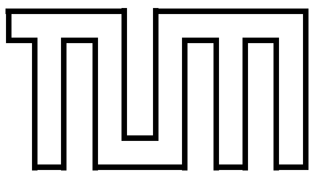


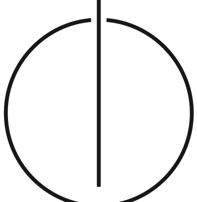
FAKULTÄT FÜR INFORMATIK
Technische Universität München

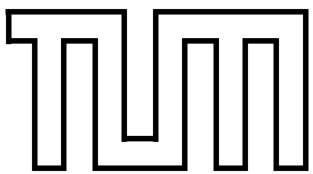
Lehrstuhl für Netzarchitekturen und Netzdienste

Master's Thesis in Informatics

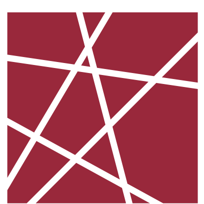
SPEEDING UP TOR WITH SPDY

Andrey Uzunov





FAKULTÄT FÜR INFORMATIK
Technische Universität München



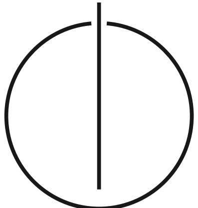
Lehrstuhl für Netzarchitekturen und Netzdienste

Master's Thesis in Informatics

SPEEDING UP TOR WITH SPDY

BESCHLEUNIGUNG VON TOR MIT SPDY

AUTHOR: Andrey Uzunov
SUPERVISOR: Christian Grothoff, PhD (UCLA)
ADVISOR: Christian Grothoff, PhD (UCLA)
DATE: November 15, 2013



DECLARATION

I assure the single handed composition of this master's thesis only supported by declared resources.

Garching, November 15, 2013

Andrey Uzunov

ACKNOWLEDGEMENTS

I would like to thank my supervisor Christian Grothoff for providing the initial idea and for the extensive support and guidance throughout the thesis. I thank Bartlomiej Polot, Matthias Wachs and a person who wants to stay anonymous for providing me with log files for the evaluations. I thank Karl Hughes for initial editorial work on first two chapters.

ABSTRACT

SPDY is a rather new protocol which is an alternative to Hypertext Transfer Protocol (HTTP). It was designed to address inefficiencies in the latter and thereby improve latency and reduce bandwidth consumption.

This thesis presents the design and implementation of a setup for utilizing SPDY within the anonymizing Tor network for reducing latency and traffic in the latter. A C library implementing the SPDY server protocol is introduced together with an HTTP to SPDY and a SPDY to HTTP proxy which are the base for the presented design.

Furthermore, we focus on the SPDY server push feature which allows servers to send multiple responses to a single request for reducing latency and traffic on loading web pages. We propose a prediction algorithm for employing push at SPDY servers and proxies. The algorithm makes predictions based on previous requests and responses and initially does not know anything about the data which it will push.

This thesis includes extensive measurement data highlighting the possible benefits of using SPDY instead of HTTP and Hypertext Transfer Protocol Secure (HTTPS) (1.0 or 1.1), especially with respect to networks experiencing latency or loss. Moreover, the real profit from using SPDY within the Tor network on loading some of the most popular web sites is presented. Finally, evaluations of the proposed push prediction algorithm are given for emphasizing the possible gain of employing it at SPDY reverse and forward proxies.

ZUSAMMENFASSUNG

SPDY ist ein neues Netzwerkprotokoll, das eine Alternative vom Hypertext Transfer Protocol (HTTP) ist. Es wurde entwickelt, um Ineffizienzen in HTTP abzubauen, und damit die Latenzzeit zu verbessern, und die Bandebreite zu reduzieren.

Diese Arbeit präsentiert den Entwurf und die Implementierung eines Setups, das die Benutzung vom SPDY im Tor-Netzwerk ermöglicht, um die Latenzzeit und den Datenverkehr beim Laden von Webseiten zu reduzieren. Eine Bibliothek in C, die SPDY implementiert, wird zusammen mit einem HTTP zum SPDY und einem SPDY zum HTTP Proxy vorgestellt. Die letzte sind die Basis vom präsentierten Entwurf.

Weiterhin konzentrieren wir uns auf das SPDY Server Push: Ein Protokollfeature, dass die Server ermöglicht, mehrere Antworten für eine Anfrage zu senden. Wir schlagen einen Algorithmus, der auf Prognosen aufbaut, für SPDY-Server und SPDY-Proxy vor. Er erfasst Statistiken zu Anfragen und Antworten und sagt vorher, was als Push gesendet werden soll, ohne am Anfang etwas zu den Daten zu wissen.

Diese Arbeit bietet umfangreiche Messdaten, die die mögliche Vorteile der Verwendung von SPDY statt HTTP und Hypertext Transfer Protocol Secure (HTTPS) (1.0 und 1.1), vor allem in Bezug auf Netzwerke mit Latenzzeit oder Verlust, illustrieren. Weiterhin werden die realen Vorteile der Verwendung von Tor mit SPDY, wenn eine der populärsten Webseiten geladen werden, präsentiert. Außerdem ist die Analyse vom vorgeschlagenen Push-Algorithmus mit Betonung mögliches Gewinns für Reverse- und Forward-Proxies gegeben.

CONTENTS

1 INTRODUCTION 1
2 BACKGROUND 3
2.1 Tor 3
2.2 Tor Browser Bundle 4
2.3 HTTP 5
2.4 SPDY 6
2.5 SPDY Server Push 8
2.6 HTTP/2.0 8
3 DESIGN 9
3.1 SPDY and Tor 9
3.1.1 Our Solution for Using SPDY over Tor 9
3.1.2 Open Design Question: Integration 10
3.2 SPDY Server Push Based on Predictions 10
3.2.1 Cost Function of SPDY Server Push 13
3.2.2 Prediction Algorithm 15
3.2.3 Description of the Proposed Algorithm 18
3.2.4 Algorithm Complexity 25
4 IMPLEMENTATION 27
4.1 SPDY Libraries 27
4.1.1 libmicrospdy Overview 27
4.2 Proxies Overview 28
4.2.1 mhd2spdy – an HTTP to SPDY Forward Proxy 28
4.2.2 microspdy2http – a SPDY to HTTP Proxy 28
4.2.3 Proxies and Tor 29
5 EVALUATIONS 31
5.1 SPDY vs. HTTP 31
5.1.1 Setup for the Experiments 31
5.1.2 Results 35
5.1.3 Discussion 44
5.2 SPDY over Tor vs. HTTP over Tor 46
5.2.1 Experimental Setup 46
5.2.2 Results 49
5.3 Evaluation of the Proposed Algorithm for SPDY Server Push 52
5.3.1 Using Web Server Access Log Files 54
5.3.2 Using ISP Log Data 55
5.3.3 Using Forward Proxies Log Data 56
5.3.4 Results 57
5.4 Evaluation of HTTP and SPDY Headers and Body Size 73
6 DISCUSSION 77
6.1 Design Alternatives for Using SPDY with Tor 77
6.1.1 Alternatives for Using Proxies 77
6.1.2 Design without Proxies 78
6.2 Benefits from Using SPDY within the Tor Network 78
6.3 Privacy Concerns when Using SPDY with Tor 79
6.3.1 SPDY Settings 80
6.3.2 SPDY Server Push 80

6.3.3 SPDY Session Creation 80
6.3.4 SPDY Session Lifetime 80
6.4 SPDY Server Push Based on Predictions 81
6.4.1 Keeping History of Previous Visits 84
6.4.2 Implementation Notes about the Proposed Algorithm 84
7 CONCLUSION AND FUTURE WORK 85

BIBLIOGRAPHY 87

APPENDIX 89
A EVALUATION OF PAGE LOAD TIME WITH SPDY OVER TOR 91
A.1 Evaluated Web Sites 91
A.2 Tor Client Configuration for Choosing Nodes 92
A.3 Benchmark Page 93
A.4 Script for Measuring Traffic Between Middle and Exit Node . 97
B EVALUATION OF THE PROPOSED ALGORITHM FOR SPDY SERVER PUSH 99
B.1 Script for Filtering Privoxy Log Files 99

LIST OF FIGURES

Figure 3.1 Setup of proxies used together with Tor. 9
Figure 3.2 Relation between Responses, Requests, Pushes, Mistakes and Assets when the number of mistakes is the minimal possible. 14
Figure 3.3 Example of the PushCost function 15
Figure 5.1 Setup for the SPDY vs. HTTP experiments. 33
Figure 5.2 Download time for the entire web page in relation to network latency, allowing 6 parallel Transmission Control Protocol (TCP) connections for all HTTP setups. Median value of 6 runs is reported. 35
Figure 5.3 Download time as in Figure 5.2, except Round Trip Time (RTT) deviation is set to 20% of RTT and packet loss rate is 5%. 35
Figure 5.4 Download time for the entire web page needed by SPDY setups in relation to the time needed by HTTP setups for RTT 20ms and different packet loss rates, allowing 6 parallel TCP connections for the latter. Median value of 6 runs is used. 36
Figure 5.5 Download times relation as in Figure 5.4, except with 40ms RTT. 36
Figure 5.6 Download times relation as in Figure 5.4, except with 60ms RTT. 37
Figure 5.7 Download times relation as in Figure 5.4, except with 80ms RTT. 37
Figure 5.8 Download times relation as in Figure 5.4, except with 100ms RTT. 37
Figure 5.9 Download times relation as in Figure 5.4, except with 120ms RTT. 38
Figure 5.10 Download times relation as in Figure 5.4, except with 160ms RTT. 38
Figure 5.11 Download times relation as in Figure 5.4, except with 200ms RTT. 38
Figure 5.12 Download times relation as in Figure 5.4, except with 240ms RTT. 39
Figure 5.13 Download times relation as in Figure 5.4, except with 400ms RTT. 39
Figure 5.14 Network traffic (in bytes) for downloading the entire web page in relation to network latency, allowing 6 parallel TCP connections for all HTTP setups. Median value of 6 runs is reported. 39
Figure 5.15 Network traffic (in packets) for downloading the entire web page in relation to network latency, allowing 6 parallel TCP connections for all HTTP setups. Median value of 6 runs is reported. 40
Figure 5.16 Network traffic (in bytes) as in Figure 5.14, except RTT deviation is set to 20% of RTT and packet loss rate is 5%. 40

Figure 5.17 Network traffic (in packets) as in Figure 5.15, except RTT deviation is set to 20% of RTT and packet loss rate is 5%. 41
Figure 5.18 Network traffic (in bytes) for downloading the entire web page for SPDY setups in relation to the traffic for HTTP setups for RTT 20ms and different packet loss rates, allowing 6 parallel TCP connections for HTTP. Median value of 6 runs is reported. 41
Figure 5.19 Network traffic (in bytes) as in Figure 5.18, except with 400ms RTT. 42
Figure 5.20 Download time for the entire web page in relation to the number of parallel TCP connections for HTTP setups averaged over 2 runs for RTT 20ms. The values for SPDY setups are median values of 6 runs. 42
Figure 5.21 Download time as in Figure 5.20, except with 40ms RTT. 42
Figure 5.22 Download time as in Figure 5.20, except with 60ms RTT 43
Figure 5.23 Download time as in Figure 5.20, except with 80ms RTT 43
Figure 5.24 Download time as in Figure 5.20, except with 100ms RTT 43
Figure 5.25 Download time as in Figure 5.20, except with 120ms RTT 44
Figure 5.26 Relative page load times for SPDY over Tor compared to HTTP over Tor. The evaluated web sites are ordered by the absolute average time for HTTP. 50
Figure 5.27 All evaluated web sites ordered from the one with greatest benefit from SPDY to the one with least. . . . 51
Figure 5.28 Relation between traffic benefit and time benefit for the first 12 web sites from the list of evaluated. 52
Figure 5.29 The best single result for Site1. 59
Figure 5.30 The best single result for Site2. 60
Figure 5.31 The best single result for Site3. 60
Figure 5.32 The best single result for Site4. 60
Figure 5.33 The best single result for Site5. 61
Figure 5.34 The best single result for Isp. 61
Figure 5.35 The best single result for Proxy1. 61
Figure 5.36 The best single result for Proxy2. 62
Figure 5.37 The best single result for Proxy3. 62
Figure 5.38 Average results of the best combination of input parameters for Site1 applied for all data sets. 64
Figure 5.39 Average results of the best combination of input parameters for Site2 applied for all data sets. 65
Figure 5.40 Average results of the best combination of input parameters for Site3 applied for all data sets. 65
Figure 5.41 Average results of the best combination of input parameters for Site4 applied for all data sets. 66
Figure 5.42 Average results of the best combination of input parameters for Site5 applied for all data sets. 66
Figure 5.43 Average results of the best combination of input parameters for Isp applied for all data sets. 67

Figure 5.44 Average results of the best combination of input parameters for Proxy1 applied for all data sets. 67
Figure 5.45 Average results of the best combination of input parameters for Proxy2 applied for all data sets. 68
Figure 5.46 Average results of the best combination of input parameters for Proxy3 applied for all data sets. 68
Figure 5.47 Increase of the average value of the cost function for Site1 when a single parameter is changed. 69
Figure 5.48 Increase of the average value of the cost function for Site2 when a single parameter is changed. 69
Figure 5.49 Increase of the average value of the cost function for Site3 when a single parameter is changed. 70
Figure 5.50 Increase of the average value of the cost function for Site4 when a single parameter is changed. 70
Figure 5.51 Increase of the average value of the cost function for Site5 when a single parameter is changed. 71
Figure 5.52 Increase of the average value of the cost function for Isp when a single parameter is changed. A change in P2 is not applicable. 71
Figure 5.53 Increase of the average value of the cost function for Proxy1 when a single parameter is changed. A change in P3 is not applicable. 72
Figure 5.54 Increase of the average value of the cost function for Proxy2 when a single parameter is changed. A change in P3 is not applicable. 72
Figure 5.55 Increase of the average value of the cost function for Proxy3 when a single parameter is changed. A change in P3 is not applicable. 73

LIST OF TABLES

Table 3.1 Differences when SPDY server push is used on reverse proxies and when with Tor. 12
Table 3.2 Calculating the probability for pushing a single asset. 16
Table 3.3 Input options for push algorithm. 17
Table 5.1 Different client/server setups used in the experiments. 32
Table 5.2 Software, hardware and other details for benchmarking SPDY and HTTP over Tor. 48
Table 5.3 Page load time per web site for HTTP and SPDY over Tor. 51
Table 5.4 Average traffic per web site. It was measured at the exit node while benchmarking the page load time. *Input* is the traffic from the middle relay to the exit node; *output* is the one from the exit node to the middle relay. 51
Table 5.5 Evaluated input parameters. 53
Table 5.6 Constant input parameters. 54
Table 5.7 Web sites whose access log files were evaluated for SPDY server push. 55
Table 5.8 Proxy users whose log files were evaluated for SPDY server push. 57
Table 5.9 The combination of input parameters which gave the best single result per data set. 58
Table 5.10 The best achieved result per data set. Pushes, hits and mistakes are normalized to the number of assets. . . 58
Table 5.11 Characteristics of the single day sample which gave the best result per data set. 58
Table 5.12 Characteristics and algorithm results of the single day sample which gave the best result per data set. . 59
Table 5.13 The combination of input parameters which gave the best average result per data set. 63
Table 5.14 Average results for the parameters in Table 5.13. Pushes, hits and mistakes are normalized to the number of assets. 63
Table 5.15 Characteristics of each data set when the parameters in Table 5.13 are used. 63
Table 5.16 Characteristics and algorithm results of each data set when the parameters in Table 5.13 are used. 64
Table 5.17 Average, minimum, median and maximum values for request, response headers and response body per web resource. All resources are considered expect those whose responses were not yet received at the time when the `onload` event was triggered. 73
Table 5.18 Average, minimum, median and maximum values for request, response headers and response body per web resource. Only *GET* requests received response with status *200* are considered. 74
Table 5.19 Estimations of traffic benefit and cost from using SPDY server push. 74

Table 6.1 Summary of the effect of the evaluated input parameters on the results. 83

LIST OF ALGORITHMS

1 Push prediction algorithm: input parameters and global maps 19
2 Helper procedures for handling maps 20
3 Main procedures of the algorithm; they are called by the web server/proxy 20
4 Other procedures and functions used by the algorithm (1) . . 21
5 Other procedures and functions used by the algorithm (2) . . 22
6 Other procedures and functions used by the algorithm (3) . . 23

AES	Advanced Encryption Standard. 3
API	Application Programming Interface. 8, 27, 28
CRL	Certificate Revocation List. 11, 56
CSS	Cascading Style Sheets. 8, 31
DNS	Domain Name System. 9
HTML	HyperText Markup Language. 8, 10, 11, 31, 32, 73
HTTP	Hypertext Transfer Protocol. ix, xi, xv, xvi, 1, 5–10, 18, 27–29, 31–36, 39–42, 44, 45, 47–49, 53, 56, 77–80, 85
HTTPS	Hypertext Transfer Protocol Secure. ix, xi, 9, 32, 44–46, 85
IETF	Internet Engineering Task Force. 6, 27
IO	Input/Output. 27, 28
IP	Internet Protocol. 4–6, 12, 44, 47, 53, 55, 79, 84
IPv4	Internet Protocol version 4. 34, 47
ISP	Internet Service Provider. 53, 55, 56
MTU	Maximum Transmission Unit. 45
NAT	Network Address Translation. 53
NPN	Next Protocol Negotiation. 27
OCSP	Online Certificate Status Protocol. 11, 56
RTT	Round Trip Time. xv, xvi, 1, 7, 11, 33–45, 74, 78
SOCKS	Socket Secure. 3–5, 9, 28, 29, 47
TBB	Tor Browser Bundle. 4, 5, 9
TCP	Transmission Control Protocol. xv, xvi, 3–6, 9, 29, 32, 34–36, 39–42, 44–46, 79, 84
TLS	Transport Layer Security. 3, 7, 9, 27–29, 32–34, 45, 56, 77, 78
URL	Uniform Resource Locator. 9, 10, 12, 18, 23, 24, 29, 48, 54, 55, 84

VM Virtual Machine. 33

1 INTRODUCTION

Tor is an overlay network for improving privacy and security on the Internet and is primarily used for enabling online anonymity and circumventing censorship. This is achieved by encrypting data flows and redirecting them through three Tor relays around the world. However, this design feature causes poor page load performance when using Tor as opposed to ordinary browsing. As an extreme example, it may happen that the client connects to a physically proximate server, but the communication must travel to another continent twice. The high latency is most likely one significant reason for Internet users to not use Tor.

The research reported in this paper aims to achieve better page load performance for end users when using Tor without changing Tor's security properties by using SPDY [Spd#2] instead of Hypertext Transfer Protocol (HTTP) for web traffic in the Tor network even when the contacted server does not support SPDY. The use of SPDY instead of HTTP to improve performance in Tor was previously proposed by Steven Murdoch [Mur10]. However, in contrast to this proposal, our design uses proxies which are separated by the Tor processes and the Tor Browser. This allows us to achieve the transition to SPDY without changes in the existing software, and with minimal or no changes in the Tor network itself.

SPDY is a relatively new web protocol and an alternative to the widely used HTTP. The former aims to resolve most of the drawbacks of the latter and has also shown good results for page load performance [Spd#2]. The difference in the timing results between SPDY and HTTP is especially large when the Round Trip Time (RTT) is large (we present experimental results showing this in Section 5.1). Since there is non-negligible latency within the Tor network, the performance is expected to be improved when using SPDY over Tor. Furthermore, the traffic within the Tor network is expected to be reduced because SPDY uses less traffic than HTTP, and this will be appreciated by the operators of Tor relays, who typically run them on a voluntary basis.

A particular focus of this thesis is the impact of the SPDY server push mechanism and its possible use in combination with Tor. SPDY push gives servers the technical ability to send multiple responses to a single request, decreasing used traffic and latency. However, what resources should be pushed under which circumstances has not been specified in the SPDY specification (and is not likely to be specified in the future as well). Moreover, no existing research on the topic could be found.

Today's websites may consist of thousands of single web pages. Thus, it is hardly convenient for a web developer to configure which additional resources should be pushed for each single page. Furthermore, many of the candidates for pushing are likely to be cached by clients, thus a manual specification process is likely to require significant tuning, especially as pushing an already cached resource would lead to additional traffic. We propose a generic learning algorithm which based on previous requests and responses tries to predict which additional resources would be suitable to push to clients. The goal would be to automatically achieve better page load times regardless of the data served by the system. Furthermore, we investi-

gate a variety of input parameters and how they affect the behavior of the algorithm when it is employed by reverse and by forward proxies. We then discuss the impact of employing SPDY server push in the context of Tor, which presents additional challenges.

The source code of the prediction algorithm developed for this thesis and the non-sensitive data which was collected are freely available at http://dev.online6.eu/spdypush.

The thesis is structured as follows. Chapter 2 gives an overview of the protocols, systems and mechanisms which are employed by our design. The proposed design for using SPDY with Tor, as well as the design of the prediction algorithm, are described in Chapter 3. Chapter 4 focuses on the implementation. The SPDY protocol, the utilization of SPDY within the Tor network, and the prediction algorithm for server push are evaluated in Chapter 5. Next, Chapter 6 includes discussion about alternative designs, achieved results in the evaluations, and new challenges which SPDY brings to Tor. Finally, Chapter 7 gives an overview of the achieved goals and ideas for future work.

2 BACKGROUND

2.1 TOR

Tor is a circuit-based low-latency anonymizing network based on nodes which are run on a voluntary basis. The source code is open-source and not covered by patents [DMS04]. Any computer with a connection to the Internet can participate in the Tor network as a client and/or onion router (also called Tor node or relay).

The general principle of Onion Routing, which Tor inherits, is to provide security and privacy for streams by using layers of encryption. An original data flow within a stream is encrypted by the initiator of the communication (the client) as many times as there are relays until the edge of the Tor network. One layer of encryption is removed at each Tor node. That is, each node decrypts whatever it receives to understand where the decrypted piece of data should be later delivered. Finally, the last node has the original data, which it uses to build a Transmission Control Protocol (TCP) connection to the target. When the contacted server replays, the data travels through exactly the same path back to the client. Each Tor node adds a layer of encryption, which can be decrypted only by the client. The latter receives the data from the server with several layers of encryption, equal to the number of nodes on the way. After decryption, the response is given to the application which uses Tor. A client application may not even know that is uses the Tor network. On the other hand, the server sees only the IP of the the Tor exit node, which is the last node of the circuit.

Tor runs as a user process on a variety of operating systems and architectures. The Tor client (called onion proxy) provides a Socket Secure (SOCKS) proxy interface and can be used by any TCP-based program which speaks SOCKS. In addition to the encrypted TCP stream, Tor and the SOCKS proxy provide a name resolution service. As a result, any TCP-based program (e.g. a web browser) may use entirely and only anonymized connections to communicate with Internet services.

It is even possible for applications which are not designed with SOCKS support to use Tor. For this purpose a program called Torsocks was created [Tor]. It basically tries to replace the system calls used by an application with those that build and use SOCKS streams. However, this does not yet work for all applications.

For communication within the Tor network, nodes use *cells* whose payload is encrypted several times using Advanced Encryption Standard (AES). The key for each layer of encryption is only known to the initiator (the client) and one specific node on the path.

The Tor client is responsible for creating circuits within the Tor network. When the creation of a new TCP stream is requested by an upstream application, the client creates a stream within a circuit. Many streams may share a single circuit. The stream built in this way uses *relay* cells to carry data on behalf of a real TCP stream.

The communication between Tor relays is secured using Transport Layer Security (TLS). There is usually just a single connection between two nodes and all circuits share it. Each circuit consists of at least one Tor relay. How-

ever, the current Tor implementation by default uses three nodes for a circuit. That means: an entry, a middle and an exit node. Nevertheless, it is possible that more routers are added to an existing circuit and hence, the latter has greater length and contains several middle nodes. The exact relays are chosen by the client based on information which it has about the network, and the configuration given by the user. Clients obtain the current state of the network – running relays and their Internet addresses, exit nodes and their policies, etc. – from another kind of Tor node, the so-called *directory servers*.

The Tor client is the only one which knows all the relays in a circuit. All others know only the previous and the subsequent node within a circuit. The choice of an exit node is based on its exit policy, which is set by its operator. The policy states which Internet Protocol (IP) addresses and port numbers the exit node is allowed to connect to; that is, which client TCP streams are allowed to exit from a specific relay. Giving this choice to the operator is important because the final destination only sees the exit node as a client and therefore, in many countries, she may be held responsible for the traffic generated by her machine.

A Tor exit node is the edge of the Tor network. This means it has access to exactly the same information as the onion proxy. That is, whatever data enters the proxy, the same is sent by the exit node to the final destination. While it is possible that another security channel is used from the exit node to the final destination, this is outside of the scope of Tor's security model. Tor users are expected to be aware that the exit node can spy on their traffic and possibly even manipulate it. While Tor tries to audit exit nodes for active manipulations, spying cannot be detected and as anyone can run an exit node, it must be expected. However, an exit node has no a-priori information about the initiator of connections (the user).

It should be stated that all the mentioned Tor node types, including the client, only describe the specific role in the Tor network. Any Tor process can have several of the roles at the same time for different circuits. From a privacy point of view, it is even advisable for users to run a Tor relay together with a client. In this way, it may be harder for adversaries to detect that the relay is used also as a client.

2.2 TOR BROWSER BUNDLE

Tor Browser Bundle (TBB) is a software package containing Tor Browser and Tor itself, which gives desktop users the opportunity to browse anonymously with a very little effort: a user needs only to download an archive file, extract it, and run a program.

The Tor Browser is based on Mozilla's Extended Support Release (ESR) Firefox branch [PCM13], but it also includes many patches which improve privacy and security. Its predominant aims are:

- Using the Tor SOCKS proxy for any kind of network connections the browser needs. A side effect is that all plug-ins should be disabled, since it is currently impossible to force them to use the proxy settings of the browser.
- Unlinkability of user's activity. An adversary can be an eavesdropper, an exit node, or a visited web site itself. Ideally, such an adversary must not be able to link a user's visit to a site A to a same user's visit to a site B. However, as of July, 2013, this is still possible in the stable version of the browser, as exit nodes can link traffic between

different tabs in the same browser instance as they are routed via the same circuit.

- Not leaving traces on the user's machine. To improve privacy in the case of an adversary taking control of the physical machine of a user at a later time, the browser must not write any state to the hard drive. Owing to this requirement, TBB restricts its use of the disk. If the user does not download any files, she can simply stop TBB, remove it from the disk, and no traces of Tor's usage should be left.

Although configuring another browser which speaks SOCKS to use Tor is possible, this is highly discouraged, because various security and privacy improvements present in the Tor Browser would no longer be available.

2.3 HTTP

The Hypertext Transfer Protocol (HTTP) is the predominant protocol employed in the World Wide Web. Almost all the elements provided by a web page, such as text documents, images, script files, videos, and so forth, are transferred via HTTP. It runs normally over TCP and is a text-based, human-readable protocol, which means that a significant amount of redundant data is transmitted over on the Internet.

The protocol utilizes two types of messages: a request message – sent by a web client to request a resource or an operation, and a response message – sent by a web server to return a status message to the client and possibly the requested resource itself. Both requests and responses consist of a number of header lines and a body part. The former contain meta information for exact identification of the requested resource, as well as parameters affecting a future communication between client and server. The optional body part in a request message contains parameters and/or files sent to the server. Similarly, the optional body part within a response message contains the requested resource or information.

The first documented version of the protocol – HTTP/0.9 – was introduced in 1991. A later version – HTTP/1.0 – is still employed on the Internet. Specific for both mentioned standards is the utilization of exactly one TCP connection for a request/response pair. That is, having sent a response, a server closes the connection. As a result, each request/response pair employs additional TCP/IP packets and has larger latency because of the TCP handshake. Furthermore, each pair possibly suffers from TCP's slow-start-mechanism.

This issue was partially addressed in HTTP/1.1, which introduced persistent connections. With the latter, a single TCP connection may be utilized for multiple request/response pairs if both the client and the server support it. Since body parts in exchanged HTTP messages do not have explicit boundaries, the preceding headers part must contain a "Content-Length" header which specifies the size of the information sent into the body. In normal usage, a client cannot reuse a persistent connection until the response to a previous request is entirely received. After that, a new request may be sent on the same connection.

Whereas HTTP/1.1 connections are persistent by default, the same behavior can be achieved in HTTP/1.0 by employing the header line "Connection: Keep-Alive" in requests and responses. However, this feature is not officially part of the standard.

A possible issue with persistent connections is the case when several clients exhaust a server by opening multiple connections without closing them. To prevent such situations, web browsers are expected to limit the number of persistent connections. In the past the limit was usually two, while currently most of the major browsers apply a value of six [Smi12], and give their users the ability to change this.

For the case where a sender does not know the size of the data at the time of sending headers, HTTP/1.1 introduced chunked transfer encoding: instead of using the "Content-Length" header, chunks of data with explicit boundaries and size are sent within the body part of HTTP messages.

Persistent connections and the use of "Content-Length" header or chunked encoding to indicate the end of a response without closing the TCP stream still do not break the traditional request-response pattern of HTTP, which requires a full round trip for each request.

This is the reason for the introduction of another HTTP/1.1 feature – HTTP pipelining. It gives clients the ability to send as many requests and as they want at whichever moment. Clients are required to send only idempotent requests via pipelining [Fie+99, Section 9.1.2 Idempotent Methods] and must be able to handle closed pipelines – a server may close a connection before answering all requests; thus, resending of the non-answered requests may be required. The only requirement for server software is to send responses in exactly the same order as they are requested. HTTP pipelining has been expected to highly improve the HTTP performance, because some round trips are eliminated. That is, a server may already have the next request at the time the former finishes sending a response. Furthermore, the overall number of IP datagrams, and respectively traffic, in the network are supposed to be reduced, since multiple requests or responses may be combined within a single datagram.

