Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DISSERTATION

Improvement Strategies for the Signaling Performance of the Session Initiation Protocol

ausgeführt am

Institute of Telecommunications
der Technischen Universität Wien

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Em.O.Univ.Prof. Dipl.-Ing. Dr.techn. Harmen R. van As
und
Prof. Dr. rer. nat. Peter Reichl

durch

Dipl.- Ing. Christoph Egger

Wien, im Juni 2012

Danksagung

Mein Dank gilt o.Univ.Prof.Dr.-Ing. Harmen R. van As, für die Betreuung dieser Arbeit und für alle hilfreichen Diskussionen, die mir immer weitergeholfen haben. Er hat mir die Technik der ereignisbasierten Simulation näher gebracht, die für diese Dissertation essentiell ist.
Ebenso möchte ich meinem zweiten Betreuer Prof. Dr. Peter Reichl für alle guten Tipps und die vielfältige Unterstützung danken sowie für die Hilfe beim Publizieren der Papers, die im Rahmen dieser Dissertation entstanden sind. Als Projektleiter in vielen Industrieprojekten hat er mich immer bekräftigt und ermuntert weiterzumachen.
Meine Kollegen am Institute for Telecommunications haben auch besonderen Dank verdient. Allen voran Marco Happenhofer und Michael Hirschbichler, die viel Zeit mit mir verbracht haben, um die technischen Details meiner Arbeit mit mir zu diskutieren. Danken möchte ich auch Joachim Fabini für die Hilfe in wissenschaftlichen Fragen, die hervorragende Arbeit als Lektor und die mannigfaltige Unterstützung in allen Industrieprojekten.
Marcus Meisel und Ivan Gojmerac möchte ich für ihren Eifer beim ausführlichen Korrekturlesen danken.
Ich bin meinen Kollegen vom Forschungszentrum Telekommunikation Wien sehr dankbar für ihren fortwährenden Einsatz und die besondere Unterstützung beim Abschluss dieser Arbeit.
Meinen Freunden möchte ich für die Unternehmungen abseits des Studiums danken, die immer für Auflockerung gesorgt haben.
Meinen Eltern ein Dankeschön für die andauernde Unterstützung während des gesamten Studiums.
Der größte Dank gebührt aber meiner lieben Frau Sandra, die mich vor allem während der schwierigen Zeiten des Studiums sehr unterstützt und bekräftigt hat und ohne die ich diese Arbeit nicht geschafft hätte.

Abstract

With the upcoming success of Internet-Protocol based packet-switched networks, existing services like traditional voice communication must be migrated or replaced by alternatives. At the same time, new services are deployed that have been impossible or at least hardly implementable in circuit-switched technology. That leads to increasing traffic in such networks, especially for specific scenarios that require a huge number of messages to be exchanged. As a consequence, the probability of a system collapse rises if not appropriately counteracted. The 3rd Generation Partnership Project has selected the Session Initiation Protocol as main signaling protocol for its Internet-Protocol Multimedia Subsystem. That protocol has some performance drawbacks that are well-known and at the time of writing, *explicit* congestion notification mechanisms have been developed by the Internet Engineering Task Force and the European Telecommunications Standards Institute. The aim of this thesis is to specify novel *implicit* congestion detection mechanisms and to analyze their applicability.

A dedicated simulation environment is developed for this purpose that is able to verify the performance drawbacks of the Session Initiation Protocol and to act as tool for further developments. In order to develop counteracting mechanisms, the behavior of proxy servers of the Session Initiation Protocol is analyzed and documented. The results of this analysis form the base of two novel implicit congestion detection mechanisms that are used as trigger for further actions in a congestion situation. First, the manipulation of the re-transmission timers of the Session Initiation Protocol enables the system to handle the upcoming load. Second, incoming requests are rejected, therefore reducing the load that is sent to the downstream host. By means of simulations it is shown that the developed mechanisms perform well and are applicable to the protocol. In addition, the author presents an approach to offload a part of message processing onto a specialized processor in order to leave the main processor more time for the remainder. Measurements and application scenarios that verify the assumptions complete the work.

Zusammenfassung

Der Erfolg von Internet-Protokoll (IP) basierten paketvermittelten Netzen führt dazu, dass bereits existierende Dienste, wie zum Beispiel traditionelle Sprachkommunikation, angepasst, oder durch neue Alternativen ersetzt werden müssen. Gleichzeitig werden neue Dienste entwickelt, die mit bisherigen leitungsvermittelten Netzen nicht oder nur sehr schwer umsetzbar waren. Das führt in weiterer Folge zu ansteigendem Datenverkehr in solchen Netzen, speziell in besonderen Szenarien, die ein erhöhtes Datenvolumen benötigen. Als Konsequenz steigt die Wahrscheinlichkeit eines Kollaps des ganzen Systems, wenn keine geeigneten Gegenmaßnahmen ergriffen werden. Das 3rd Generation Partnership Project hat das Session Initiation Protocol (SIP) als Signalisierungsprotokoll für das IP- Multimedia Subsystem ausgewählt. Dieses Protokoll hat einige bereits bekannte Nachteile bezüglich Leistungsfähigkeit unter Hochlast. Die Internet Engineering Task Force und das European Telecommunications Standards Institute entwickeln bereits *explizite* Mechanismen, mit deren Hilfe betroffene Knoten einen anstehenden Datenstau mitteilen können. Das Ziel dieser Arbeit ist es, *implizite* Mechanismen zu entwickeln, und deren Anwendbarkeit zu untersuchen.

Zu diesem Zweck wird eine Simulationsumgebung entwickelt, die die Nachteile des Protokolls verifizieren kann und auch als Werkzeug für neue Entwicklungen verwendet wird. Für die Entwicklung der neuen, impliziten Mechanismen wird das Verhalten von Proxy Servern des Session Initiation Protokolls untersucht und dokumentiert. Die Ergebnisse dieser Analyse werden benützt um zwei neue Ansätze zur impliziten Datenstauerkennung zu entwickeln. Diese werden dazu verwendet, um im Falle eines erkannten Datenstaus entsprechende Gegenmaßnahmen zu ergreifen. Als Gegenmaßnahmen präsentiert die Arbeit auf der einen Seite eine dynamische Manipulation der Timer, die das wiederholte Übertragen von SIP Nachrichten steuern. Diese Änderung gibt dem System genug Zeit zur Abarbeitung der anstehenden Last. Auf der anderen Seite wird eine bestimmter Teil der eingehenden Nachrichten abgewiesen, um die Last am betroffenen System zu reduzieren. Mit Hilfe von Simulationen wird gezeigt, dass die neuen Mechanismen funktionieren und auf das Protokoll anwendbar sind.

Weiters wird ein Ansatz präsentiert, der einen Teil der Verarbeitung der Nachrichten auf einen speziellen Prozessor auslagert um dem Hauptprozessor mehr Zeit für den restlichen Teil zu lassen. Messungen und Anwendungsszenarien schließen die Arbeit ab.

Contents

1. Introduction **1**
1.1. Motivation . . . 2
1.2. Main Contribution . . . 4
1.3. Structure of this Work . . . 4

2. State of the Art and Related Work **7**
2.1. Session Initiation Protocol . . . 7
2.1.1. Concepts . . . 7
2.1.2. Transactions . . . 21
2.1.3. Performance Metrics . . . 25
2.2. Related Work . . . 27

3. SIP Network Simulation **29**
3.1. Simulation Concepts . . . 29
3.1.1. Event-based Simulation . . . 29
3.1.2. Random Number- and Variate-Generation . . . 34
3.1.3. Logging and Analysis of Simulation Data . . . 35
3.2. Event-based Simulation Environment . . . 36
3.2.1. Simulation Kernel . . . 37
3.2.2. Network and Protocols . . . 37
3.2.3. Random Number Generator . . . 38
3.2.4. Logging . . . 38
3.3. Model Creation for SIP based Networks . . . 42
3.3.1. User Agent . . . 42
3.3.2. Proxy . . . 45
3.3.3. Link . . . 49
3.3.4. SIP Messages and Transactions . . . 49
3.3.5. SIP Services . . . 51

4. Performance Improvement Strategies 55
4.1. SIP Performance Drawbacks Verification 56
4.1.1. Drawbacks of SIP Retransmissions 57
4.1.2. SIP Node Load Characteristics 63
4.2. Implicit Load and Congestion Detection 68
4.2.1. Simulation Scenario 68
4.2.2. Delay-based Congestion Detection 69
4.2.3. Pending-Transaction-based Detection 72
4.3. Congestion Handling 85
4.3.1. Simulation Scenario 86
4.3.2. Adaptive Timer Manipulation 86
4.3.3. Rate-Limiting 94
4.4. Offloading 95
4.4.1. Simulation Scenario 97
4.4.2. Results 98
4.5. Application Scenarios 100
4.5.1. Presence 100
4.5.2. Register 110
4.6. Summary and Strategy Recommendations 115

5. Measurement-based Verification 117
5.1. Measurement Set-Up 117
5.2. Measurement Results 118
5.2.1. Unprotected System 119
5.2.2. Delay-based Detection 123
5.2.3. Pending-Transaction-based Detection 127
5.3. Limiting Factors 134
5.3.1. SIPp arrival Process 134
5.3.2. Java Garbage Collector 135

6. Summary and Conclusions 137

A. Tables 143

Bibliography 152

1. Introduction

During the last couple of years, the transition from circuit switched technology for communication networks to packet switched technology based on the Session Initiation Protocol (SIP) [53] has become a key factor for the future success of the telecommunications industry as a whole. This transition process has started with fixed networks and is currently at the point of revolutionizing the mobile network landscape which, since the upcoming large-scale deployments of Long Term Evolution (LTE), does no longer provide circuit switched functionality at all. Therefore, all circuit switched services, particularly voice, must be migrated to the packet switched world or replaced by packet-switched alternatives. Main candidate for implementing signaling and control functionality in packet-switched IP networks is SIP which has been selected by the 3rd Generation Partnership Project (3GPP) [2] as main call signaling protocol for its IP Multimedia Subsystem (IMS) starting with release 5.

Simultaneously, the packet switched transition drives the deployment of new services, for instance presence services, in addition to traditional voice communication and will substantially increase the usage of SIP systems. With growing acceptance of these new services, the amount of SIP traffic is expected to increase, especially for specific scenarios causing a huge number of messages (for instance the update of the presence status at the beginning or end of a business day, or after a power failure). As a consequence, in such situations the probability of a collapse of the whole system rises unless counteracted by appropriate configuration of system parameters in the relevant SIP nodes like proxy servers or user agents.

In general, the performance of SIP as signaling protocol depends amongst other factors also on the choice of the underlying transport protocol, which can be either the reliable Transmission Control Protocol (TCP) [48], the unreliable User Datagram Protocol (UDP) [47], or some other protocol like the Stream Control Transmission Protocol (SCTP) [58].

When using UDP as transport protocol, SIP is in charge of ensuring that its messages will be reliably delivered, which is achieved by retransmitting messages in case they

have been lost. Message retransmission is triggered by specific timers which fire as soon as the time span until receiving a response exceeds a certain threshold. However, under heavy load this approach tends to introduce even more load to the system, because delayed messages may be wrongly interpreted as lost, and, based on this, retransmissions are triggered. In the worst case, this behavior may further evolve into a collapse of the whole system, and, once collapsed, it is hard for the system to recover even if the load starts shrinking substantially [28].
The work of this thesis deeply analyzes this negative effect using simulations and develops counteraction mechanisms in order to prevent collapse situations. The presented approaches are verified using measurements on a real system. Two common application scenarios are discussed that may lead to a congestion collapse. Furthermore the application of the newly developed mechanisms onto those situation is presented. It is shown that these low-complexity mechanisms add significant stability to the system and that collapse situations are reliably avoided.

1.1. Motivation

It is well-known that SIP lacks a suitable overload control mechanism and the Internet Engineering Task Force (IETF) as well as the European Telecommunications Standards Institute (ETSI) currently work on extensions for SIP to cope with this drawback [51] [17]. However, these extensions specify an explicit congestion notification mechanism that sends either dedicated messages using a new protocol or add a chunk of data to SIP messages in order to signal upstream entities a given overload situation. In such a situation, a server that already suffers from overload must generate that congestion information, and this is counterproductive. Additionally, the congestion information has to be sent to an upstream host that is responsible for the current overload situation and this host needs to react upon receiving that information. Such mechanisms need special implementations on all nodes that have to advertise their load state and need additional resources on hosts that already are in high-load situations. This combination is not optimal and there is a need for research on implicit mechanisms that sense the overload state without explicit knowledge of internal states or processes of the receiving entity.
Other protocols used for IP-based communication have implicit mechanisms that let them approximate how much data can be sent to a specific destination. One example is TCP that uses sequential acknowledgements that have to be awaited to infer on

network conditions between sender and receiver. By doing so, sender and receiver always use the optimal amount of resources for the transmission of data. As a result of extensive research and standard development, many congestion avoidance algorithms for TCP are available [3].

As far as the underlying transport protocol is concerned, SIP can employ the reliable TCP as well as the unreliable UDP or other protocols like SCTP. For TCP, either a connection set-up is necessary for each SIP message to be sent to the proxy or a TCP connection has to be kept open to all connecting clients. Both of these options introduce an overhead, hence big operators tend to prefer UDP as transport protocol, which implies certain consequences. Most importantly, when using TCP, the transport layer takes care of successful transmission of the messages, whereas in the case of UDP, SIP itself needs to ensure that messages are delivered successfully to the destination. In the latter case, this is achieved by means of the SIP retransmission timer E (see [53], section 17.1.2.2). Following the standard, a single request may be transmitted up to eleven times in total before the transmission attempts are terminated. This behavior of SIP has been specified in order to cope with the unreliable nature of Internet traffic. As a side effect, however, retransmissions are also triggered if messages are not lost, but delayed because of congestion (that is, delay longer than SIP timer T1). Processing these retransmissions at a server creates additional congestion, which after all is unnecessary. In a congested system there is a high probability that requests never reach their destination. Moreover, even if the request reaches the destination, the response might fail to reach the originating node. Thus, in the worst case, the system has to handle up to eleven SIP requests and up to eleven responses without any other effect than using processing power of the nodes and filling up the links with redundant data.

This thesis proposes a novel mechanism that is able to detect an imminent collapse of a SIP proxy and that can be used as a trigger for further actions. It develops implicit mechanisms for SIP that sense the congestion situation of a downstream host by continuously analyzing implicitly available information. Such information is for example the difference between the time stamps when a request has been sent and when the corresponding response has been received, denoted as Round Trip Time (RTT), or the number of pending SIP transactions. The implicit congestion detection mechanisms are developed by means of a dedicated simulation environment. The simulation results are validated using the industry-standard SIP message generation tool SIPp to estimate the applicability of the new mechanisms onto real-world systems.

Finally, two application scenarios that can lead to congestion collapse situations are presented.

