

When data arrives on the multiplexing socket, we peek at the peer's address (without removing the datagram from the socket) in order to look up the connection object from which the datagram was sent. This gives us the correct `ssl_t` to use when reading the packet sitting on the buffer. However, this tacitly assumes a one-to-one relationship between peer OR addresses and the corresponding connection objects, which we use to uniquely demultiplex the incoming packets. This is incorrect, however, since pairs of ORs may have multiple connections between them: they connect simultaneously, they prepare a new connection to rotate keys, or one side restarts Tor ungracefully. Each connection object maintains a pointer

to another connection to the same OR, and Tor manages this ad-hoc linked list whenever connections are added or removed. In the current Tor implementation, each connection object has a unique pair of TCP ports and associated sockets, and thus this is of no concern since incoming data is uniquely tied to a single connection. However, we multiplex all data from the same OR over a single UDP connection and we rely on the UDP/IP packet's source address and port to uniquely identify the connection.

Our solution is to further customize our DTLS BIO layer to prepend a DTLS ID to each packet. When we peek at the sender's address for data that has arrived on the multiplexing socket, we now also peek at the first four bytes of the packet. Both the address and the DTLS ID are used to retrieve the proper connection object. While an active adversary can modify the DTLS ID, doing so will result in the packet failing to decrypt, which has the same effect as modifying any bit of the encrypted payload without the DTLS ID.

6.5.2 Establishing a Circuit

Recall the semantics for establishing circuits in Tor, presented in section 3.1. When an OR wishes to establish or extend a circuit to another OR, it first checks to see if there is an existing TCP connection to that OR. It establishes a connection only if there is no existing connection, and reuses the existing one if possible. The circuit building OR then sends a CREATE cell along the connection to the next OR in the circuit. When a CREATE cell is received, the handshaking involved in establishing encryptions keys is performed and the result is returned in a CREATED cell. Both ends of the circuit hop now have a circuit ID that is used for all future cells bound to the circuit. Our goal is to demultiplex circuits from using a TCP connection for their data, and so we create UTCP sockets for each circuit.

Each TCP-over-DTLS circuit in Tor will now be associated with a UTCP circuit object which wraps a UTCP socket. This UTCP circuit maintains its UTCP socket, its parent connection, and port information used during TORTP and TCP/IP header generation and translation. Each TCP-over-DTLS connection is similarly mapped to a UTCP connection object. The UTCP connection object manages the set of UTCP sockets that are virtually bound between the ORs at the ends of the connection: the set of all circuits that are built over this connection, a listening socket to accept new connections, a special circuit for inter-OR communications, and counters used in port allocation for new circuits.

Figure 6.6 shows the message sequence to establish circuits. When the initial DTLS connection is created between two ORs, each end creates a UTCP listening socket that is used to accept circuits from its peer. When the connector processes the CREATE cell, it first establishes a new UTCP connection to its peer. This causes its UTCP connection object to generate a new port ID (unique among those shared between those two particular hosts). The user-level connect call is made, and the stack emits a TCP/IP packet with its SYN flag set. This packet is translated

into a TORTP packet, and the port ID that was generated is placed in the TORTP header, which is then sent to the peer over DTLS. The receiving peer checks for a set SYN flag before injecting each packet; if the SYN flag is set then it will call the user-level TCP accept function to get a socket for the new UTCP circuit after it injects the SYN packet. The new UTCP circuit object is added to a list of accepted UTCP circuits that have not yet been bound to a Tor circuit. After the connector finishes establishing its UTCP connection, it sends the CREATE cell as the first packet of data. The receiver retrieves the previously accepted circuit by the port ID, and annotates it with the circuit ID specified in the CREATE cell. When Tor later processes the CREATE cell, it knows that the connection from which it came is a UTCP connection, and retrieves the appropriate UTCP circuit from the UTCP connection object using only the circuit ID.

Tor occasionally sends cells with a circuit ID set to 0. This includes keep-alive messages designed to ensure that established but inactive connections do not get dismantled. We create a base circuit (with circuit ID 0) to accommodate such messages. All traffic that is sent over the connection without a particular circuit is now mapped to our base circuit.

6.5.3 Sending and Receiving Data

TCP Tor reads cells from a kernel-level TCP socket and places them on a single input buffer. Our modified Tor uses separate user-level TCP sockets for each circuit, but all cells that are read (across all circuits that share a connection) are placed onto the same input buffer (that TCP Tor would use) so as to not interfere with the remaining datapath. Similarly, data that is written to the socket in TCP Tor is instead written to a user-level socket in our version. We add the additional functionality to forward the packets that our TCP-stack generates to the circuit queues that hold data that is ready to dispatch on the network. Figure 6.7 shows the sequence of sends and receives for the processing and relaying of a cell for TCP-over-DTLS Tor.

When the UDP multiplexing socket provides UTCP with a packet, the peer address and DTLS ID are used to retrieve the appropriate DTLS object for decryption. The resulting TORTP packet is forwarded to the `rx()` method, which uses the TORTP header to retrieve the appropriate UTCP circuit object bound to the current UTCP connection object. The circuit contains the information needed to generate the corresponding TCP/IP packet, which is then injected into the stack. After injecting, the user-level socket may now be readable (if the packet contains in-order data), and so a read attempt is made. If data is available, it is placed onto the the input buffer for the connection, and Tor's processing of the cells is performed as usual.

When Tor would formerly write data to the network, it now writes it to the user-level stack. Cells being written to a TCP-over-DTLS connection will instead use the user-level write function on their circuits specific socket. The user-level

stack will then emit TCP/IP packets via the `tx()` method. The `tx` method uses the TCP ports in generated header to determine the UTCP circuit, performs the TCP/IP to TORTP translation, and encrypts the packet with the DTLS object that corresponds with its destination. Finally, DTLS sends the packet to the intended destination over the UDP multiplexing socket.

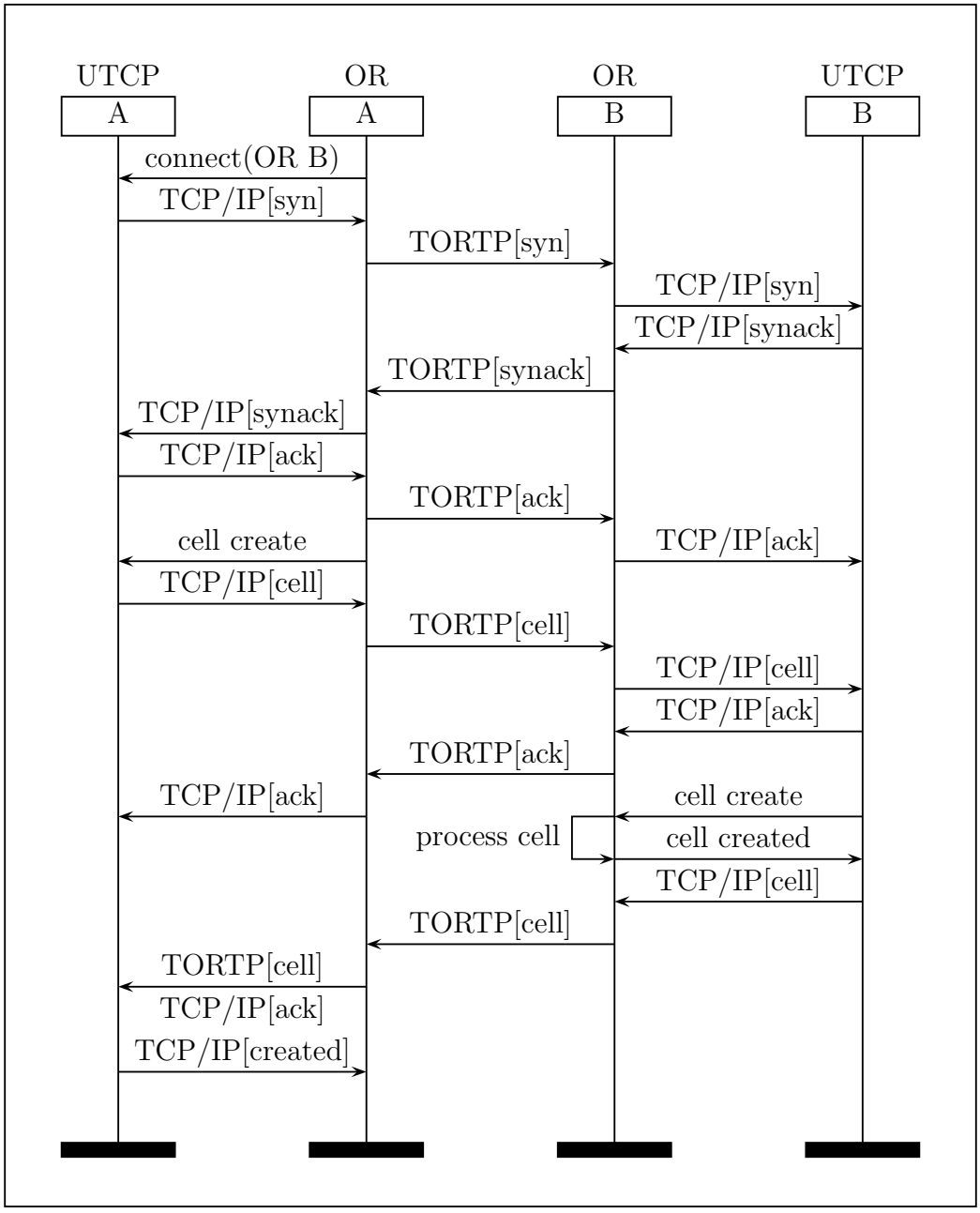

Figure 6.6: Message sequence diagram for establishing a new circuit between UDP ORs. The two columns labelled *A* correspond to a single machine, and the two columns labelled *B* correspond to a separate machine. The messages sent between the middle columns represent data sent over the network, whereas the remaining messages are local process calls. The outer columns are the user-level TCP stack and the inner columns are the ORs.