However, HTTP pipelining has never been widely deployed. The only major browser employing it is Opera, which has had pipelining enabled by default for a long time [Not11]. Similarly, Firefox supports it as well, but it is disabled by default. However, HTTP pipelining is enabled in the Tor Browser and is claimed to aid page load performance [Per12]. The feature is not reported to be ever implemented in Internet Explorer. On the other hand, Google Chrome had experimental support until recently – since version 26, the possibility to enable pipelining was removed [mme13].

The primary concern of the browser vendors is the existence of many servers and middle boxes (e.g. proxies) which do not support pipelining. This leads to problems which directly affect user's experience. Another possible problem is the head-of-line blocking – receiving of a big resource will block all other pending requests on the same connection.

The Internet Engineering Task Force (IETF)'s working group Httpbis still works on improvements of the specification of HTTP/1.1.

2.4 SPDY

SPDY is an application-layer protocol designed by Google to provide better page load performance compared to the widely used HTTP. This is largely achieved by reducing bandwidth consumption and using only a single TCP connection per server. The latter allows TCP to operate more often at peak throughput and furthermore, enables SPDY to avoid retransmitting state for each HTTP request. Multiple requests can be transmitted without waiting for responses and the respective replies can be interleaved. Requests can

specify a priority; thus, important elements for showing a web page can be sent before others. Most of these features are attributed to the framing layer utilized in SPDY – all data sent on the connection is enclosed in frames and therefore, the boundaries of requests and responses are clear as opposed to HTTP.

SPDY is used always over TLS and normally clients should use only one long-lived SPDY connection (also called session) to a single host. Each connection is used by multiple streams. A single stream represents sending a request and its response.

Aiming at easy adaptation, SPDY uses the same HTTP headers in requests and responses. However, it uses DEFLATE to compress them. While this largely reduces the traffic generated by requests and small sized responses, the CRIME attack [RD12] showed the danger of encrypting compressed data – an attacker is able to retrieve a string from encrypted messages, when it is contained in multiple similar messages, which for instance is exactly the case with cookies in requests. For this reason, other methods of compressing headers have been researched, for example applying the same algorithm but leaving the cookie header line uncompressed.

While SPDY operates in a similar way to HTTP pipelining, a small overhead in terms of traffic is added because of frame headers. However, the introduction of frames eliminates many of the problems of pipelining: A server can send responses in any order; there is explicit correlation between a request and response frames; body parts of different responses can be interleaved; and so forth. Unfortunately, the specification of the new protocol is not exhaustive, specific issues are not addressed, and some problems are left to implementers. For instance, head-of-line blocking is still possible in SPDY and current server implementations suffer from it.

Some new features were introduced by the protocol. Firstly, settings frames affecting current and possible future sessions between the same peers are exchanged and possibly stored by clients. Secondly, application layer flow control is applied to decrease the speed or interrupt sending of data if the other party cannot handle it right now. Lastly, SPDY ping is employed to measure the RTT to the other peer and also keep the connection alive.

Since 2012, two versions of SPDY have been in use – SPDY/2 [BP#1] and SPDY/3 [BP#2]. SPDY is supported by most of the major desktop browsers – Google Chrome, Firefox, Opera and Internet Explorer (from version 11) – and by the major open-source browsers for mobile devices [Spd#1]. Furthermore, some of the largest web sites – among them Google and Twitter – support it. Among the popular open-source server software, mod_spdy for Apache and a patch for nginx have complete SPDY support.

The protocol version to be applied for a session is negotiated during the TLS handshake via a TLS extension called Next Protocol Negotiation (NPN) [Lan12#1].

SPDY/3 was designated by Httpbis as the starting point for the future standard HTTP/2.0. In parallel with the work on the latter, the work in Google on SPDY itself continues. Since 2013, SPDY/3.1 and several alpha versions of SPDY/4 have been tested by Google and other companies. The specification of SPDY/3.1 [BP13] was published in September, 2013. Shortly after that, it was announced that both Firefox and Chrome would soon retire SPDY/2 [MA13].

2.5 SPDY SERVER PUSH

SPDY allows servers to send multiple responses to a single request. To push a resource, a server has to send a frame for opening a new stream, which contains identification of the pushed resource, possibly more headers within another frame, and the response body of the pushed resource. Servers are allowed to open new streams only while the primary one – that for the original request – is open. This means a server must open a new stream within the time frame from the receiving of the original request to the sending of the last frame for the response body. To prevent race conditions, for instance when a client requests a resource and a server meanwhile pushes the same one, the opening frame and the headers must be sent before the response headers for the primary request. It is not specified when the resource body should be pushed.

Clients are required to cache pushed resources in the same way as they do for any other received resource. At the time a pushed resource is needed, it is obtained from the cache. However, pushing an already cached resource is still possible, for example because of a previous visit to the same page. That is why, at any time after receiving an opening frame for a pushing stream, the client is allowed to close such a stream with an additional frame. Nevertheless, unneeded pushes cause redundant traffic in the network.

The Application Programming Interface (API) of mod_spdy currently allows server push by specifying pushed content via an "X-Associated-Content" header. This should be decided by a website maintainer for all resources which would benefit from pushing additional content. Obviously, server push should be applied only on requesting a resource which links to other elements. Such resources are HyperText Markup Language (HTML) pages, Cascading Style Sheets (CSS) files, JavaScript files and so forth, but a link to another element is possible as well in the response headers (e.g. the "Link" header).

2.6 HTTP/2.0

The Httpbis working group works on the next version of HTTP. It is based on SPDY but already has some major differences. Some of key differences are:

- The group is discussing making encryption in HTTP/2.0 optional, instead of mandatory;
- Reprioritization of requests is needed to avoid head-of-line blocking;
- A new method for compressing headers is being introduced, which is believed to be less vulnerable to attacks.

3 DESIGN

3.1 SPDY AND TOR

There exist several different possibilities to make Tor employ SPDY. The protocol is already supported by Firefox and respectively by the Tor Browser. The latter comes with the extension HTTPS Everywhere which replaces the requested Uniform Resource Locator (URL)s by their secure equivalent for multiple web sites. Thus, it would be possible to utilize SPDY over Tor when web servers support it. However, SPDY is disabled by default in the Tor Browser because of privacy concerns [PCM13, 4.5. Cross-Origin Identifier Unlinkability, 6. SSL+TLS session resumption, HTTP Keep-Alive and SPDY] (see also Section 6.3). Moreover, the fact that the vast majority of web servers will only support HTTP in the near future makes us consider solutions where SPDY is used within the Tor network, but the destination server is still contacted via HTTP.

3.1.1 *Our Solution for Using SPDY over Tor*

In our setup, we use two proxies (see Figure 3.1). The first one receives all incoming HTTP requests from the Tor Browser, and sends equivalent SPDY requests to the second proxy. The former runs on the same machine where TBB runs, and operates between the browser and the Tor SOCKS proxy. By default, the Tor Browser is preconfigured to use the SOCKS proxy for all resource as well as all Domain Name System (DNS) requests. Firefox gives us the ability for setting additionally an HTTP proxy. That is, we use the new HTTP to SPDY proxy for all HTTP requests, but the SOCKS one is still used for Hypertext Transfer Protocol Secure (HTTPS) and DNS requests.

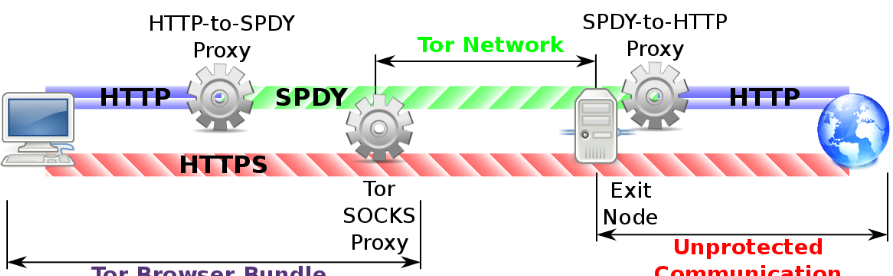


Figure 3.1: Setup of proxies used together with Tor.

The former establishes connections to SPDY to HTTP proxies which operate on exit nodes. From performance point of view, a single SPDY session per proxy should be used since there is no benefit when employing more – the Tor nodes communicate to each other via single TCP connection. The SPDY sessions employ the Tor network and therefore they have to be established through the SOCKS proxy. Moreover, the utilization of the Tor network removes the need of TLS connection for SPDY. Thus, the TLS layer may be omitted to possibly reduce computation time as well as data traffic.

The SPDY to HTTP proxy should be run on the exit node. In fact, it can be run anywhere, but this might introduce unnecessary round trips to an additional physical location. Furthermore, the new architecture would be deployed faster when the proxies are bundled with the Tor executables.

However, it is worth mentioning that both type of proxies operate outside of the Tor network. While the first one is under the control of the Tor user, the second is part of the untrusted way to the destination. That is, its operator has all the information in unencrypted form. Nevertheless, the situation is exactly the same when pure HTTP is used over Tor: The operator of an exit node has access to all the communication passing through it.

3.1.2 *Open Design Question: Integration*

Translating HTTP traffic to SPDY is meaningful only when there is a second proxy running on the exit node. Thus, in such a design, Tor clients have to be able to find which exit nodes run SPDY to HTTP proxies and hence, suitable nodes for building circuits should be chosen. Currently this remains an open design decision.

One possible solution is the introduction of a new relay flag for indicating that exit nodes are able to do the translation. However, this would require changes to the Tor code itself and thus take some time to deploy.

An alternative is to use the exit policy; one might designate a particular rule in an exit policy, for example an explicit rule that blocks HTTP traffic to a reserved, non-routable IP address to flag exits that support SPDY. This has the advantage that it can be deployed without any changes to Tor.

Finally, Tor may choose to require SPDY support by default for all exit nodes starting at a particular protocol version.

Our overall design does not depend on a particular method and we thus leave this choice to the Tor developer community.

3.2 SPDY SERVER PUSH BASED ON PREDICTIONS

We investigate automatized usage of the SPDY server push feature for SPDY to HTTP proxies. This can be employed in the already explained design for Tor, or when the proxy is used as a reverse one. The same approach can be applied for SPDY servers, but we focus on the cases when the executable does not have direct access to the server's local data, but is rather an intermediary. Further in the text, the words *proxy* and *server* are used interchangeably in the context of SPDY.

Since the proxy is located on the way between client and server, it sees only a sequence of requests, all their request and response headers, destination server, and possibly source identifier. Hence, we cannot be sure which resources are the first requested by the browser (e.g. an HTML page), which are additional elements needed for displaying a page and so forth. Therefore, we have to make some assumptions about the sequence of requests. For future explanations, several terms are used here with special meaning:

- *Page load* – this represents the request of an HTML page and all consequent requests. That is, all requests made by the browser when a user writes an URL into the address bar and loads a web page. We have to decide when a page load ends, because a web page may issue AJAX request infinitely until the browser's tab is still open. Thus, for last el-

ement into a page load, we choose the one after whose response there are no more requests within a period of time called *page load timeout*.

- *Page* – this represents the first resource requested within a page load.
- *Asset* – any other resource within a page load after the page.
- *Session* – it is equivalent to a SPDY session, has its length and includes all resources requested during the time of the session.

One session includes at least one page load. One page load consists of at least one request – the page. We assume that the page – the first requested element in a page load – is always an HTML page, and all subsequent assets are related to it. Therefore, all resources chosen for pushing should be sent before responding to the page request. Actually, the information for the pushed resource has to be sent before the response headers for the page are sent to avoid an explicit request for the same asset. That is, the *SYN_STREAM* frame for the push has to be sent before the *SYN_REPLY* for the response, and this is the only requirement. The body of the pushed asset may be sent immediately after its headers, after the page headers, or after the page body. This is implementation specific.

Obviously, some of the assumptions made in this model do not precisely correspond to real-world behaviour. Overlapping of page loadings may occur (e.g. loading web pages in several browser tabs at the same time). This means that some pages and all their assets will be detected as assets of another page. Furthermore, AJAX requests may be detected as pages, but also as assets for page loads in different tabs. The same is valid for requests issued by browser plug-ins as well as browser specific requests, for instance, downloading Online Certificate Status Protocol (OCSP) or Certificate Revocation List (CRL) information.

According to the SPDY protocol, clients have the ability to cancel pushing of resources. That is, a client may send one or more SPDY *RST_STREAM* frames to stop the sending of any resource by closing the respective SPDY stream. Depending on the RTT and the size of the pushed asset, the server may or may not receive such a frame early enough to cancel sending the resource. However, for simplicity and considering the possible privacy issues when SPDY is used with Tor (see Section 6.3), we do not consider this client ability in our evaluation. Specifically, we assume that clients do not create *RST_STREAM* frames and thus ignore the traffic that might be generated by these *RST_STREAM* frames. Furthermore, we always assume that the full body of a pushed response is received by the client, even if the choice to push it was a mistake and in practice some clients might abort the transmission. Nevertheless, we do not count such pushes as a hit in our evaluation.

Furthermore, there are some differences when a heuristic to push resources from the proxy to the client is used with a reverse proxy without Tor vs. with a forward proxy with Tor. We summarize the key differences in Table 3.1. Moreover, when the idea for employing different Tor circuits for different tabs in the Tor Browser (see [PCM13, Section 4.5. Cross-Origin Identifier Unlinkability, 11. Exit node usage] and Section 6.3) is respected, this would require also different SPDY sessions for each tab. As a result, even fewer page loads within a proxy session might be expected. On the other hand, an advantage might be that those page loads would be related to each other, as all pages would be from a single web site, this might make it easier for the learning algorithm to predict which assets should be pushed.

Push for Reverse Proxies	Push for Tor
The user's IP is obvious; thus, based on it and the "User-Agent" header, different sessions of the same user might be linked with high probability. As a result, more historical data per user may be considered.	Each user is anonymous and hence, we should design under the assumption that one session of a user cannot be linked to another one of the same user. (Nevertheless, this still might be possible, for instance due to cookies.)
The length of a session depends on the user agent and the proxy. It is expected to be rather long as there are performance benefits for the end user. As of October, 2013, Firefox applies three minutes as a SPDY session timeout while Chrome does not seem to close sessions for hours. Thus, a busy SPDY session can last for very long time.	A SPDY session over Tor should be shorter than a normal one: the default lifetime of a Tor circuit, which is ten minutes, should be considered. That is, we have fewer page loads per session, and much shorter historical data per user. Otherwise, exit nodes would be able to link visits to different sites to the same user and possibly obtain enough data to deanonymize users.
The set of accessed URLs is known in many cases, and depending on the specific web site, it can be considerably small.	It is expected that very diverse URLs are seen by the proxy. That is, the set of accessed URLs is potentially large.
It is possible to initialize the proxy with old data (e.g. access logs) and the pushing may start immediately to skip the learning phase.	Exit node operators are not supposed to log, as this would violate Tor user's privacy and make them a target for adversaries.
All pushed resources belong to the same origin. Otherwise, browsers must refuse them. Server push would not be beneficial when most or all of the assets are served are with origin different from the page.	The proxy works as a forward one. That is, requests to multiple servers are sent through it. Depending on the way the first proxy (see Figure 3.1) creates sessions, there may or may not be requests to multiple servers within a single session and/or page load. In the same way, pushed resources may or may not be from the same origin.
When the proxy is on the same machine or in the same local network as the web server, a push mistake costs only the data sent via the SPDY session: there is only a SPDY response.	A push mistake is twice as more expensive since a pushed asset should first be gathered by its original location. That is, more traffic is generated within the Tor network, and to the destination (Tor exit traffic).

Table 3.1: Differences when SPDY server push is used on reverse proxies and when with Tor.

3.2.1 *Cost Function of SPDY Server Push*

When using SPDY server push, one is interested in the proportion of correctly pushed resources – push hits – relative to the number of requests that otherwise the client has to issue. Moreover, it is important how many mistakes are generated. The mistakes may not matter for some services, because it is more important that the browser receives all assets for a page as soon as possible. In contrast, a lower push rate may be preferred in other situation, when mistakes are not tolerated. For instance, push mistakes made by a Tor exit node would introduce more traffic within the Tor network and furthermore, the operators would need more bandwidth for their relays. Thus, server push is the trade-off between latency improvement and additional traffic.

It is important how many requests are explicitly sent by the client (Equation 3.1). They are equal to the number of assets which have to be requested without push minus the correctly pushed resources. Ideally, this value would be zero when a perfect algorithm/oracle is used. When no push is used or only mistakes are generated, the requests are equal to the assets.

$$Requests = Assets + Mistakes - Pushes \tag{3.1}$$

where $Requests \in [0, Assets]$.

This value presents the difference between the total number of assets and the push hits, but it does not indicate the absolute number of mistakes. Mistakes require unnecessary responses sent by the SPDY server and therefore, their number is part of the total number of responses (Equation 3.2). That is, when no mistakes are generated, the responses are equal to all assets needed by a page, regardless whether they are pushed or explicitly requested by the client. This value may be arbitrary large if the push rate is large as well.

$$Responses = Assets + Mistakes \tag{3.2}$$

where $Responses \in [Assets, Assets + Pushes]$, when $Mistakes = Pushes$.

Obviously, the best prediction algorithm would push exactly $Pushes = Assets$ resources with no mistakes. This leads to $Requests = 0$ and $Responses = Assets$. That is, both mentioned parameters have minimal values. Thus, our aim is to minimize explicit requests, as well as responses sent – either pushed, or not. Figure 3.2 illustrates the relation between the values when the algorithm does not produce mistakes. Ideally, the push rate of the algorithm would be controlled by input parameters.

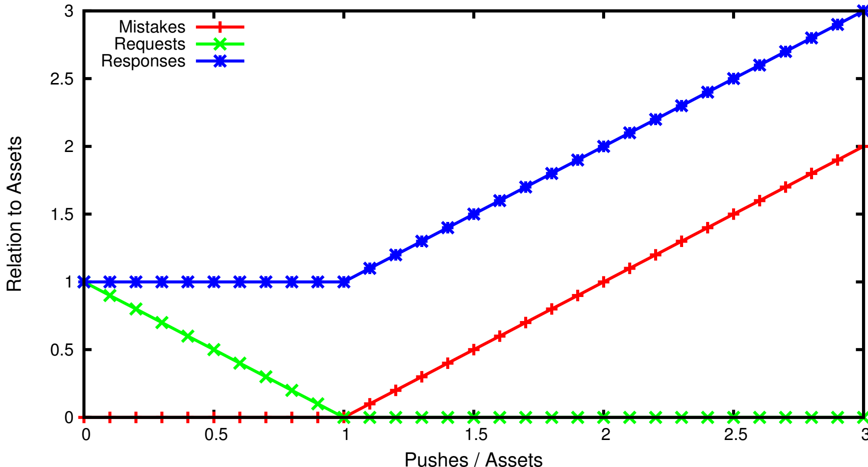


Figure 3.2: Relation between Responses, Requests, Pushes, Mistakes and Assets when the number of mistakes is the minimal possible.

For evaluating the cost of the push, we introduce a cost function which is dependent of the preferences of the server/proxy operator. Thus, we introduce two parameters which can be set according to different scenarios:

- $C_{req} (C_{req} > 0)$ – cost of a request. If the algorithm misses the chance to push a resource, it increases the number of requests and thus latency and bandwidth consumption.
- $C_{resp} (C_{resp} > 0)$ – cost of a response. If the algorithm pushes resources that are not required, it creates additional response traffic.

We introduce a function called $\mathrm{PushCost}$ for calculating the trade-off of SPDY push. While the benefit represents saved round trips (and latency respectively), the cost represents traffic introduced by mistakes. Therefore, the values should be normalized using the parameters C_{req} and C_{resp}. Hence, the cost function, normalized to the number of assets (and thus the total cost of the page) is:

$$\mathrm{PushCost} := \frac{C_{req} \times \mathrm{Requests} + C_{resp} \times \mathrm{Responses}}{(C_{resp} + C_{req}) \times \mathrm{Assets}} \tag{3.3}$$

If we use:

$$\omega := \frac{C_{req}}{C_{resp} + C_{req}} \tag{3.4}$$

where $\omega \in (0, 1)$ is a parameter representing the trade-off, the cost function becomes:

$$\mathrm{PushCost} := \frac{\omega \times \mathrm{Requests} + (1 - \omega) \times \mathrm{Responses}}{\mathrm{Assets}} \tag{3.5}$$

When $\mathrm{Requests}$ and $\mathrm{Responses}$ are substituted according to Equation 3.1 and Equation 3.2, the function transforms to:

$$\mathrm{PushCost} := 1 + \frac{\mathrm{Mistakes} - \omega \times \mathrm{Pushes}}{\mathrm{Assets}} \tag{3.6}$$

It can have the following only positive values:

- $\mathrm{PushCost} \approx 0$ when $\omega \approx 1, \mathrm{Pushes} = \mathrm{Assets}, \mathrm{Mistakes} = 0$. That is, when ω is about 1, the push rate is 100% and there are no mistakes.

- $(1-\omega) \leqslant \mathtt{PushCost} < 1$ when $\frac{\mathtt{Mistakes}}{\mathtt{Pushes}} < \omega$. That is, when the proportion of wrong pushes is smaller than ω. Push is beneficial for the given scenario (given parameters respectively).

- $\mathtt{PushCost} = 1$ when $\mathtt{Pushes} = 0$ or $\frac{\mathtt{Mistakes}}{\mathtt{Pushes}} = \omega$. That is, when no push is used or when the proportion of wrong pushes is equal to ω. In this case, push did not hurt or help system performance.

- $\mathtt{PushCost} > 1$ when $\frac{\mathtt{Mistakes}}{\mathtt{Pushes}} > \omega$. That is, when the proportion of wrong pushes is larger than ω. Hence, push is not beneficial for the operator. If ω is small enough and there are some mistakes, the cost function may always have such value.

The parameter ω is specific for each application. A greater value gives more weight to the possible latency improvements when fewer explicit requests are issued. On the other hand, lower value gives more weight to the possible push mistakes; that is, the additional traffic is not tolerated. The best push rate for a given ω is when the cost function has its global minimum. Figure 3.3 illustrates an example of the cost function for different values of ω when the mistakes increase quadratically on increasing the push rate.

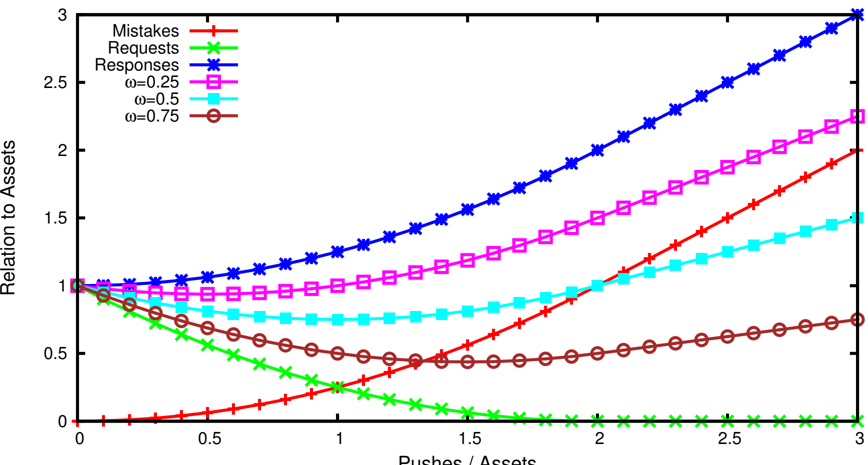


Figure 3.3: An example of the PushCost function for different values of ω under the assumption that the performance of the push heuristic is given by $\frac{\mathtt{Mistakes}}{\mathtt{Assets}} = \left(\frac{\mathtt{Pushes}}{2 * \mathtt{Assets}}\right)^2$.

Note that here, we consider requests and responses only for assets. The proportion of pages among all resources may or may not be negligible depending on the specific web site.

3.2.2 *Prediction Algorithm*

We propose a prediction algorithm – called *basic* – for SPDY server push based only on statistical data. The algorithm has two variations, called *basic-less* and *basic-more*, which are respectively less or more aggressive when pushing.

The principal idea behind the algorithm is to collect statistical data about which assets are requested and how often by clients after requesting certain page – within a specific page load. When the same page is requested in the future, the algorithm decides whether some assets should be pushed, which exactly and how many of them. The algorithm starts with zero knowledge, begins collecting data immediately after the application is launched and possibly starts pushing data almost immediately. That is, it does not have separate learning phase. Learning continues during the whole lifetime of the proxy.

Assets are selected for pushing when the algorithm is confident enough that they would be requested within the same page load. That is, when the calculated probability is at least the *Probability threshold*. This value is computed according to the *Algorithm used* (Table 3.2). The following data is collected and used for the calculation:

- N_{page} – the number of times a page has been seen by the algorithm.
- N_{asset} – the number of times an asset has appeared within the same page load as the given page.
- M_{page} – the number of times a page has been seen after the first page load within a session.
- M_{asset} – the number of times an asset has appeared in page loads before the given page within same session. It may be larger than M_{page}.

Algorithm	Probability
basic	$P_b = \frac{N_{asset}}{N_{page}}$
basic-less	$P_{bl} = P_b \times \left(1 - \frac{1}{N_{page}}\right)$
basic-more	$P_{bm} = P_b + (1 - P_b) \times \text{MIN}\left(1, \frac{M_{asset}}{M_{page}}\right)$

Table 3.2: Calculating the probability for pushing a single asset.

That is, P_b represents the frequency an asset appears in the page load of a page. On the other hand, *basic-less* slightly modifies this value: less weight is given to pages with fewer appearances so far. Hence, a small push rate is expected especially at the beginning of the application's lifetime.

In contrast to both mentioned algorithms, *basic-more* aims to increase the push rate. It considers the cases when an asset needed by the given page is not requested within its page load, because it has been retrieved earlier in the session after another page.

Different input parameters are used to control the push rate and possibly tune the behavior for a specific web site or proxy (see Table 3.3).

Name	Purpose	Possible Values
Algorithm used	Different algorithms calculate the probability for pushing in different ways.	basic, basic-less or basic-more
Maximum page load timeout	The resource after this timeout will be treated as a page starting a new page load.	$[1ms, \infty]$
Page load timeout	The resource after this timeout will be still treated as an asset only if the current statistics says that it is an asset.	$[1ms, \infty]$
Probability threshold	Assets are pushed only when the algorithm has calculated at least this probability.	$[0, 1]$
Minimum history	Assets may be considered for pushing only if the page has been requested at least this number of times so far. Large value will reduce pushes at the beginning of the program's lifetime.	$[1, \infty]$
Maximum pushed assets	Sets the maximum number of assets which are allowed to be pushed for a single page load. Smaller value will reduce the push rate.	$[0, \infty]$ or -1 to ignore it
No push for mobile devices	Do not push assets to mobile devices.	true or false
Consider browser	Separate statistical data will be kept for different user agents.	true or false
Maximum asset size	Push only assets with body size at most this value.	$[0B, \infty]$ or -1 to ignore it
Keep history per user	Different sessions of the same user are linked and previously requested resources are not pushed again.	true or false
Assume pushes as hits	If true, all pushes which have not been canceled by the client are assumed to be hits and therefore statistics for them is updated.	true or false

Table 3.3: Input options for push algorithm.

Another parameter, which is important for the algorithm, but cannot be entirely controlled by the operator, is the SPDY session length. Longer sessions would decrease the number of mistakes while possibly increase the push rate.

It should be mentioned that the algorithm cannot know if a push is a mistake or a hit. A client may cancel a push, but this might or might not mean

a mistake made by the algorithm. In the same way, a push may be accepted by the client, but it still might be an unnecessary resource – a mistake. Furthermore, an already pushed resource might be explicitly requested due to race conditions, a misbehaving client, or a client whose internal state has changed in the course of a session (e.g. a cached resource has expired). The parameter *Assume pushes as hits* controls whether uncanceled pushes contribute to the statistics as if they were explicitly requested.

3.2.3 *Description of the Proposed Algorithm*

Algorithm 1 lists all input parameters and all global maps which are utilized later in various procedures. Auxiliary procedures for updating values in the maps are listed in Algorithm 2. Note that they update the map which is passed as a parameter.

The most important procedure of the algorithm is `SENDING` (see Algorithm 3). It is called by the application when a response is ready to be sent to the client. That is, response status and size are known. The procedure takes the following input arguments:

- `session` – unique identifier of each SPDY session given to the algorithm by the application.
- `resid` – unique identifier of a request within given session (e.g. a sequence number of the requests within a session); given by the application.
- `url` – requested URL. The same resource might be requested multiple times per session.
- `method` – HTTP request method.
- `agent` – user agent string from the request.
- `status` – HTTP response status code.
- `size` – body size of a response.

Within this call, statistical data may be updated. Moreover, if a page is requested, assets may be pushed. The procedure starts with checking request data: If the requested resource is "facicon.ico", the request is ignored by the algorithm. The preliminary tests proved that such requests are independent of requesting other resources, and therefore confuse the algorithm on differentiating pages from assets. (However, such host-specific resources would be perfect candidates for pushing on the first request per session.)

Further, the requesting client is checked against a list of known user agent strings. Thus, the algorithm does not consider requests made by robots because they would confuse the learning. Nevertheless, considering the data available for the evaluations in Subsection 5.3.1, it was observed that there are still robots using valid browser user agent string, which causes mistakes.

If the algorithm is configured to ignore requests from mobile devices, the user agent is checked against a list of known mobile browsers.