1.2. Main Contribution

The main contribution of this thesis consists of two parts. The first part is the extension of an already existing event-based simulation tool with SIP modules in order to analyze the performance of the protocol and to verify new enhancement strategies. In the second part, the known performance disadvantages of SIP are verified using the new simulation modules and novel proposals for improvements are presented. The thesis classifies congestion control into a congestion *detection* phase, that is, identification of situations that may possibly lead to congestion, and a congestion *handling* phase, that is, applying a strategy upon detection in order to reduce the input load onto the congested entity. Handling strategies, on their parts, are partitioned into lossless strategies, that do not reject any incoming requests but try to overcome a congestion situation by giving the system enough time to handle the load. If that is not sufficient and the congestion situation remains, a lossy strategy, that rejects a fraction of the load is applied. These approaches have also been published by the author in [16], that describe how dynamic manipulation of the re-transmission timers affects the performance of SIP. In [14], the behavior of SIP proxies is presented under various load situations and in [15] the delay-based congestion detection mechanism is shown. [13] demonstrates a congestion detection mechanism that uses the number of pending transactions.
In order to verify the results that have been gained through simulations, a real SIP system has been set-up and the new mechanisms have been tested thoroughly. It is shown that these mechanisms increase the overall stability of the system and as a result also the number of successful SIP transactions that a congested system can handle. Each single successful SIP transaction means revenues for an operator of a SIP system and it is therefore of major interest to keep their number as high as possible.

1.3. Structure of this Work

The chapters of this thesis are organized as follows. Chapter 2 gives the reader basic knowledge about the concepts of SIP and performance metrics. This section also

presents other work that is related to the contents of this thesis. Chapter 3 outlines the simulation environment that has been used to achieve the results and the concepts for creating simulation models for SIP. Chapter 4 presents strategies to improve the performance of the protocol in order to cope with the problems. This chapter is the main contribution of this thesis and proposes two new implicit congestion detection mechanisms that are able to prevent collapse situations and an analysis of an offloading strategy that moves a part of the protocol processing to an external processor. Application scenarios are presented in order to prove the applicability of the assumptions. The simulated results are verified using measurements in a laboratory environment in section 5. Chapter 6 draws conclusions and suggests further research questions that may be realized in future work.

2. State of the Art and Related Work

This section introduces the reader to the concepts of the Session Initiation Protocol (SIP), that is, the SIP messaging concepts as well as the SIP message format and SIP transactions. Following, a review of SIP performance metrics utilized in the later chapters is presented. Finally, this section summarizes related work that has been done in the area of SIP performance.

2.1. Session Initiation Protocol

The Session Initiation Protocol has been standardized by the IETF within RFC 3261 [53]. SIP is an application-layer signaling protocol for creating, modifying, and terminating multimedia sessions as well as to deliver information that is usually non-session-based such as a presence state or an instant message. The sessions can be voice calls, multimedia distributions, whiteboard applications or any sort of multimedia communication. This chapter provides basic information about this protocol, starting with basic concepts, explaining SIP User Agent, Proxy server and the message format in section 2.1.1. Section 2.1.2 goes into the details of SIP transactions, before section 2.1.3 presents a metric for performance evaluations of SIP. The majority of this section has also been summarized in the diploma thesis of the author in [11].

2.1.1. Concepts

The functionality of SIP is based on a server-client model: a server running on a host is waiting for incoming messages, and depending on their content, a multimedia session is established, or an instant message is received and displayed on the terminal. When SIP is used to start a session the messages carry information on the details so that the communication partners can agree on media types. SIP also defines mechanisms

for authorizing and authenticating users, and a registration function that allows the users to publish their current location. Proxy servers are used to route SIP messages from the originator to the destination.

SIP User Agent

A SIP User Agent (UA) is software executed either running on a computer, a mobile device, or an embedded device similar to a plain old telephony service (POTS) telephone on behalf of a user providing connectivity to SIP services. The SIP UA is basically divided into a user agent client (UAC) part as well as a user agent server (UAS) part. The UAC generates requests based on external triggers such as a user clicking a button, and receiving and processing a response. The UAS on the other hand receives requests, processes them and generates responses, depending on the request-specific type and on user input. A valid SIP request according to [53] contains at least a start-line storing the method and the request-URI, and six header fields that are explained later in this section.
SIP makes use of so-called server and client transactions that involve a UAC and a UAS. A transaction comprises all messages from the first request sent by the UAC up to a final response sent from the UAS to the UAC. For this purpose the UA implements a transaction user (TU) that keeps track of all currently active transactions. The user or the provider configures the SIP UA by means of a user interface. UA settings are typically stored in a configuration storage that is used by the TU in order to find for example the outgoing proxy server. The transactions themselves use the operating system TCP/UDP/IP stack to send the messages to the desired destination. Typically a UA provides some sort of user interface, either graphical on a computer or smart phone, or as telephone receiver and buttons on a hard phone.

SIP Proxy

Generally speaking, proxies are elements that route requests to servers and responses to clients [53]. SIP extends this principle and introduces multiple proxy types with various functionality, like:

- route requests to the users current location
- implement provider call-routing policies
- authenticate and authorize users for services
- registration to store the current locations of the users

The most important proxy type for a provider of SIP based services is the registrar. A UA can register itself at the registrar such that messages directed to the UA will find their destination non-depending on the UA's current point of attachment (IP address). This has the benefit that a UA willing to send messages to other UAs only has to know the address of the proxy and not all UAs transport addresses. Moreover, authentication and authorization are important tasks implemented at a registrar to prevent misuse of resources.
A proxy server can operate either in stateless or in stateful mode for each request [53]. In stateless mode, a proxy simply forwards each request downstream to a single SIP node, the so-called next SIP hop. The next hop is determined by a routing decision based on the SIP request and header values after which the proxy discards information after the message has been forwarded. In stateful mode, on the other hand, the proxy server remembers all information on each incoming and sent requests and corresponding responses, and uses this state information for the processing of future messages. A stateful proxy is therefore a SIP transaction processing engine and handles incoming messages with the following stages:

1. Parsing and validation of the request
2. Preprocessing of routing information
3. Determine target of the request
4. Forward the request to the corresponding target(s)
5. Process responses

These steps are performed by any stateful proxy server and can be adopted to model a SIP proxy for simulation as will be shown in section 3.3. Each message that arrives at a proxy server must be analyzed for errors and parsed to create an internal representation of the message. The following section describes the format of SIP messages and some mandatory header fields.

SIP Message Format

SIP is a text based protocol, very similar to the Hypertext Transfer Protocol (HTTP), which uses the UTF-8 charset for encoding. A SIP message can either be a request sent by a user agent client or a response sent by a user agent server. Both messages

have a start line, one or more header fields and, following an empty line, an optional message body.

```
sip-message  =  start-line
        *message-header
        empty-line
        [ message-body ]
start-line = Request-Line / Status-Line
```

Listing 2.1: BNF Syntax of SIP Messages

Each line has to be terminated by a carriage-return line-feed sequence and the empty line has to be present even if no message body exists. Most of the syntax of SIP is identical to HTTP 1.1, however, SIP is not an extension of HTTP.

SIP Request: Requests have a request line as start line, which consists of a method name, and a request URI followed by the protocol version SIP/2.0:
`Request-Line = Method SP Request-URI SP SIP-Version CRLF`
RFC 3261 [53] defines six methods:

- REGISTER for registering a contact at the registrar
- INVITE for setting up a session
- ACK for acknowledging the establishment of a session
- CANCEL for canceling a pending request
- BYE for terminating a session
- OPTIONS for querying servers about their capabilities

Multiple extending RFCs (for example RFC 3428 [6], RFC 3903 [44], and RFC 3265 [50]) define several additional methods:

- MESSAGE for instant messaging
- SUBSCRIBE, NOTIFY for event notification
- PUBLISH for event state publication

The request-URI identifies the destination to which the request is being addressed. To be compliant with RFC 3261 [53], the version field has to be set to SIP/2.0. This is an example of a valid request line:
`INVITE bob@ims1.tuwien.ac.at SIP/2.0`

SIP Response: Response messages start with a status line, which consists of the protocol version followed by a numeric status code and its textual representation.
`Status-Line = SIP-Version SP Status-Code SP Reason-Phrase CRLF`
The SIP version must be set to SIP/2.0 as in the request messages. The status code field is a three digit status code which indicates the success or failure of the corresponding request. The values of this field are categorized in the following way:

- 1xx: Provisional response: indicates that the request was received and is in process.
- 2xx: Success: the request was successfully received, understood and accepted.
- 3xx: Redirection: further actions are necessary for completing the request.
- 4xx: Client error: the request has bad syntax or cannot be processed at this server.
- 5xx: Server error: the request is OK, but the server had some internal error.
- 6xx: Global failure: the request cannot be processed at any server.

Values $<$200 are called provisional responses, $\geq$200 are called final responses. The reason phrase intends to be a human readable representation of the status code. RFC 3261 [53] recommends how this reason phrase should look like, but implementations may choose their own text. This is an example of a valid status line:
`SIP/2.0 180 Ringing`

SIP Header Fields: SIP header fields follow the start line in both, requests and responses. RFC 3261 [53] defines mandatory header fields, which have to appear in each SIP message, and optional ones which are only included when required. SIP header fields consist of a header field name, a colon, and the header field value:
`header = "header-name" COLON header-value *(COMMA header-value)`
It is also possible that multiple header fields of the same name appear in the same message. They can then be combined into one line using a comma.
The six mandatory fields are the most important ones for SIP:

- From: This field specifies the logical sender of a request. It contains a SIP URI and optionally a display name. Different SIP entities use this field for selecting some processing rules for a particular request (for example automatic call rejection). This is an example of a valid *From* header field:
  `From:  Alice <sip:alice@ims2.tuwien.ac.at>`
- To: This field specifies the logical recipient of a request. In most cases this will be the address-of-record of the desired user or resource. This header field is not

used for routing a request, it is thought for human reading and thus it is also capable to hold a display name. This is an example of a valid *To* header field:
`To: Bob <sip:bob@ims1.tuwien.ac.at>`

- CSeq: This field is used to match received responses with previously sent requests. It consists of a sequence number and a method name which must match the method of the corresponding request. The sequence number has to be expressible within a 32-bit integer and less than 2^{31}. This is an example of a valid CSeq header field:
`CSeq: 4833 INVITE`

- Call-ID: This field is used to group together a set of messages which belong to the same SIP dialog, e.g.,:
`Call-ID: f81d4fae-7dec-6@128.131.0.1`

- Max-Forwards: This field sets the number of hops a message can pass through its way to its destination. At each SIP hop the number is decreased, when its value reaches zero before it reaches its final destination, a *483 Too Many Hops* response is sent.

- Via: Each proxy which processes a request adds a Via header field, so at its final destination, the message route is documented. The corresponding responses are sent across by reversing the Via header and following the list of nodes in reverse order. Example:
`Via: SIP/2.0/UDP p1.ims1.tuwien.ac.at`

SIP Message Body: All SIP messages can contain a message body of any type, even multipart bodies encoded using MIME. Specific header fields define the message body type in more detail:

- Content-Type: This field determines the type of the message body. Any defined MIME type is valid (for example *application/sdp*)
- Content-Length: Contains the length of the body in bytes
- Content-Disposition: This field extends the traditional MIME types and describes how the content should be interpreted by the User Agents

If not otherwise specified, the character encoding is UTF-8 for Content-Types of "Text". As within emails, the message body has to be separated from the header fields by an empty line.

In most cases, the Session Description Protocol (SDP) will be used as message body, because it is suited to describe multimedia sessions.

SIP Sessions

When two users want to communicate with each other, SIP provides the required signaling mechanisms and negotiates the type of session they want to establish. This is typically done by means of the Session Description Protocol (SDP [22]).

Session Description Protocol: The SDP contains all necessary information, including IP address, port numbers, codecs for audio and video. However, it is worth to mention that this encoding of layer 3 and 4 information is a breach of the Open Systems Interconnection (OSI) model which is the main reason for troubles that SIP faces when deployed over NAT. It is a simple textual format, an example is shown below:

```
v=0
o=Bob 4558285710 4586432945 IN IP4 128.130.0.1
s=Test IMS Session
c=IN IP4 128.130.0.1
t=0 0
m=audio 21534 RTP/AVP 0
a=sendrecv
m=video 21535 RTP/AVP 31
a=sendrecv
```

Listing 2.2: Example SDP Data

This example is SDP data with the following meaning: It is sent by Bob (o= line) from the IPv4 Address 128.130.0.1 (c= line), and has the subject "Test IMS Session" (s= line). Bob wants to receive audio coded with G711 μ-law (corresponds to "0", first m=line) at port number 21534 and video coded with H.261 (corresponds to "31", second m= line) at port number 21535. The v= line is the version information, a= contains optional information about the media. In this case it means that the media streams are bidirectional. Refer to RFC 4566 [22] for additional information on SDP. Although SDP is the most common way to describe a multimedia session, SIP does not depend on it. Any format of describing a session can be used.

Establishment of a Session: SIP uses an offer/answer model. A User Agent Client sends a request to an User Agent Server which replies with an answer. This means that the UAC first sends a session invitation with all its possible capabilities (offer).

Then the UAS checks this information, selects the adequate parts, adds own information and capabilities and sends it back (answer). After this procedure, both parts have the same view of the session and know each other's transport addresses and are therefore able to establish a session.

Entities involved in a Session: The most simple SIP session scenario involves only two User Agents and no Proxy servers. Therefore the UAC has to know the transport address of the UAS. Alternatively, the UAC can send messages that are directed to the UAS to its Proxy server which forwards them accordingly. A special form of a proxy server is a forking proxy (Figure 2.1), which forwards the SIP message to more than one destination location. This has the purpose that a user can be accessible at different locations at the same time (for example at a normal PC with a SIP software running on it and a SIP phone).

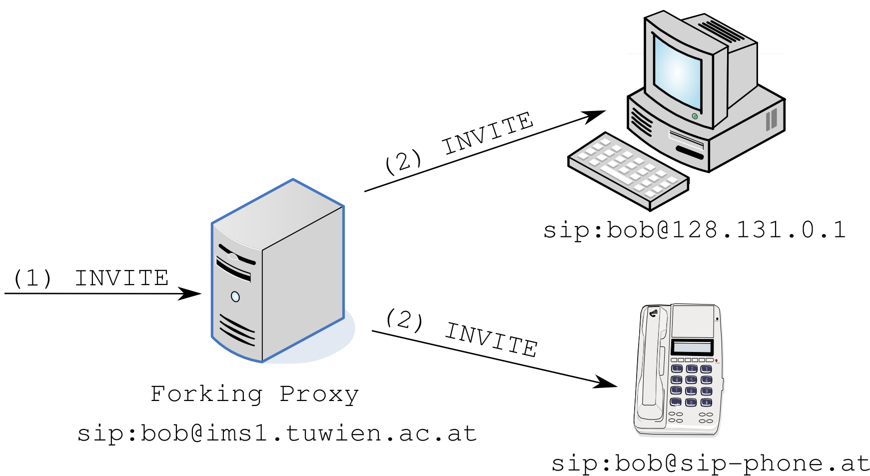


Figure 2.1.: Forking Proxy

A redirect server does not relay a message itself towards its destination, but it tells the originator of a message to try another location instead (Figure 2.2). This server is not a proxy server because it does not forward any message towards the terminating direction.