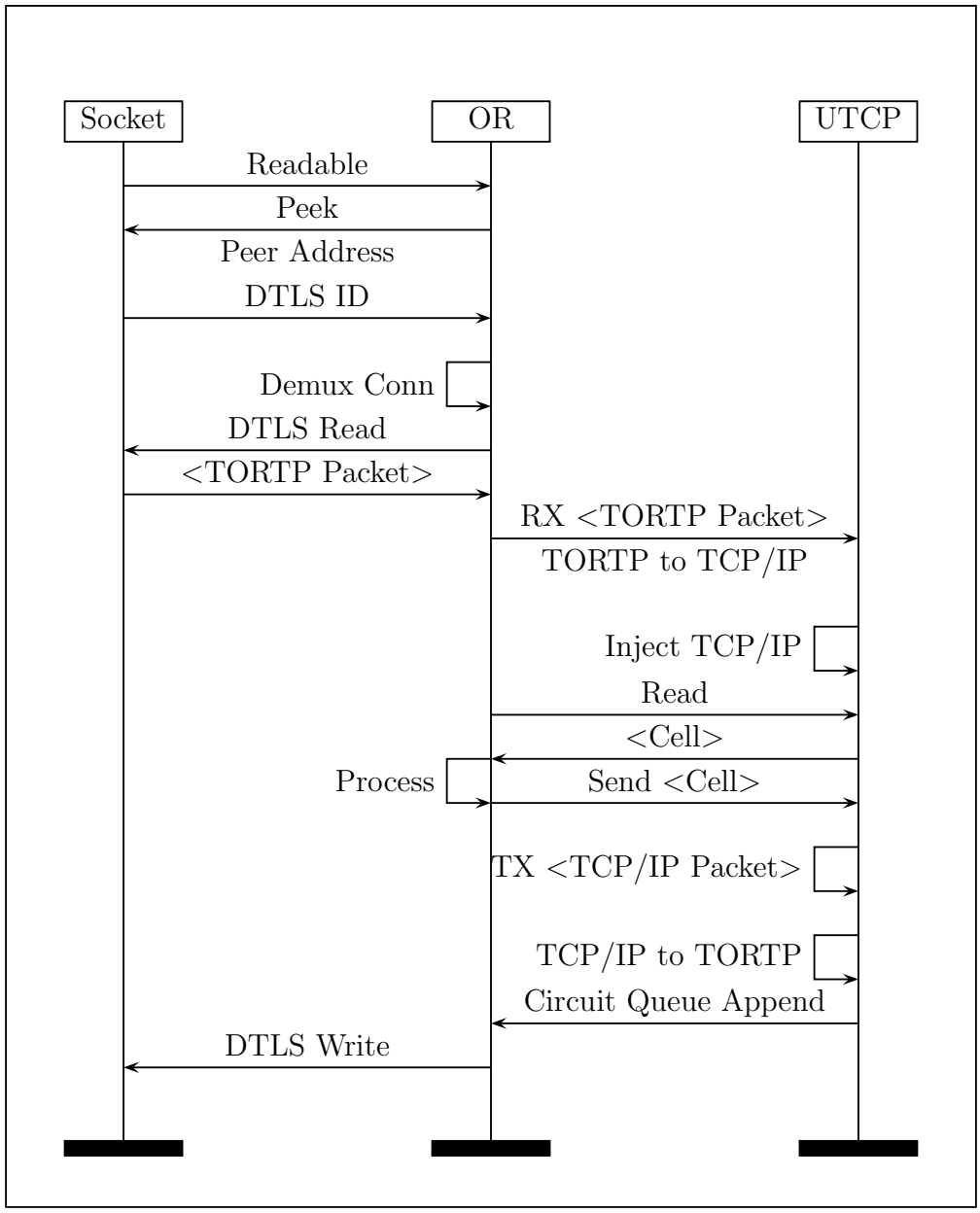

Figure 6.7: Message sequence diagram for sending and receiving data in TCP-over-UDP Tor. *Socket* corresponds to communication with the multiplexing UDP socket. *OR* corresponds to the main thread of execution in Tor that handles reads and writes. *UTCP* corresponds to the user-level TCP stack.

Chapter 7

Experimental Results

In this chapter we explore the experimental results of our proposed TCP-over-DTLS implementation. We first perform static profiling and timing experiments to ensure that our modifications are not contributing large unnecessary latency. We then perform an experiment with a single bulk data stream to determine an upper bound on throughput and latency. Finally, we contrast the throughput and latency of our modified Tor with the TCP version of Tor for packet-dropping experiments. Due to a bug in OpenSSL's DTLS implementation that translates reordered packets into dropped packets, we omit results for packet reordering.

7.1 Profiling and Timing Results

Our UDP implementation expands the datapath of Tor by adding new methods for managing user-level TCP streams and UDP connections. We profile our modified Tor and perform static timing analysis to ensure that our new methods do not degrade the datapath unnecessarily. Experiment 10 was performed to profile our new version of Tor.

Profiling results showed the expected results: AES encryption and DTLS handshaking consume the vast majority of operations. Alarmingly, it was found that the socket creation method `user_socket` takes a few milliseconds in the best case and upwards of an entire second in the worst cases. After tracing through, the poorly implemented component is a Daytona method that creates a linked list of memory yet needlessly allocates it as a giant contiguous chunk. A comment in the code indicates this is not desirable and is left as future work to repair. It will be repaired before entering into the Tor source code, but since it only effects allocation of sockets we disregarded it for our experiments.

Profiling also indicated that Daytona's implementation of `user_poll` was flawed, as it was taking an unreasonable amount of time to execute. We used `user_poll` only to determine if a socket for which we just provided a packet is now readable. Since the sockets were set to be non-blocking, we replaced the logic of polling for

Experiment 10 Timing analysis of our modified TCP-over-DTLS datapath.

1: TCP-over-DTLS Tor was modified to use `libspe` to time the following aspects of the datapath:
   - demultiplexing of a new UDP packet,
   - injection of a new packet,
   - emission of a new packet,
   - the TCP timer function, and
   - the entire datapath from reading a packet on a UDP socket, demultiplexing the result, injecting the packet, reading the stream, processing the cell, writing the result, and transmitting the generated packet.

2: Our UDP Tor was modified to compile with `gprof`.
3: The timing server was initialized.
4: The latency proxy was initialized with a latency of 50 ms and a drop rate of 0%.
5: The local network was configured to use the latency proxy.
6: The timing client connected to the timing server through a single OR.
7: Data travelled through the network for several minutes.
8: The profiling result for an active OR (i.e. on the active circuit) along with its `libspe` report was saved.

readability and reading if successful with simply attempting to read from the socket and expecting failures to be common.

The remainder of this section details the results for each timed component. The scale on the figures vary depending on the reported values.

7.1.1 Demultiplexing

Figure 7.1 shows the results for demultiplexing a UDP packet. This part of the datapath includes peeking at the DTLS ID and the sender's address, and retrieving it from a lookup table. Our experiments had a small number of peers, so it would be useful to re-examine its speed once deployed in the Tor network when managing thousands of peers. Our results indicate that demultiplexing takes a few microseconds to perform.

7.1.2 Receiving

Figure 7.2 shows the results for injecting a new TCP packet. This part of the datapath includes DTLS decryption, preprocessing, injecting the packet, and possibly invoking the transmit method to send an acknowledgement. There are four trends in the CDF, appearing at the 1st, 55th, 80th, and 95th percentiles. These correspond to receive operations that require different amounts of time to process. For

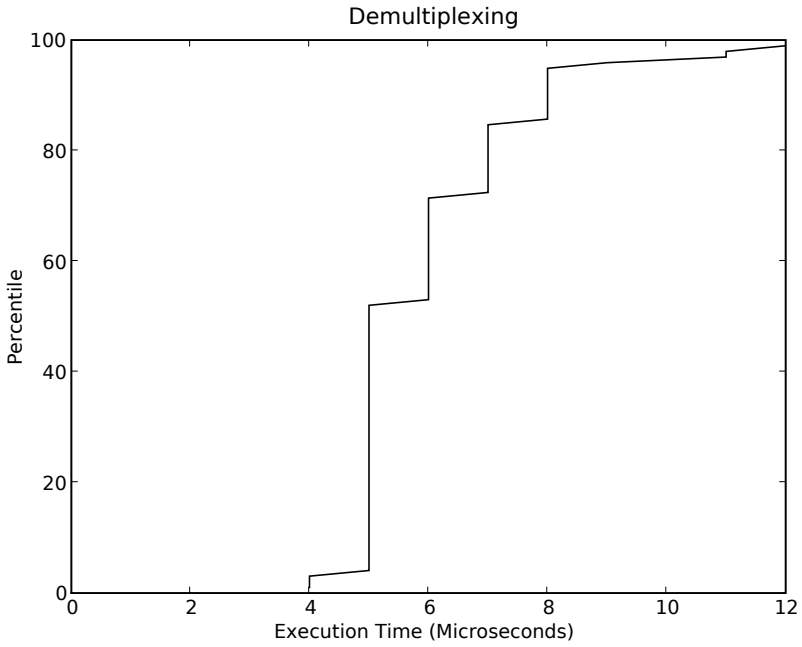

Figure 7.1: Time to demultiplex a UDP packet (CDF).

instance, operations that trigger sending acknowledgements in return will require significantly more time to encrypt the acknowledgement then simply processing a single TCP packet, and so these would be represented by the 80 to 95th percentiles. At the tail end, we see rare but lengthy durations; these likely correspond to processing synchronization packets as they require allocating memory for a new socket.

7.1.3 Transmitting

Figure 7.3 shows the results for transmitting a new TCP packet. This part of the datapath includes header translation for the packet, DTLS encryption, and dispatching it over the wire. Here we see three trends in the CDF, appearing at the 1st, 35th, and 50th percentiles. These correspond to transmitting packets of different sizes (e.g. ACKs, single cell packets, and full frames)—larger packets require longer encryption operations.