Algorithm 1 Push prediction algorithm: input parameters and global maps

Require:

▷ Input parameters

1: O_{im}: ignore mobile browsers
2: O_{a}: algorithm used
3: O_{plt}: page load timeout; ignored if request is detected as asset
4: O_{maxplt}: maximum page load timeout
5: O_{pt}: probability threshold
6: O_{minh}: minimum history
7: O_{maxa}: maximum pushed assets
8: O_{maxs}: maximum asset size allowed for push
9: O_{kh}: keep history
10: O_{ch}: assume non-canceled pushes as hits

▷ Maps for keeping global statistics

11: $stat : Pages \to \mathcal{P}(Assets)$ – all possible assets in page loads of a given page
12: $stat_n : Pages \to \mathcal{P}(Assets \to \mathbb{N})$ – number of appearances of a given asset within page loads of a given page
13: $stat_more_n : Pages \to \mathcal{P}(Assets \to \mathbb{N})$ – counts appearances of assets before given page within same session (used only by 'basic-more')
14: $assets : Assets \to \mathbb{N}$ – counts how many times a resource (URL) is considered as an asset
15: $pages : Pages \to \mathbb{N}$ – counts how many times a resource (URL) is considered as a page
16: $pages_more : Pages \to \mathbb{N}$ – counts how many times a resource (URL) is considered as second page or subsequent in session (used only by 'basic-more')
17: $sizes : Assets \to \mathbb{Q}$ – keeps average size per asset

▷ Maps for keeping session's specific information

18: $last_time : Sessions \to \mathbb{T}$ – keeps the time of the last response per session
19: $pushes : Sessions \to \mathcal{P}(Assets \times \mathbb{Q} \times Resids \times Pages)$ – all pushed resources per session with their size, page load and page
20: $last_page : Sessions \to Resids \times Pages$ – last page per session
21: $pages_dirty : Sessions \to \mathcal{P}(Resids \to \{false, true\})$ – keeps information whether the assets within each page load are not allowed to be added to the statistics
22: $dups : Sessions \to \mathcal{P}(Resids \to \{false, true\})$ – keeps information whether resources in given page loads are not unique
23: $pageloads : Sessions \to \mathcal{P}(Resids \to \mathcal{P}(Resources))$ – all resources for each page load per session
24: $ses_resources : Sessions \to \mathcal{P}(Resources)$ – all resources per session

Ensure:

25: Updates statistics kept in global maps and sends multiple responses for a single requests based on predictions

Algorithm 2 Helper procedures for handling maps

```
1: procedure SET(map,key,value)
2:     map ← map\{key ↦ map(key)} ∪ {key ↦ value}
3: end procedure

4: procedure SET_DEEP(map,key1,key2,value)
5:     inner_map ← map(key1)
6:     SET(inner_map, key2, value)
7: end procedure

8: procedure INC(map,key)
9:     SET(map, key, map(key) + 1)
10: end procedure

11: procedure INC_DEEP(map,key1,key2)
12:     inner_map ← map(key1)
13:     INC(inner_map,key2)
14: end procedure
```

Algorithm 3 Main procedures of the algorithm; they are called by the web server/proxy

```
1: procedure SENDING(session,resid,url,method,agent,status,size)
2:     if ¬(url ∈ '*/favicon.ico') ∧ (agent ∈ known_browsers) ∧
   ¬(O_im ∧ (agent ∈ mobile_browsers)) then
3:         if IS_PAGE(session,url) then
4:             PROC_PAGE(session,resid,url,method,status)
5:         else
6:             PUSH_CANCELED(session, url)
7:             (page_resid, page) ← last_page(session)
8:             PROC_ASSET(session,url,method,status,size,page_resid,page)
9:         end if
10:     end if
11:     SEND_RESPONSE(session,url)
12:     SET(last_time, session, GET_CURRENT_TIME)
13: end procedure

14: procedure PUSH_CANCELED(session,url)
15:     for all (a, s, r, p) ∈ pushes(session) do
16:         if url = a then
17:             SET(pushes, session, pushes(session)\{session ↦ (a, s, r, p)})
18:         end if
19:     end for
20: end procedure

21: procedure SESSION_CLOSED(session)
22:     if O_ch then
23:         for all (asset, size, page_resid, page) ∈ pushes(session) do
24:             PROC_ASSET(session, url, 'GET', 200, size, page_resid, page)
25:         end for
26:     end if
27: end procedure
```

Algorithm 4 Other procedures and functions used by the algorithm (1)

```
1: procedure PROC_PAGE(session,resid,url,method,status)
2:     SET(last_page, session, (resid, url))
3:     ADD_RESOURCE(session, resid, url)
4:     if status ∈ {200, 301, 302} then
5:         if pages(url) ⩾ O_minh then
6:             candidates ← ∅
7:             for all asset ∈ stat(url) do
8:                 if IS_ALLOWED_FOR_PUSH(session,url) then
9:                     prob ← GET_PROBABILITY(url,asset)
10:                    if prob ⩾ O_pt then
11:                        candidates ← candidates ∪ (asset, prob)
12:                    end if
13:                end if
14:            end for
15:            if (O_maxa ⩾ 0) ∧ (O_maxa < |candidates|) then
16:                num ← O_maxa
17:            else
18:                num ← |candidates|
19:            end if
20:            while num > 0 do
21:                (asset, prob) ← MAX_PROB(candidates)
22:                asset_size ← PUSH_RESPONSE(session, asset)
23:                SET(pushes, session, pushes(session) ∪
    (asset, asset_size, resid, url))
24:                candidates ← candidates\(asset, prob)
25:                num ← num − 1
26:            end while
27:        end if
28:        if (O_a = 'basic − more') ∧ (|ses_resources(session)| > 1) then
29:            INC(pages_more,url)
30:            for all resource ∈ ses_resources(session) do
31:                INC_DEEP(stat_more_n,url,resource)
32:            end for
33:        end if
34:        INC(pages,url)
35:    else
36:        SET_DEEP(pages_dirty, session, resid, true)
37:    end if
38: end procedure

39: procedure PROC_ASSET(session,url,method,status,size,page_resid,page)
40:    if (method = 'GET') ∧ (status = 200) then
41:        ADD_RESOURCE(session, page_resid, url)
42:        RECORD_ASSET(url, size)
43:        dup ← dups(session)
44:        dirty ← pages_dirty(session)
45:        if ¬dup(page_resid) ∧ ¬dirty(page_resid) then
46:            SET(stat, page, stat(page) ∪ url)
47:            INC_DEEP(stat_n, page, url)
48:        end if
49:    end if
50: end procedure
```

Algorithm 5 Other procedures and functions used by the algorithm (2)

1: **function** IS_PAGE(session,url)
2: $now \leftarrow$ GET_CURRENT_TIME
3: **if** $last_page(session) = \emptyset$ **then**
4: **return** true
5: **else if** $(now - O_{maxplt}) > last_time(session)$ **then**
6: **return** true
7: **else if** $((now - O_{plt}) > last_time(session)) \wedge ((assets(url) = 0) \vee (assets(url) < pages(url)))$ **then**
8: **return** true
9: **end if**
10: **return** false
11: **end function**

12: **function** IS_ALLOWED_FOR_PUSH(session,url)
13: **if** $url \in ses_resources(session)$ **then**
14: **return** false
15: **else if** $(assets(url) = 0) \vee (assets(url) < pages(url))$ **then**
16: **return** false
17: **else if** $O_{kh} \wedge (url \in$ GET_CLIENT_HISTORY$(session))$ **then**
18: **return** false
19: **else if** $(O_{maxs} \geqslant 0) \wedge (O_{maxs} < sizes(url))$ **then**
20: **return** false
21: **end if**
22: **for all** $(asset, size, page_resid, page) \in pushes(session)$ **do**
23: **if** $url = asset$ **then**
24: **return** false
25: **end if**
26: **end for**
27: **return** true
28: **end function**

29: **function** GET_PROBABILITY(page,asset)
30: $page_assets \leftarrow stat_n(page)$
31: $n \leftarrow page_assets(asset)$
32: $N \leftarrow pages(page)$
33: $basic_prob \leftarrow \frac{n}{N}$
34: **switch** O_a **do**
35: **case** 'basic'
36: $prob \leftarrow basic_prob$
37: **end case**
38: **case** 'basic − less'
39: $prob \leftarrow basic_prob \times \left(1 - \frac{1}{N}\right)$
40: **end case**
41: **case** 'basic − more'
42: $page_more_assets \leftarrow stat_more_n(page)$
43: $m \leftarrow page_more_assets(asset)$
44: $M \leftarrow pages_more(page)$
45: $prob \leftarrow basic_prob + (1 - basic_prob) \times MIN\left(1, \frac{m}{M}\right)$
46: **end case**
47: **end switch**
48: **return** prob
49: **end function**

Algorithm 6 Other procedures and functions used by the algorithm (3)

```
1:  procedure RECORD_ASSET(url,size)
2:      n ← assets(url)
3:      avg ← (sizes(url) × n + size) / (n + 1)
4:      SET(sizes, url, avg)
5:      INC(assets,url)
6:  end procedure

7:  procedure ADD_RESOURCE(session,page_resid,url)
8:      pl ← pageloads(session)
9:      if url ∈ pl(page_resid) then
10:         SET_DEEP(dups, session, page_resid, true)
11:     else
12:         SET_DEEP(pageloads, session, page_resid, pl(page_resid) ∪ url)
13:     end if
14:     SET(ses_resources, session, ses_resources(session) ∪ url)
15: end procedure
```

Next, decision should be taken if the requested resource is a page or an asset (see IS_PAGE in Algorithm 5). If there is no page yet considered for the session, obviously a new page is requested. If the *Maximum page load timeout* has passed since the last response in the session, the request is considered as a page request as well. It is a page also when the *Page load timeout* has passed and the resource was counted as an asset zero times or fewer times than as a page. If no one of the conditions was met, the request is considered as an asset. Here, we do not use only a single timeout value, because in the data investigated in Subsection 5.3.1, it was observed that some mobile clients may issue requests within a single page load delayed seconds from each other, possibly due to high latency. However, the *Maximum page load timeout* limits very long page loads to let the algorithm recover from a missed beginning of a new page load.

The decision is not perfect and therefore the maps `assets` and `pages` are employed to count how many times a resource is considered as one of both types. This information is used for further decisions.

When a new page is requested, it is the exact moment for pushing assets. This happens in PROC_PAGE (see Algorithm 4). The `resid` and `url` of the page are saved to be used for the consequent assets within the same page load. ADD_RESOURCE is applied to add a `url` to a set of all resources for the session and to the set of resources within a page load. The first one is used to forbid an asset for pushing if it has already been requested within the session. The second set is used to detect if there are duplicate resources (same URLs) within a page load, which indicates wrong assumptions made by the algorithm. Preliminary tests showed that robots who could not be filtered may cause such situation. In this case, the page load is marked (added to dups) and further assets within it will not update the statistics as the behavior is not expected.

Let us turn back to PROC_PAGE. The processing continues only if the response code is *200*, *301*, or *302*. We want to stay on the safe side and should not consider other pages to avoid erroneous updates of the statistics due to error pages, for instance. If the response has a different code, the page is marked as "dirty" – its assets will not update the statistics.

Moreover, if the page has been already seen at least *Minimum history* times, then all checks are passed and pushing for the given page is allowed. All assets seen in page loads of the given page are examined in IS_ALLOWED_FOR_PUSH. If an asset has been requested within the same session, it is ignored. If the current state of assets and pages tells us that it is not an asset, but a page, then it is ignored as well. If *Keep history per user* is employed, the available history of resources sent to the client in previous sessions is examined. The proposed algorithm does not answer the question of how such history should be kept (see Subsection 6.4.1 for discussion).

Returning back to IS_ALLOWED_FOR_PUSH, the next check is applied when the parameter *Maximum asset size* is set. The algorithm keeps average size of all assets – the resources can be dynamic and do not need to have the same size. When the option is used, an asset is allowed for push only if its current average size is not larger than the limit. Finally, if the same resource has already been pushed in the same session, it is ignored as well.

For all assets which successfully passed the test, the probability is calculated according to the algorithm applied (see Table 3.2). All assets with calculated probability which is larger or equal the input *Probability threshold* are added to a set of candidates. O_{maxa} candidates which have highest calculated probability are pushed to the client. On pushing, a tuple of the URLs of the asset and the last page, as well as asset size and the ID of the page load (resid) are added to a list for further processing.

After the pushing phase is done, if the algorithm *basic-more* is used, its statistics is updated. That is, all the resources in the given session are marked that they have appeared before the given page.

If the call to IS_PAGE in SENDING returns false, then the resource is considered as an asset. Firstly, if the asset has been pushed to the client without being canceled, and now it is being requested, the algorithm is affected exactly as if the push has been explicitly canceled – PUSH_CANCELLED is called. Hence, the push is forgotten while the explicit request would possibly contribute to the global statistics. Next, PROC_ASSET is called. The algorithm considers only assets returned to *GET* requests with response code of *200*. The asset is added to the resources of the session and to those of the current page load. Its average size is also updated. However, the statistics that the asset appears after the last page is updated only if there are no duplicate resources within the page load and if the page load is not marked as dirty. As already mentioned, the first one indicates erroneous decisions about the page load while the second presents the case when the response status of the page does not allow us to consider its assets.

Finally, the requested resources – either a page, or an asset is sent as a response to the client.

The second procedure which might be called by the application is PUSH_CANCELED. It is invoked when a *RST_STREAM* frame is received to notify that the client does not want the pushed resource. In such case the resource is removed from the set of pushed assets.

The last procedure called by the application – SESSION_CLOSED – notifies the algorithm that the session has been closed by the client or the server. Since the algorithm cannot know if the pushed resources were hits, the input parameter *Assume pushes as hits* is applied. If it is true, then the assets which are still in the set of pushed resources are considered exactly as if they were explicitly requested assets. That is, all statistics kept is updated for each of those resources.

It should be mentioned that the input parameter *Consider browser* was not used in the given algorithm for simplicity. When the option is used different global maps for "different browser" should be used to keep separate statistics. That is, individual maps per each browser are needed for replacing $stat$, $stat_n$, $stat_more_n$, $assets$, $pages$, $pages_more$, and $sizes$. Here, the meaning of "different browser" is up to the implementer. It may be, among the others, browser name, browser version, browser major version, browser and platform, and so forth.

3.2.4 *Algorithm Complexity*

The algorithm would have the worst complexity when the number of pages is small, the number of assets is huge, sessions are very long and the input parameters are chosen such that the push rate is large.

Function `SENDING` is called on each request and its time complexity depends on the *Maximum pushed assets* parameter. If the latter is set, the time complexity is $\mathcal{O}(n \log n)$ due to the required sorting in `PROC_PAGE`. n is the total number of requests passed to the algorithm so far as in the worst case all of them would be selected for pushing.

`PUSH_CANCELED` has time complexity of $\mathcal{O}(1)$. If the *Assume pushes as hits* parameter is set, `SESSION_CLOSED`'s complexity is $\mathcal{O}(n)$ where n is the number of pushed assets for the given session.

The space complexity of the whole algorithm is $\mathcal{O}(kn^2)$ where k is the number of opened sessions at the same time, and n is the number of resources seen so far by the algorithm. The space used by the algorithm increases with the lifetime of its instance. The presented algorithm does not consider any space limits. In a real scenario when a memory limit is reached, statistical data for pages and assets which are rarely requested should be removed to free memory.

4 IMPLEMENTATION

4.1 SPDY LIBRARIES

Since SPDY is a relatively new protocol, there are only a few other projects that currently offer a library for implementing SPDY server or client.

As of the beginning of this project, spdylay [Tsu] was the only one which supported most of the client and server features of SPDY/2 and SPDY/3, and it has been constantly improved since then. Its API is slightly low-level: for example, the user has to manually invoke the functions for reading and writing data for a specific SPDY session; thus, spdylay client code will typically be rather complex.

This thesis introduces a new library called libmicrospdy. Its central goal was to provide simple high-level server API, similar to that of GNU libmicrohttpd [Gro]. Our goal is to eventually also support the various styles of event loops supported by GNU libmicrohttpd; however, for now, libmicrospdy only supports single-threaded operation where the server handles all connections using a single `select`-based event loop.

4.1.1 *libmicrospdy Overview*

libmicrospdy's protocol implementation is based on [BP12], which was the only SPDY Protocol draft submitted as such to the IETF at the time of starting the project.

The prefix `SPDY_` is used for all API calls. Required API calls for event loop integration are provided: `SPDY_get_fdset`, `SPDY_get_timeout` and `SPDY_run`. On events (such as new session or new request) the latter calls the callback functions which were provided when the user initially started the libmicrospdy daemon.

As suggested in the draft [BP12], the library is separated internally into a *framing layer* and an *application layer*. The library's user has neither knowledge, nor access, nor the need to know anything about the framing layer. The same is valid for the underlying Input/Output (IO) system with the exception that the file descriptors for the client sockets are accessible. However, the user must use them only for passing them to `select` calls.

Having received a request, the user is required to prepare HTTP headers and an HTTP body for the response, using the respective API calls. There are two ways of providing body data: 1) directly to the responsible function as an array of characters, or 2) chunk by chunk via a callback using a chunk size specified by the user. Using callbacks makes it possible to handle replies that are too large to fit into memory.

Although not mentioned in [BP12], the browsers use SPDY/3 *only* over TLS and *only* after it is negotiated via TLS extension (Next Protocol Negotiation (NPN) [Lan12#1]). However, libmicrospdy now supports both SPDY over TLS and SPDY without encryption. The latter case implies that a connecting client will know in advance that the server speaks only SPDY, since there is no negotiation phase. libssl (OpenSSL) [Ope] is used by the library for IO operations when TLS is applied, because it supports NPN. However,

libmicrospdy internally has a clean interface towards the IO operations; it should thus be possible to support other TLS implementations in the future.

Following the initial specification, zlib [GA] is used for HTTP header compression.[1] Compression of the HTTP body data is not provided by the library. It is up to the user to give the API calls compressed data and respectively the required HTTP headers.

Currently libmicrospdy does not support all the SPDY features. However, the library does support everything needed for handling *GET* and *POST* requests. The framing layer uses internal callbacks which should make it easy to extend the library to support some of the not yet implemented SPDY frames as well as possible new types in future versions of the protocol.

Since May, 2013, libmicrospdy's code has been a part of the GNU libmicrohttpd's source tree.

4.2 PROXIES OVERVIEW

There exist several HTTP to SPDY and SPDY to HTTP proxies. spdylay itself comes with the program *shrpx*, which supports several different proxy modes, including a reverse and a forward proxy. However, the HTTP to SPDY proxy cannot utilize SOCKS, and the SPDY to HTTP proxy cannot be used as a forward one, which is required for the setup proposed on Figure 3.1. Thus, two new proxies were created to be utilized with Tor, and we believe that they are also faster than shrpx. Both can be found in the GNU libmicrohttpd's source tree.

4.2.1 *mhd2spdy – an HTTP to SPDY Forward Proxy*

mhd2spdy is a proxy based on GNU libmicrohttpd as an HTTP server and spdylay as a SPDY client. The program is single-threaded and all operations happen in a select-based event loop. It must be started with an address of a SPDY to HTTP proxy and must be given the type of communication with the latter: TLS encryption or raw SPDY protocol.

The proxy establishes a single SPDY session to the other one and multiplexes all requests. No caching is provided at this time.

Modifications to the SPDY client were made to reduce the number of SPDY *RST_STREAM* frames. If the client closes a stream, frames for the same one may continue to arrive due to the latency to the second proxy. By default, spdylay sends a *RST_STREAM* frame for each received illegal frame. Since the latency in Tor may be high, such a behavior might largely increase the traffic between both proxies. Thus, mhd2spdy is modified to send at most one *RST_STREAM* per stream.

4.2.2 *microspdy2http – a SPDY to HTTP Proxy*

microspdy2http is a proxy based on libmicrospdy as a SPDY server and libcurl [Ste] – its multi interface – as an HTTP client. The program is single-

1 Compressing headers with the *DEFLATE* algorithm of zlib is known to create the security risk of exposing words from the compressed text (the so-called CRIME attack [RD12]), which in the case of HTTP headers may include the cookie. It is particularly dangerous to compress request headers, as they contain always the cookie – if the later has been set – whereas responses typically only include the cookie when it is first set. libmicrospdy enables HTTP header compression; thus, users of the library should be careful about not retransmitting *Set-Cookie* directives with each response.

threaded and all operations happen in a select-based event loop. It may be started in either TLS, or raw SPDY mode.

The proxy has several different modes:

- Default – the proxy expects to receive full URLs in the ":path" SPDY header of requests. Hence, libcurl requests exactly the same HTTP resource.
- Transparent proxy – libcurl builds a URL for retrieving based on the ":host" and ":path" SPDY headers.
- Reverse proxy – all HTTP connections are established to a specified backend address.

microspdy2http supports setting a timeout value which is used for the HTTP connections. That is, when the response is not fully received within the given number of seconds, this connection is canceled and discarded. No caching is provided at this time.

4.2.3 *Proxies and Tor*

For using the proxies with Tor, microspdy2http is run on exit nodes in transparent mode with disabled TLS. To make the first proxy find the second one, always the same address must be used. As a result, the address 192.0.2.80:9980 was chosen. The prefix 192.0.2.0/24 is forbidden for routing on the Internet [ACV10], but is still not a private address. That is, it is treated by Tor as global address space. The same address must be set to the loopback interface of the exit node and furthermore, the application layer address used must be allowed in the relay's exit policy. Finally, microspdy2http is set to listen on 192.0.2.80:9980, and only local process can connect to the proxy.

mhd2spdy is configured to establish SPDY sessions – without using TLS – only with 192.0.2.80:9980. To simplify the deployment of a SOCKS client, which communicates with the Tor proxy, the program torsocks [Tor] is employed. The latter is loaded in the process space of a program and tries to override all its network calls to establish and use SOCKS communication.

It should be mentioned that some simple benchmarks have shown that using TLS between both proxies has little effect on performance. However, we chose to not use encryption as more control of flushing data written to the TCP sockets should happen. Otherwise, OpenSSL controls these operations.

5 EVALUATIONS

In this chapter, we evaluate the SPDY protocol versus HTTP and SPDY used within the Tor network according to the design presented in Section 3.1. Then, the presented prediction algorithm for SPDY server push is evaluated and finally, results from a measurement about HTTP and SPDY header and body sizes are given to make assumptions about the traffic changes when SPDY push is employed.

5.1 SPDY VS. HTTP

Experiments were made to compare the performance of the SPDY protocol to the well-known HTTP. libmicrospdy was used as a SPDY implementation.

5.1.1 *Setup for the Experiments*

All the different setups were compared in the experiments. They are summarized in Table 5.1. All the clients request always the same web page and all the resources needed to properly display it, which are all on the same host. For the purpose of these experiments a real large web page from NASA was chosen and minimized to a relatively big HTML file ($\approx$ 385 KB) and its resources: 3 CSS files (10–30 KB each) and 1311 image files of different type (100 B–30 KB each). All the HTTP body data received for a full page request (including all resources) is exactly 3,560,921 B. Such a large number of resource requests leads to bad user experiences with HTTP, especially when the connection to the server suffers big latency. This particular experiment might be slightly extreme; we picked this site, as we wanted to ensure that SPDY improvements would be clearly visible in the experimental results.

Short Name of the Setup	Protocol	Client Software	Server Software	Note
HTTP/1.0	HTTP/1.0	curl	Apache	Single request per TCP connection
HTTP/1.1	HTTP/1.1	curl	Apache	Multiple requests per TCP connection
HTTPS/1.0	HTTP/1.0 + TLS	curl	Apache + mod_ssl	Single request per TCP connection
HTTPS/1.1	HTTP/1.1 + TLS	curl	Apache + mod_ssl	Multiple requests per TCP connection
SPDY/fileserver	SPDY/3 + TLS	spdycat from libspdylay	Small web server serving files from the file system, realized with libmicrospdy	All requests in a single TCP connection
SPDY/proxy	SPDY/3 + TLS	spdycat from libspdylay	SPDY-to-HTTP proxy realized with libmicrospdy and libcurl	All requests in a single TCP connection at the SPDY side, multiple requests per TCP connection at the HTTP side; the HTTP requests are served by Apache on localhost

Table 5.1: Different client/server setups used in the experiments.

For SPDY measurements, we used spdycat – part of spdylay – for the SPDY client to request the HTML page and all its resources within a single call. For the HTTP measurements, the curl command-line tool was used to first retrieve the HTML file (always in a separate TCP session), and after that, (possibly parallel) requests were made for all of the other elements. HTTP pipelining was not considered in this set of experiments.

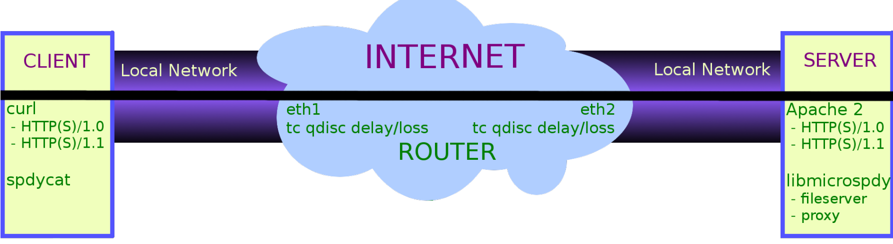


Figure 5.1: Setup for the SPDY vs. HTTP experiments.

Figure 5.1 illustrates the testbed used for the experiments. All the client requests are made from the machine called *CLIENT* and all the server software is on the *SERVER*. Between them, there is a *ROUTER* which plays the role of the Internet. Different delays and loss rate of packets on the way from the client to the server are introduced using the Linux *traffic control* tool (`tc`) and its *queueing discipline* (`qdisc`) at both interfaces of the *ROUTER* to simulate different real world latencies – RTT – and packet loss rates. The exact command used to introduce the desired latency is:

```
# tc qdisc add dev <interface> root netem delay <delay>ms
<dev>ms 25% loss <loss>% 25%
```

It is used with same values on both interfaces – *eth1* and *eth2* – to obtain a symmetric link. This means that the resulting RTT is twice the specified delay.

The desired concurrency for HTTP setups is accomplished with `xargs`:

```
# cat <file-with-all-URLs> | xargs
-n <number-of-URLs-passed-to-one-curl-call>
-P <number-of-concurrent-curl-processes>
curl <curl-option1> <...>
```

For running the experiments the following hardware and software was used:

- Three Virtual Machine (VM)s run as guest OS Debian x86_64 GNU/Linux with kernel 2.6.32-5-xen-amd64.
- Each VM has two virtual CPUs – Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz – and 257,884 KB of memory.
- The physical machine, running all the VMs, has 32 CPUs – Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz – and 1,0426,236 KB of memory and runs as host OS x86_64 GNU/Linux with kernel 2.6.32-5-xen-amd64.

Relevant details about the used application software and its configuration include:

- TLS compression is turned off everywhere.
- All TLS setups use the same certificate and support only RC4-SHA for the cipher suite.

- When running curl in parallel for the HTTP setups, the full TLS handshake must be done for each process. Note that this is not what typical browsers would do, as they can cache TLS session information and thus use a simpler TLS handshake for all but the first connection.
- SPDY headers in requests are not compressed, while those in the responses are.
- Apache issues three more headers in the responses than libmicrospdy when it directly accesses the disk. This means that SPDY/fileserver produces less raw header data compared to all the other implementations.
- Internet Protocol version 4 (IPv4) is used for the network protocol.

In the experiments, the wall clock time needed for requesting and receiving the whole web page with all its resources for each different protocol setup is measured as well as the traffic throughout the *ROUTER*. For traffic measurements, we report the number of bytes and the number of packets forwarded.

Two sets of experiments were performed:

1. Simulating browser connection(s) to one host: For a given RTT from client to server and a given packet loss rate, the web page with its resources is requested using all the tools in Table 5.1. The HTTP tools use 6 parallel processes. Thus, HTTP/1.1 tools use exactly 6 TCP connections to request all of the resources; HTTP/1.0 tools use in every one moment at most 6 TCP connections – parallel processes respectively – and new processes are created until all of the resources are received; SPDY tools always use just one process/TCP connection. We are using 6 as this matches the behavior of most of the current versions of the popular browsers [Smi12].
2. For a given RTT from client to server and a given packet loss rate the web resource is requested 50 times, using all the tools in Table 5.1. Each time, the HTTP tools use $n \in [1, 50]$ parallel TCP connections. Thus, HTTP/1.1 tools use exactly n TCP connections – parallel processes respectively – to request all of the resources; HTTP/1.0 tools use in every one moment at most n TCP connections – parallel processes respectively – and new processes are created until all of the resources are received; SPDY tools always use just one process/TCP connection.

5.1.2 *Results*

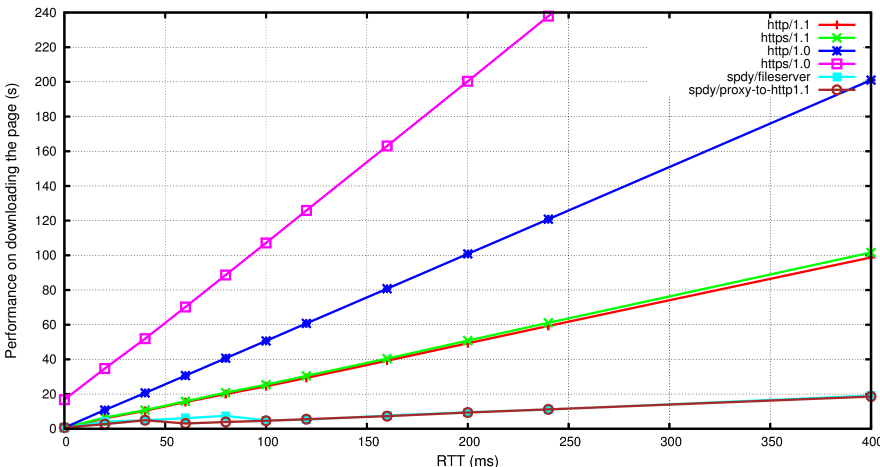


Figure 5.2: Download time for the entire web page in relation to network latency, allowing 6 parallel TCP connections for all HTTP setups. Median value of 6 runs is reported.