Message Flows for selected SIP Services

The following section lists examples of SIP messages and how they are relayed between the different entities if the service requires that. The first example is the SIP

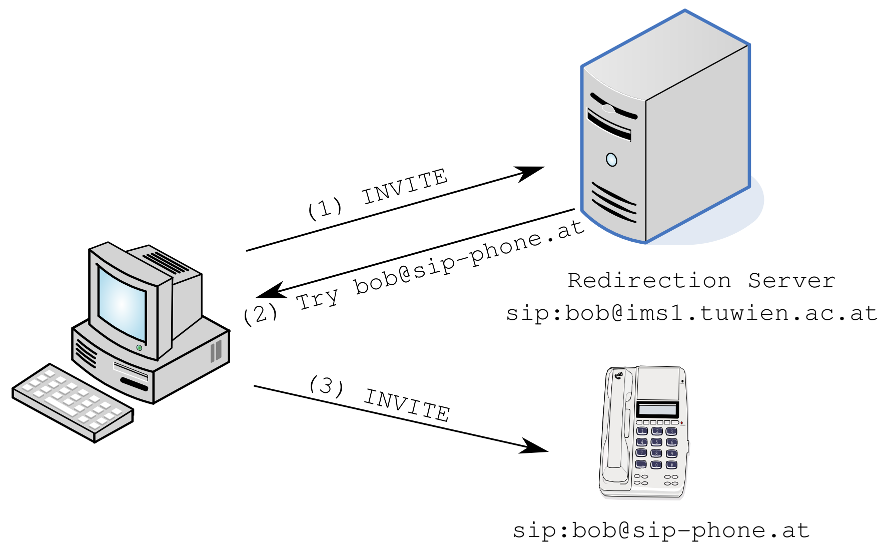


Figure 2.2.: Redirection Server

registration, which is necessary if a UAC shall be reachable using a URI rather than a plain IP address. Subsequently, the message flow for a session set-up is shown that is used to exchange session-specific information for a second connection between UAC and UAS. The third service that is presented in this section is the instant message service as representative for all other services that use non-invite transactions.

SIP Registration: Assume that Bob wants to register his current contact address at the registrar of the SIP domain `ims1.tuwien.ac.at`. This is required in order to store a generic SIP URI mapping to the current transport address. Figure 2.3 depicts a basic message flow for that example. The first REGISTER message that Bob's UAC sends to the registrar is shown in listing 2.3.

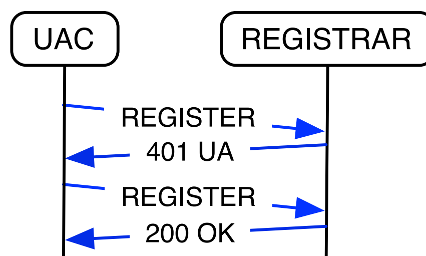


Figure 2.3.: Message Flow for SIP Registration

```
REGISTER sip:ims1.tuwien.ac.at SIP/2.0
Via: SIP/2.0/TCP 128.130.0.1:5060;branch=hgur84ft45
Max-Forwards: 40
To: <sip:bob@ims1.tuwien.ac.at>
From: <sip:bob@128.131.0.1>;tag=34565
Call-ID: fjgjgiirivklo0448758@128.130.0.1
Cseq: 1 REGISTER
Contact: <sip:bob@128.130.0.1>
Content-Length: 0
```

Listing 2.3: (1) Registration Request without Credentials

The request URI contains the SIP domain with which Bob intends to register. The To field indicates the address of record (AoR) of Bob, the From field contains the current location where he is reachable. The server answers with the following response:

```
SIP/2.0 401 Unauthorized
Via: SIP/2.0 TCP 128.131.89.89:5060;branch=hgur84ft45
To: <sip:bob@ims1.tuwien.ac.at>;tag=58432
From: <sip:bob@128.131.0.1>;tag=tag=34565
Call-ID: fjgjgiirivklo0448758@leukon
Cseq: 1 REGISTER
www-authenticate: digest realm="ims1.tuwien.ac.at", qop="auth",
    nonce="ea9c8e88df84f1cec4341ae6cbe5a359",
    opaque="", stale=FALSE, algorithm=MD5
Content-Length: 0
```

Listing 2.4: (2) 401 Unauthorized Response

This message contains a new header `www-authenticate` that includes a *nonce* value A nonce is a number used once by which the server challenges the UAC to calculate a response value as specified in RFC 2617 [20]. As next step, the UAC calculates the requested response value using its stored username and password and sends to the registrar by using a subsequent registration request:

```
REGISTER sip:ims1.tuwien.ac.at SIP/2.0
Via: SIP/2.0/TCP 128.130.0.1:5060;branch=hgur84ft45
Max-Forwards: 40
To: <sip:bob@ims1.tuwien.ac.at>
From: <sip:bob@128.131.0.1>;tag=34565
Call-ID: fjgjgiirivklo0448758@128.130.0.1
Cseq: 2 REGISTER
Contact: <sip:bob@128.130.0.1>
Authorization: Digest username="bob", realm="ims1.tuwien.ac.at"
    nonce="ea9c8e88df84f1cec4341ae6cbe5a359", opaque="",
    response="dfe56131d1958046689d83306477ecc"
Content-Length: 0
```

Listing 2.5: (1) Registration Request with Credentials

The `Authorization` header contains the response value as computed by the UAC based on the server challenge and the UAC's secret password. The registrar checks

if username and password are correct. If yes, the server sends the following response to Bob's UA:

```
SIP/2.0 200 OK
Via: SIP/2.0 TCP 128.131.89.89:5060;branch=hgur84ft45
To: <sip:bob@ims1.tuwien.ac.at>;tag=58432
From: <sip:bob@128.131.0.1>;tag=tag=34565
Call-ID: fjgjgiirivklo0448758@leukon
Cseq: 2 REGISTER
Contact: <sip:bob@128.131.0.1>
Expires: 7200
Content-Length: 0
```

Listing 2.6: (2) 200 OK Response

After completing this sequence Bob's UA is registered and the Registrar knows where to forward SIP requests for Bob incoming to the SIP domain `ims1.tuwien.ac.at`.

Session Set-Up and Termination: This section presents the message flow of a voice call that is set up by a user named Alice who wishes to call Bob. The messages that the involved entities send are shown in figure 2.4. As first message Alice's UA sends is an invitation request (1) to its outbound proxy, which looks like shown in listing 2.7. The proxy forwards the message as follows.

```
INVITE sip:bob@ims1.tuwien.ac.at SIP/2.0
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Max-Forwards: 40
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Contact: alice@128.130.0.2
Cseq: 1 INVITE
Content-Type: application/sdp
Content-Length: 151

v=0
o=cb 49584878383 485849305 IN IP4 128.130.1.1
s=-
c=IN IP4 128.130.1.1
t=0 0
m=audio 20000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Listing 2.7: (1) Invitation Request

The proxy looks up the inbound proxy by means of the Domain Name System (DNS) first (according to RFC 3263 [52]). Then it answers with *100 Trying* and forwards the INVITE to the inbound proxy of Bob (2). Listing 2.8 shows the resulting response message.

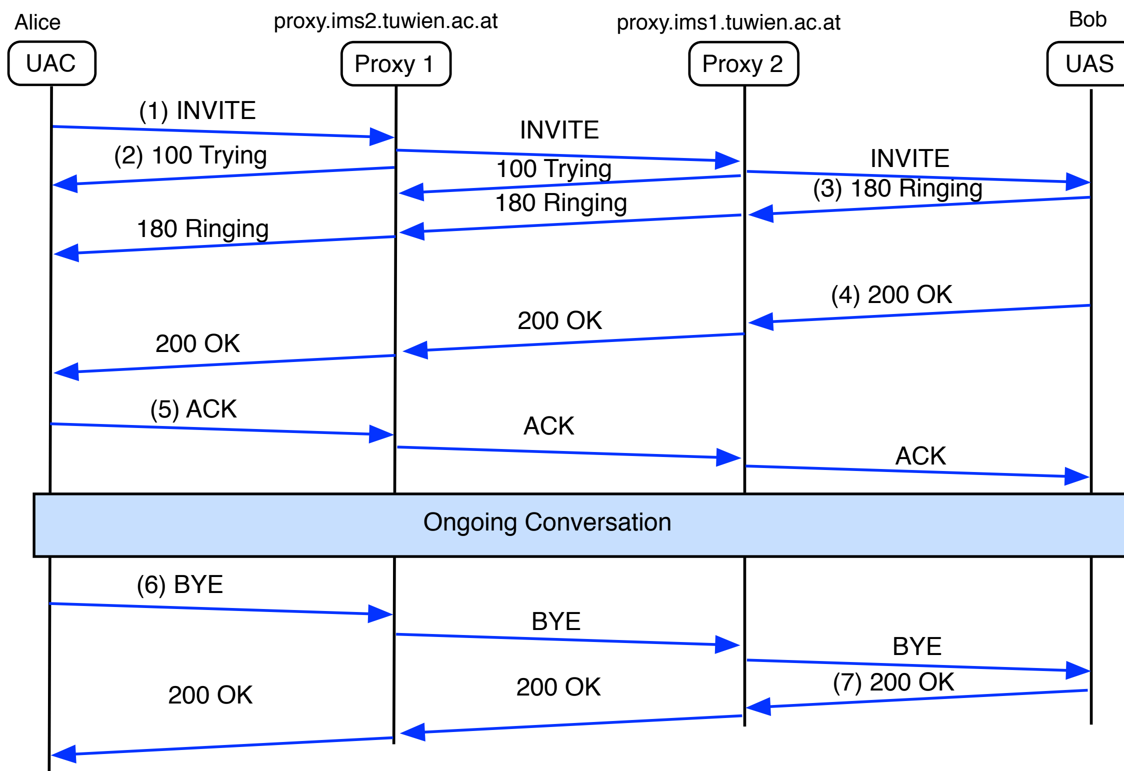


Figure 2.4.: Session Scenario

```
SIP/2.0 100 Trying
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Cseq: 1 INVITE
```

Listing 2.8: (2) 100 Trying Response

The inbound proxy of Bob forwards the request to the UA of Bob based on the contact information of the registration that happened earlier. The UA of Bob answers with *180 Ringing* (3):

```
SIP/2.0 180 Ringing
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Via: SIP/2.0/TCP proxy.ims2.tuwien.ac.at;branch=58332d
Via: SIP/2.0/TCP proxy.ims1.tuwien.ac.at;branch=438533
Via: SIP/2.0/TCP 128.131.0.1;branch=gkeice
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Cseq: 1 INVITE
```

Listing 2.9: (3) 180 Ringing Response

When Bob picks up, his UA sends a *200 OK* to its outbound proxy which sends the response back to the inbound proxy of Alice which again forwards the response to the UA of Alice (4):

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Via: SIP/2.0/TCP proxy.ims2.tuwien.ac.at;branch=58332d
Via: SIP/2.0/TCP proxy.ims1.tuwien.ac.at;branch=438533
Via: SIP/2.0/TCP 128.131.0.1;branch=gkeice
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Contact: bob@128.130.0.1
Cseq: 1 INVITE
```

Listing 2.10: (4) 200 OK Response

The UE of Alice answers with an ACK request, which has no response (5) in order to acknowledge the reception of the 200 OK:

```
ACK sip:bob@ims1.tuwien.ac.at SIP/2.0
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Max-Forwards: 40
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Cseq: 1 ACK
```

Listing 2.11: (5) ACK Request

Now the signaling of the UAs is complete and they know each others transport addresses for the media. The UAs are now able to start a multimedia connection according to the negotiated parameters.
In order to end the conversation, Alice hangs up and that results in Alice's UA sending a BYE message to Bob (6):

```
BYE sip:bob@128.130.0.1 SIP/2.0
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Max-Forwards: 40
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Cseq: 2 BYE
```

Listing 2.12: (6) End of the session (BYE)

Bob's UA ends the media streams and answers with a 200 OK (7):

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP 128.130.1.1:5060;branch=884kd8vc
Via: SIP/2.0/TCP proxy.ims2.tuwien.ac.at;branch=58332d
Via: SIP/2.0/TCP proxy.ims1.tuwien.ac.at;branch=438533
```

```
Via: SIP/2.0/TCP 128.131.0.1;branch=gkeice
From: Alice <alice@ims2.tuwien.ac.at>;tag=45832ec
To: Bob <bob@ims1.tuwien.ac.at>
Call-ID: fkfir949difivkfkio43@ws
Cseq: 2 BYE
```

Listing 2.13: (4) 200 OK Response to a BYE

The reception of the 200 OK response closes the session.

Instant Messaging: Instant messaging is standardized as extension in RFC 4328 [6] and defines an additional method for SIP: MESSAGE. This service is similar to the Short Message Service (SMS) and does not define any explicit association between messages except that of a user's own imagination. The destination of the MESSAGE request is defined by its request-URI that is typically instant message recipient's address of record (AoR). Proxy servers that are involved in the routing of that request have to handle them according to the rules of RFC 3261 [53] and should not send a provisional response according to RFC 4320 [56]. A UAS that receives a MESSAGE request has to respond with a final response immediately. Figure 2.5 shows the basic message flow for that service. The messages are forwarded by the proxies similar as

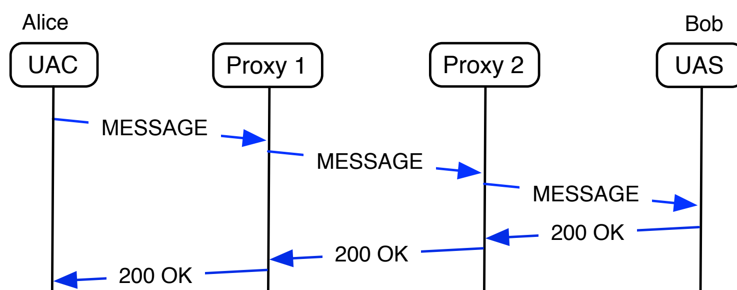


Figure 2.5.: Message flow for SIP MESSAGE

the messages of the session set-up presented in the previous section. Therefore only the initial request sent by the UAC, as well as the first final response that is sent by the UAS is shown.