7.1.4 TCP Timer

Figure 7.4 shows the CDF for the TCP timer. The timer is not part of the main datapath, but is invoked every 10 milliseconds to perform some timer-based TCP operations: retransmission, delayed acknowledgements, etc. It increments the clock, checks for assigned work, and possibly invokes the transmission function. It only

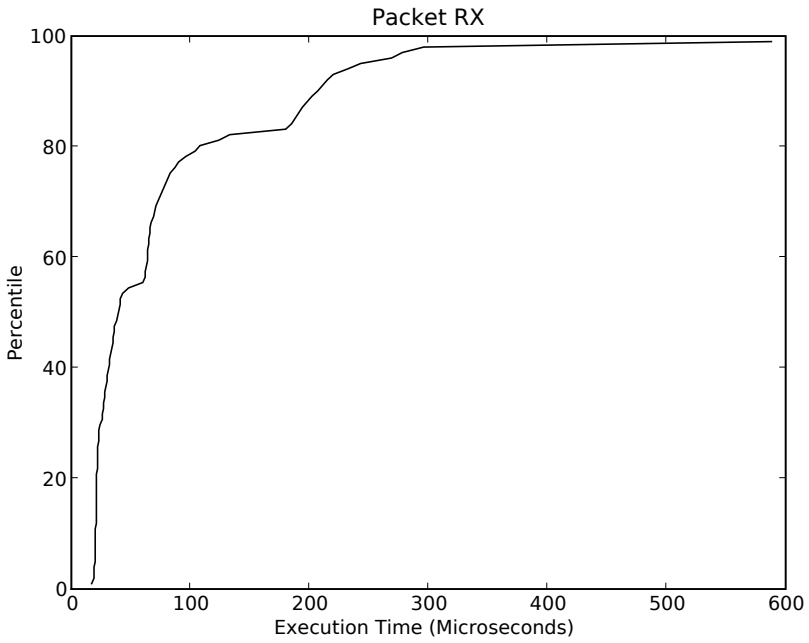

Figure 7.2: Time to inject a packet (CDF).

performs transmissions less than one percent of the time, which makes for a CDF that would indicate how long it takes to increment a counter. Therefore we create this CDF with measurements of the timer function only when it actually performs work, which amortizes to once per second. It shows a normal distribution with a timespan between 60 and 120 microseconds, which is the expected duration of a call to the TX function.

7.1.5 Datapath

Figure 7.5 shows the CDF for the duration of the entire UDP datapath. This is the end-to-end operation: demultiplexing a UDP packet, injecting (and possibly acknowledging), reading from the user-level TCP stream, processing the read cells, writing the result to the socket, and possible emitting a packet to the next hop. It shows five trends at the 1st, 15th, 35th, 70th, and 95th percentiles. As we have seen, these correspond with different execution flows that take different times in expectation. The quickest performance occurs when receiving an ACK that requires neither processing nor any transmission. The slowest performance occurs when receiving a SYN that requires allocating memory using the poor implementation of `user_socket`.

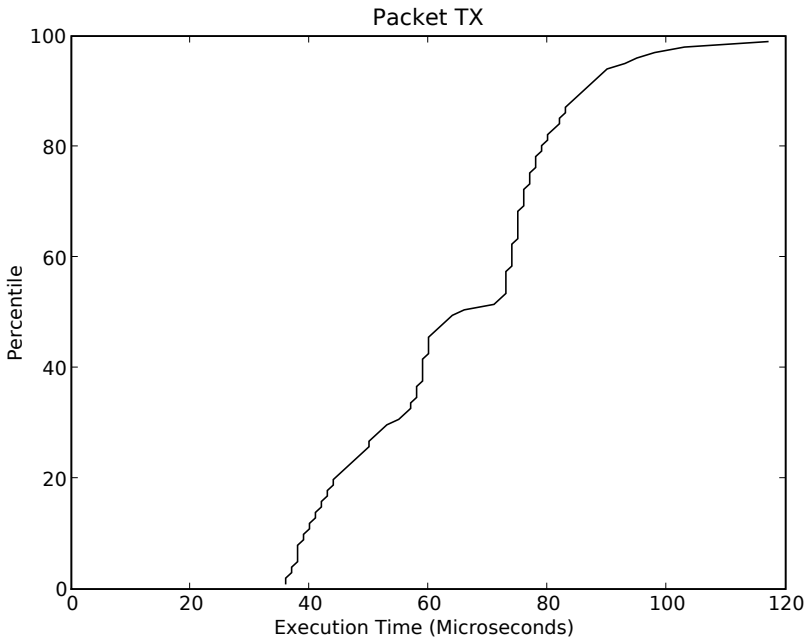

Figure 7.3: Time to emit a packet (CDF).

7.1.6 Summary

The performance results indicate that our modifications to Tor are not going to contribute significantly to latency. The TCP timer is negligible. We have increased the datapath latency to an expected value of 250 microseconds per OR, or 1.5 milliseconds along a circuit of length three and back. This is still an order of magnitude briefer than the round-trip times between ORs on a circuit (assuming geopolitically diverse circuit selection). Assuming each packet is the size of a cell (a conservative estimate as our experiments have packets that carry full dataframes) we have an upper bound on throughput of 4000 cells per second or 2 MB/s. While this is a reasonable speed that will likely not form a bottleneck, Tor ORs that are willing to denote more than 2 MB/s of bandwidth may require better hardware than the Thinkpad R60 used in our experiments.

7.2 Basic Throughput and TCP Tuning

TCP tuning is an art in its own right, and producing the optimal TCP stack suited for Tor's purposes is left as substantial future work. However, initial experiments comparing basic throughput for a single client connecting through UDP Tor showed that our modified version operated with half the efficiency of the TCP version of Tor. Since the OS used the same TCP stack as our user-level stack, this observation necessitated deep and thorough inspection into the cause for this severe performance

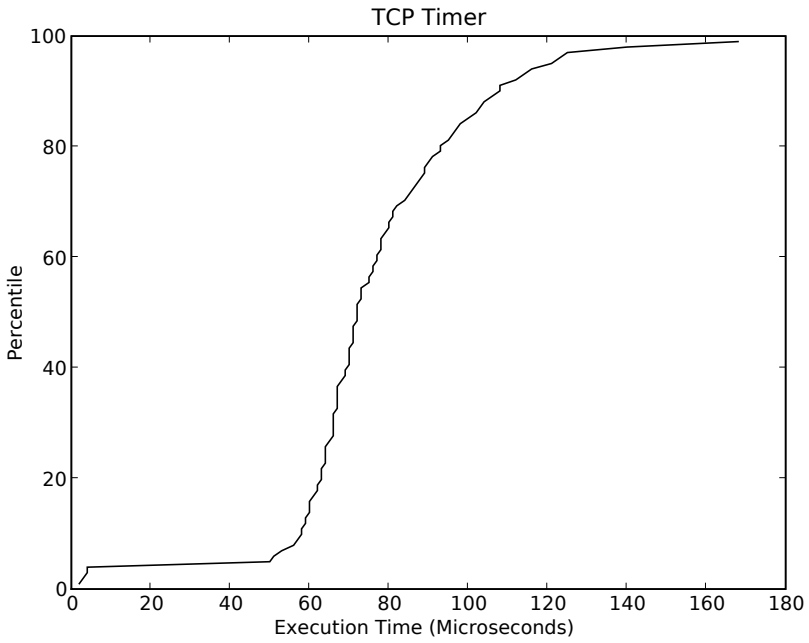

Figure 7.4: Time to perform a TCP timer task (CDF).

degradation. A number of mistakenly implemented features were uncovered, mainly dealing with how congestion control variables were incremented. These do not correspond with current TCP bugs inside the Linux TCP implementation; they occurred in the reimplementation of some system calls in user-space, or have since been resolved since Daytona's creation.

Another problem occurred with retransmissions; they were being sent too frequently. When a retransmission occurred, the congestion window dropped to one, meaning only one packet is sent in the next RTT. While slow-start gets data moving along again, unnecessary retransmissions are costly to the throughput. It was uncovered that the retransmission timeout (RTO), which is the time the TCP stack waits before retransmitting, was not being computed according to the specification in RFC 2988 [48]. Specifications mandate that it is set to the smoothed RTT plus four times its standard deviation, whereas the Linux implementation uses only one times the standard deviation. Worse, this is based on transmitting data to a peer whose operating system manages the TCP stream. Since the data enters into a UDP buffer where it is then processed internally by Tor, acknowledgements might be delayed variably by potential latency and congestion in Tor. The standard deviation becomes more relevant in this setting, and so we changed the computation of the RTO to reflect RFC 2988, and observed an increase in performance.

We perform Experiment 11 to compare the basic throughput and latency of our modification to Tor, the results of which are shown in Table 7.1. We can see that the UDP version of Tor has significantly slower throughput. As mentioned, negotiating the throughput up to this value took large amounts of TCP tuning and

Experiment 11 Basic throughput and delay for TCP and TCP-over-DTLS versions of Tor.

1: A local Tor network running six routers on a local host was configured to communicate through the `latency_proxy`. The latency was set to 50 milliseconds. Packet reordering was disabled. Packet dropping was disabled.
2: The local network uses first the original TCP version of Tor, and then our modified TCP-over-DTLS version of Tor.
3: Two OPs are configured to connect to the Tor network using the `latency_proxy` along a fixed, shared circuit.
4: A delay timing client connects through one OP to a timing server to sporadically request the time to measure latency. The results indicate the base latency imposed by the `latency_proxy`, as there is no other data being sent through the network.
5: A throughput timing client connects through one OP to a timing server to begin a bulk stream transfer to measure throughput.
6: Measurements for delay were collected in the presense of other traffic, and recorded separately from those of step 4.
7: Data was collected for over a minute, and each configuration was run a half dozen times to obtain confidence intervals.