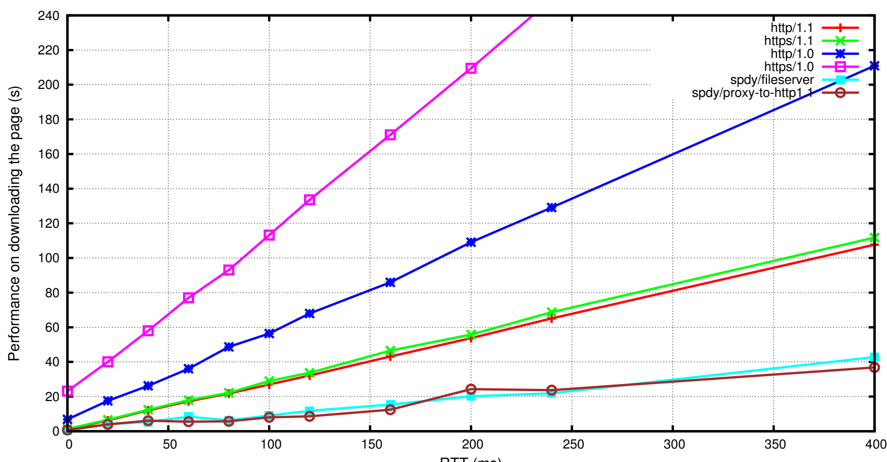


Figure 5.3: Download time as in Figure 5.2, except RTT deviation is set to 20% of RTT and packet loss rate is 5%.

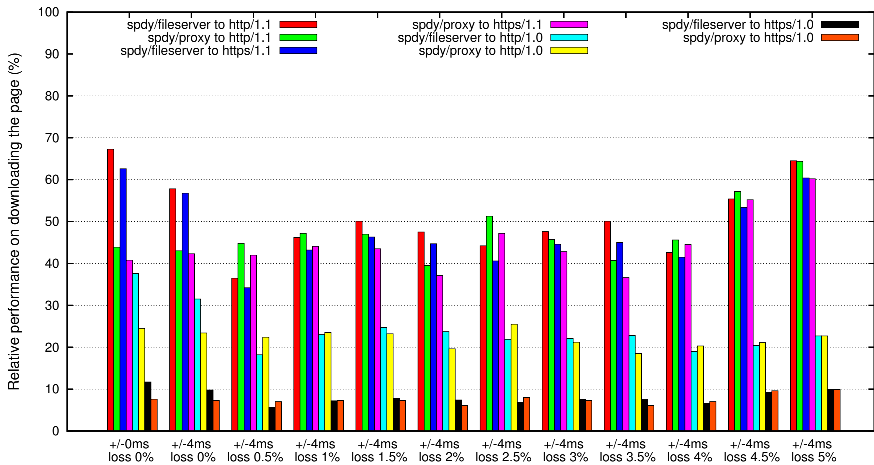


Figure 5.4: Download time for the entire web page needed by SPDY setups in relation to the time needed by HTTP setups for RTT 20ms and different packet loss rates, allowing 6 parallel TCP connections for the latter. Median value of 6 runs is used.

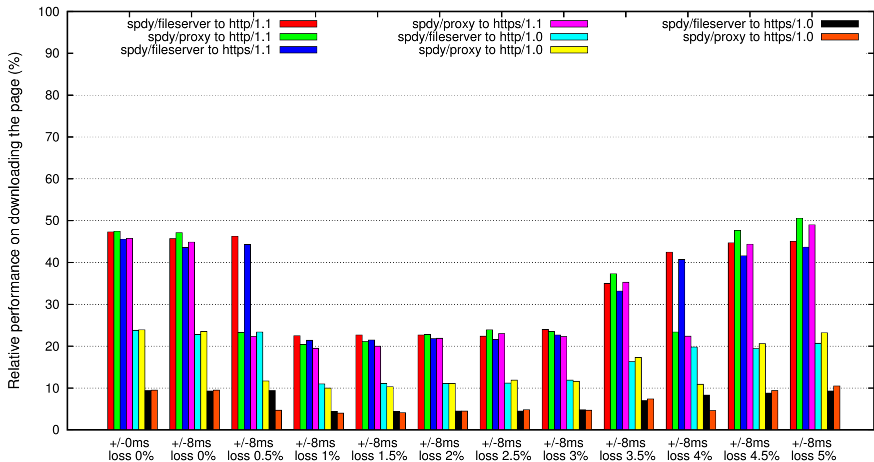


Figure 5.5: Download times relation as in Figure 5.4, except with 40ms RTT.

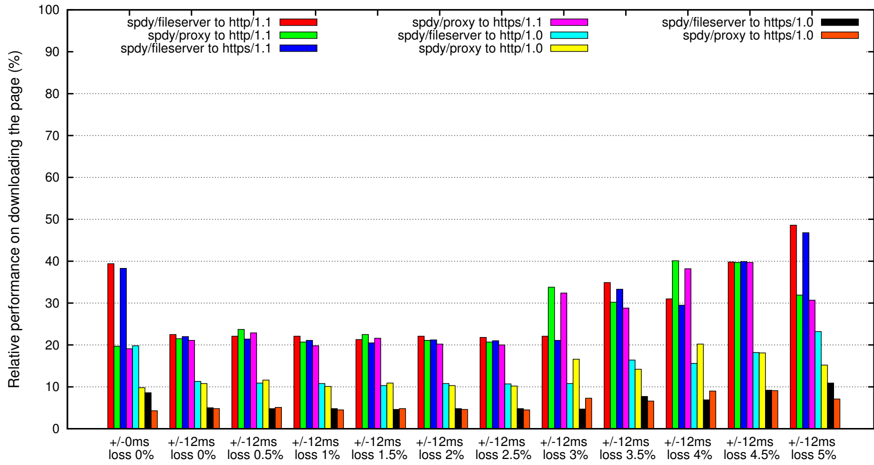


Figure 5.6: Download times relation as in Figure 5.4, except with 60ms RTT.

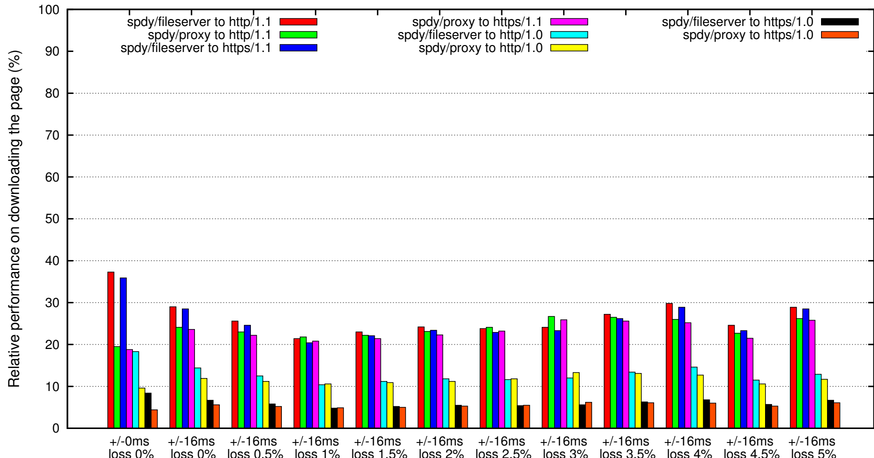


Figure 5.7: Download times relation as in Figure 5.4, except with 80ms RTT.

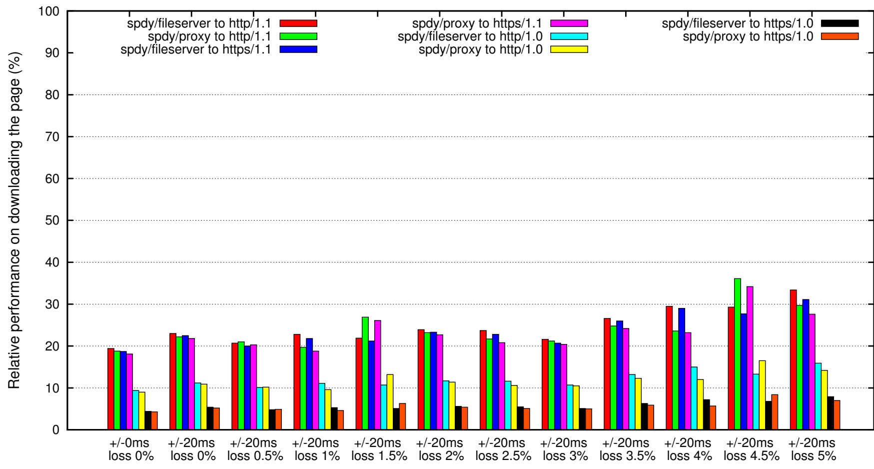


Figure 5.8: Download times relation as in Figure 5.4, except with 100ms RTT.

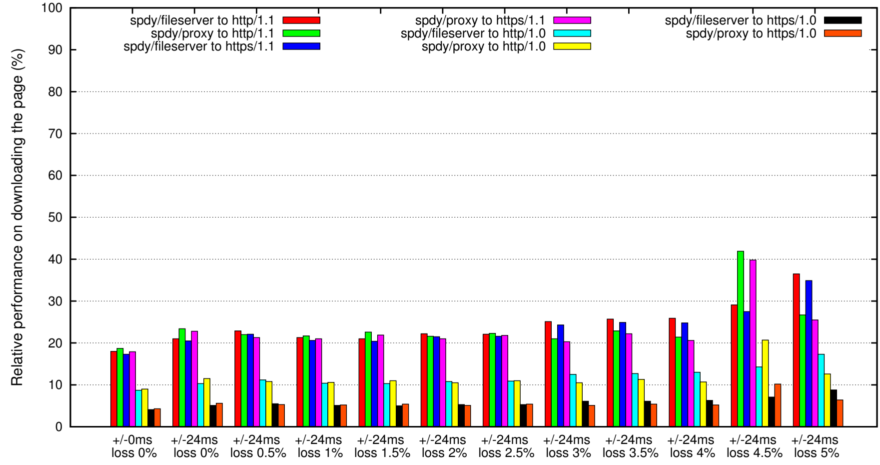


Figure 5.9: Download times relation as in Figure 5.4, except with 120ms RTT.

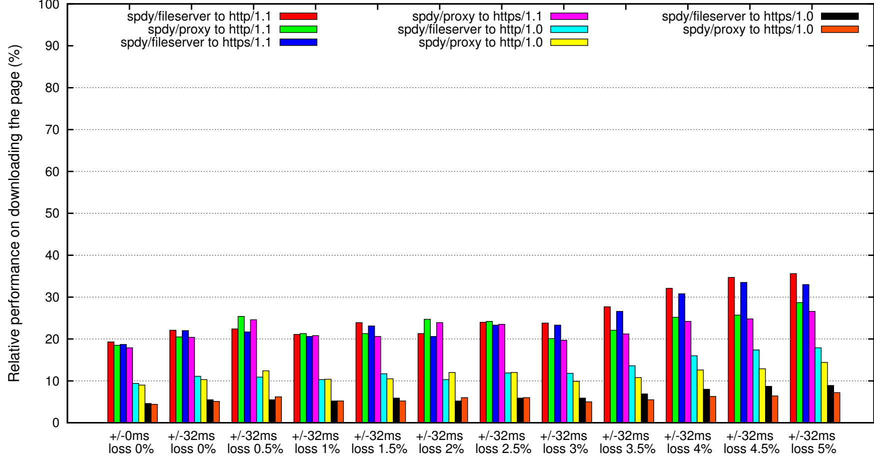


Figure 5.10: Download times relation as in Figure 5.4, except with 160ms RTT.

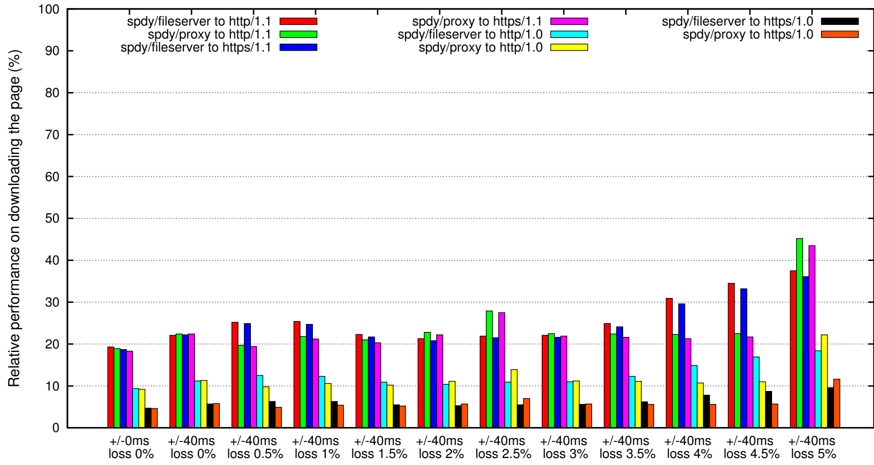


Figure 5.11: Download times relation as in Figure 5.4, except with 200ms RTT.

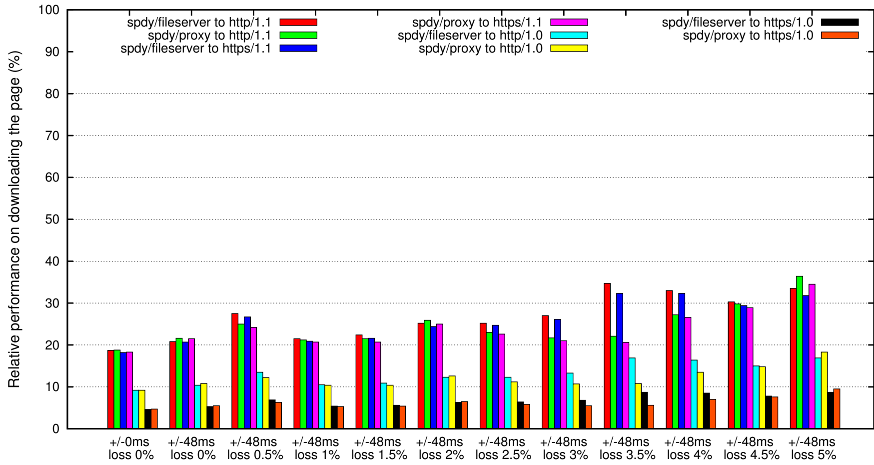


Figure 5.12: Download times relation as in Figure 5.4, except with 240ms RTT.

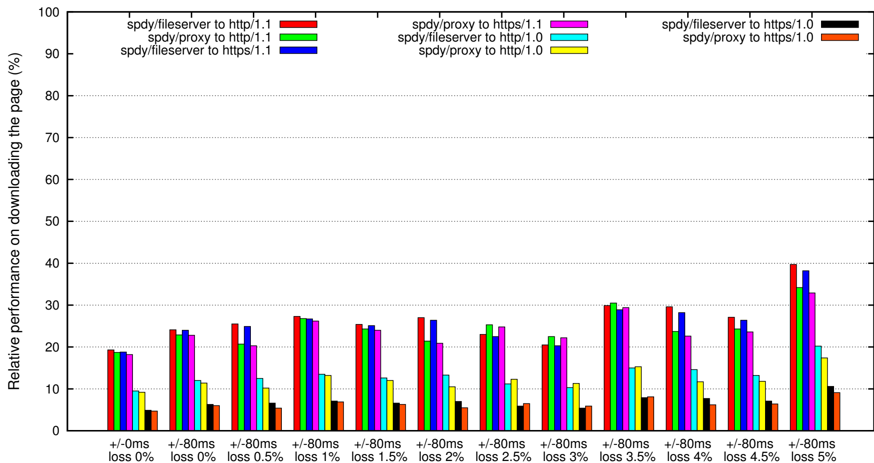


Figure 5.13: Download times relation as in Figure 5.4, except with 400ms RTT.

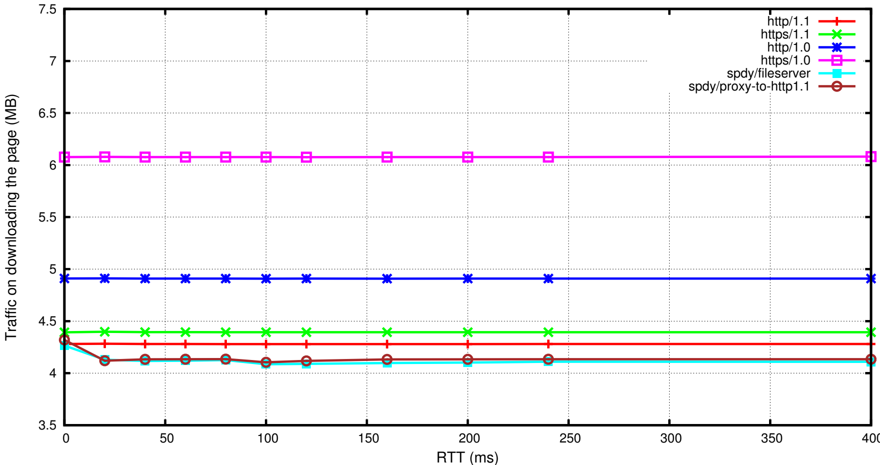


Figure 5.14: Network traffic (in bytes) for downloading the entire web page in relation to network latency, allowing 6 parallel TCP connections for all HTTP setups. Median value of 6 runs is reported.

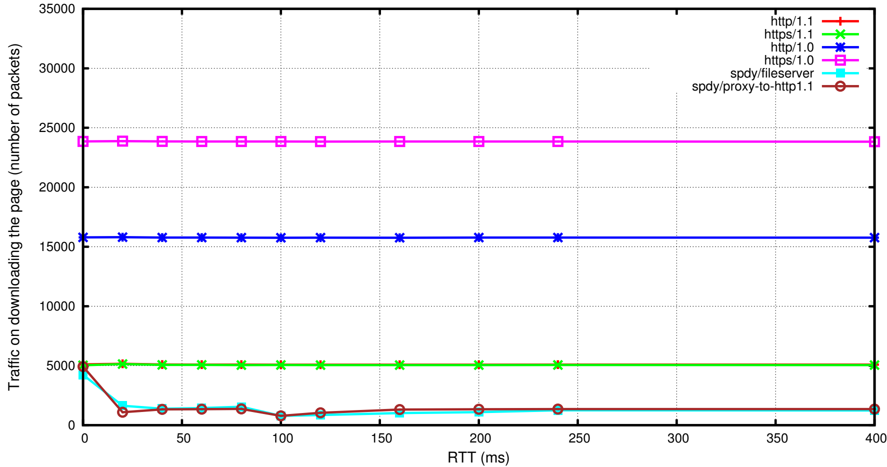


Figure 5.15: Network traffic (in packets) for downloading the entire web page in relation to network latency, allowing 6 parallel TCP connections for all HTTP setups. Median value of 6 runs is reported.

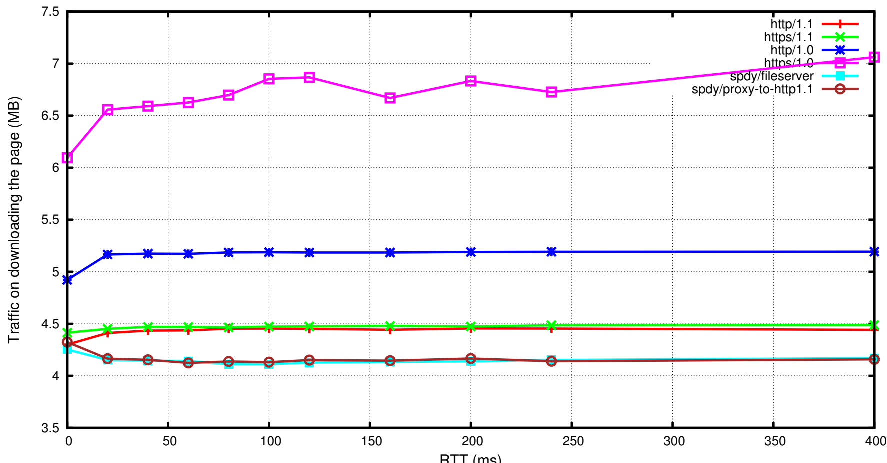


Figure 5.16: Network traffic (in bytes) as in Figure 5.14, except RTT deviation is set to 20% of RTT and packet loss rate is 5%.

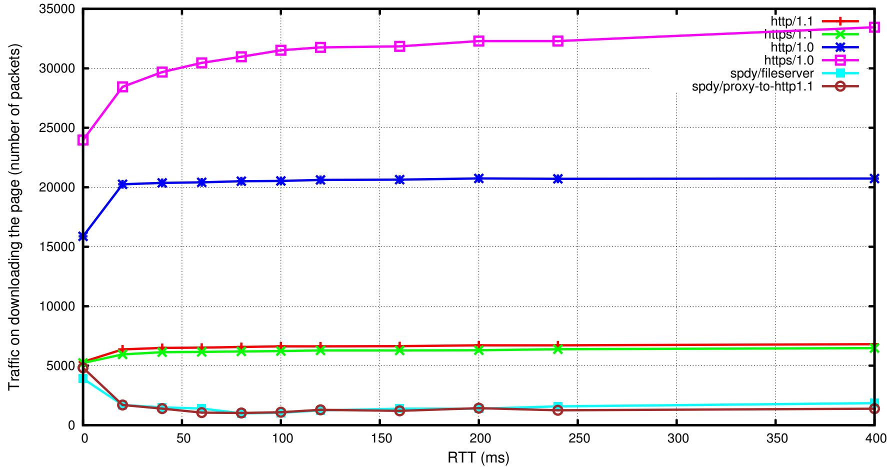


Figure 5.17: Network traffic (in packets) as in Figure 5.15, except RTT deviation is set to 20% of RTT and packet loss rate is 5%.

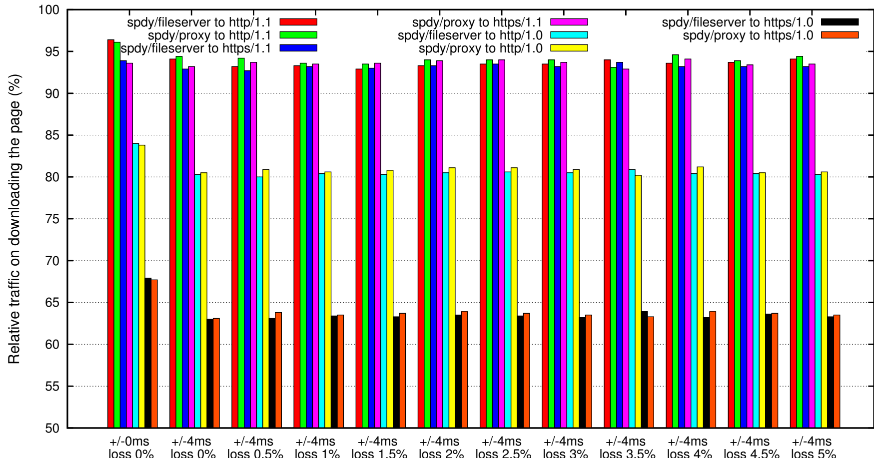


Figure 5.18: Network traffic (in bytes) for downloading the entire web page for SPDY setups in relation to the traffic for HTTP setups for RTT 20ms and different packet loss rates, allowing 6 parallel TCP connections for HTTP. Median value of 6 runs is reported.

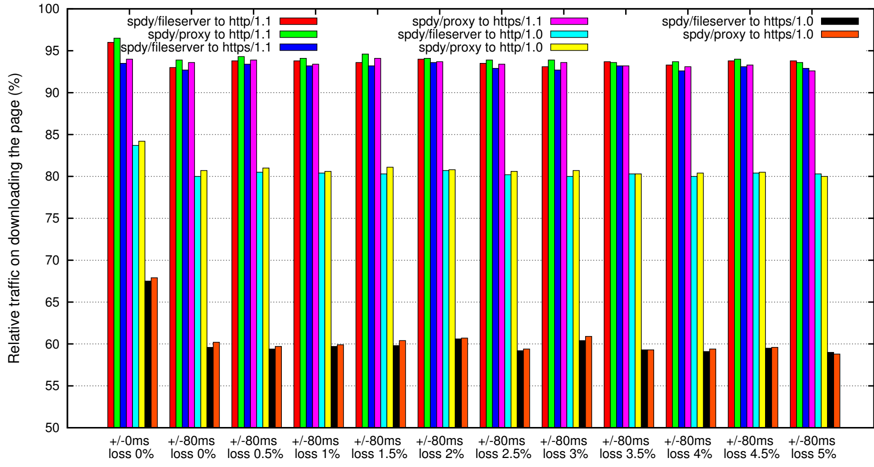


Figure 5.19: Network traffic (in bytes) as in Figure 5.18, except with 400ms RTT.

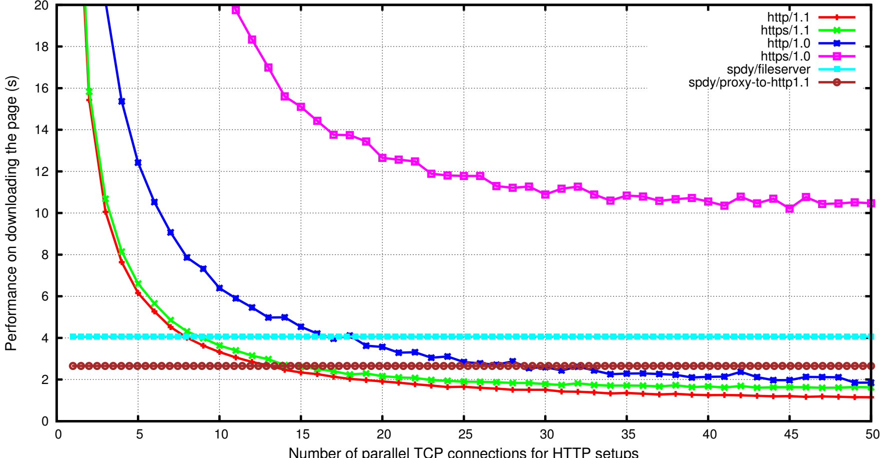


Figure 5.20: Download time for the entire web page in relation to the number of parallel TCP connections for HTTP setups averaged over 2 runs for RTT 20ms. The values for SPDY setups are median values of 6 runs.

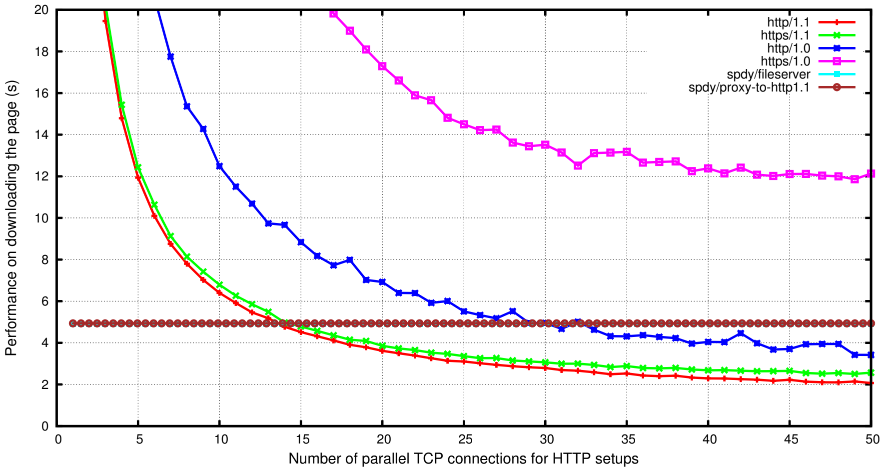


Figure 5.21: Download time as in Figure 5.20, except with 40ms RTT.