```
MESSAGE sip:bob@ims1.tuwien.ac.at SIP/2.0
Via: SIP/2.0/TCP uac.ims2.tuwien.ac.at:5060;branch=z9hG4bK123dsghds
Max-Forwards: 40
To: <sip:bob@ims1.tuwien.ac.at>
From: <sip:alice@ims2.tuwien.ac.at>;tag=34565
Call-ID: fjgjgiirivklo0448758@128.130.0.1
Cseq: 1 MESSAGE
Content-Type: text/plain
```

```
Content-Length: 25

Text can be placed here.
```

Listing 2.14: (1) MESSAGE Request

This message contains the text “Text can be placed here” and upon reception the UAS displays this text on its user interface. This message is forwarded by Proxy1 and Proxy 2 to the UAS that answers with the following response:

```
SIP/2.0 200 OK
Via: SIP/2.0 TCP proxy2.ims1.tuwien.ac.at:5060;branch=z9hG4bK8485jfj
Via: SIP/2.0 TCP proxy1.ims2.tuwien.ac.at:5060;branch=z9hG4bK45245g3
Via: SIP/2.0 TCP uac.ims2.tuwien.ac.at:5060;branch=z9hG4bK123dsghds
To: <sip:bob@ims1.tuwien.ac.at>;tag=58432
From: <sip:alice@ims2.tuwien.ac.at>;tag=tag=34565
Call-ID: fjgjgiirivklo0448758@128.130.0.1
Cseq: 1 MESSAGE
Content-Length: 0
```

Listing 2.15: (2) 200 OK Response

The response is forwarded in reverse order (Proxy2 - Proxy1) back to the sender of the first initial request, that is, the UAC. With reception of the response, the transaction is complete on the UAC. This service is virtually the most basic form of the non-INVITE transaction as it consists of a single request message that the receiving UAS must respond to with a response message. In most cases that response will be a 200 OK.
More basic message flows for SIP can be found in RFC 3665 [35].

2.1.2. Transactions

As already presented by the examples of the previous section, the SIP standard RFC3261 [53] defines two types of SIP messages: INVITE messages, used for services requiring direct human interaction (for example voice call) and non-INVITE messages for services without direct human interaction (for example instant message). Each of these two message types defines its own client and server transactions. Transactions never exist as single individual, they are always created as a pair of client- and server transaction which are virtually bound by IDs and are not created in advance but for a service to take place. The Transaction User (TU) keeps references to all transactions. It creates a new client transaction if it wishes to send a new request and passes it the request to be sent. All SIP entities have to implement a TU except stateless proxies, which do not use transactions at all. This section shows how SIP messages and

transaction correspond to each other and highlights the differences between INVITE and non-INVITE transactions.

Non-INVITE Transactions

Non-INVITE transactions consist of all SIP messages that do not contain INVITE in the request line and are used for all services which do not directly require human interaction. Of course, human interaction is probably necessary for almost all services, however, the point here is that the complete delivery of a service to some host takes place without that. For example the delivery of an instant message and displaying it on a device can be done automatically, whereas the establishment of a voice call requires the user to pick up the phone when being called.

A large variety of services can be provided by means of non-INVITE messages such as presence, instant messaging or registering at the registrar. Therefore a SIP infrastructure handles more non-INVITE messages than INVITE messages, making the former ones more interesting for performance research issues. Figures 2.6 and 2.7 depict the server and client state machines for non-INVITE transactions when UDP is used as underlying transport. The client state machine works as follows: after sending a message the retransmission timer E is started and if the node does not receive a response within some time (T1), the message is retransmitted. The retransmission timer value is doubled at any retransmission until it reaches a specific limit (T2). From this moment onwards, a couple of additional retransmission attempts are started, after intervals of T2 each, until the overall limit of timer F is reached. Note that the corresponding standard proposes default values of T1 = 500ms, T2 = 4s and F = 64*T1. As major consequence, a single request may be transmitted up to eleven times before the transmission attempts are terminated. As soon as a final response (2xx-6xx) is received, the state of the transaction changes to *Completed* and a final timer K is started to detect possible response retransmissions. In this thesis, we follow RFC4320 [56] and do not send provisional responses to SIP requests. A non-INVITE server transaction, once created, starts in the state *Trying* and changes to *Proceeding* if the TU signals the receipt of a provisional response (1xx). The reception of a final response (2xx-6xx) leads to a state change to *Completed* and timer Timer J is started to detect possible request retransmissions. If a latter is received in the states *Proceeding* and *Completed*, the current response is retransmitted.

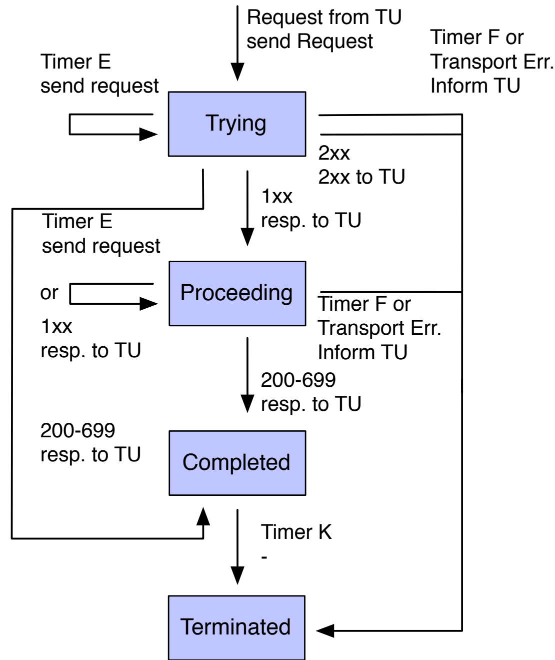


Figure 2.6.: Non-INVITE Client Transaction (cf. [53])

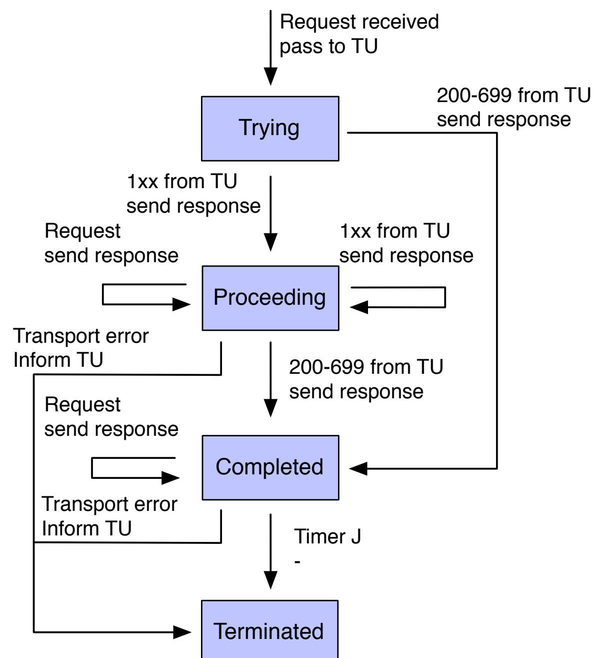


Figure 2.7.: Non-INVITE Server Transaction (cf. [53])

INVITE

INVITE messages are intended for services requiring to set up one or more additional data flows between two hosts for a previously unknown time, that is, a session.
The hosts use SIP in this case to agree on a set of common procedures to exchange media data such as voice or video. As the name implies, the client has to send an INVITE message and simultaneously create an INVITE client transaction. The server on the other hand, has to create a corresponding server transaction for each received INVITE message. Figures 2.8 and 2.9 depict the server and client transactions. Like non-INVITE transactions, INVITE transactions consist of several states that change corresponding to received or sent messages. The main difference in when compared to non-INVITE transactions is that INVITE transactions implement a three-way handshake in order to ensure that the response message reaches the originator of the request. This is required because the response message contains essential information to set-up a session, being accomplished by sending an ACK message as also shown in section 2.1.1.

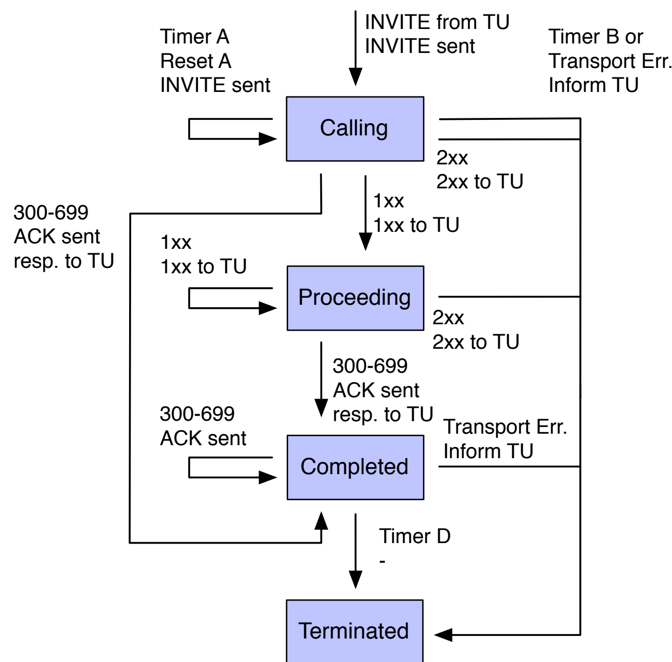


Figure 2.8.: INVITE Client Transaction (cf. [53])

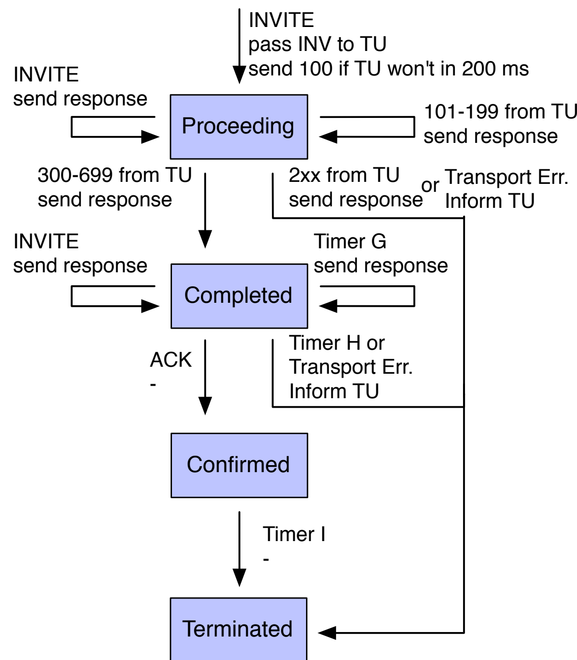


Figure 2.9.: INVITE Server Transaction (cf. [53])

2.1.3. Performance Metrics

The previous sections explained the basic concepts of SIP and how messages are handled by the involved hosts. With the increasing deployment of SIP based systems and therefore an increasing number of users that utilize new services, the chance for overload on some nodes rises. It is therefore important to continuously track the current load on all SIP nodes of a telecom operator to be able to detect possibly dangerous situations on the one hand as well as to apply load reducing procedures if necessary on the other hand. For this purpose, continuous measurements on respective hosts are required and thus, measurement metrics need to be defined. The IETF has created a working group called *IP Performance Metrics* (IPPM) and developed a set of metrics that can be applied to the quality, performance and reliability of Internet data delivery services [32]. The base framework for these metrics is defined in RFC 2330 [46]. However, this framework is designed for the network layer and is not suitable for application layer protocols. With further progress, the IETF has started another working group in order to define metrics for the remaining layers and named it *Performance Metrics for Other Layers* (PMOL). This group has standardized its first approach within RFC 6067 [8], but, however, the scope of this document

is limited to an end-to-end perspective. This perspective does not allow to profile the performance of intermediate entities in the signaling path because it provides only an outside view. Still, it is necessary to define the used measurement metric between two hops in order to detect possible performance problems on a specific intermediate hop. Additionally it is required that the analysis of measurement results based upon these metrics permits a clear differentiation between a congested (that is, overloaded) and a non-congested system.

The *Quality of Signaling* (QoSg) metrics, defined in [23], [25] and [26] have been chosen because they fulfill these requirements. The next sections present the metrics that are essential for the following performance analysis, that are:

- *Final Response Delay* (fRpD): Time span between sending a request until a final response is received
- *Request Transmits* (RT): Total number of requests sent per transaction
- *Success Rate* (SR): Ratio of number of successful transactions to injected transactions

Final Response Delay

The Final Response Delay (fRpD) is the time between sending the fist request until the first final response is received [23]. Figure 2.10 depicts its calculation. This metric

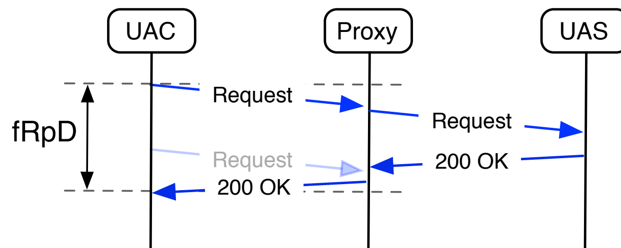


Figure 2.10.: Final Response Delay Definition

is important because the delay a server generates is highly dependent on it's current load and therefore is possibly a good indicator for the load state of a proxy server.

Request Transmits

The number of requests sent by the user agent client for a single transaction is represented by the Request Transmits (RT) metric [23]. For a well-configured system

that is in normal operation all SIP transactions receive a final response message after sending one request. That is in contrast to a congested system that requires more time to process requests and if that takes longer than Time E seconds, the client transaction re-transmits requests according to its state machine until it receives a response. This metric gives a good estimation on the amount of traffic that is sent to a proxy server and this is also a good indicator for congestion situations.

Success Rate

A transaction terminates successfully if a final response is received before the time out occurs. The ratio of successful transactions to all transactions is the Success Rate (SR) [23]. For performance analysis, measuring this value makes sense only on a proxy server that handles a large number of requests for many users. It offers an estimation on the amount of SIP transactions that a proxy server is able to handle.

2.2. Related Work

The SIP standard [53] proposes that a congested proxy sends a response, indicating that it is currently congested. That response includes a time span that has to be awaited until further requests may be sent. The authors of [45] show that current SIP overload controls are inadequate by means of simulations and present several novel algorithms to control SIP network overload. They demonstrate that their improved algorithms increase the goodput across several load levels and are robust to packet loss and delay. Unfortunately, the original SIP standard approach does also not satisfy the requirements that the IETF defined later for management of overload in SIP-based systems [51].

The European Telecommunications Standards Institute has specified an architecture to transport load and congestion state information by means of dedicated messages in [17]. This document describes the core features of the Next Generation Network (NGN) Overload Control Architecture (NOCA) and the Generic Overload Control Application Protocol (GOCAP) that is the architecture, entity behaviors, and protocol all grouped together. The GOCAP describes how required information is sent across the network between the single entities using Diameter [5] or SIP. The NOCA is an architecture that aims to provide feedback based processing load control.

SIP signaling delay is subject of a detailed investigation in [37] which analyzes the contribution of individual network components of the 3GPP IMS to overall delay. It is

shown that the processing and queueing delay of SIP messages depends on the current load of the system, that is, the higher the number of messages to be processed, the higher is the delay caused by processing. Additional delay may have different causes. If, for instance, the operator uses an access technology which relies on shared link usage (for example High Speed Packet Access [1]), delay depends on the current number of active users in the cell. In addition, specific IP based services used by the IMS, like for instance the Domain Name System (DNS) [41] and Diameter, as well as database access can further contribute to delay. All these individual delays sum up to the total end-to-end delay, that is the time span between sending the message at the User Agent Client and receiving it at the User Agent Server. Beyond that, for a SIP transaction to be successful, every request has to be answered with a response, which itself has to traverse all links and nodes as well, bringing in additional delay.
Hilt and Widjaja discuss in [28] how to model SIP entities for simulation and different network topologies of SIP servers. As a result, they show that a SIP server collapses under heavy load and analyze the conditions necessary for recovery. Additionally, it is demonstrated how the collapse of an edge server affects other servers. The paper discusses several overload control mechanisms and how transport protocols impact the performance of SIP. More specifically, it is demonstrated that the recovery from such a congestion collapse situation needs a considerable amount of reduction of input load.