Configuration	Network Throughput	Circuit Delay	Base Delay
TCP Tor	176 ± 24.9 KB/s	1026 ± 418 ms	281 ± 12 ms
TCP-over-DTLS Tor	111 ± 10.4 KB/s	273 ± 31 ms	260 ± 1 ms

Table 7.1: Throughput and delay for different reordering configurations. The base delay figures are measurements taken in the absence of other traffic, while the circuit delay figures are the measurements while bulk data transfer is occurring on another circuit.

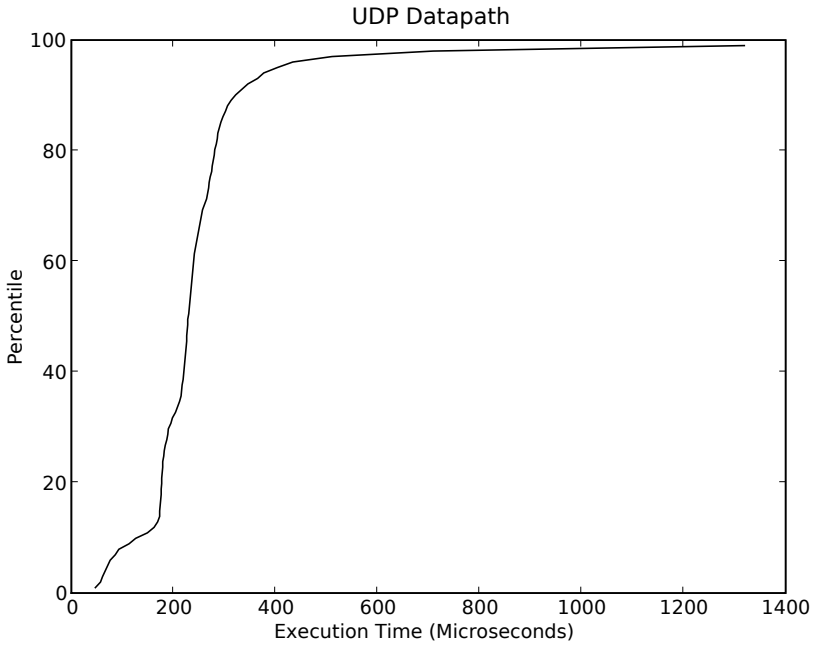

Figure 7.5: UDP Datapath Duration (CDF).

debugging the user-level TCP stack. In particular, errors were uncovered in Daytona's congestion control implementation. While there may be slight degradation in performace when executing TCP operations in user-space instead of kernel-space, both implementations of TCP are based on the same Linux TCP implementation. We would expect comparable throughputs as a result, and predict that with more effort to resolve outstanding bugs, or the integration of a user-level TCP stack heavily optimized for Tor's needs, the disparity in throughputs will vanish. We discuss this further in the future work section.

The circuit delay for a second stream over the same circuit indicates that our UDP version of Tor vastly improves latency in the presence of a high-bandwidth circuit. When one stream triggers the congestion control mechanism, it does not cause the low-bandwidth client to suffer great latency as a consequence. In fact, the latency observed for TCP-over-DTLS is reflected by the base latency imposed by the `latency_proxy`. TCP Tor, in contrast, shows a three-and-a-half fold increase in latency when the circuit that it multiplexes with the bulk stream is burdened with traffic. The disparity in latency for the TCP version means that information is leaked: the link between the last two nodes is witnessing bulk transfer. This can be used as a reconnaissance technique; an entry node, witnessing a bulk transfer from an client and knowing its next hop, can probe potential exit-nodes with small data requests to learn congestion information. We discuss this further in the future work section.

We conclude that our TCP-over-DTLS, while currently suffering lower throughput, has successfully addressed latency introduced by the improper use of the con-

Version	Configuration	Network Throughput (KB/s)	Circuit Thrput (KB/s)	Thrput Degr.	Effective Drop Rate
TCP-over-DTLS	No dropping	284 ± 35	47.3 ± 5.8	0 %	0 %
	0.1 % (remain.)	261 ± 42	43.5 ± 7.0	8 %	0.08 %
	0.1 % (shared)	270 ± 34	45.2 ± 5.6	4 %	0.03 %
TCP	No dropping	221 ± 6.6	36.9 ± 1.1	0 %	0 %
	0.1 % (remain.)	208 ± 14	34.7 ± 2.3	6 %	0.08 %
	0.1 % (shared)	184 ± 17	30.8 ± 2.8	17 %	0.03 %

Table 7.2: Throughput for different dropping configurations. "Thrput" stands for throughput. "Degr" stands for degradation.

gestion control mechanism. We expect that once perfected, the user-level TCP stack will have nearly the same throughput as the equivalent TCP implementation in the kernel. The response latency for circuits in our improved Tor is nearly independent of throughput on existing Tor circuits travelling over the same connections; this improves Tor's usability and decreases the ability for one circuit to leak information about another circuit using the same connection through interference.

7.3 Multiplexed Circuit with Packet Dropping

Packet dropping occurs when a packet is lost while being routed through the Internet. Packet dropping, along with packet reordering, are consequences of the implementation of packet switching networks and are the prime reason for the invention of the TCP protocol. In this section, we perform an experiment to contrast the effect of packet dropping on the original version of Tor and our improved version.

We hoped also to include results for packet reordering in addition to packet dropping. Unfortunately the DTLS library currently does not behave properly in the event of a reordered packet. A *bona fide* attempt to repair DTLS was made, but it proved cumbersome (and tangential to the content of this thesis) and so a bug report was made to the OpenSSL project. As such, all reordered packets in TCP-over-DTLS have the same effect as a dropped packet making experimental quantification meaningless. This is a significant limitation with DTLS for our purposes[1], which will postpone deployment of our proposed TCP-over-DTLS transport layer until resolved.

We performed Experiment 12 to investigate the effect of packet dropping. The results are presented in Tables 7.2 and 7.3. We reproduce our results from Tables 5.2 and 5.3 to contrast the old (TCP) and new (TCP-over-DTLS) transports.

[1] Any packet reordering triggers congestion control throttling inappropriately, as the TCP stack falsely believes it has congested the network to the point where packets are dropping.

Experiment 12 Determining the effect of packet dropping on circuit multiplexing for TCP-over-DTLS Tor.

1: A local TCP-over-DTLS Tor network of six routers on a single host was configured to communicate through the `latency_proxy` to control packet dropping. The latency was set to 50 milliseconds for all experiments. Packet reordering was disabled.
2: Eight clients built circuits that were fixed so that the second and third ORs were the same for each client, but the first hop was evenly distributed among the remaining ORs. Figure 5.21 illustrates this setup.
3: There were three runs of the experiment. The first did not drop any packets. The second dropped 0.1% of packets on the shared link, and the third dropped 0.1% of packets on the remaining links. As with Experiment 8, this creates a disparity between effective drop rates, which we report.
4: The ORs were initialized and then the clients were run until circuits were established.
5: Timing clients connected to each OP and used Tor to tunnel their connection to a timing server running on the local machine.
6: Data was collected for over a minute.

Interestingly, throughput is much superior for the TCP-over-DTLS version of Tor. This is likely because the TCP congestion control mechanism has less impact on throttling when each TCP stream is separated. One may back off, but the others will continue sending, which results in a greater throughput over the bottleneck connection. This is reasonable behaviour since TCP was designed for separate streams to function over the same route. If congestion is a serious problem then multiple streams will be forced to back off and find the appropriate congestion window. Importantly, the streams that send a small amount of data are much less likely to need to back off, so their small traffic will not have to compete unfairly for room inside a small congestion window intended to throttle a noisy connection. The benefits of this is clearly visible in the latency: cells can travel through the network considerably faster in the TCP-over-DTLS version of Tor.

The TCP-over-DTLS version has its observed throughput and latency affected proportionally to packet drop rate. It did not matter if the drop was happening

Version	Configuration	Average Latency	Latency Degradation	Effective Drop Rate
TCP-over-DTLS	No dropping	428 ± 221 ms	0 %	0 %
	0.1 % (remaining)	510 ± 377 ms	20 %	0.08 %
	0.1 % (shared)	461 ± 356 ms	7 %	0.03 %
TCP	No dropping	933 ± 260 ms	0 %	0 %
	0.1 % (remaining)	983 ± 666 ms	5.4 %	0.08 %
	0.1 % (shared)	1053 ± 409 ms	12.9 %	0.03 %

Table 7.3: Latency for different dropping configurations.

on the shared link or the remaining link, since the shared link is not a single TCP connection that multiplexes all traffic. Missing cells for different circuits no longer cause unnecessary waiting, and so the only effect on latency and throughput is the effect of actually dropping cells along circuits.

7.4 TCP Censorship Attack

For a final experiment, we performed the TCP censorship attack on the TCP version of Tor. This experiment is done somewhat for fun, as there is no evidence that this attack is currently being mounted against traffic between ORs. However, the Chinese government does censor broad segments of the Internet, including Tor's webpage [39], using this technique. Ostensibly, one of Tor's *raisons d'être* is to fight exactly this censorship.

The Chinese government performs this attack by sending spurious TCP packets with the reset flag set [11]. We perform this attack by modifying the `latency_proxy` to occasionally set the reset flag. Our TCP-over-DTLS implementation removes this attack vector by securing the important components of the TCP header along with the TCP payload—an adversary cannot create, modify, or replay a TCP header in our system. We admit that a censorship attack on encrypted traffic is still possible if the adversary controls the router: they simply fail to forward the traffic, or perturb the payload so that the decryption fails. However the reset attack is employed because it can function using only a sample of the traffic. When a connection is reset, then the peers' data flow is impeded for some time after, whereas blocking UDP traffic requires finding and censoring every packet being sent.

We perform Experiment 13 to examine the effect of false TCP resets between ORs in TCP Tor.

Experiment 13 Determining the behaviour of Tor during a censorship attack using the reset flag attack vector.