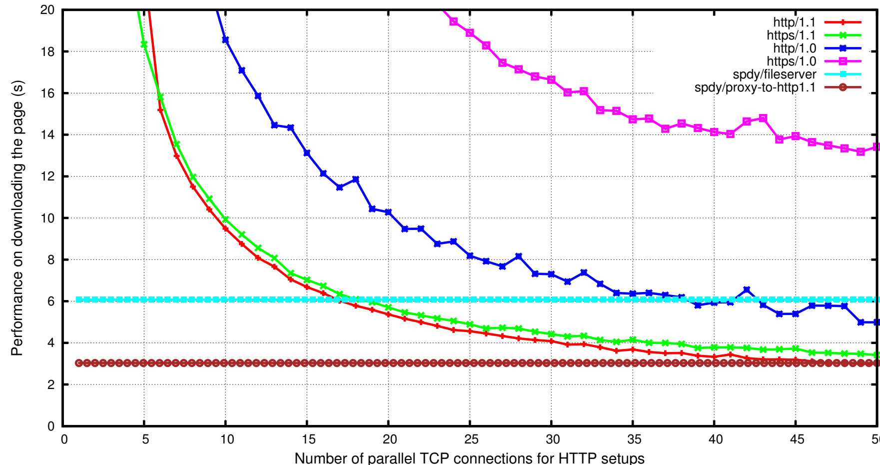


Figure 5.22: Download time as in Figure 5.20, except with 60ms RTT

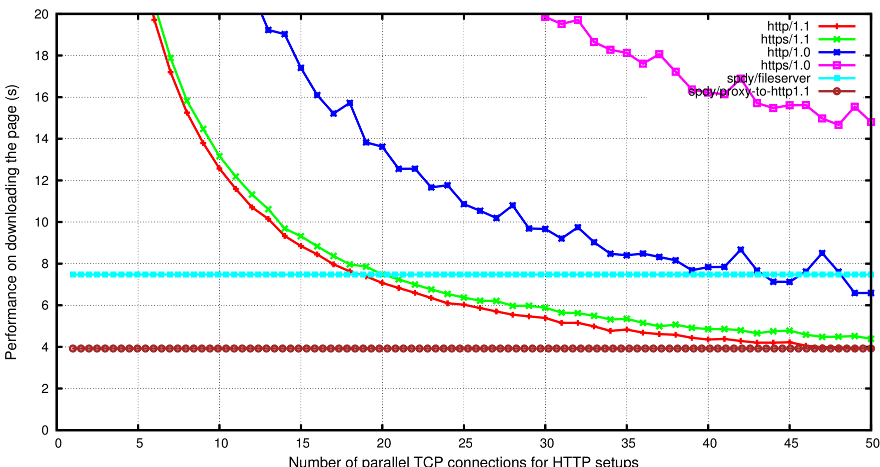


Figure 5.23: Download time as in Figure 5.20, except with 80ms RTT

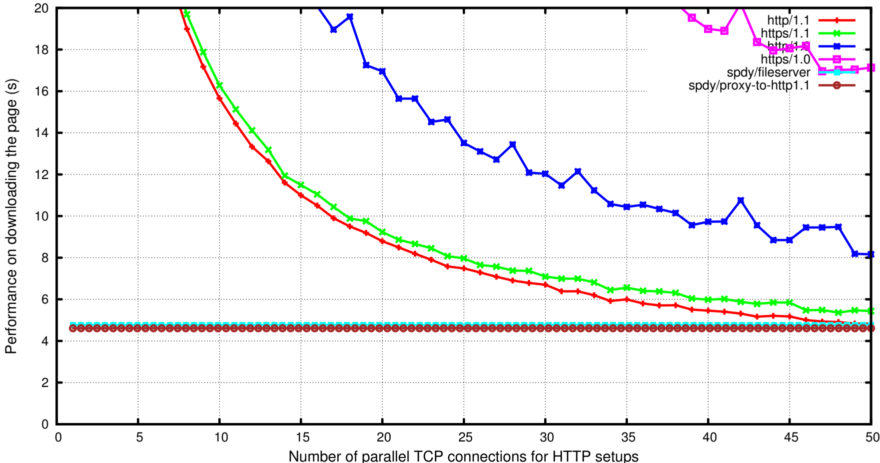


Figure 5.24: Download time as in Figure 5.20, except with 100ms RTT

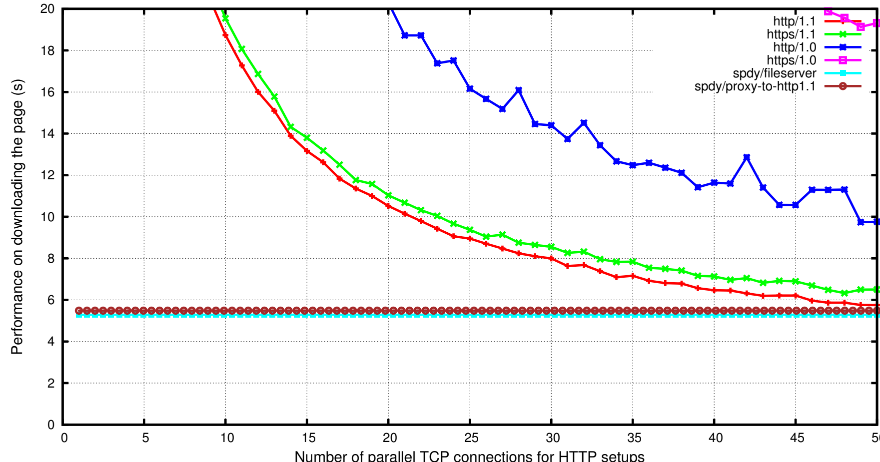


Figure 5.25: Download time as in Figure 5.20, except with 120ms RTT

5.1.3 *Discussion*

The results show that the tested SPDY setups need less time and produce less traffic compared to all of the HTTP setups in almost all of the cases. Of course, these experiments send requests only to a single host. Real world web sites may include resources from multiple hosts. In the latter case SPDY may not show such good results; however, splitting web sites over multiple servers is partially an artifact of operators trying to speed up HTTP (as serving from multiple servers allows browsers to issue more requests in parallel).

The only situations in the experiments when the HTTP setups are faster than SPDY are:

- When the RTT is negligible (e.g. in fast local networks or for *localhost*). The results for the needed time and for the produced traffic are very close but HTTP tends to win. However, it is important to notice that libmicrospdy is currently single-threaded. The latter means that Apache has the advantage of handling incoming requests in parallel and therefore this affects the results.

- When the RTT is small and a great number of concurrent TCP connections is used. This may be seen as the case when a browser opens connections to multiple hosts. On increasing the latency to the server, HTTP speed slows down. Notice here that the single-threaded SPDY server affects the results again.

As it can be seen on Figure 5.14 and Figure 5.15, when there is no loss in the network, each setup uses almost constant traffic regardless of the RTT. The SPDY traffic is around 95% of the traffic for HTTP/1.1 and HTTPS/1.1 and much lower compared to HTTP/1.0 and HTTPS/1.0 (Figure 5.18 and Figure 5.19). The reason for the reduced traffic are compressed headers (only in responses, the used SPDY client for now does not use compression in requests, because of security concerns) and the lower number of IP packets. While HTTP/1.1 normally uses at most one TCP/IP packet for sending a small response, SPDY employs frames and therefore in one packet, there could be multiple frames containing headers and data. That leads to fewer packets, and fewer TCP and IP headers respectively, but increases the size

of the TCP payload, because of the SPDY frame headers. Owed to the Maximum Transmission Unit (MTU) size this is useful primarily for multiple very small files (e.g. small images). In case of negligible RTT, the requests are received and handled very quickly by the SPDY server, and the output buffers are flushed more frequently. This produces smaller packets and thus there are then no traffic benefits at all. However, 5% less traffic is not so much but it should be considered that when the security problem with compressing SPDY headers is solved (note that Chrome already uses modified zlib and compresses part of the headers [Lan12#2]), even better results can be expected.

These bandwidth savings cannot entirely explain the performance advantages of SPDY. For an RTT of 400ms, the SPDY server and the proxy complete the download of the entire page in around 20s, which is less than 20% of the time needed by HTTP/1.1 and HTTPS/1.1. Even for small RTTs such as 20ms (Figure 5.4) SPDY still gives significant improvements, taking just around 40% of what HTTP/1.1 requires. Thus, SPDY's use of a single TCP connection (which has a chance to better adjust its congestion control parameters to the network) must be an additional significant contributor. Figure 5.20 to Figure 5.25 show that SPDY performs better than the other setups even when more than 6 parallel processes (and TCP connections) are used by HTTP. For RTT around 100ms already 50 concurrent TCP connections are needed by HTTP/1.1 to achieve the same performance as SPDY. With such a large number of connections, TCP congestion control is hardly effective.

SPDY's performance advantage slightly decreases with higher levels of packet loss. This might be explained by the single TCP connection's congestion control being more strongly affected by a single loss. Given probabilistic packet loss, concurrent TCP flows can recover faster (as some streams are unaffected) and thus obtain higher throughput.

When comparing the SPDY/proxy and SPDY/fileserver setups, one should keep in mind that they behave almost the same way, with small exceptions:

- In general the traffic should be almost the same. However, the proxy uses slightly more traffic, because of three more headers which are added by Apache to the responses.
- Without any latency in the network SPDY/fileserver is clearly faster since it directly reads files from the system while SPDY/proxy receives the data from Apache via libcurl. However, it is hard to see this difference when there is latency.

SPDY/proxy uses the same end server as HTTP/1.1, but it shows better results and even provides TLS. This demonstrates that our proxy can be a good way to enable faster access to existing HTTP-only servers without modifying them. (Note here that multiple concurrent clients were not tested, as the library is currently single-threaded whereas Apache would use multiple cores.)

SPDY always includes encryption using TLS. Despite the overheads of providing an encrypted connection, SPDY outperforms HTTP in terms of bandwidth consumption and latency.

5.2 SPDY OVER TOR VS. HTTP OVER TOR

For testing if Tor users would benefit from employing SPDY, we measured the page load time of real popular web sites and the traffic which they generate into the Tor network with and without SPDY. While the first one is important for the end user, the second is significant for relay operators.

5.2.1 *Experimental Setup*

To accomplish this, a small web page was created (see Appendix A.3). It is used for loading multiple web pages consequently into a frame and measuring the time needed for loading each of them. JavaScript code controls the whole process. Page load time in this measurements denotes the time from the start of the loading until the moment when the JavaScript event `onload` is triggered for the frame. At that moment the time is measured and a blank page is loaded into the frame. Note here that modern web sites extensively utilize AJAX and therefore, loading page elements which do not affect the `onload` event is common. As a result, more resources are likely to be loaded after the event and new ones may continue to be requested as long as the page stays open into the browser. Here, we consider only the `onload` event, which may not meet the users' subjective feeling of page load. Moreover, less traffic may be produced, since the browser closes all TCP connections for the page when a blank one is loaded into the frame. Furthermore, the elements requested asynchronously via JavaScript may or may not be fully downloaded before the `onload` event. Thus, in our measurements, a single web site may produce different amount of traffic in different loads even if we assume that the loaded content is always the same, which is rarely true, especially for web sites employing AJAX. However, running the experiments multiple times should give us reasonable average results.

The Top Sites list from Alexa [Ale], retrieved on 6th of July, 2013, was used. Not all web sites could be used for the measurements. Firstly, some of them do not allow to be displayed into a frame, because they apply the "X-Frame-Options" header and modern browsers refuse to load the content. Secondly, a few sites utilize *framekiller* scripts[1]. Thirdly, some web sites redirect the user always to their HTTPS versions and therefore cannot be used together with the SPDY proxies. To conclude, we used all the sites among the first 100 in the list which do not belong to the mentioned problematic categories. The final list of 54 web sites is given in Appendix A.1.

While measuring, the benchmark page uses a timeout of 120 seconds. This is the maximal time allowed for a single page load. When it is reached, this page load is discarded from the results. Such timeout allows all of the tested pages to be fully loaded under normal conditions. However, during the experiments, it has been observed that particular servers – primarily serving Asian web sites – may stop sending data on some TCP streams without closing them. Thus, the timeout is applied to detect such situations.

For having comparable results, the Tor client is configured to employ always the same nodes for building circuits:

- Entry node: 109.105.109.162, ndnr1, located in Sweden
- Middle node: 38.229.70.61, Ramsgate, located in the USA

1 A framekiller is a piece of JavaScript code which forces browsers to open the same web page into the whole browser tab and therefore replace the one which includes the frame.

- Exit node: 131.159.15.91, spdytor2, located in Garching, Germany

The first two relays were chosen for being in different continents, and having *fast* and *stable* relay flags. The last one is operated by us and runs microspdy2http. The directives *EntryNodes* and *ExitNodes* are used in the Tor configuration file to set the first and the third node. For setting the middle relay, *ExcludeNodes* is applied to forbid everything from the IPv4 address space, but the addresses of the three relays. All needed configuration lines can be found in Appendix A.2.

Specific settings configured in the Tor Browser:

- HTTPS Everywhere is disabled, since we want to compare HTTP with SPDY.
- The memory cache in the browser is disabled for forcing the latter to request always all page elements. This is achieved by setting *browser.cache.memory.enable* to *false* into the configuration page of the Tor Browser.
- The HTTP pipelining is disabled for the HTTP proxy because mhd2spdy supports it, but does not benefit from it – it is better when the browser issues more connections. This is achieved by setting *network.http.proxy.pipelining* to *false* into the configuration page of the Tor Browser. Note that this does not affect the usage of pipelining when the browser communicates directly with the SOCKS proxy.

Two values are measured:

1. Page load time, using the JavaScript `onload` event.
2. Traffic within the Tor network while loading the same web sites. We measure particularly the number of IP datagrams and their overall size between the middle and the exit node by using *iptables* at the exit node (the script can be found in Appendix A.4).

Furthermore, SPDY headers compression is enabled in both requests and responses for experimenting how much the benefit would be for the Tor network. Note that sdpylay's code was modified to enable the compression for client frames.

More details showing the specific software versions involved in the measurements can be found in Table 5.2.

Tor Browser Bundle version	2.4.17-beta-2	includes Tor 2.4.17-rc
Tor version of all Tor routers used	2.4.17-rc	
libmicrohttpd version	SVN revision 30125	mhd2spdy and microspdy2http are part of it
spdylay version	1.0.0	used by mhd2spdy
curl version	7.26.0	used by microspdy2http
Benchmark page version	1.3, 2013-07-18	
Client's OS	Debian 7.0 64-bit	
Client's CPU	Intel Core 2 Duo, 2 * 2.0 GHz	
Client's memory	4 GB	
Client's Internet connection	High speed academic network	only IPv4
Client's physical location	Munich, Germany	
Period of the benchmarking	07 – 22 October, 2013	

Table 5.2: Software, hardware and other details for benchmarking SPDY and HTTP over Tor.

Note that the exit node used for benchmarking is configured with "MaxAdvertisedBandwidth 10 KB", which makes it not being selected by other Tor clients. In this way, we make sure that the relay is not under high CPU usage and more important, minimize the chance that there is other traffic between it and the middle relay.

For the experiments, the benchmark page was configured with three seconds interval between page loads, 120 seconds timeout and five iterations. The evaluation was done in the following way for all web sites in Appendix A.1:

1. Five random URLs are taken from the list and inserted in the benchmark page.
2. The configuration of HTTP proxy is cleared in the Tor Browser. The five web sites are loaded five times each to benchmark HTTP over Tor. At the same time the script is used on the exit node to measure the overall traffic for all 25 page loads.
3. The local mhd2spdy proxy is set as an HTTP proxy in the configuration of the Tor Browser. The same five web sites are loaded five times each to benchmark SPDY over Tor. At the same time the script is used on the exit node to measure the overall traffic for all 25 page loads.
4. Point 2. is repeated.
5. Point 3. is repeated.

6. If a single page load does not succeed – timeout occurs – the benchmarking of all five web sites starts from the beginning.

Finally, we have ten page load times for each web site for HTTP and for SPDY. All 20 loads of a single site happen within 20-30 minutes. That is, these 20 values are comparable to each other. Some short temporary increases of the latency within the Tor network or on the way to the requested web site would appear in the results as rare outliers.

For investigating how the possible latency improvements are related to the possible traffic reduction, we measured the first 12 web sites from Appendix A.1 individually. That is, the same steps mentioned above were followed, but a single web site was loaded each time. Thus, the script measuring the traffic gave results for a single web site. Note that the interval in the benchmark page was set to 50 seconds. Hence, all values for a single web site were collected within 15-20 minutes.

5.2.2 *Results*

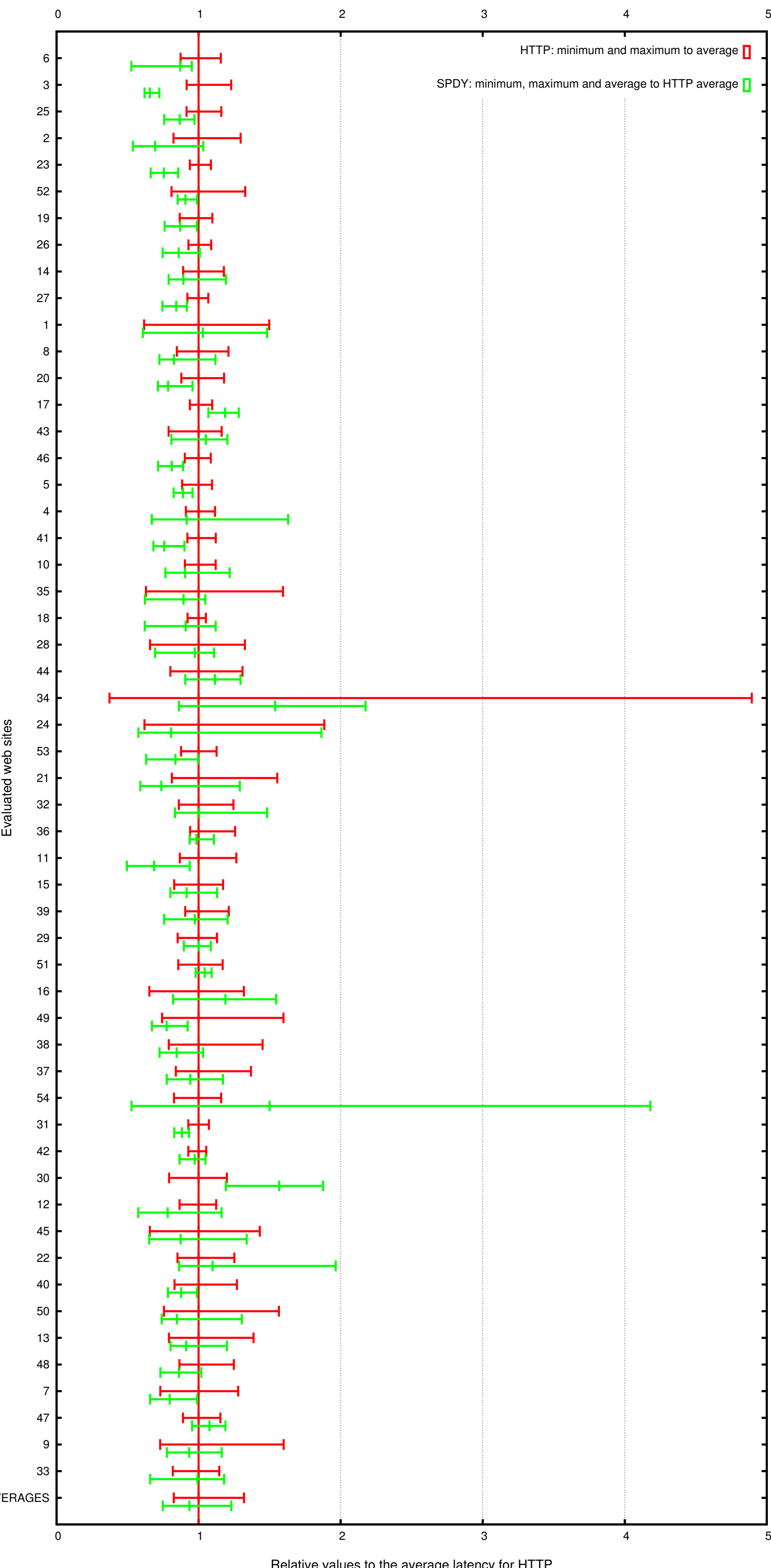


Figure 5.26: Relative page load times for SPDY over Tor compared to HTTP over Tor. The evaluated web sites are ordered by the absolute average time for HTTP.

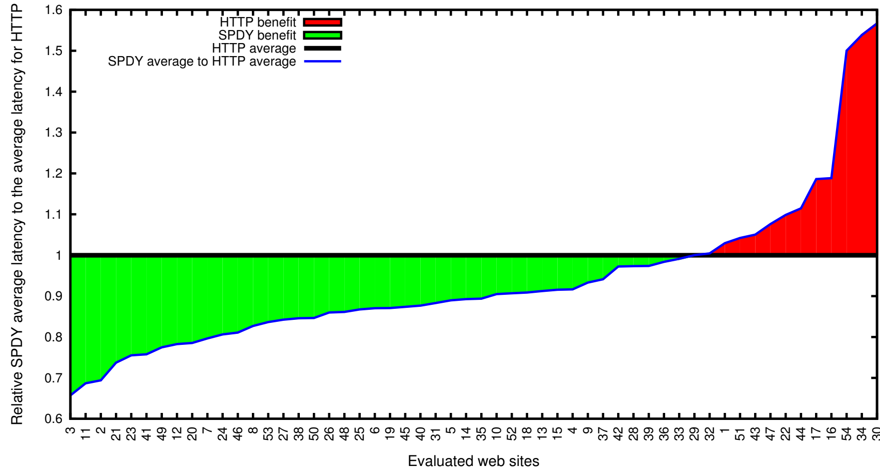


Figure 5.27: All evaluated web sites ordered from the one with greatest benefit from SPDY to the one with least.

	Average	Minimum	Median	Maximum
Page load time with HTTP (ms)	19,342.9	15,757.2	18,818.6	25,682.6
Page load time with SPDY (ms)	18,504.6	14,709.4	18,032.2	24,712.9
Difference HTTP – SPDY (ms)	838.3	1,047.8	786.4	969.7
Ratio SPDY / HTTP	0.957	0.934	0.958	0.962
Ratio SPDY / HTTP (averages of the ratio for each site)	0.935	0.928	0.952	0.970

Table 5.3: Page load time per web site for HTTP and SPDY over Tor.

	HTTP	SPDY	Difference	Ratio
Number of input packets	536	499	37	0.931
Number of output packets	361	310	51	0.859
Total number of packets	897	809	88	0.902
Input bytes	137,941	107,374	30,567	0.778
Output bytes	1,163,823	1,092,012	71,811	0.938
Total bytes	1,301,764	1,199,386	102,378	0.921

Table 5.4: Average traffic per web site. It was measured at the exit node while benchmarking the page load time. *Input* is the traffic from the middle relay to the exit node; *output* is the one from the exit node to the middle relay.

The same measurements were performed a week before the week presented here. The average results were almost the same. However, there were significant differences for some of the web sites.

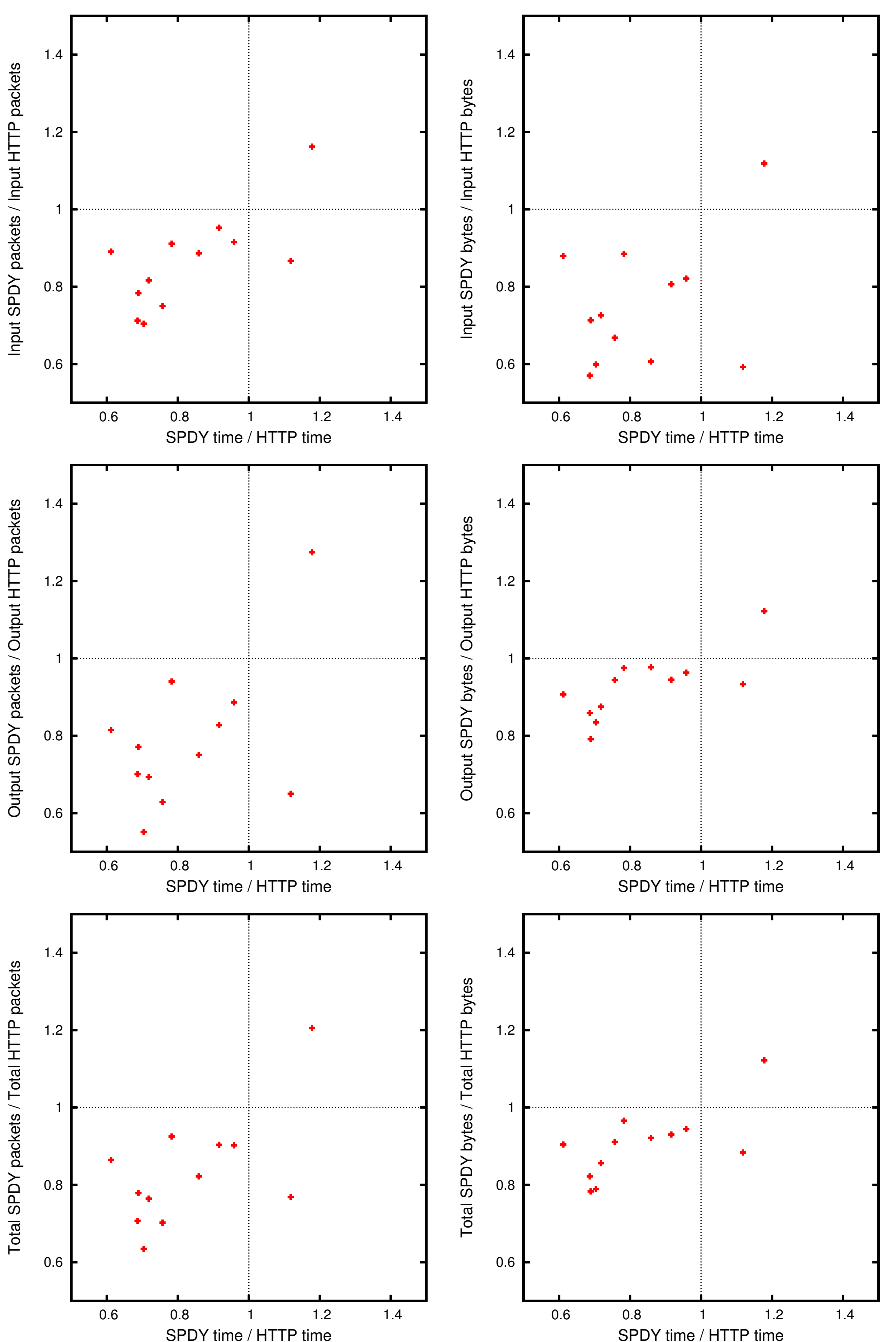


Figure 5.28: Relation between traffic benefit and time benefit for the first 12 web sites from the list of evaluated.

5.3 EVALUATION OF THE PROPOSED ALGORITHM FOR SPDY SERVER PUSH

For investigating if the proposed algorithm can be useful in different practical scenarios, we evaluated it against different types of log files. We used web server access log files to prove if the algorithm is suitable for SPDY reverse proxies and SPDY servers as well.

Nevertheless, for privacy reasons we did not try to collect log information about the exit traffic of Tor exit nodes. Instead, we utilized logs of forward

proxies run on volunteer's client machines, as well as log data about HTTP traffic passing an Internet Service Provider (ISP).

To evaluate the algorithm, we tried to determine the set of input parameters and push rate values for which the cost function has minimal value. We decided to use $\omega = 0.5$ as a parameter for the cost function. That is, a single prediction mistake offsets the benefit of a single hit.

For the purpose of the evaluations, users were uniquely identified by the source identifier in the log files and the user agent used. We are aware that this method is not perfect as different users using the same user agent may share the same global IP address due to Network Address Translation (NAT).

ID	Name	Values
P1	Algorithm used	B = basic L = basic-less M = basic-more
P2	Maximum asset size for push	-1 1,000 10,000
P3	No push for mobile devices and Consider browsers	F = false A = ignore mobiles B = ignore mobiles and consider browsers
P4	Keep history	F = false T = true
P5	Session timeout	180 s 3,600 s
P6	Probability threshold	from 0 to 1 with step of 0.1

Table 5.5: Evaluated input parameters.

The algorithm was run for all possible combinations of the parameters listed in Table 5.5. When P3 had value B, we used different statistics depending only on the browser name (e.g. *Firefox*). That is, browser version, platform, and so forth were not considered.

For P4, we simply remembered all requests for each user and applied this information in future sessions of the same user. It should be mentioned that this is not absolutely precise – the browser might not always keep the requested resources in the cache.

P5 may be limited by clients as well, but here we want to show how a longer session affects the algorithm compared to a shorter one.

The input parameters in Table 5.6 stayed constant for all runs.

For choosing reasonable values for C1 and C2, preliminary tests with part of the data from Subsection 5.3.1 were performed. The number of resources considered by the algorithm as both page load and asset was compared for different values of C1 and C2. The least confusions were seen when C2 is either 1s or 2s with slightly better results for 1s. Values between 15s and 30s for C1 gave almost the same results.

Preliminary tests with the same data were performed for C3 and C4 as well. Increasing C3 may significantly reduce the number of pushed resources. However, applying larger values for C3 did not seem to reduce the

ID	Name	Value
C1	Maximum page load timeout	20s
C2	Page load timeout	1s
C3	Minimum history	1
C4	Maximum pushed assets	-1
C5	Assume pushes as hits	Having the log files, we explicitly know what is a push hit and what a mistake

Table 5.6: Constant input parameters.

proportion of mistakes compared to total pushes. On the other hand, C4 affects results in a different way depending on the input data – the number of assets per page load. It should be further investigated how C4 can be selected to be beneficial for the algorithm.

Thus, we chose to use default values for C3 and C4. That is, both parameters are ignored.

We are interested in answering the the following questions:

- For what kind of input data the algorithm gives best results and for which combination of input parameters?
- Which combination of input parameters gives the best results over the entire date set?
- How much do individual parameters affect the result?

For the evaluation purposes, we count the push hits and mistakes that would have been generated by the algorithm. A push is considered as a hit if the same URL is requested later in the same session. It should be mentioned that a hit is counted even if the resource is requested in a different page load after the one in which it has been pushed. After counting the hit, the push contributes for updating the statistics in the global maps (Algorithm 1) as all other requests in the log file. On the other hand, all pushes whose URLs have not been subsequently requested in the session are counted as mistakes when the session is considered as closed. They do not affect the maps in any way.

5.3.1 *Using Web Server Access Log Files*

Apache 2 access log files in Combined Log Format [Apa] were used for this evaluation. Preprocessing was required before passing the log files to the algorithm. The entries in the files are not expected to be in chronological order: there are race conditions between threads which write to the files. Furthermore, the precision of the time field in the logs is in seconds and hence, it is not enough for correctly sorting the entries. Thus, we used the *Referer* entry to reorder lines having the same values for *remote host*, *time* and *User-Agent*: A line with a requested resource which is Referer for other resources within the same set has to be placed before them; Lines with Referer which is not in the set have to be placed at the beginning of the set. The exact order of assets of the same page having same time value cannot be determined, but this is not important for the proposed algorithm. However, inaccuracies are possible even after the reordering. For instance, some

clients may not include Referer header in requests, multiple page loads may happen within the same second, and so forth.

For checking if a request is made by a known browser, the list provided by Browser Capabilities Project [Bro] was used (version 5020 of *full_php_browscap.ini* from the 29 July, 2013).

Since we did not have access to variety of log files, the available logs for the web sites in Table 5.7 was separated by days. That is, each time the algorithm received a single day sample for one of the web sites as input data.