Another detailed discussion of SIP server overload is presented in [29] and [62], where the authors use Markov-modulated Poisson process models and fluid-based Matlab simulations to show that even low CPU utilization can lead to a collapse. They show as well how to calculate a stability condition to avoid a system crash.
The authors of [63] present an approach that re-uses the idea of offloading TCP processing [42] to propose a so-called SIP offload engine that takes over the parsing of received SIP messages to relieve a system's main processor from this task. The benchmarking results show that a gain of throughput can be achieved depending on the server's pipelining architecture but also that this approach is not able to prevent a receive live lock in overloaded servers. Finally, much related work discussing various performance aspects of SIP-based systems has been published so far. Among the many papers, it is beneficial to point out [61] and [19] for a discussion of the performance of VoIP over High Speed Packet Access (HSPA) networks. Furthermore [18] as well as [36] present a comprehensive performance analysis of HSPA in live networks and [12] a Quality-of-Service enforcement architecture for NGN services.

3. SIP Network Simulation

This chapter highlights one part of the main contribution of this thesis, the extension of the simulation tool IBKsim [60] for SIP. Simulation in general is able to forecast the behavior of systems, even if they are too complex to be modeled in analytical or numerical methods. The simulation results however are only as good as the models that imitate the real-world behavior, but in comparison to analytical or numerical methods it offers much more flexibility. The following sections present simulation strategies and the used tool IBKsim. Following is a presentation on the simulation models for SIP that have been implemented for performance analysis.

3.1. Simulation Concepts

This section gives a basic insight on concepts that have been used for the implementation and extension of IBKsim. The main topics include event-based simulation and its components, how random numbers are generated for simulation and a discussion on logging alternatives.

3.1.1. Event-based Simulation

Event-based simulation imitates the behavior of a system by executing events in a chronological order, whereby the simulation time hops from one event time point to the next event time point [54]. Events that have been executed are then removed from the event list and any event is able to create new ones that are then inserted into this list. The list must be sorted in chronological order because always the event with the smallest time must be executed first and new events can be inserted anywhere in the list. Events in the context of SIP are for example, a user that sends an instant message, or makes a call to someone and this event triggers multiple following events such as sending a concrete SIP message that reflects the intention of the user.

Components

According to [34], discrete event-based simulations have the following common components:

Event Scheduler: This component holds the chronologically ordered event list and executes always the topmost event in the list. It is also the most frequently executed components of the simulation because it is always called before every event.

Simulation Clock and Time-advancing Mechanism: The simulator used for this thesis uses the so-called event-driven approach: a global variable that represents the current simulated time is held by the scheduler. The scheduler increments the time automatically to the next earliest event and then executes that event. This is in contradiction to the unit-time approach where the time is incremented by one unit and then the list is checked if there are any events to be executed.

System State Variables: These variables are global and represent the current state of the system such as for example the number of jobs in the queue.

Event Routines: These include the actual steps that must be executed in each single event and can update system state variables or create and schedule new events. For example, in a SIP system an event can be the creation of a new SIP message or the arrival of this message at a proxy server.

Input Routines: When starting the simulation, these routines read the configuration of the simulation from a configuration file at the beginning of a simulation.

Report Generator: Generates reports either during the simulation or at the end and writes them to a file or other output devices.

Initialization Routines: Set the initial state of the system to be simulated, initialize random-number generators and other operations that are necessary before starting the actual simulation

Trace Routines: Print out intermediate variables during the simulation and help debugging the program.

Dynamic Memory Management: As the number of entities that are required for a single simulation run is not known in advance, periodic garbage collection is necessary. Modern programming languages already provide this automatically, however, if not, the programmer has to take care of this. In the case of IBKsim, that is written in C++, the programmer has to implement either a garbage collection mechanism, or has to take care that each object no longer used is deleted immediately.

Main Program: The main program starts at first the input routines, then the initialization process, starts the actual simulation and finally it calls the output routines.

Summarizing, a discrete event-based simulation consists of a list that stores the events and the time points when they must be executed, a simulation kernel that takes the topmost event from the list and executes all events stored in the list in this manner. An input and an initialization routine take care that the simulation scenario is created in memory, is initialized correctly and a report generator writes out the results. Figure 3.1 shows the flow of control for the event-driven approach that has been chosen for IBKsim.

Arrival and Service Time Distribution

The components described in the previous section are common to all event-based simulations. When simulating a communications network of multiple nodes and links between these nodes, a message generation process must be implemented in some nodes to imitate the behavior of users that want to communicate. Due to the fact that SIP communication is mostly user-triggered, and the behavior of the users can not be predicted, the simulation uses random models to abstract. For this purpose, and since creation of true random numbers with enough entropy by computers is a non-trivial task, so-called pseudo-random numbers are used in this work. Pseudo-random numbers have the advantage, that one can recreate the sequence of numbers, when seeded with the same input. This effect is helpful for replicating interesting behavior of a simulation model. An automatically generated series of numbers is called pseudo-random if it is not possible to reject the hypothesis of being random with a given significance level when tested with a statistic test [55].
If for example the influence of a high number of users using their VoIP telephones to a SIP system has to be simulated, a pseudo-random number generator calculates the time points when these users want to send an instant message and thus initiate

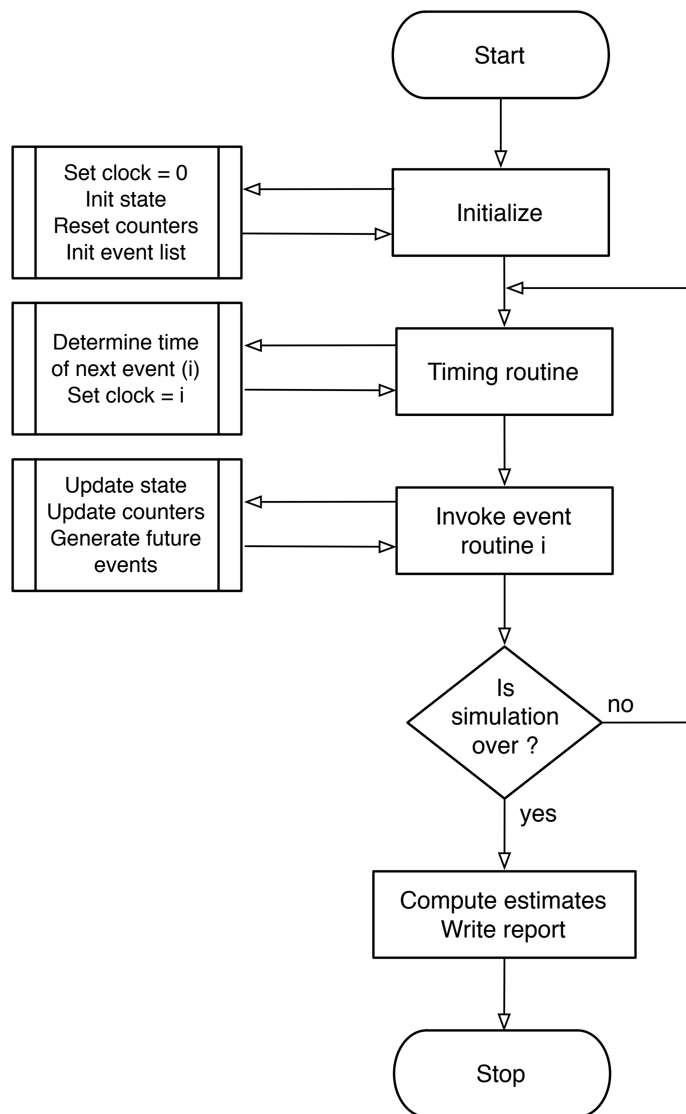


Figure 3.1.: Event-driven Flow Control (cf. [39])

sending of a SIP message. If these messages *arrive at times* $t_1, t_2...., t_j$*, the random variables* $\tau_j = t_j - t_{j-1}$ *are called the inter arrival times* [34]. It can further be assumed that all τ_j form a sequence of independent and identically distributed (IID) random variables. If the inter arrival times are IID and are exponentially distributed, then the arrival process is a so-called Poisson arrival process, or in other words (cf. [39]), the stochastic process $\{N(t), t \geq 0\}$ is said to be a Poisson process if:

1. Messages arrive one at a time.
2. The number of arrivals in the time interval $(t, t+s]$ $(N(t+s) - N(t))$ is independent of $\{N(u), 0 \leq u \leq t\}$
3. The distribution of $N(t+s) + N(t)$ is independent for all $t, s \geq 0$

Property one can be assumed valid, as it is highly unlikely that two messages arrive at exactly the same time. Property two states that the number of arrivals in the interval $(t, t+s]$ is independent of the number of arrivals in the earlier time interval $[0, t]$ and also of the times at which these arrivals occur [39]. Property three assumes that the arrival rate of messages does not depend on the time of day, and that is not given for most real-life systems. Though, if the time span that is chosen for performance analysis is short enough, the arrival rate has been found to be relatively constant and the Poisson process is a good model for this interval [39]. Therefore, the number of messages that arrive at a SIP system within a given time span, can generally be modeled by a Poisson arrival process, which is used for the research of this thesis.
When these messages arrive at a SIP proxy server it has to process them and this is called the service process. The time that each message is processed within that server is called the service time and it is common to assume that the service times are also IID random variables [34]. Most frequently used is the exponential distribution, but other distributions as Erlang are also possible.
Closing the circle back to event-based simulation, both, the message-generation (arrival) process and the service process generate events. The message-generation process takes Poisson-distributed random numbers from a pseudo random-number generator, considers these values to be time points when a new message has to be sent and inserts that as events into the event list. That process runs during the whole simulation time. The service process, though, is modeled differently. The simulator generates a “start” event when a message arrives at the processor. Following, the simulator takes a random number from a second pseudo-random number generator

and takes it as the time when the service process is finished and inserts a "finish" event into the event list. The next section describes the pseudo random numbers generation.

3.1.2. Random Number- and Variate-Generation

For simulation it is necessary to generate pseudo-random numbers that are distributed in various manners. A message arrival process for example can be represented by Poisson distributed numbers [39] and the service times of a service process by an exponential distribution. A simulator generates random numbers of a specific distribution in two steps: first a uniformly distributed sequence of random numbers in the range [0..1] is obtained (random-number generation) and then this sequence is transformed to the desired distribution (random-variate generation). A random-number generator should have the following properties according to [34]:

- Efficiently computable. As simulation typically needs a large number of random numbers, the generation should need less processor time.
- Large Period. If the period is too small, this would result in repeated event sequences.
- Generate independent and uniformly distributed values. Successive numbers should not correlate as correlation indicates dependence.

In the event-based simulator IBKsim, the Mersenne twister [40] has been implemented. This generator fulfills all of the above criteria. Its speed is according to the authors comparable to other modern generators and is very efficient. For a particular choice of parameters, the period of the generator is $2^{19937} - 1$ and thus large enough. It generates equidistributed random numbers of up to 32-bit accuracy.
For the random-variate generation, inverse transformation has been chosen for the IBKsim implementation. It is based upon the following property [9]. If F is a continuous distribution function on $\mathbb{R}$ with inverse F^{-1} defined by

$$F^{-1}(u) = inf\{x : F(x) = u, 0 < u < 1\} \tag{3.1}$$

and U is a uniform $[0, 1]$ random variable, then $F^{-1}(u)$ has the distribution function F. And, if X has the distribution function F, then $F(x)$ is uniformly distributed on $[0, 1]$. Using 3.1, any random variate can be generated provided that its inverse is explicitly

known. Understanding the exponential distribution is essentially throughout this thesis, so an example of generating exponentially distributed numbers is given for clarity purposes. The exponential distribution is defined by its density

$$f(x) = \lambda \cdot e^{-\lambda \cdot x}, x \geq 0 \tag{3.2}$$

and distribution

$$F(x) = 1 - e^{-\lambda \cdot x} \tag{3.3}$$

functions. Its inverse function calculates as

$$F^{-1}(U) = -\frac{1}{\lambda} log(U) \tag{3.4}$$

and can efficiently be used to generate exponentially distributed numbers using the Mersenne uniform random number generator. These random numbers can be used for example as time points when a new message has to be sent during the simulation. Multiple instances of different distributed generators can be used independently in the same simulation. During the simulation run, data is generated by the model that has to be written to files for later analysis. This is done by the logging module throughout the whole simulation and described in the following section.