1: The latency proxy was modified to set the reset flag in TCP headers on packets destined for a specific OR port. They would set the flag after 1000 successful packet deliveries.
2: The latency proxy was initialized with a latency of 50 ms and a drop rate of 0%.
3: The local network was configured to use the latency proxy and run until circuits are established.
4: Six clients formed circuits with a common link between a pair of ORs. This shared link is the link being subjected to the censorship attack.
5: Each client begins a bulk data transfer from a timing server.

Our experiment revealed that the TCP version of Tor is not robust against this attack. Data would flow successfully until the reset packet was injected, at which point the connection was removed and all circuits along the path stalled. The TCP

connection being tunnelled through did not seem to resume on its own, and data only resumed being transferred when the client application was restarted. This attack cannot succeed in the UDP version of Tor: any UDP packet that is successfully decrypted will make forward progress in the tunnelled TCP connection, and the adversary is unable to make modifications to the encrypted and authenticated payload.

7.5 Summary

In this chapter we performed experiments to explore our new TCP-over-DTLS transport layer for Tor. We performed a static timing analysis of our modified datapath and determined that it will neither form a bottleneck nor contribute noticeably to latency. We compared throughput and latency for our modified Tor with the existing Tor, and found that latency is improved substantially with our modified version when a low-bandwidth client communicates alongside a high-bandwidth client. Throughput is worse for bulk transfer for a single circuit in our modified version, but we suspect that it can be improved further with tuning. We found that many multiplexed circuits had much better throughput in our modified version than TCP Tor, and unlike TCP Tor, our version had no cross-circuit interference when packets were dropped on highly multiplexed streams.

Chapter 8

Conclusions

The final chapter of this thesis presents a number of directions for future work, including steps that are needed to be done before our TCP-over-DTLS transport can be deployed. This is followed by concluding remarks about our contributions.

8.1 Future Work

The Tor network is already an area of active research. This thesis adds future research directions, which we present in this section. In particular, we discuss the need for real-world experiments for our deployment, emphasize the need to improve OpenSSL's DTLS implementation, and consider various optimizations of our extensions. We also consider the feasibility of an obvious extension: using TCP stacks only at the end points of a circuit and using UDP to forward all traffic otherwise.

8.1.1 Real-World Benefits

We hope that our TCP-over-DTLS transport layer results in tangible improvements for the real Tor network, but we must begin to use it in the real-world to test our hypothesis. Our changes require intrusive and thorough testing to ensure they function indefinitely without failing, and are resilient to all network problems and attacks. Then our transport layer can be introduced into the stock Tor release and encourage Tor node operators to update. Once a majority of users begin to use UDP connections, research can explore whether our proposal has been a benefit to the Tor network. It will be difficult to isolate all the variables with the live Tor network (such as the number of ORs and the ratio of users to ORs), but hopefully the trend will be sufficiently significant to ascertain a conclusion. Additionally, once most ORs use UDP, then we can determine if the reduced demand on open sockets solves the problem of socket proliferation on some operating systems.

8.1.2 Improving DTLS

A number of bugs with OpenSSL's deployed DTLS implementation have been uncovered in the process of implementing our TCP-over-DTLS transport layer. They have been reported to the OpenSSL team, but remain unresolved. Most notable is the packet reordering flaw that prevents our improvement from being deployed in the real world; a functional and secure UDP transport layer is necessary for our purposes.

Moreover, it would be good to standardize the concept of a single UDP port multiplexing many DTLS connections. We implemented our own BIO layer that included a stream ID before the actual data. This was used in Tor to choose which TLS object will attempt to decrypt the received data. It would be useful if DTLS could manage all TLS objects that are being shared over the same port internally, as multiplexing data for UDP sockets is a fundamental application of UDP. (The authors of DTLS argue that DTLS is worthwhile to use for applications that would otherwise require opening a great many sockets, but provide no automatic support for this multiplexing.)

We also must encrypt each acknowledgement header using DTLS. This process automatically increases the size of an acknowledgement message by a few fold. Some of this increase is necessary padding to allow a 128-bit block cipher to operate. Since this is currently unutilized data, it might be useful to have a special acknowledgement packet that contains acknowledgements for each different TCP circuit multiplexed over the same connection. This would reduce the number of acknowledgements that need to be sent by amalgamating them all onto the same packet, but would require modifying the TCP stack to be aware that different streams are being acknowledged in order to prevent redundant acknowledgements.

8.1.3 Optimized TCP Stack

Our TCP-over-DTLS makes use of a user-level TCP stack that is almost entirely Linux's TCP implementation. It is expected that such a general-purpose TCP implementation is not optimized for any particular application, and it is conceivable that Tor's use of TCP has unique characteristics that may make certain optimizations worthwhile. For example, all reads and writes occur for fixed-size cells. Considering that the stack is used uniquely for Tor's own sockets, we can make changes to the TCP stack without impacting interoperability with other applications. For instance, we can save bandwidth by representing sequence and acknowledgement numbers in terms of cells and not bytes (provided packets contain cell-aligned payloads). Direct access to the TCP stack allows us to experiment with a great number of TCP tuning parameters to improve performance. We uncovered a couple of mistakes in the implementation while simply making the user-level TCP stack perform comparably to the kernel's implementation—it is likely possible to optimize the user-level stack further.

Another obvious optimization is based on the observation that, in our application, TCP sockets are read from immediately after inserting a new packet. It should be a straightforward modification to use a return value from the packet injection routine to indicate if the TCP payload in the packet is exactly the data that would be returned from a read operation. TCP would use the header to update state, compute metrics, generate acknowledgements, etc., but it would not actually copy the payload into its own buffers. The calling function that injected the packet, after receiving an indication from the TCP stack that the payload contains in-order data, would then simply use the TCP payload of the packet it injected instead of performing a subsequent read call.

8.1.4 TCP Stack Memory Management

Another limitation to which we briefly alluded was the memory requirements in creating new sockets. This is often a slow operation due to its sub-optimal implementation of acquiring a large contiguous block of memory. In its current state, memory management is a significant problem—each user-level socket consumes nearly 10 MB of memory. This means that opening one thousand sockets will result in failed memory allocations. Even if it were possible, memory thrashing and power consumption become unnecessary burdens. Therefore we must address and repair the memory management before we can deploy user-level TCP stacks. We know that we will be needing thousands of sockets to buffer fixed-size cells of data, but data is only buffered when it arrives out-of-order or has not been acknowledged.

It is reasonable to believe we can achieve this goal using less than 10 MB per socket using dynamic memory management, such as a shared cell pool. Data needs to be buffered when it arrives out-of-order, and when it is waiting to be acknowledged. If we assume data is read immediately after arriving, does not arrive out of order, and is not dropped, then a Tor node that is willing to volunteer 1 GB/s over (on average) 200 ms round-trip links, would need to buffer 200 MB of unacknowledged data—equivalent in memory consumption to 20 open sockets in our implementation. Clearly, 1 GB/s and 200 ms are exaggerated values, and so we conclude that the ability to open many sockets without requiring 10 MB per socket is a reasonable goal. While packet dropping, congestion control, and out-of-order delivery will increase memory requirements, these factors are unlikely to be a sudden problem for all sockets simultaneously (unless the computer loses its Internet connection in which case no new data will be arriving).

Another goal would be the unification of cell memory throughout Tor. Instead of copying cells from various buffers, each cell that enters Tor can be given a unique cell from the cell pool for its memory until it is no longer needed. A state indicates where this cell currently exists: input TCP buffer, input Tor buffer, in processing, output Tor buffer, output TCP buffer. This ensures that buffers are not allocated to store empty data, which reduces the overall memory requirements. Moreover it localizes memory for cells that enter the system at the same time—this will result

in less paging when accessing data for different sockets. Since ORs are routers, they need to continually access the memory for different sockets and so spreading them widely across memory pages is inefficient. Each cell also keeps track of its socket number, and its position in the linked list of cells for that socket. While each socket must still manage data such as its state and metrics for congestion control, this is insignificant as compared to the current memory requirements. This permits an arbitrary number of sockets, for all operating systems, and helps Tor's scalability if the number of ORs increases by orders of magnitude.

This approach results in the memory requirements of Tor being a function of the number of cells it must manage at any time, independent of the number of open sockets. Since the memory requirements are inextricably tied to the throughput Tor offers, the user can parameterize memory requirements in Tor's configuration just as they parameterize throughput. A client willing to denote more throughput than its associated memory requirements will have its contribution throttled as a result. If network conditions result in a surge of memory required for Tor, then it can simply stop reading from the UDP multiplexing socket. The TCP stacks that sent this unread data will assume there exists network congestion and consequently throttle their sending. More usefully, we can use the window size in the TORTP header to reflect the size of our cell pool. Doing this in such a way that does not admit a trickle-flood attack may not be possible—at the very least, granular values such as empty and full, will be necessary.

8.1.5 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) is a message-based transport protocol. It provides similar features to TCP: connection-oriented reliable delivery with congestion control. However, it adds the ability to automatically deliminate messages instead of requiring the receiving application to manage its own delimiters. The interface is based on sending and receiving messages, which is appropriate for Tor's cell-based transport.

SCTP also adds a feature well-suited to our purposes—multiple streams over the same connection. SCTP allows multiple independent ordered streams to be sent over the same socket. We can use this feature to send cells from different circuits on different streams. The in-order delivery guarantee is only provided for messages sent on the same stream, which is exactly the behaviour we want for cell from different circuits.

While SCTP is not as widely deployed as TCP, the concept of using a user-level SCTP stack suited for Tor remains feasible. Experimentation must be done to determine if SCTP's congestion control mechanism, which shares metrics and computations across all streams, acts fairly towards streams that send little data when compared to the streams that invoked congestion control.