ID	Type	Estimated number of pages[2]	Average number of log lines per day[3]	Number of log days
Site1	Home page of an open source project	239,000	122,374	48
Site2	Informational movie database	40,000	8,680	101
Site3	Personal web site	418	4,846	210
Site4	Financial portal	159,000	135,631	16
Site5	Older version of Site4	Not available	33,655	3

Table 5.7: Web sites whose access log files were evaluated for SPDY server push.

5.3.2 *Using ISP Log Data*

Since the access log evaluation cannot answer if SPDY server push would be suitable for Tor – with Tor, the SPDY proxy works as a forward one – we evaluate also ISP log data. For achieving this evaluation, the web traffic on a high speed link of a large German ISP was considered. The data for this research is based on the same raw data that was collected for [Gro13]. For exact details on the collection method we refer to [Gro13, 2. Data Collection]. For our evaluation, five data sets with about 500,000 HTTP requests over a period of 31 hours were collected. For each request, the following information was recorded:

- A unique identifier of the requested resource.
- A unique identifier of user's session. That is, a number distinguishing all requests made by a single source IP address within certain time (300 seconds).

2 The estimated number is given by Google on the top of the search results when putting the domain name of each web site to this URL: https://www.google.com/search?hl=en&q=site:example.com . Note that it may be inaccurate.

3 Note that this value indicates the raw content of the log files before any filtering was applied. That is it includes all kind of requests issued by robots. Moreover, some of the web servers have Subversion running on them and therefore its activity is also included in the given value.

- Time of the request, in milliseconds.
- "User-Agent" string from the request.

The differences compared to the access log files are:

- The exact order of the requests is known.
- There is no correlation between requests and responses. Thus, the response status code and size are unknown. We assume status *200* for all responses and ignore the size.
- The session length is limited by an absolute value. Thus, during the evaluations we can only split it in several sessions if P5 is less than 300s.
- There is no correlation between different sessions of the same user – exactly as it is in Tor – unless if a 300s-long session is split into multiple ones.
- There is no information about the HTTP method. Thus, we assume that it is always *GET*.

We expect that the collected data is very close to what a Tor exit node sees and therefore, it is suitable to evaluate if SPDY server push is appropriate to be used with the SPDY proxies and Tor.

Before being given to the algorithms, the data was converted to Combined Log Format [Apa]. That is, it was modified to access log data with empty *Referer*, *-1* for response size and source ID for *remote host*.

Each sample is given to the algorithm separately from the others. That is, an ISP data sample is evaluated as a single day sample in Subsection 5.3.1.

5.3.3 *Using Forward Proxies Log Data*

We evaluated the algorithm with log files from local forward proxy utilized on the client's machine as well. We believe that the data of a forward proxy is similar to that one collected on the ISP and hence, it is suitable for evaluating SPDY server push for Tor.

Three users volunteered to run Privoxy [Pri] and configure their browsers to use it for an HTTP proxy (see Table 5.8). The default Privoxy configuration was used with the addition of "debug 512". This option makes the proxy write log data in Common Log Format [Apa]. Thus, we collect the same data as in access logs, but without *Referer* and *User-Agent*. However, reordering is not needed – the lines are believed to be in chronological order – and the browser is always the same.

In contrast to the access logs, all requests are made from a valid browser, but there are multiple lines which interfere with the way we assume what is a page, a page load and an asset (see Section 3.2). The following requests issued by the browser are not related to the page loads:

- Suggestions (suggest queries) when users start writing in the address bar or another search bar in their browsers.
- Safe browsing (site check) requests to possibly check if a visited web site is malicious.
- OCSP and CRL requests for checking the validity of a certificate employed by a TLS web site.

ID	Used browser	Average number of log lines per day	Number of log days	Note
Proxy1	Opera	6,962	36	
Proxy2	Firefox	67,900	18	The user kept a huge amount of loaded web pages / opened browser tabs in each moment.
Proxy3	Chrome	1,414	6	The proxy was set for the whole system. Thus, there was non-browser traffic which was not filtered.

Table 5.8: Proxy users whose log files were evaluated for SPDY server push.

- Browser plug-ins may as well issue any kind of requests at any moment.

For trying to filter some of the mentioned requests, as well as too long lines truncated by Privoxy, the script in Appendix B.1 was used.

Each time, the algorithm received a single day sample of one of the data sets (one of the proxy users) as input data.

5.3.4 *Results*

Here, we present the evaluations for all of the log file types. In the results, requests from unknown browsers, as well as requests for "favicon.ico" are not considered. That is, such requests affect only the given numbers of log lines.

Table 5.9 to Table 5.12 and Figure 5.29 to Figure 5.37 illustrate the single day sample per data set and the input parameters for which the algorithm achieved the best results (the least value of the cost function). In the figures, the push rate is a function of P6. That is, all points correspond to a certain value of P6, while all other input parameters are those for which the algorithm gave the best result (Table 5.9).

	P1	P2	P3	P4	P5	P6
Site1	B	-1	F	T	3,600	0.3
Site2	M	-1	F	F	3,600	0.5
Site3	B	-1	B	T	3,600	0.5
Site4	L	-1	F	F	3,600	0.5
Site5	B	-1	A	F	3,600	0.5
Isp	L	-1	F	F	3,600	0.4
Proxy1	M	-1	F	F	3,600	0.6
Proxy2	B	-1	F	F	3,600	1
Proxy3	B	-1	F	F	180	0.5

Table 5.9: The combination of input parameters which gave the best single result per data set.

	Pushes	Hits	Mistakes	PushCost
Site1	1.076918	0.956006	0.120912	0.582453
Site2	0.522088	0.47147	0.050618	0.789574
Site3	0.862434	0.746032	0.116402	0.685185
Site4	0.085871	0.058962	0.02691	0.983974
Site5	0.319083	0.222607	0.096476	0.936934
Isp	0.038079	0.024998	0.01308	0.994041
Proxy1	0.097733	0.066942	0.03079	0.981924
Proxy2	0.000779	0.000779	0	0.999611
Proxy3	0.005669	0.004535	0.001134	0.998299

Table 5.10: The best achieved result per data set. Pushes, hits and mistakes are normalized to the number of assets.

	Log lines	Sessions	Average session length (s)	Average page loads per session	Average assets per page load
Site1	1,235,889	43,974	173.17	2.78	7.51
Site2	12,481	653	116.45	1.92	6.07
Site3	2,093	637	39.39	1.14	0.26
Site4	143,311	8,431	384.18	2.19	1.22
Site5	25,533	380	498.77	10.32	1.32
Isp	495,077	10,680	112.00	5.31	4.95
Proxy1	7,733	6	4,289.67	63.17	17.22
Proxy2	14,464	2	24,602.00	1,684.00	2.67
Proxy3	1,056	17	131.47	8.35	6.21

Table 5.11: Characteristics of the single day sample which gave the best result per data set.

	% of unique pages[4]	% of unique assets[5]	% of delayed hits[6]	Average mistake body size
Site1	0.54	0.08	3.03	54,209.87
Site2	52.67	13.00	0.78	14,518.64
Site3	8.83	8.47	0.00	27,650.86
Site4	19.71	25.49	1.73	7,877.17
Site5	7.09	13.65	3.72	19,319.53
Isp	68.97	65.40	2.90	0.00
Proxy1	87.86	61.24	7.55	13,650.61
Proxy2	82.84	50.11	0.00	0.00
Proxy3	67.61	89.68	25.00	0.00

Table 5.12: Characteristics and algorithm results of the single day sample which gave the best result per data set.

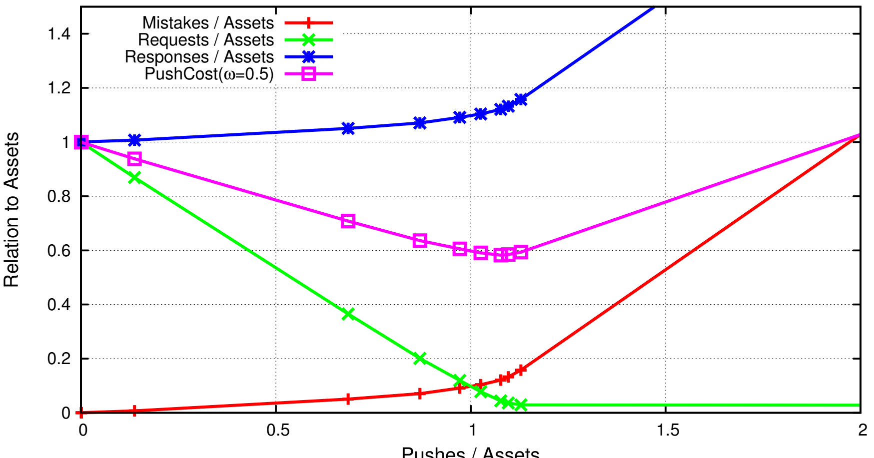


Figure 5.29: The best single result for Site1.

4 The proportion of pages with unique URL. All others pages are requests for URLs which have been already seen.

5 The proportion of assets with unique URL. All others assets are requests for URLs which have been already seen.

6 Proportion of hits which has been pushed in one page load, but are needed in a subsequent one.

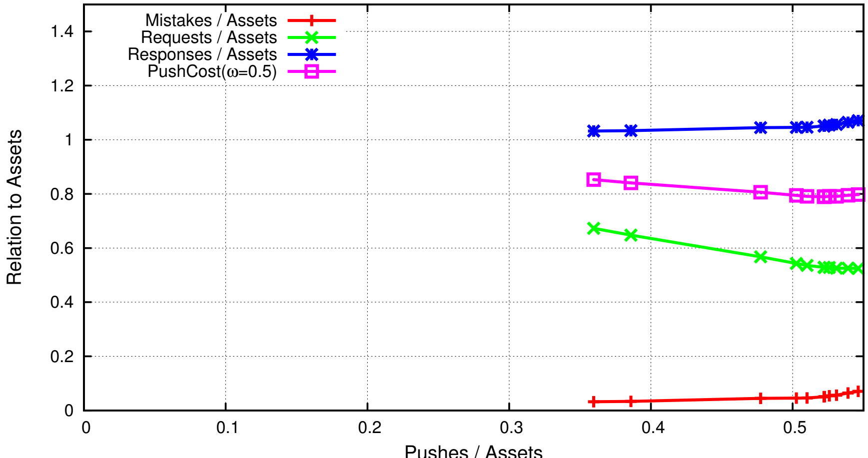


Figure 5.30: The best single result for Site2.

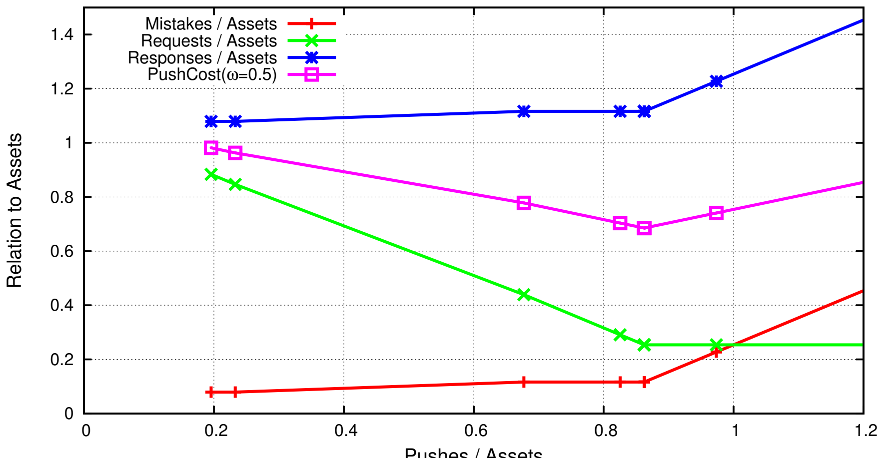


Figure 5.31: The best single result for Site3.

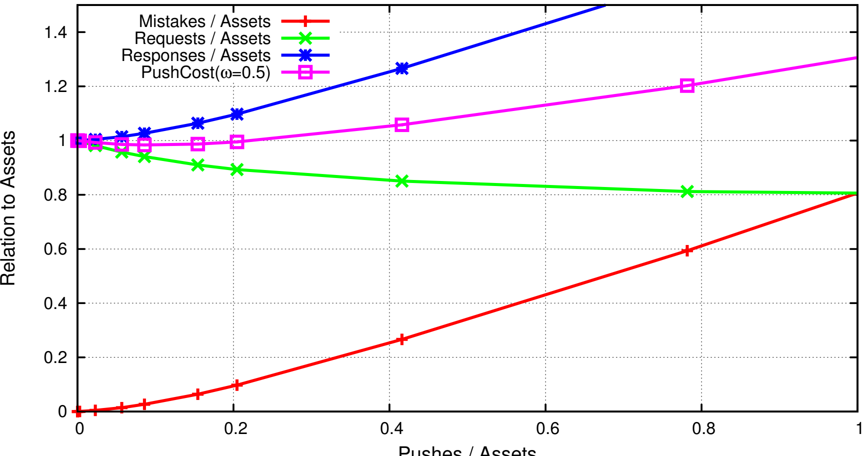


Figure 5.32: The best single result for Site4.

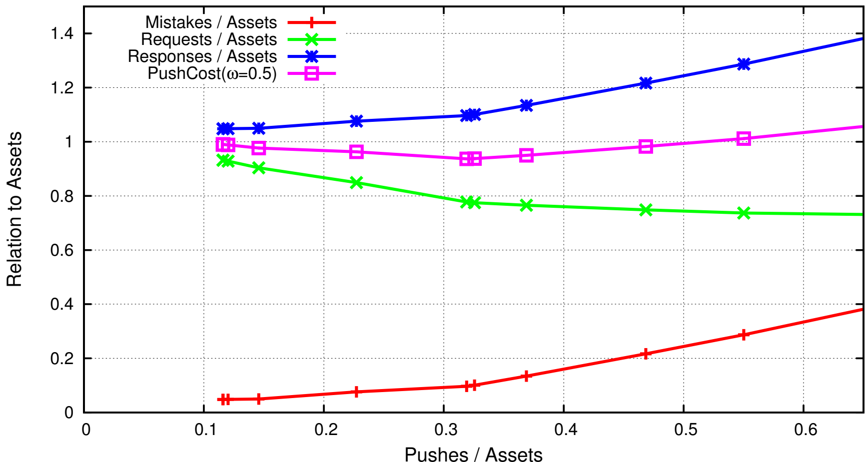


Figure 5.33: The best single result for Site5.

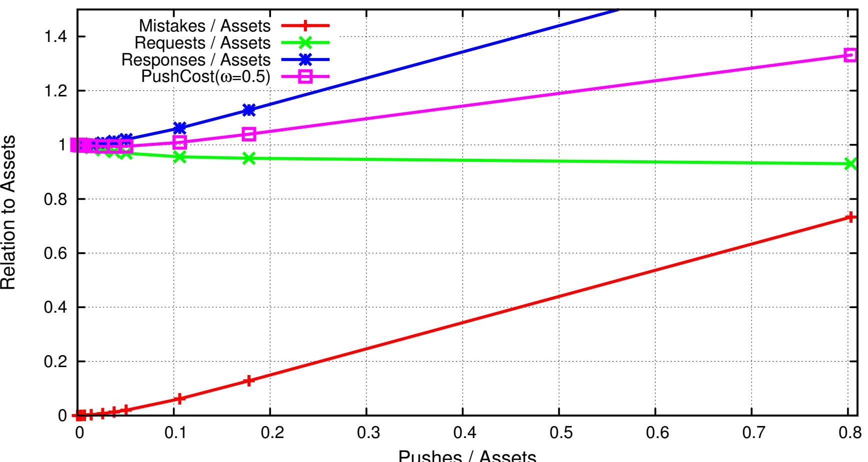


Figure 5.34: The best single result for Isp.

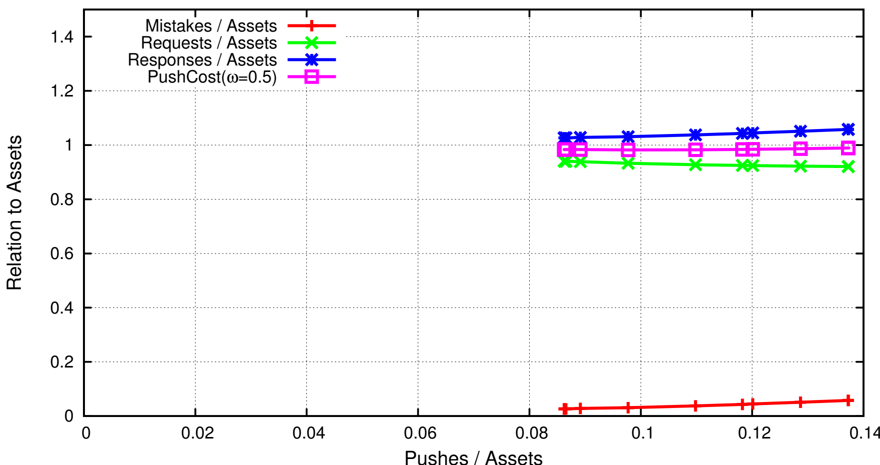


Figure 5.35: The best single result for Proxy1.

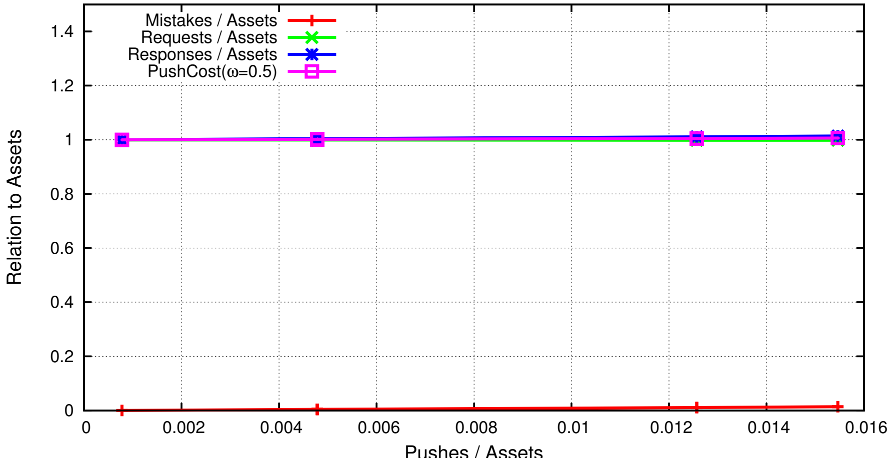


Figure 5.36: The best single result for Proxy2.

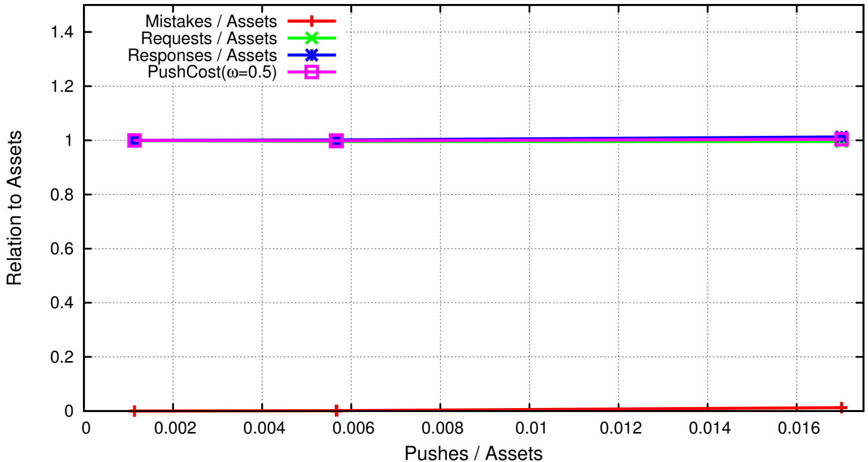


Figure 5.37: The best single result for Proxy3.

The best combination of input parameters per data set was investigated. This is the combination which gave the best average results over all single day samples per data set (see Table 5.13). The best combination for each data set is compared to all other sets in Figure 5.38 to Figure 5.46.

	P1	P2	P3	P4	P5	P6
Site1	B	-1	B	F	3,600	0.4
Site2	B	-1	F	T	3,600	0.2
Site3	B	-1	B	F	180	0.6
Site4	L	-1	F	F	180	0.5
Site5	L	-1	A	F	3,600	0.4
Isp	L	-1	F	F	180	0.4
Proxy1	B	1,000	F	F	180	0.6
Proxy2	B	1,000	F	F	180	1
Proxy3	L	-1	F	F	180	0.2

Table 5.13: The combination of input parameters which gave the best average result per data set.

	Pushes	Hits	Mistakes	PushCost
Site1	0.526579	0.355154	0.171425	0.908135
Site2	0.348127	0.278886	0.069241	0.895178
Site3	0.214367	0.139239	0.075128	0.967944
Site4	0.031365	0.020755	0.01061	0.994927
Site5	0.089608	0.062984	0.026624	0.98182
Isp	0.024772	0.016626	0.008146	0.99576
Proxy1	0.018876	0.014342	0.004534	0.995096
Proxy2	0.000469	0.0004	0.00007	0.999835
Proxy3	0.001117	0.000756	0.000361	0.999803

Table 5.14: Average results for the parameters in Table 5.13. Pushes, hits and mistakes are normalized to the number of assets.

	Average sessions	Average session length (s)	Average page loads per session	Average assets per page load
Site1	3,031.48	397.66	3.66	4.38
Site2	365.67	140.53	1.99	4.77
Site3	1,162.77	10.97	1.70	0.07
Site4	7,369.19	19.23	1.78	2.92
Site5	483.00	546.84	9.40	1.85
Isp	8,179.20	111.77	5.82	6.00
Proxy1	22.31	503.77	49.96	4.51
Proxy2	3.33	19,166.06	748.70	12.80
Proxy3	9.33	156.39	9.89	13.06

Table 5.15: Characteristics of each data set when the parameters in Table 5.13 are used.

	% of unique pages[7]	% of unique assets[8]	% of delayed hits[9]	Average mistake body size
Site1	7.22	2.14	1.30	39,268.43
Site2	56.44	20.16	1.16	14,221.67
Site3	4.00	32.07	2.42	61,015.98
Site4	35.77	63.41	4.05	15,426.77
Site5	8.25	13.43	2.86	37,383.32
Isp	80.81	67.92	3.59	0.00
Proxy1	73.70	68.72	2.01	434.34
Proxy2	89.53	19.35	44.58	7,232.40
Proxy3	74.55	96.31	0.00	0.00

Table 5.16: Characteristics and algorithm results of each data set when the parameters in Table 5.13 are used.

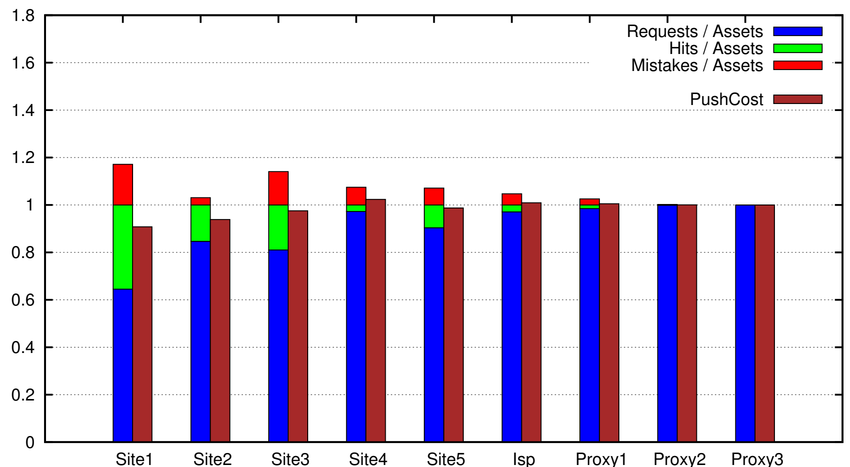


Figure 5.38: Average results of the best combination of input parameters for Site1 applied for all data sets.

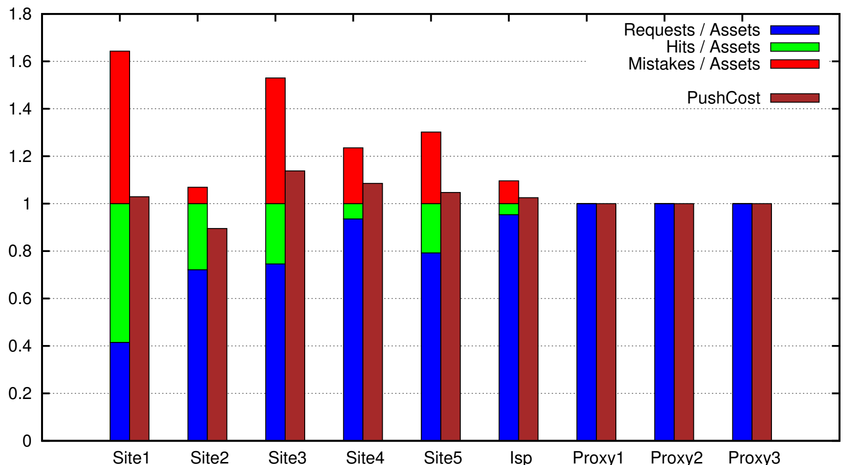


Figure 5.39: Average results of the best combination of input parameters for Site2 applied for all data sets.

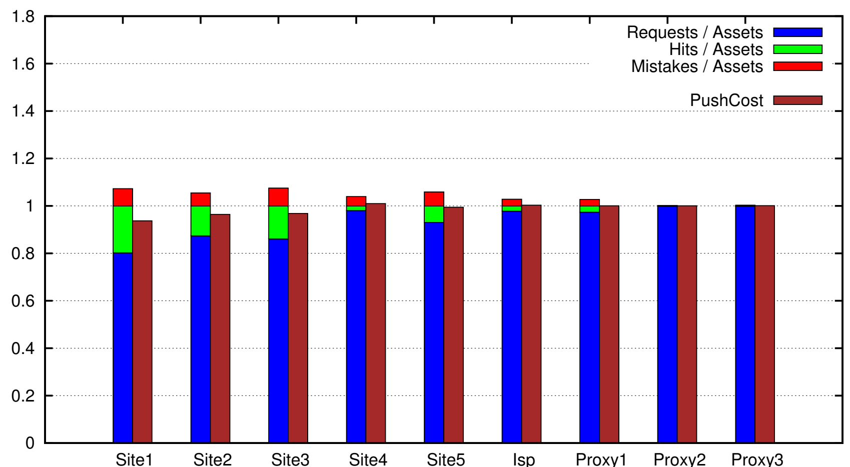


Figure 5.40: Average results of the best combination of input parameters for Site3 applied for all data sets.

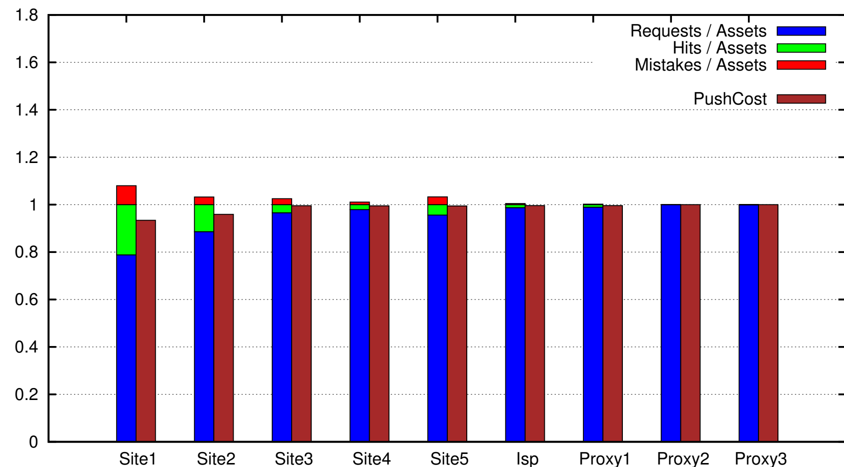


Figure 5.41: Average results of the best combination of input parameters for Site4 applied for all data sets.

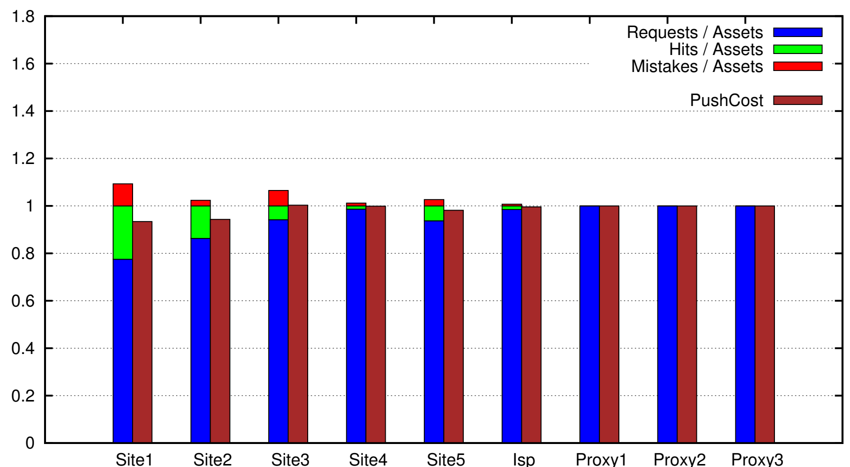


Figure 5.42: Average results of the best combination of input parameters for Site5 applied for all data sets.