3.1.3. Logging and Analysis of Simulation Data

Logging is an important task of simulation as it creates the output of the scenario that can later be used for analysis. Depending on the desired result, two methods can be used, periodic logging or one-time logging. If a single mean value as well as variance or standard deviation of some parameter is desired, the logger has to collect the values either periodically or on event basis, to calculate at the end the desired mean, and write it to a file. In order to keep the simulation effective in terms of memory usage, values can also be calculated cumulatively instead of keeping everything in memory until the simulation ends. On the other hand, if the chronological development of some value is the desired simulation output, the logger has to write the values periodically to a file. The latter method however, must be used very carefully, because the file size may grow significantly depending on the size of the simulation scenario and the number of nodes that log their parameters.

Collection and Calculation of Values

In any case, either periodic logging or one-time logging at the end of a simulation, the desired data needs to be collected. Depending on the type of value the simulation has to log, it must be reset to zero after each logging cycle correspondingly. That is, for each parameter to collect, it must be decided in advance (for example within the implementation of the logging methods) how its value has to be handled during the simulation run. For example the number of requests received within one logging interval must be reset to zero after each logging procedure. A second example is a time value for which all corresponding values have to be collected and then the mean value has to be computed (for example mean response time of all requests sent during the whole simulation run). That procedure of resetting values after logging cycles must be handled carefully as otherwise the simulation results are unusable or wrong.

Logging in Simulation

The simulation environment IBKsim is a discrete event-based simulator, which means, all tasks that have to be performed are handled within events. That means that also the logging procedures are executed as events by the simulation kernel. This routine enables implicit tracking of simulation time and therefore automatic logging at the desired points in time. One logging event must be created at the beginning of the simulation run and after executing all log work the next logging event is placed by the logger objects and executed after some pre-configured time.
After describing the theoretic concepts of event-based simulation, the following section explains how they are implemented in IBKsim practically.

3.2. Event-based Simulation Environment

The simulation tool IBKsim evolved out of the tool IKNsim [54] which has been designed to support studies about mobile agents. It has been used by many PhD students of the Institute of Broadband Communications at the Vienna University of Technology and has continuously been extended to be a more general network simulator. A detailed description goes beyond the scope of this thesis, however, this section highlights the concepts that have been necessary to implement SIP. Although it would have been possible to use another existing simulator, IBKsim offered most flexibility because the source code is fully available. It offers a very slim kernel and

lots of ready-to-use modules such as queues, servers, or links. In the following sections the main components of the simulator are described briefly.

3.2.1. Simulation Kernel

The simulation kernel is very slim and consists essentially out of four classes:

simControl dispatches the topmost event from the event list, sets the simulation time to the events time and executes it.

simQueue implements the event list and the necessary methods for dispatching the topmost event and adding new events to it with automatic sorting according to the time when it must be executed.

simEntity is the base class from which all simulation entities (for example a SIP UAC) derive.

simEvent encapsulates the event and all parameters that are potentially necessary for its execution.

The dependencies between the classes is shown in figure 3.2. The main class holds an object of the class simControl that has the init, loop and report methods implemented, and holds a reference to the simQueue object. Throughout the simulation, that is, the loop method, it requests the topmost simEvent from the simQueue object and thereby the corresponding simEntity object that implements the handleEvent method. This method is empty in the simEntity class and therefore virtual and has to be implemented in each specialized class. Figure 3.2 depicts an example *ibkSIPEntity* that is part of the SIP implementation. Objects of these classes are created and initialized upon startup of the *init* routine of the simulator. The simulation is started subsequently by invoking the *loop* method of simControl. Based on the simEntity class, simple components as queues and servers are already implemented in the simulator and can be used by more complex simulations like larger networks or specific protocols.

3.2.2. Network and Protocols

Entities that are necessary for the implementation of specific protocols, like for example SIP are derived from the simEntity class. By doing so it is assured that new

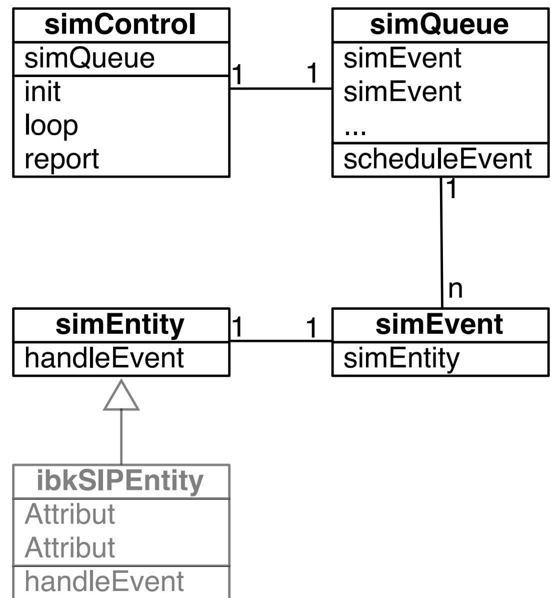


Figure 3.2.: IBKsim Kernel

protocol modules have a common interface with the simulation kernel. An addressing scheme has been developed and implemented that uniquely identifies each element in the network. Protocols that are implemented on top can make use of this scheme, but can also implement their own addressing scheme. This layered structure is a benefit because it is similar to real-world protocol stacks and allows a flexible choice of abstraction. SIP, for example could be implemented directly on top of the IBKsim addressing scheme or, after implementing IP and UDP, on top of that.

3.2.3. Random Number Generator

The random number generator is implemented conferring to the theoretical concepts presented in section 3.1.2. It is divided into two steps: generating equidistributed random numbers and transforming these into the desired distribution by means of reverse transformation.

3.2.4. Logging

Figure 3.3 depicts how the logging procedure is implemented in IBKsim. All objects that log their values must implement the getLogs method that is called by the logger. The logger object is created in advance to the simulation and holds a list of all objects

that are configured to log. Beside other events, the kernel has in its event list the first and initial log event that is placed there by the logger itself. When the kernel executes the log (1) event, it calls the getLog method of the logger (2) that loops through all logObjects in its list by calling their respective getLogs method (3). After finishing all logging methods, the logger places the next logging event onto the event list of the kernel (4). In order to get precise simulation output, different logging

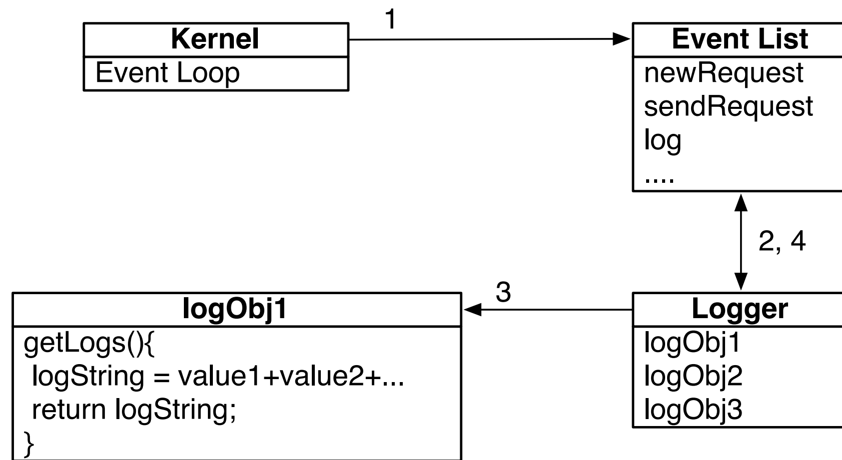


Figure 3.3.: Logging Procedure

strategies have to be applied depending on the type of value to log. One example is the success rate of SIP transactions (see also 2.1.3). The success or failure of a transaction can be determined as soon as a response is received or a time-out event occurs. The duration until this happens is unknown in advance and the success or failure of a specific transaction needs to be assigned to the time point when it was initially created and therefore the logger has to wait until either a response arrives or a time-out occurs. At the end of the simulation, the success state can be assigned to the corresponding time frame and the success rate can be calculated. Logging the processor utilization on the other hand is more simple: it is only necessary to measure the time the processor is utilized in the last logging period and divide it by the duration of the logging period. No collection of values and no post-processing is necessary for this type of value.
IBKsim logs the the QoSg values (see section 2.1.3) in the way described in the next sections.

Final Response Delay

This value can not be calculated until a transaction has been either successful terminated, or not. After a single transaction terminates, the fRpD (section 2.1.3) is calculated as

$$\delta_f = t_r - t_s \tag{3.5}$$

where δ_f represents the fRpD, t_r is the simulation time when the first final response has been received by the UAC, and t_s is the time when the UAC has sent the first initial request. It is obvious, that this value can not be calculated until a final response has been received and therefore the logger has to wait until then. For later analysis, the value is assigned to t_s, that is the fRpD of a specific transaction that has been started at this time point. At the end of the simulation, the logger has a two dimensional list containing fRpD values mapped to the chosen t_s, that can directly be plotted or used for calculations. A large number of values are created during the simulation run, which is unpractical. One way to deal with this efficiently is, to sort the values into bins, that represent a short time interval. That can be done in the following way: denote the whole simulation duration as D and the width of one bin as w_b. Then the number of bins n_b is

$$n_b = \frac{D}{w_b} \tag{3.6}$$

and the time interval $[t_{lower,i}, t_{upper,i})$ for each bin $i, i \leq n_b$ is given by

$$t_{l,0} = t(0) \tag{3.7}$$

$$t_{l,i} = t_{u,i-1} \tag{3.8}$$

$$t_{u,i} = t_{l,i} + w_b \tag{3.9}$$

whereas $t(0)$ is the time point when the logging during the simulation starts. Then each logged fRpD value $D_f(t)$ at time t corresponds to bin $i, i \leq n_b$ if

$$t_{l,i} \leq t < t_{u,i} \tag{3.10}$$

The simulator can then calculate statistics for each bin and report these values rather than all single values. The advantage of this technique is that it uses less disk space depending on the simulation, while not losing any degree of significant accuracy.

Success Rate

The calculation of the success rate of transactions follows the same procedure as for the fRpD. A transaction is successful, if it has received a final response message and failed if it timed out and did not receive a final response. At the end of a simulation run, the logger creates bins as described earlier and counts the number of successful and failed transactions which started (sent their first initial request) within the time interval the respective bin represents. The success rate λ_s calculates as

$$\lambda_s = \frac{n_s}{n_s + n_f} \tag{3.11}$$

where n_s is the number of successful transactions within the bin and n_f is the number of failed ones.

Total Number of Transmissions

The number of transmissions calculates as

$$n_t = 1 + n_r \tag{3.12}$$

that is the initial request (1) plus n_r, the number of retransmissions that have been sent until the transaction has received a final response. As with the success rate, these numbers are sorted into bins of width w_b.

Processor Utilization

To calculate the processor utilization, the logger needs to track the time the processor is executing some service during log periods (service time). This time in relation to the log period equals the utilization and is calculated in two different ways. If the processor is currently (i.e. whilst logging) executing a service the service time t_s calculates as

$$t_s = t_t + (t(c) - t(s)) \tag{3.13}$$

where t_t equals the total time the processor executed a service during the current log period, $t(c)$ is the current simulation time and $t(s)$ is the start time of the service the processor is currently executing. If the processor is not executing a service at

present, the service time equals the total service time:

$$t_s = t_t \tag{3.14}$$

Although, there are various other values that could be logged during a simulation, the ones described above have shown to be most important for performance analysis for SIP.
This section has explained the basic concepts of the event-based simulation tool IBKsim, its kernel structure, and how protocols can be implemented. Further it has described how the reporting, or logging for three major metrics are implemented. The next section describes the development of SIP entity models.

3.3. Model Creation for SIP based Networks

The simulation tool IBKsim has been extended with SIP modules, which provide functionality similar to a regular SIP client and proxy, as well as modules for queueing inside SIP proxies, which are used by a proxy module to simulate processing and database access. The modules for SIP clients consist of a stateful UA which implements a UAC that generates new messages according to a Poisson process and a UAS that responds to received messages. The proxy module implements a transaction stateful proxy according to RFC3261 [53]. The user agent and proxy modules hold a list of corresponding client and server transactions that belong to them. SIP message processing, including the not exhaustive list of: creation of new messages, sending over a link, receiving, processing, queueing, responding and transaction timer handling is implemented according to event-based simulation. This means that a simulation kernel executes events that have been created by the various entities involved in the SIP simulation.