8.1.6 Optimize Demultiplexing of Circuits

Demultiplexing occurs three times within Tor: DTLS ID and address for retrieving the DTLS objects for decryption, Port ID for retrieving the UTCP circuit for TORTP to TCP/IP translation, and the circuit ID within the cell itself for Tor's purposes. While DTLS ID indicates the connection and is specified outside the encrypted TLS payload, the remaining two contain redundant information that identifies the circuit on the connection. It would be preferable to streamline these identifiers into a single identifier that specifies all this information, however this complicates the logic. Indeed, we re-multiplex the circuit's cells back onto the connection's buffer, where they are later demultiplexed by circuit ID, simply to reuse Tor's existing datapath as much as possible. TORTP packets carry a Port ID that implies the circuit ID for all cells they carry. Tor could remove circuit IDs from the cells when they write them to the user-level socket, and insert circuit IDs back onto cells when reading from the user-level socket.

8.1.7 Probing Attack

The observation of TCP stream correlation motivates a privacy-degrading attack against Tor. We observed that a bulk transfer effects the delay of a timing probe along shared circits. This leaks information about the congestion along a link. It may be possible to probe a number of routers to determine which are experiencing a congested link that might correspond to the stream that we know we are relaying to an OP. Reducing the anonymity set for an anonymous OR is a privacy concern.

Tor rotates its circuits every ten minutes. Suppose the entrance node notices a bulk transfer when it begins, and probes various ORs to determine the set of possible third ORs. It could further reduce this set by re-probing after nine minutes, after which time most of the confounding circuits would have rotated to new links.

It would be interesting to see how feasible this attack is in the wild. Perhaps probing more than a hundred nodes is infeasible, however it does give a 10% chance of success that can be increased by performing this attack whenever possible. If it is a feasible attack, then it would give an impetus to upgrade the Tor network to our TCP-over-DTLS transport, which reduces cross-circuit interference. The feasibility of this attack on our improved Tor would then be useful to examine.

This is similar to the trickle-flood attack of Murdoch et al. [46]. In their attack, they observe a circuit in Tor (the trickle), and selectively perform a denial-of-service attack against every OR out-of-band (the flood). This results in observable latency on the Tor circuit when the correct OR is flooded, revealing the final node. Our attack is more discreet to perform but less generic in the circuits it targets. The flood is the bulk transfer that is being performed by the client, and the trickle is our in-band probing of circuit latency for each OR.

8.1.8 UDP Forwarding

One extension to our TCP-over-DTLS transport would be to use user-level TCP stacks at the endpoints of a circuit, and have middle nodes only forward the UDP payload (i.e. the TCP packet) without providing it to its local TCP stack. The middle node would inspect the payload to retrieve the cell, determine the circuit for which it is destined, fetch the onion key, perform the encryption/decryption operation, and finally forward the TCP packet to the last hop. This datapath will perform the operations on the TCP/IP payload directly. The next hop will provide it to its TCP stack, which generates an acknowledgement message that will also be forwarded by the middle node back to the first node. The first hop will be responsible for resending data if a packet drops, which will be again sent through the middle node. This was the mechanism used by the Freedom Network [26] and has also been suggested for Tor [75].

Theoretical Benefits

This strategy has a number of benefits in computational complexity and network behaviour. Computationally, it saves the middle node from performing unnecessary operations: packet injection, stream read, stream write, packet generation, and packet emission. It also removes the responsibility of the middle node to handle retransmissions, which means a reduction in their memory requirements. The initial endpoint of communication will be responsible to retransmit the message if necessary. We have shown that computational latency is insignificant in Tor, so this is simply an incidental benefit.

The tangible benefit of UDP forwarding is to improve the network by allowing the ORs to function more exactly like routers. When cells arrive out of order at the middle node, they will be forwarded regardless, instead of waiting in input buffers until the missing cell arrives. Since the middle node does not need the cells in order, this delay is unnecessary. Moreover, by having the sender's TCP stack view both hops as a single network, we alleviate problems introduced by disparity in network performance. Currently, congestion control mechanisms are applied along each hop, meaning that an OR in the middle of two connections with different performance metrics will need to buffer data to send over the slower connection. The middle node's TCP stack will perform congestion control properly, but the original sender will continue sending rapidly as they view the connection as high-bandwidth. Tor provides its own congestion control mechanism, but does not have the sophistication of TCP's congestion control due to the inability to control kernel-level TCP variables. By using ORs to forward UDP packets end-to-end, the path through Tor will be viewed as a single pipe with congestion control applied based on its aggregate performance. TCP is designed to operate optimally in this manner, which motivates the hypothesis that network performance will improve with UDP forwarding.

Theoretical Concerns

We require experimentation to determine if this proposal is actually beneficial. Previous research has stated that this results in a reduced memory requirement for middle nodes [75]. However, the endpoints will each have increased delay before messages are acknowledged, which means that data will be buffered in TCP for a longer time. The capacity of the TCP connection between the end points, measured by its bandwidth-delay product, will increase. We expect an equilibrium for total system memory requirements. Worse, the memory requirements shift from being evenly distributed to occurring only on exit-nodes—who are already burdened with extra responsibilities. Since a significant fraction of Tor nodes volunteer only to forward traffic, it is reasonable to use their memory to ease the burden of exit-nodes. Additionally, reordered packets will no longer cause delay at middle nodes. However, any out-of-order packets must be reordered before the data can be read, and so the performance benefit may be negligible.

The increased round-trip time of the circuits require employing TCP extensions for long-delay paths [34]. Circuits with long delays will also suffer reduced throughput, and so using congestion control on as short a path as possible will optimize performance. If a packet is dropped along the circuit, the end point must now generate the retransmission message, possibly duplicating routing effort during redelivery. Since OR bandwidth is a limitation, it may be the case that increasing retransmission bandwidth is actually worse than using TCP at each hop and perfecting Tor's own congestion control mechanism. Tor's congestion control was limited by the fact that it cannot set the congestion window inside the operating system from user space—our user-level TCP stack in Tor allows direct access of TCP variables. It may be more efficient to have nodes along a circuit return their CWND for the next hop, and have each node use the minimum between their CWND and the next hop's CWND. Each node then optimizes their sending while throttling their receiving.

Implementation

To implement UDP forwarding for a Tor circuit, the middle node will have a connection from the first node created using a CREATE cell, and will then be told to connect to the next OR using an EXTEND cell. The middle node will establish a new user-level TCP connection to the third OR, and associate the two connections for the circuit in memory. Later, when data arrives that will be relayed, the node performs the circuit and onion key look ups and then performs the encryption/decryption operation inside the packet. The node then changes the sequence and acknowledgement numbers to reflect the next data segment expected by the next node.

To access the circuit ID, we will need to ensure that packets always contain payloads that are cell-aligned. Careful reckoning of sequence and acknowledgement numbers will be needed since not all cells are forwarded down the circuit but are

instead delivered locally. Cell-aligned payloads have been successfully implemented with our UTCP stack: we simply set the maximum segment size (MSS) of the user-level connection to be exactly the size of the maximum number of entire cells that can be sent over the actual link. The user-level TCP stack will never emit a packet that contains more payload data than the MSS. Since the payload will arrive either intact or not at all, we have that acknowledgements will always be for multiples of entire cells, and so any retransmission must also be cell-aligned. The TCP stack generated packets by adding unacknowledged data to the payload until the next addition will exceed the MSS. Since all the data buffers are cell-aligned, and the MSS is cell-aligned, our TCP stack will either have insufficient data or exactly fill an MSS; either way ensures that cells do not fragment. We observe that the TCP probe function will violate this principle of buffering cell-aligned data, and so another mechanism will have to be implemented in our TCP stack. This is a reasonable change to make because our stack will only interact with other instances of Tor running our stack. With payloads that are guaranteed to be cell-aligned, we can now use fewer bits for sequence and acknowledgement numbers sent over the wire by expressing them as multiples of cell sizes.

ORs that intend on using this system must have an maximum transmittable unit (MTU) at least the size of a cell plus TORTP and DTLS headers to avoid fragmentation. Fragmentation obviates all improvements of UDP forwarding since an entire cell must be available to perform AES and the circuit look up operations at every OR, even if they will just be quickly forwarded. It is unreasonable to fragment and assemble half a cell for the link that is intending on simply performing UDP forwarding; it would require performing nearly the same logic that is already handled by the TCP implementation. Moreover, it would interfere with the TCP statistics pertaining to the round trip time metrics if some partial cells are buffered while their missing piece is forwarded soon after arrival. If the MTU between two ORs is slightly less than two full cells plus the TORTP and DTLS overhead, then packets will waste a large amount of available bandwidth. This is likely significantly less efficient than using TCP-over-UDP at each hop without the restrictions imposed on the MSS. SCTP may find some application here—we can use SCTP's unordered delivery of entire cells as the transport mechanism to maximize usage of packets without reimplementing logic, and use TCP at the end points for stream assembly and congestion control.

Low-cost Privacy Attack

UDP forwarding may introduce an attack that permits a hostile entrance node to determine the final node in a circuit. Previously each OR could only compute TCP metrics for ORs with whom they were directly communicating. The new system would have the sender's TCP stack communicate indirectly with an anonymous OR. Connection attributes, such as congestion and delay, are now known for the longer connection between the first and last nodes in a circuit. The first node can determine the RTT for traffic to the final node. It can also reliably compute the

RTT for its connection to the middle node. By using techniques to estimate the RTT between the second node and performing an experiment to determine the RTT to every other UDP node in the Tor network, the adversary may be able to eliminate large numbers of ORs from the anonymity set deemed incapable of being the final node. If it can reduce the set of possible final hops, other reconnaissance techniques can be applied, such as selectively flooding each OR outside of Tor and attempting to observe an increased latency inside Tor [46]. By extention, all metrics that TCP computes can be amalgamated to form a privacy-devastating connection fingerprint: congestion window, slow-start threshhold, occurrence of congestion over time, standard deviation in round-trip times, etc. These can each be computed over time and as a distribution. The OR can also determine connection attributes for each UDP OR in the network and match the observed metrics to its collection of metrics. If a TCP stack is used starting with the OP, then the middle node can use the TORTP headers to simulate the TCP stack. Since they know both adjacent ORs, then connection metrics will known except for the network behaviour from the OP to the first OR. The observed RTT that is unaccounted for by the sum of the RTTs to known ORs indicates the geographic distance between the OP is and the first OR. Other connection metrics are based on last-kilometer network behaviour, which may elucidate the OP's ISP.