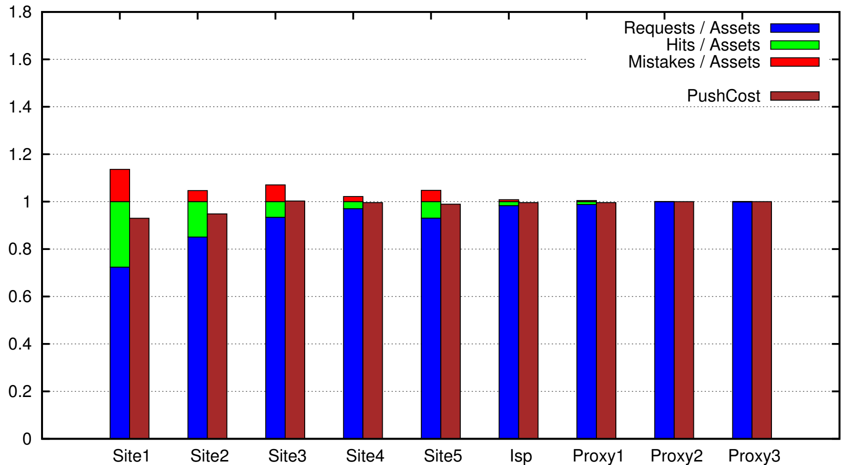


Figure 5.43: Average results of the best combination of input parameters for Isp applied for all data sets.

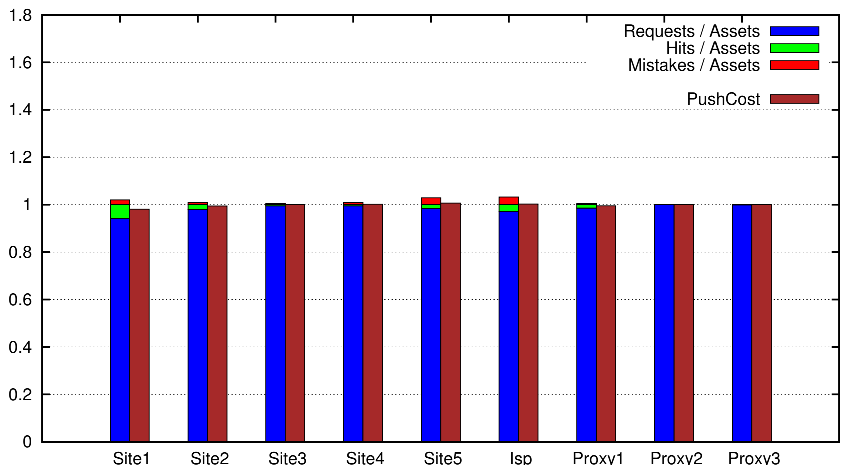


Figure 5.44: Average results of the best combination of input parameters for Proxy1 applied for all data sets.

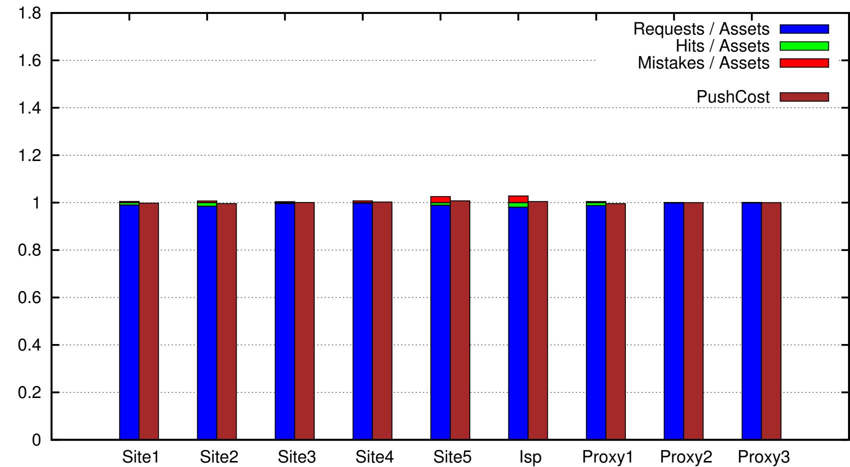


Figure 5.45: Average results of the best combination of input parameters for Proxy2 applied for all data sets.

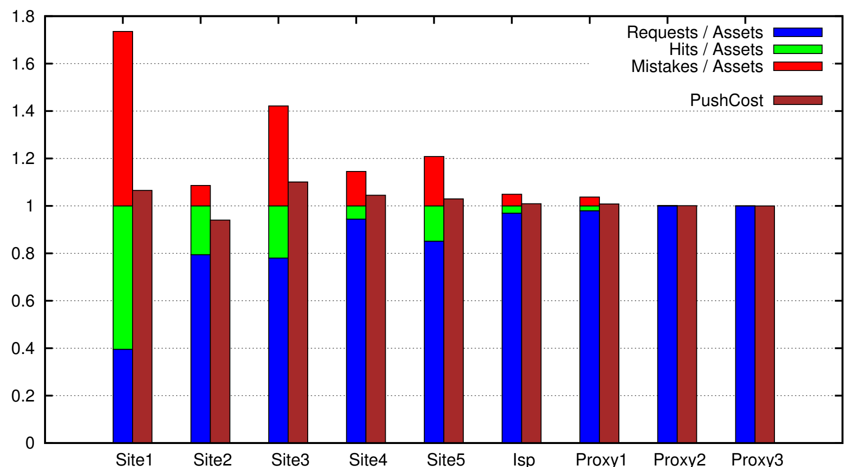


Figure 5.46: Average results of the best combination of input parameters for Proxy3 applied for all data sets.

On Figure 5.47 to Figure 5.55, it can be seen how a change to a single input parameter affects the average result (the cost function) of the best combination of parameters per data set. That is, the combinations in Table 5.13 and the cost function in Table 5.14 are the reference. Note that the case when P6 is zero is not presented as it usually increases the cost function drastically compared to all other changes.

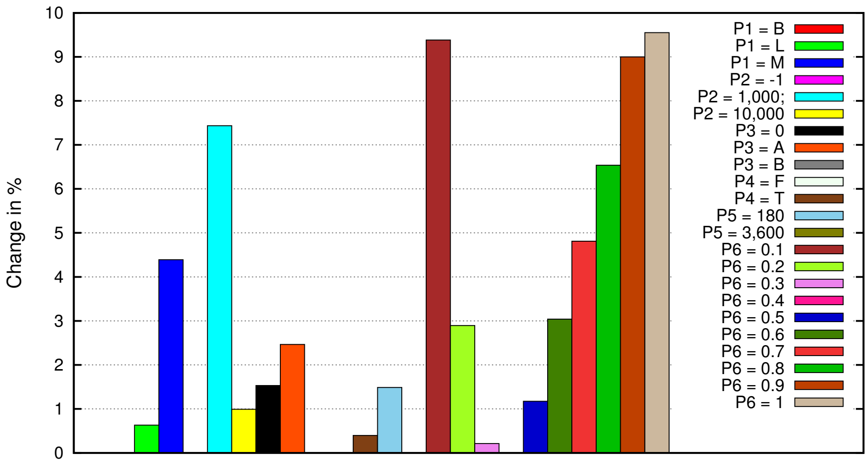


Figure 5.47: Increase of the average value of the cost function for Site1 when a single parameter is changed.

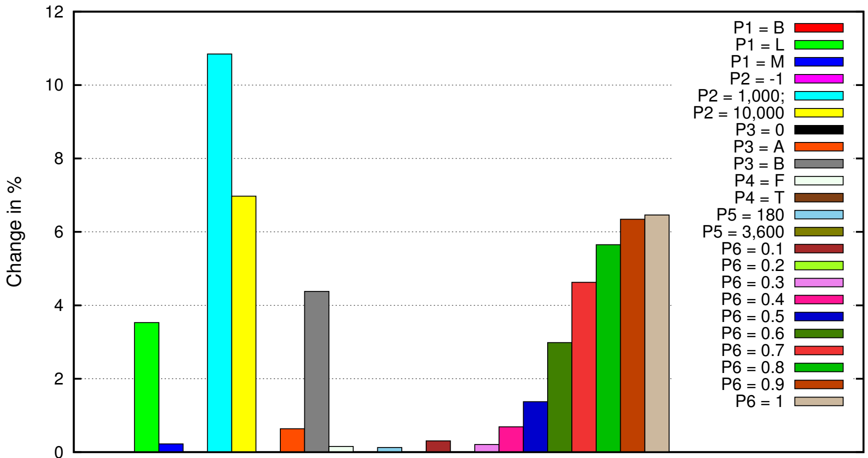


Figure 5.48: Increase of the average value of the cost function for Site2 when a single parameter is changed.

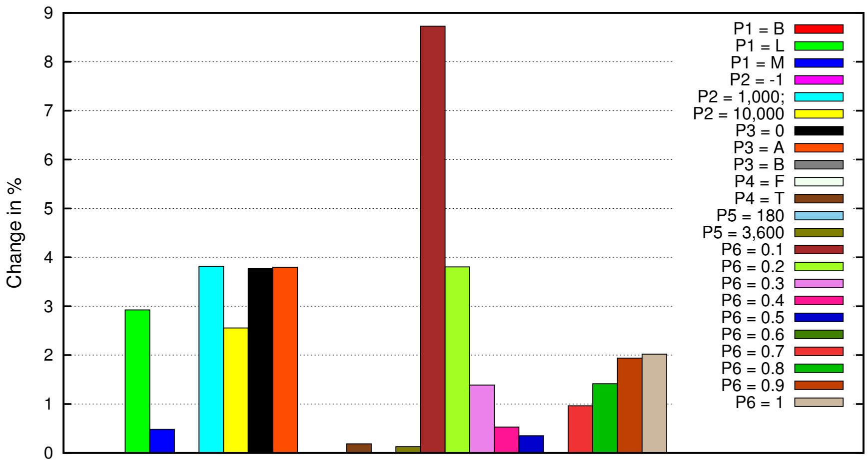


Figure 5.49: Increase of the average value of the cost function for Site3 when a single parameter is changed.

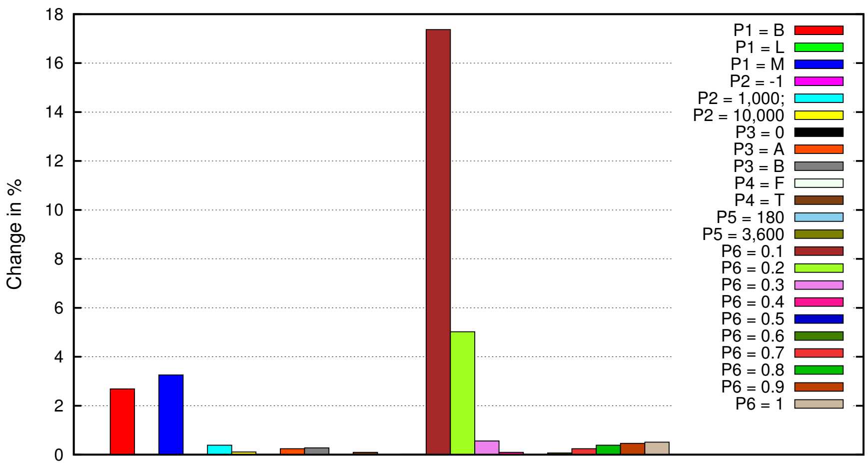


Figure 5.50: Increase of the average value of the cost function for Site4 when a single parameter is changed.

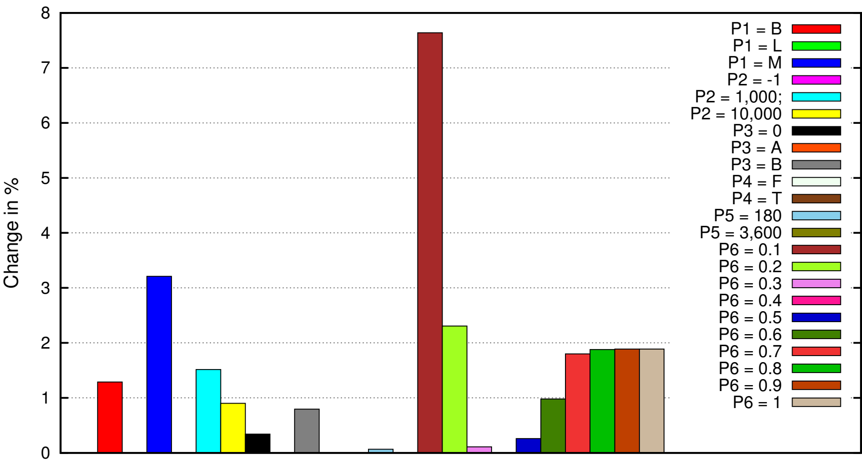


Figure 5.51: Increase of the average value of the cost function for Site5 when a single parameter is changed.

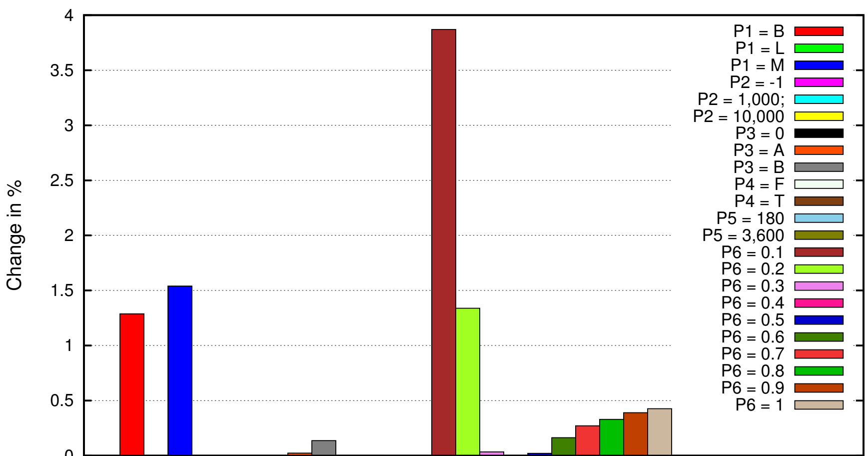


Figure 5.52: Increase of the average value of the cost function for Isp when a single parameter is changed. A change in P2 is not applicable.

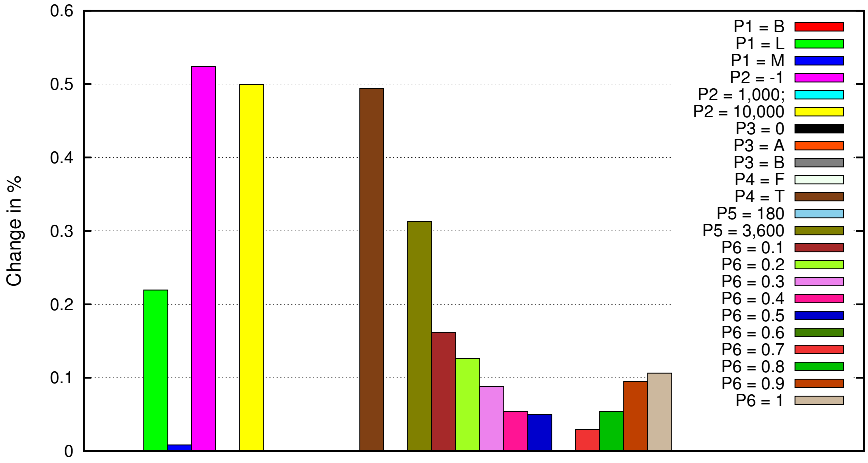


Figure 5.53: Increase of the average value of the cost function for Proxy1 when a single parameter is changed. A change in P3 is not applicable.

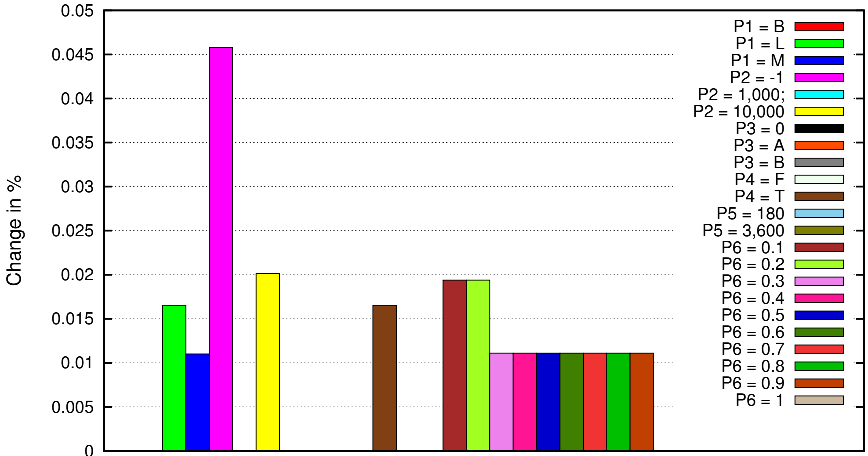


Figure 5.54: Increase of the average value of the cost function for Proxy2 when a single parameter is changed. A change in P3 is not applicable.

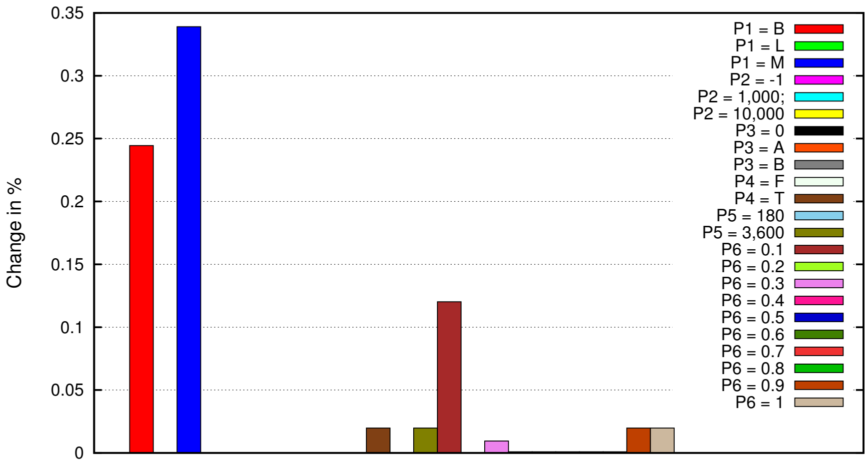


Figure 5.55: Increase of the average value of the cost function for Proxy3 when a single parameter is changed. A change in P3 is not applicable.

5.4 EVALUATION OF HTTP AND SPDY HEADERS AND BODY SIZE

For estimating the possible benefit and cost of SPDY server push in terms of traffic, headers and body sizes have to be considered. For this purpose, the same 54 web sites from Appendix A.1 were benchmarked. Since they are among the most popular web sites on Internet, they might be somewhat atypical as they are likely better optimized for latency and bandwidth consumption than the average website. Thus, improving performance for these popular websites is both harder, and – due to their popularity – more significant in its impact.

The benchmark page from Appendix A.3 was used to load all web sites from the list once. Note that the same limitations exist as explained in Subsection 5.2.1 with regard to the JavaScript `onload` event.

The measurements happened in Firefox and the data was collected using Firebug [Fir] and its extension NetExport [Net]. The benchmarking was performed on 13th of August, 2013. Note that the results include the pages; that is, the HTML resources.

	Average	Minimum	Median	Maximum
Request header size (B)	497	268	384	3,789
Response header size (B)	355	51	338	1,290
Response body size (B)	15,319	0	4,162	16,827,709

Table 5.17: Average, minimum, median and maximum values for request, response headers and response body per web resource. All resources are considered expect those whose responses were not yet received at the time when the `onload` event was triggered.

	Average	Minimum	Median	Maximum
Request header size (B)	490	268	380	3,789
Response header size (B)	354	131	335	1,290
Response body size (B)	11,281	0	4,357	613,129

Table 5.18: Average, minimum, median and maximum values for request, response headers and response body per web resource. Only *GET* requests received response with status *200* are considered.

Table 5.18 lists the results only for such resources which are allowed to be pushed by the proposed algorithm. We are interested in the median values since the average ones are affected by some outliers. However, these are the row values. Assuming that headers are compressed, the values should be smaller. Since it is not yet sure which compressing algorithm would eventually be used with SPDY, we estimate the compression ratios by using the results achieved by gzip in [Gro13, Table 5]. That is, the compressed request headers have a size of 14 % of the original one, while the compressed response headers are expected to be compressed to 18 % of the original size. These compression ratios are used in Table 5.19, where we try to estimate the traffic benefit from a push hit – this would prevent issuing a request by the client – and the cost of a push mistake – one unneeded response would be sent.

	SPDY frame header	SPDY frame payload	Total
Benefit from a push hit (B)	1 * 18[10]	53	71
Cost for a mistake (only the headers) (B)	2 * 12[11]	60	84
Cost for a mistake (only the body) (B)	16[12]	4,357	4,373
Cost for a mistake (total) (B)	40	4,417	4,457

Table 5.19: Estimations of traffic benefit and cost from using SPDY server push.

The obvious conclusion that might be made from the estimations is that the benefit and only the headers of the cost are about the same. In contrast, the body part of a mistake costs about 62 as much bandwidth, as the bandwidth benefit of a possible hit. When the RTT from a client to a server is large, one can expect that the full body of a mistake is already on the wire at the time the server might receive a *RST_STREAM* for canceling the push. For such scenarios, we can assume that the cost of a mistake includes the body size of the resource. This means that even if push canceling is used in Tor, it might not help to prevent the high amount of additional traffic generated by possible mistakes. However, just as the bandwidth costs are likely more dramatic for Tor due to the higher delay for cancellations, the reduction in latency would also be more significant.

10 A single *SYN_STREAM* frame is assumed.
11 One *SYN_REPLY* and one *HEADERS* frame are assumed. The client may send *RST_STREAM*, which is 16 B more.
12 A single *DATA* frame is assumed. Depending on the server implementation, multiple frames might be used for the same body size.

Considering the cost function for push (Equation 3.6), ω should have a value of about $\frac{1}{63}$ in scenarios when additional bandwidth is not tolerated; that is, the possible latency improvements are ignored. It should be noted that the heuristics for SPDY push which were described in this thesis do not perform well at such a small value for ω.

One possible alternative is to consider the case where only small resources are pushed. This might decrease the push rate (depending on the site), but would significantly decrease the cost of mistakes.

6 DISCUSSION

We start this chapter with discussion about alternative designs of involving SPDY with the Tor network. Next, the results presenting the benefits for Tor from SPDY are discussed. Furthermore, explanation of security concerns of SPDY and their possible mitigations is presented. Lastly, discussion about the evaluation of the proposed algorithm for push predictions, as well as notes about its implementation, is given.

6.1 DESIGN ALTERNATIVES FOR USING SPDY WITH TOR

6.1.1 *Alternatives for Using Proxies*

There are several different alternatives how SPDY can be used with Tor when the target web server supports only HTTP.

The SPDY client may be implemented:

- In the browser. We believe that the best solution in terms of performance is when the Tor Browser directly speaks SPDY. However, as of 2013, this cannot be done easily, because of two reasons. Firstly, Firefox is not known to support SPDY without TLS and therefore, cannot take advantage of a SPDY to HTTP transparent proxy. Secondly, Firefox is not known to support SPDY proxy natively and therefore, cannot employ a SPDY to HTTP forward proxy. In contrast, Google Chrome and Opera Browser have this feature. Such an ability may be used for improving performance in high latency networks, for instance in mobile networks. Therefore, it can be expected that more browsers will support SPDY proxy sooner or later.
- As an HTTP to SPDY proxy. This is the solution we used in this thesis, which has the advantage of not requiring invasive modifications to the other components.
- In the Tor client. While implementing SPDY in the Tor client has some usability benefits, adding significant additional logic to the Tor process would require an extensive security audit and still increase the trusted code base. With external processes handling SPDY, it becomes easier to isolate (jail) this code and thus limit the impact of possible vulnerabilities in it. Hence, we believe that adding SPDY support to the Tor client itself is not ideal.

The translation from SPDY to HTTP can happen:

- In the Tor process on the exit node. Here, again the same usability vs. security considerations apply.
- In a proxy on the exit node. This is what we implemented and measured and we believe that this is a good design for the future.
- In a proxy on another machine. This case involves a third party which has to be trusted and is likely to unnecessarily require round trips to an additional physical location. Thus, the idea should not be considered.

As a result, the current choice is either to integrate the translation to and from SPDY within Tor or by using proxies: one on the client and one on the exit relay. We decided to employ proxies on both sides of the Tor network because we believe in their benefits. Firstly, following the principle of separation of concerns, the new functionality – employing SPDY over Tor – is divided from the Tor processes as it is just a new way to use the Tor network. Moreover, Tor is supposed to secure and anonymize streams regardless what kind of data they carry. In addition, when the translation happens in different processes, the implementation should be easier and one can be sure that no new bugs are introduced in the core logic of Tor itself. It is even more important as the SPDY protocol is work in progress. Therefore, frequent changes are expected which should be reflected in the implementation. We believe that this would be easier when proxies are used.

We want to note that in discussions at the Tor developer meeting in July 2013, some Tor developers were skeptical about employing a proxy because: 1) the latter has to be run additionally by the exit node operator; and 2) it is in fact a man-in-the-middle proxy, which is not desired. According to them, the translation should happen within the Tor process on the exit node. We believe this argument is false as there is a difference between an exit that runs a man-in-the-middle attack and an exit that runs a benign proxy (which is not an attack as long as it does not modify or log the traffic).

6.1.2 *Design without Proxies*

If Tor really benefits from SPDY and the latter is implemented within the Tor infrastructure without employing proxies, the probable solution will likely be similar to this:

- The Tor Browser uses always SPDY when an *http://* resource is requested, that is, when end-to-end encryption is not used.
- The Tor client chooses exit nodes with a Tor version which is known to support the SPDY to HTTP translation.
- While building a circuit, the Tor client notifies the exit node via new cell type that the received stream should be first translated and then redirected to the destination.
- SPDY without TLS is used from the Tor Browser to the exit node, which transforms the former to HTTP and sends request to the web server.

6.2 BENEFITS FROM USING SPDY WITHIN THE TOR NETWORK

Considering the results in Section 5.1, one can expect that employing SPDY with Tor would improve the performance. However, we cannot expect neither such great improvement, nor same improvement for different web sites, because we utilize SPDY only from the client to the exit node. That is, HTTP is still applied from there to the final destination. This means that we should expect greater improvements for the fastest web sites – those the RTT to which is smaller compared to the latency of the Tor network. On the other hand, only a tiny or no improvement is expected for slow web sites. Nevertheless, SPDY should not be slower than HTTP.

Firsts of all, the results presented on Figure 5.26 illustrate huge deviations from the average values for certain web sites. Sites like (34) and (54) give

different absolute values each time when they are benchmarked. Moreover, such Asian web sites behave strange when they are requested via Tor. The following scenarios were observed:

- Answering with TCP RST to a TCP SYN.
- Answering with two TCP SYN ACK segments with different sequence numbers to a single TCP SYN.
- Stop sending any data on a TCP connection without closing it.
- Pause sending data on a single TCP connection for a large amount of time (30, 60 or even more seconds).

It is clear that on average the latency in Tor is better when SPDY is used (see Table 5.3), although the benefit is not so large as it was expected. Web sites are then around 4 % faster on average.

On Figure 5.26, it can be seen that the ten fastest web sites with HTTP over Tor are clearly faster when SPDY is applied. However, some of the slowest web sites are also faster with SPDY.

On Figure 5.27, one observes that 41 out of 54 web sites are faster with SPDY. The greatest benefit is for (3), which is 34 % faster with SPDY. On the other hand, three sites are over 50 % faster when HTTP is used. The site with least SPDY benefit – (30) – showed almost always worse results with SPDY.

It should be mentioned that when using the proxies, HTTP pipelining is not employed from the exit node to the requested web sites. Tests with libcurl in June, 2013, showed that its pipelining implementation is not yet reliable enough to be used in microspdy2http. On the other hand, when HTTP is used with Tor, the Tor Browser utilizes pipelining directly to the destination which is reported to be very efficient [Per12].

The difference of pipelining utilization may be the answer why some web sites are always slower when the proxy setup is used with Tor.

As expected, there is less traffic between Tor relays when SPDY is used (see Table 5.4). The number of all TCP/IP packets exchanged between the middle and the exit node is 10 % smaller with SPDY which should explain the latency benefits. The reduced total number of bytes is 8 % predominantly due to the compression of SPDY headers. The input bandwidth of the exit node is reduced by 22 % while the output only 6 % because almost all the data in requests consists of headers, which are compressed in the used SPDY setup.

From Figure 5.28, one cannot conclude that the latency improvement is a function of the traffic improvement. The reason is that the traffic is measured only within the Tor network, while the time results are affected from the latency to the destination as well. Moreover, it can be noticed that one web site clearly used less traffic, but gave worse page load time for SPDY.

6.3 PRIVACY CONCERNS WHEN USING SPDY WITH TOR

There are some specific SPDY characteristics which may raise privacy concerns and possibly make SPDY – in the way it is currently specified – require adaptation for use with Tor, where privacy and security are very important. Most of the concerns are aimed to both end-to-end SPDY communication and communication via SPDY proxies.

6.3.1 *SPDY Settings*

The first issue may arise from the ability of SPDY settings to be persisted. A server/proxy may request from a client to persist some values and send them back in another session in the future. When used with Tor, clients should not do this, because they will leave traces of visits to a web site on the local machine and furthermore, in a future session, the server will link the current visit to a previous one.

6.3.2 *SPDY Server Push*

A client has the ability to cancel a pushed resource by closing its stream, for example when a server attempts to push a resource which has already been cached. Thus, the server/proxy is able to probe if the client has recently requested the same resource. To resist on such an attack, a client should either reject all push attempts or pretend to accept all of them.

6.3.3 *SPDY Session Creation*

A notable feature of SPDY is the creation of a single connection per host. This means that resources from a same host address loaded in different tabs share the same SPDY session, which is the case with third-party elements such as advertisements. Such requests may leak information about sites which a user opens within a short time. However, with the current stable version of the Tor Browser (as of July, 2013), an exit node can detect this even with HTTP – a single Tor circuit is used for all HTTP connections within a short time – while the target web server cannot do it assuming that no other information aids tracking (e.g. cookies).