Sections 3.3.1 and 3.3.2 explain the simulation models of UA and proxy in more detail, section 3.3.3 the simulation of links, before section 3.3.4 explains briefly SIP transactions. Finally, section 3.3.5 shows representative SIP services that are later used for performance analysis.

3.3.1. User Agent

A UA is the entity that sends or receives messages on behalf of the user in order to either set-up a multimedia call or to transmit non session-based information such as

an instant message. It can either play the role of a UAC and generate new requests or UAS and answer to received request messages or both, according to [53]. This is obviously also valid for simulated UAs, but, for simplicity reasons, several UAs can be aggregated into one (in order to keep the simulation scenario small).

User Agent Client

A high number of independent clients that create new messages at a low rate can be modeled as Poisson process (see also section 3.1.1). A single instance of a simulated client can therefore represent a high number of clients in a real environment and if these clients all together send messages at an aggregated rate λ, a single instance of the simulated client can be used to generate new messages at the same rate λ. Inter-arrival times are distributed negative exponentially according to a Poisson process [43]. The simulator uses a pseudo random number generator to produce uniformly distributed numbers and applies reverse probability integral transformation ([38] and section 3.1.2) to calculate negative exponential distributed random numbers (equation 3.15):

$$t_u = -\frac{1}{\lambda} \cdot log(u) \tag{3.15}$$

The variable u hereby represents a number generated by a uniform distributed random number generator in the range (0..1] and t_u is the time interval between the newly generated messages. New messages in terms of SIP means SIP requests that the simulation generates at a specific rate. Depending on the service which has to be analyzed (for example instant message or presence), the client generates different types of messages (for example MESSAGE or NOTIFY) as well as a distinct number of messages which are necessary for the specific service (see section 2.1.1). Within the implementation for the SIP extension for IBKsim, a transaction manager (TM) has been implemented that handles all newly generated messages and all incoming messages as depicted in figure 3.4. For each request that the TM receives, it looks up in the list of client transactions if there already exists a corresponding one. In case it finds one, it assigns the request to it or, otherwise creates a new client transaction. For each response that arrives at the TM and has a corresponding client transaction the TM assigns that response to the corresponding client transaction. If the response does not have a corresponding client transaction, the message is discarded. The client transactions then have to process the messages according to the client state machine and thus are responsible for all re-transmissions and all other timer events.

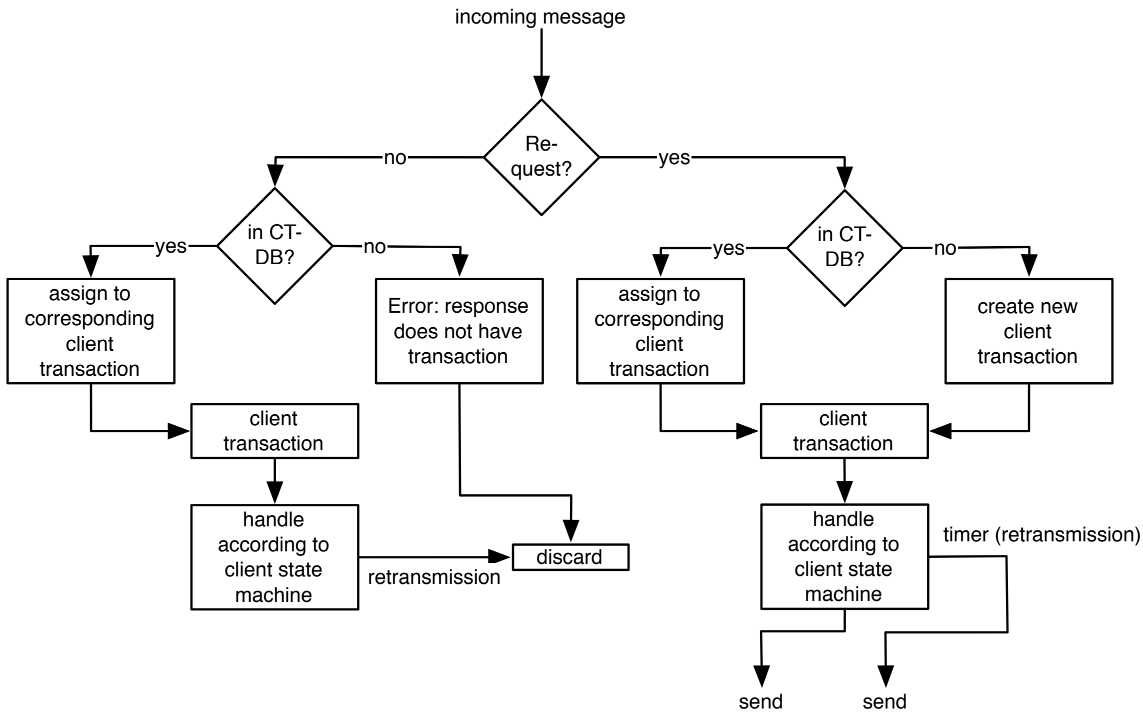


Figure 3.4.: User Agent Client Transaction Manager

User Agent Server

The UAS simulation receives requests and responds to them according to their content. In terms of SIP, the UAS is the entity that receives information or service requests and displays it to the user. In terms of simulation and for performance analysis, it saves logging information if required, and responds to the received requests. As for the UAC, a high number of UAS in a real system are aggregated into a single one for the simulation. Figure 3.5 depicts the transaction manager of the UAS. Incoming requests are assigned to a server transaction (ST) if it already exists, or a new ST is created, which processes the request accordingly. If the transaction manager receives a response it retrieves the corresponding ST from the ST database (ST-DB).

These client and server transaction managers ensure that the messages are treated according to RFC3261 [53] and the implementations of client and server transactions correspond to the respective state machine.

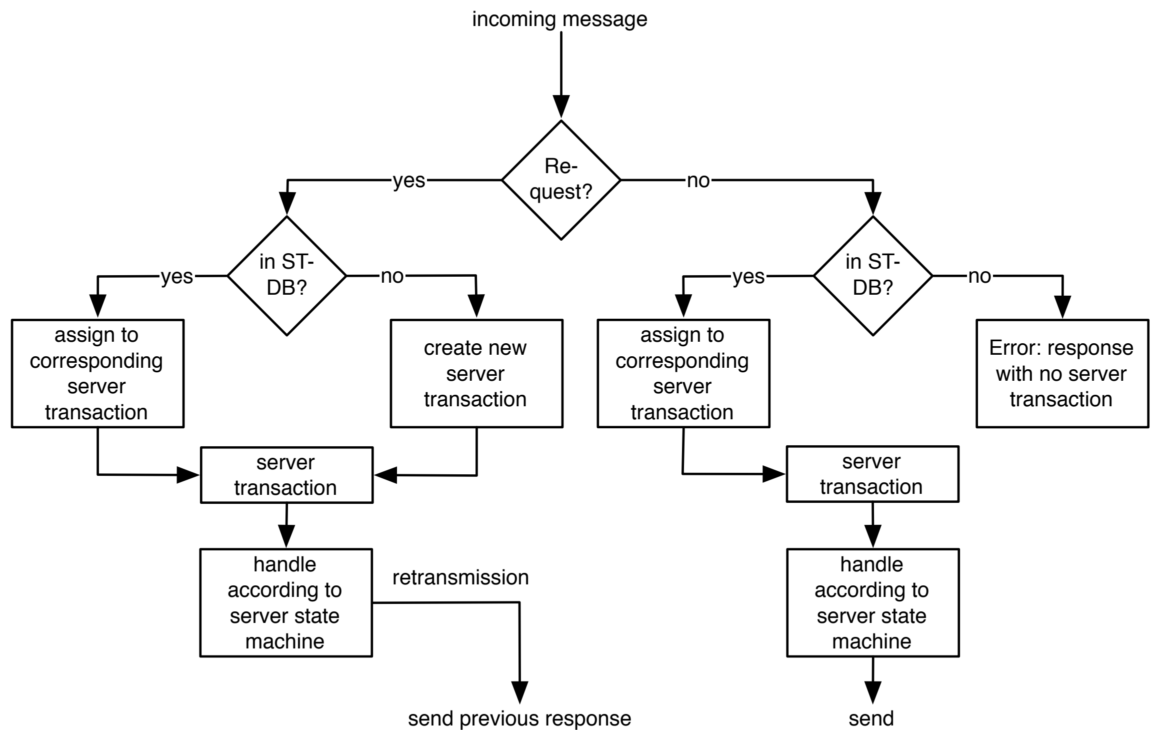


Figure 3.5.: User Agent Server Transaction Manager

3.3.2. Proxy

SIP [53] differentiates between stateful and stateless proxy servers. Stateful proxy servers create a server transaction for each received request and a client transaction for each request to forward. As a result they themselves are responsible for re-transmissions. Stateless proxy servers on the other hand do not create server and client transactions at all, but simply forward received messages towards their destination. This means that stateless servers are not able to fork requests or to distinguish between original requests and retransmissions. This thesis focuses on performance evaluations for stateful servers, since stateless servers are basically simple forwarders with not much potential for performance improvements. However, stateful servers can further be broken down to *call* stateful servers, which keep state for a dialog from the initial INVITE request until the final BYE request, and *transaction* stateful servers that keep state of transactions but not of dialogs. It is inherent that call stateful servers are also always transaction stateful. The models created for this thesis base on models developed by [57] and are explained in the following sections.

Scheduling in SIP Proxies

Simple models of a proxy server (for example section 3.3.1 of [57] and depicted in figure 3.6) assume that each message, independent of type and content, will be queued in a First-Come-First-Served (FCFS) queue. After waiting some time (until earlier arrived messages are processed) they are processed by the processor. A more realistic

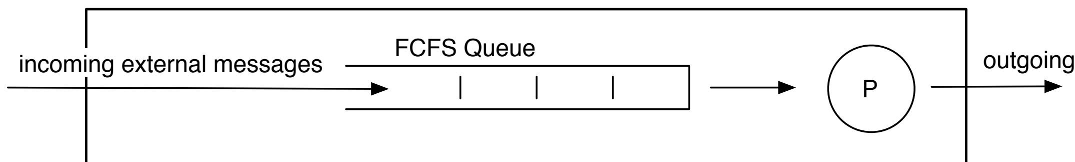


Figure 3.6.: First-Come-First-Served Model

model of a proxy server has to take into account, that the processing of a message can be divided into two parts: *parsing* and *routing*. For parsing, the text based SIP messages are syntactically analyzed, broken down into parts, and converted into internal representations as first step. Second, the newly created internal message representations have to be analyzed to infer on their later processing. For a SIP proxy server this means that it has to determine the messages destination and forward them towards there, which is known as routing. For a UAS it means extracting the content of the message, probably displaying something to the user and creating and sending responses. Each of the two parts of processing a SIP message in a proxy server uses a specific part of processing time. According to [57] it is assumed that parsing of a message always takes approximately the same amount of processing time, independent of its type and content. The simulation model distinguishes requests and responses and assumes further (according to [28]) that a response needs less processing time than a request. These assumptions lead to the proxy simulation model depicted in figure 3.7. Within this model, each message type refers to one of the four specific queues which are served by the processor P, according to priority scheduling (that means that a message from the FCFS queue with the highest priority is always taken first and forwarded to the processor; if there are no messages in the first queue left, a message with the next lower priority is taken, and so on). The following priorities are given to the message types:

1. Incoming (unparsed) external messages

2. Self-created re-transmissions

3. Parsed responses

4. Parsed requests

Note that the author deliberately assumes that message reception and parsing is of highest priority, as it leads to the creation of the basic internal message representations in the memory. This model is based on an interrupt-driven system and therefore the network interface card triggers an interrupt upon reception of a message. That means further, the normal operation (that is SIP routing in terms of the simulation model) is interrupted as often as a new SIP message is received. After creating these structures, each message is again queued for routing until it is processed (routed). Within the routing process, the simulator creates transactions for each new request and they set timers which possibly create retransmissions of requests which will then be sorted into the re-transmission queue.

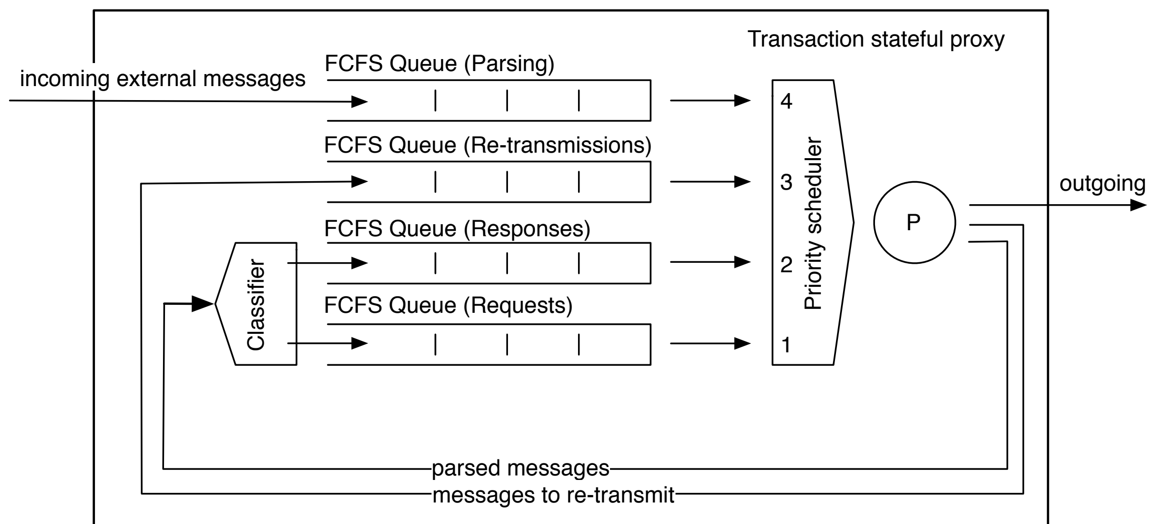


Figure 3.7.: Request/Response distinguishing Model

Advanced Simulation Model

The simulation model for the proxy is depicted in detail in figure 3.8. Basically, it distinguishes a queueing Model (upper part) and a transaction manager (lower part) as main components. The queues for self-created retransmissions and already parsed requests and responses are following in terms of decreasing priority. Upon the processing of a message, the SIP-specific transaction manager creates or deals with

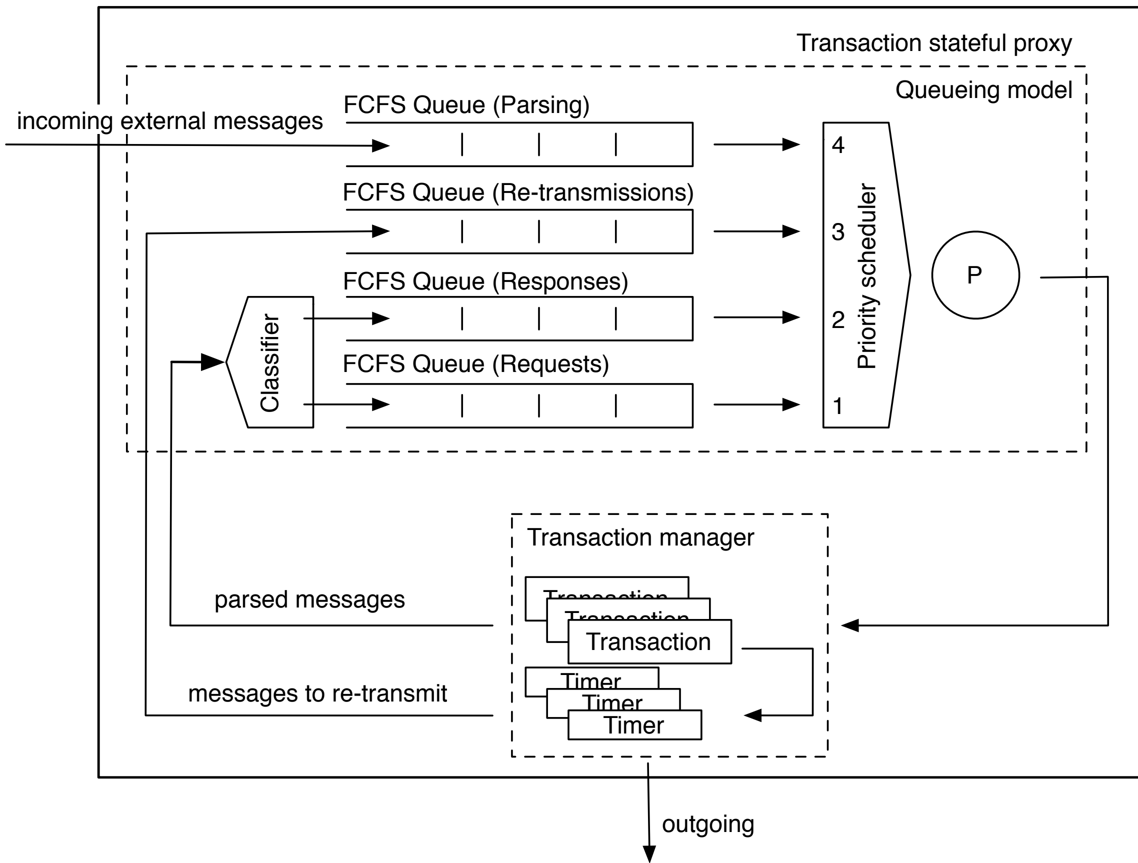


Figure 3.8.: Advanced Proxy Simulation Model

the corresponding state machine and returns the message into the queuing model for a second time, either as a re-transmission, request or response message. According to [28], a response needs less processing time than a request. In our model, the processor P simulates both parsing and routing by applying a certain processing delay to each message. The sum of these processing times eventually represents the output of the simulation model.

By partitioning the model this way, it is possible to change parameters of each part independently. SIP parameters, for example re-transmission timers can be altered but the processing of messages happens in the same manner. Another example is the ratio of processing time that is needed for parsing and routing that can be modified independent of the SIP implementation in the simulator.

3.3.3. Link

The type of link between SIP entities plays a major role when evaluating overall performance. Observe as example the difference between a narrow-band wireless connection in comparison to a wide-band Digital Subscriber Line (DSL). This thesis abstracts from link-specific properties and therefore uses stochastic distributions to simulate loss and delay. The result is a more precise output that is not influenced by specific behavior of a lower layer. The link implementation uses two independent random number generators to calculate the delay and loss probability for each individual packet. The generator for the loss uses a uniform distribution in the range [0..1] and discards a packet if the generated number n_g is larger than the pre-configured loss rate n_c:

$$\begin{aligned} l = 0 &: n_g \leq n_c \\ l = 1 &: n_g > n_c \end{aligned} \tag{3.16}$$

The number l corresponds in the implementation to a Boolean value that is directly used to discard a packet. Because the generated numbers are uniformly distributed, that procedure leads to an overall loss rate of n_c, which has been configured during the set-up of the simulation scenario.
In order to focus on the effects of processing delay rather than specific link behavior, the values that the generator for delay delivers are deterministic, that is, no link delay variation is generated and the values are constant.

3.3.4. SIP Messages and Transactions

Having surveyed the topmost components of SIP communications, this section turns into a more detailed view on SIP transactions, which are the basic building blocks of SIP-based communication. Any entity sending a SIP request (except stateless proxies) creates a transaction for each request sent or received. Generally two basic types can be distinguished: non-INVITE transactions where no direct human interaction is necessary (for example delivering an instant message) and INVITE transactions where direct human interaction is normally required (for example a voice call). Human interaction always introduces some delay in comparison to direct machine-to-machine communication. For example when the phone rings a response message is not sent until the user picks up the phone. For this reason the distinction into non-INVITE and INVITE transaction has been created and additionally a separate transaction state machine for each of the two.

A transaction always consists of an initial request and one or more responses to this request, including provisional and final responses [53]. As SIP transactions are used for communication, they consist always of a client part which is responsible for sending requests and receiving responses and a server part which receives requests and sends responses to these requests. An UA always implements both and therefore can send new requests on the one hand (client part, e.g., make a call) and receive requests (server part, e.g., receive an instant message) on the other hand. Proxies (except stateless proxies) that forward a received request create a separate server transaction which handles requests from the originator of the initial request and a client transaction which sends the request to the next downstream host. Figure 3.9 shows an example where the UAC executes a client transaction that sends a request to its proxy that has a corresponding server transaction. The proxy always executes a client transaction that forwards the request to the UAS's server transaction.

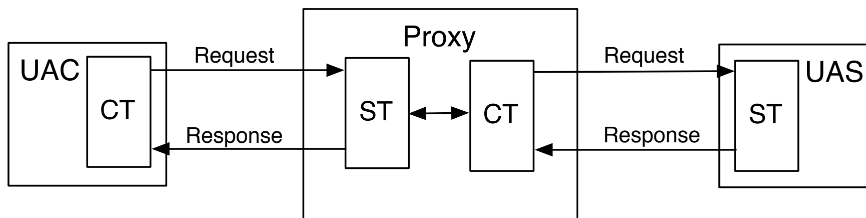


Figure 3.9.: Transaction Relationships (cf. [53])