This privacy degradation in the all-UDP possible future network is hypothesized but remains untested. An analysis of the feasibility of this attack is necessary before applying UDP forwarding to the Tor network. If an entrance node can determine the exit node based on the set of TCP attributes then privacy degrades to a two-node circuit—where UDP forwarding has no purpose. Alternatives include adding more nodes to the circuit; this would create additional degrees of freedom in the connection metrics that would add noise to the data collected. However, adding an extra OR is almost certainly slower than any transport benefits of UDP forwarding. Another option is to have the middle node interfere with the connection to ensure that any metrics that are computed are sufficiently inaccurate so as to increase the anonymity set of possible final nodes. Since TCP collects and relies on these metrics to ensure that the connection behaves optimally, it is likely that destroying their accuracy will be much less efficient than simply using TCP along each hop at peak efficiency.

TCP Fingerprinting

TCP fingerprinting is a reconnaissance technique that uses variations in the implementation of TCP to determine the operating system of the peer; this is a potential loss of privacy in Tor. While TCP fingerprinting is not a concern for single-hop TCP (since any entity can perform TCP fingerprinting out-of-band knowing the IP address), the use of UDP forwarding permitted TCP fingerprinting to be done on ORs for whom the IP address is hidden. This can allow a privacy disclosure to happen between ORs that are meant to remain anonymous. Since each OR publishes their precise operating system in the network status, learning the OS of an

anonymous OR will reduce its anonymity set.

While TCP fingerprinting may be a concern for some implementations of UDP forwarding, it is not a concern for our TCP-over-UDP tunnel. This is because the user-level TCP implementation will be identical for all ORs, preventing this reconnaissance technique from reducing the anonymity set of the final hop. To ensure this is true, however, all TCP tuning and optimization discussed in this chapter must completed before deploying our improved Tor. If we permit different versions of Tor to have fingerprintable TCP stacks, then ORs that are separated will have their version of Tor fingerprintable. Each OR publishes their version, and so an adversary that can fingerprint the version of an anonymous entity can reduce its anonymity set. This is only an issue with UDP forwarding—TCP tuning and optimization can evolve iteratively in the current form without concern.

8.1.9 Windows ORs and Reputation

Our work permits Windows computers to volunteer as ORs. However, this comes with a caveat: Windows machines are vastly more prone to malware and compromise than stalwart Unix-based machines. Most OR private keys are stored in plaintext on the host, which means that Windows ORs are inherently more prone to having their keys compromised than Unix-based machines. Additionally, large numbers of Windows computers are involved in various botnets. An adversary who controls a botnet could now have each computer in the botnet volunteer as a hostile OR and remove any privacy for circuits are built entirely through machines they control. Reputation systems and research in countering Sybil attacks [78] will mitigate this risk, but it is important to mention its existence.

8.1.10 Web Browsing Mode for Tor

Tor's primary use is for web browsing, and this is the application that has seen the most effort towards developing usability tools for end-users [18]. Latency is critical for interactive web browsing; most people who do not require Tor to circumvent censorship tend not to use Tor if it introduces noticeable latency. We identified in Section 5.1 that all HTTP GET requests require twice the necessary latency when performed through Tor—a TCP connection is first established, the result is returned, and then the HTTP GET is posted. This can be streamlined into a single operation without destroying Tor's ability to anonymize TCP traffic in general.

Moreover, when clients make an HTTP request to a server, they are likely to make a few more requests soon after—HTML pages encode other resources such as images which must then be loaded. Given that these requests will, with high probability, soon be made by the client, it may be useful if the exit-node performs all the work necessary to load the webpage, compresses it to preserve bandwidth (HTML is highly compressable), and sends the result as a package which is interpreted by the

client. This technique is likely to save a great deal of latency for interactive web-browsing, and encourage greater adoption of Tor by casual users. We have shown in Section 5.2 that computational latency is insignificant, and so the addition of a compression operation should not be detrimental. The reduced data size, resulting in faster AES and network operations, may even compensate for the compression step. Techniques much like this were performed by early graphical web browsers for wireless handheld devices like the Palm Pilot [20]. The Freedom network also used this technique to improve performance for anonymous web browsing [26].

8.2 Summary

Anonymous web browsing is an important step in the development of the Internet, particularly as its grows ever more inextricable from daily life. The Internet is a highly democratic, participatory, inclusive, and collective medium, creating the global village of which McLuhan imagined [42]. The ability for anyone to begin publishing as they will, and for anyone else to choose to absorb as they will, has made the Internet a medium unlike any other. The Internet also provides its users with the ability to be unseen as they publish their remarks. This has given rise to a fallacious belief that the Internet affords privacy, and many independent journalists have painfully learned that limits to their freedom of expression in other media extend to the Internet as well. Fortunately, not all computers on the Internet geographically reside in authoritarian nations with strict bans on government criticism. Thousands of citizens, living in countries with a strong commitment to the principles of freedom of expression, have volunteered their computers to relay the Internet traffic on behalf those whose nations would imprison them for expressing their beliefs. Tor is a PET that provides Internet anonymity using volunteers to relay traffic, and using multiple relays in series to ensure that no entity (other than the client) in the system is aware of both the source and destination of messages.

Relaying messages increases latency since traffic must travel a longer distance before it is delivered. Latency becomes an immediate disincentive to use Tor among those who have less to fear by losing their privacy. To improve the usability of Tor, we examined where this latency occurs. An initial experiment observed the total end-to-end latency, and discounted transport latency. We learned that a substantial delay exists that is not transport latency. Further exploration determined that computational latency along the datapath is not a concern, but instead latency occurred when data sat idly in buffers due to congestion control. Since multiple Tor circuits are multiplexed over a single TCP connection between routers, we observed cross-circuit interference due to the nature of TCP's in-order, reliable delivery and its congestion control mechanisms. We wanted to demultiplex individual circuits, so that each has a unique TCP connection—but we also want to continue hiding details about traffic along each circuit from observers, and moreover we want to reduce the number of sockets that need to be opened to address scalability issues and known problems on some operating systems.

Our solution was the design and implementation of a TCP-over-DTLS transport between ORs. Each circuit was given a unique TCP connection, but the TCP packets themselves were sent over the DTLS protocol, which provides confidentiality and security to the message header. The TCP implementation is provided in user-space, where it acts like a black box that translates between data streams and TCP/IP packets. TCP/IP packets that are emitted are sent over DTLS to the peer using a single UDP socket in the operating system—this single socket multiplexes all data connections to all peer ORs. Packets that are received from this socket are forwarded to the user-level TCP stack, and the corresponding user-level sockets become readable.

We performed experiments on our implemented version using a local experimentation network and showed that the observed cross-circuit interference has vanished. We observed that one circuit using a lot of bandwidth no longer caused large latency for multiplexed circuits that need only a tiny bandwidth. This is because in our new version, these Tor circuits no longer shared a single TCP stream competing for bandwidth inside the congestion window. We also showed that dropping packets on TCP connections shared by many circuits significantly affects throughput and latency for the TCP version of Tor, but this no longer occurred with our TCP-over-DTLS version.

We detailed future work that this thesis motivates. Of particular interest were the need to make some improvements to DTLS, the need to overhaul the memory management of our user-level TCP stack, and the possibility of employing UDP forwarding in our TCP-over-DTLS system. OpenSSL's DTLS implementation has a few outstanding problems that we uncovered during our experimentation, and these must be fixed prior to any deployment. We briefly described why the limited number of sockets that can be opened is an artificial problem. We suggest a new memory management structure for the user-level TCP stack and Tor that will permit an arbitrary number of sockets and improve performance through the localization of cells on memory pages. Finally we provide a discussion of the possibility of UDP forwarding, where middle ORs on a circuit only forward the TCP/IP packets instead of inserting them into their TCP stack. We cautioned that this approach leaks more details about connection metrics to each system, which may admit devastating privacy attacks. We proposed that these concerns must be studied before any deployment of UDP forwarding can be given to individuals who use Tor to protect their privacy and their freedom.

Bibliography

[1] Adium. A free instant messaging application for Mac OS X. http://www.adiumx.com/.

[2] AN.ON Anonymity.Online. About Jondo. http://anon.inf.tu-dresden.de/help/jap_help/en/help/about.html.

[3] Anonymizer, Inc. Anonymizer FAQ. http://www.anonymizer.com/company/about/anonymizer-faq.html.

[4] Randal Archibold. A 17-Year Nightmare of Identity Theft Finally Results in Criminal Charges. *New York Times*, 2007.

[5] Ask.com. About AskEraser. http://sp.ask.com/en/docs/about/askeraser.shtml, 2008.

[6] BBC News. Massacre in Tiananmen Square. http://news.bbc.co.uk/onthisday/hi/dates/stories/june/4/newsid_2496000/2496277.stm, 1989.

[7] N. Borisov, I. Goldberg, and E. Brewer. Off-the-Record Communication, or, Why Not To Use PGP. *Workshop on Privacy in the Electronic Society*, 2004.

[8] CBC News. 45% of Canadians rebuff retailers' requests for personal info: survey. http://www.cbc.ca/consumer/story/2008/07/04/retail-privacy.html?ref=rss, 2008.

[9] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[10] Thomas Claburn. Digg Yields To The Wrath Of The Crowd. http://www.informationweek.com/news/internet/showArticle.jhtml?articleID=199203167, 2007.

[11] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies*, pages 20–35, 2006.

[12] CNN. YouTube ordered to reveal its viewers. http://www.cnn.com/2008/TECH/biztech/07/03/youtubelawsuit.ap/index.html, 2008.