However, when an end-to-end SPDY connection is applied, it is encrypted. Thus, exit nodes cannot link visits of different web sites to the same user, but for the target web server, this is still possible.

The design goal of the Tor Browser is to open a new circuit for each newly opened browser tab if an address from the same host is not loaded in another tab [PCM13, Section 4.5. Cross-Origin Identifier Unlinkability, 11. Exit node usage]. This is currently being developed for the future versions of the Tor Browser. Something similar should be applied also for SPDY: a separate SPDY session employing the Tor circuit responsible for the respected tab should be used.

6.3.4 *SPDY Session Lifetime*

Furthermore, SPDY sessions are intended to be long-lived. In theory, if a connection is busy, neither the client, nor the server would close it until the number of requests or pushes reaches 2^{30}. In contrast, a Tor circuit has a lifetime of 10 minutes and is then closed if there are no other streams using it. The reason is that a new path should be employed. A SPDY client used over Tor should limit the session lifetime even if there are still requests to be sent on the same one. Otherwise, Tor may never close the responsible circuit.

6.4 SPDY SERVER PUSH BASED ON PREDICTIONS

The results from the evaluation of the proposed prediction algorithm for SPDY push in Subsection 5.3.4 clearly illustrate that it would be useful for certain types of input data. The relation between the number of requests, responses, mistakes and pushes presented on Figure 5.29 is very close to the ideal case on Figure 3.2. That is, for that single day for Site1, the algorithm is capable to push almost all assets ($\approx$ 96 %) with relatively small number of mistakes ($\approx$ 12 %). Moreover, the push rate can be controlled by a single parameter – our *Probability threshold*.

However, from Table 5.11, it can be seen that this single day sample contains more than 1.2 million requests to Site1. Furthermore, the set of requested resources is rather small – the proportion of unique pages and assets in Table 5.12 is less than 0.6 %. As a result, there is a sufficient amount of statistical data available to the algorithm to make good predictions.

A good result was also achieved for Site3 (Figure 5.31), although it is a small site with a very small number of assets (0.26 per page on that day).

On the other hand, the best results for Isp, Proxy1, Proxy2, and Proxy3 are not encouraging. The best achieved value of the cost function is over 0.98, and the push rate is bellow 10 % for Isp and Proxy1, and negligibly low for Proxy2 and Proxy3.

It can be concluded from the results, that the *Probability threshold* parameter is by itself not enough to control the push rate in the range from 0 to at least 1. This is especially visible when *push-more* is used (see Figure 5.30 and Figure 5.35).

During the evaluation, it was observed that same combination of parameters might give very different results for separate single day samples of same data set. That is, the number of visitors and the visited pages may vary every day, which affects the algorithm's performance. This can be seen also from the fact that the combination of parameters for the best single result and for the best average result per data set are different. Furthermore, each data set requires its specific parameters.

It is clearly possible to find a combination of parameters for each site that gives the best average results when all single day samples per site are considered. In Table 5.14, one can see that the resulting algorithm gives good results for Site1, Site2 and Site3 for $\omega = 0.5$. On the other hand, the results for Proxy2 and Proxy3 are not showing a real benefit. Here, we conclude that push is not beneficial: the cost function is almost 1. At the same time, there is a very small benefit for Isp and Proxy1, and almost the same result is given by Site4. Common among the last mentioned three data sets is the average fraction of unique assets, which is in the range of 63 % - 69 %. It can be seen in Table 5.16 and Table 5.14 that the larger this value is, the more the cost function increases with the exception of Site5 and Proxy2. We conclude that this is because more diverse assets are requested in data sets with a larger fraction of unique assets. Nevertheless, the algorithm is given input data only for a single day (a bit longer for Isp). It can be speculated that if the algorithm was given input data for a longer period, the fraction of unique resources might be reduced and the algorithm would then give better results as it would have more data.

Since the Isp data set is closest to what a Tor exit traffic would be, it may be concluded that there would be a very small benefit from SPDY push within the Tor network under the assumption that the chosen value for ω is suitable for Tor. However, according to the presented evaluation of header

and body sizes in Section 5.4, a much smaller value of ω should most likely be used. Thus, the proposed algorithm would not be suitable for Tor.

Turning to the results for Proxy2 (see Table 5.15), it may be seen that a SPDY proxy session might be rather long. Thus, the timeout is not enough and a parameter limiting the maximum lifetime should be used. Even though the session is long, this did not seem to improve the result for the push prediction heuristic.

It is clear from the results (see Table 5.13) that each data set requires different combination of parameters for achieving the best average results. Furthermore, a combination giving the best results for one data set might be disadvantageous for other data sets. Figure 5.39 and Figure 5.46 illustrate this most clearly. This is also rather disappointing, as this means that predicting resources to push would still require manual tuning, limiting the utility of the presented heuristic. However, it is conceivable that given larger data sets, a variant which works acceptably for many or even most sites might be found.

Table 6.1 summarizes how the individual input parameters affect the best average results considering Figure 5.47 to Figure 5.55. Nevertheless, the results are not always unambiguous. That is, one value of a parameter might be very advantageous for one data set while being disadvantageous for another one. Moreover, it might be speculated that the effect of a single parameter is dependent on the choice for all others.

It should be mentioned that the design of the prediction algorithm did not regard the possibility to push resources that might be needed in a subsequent page load in the same session. That is, it aims to push assets only for the current page load. However, one can see in Table 5.16 that a small fraction of the hits (1 % - 4 %, when the result for Proxy2 is ignored) is actually needed by another page load. This behavior may slightly increase the mistakes if the session length is shorter.

Name	Comment
Algorithm used	The data sets which benefit much from push gave best average results for *basic*. *basic-less* is the best for some of the others, but the push rate with it is rather low. *basic-more* gave good results only for some single day samples. However, for different data sets, a change in the parameter leads to different increase in the cost function.
Maximum asset size for push	The size limit affects each data set in different way since the size of the assets might be various. The cost increases much for Site1 to Site3 when a small value is applied because the push rate is drastically reduced, but in this case the total size of all mistakes is very low. It should be noted that our cost function does not really consider the size of an individual mistake and thus the size limit's benefits are understated in the plots. Nevertheless, Proxy1 and Proxy2 gave best average results when the parameter is 1,000. It might be because the latter somehow actually helps to reduce wrongly pushed pages and respectively mistakes.
No push for mobile devices and Consider browsers	All three values might be beneficial depending on the input data. Site1 and Site3 demand separate statistics depending on the requesting browser. Both are considerably affected when another value is used. In contrast, the same value is highly disadvantageous for Site2.
Keep history	Only Site2 gave best average result when this one is used. However, the difference when the parameter is applied and when not is negligible for all data sets except Proxy1 and Proxy2.
Session timeout	There is no big difference for both evaluated values. Only Site1 clearly needs longer sessions, while the result for Proxy1 is deteriorated by the larger value. However, almost all best single cases were achieved with the value of 3,600 s (see Table 5.9). On the other hand, Table 5.15 displays that all average values of the session length are below 550 s (ignoring Proxy2 in whose data set the sessions are rarely closed due to constant issuing of requests). Thus, a conclusion could be made that the optimal value of the parameter is slightly larger than 180 s.
Probability threshold	Most of the data sets needed a value of 0.4 - 0.5 for both the best case and the best average results. It was observed that for a given ω, the cost function has a minimal value when the threshold has a certain value depending on ω. That is, higher ω value requires lower threshold and vice versa.

Table 6.1: Summary of the effect of the evaluated input parameters on the results.

6.4.1 *Keeping History of Previous Visits*

The algorithm can use history of all resources sent to the same client into previous sessions. Such resources are not allowed to be pushed again. For implementing this feature, we can list two options:

- Keeping the history at the server/proxy. The application tracks all resources sent to the client and keeps a record for a given period of time. Nevertheless, neither the user can be always uniquely identified, nor the server can know when the client has discarded a cached resource. Thus, such an approach may reduce push mistakes but will also reduce push hits, and moreover, will require additional space on the server. This option was used for evaluating the algorithm. See Table 6.1 for discussion.

- Keeping the history at the client. The client knows what it has in the cache. An efficient way to send this information to the server is needed. For this purpose, a Bloom filter[1] could be utilized. It might be sent within a *DATA* frame without stream ID immediately after a SPDY session is opened. The frame should be combined together with the first request (*SYN_STREAM*) and/or the first *SETTINGS* frame in a single TCP/IP packet. Thus, no additional packets would be transmitted. The size of the Bloom filter should be such that it can fit together with the other frames into the packet. If one assumes that 1,000 B should be employed for the Bloom filter, the implementation in squid may be considered [Squ] to see how many elements can be stored. That is, squid employs Bloom filters for checking if an URL is in its cache. It utilizes five bits per set entry and the probability of a false positive on checking an item against the filter is very small. Hence, 1,000 B may be used for 1,600 entries. This value is more than enough for the cached URLs per origin. However, if the same approach is applied for SPDY forward proxies, the set of URLs is limited by the settings of the browser and would potentially require a much larger Bloom filter.

However, both approaches raise privacy issues. While the Bloom filter is less critical as its probabilistic nature leaves some room for plausible deniability, we believe they still should not be used together with Tor, as sites or proxies at exit relays may easily tag users by gaining information about their cache.

6.4.2 *Implementation Notes about the Proposed Algorithm*

The algorithm needs to keep global statistics used for predictions, as well as session related information. Hash tables should be used for performance.

The only data structures which have to keep data in raw format are the maps `stat` and `sizes`. The first one keeps raw URLs, because the application has to know which resource to push. The second one keeps raw numbers showing the size of the assets. All other structures should keep hash values for space and time efficiency.

Nevertheless, keeping the `stat` URLs in raw form raises privacy concerns because the strings may include private data.

---

1 A Bloom filter is a data structure having low space requirements which is used to check if an item exists in a set. A query returns false positives with certain probability depending on the size of the filter and the number of elements in the set.

7 CONCLUSION AND FUTURE WORK

We presented that the SPDY protocol improves the performance of web servers and reverse proxies compared to HTTP and HTTPS. We also demonstrated that SPDY could be used to reduce the latency and traffic within the Tor network, if the proposed or similar design is adopted by the Tor community. The proposed algorithm for SPDY server push based on predictions gave promising results for application at reverse proxies or directly at web servers. Nevertheless, the use of SPDY server push by proxies at Tor exit nodes does not seem to be beneficial for the Tor network as collecting statistics would be invasive and the results would still likely be mediocre, resulting in costly mistakes.

In future work, the proposed design and its implementation for utilization of SPDY with Tor should be optimized in such a way that the observed performance degradation of some web sites is eliminated. This will likely require improvements to the networking code of the proxies. Furthermore, the same or a similar design should be integrated with Tor. The open question of how exit nodes with SPDY support could be identified should be solved. Changes to the SPDY protocol might be required as well to meet the requirements for privacy.

Regarding the push predictions, the proposed algorithm should be optimized to give more control to the operator for choosing the amount of pushed resources. Moreover, an easier way to select appropriate input parameters for given scenario should be investigated.

[ACV10] J. Arkko, M. Cotton and L. Vegoda. *IPv4 Address Blocks Reserved for Documentation*. RFC 5737 http://www.rfc-editor.org/rfc/rfc5737.txt. RFC Editor, Jan. 2010.

[Ale] *Alexa Top 500 Global Sites*. URL: http://www.alexa.com/topsites (visited on 07/06/2013).

[Apa] *Log Files – Apache HTTP Server Version 2.2*. URL: http://httpd.apache.org/docs/2.2/logs.html#accesslog (visited on 10/2013).

[BP#1] M. Belshe and R. Peon. *SPDY Protocol – Draft 2*. URL: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft2 (visited on 10/2012).

[BP#2] M. Belshe and R. Peon. *SPDY Protocol – Draft 3*. URL: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3 (visited on 10/2012).

[BP12] M. Belshe and R. Peon. *SPDY Protocol*. Internet-Draft http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00. IETF Secretariat, Feb. 2012.

[BP13] M. Belshe and R. Peon. *SPDY Protocol – Draft 3.1*. Sept. 2013. URL: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1 (visited on 10/2013).

[Bro] *Browser Capabilities Project*. URL: http://tempdownloads.browserscap.com/ (visited on 10/2013).

[DMS04] R. Dingledine, N. Mathewson and P. Syverson. "Tor: The Second-Generation Onion Router". In: *In Proceedings of the 13 th Usenix Security Symposium*. http://www.freehaven.net/anonbib/cache/tor-design.pdf. 2004.

[Fie+99] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 http://www.rfc-editor.org/rfc/rfc2616.txt. RFC Editor, June 1999.

[Fir] *Firebug*. URL: http://getfirebug.com/ (visited on 2013).

[GA] J. loup Gailly and M. Adler. *zlib*. URL: http://www.zlib.net/ (visited on 12/2012).

[Gro] C. Grothoff. *GNU libmicrohttpd*. URL: http://www.gnu.org/s/libmicrohttpd/ (visited on 2013).

[Gro13] C. Grothoff. *A Benchmark for HTTP 2.0 Header Compression*. Tech. rep. July 2013.

[Lan12#1] A. Langley. *Transport Layer Security (TLS) Next Protocol Negotiation Extension*. Internet-Draft http://tools.ietf.org/html/draft-agl-tls-nextprotoneg-04. IETF Secretariat, May 2012.

[Lan12#2] A. Langley. *Issue 139744: Security: SSL compression infoleak*. Aug. 2012. URL: http://code.google.com/p/chromium/issues/detail?id=139744#c33.

[MA13] P. McManus and F. Akalin. *spdy-dev – Google Groups: SPDY 3.1 spec out*. Sept. 2013. URL: https://groups.google.com/forum/#!msg/spdy-dev/_uvxTJkeCP0/VoKNaKxOb9YJ (visited on 10/06/2013).

[mme13] mmenke@chromium.org. *Issue 233863: HTTP Pipelining: Multiple Parallel asynchronous XMLHttpRequest (aka. ajax) block each other*. June 2013. URL: http://code.google.com/p/chromium/issues/detail?id=233863#c7.

[Mur10] S. J. Murdoch. *Using the SPDY protocol to improve Tor performance*. Draft https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-using-spdy.txt. The Tor Project, Feb. 2010.

[Net] *NetExport | Software is hard*. URL: http://www.softwareishard.com/blog/netexport/ (visited on 08/2013).

[Not11] M. Nottingham. *Making HTTP Pipelining Usable on the Open Web*. Internet-Draft http://tools.ietf.org/html/draft-nottingham-http-pipeline-01. IETF Secretariat, Mar. 2011.

[Ope] *OpenSSL: The Open Source toolkit for SSL/TLS*. URL: http://www.openssl.org/ (visited on 12/2012).

[PCM13] M. Perry, E. Clark and S. Murdoch. *The Design and Implementation of the Tor Browser*. Draft https://www.torproject.org/projects/torbrowser/design/. The Tor Project, Mar. 2013.

[Per12] M. Perry. *Randomize non-pipelined requests to defend against traffic fingerprinting*. https://trac.torproject.org/projects/tor/ticket/5282#comment:6. The Tor Project, Tor Bug Tracker & Wiki. Apr. 2012.

[Pri] *Privoxy*. URL: http://www.privoxy.org/ (visited on 2013).

[RD12] J. Rizzo and T. Duong. "The CRIME Attack". In: *ekoparty Security Conference 8th edition*. http://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf. Buenos Aires, Sept. 2012.

[Smi12] P. Smith. *Professional Website Performance: Optimizing the Front-End and Back-End*. Wrox, Nov. 2012.

[Spd#1] *Can I use SPDY networking protocol*. URL: http://caniuse.com/spdy (visited on 08/2013).

[Spd#2] *SPDY: An experimental protocol for a faster web*. URL: http://www.chromium.org/spdy/spdy-whitepaper (visited on 10/2012).

[Squ] *SquidFaq/CacheDigests - Squid Web Proxy Wiki*. URL: http://wiki.squid-cache.org/SquidFaq/CacheDigests (visited on 09/2013).

[Ste] D. Stenberg. *curl and libcurl*. URL: http://curl.haxx.se/ (visited on 2013).

[Tor] *torsocks*. URL: http://code.google.com/p/torsocks/ (visited on 04/2013).

[Tsu] T. Tsujikawa. *Spdylay - SPDY C Library*. URL: http://spdylay.sourceforge.net/ (visited on 05/2013).

APPENDIX

EVALUATION OF PAGE LOAD TIME WITH SPDY OVER TOR

A.1 EVALUATED WEB SITES

The numbers are used in the graphics. The order is the same as in the Alexa's list.

1. http://www.amazon.com/
2. http://www.baidu.com/
3. http://www.wikipedia.org/
4. http://www.taobao.com/
5. http://www.yahoo.co.jp/
6. http://www.bing.com/
7. http://www.sina.com.cn/
8. http://www.ask.com/
9. http://www.163.com/
10. http://www.ebay.com/
11. http://www.hao123.com/
12. http://www.weibo.com/
13. http://www.amazon.co.jp/
14. http://www.xvideos.com/
15. http://www.fc2.com/
16. http://www.microsoft.com/
17. http://www.conduit.com/
18. http://www.tmall.com/
19. http://www.babylon.com/
20. http://www.xhamster.com/
21. http://www.craigslist.org/
22. http://www.sohu.com/
23. http://www.amazon.de/
24. http://www.apple.com/
25. http://www.soso.com/
26. http://www.pornhub.com/
27. http://www.imdb.com/
28. http://www.amazon.co.uk/
29. http://www.xnxx.com/
30. http://www.ifeng.com/
31. http://www.bbc.co.uk/
32. http://www.neobux.com/
33. http://www.jd.com/
34. http://www.360.cn/
35. http://www.mywebsearch.com/
36. http://www.aol.com/
37. http://www.redtube.com/
38. http://www.alibaba.com/
39. http://www.go.com/
40. http://www.blogspot.in/
41. http://www.dailymotion.com/
42. http://www.vube.com/

43. http://www.adf.ly/
44. http://www.amazon.fr/
45. http://www.youporn.com/
46. http://www.about.com/
47. http://www.cnn.com/
48. http://www.ebay.de/
49. http://www.imgur.com/
50. http://www.rakuten.co.jp/
51. http://www.adobe.com/
52. http://www.directrev.com/
53. http://www.amazon.cn/
54. http://www.ku6.com/

Note that it is possible to load the following web sites into a frame as well. However, they were removed from the list due to inability to successfully complete enough number of page loads within the given timeout value of two minutes.

http://www.qq.com/
http://www.youku.com/

A.2 TOR CLIENT CONFIGURATION FOR CHOOSING NODES

```
StrictNodes 1
##spdytor2
ExitNodes 131.159.15.91
##ndnr1
EntryNodes 109.105.109.162
##middle node is 38.229.70.61 - Ramsgate
##exclude everything but 38.229.70.61,
##109.105.109.162 and 131.159.15.91
ExcludeNodes 0.0.0.0/3, 32.0.0.0/6, 36.0.0.0/7,\
38.0.0.0/9, 38.128.0.0/10, 38.192.0.0/11,\
38.224.0.0/14, 38.228.0.0/16, 38.229.0.0/18,\
38.229.64.0/22, 38.229.68.0/23, 38.229.70.0/27,\
38.229.70.32/28, 38.229.70.48/29, 38.229.70.56/30,\
38.229.70.60/32, 38.229.70.62/31, 38.229.70.64/26,\
38.229.70.128/25, 38.229.71.0/24, 38.229.72.0/21,\
38.229.80.0/20, 38.229.96.0/19, 38.229.128.0/17,\
38.230.0.0/15, 38.232.0.0/13, 38.240.0.0/12,\
39.0.0.0/8, 40.0.0.0/5, 48.0.0.0/4, 64.0.0.0/3,\
96.0.0.0/5, 104.0.0.0/6, 108.0.0.0/8, 109.0.0.0/10,\
109.64.0.0/11, 109.96.0.0/13, 109.104.0.0/16,\
109.105.0.0/18, 109.105.64.0/19, 109.105.96.0/21,\
109.105.104.0/22, 109.105.108.0/24, 109.105.109.0/25,\
109.105.109.128/27, 109.105.109.160/31, 109.105.109.163/32,\
109.105.109.164/30, 109.105.109.168/29, 109.105.109.176/28,\
109.105.109.192/26, 109.105.110.0/23, 109.105.112.0/20,\
109.105.128.0/17, 109.106.0.0/15, 109.108.0.0/14,\
109.112.0.0/12, 109.128.0.0/9, 110.0.0.0/7, 112.0.0.0/4,\
128.0.0.0/7, 130.0.0.0/8, 131.0.0.0/9, 131.128.0.0/12,\
131.144.0.0/13, 131.152.0.0/14, 131.156.0.0/15,\
131.158.0.0/16, 131.159.0.0/21, 131.159.8.0/22,\
131.159.12.0/23, 131.159.14.0/24, 131.159.15.0/26,\
```

```
131.159.15.64/28, 131.159.15.80/29, 131.159.15.88/31,\
131.159.15.90/32, 131.159.15.92/30, 131.159.15.96/27,\
131.159.15.128/25, 131.159.16.0/20, 131.159.32.0/19,\
131.159.64.0/18, 131.159.128.0/17, 131.160.0.0/11,\
131.192.0.0/10, 132.0.0.0/6, 136.0.0.0/5,\
144.0.0.0/4, 160.0.0.0/3, 192.0.0.0/2
```

A.3 BENCHMARK PAGE

The source code of the page used in the experiments is listed here. Its current online version can be found on http://dev.online6.eu/spdytor/benchmark.htm.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Benchmark page: page load time in iframe with JavaScript
</title>
<script type="text/javascript">
<!--
var begin_time;
var page_start_time;
var urls = new Array();
var times = new Array();// 2D
var current_url = -1;
var current_iteration = -1;
var pause_between = 0;
var iterations = 1;
var stopped = false;
var timeout = 10;
var timeout_fired = false;
var timeoutID;

function run_benchmark()
{
  begin_time = (new Date()).getTime();
  urls = new Array();
  times = new Array();
  var fr = document.getElementById('iframe_container');
  var lines =
    document.getElementById('urls').value.split(/\r\n|\r|\n/);
  iterations = document.getElementById('iterations').value;
  timeout = document.getElementById('timeout').value;

  for(var i=0,j=0; i<lines.length; i++)
  {
    if(lines[i] && lines[i].length > 0) urls[j++]=lines[i];
  }
  pause_between =
    document.getElementById('pause_between').value;
  current_url = -1;
  current_iteration = -1;
  display_data();
  current_url = 0;
```

```
  current_iteration = 0;

  stopped = false;
  load_current();
}

function page_loaded(process_it)
{
  if(process_it!=true && timeout_fired)
    return;
  clearTimeout(timeoutID);
  var load_time = (new Date()).getTime() - page_start_time;
  var fr = document.getElementById('iframe_container');
  fr.onload=null;
  fr.src='about:blank';
  if(!timeout_fired)
  {
    if(!times[current_url])
      times[current_url] = new Array();
    times[current_url][current_iteration] = load_time;
  }

  display_data();
  current_url++;
  if(current_url == urls.length)
  {
    current_url=0;
    current_iteration++;
  }
  if(current_iteration < iterations)
    setTimeout(load_current, pause_between * 1000);
  else stop_benchmark();
}

function stop_slow_results()
{
  timeout_fired = true;
  if(!times[current_url])
    times[current_url] = new Array();
  times[current_url][current_iteration] = 'TIMEOUT';
  page_loaded(true);
}

function load_current()
{
  if(stopped) return;
  timeout_fired = false;
  var fr = document.getElementById('iframe_container');
  fr.onload=page_loaded;
  page_start_time = (new Date()).getTime();
  timeoutID=setTimeout(stop_slow_results, timeout * 1000);
  fr.src = urls[current_url];
}
```

```
function display_data()
{
  var res = document.getElementById('results');
  var res_table = document.getElementById('results_table');
  var stat_table = document.getElementById('statistics_table');

  var done = (current_url+1
    + urls.length*Math.max(current_iteration,0));
  var time_passed = (new Date()).getTime() - begin_time;
  var time_left = time_passed / done
    * (urls.length * iterations - done);

  var text_status = '<pre><div>Done: '+done+' / '
        +(urls.length * iterations)
        +' urls</div><div>Time passed: '
        +(time_passed/1000)
        +' s</div><div>Time left (est): '
        +(time_left/1000)
        +' s</div></pre>';
  var text_results = "COUNT; URL; LOAD_TIME(ms)\n";
  var text_statistics =
    "COUNT; URL; ITERATIONS; MEDIAN; AVG; MIN; MAX\n";
  var sum = new Array(); //2D
  var statistics = new Array(); //2D
  var cnt = 1;
  for(var i=0; i< iterations; i++)
  {
    for(var u=0; u< urls.length; u++, cnt++)
    {
      var curtime;
      if(!statistics[u])
      {
        sum[u] = 0;
        statistics[u] = new Array();
      }

      if(i<current_iteration ||
        i==current_iteration && u<=current_url)
      {
        if('TIMEOUT' != times[u][i])
        {
          curtime = times[u][i];
          statistics[u][statistics[u].length] = curtime;
          sum[u] += curtime;
        }
        else
        {
          curtime = 'TIMEOUT';
        }
      }
      else
      {
```

```
        curtime = 'N/A';
      }
      text_results += (cnt)+'; '+urls[u]+'; '+curtime+'\n';
    }
  }

  for(var u=0; u< urls.length; u++)
  {
    statistics[u].sort(function(a,b){return a-b});
    var median = statistics[u].length%2
      ? statistics[u][(statistics[u].length - 1)/2]
      : (statistics[u][statistics[u].length/2 - 1]
              +statistics[u][statistics[u].length/2])/2;
    text_statistics += (u+1)+'; '
      +urls[u]+'; '
      +statistics[u].length +'; '
      +median+'; '
      +(sum[u] / statistics[u].length)+'; '
      +statistics[u][0]+'; '
      +statistics[u][statistics[u].length - 1]+'\n';
  }

  res.innerHTML = text_status;
  res_table.value = text_results;
  stat_table.value = text_statistics;
}

function stop_benchmark()
{
  stopped = true;
  clearTimeout(timeoutID);
  var fr = document.getElementById('iframe_container');
  fr.onload=null;
  fr.src='about:blank';
}
//-->
</script>
</head>

<body>
<h1>Benchmark page: page load time in iframe with JavaScript
</h1>
checkip.dyndns.org:
<iframe id="iframe_ip" width="400" height="40"
src="http://checkip.dyndns.org"></iframe>

<form action="" method="post" name="feedback">
<div>Seconds between page loads:
<input type="text" value="3" name="pause_between"
id="pause_between" /></div>
<div>Timeout for a page load (seconds):
<input type="text" value="60" name="timeout" id="timeout" />
</div>
```

```
<p>Addresses that will be loaded in this order:</p>
<textarea name="urls" id="urls" cols="100" rows="6">
http://www.ebay.de/
http://www.spiegel.de/
</textarea>
<div>Number of iterations:
<input type="text" value="1" name="iterations" id="iterations" />
</div>
<p><a href="javascript:run_benchmark();">Run it</a>
(results so far will be deleted!)
<a href="javascript:stop_benchmark();">Stop it</a></p>
</form>

<p>page will load here:</p>
<iframe id="iframe_container" width="640"
height="120" src="about:blank"></iframe>
<div id="results">Status here...</div>
<div>Statistics:</div>
<textarea name="statistics_table" id="statistics_table"
cols="100" rows="6">
</textarea>
<div>All results:</div>
<textarea name="results_table" id="results_table"
cols="100" rows="6">
</textarea>
<hr />
<i>ver. 1.3, 2013-07-18, Copyright (C) Andrey Uzunov</i>
</body>
</html>
```

A.4 SCRIPT FOR MEASURING TRAFFIC BETWEEN MIDDLE AND EXIT NODE

```
#!/bin/bash -eu

if [ "1" -gt "$#" ]
then
        echo "usage: script ip"
        exit
fi

IP="$1"

iptables -A INPUT -s "$IP" -j ACCEPT
iptables -A OUTPUT -d "$IP" -j ACCEPT

echo "Wait for keypress..."

read -n 1 -s

INPUT_PACKETS=`iptables -L INPUT -vxn | \
tail -1 | egrep "[[:digit:]]+" -o | head -1`
```

```
INPUT_BYTES=`iptables -L INPUT -vxn | \
tail -1 | egrep "[[:digit:]]+" -o | head -2 | tail -1`
OUTPUT_PACKETS=`iptables -L OUTPUT -vxn | \
tail -1 | egrep "[[:digit:]]+" -o | head -1`
OUTPUT_BYTES=`iptables -L OUTPUT -vxn | \
tail -1 | egrep "[[:digit:]]+" -o | head -2 | tail -1`

iptables -D INPUT -s "$IP" -j ACCEPT
iptables -D OUTPUT -d "$IP" -j ACCEPT

printf "IP: %s\nINPUT_PACKETS: %s\nINPUT_BYTES: %s \n\
OUTPUT_PACKETS: %s\nOUTPUT_BYTES: %s\n" "$IP" \
"$INPUT_PACKETS" "$INPUT_BYTES" "$OUTPUT_PACKETS" "$OUTPUT_BYTES"
```

B EVALUATION OF THE PROPOSED ALGORITHM FOR SPDY SERVER PUSH

B.1 SCRIPT FOR FILTERING PRIVOXY LOG FILES

```
#!/bin/bash -eu

cat "$1" |\
grep -v "ocsp." |\
grep -v "/ocsp" |\
grep -v "/crl" |\
grep -v "crl." |\
grep -v "safebrowsing" |\
grep -v "suggestqueries" |\
egrep -v "clients[0-9]\.google\.com" |\
egrep -v "sitecheck[0-9]\.opera\.com" |\
grep -v "\[too long, truncated\]" > "$2"
```