[13] Cross Country Checkup. Was China the right choice for the Olympics? http://www.cbc.ca/checkup/archive/2008/080803CC.mp3, August 2008.

[14] cuil. Your Privacy. http://www.cuil.com/info/privacy/, 2008.

[15] G. Danezis, R. Dingledine, D. Hopwood, and N. Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol, 2002.

[16] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. *Proceedings of the 13th USENIX Security Symposium*, 2004.

[17] Express India. Gurgaon techie held for posting derogatory messages against Sonia Gandhi on Orkut. http://www.expressindia.com/latest-news/Gurgaon-techie-held-for-posting-derogatory-messages-against-Sonia-Gandhi-on-Orkut/311070/, 2008.

[18] Firefox Add-ons. Torbutton 1.2.0. https://addons.mozilla.org/en-US/firefox/addon/2275, 2008.

[19] Glenn Fleishman. Cartoon Captures Spirit of the Internet. http://www.nytimes.com/2000/12/14/technology/14DOGG.html?ei=5070&en=cb70a7f40e8c6c87&ex=1219636800, 2000.

[20] Armando Fox, Ian Goldberg, Steven D. Gribble, David C. Lee, Anthony Polito, and Eric A. Brewer. Experience with top gun wingman: A proxy-based graphical web browser for the 3com palmpilot. In *Proceedings of Middleware '98, Lake District*, 1998.

[21] Chris Gaither. Ending tussle, Google adds privacy link to home page. http://latimesblogs.latimes.com/technology/2008/07/google-privacy.html, July 2008.

[22] Michael Geist. Face to Face with the Great Firewall of China. http://www.michaelgeist.ca/resc/html_bkup/may22005.html, 2005.

[23] Ian Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, University of California, Berkeley, 2000.

[24] Ian Goldberg. Privacy-enhancing Technologies for the Internet, II: Five Years Later. In *Workshop on Privacy Enhancing Technologies*, 2002.

[25] Ian Goldberg. *Privacy-Enancing Technologies for the Internet III: Ten Years Later*, chapter 1 of Digital Privacy: Theory, Technologies, and Practices, pages 3–18. Auerbach, December 2007.

[26] Ian Goldberg and Adam Shostack. Freedom Network 1.0 Architecture and Protocols, 2001.

[27] Ian Goldberg, David Wagner, and Eric A. Brewer. Privacy-enhancing Technologies for the Internet. In *IEEE COMPCON*, 1997.

[28] Amy Harmon. Verizon to Reveal Customers in Piracy Case. *New York Times*, 2003.

[29] P. Howard. WIA: Blogger Arrests Around the World. http://www.wiareport.org/wp-content/uploads/wiar_2008_final.pdf, 2008.

[30] Human Rights Watch. "Race to the Bottom" Corporate Complicity in Chinese Internet Censorship. http://www.hrw.org/reports/2006/china0806/, 2006.

[31] Information Sciences Institute. RFC793 - Transmission Control Protocol. http://www.faqs.org/rfcs/rfc793.html, 1981.

[32] Internet Engineering Task Force. RFC1122 - Requirements for Internet Hosts - Communication Layers. http://www.faqs.org/rfcs/rfc1122.html, 1989.

[33] Andrew Jacobs. IOC agrees to Internet blocking at the Games. *International Herald Tribune*, 2008.

[34] V. Jacobson and R. Braden. RFC1072 - TCP Extensions for Long-Delay Paths. http://www.faqs.org/rfcs/rfc1072.html, 1981.

[35] Leslie John, Alessandro Acquisti, and George Loewenstein. Inconsistent Preferences for Privacy. In *Behavioral Decision Research in Management Conference*, 2008.

[36] Alan Kay. STEPS Toward The Reinvention Of Programming. Distinguished Lecture Series, 2008.

[37] Donald Kerr. Remarks and Q&A by the Principal Deputy Director of National Intelligence. GEOINT Symposium, 2007.

[38] Jemima Kiss. Cisco investor highlights human rights issue. *guardian.co.uk*, 2006.

[39] Isaac Mao. Torproject.org Blocked by GFW in China: Sooner or Later? https://blog.torproject.org/blog/torproject.org-blocked-gfw-china:-sooner-or-later

[40] Mike Masnick. Since When Is It Illegal To Just Mention A Trademark Online? http://www.techdirt.com/articles/20050105/0132239.shtml, January 2005.

[41] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining Light in Dark Places: Understanding the Tor Network. In *Privacy Enhancing Technologies*, 2008.

[42] Marshall McLuhan. *Understanding Media: The Extensions of Man*. The MIT Press, 1994.

[43] N Modadugu and E Rescorla. The Design and Implementation of Datagram TLS. *Network and Distributed System Security Symposium*, 2004.

[44] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster Protocol — Version 2. IETF Internet Draft, 2003.

[45] Jonathan Montpetit. Banks not reporting cybercrime, police say. http://www.theglobeandmail.com/servlet/story/RTGAM.20080529.wgtcybercrime0529/BNStory/Technology/, 2008.

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

[47] Pangea Day. About Pangea Day. http://www.pangeaday.org/aboutPangeaDay.php.

[48] V. Paxson. RFC2988 - Computing TCP's Retransmission Timer. http://www.faqs.org/rfcs/rfc2988.html, 2000.

[49] Robert Paxton. *Europe in the Twentieth Century*. Wadsworth Publishing, 2006.

[50] C. Pfleeger and S. Pfleeger. *Security in Computing*. Prentice Hall, 2003.

[51] Ponemon Institute. Airport Insecurity: The Case of Missing & Lost Laptops. http://www.dell.com/downloads/global/services/dell_lost_laptop_study.pdf, 2008.

[52] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A User-Level TCP Stack. http://nms.lcs.mit.edu/ kandula/data/daytona.pdf.

[53] Privacy Commissioner of Canada. Identity Theft: What it is and what you can do about it. http://www.privcom.gc.ca/fs-fi/02_05_d_10_e.asp, 2003.

[54] Privacy International. Leading surveillance societies in the EU and the World. http://www.privacyinternational.org/article.shtml?cmd[347]=x-347-545269, 2006.

[55] Privacy Rights Clearinghouse. A Chronology of Data Breaches. http://www.privacyrights.org/ar/ChronDataBreaches.htm, 2008.

[56] Privacy Rights Clearinghouse. Consumer and Privacy Groups Urge Google to Post a Link to Its Privacy Policy from Its Home Page. http://www.privacyrights.org/ar/Google-HomePage-Alert-080603.htm, June 2008.

[57] Reporters without Borders. Information supplied by Yahoo! helped journalist Shi Tao get 10 years in prison. http://www.rsf.org/article.php3?id_article=14884, 2005.

[58] Reporters without Borders. Another cyberdissident imprisoned because of data provided by Yahoo. http://www.rsf.org/article.php3?id_article=16402, 2006.

[59] Reporters without Borders. Still no reaction from Yahoo! after fourth case of collaboration with chinese police uncovered. http://www.rsf.org/article.php3?id_article=17509, 2006.

[60] Reporters without Borders. Yahoo! implicated in third cyberdissident trial. http://www.rsf.org/article.php3?id_article=17180, 2006.

[61] John Ribeiro. Google Defends Helping Police Nab Defamer. *IDG News Service*, 2008.

[62] Royal Canadian Mounted Police. Identity Theft. http://www.rcmp-grc.gc.ca/scams/identity_theft_e.htm, 2004.

[63] Maggie Shiels. Google accused on privacy views. *BBC News*, 2008.

[64] David Simon and Edward Burns. *The Corner: A Year in the Life of an Inner-City Neighborhood*. Broadway, 1998.

[65] W. Stevens. RFC2001 - TCP Slow Start, Congestion Avoidance, Fast Retransmit. http://www.faqs.org/rfcs/rfc2001.html, 1997.

[66] Sarah Stirland. Cisco Leak: 'Great Firewall' of China Was a Chance to Sell More Routers. http://blog.wired.com/27bstroke6/2008/05/leaked-cisco-do.html, 2008.

[67] Jennifer Stoddart. Letter to Mr. David C. Drummond, Senior Vice President, Corporate Development and Chief Legal Officer, Google, regarding 3D online mapping technology. http://www.privcom.gc.ca/media/let/let_070911_01_e.asp, August 2007.

[68] Paul Syverson, Michael Reed, and David Goldschlag. Onion Routing access configurations. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 1, pages 34–40. IEEE CS Press, 2000.

[69] Richard Taylor. The great firewall of China. http://news.bbc.co.uk/2/hi/programmes/click_online/4587622.stm, 2006.

[70] The Russia Journal. St Petersburg witnesses anti-Putin flash mob. http://www.russiajournal.com/node/17338, March 2004.

[71] TorStatus. Tor Network Status. http://torstatus.kgprog.com/.

[72] United Nations. Universal Declaration of Human Rights. http://www.unhchr.ch/udhr/lang/eng.htm.

[73] U.S. House of Representatives Committee on Foreign Affairs. Statement of Chairman Lantos at hearing, Yahoo! Inc.'s Provision of False Information to Congress. http://foreignaffairs.house.gov/press_display.asp?id=446.

[74] U.S. House of Representatives, Committee on International Relations. The Internet in China: A Tool for Freedom or Suppression? http://www.foreignaffairs.house.gov/archives/109/26075.pdf, February 2006.

[75] Camilo Viecco. UDP-OR: A Fair Onion Transport Design. http://www.petsymposium.org/2008/hotpets/udp-tor.pdf, 2008.

[76] R. Wendolsky, D. Herrmann, and H. Federrath. Performance Comparison of Low-Latency Anonymisation Services from a User Perspective. In *Privacy Enhancing Technologies*, pages 233–253, 2007.

[77] Alma Whitten. Google Public Policy Blog: Are IP addresses personal? http://googlepublicpolicy.blogspot.com/2008/02/are-ip-addresses-personal.html,
2008.

[78] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against Sybil Attacks via Social Networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 267–278, 2006.