This thesis focuses on non-INVITE transactions and assumes that the number of such messages is much higher in a provider's network because a large variety of services (for example instant messaging, presence, registration) use them in contrast to INVITE transactions that are currently only used by one service (voice and video call). Hence non-INVITE transactions are more attractive for performance analysis than INVITE transactions.

State Machines

As introduced in section 2.1.2, transactions are handled completely by state machines. [53] describes four different types of transactions and thus also four different state machines. As non-INVITE transactions are used by many more services, they are more suitable for performance evaluations and the only ones considered for this thesis. Figure 3.10 depicts how a new request is sent and how the components interact. When the simulation triggers sending of a new request, a newRequest event is dispatched

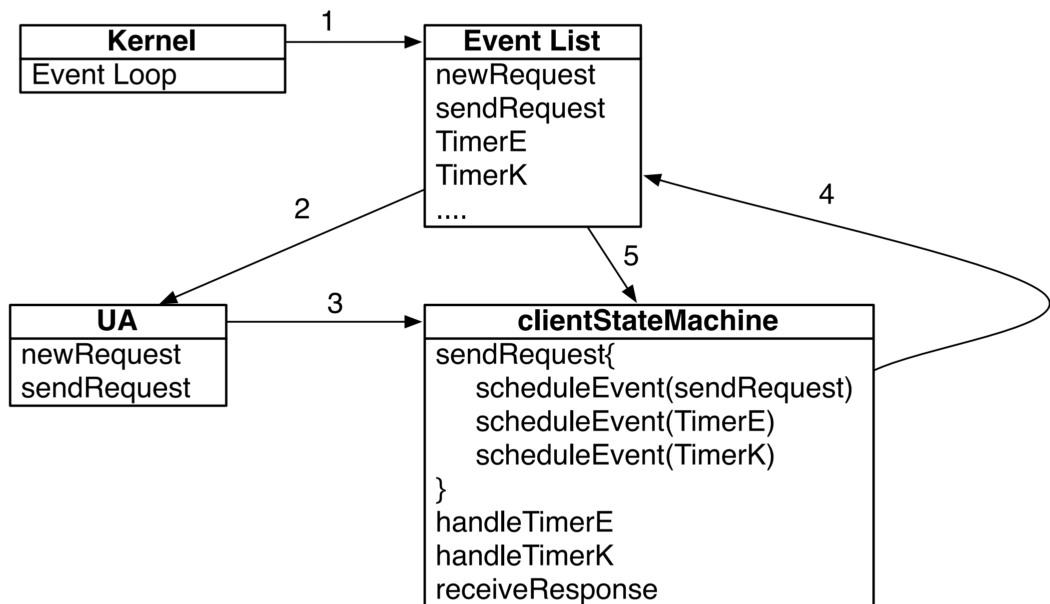


Figure 3.10.: Simulation environment: sending new request

from the event list by the kernel (1). That invokes the *newRequest* method of the UA (2), which creates a new client state machine and handles the data of a newly created request to it (3). The state machine then has to create a "send" event and submit it to the event list (4). Events for SIP timers are also submitted at the same time. Later, the simulation kernel dispatches the sending event and calls the respective method of the link object which transmits the message to the next SIP entity.

3.3.5. SIP Services

The value of SIP is created by a vast number of possible services which can be applied using SIP and it is therefore interesting for simulation and performance evaluation to analyze these contrary to simple protocol performance. The following sections describe the concepts of the services which have been chosen for this thesis and their simulation implementation.

Instant Message

Instant messaging, also known as Short Message Service (SMS) is certainly the most successful service realized in current GSM and UMTS networks as SMS and it will likely remain important for future mobile networks. It is therefore interesting to analyze its performance by means of simulation.

This service is realized by the SIP MESSAGE request, shown in figure 3.11. The instant message to deliver to a UAS is contained within the body of the request, the response acknowledges the reception of the request. Optionally, one or multiple re-transmissions can happen, according to the non-INVITE state machine. For per-

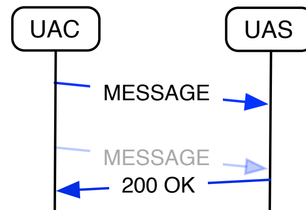


Figure 3.11.: Message Service

formance evaluation the simulation logs the fRpD and the SR of transactions (i.e. transactions that receive a final response). The instant message service uses one MESSAGE request to deliver one instant message and could also be used as background traffic to test other services or to analyze the performance of proxies during occurrence of a high number of messages during special events (e.g., new year or television voting events).
To simulate this service an aggregated UAC as sending part and an aggregated UAS as receiver is necessary.

Registration

A client registers itself at a registrar in order to become reachable via its generic SIP URI. The registrar usually includes a authentication with a challenge-response algorithm that is depicted in figure 3.12. Within the first "401 Unauthorized" response,

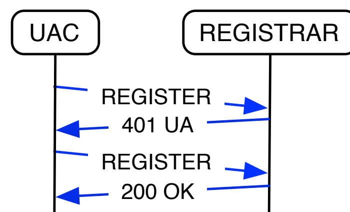


Figure 3.12.: Register

the registrar includes a "nonce" (number used once) value that is used by the UAC to encrypt its credentials. The encrypted credentials are sent with the second Register message to the registrar which answers with a 200 OK response if the credentials are

correct. This response includes an expires header to inform the client when it must re-register latest to keep the mapping active.
Relevant for performance analysis point of view is a scenario, where some part of the network fails (e.g., part of the access network or an edge proxy) and after recovery all affected UAs try to register themselves at the registrar within a short time span. Interesting measures are also the fRpd as well as the number of successful registrations. A registration is successful if the corresponding client transaction has received a 200 OK response.

Presence

The SIP presence service is based on the event notification framework extension for SIP [50] and is extended by an event state publication mechanism [44]. The purpose of this service is to distribute information about the state of a user to his buddy list. One example is the line-state of a user's phone such that other users know if a call is possible before they place it.
To provide a presence service using SIP two options can be used:

- point-to-point
- server-based

For point-to-point presence all UAs address their presence messages directly to the other users' UAs like standardized within [50]. Figure 3.13 shows a message flow for one UA (watcher) that watches the state of three other UAs (presentities). The

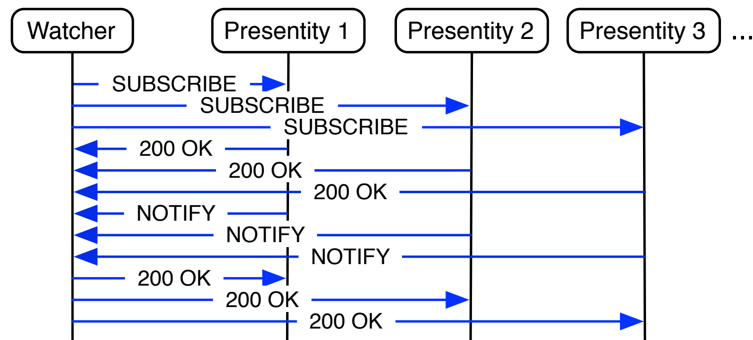


Figure 3.13.: Presence Service

implementation of the presence service using this technique creates much traffic if presentities are watched by multiple watchers.

To simulate this service, watcher and presentity have to be created using the UAC/UAS implementations described earlier and multiple instances of them must communicate through a proxy server. A relevant performance metric for this scenario is the number of watchers that manage to retrieve the state of the presentities in their buddy list (number of active subscriptions) over a highly loaded system.

4. Performance Improvement Strategies

It is well known that SIP lacks a suitable congestion protection mechanism [45] as the standard proposes only a response code that informs that the proxy is currently congested and additionally allows to add a “Retry After” header, which defines a time interval in which no further requests should be sent. Unfortunately, this solution does not work in several scenarios [51]. Therefore the standardization currently specifies extensions for SIP to transport congestion state information through the SIP network. However, the congestion notification is realized either explicitly by sending dedicated messages [17] or by adding a chunk of data to SIP messages [21] that describes the current congestion state.

The following sections present a new mechanism that is based on information that is present implicitly and already available in each SIP protocol stack. This mechanism can be divided into two parts: *congestion detection* and *congestion handling*. Congestion detection collects information such as the delay between sending a request and receiving the corresponding response or the number of newly sent requests in the past second in order to detect a possible congestion situation. Congestion handling takes the information of the detection part and applies some method to reduce the outgoing load that caused that congestion. In the case of SIP, congestion handling can be applied either in a lossless or a lossy mode. The lossless mode is done for example by increasing the timer that is responsible for retransmitting a request and therefore giving the congested proxy server enough time to process the backlog, by reducing a huge and unnecessary load, created by retransmissions. No new incoming requests have to be rejected by this mode and thus the revenue for the operator is not decreased. In many cases this method may be sufficient and the congestion situation disappears. In the case it does not disappear, the nominal SIP load exceeds the node’s or network’s capacity and lossy modes can be applied to solve the congestion. These reduce the outgoing load by rejecting a part of new incoming requests. However, as

this method rejects service requests sent by customers, the operator's revenue suffers from it. These two methods enhance the performance of a SIP proxy server either by applying minor changes to the protocol or by adding implicit sensing mechanisms to detect an already present congestion and handle it accordingly. However, these methods do not change the implementation of the SIP proxy server itself. Optimizing that may enhance the performance even more. A promising approach is to improve the parsing part of a SIP proxy server, the module responsible for reading incoming SIP messages and creating internal data structures that are later used to analyze the message. As SIP is a text-based protocol, the processing requirements of this part is not negligible [63] and offloading it to some external processor can improve the performance significantly. This is for example also done for TCP where processing intensive operations such as checksum calculations are offloaded to the network interface card [7].
The current section is organized as follows. Section 4.1 verifies current known performance drawbacks of SIP. Section 4.2 describes methods to detect possible congestion situations implicitly, and section 4.3 how congestion situations can be handled. Section 4.4 presents an idea on offloading some parts of processing of incoming SIP requests to an external processor. Finally, section 4.5 gives some application scenarios.
It must be mentioned here that the results of this section are published within the papers [16], [14], [15] and [13].

4.1. SIP Performance Drawbacks Verification

In order to develop performance improvements for SIP, the performance of the current standard has to be analyzed and documented. To this end, we resort to a dedicated discrete event-based simulator, the so-called IBKsim [60] implemented at the Vienna University of Technology for Linux in C++. In order to obtain the results presented in the following sections, the current version of IBKsim has been complemented with an implementation of SIP UAC and UAS non-INVITE transaction state machines as well as a model of a transaction stateful proxy server as described section 3.3. The following section 4.1.1 analyzes problems that occur because of SIP's standardized retransmission algorithm when a system comes into high-load and overload situations, before section 4.1.2 gives a more detailed analysis of the behavior of SIP nodes in various load situations.

4.1.1. Drawbacks of SIP Retransmissions

This section explains which problems can be caused by the retransmission algorithm of SIP [53] in certain high-load situations. As mentioned, SIP differentiates mainly between two transaction types: INVITE transactions and non-INVITE transactions. INVITE transactions are used to create sessions, where the initial request carries a session description for agreeing on a set of compatible media types. Non-INVITE transactions, on the other hand, carry information that is usually not session-based, such as a registration message or an instant message. In both cases, special network nodes, the so-called SIP proxy servers, enable the message routing, know the users current location, and authenticate and authorize users for specific services.

As far as the underlying transport protocol is concerned, SIP can employ the reliable TCP as well as the unreliable UDP. In the latter case, SIP achieves reliable message transmission by means of re-transmission of messages, which is controlled by the retransmission timer E. Following the standard, a single request may be transmitted up to eleven times before the transmission attempts are terminated. The authors of SIP have specified this behavior in order to cope with the unreliable nature of Internet traffic. As a side effect, however, retransmissions are also triggered if messages are not lost, but delayed because of congestion (final response delay longer than SIP timer T1). Processing these retransmissions at a server creates additional congestion. As we will demonstrate later, in a congested system there is a high probability that requests never reach the destination. Moreover, even if the request reaches the destination, the response might fail to reach the originating node. Thus, in the worst case, the system has to handle up to eleven SIP requests and up to eleven responses without any other effect than using processing power of the nodes and flooding the links with redundant data.

The following section describe a plausible scenario that leads to congestion and in the worst case to a congestion collapse.

Scenario

An elementary scenario has been chosen to demonstrate SIP congestion behavior. Many user agents are aggregated into a single one for simplicity reasons, and one proxy server handles all messages for these clients (see figure 4.1). The server has been configured to be able to handle up to 100 SIP transactions per second. Hence, the time required by the server to handle one SIP message is negative exponentially

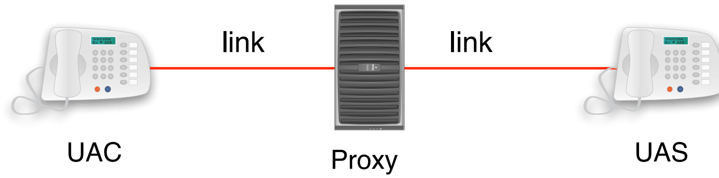


Figure 4.1.: Simulation Scenario

distributed with a mean of 0.01 seconds. The link between the nodes and the proxy server is characterized by 0.3 ms delay and a packet loss rate of 0% in order to focus on the effects of processing delay rather than link loss. A background arrival rate of 80 SIP requests per second simulates a processor utilization on the proxy server of about 80%. At $t = 50$, 40 additional SIP requests per second for 10 seconds duration simulate a short rate peak that causes congestion in the system. The simulation run logs the following values on a per-second granularity:

- Processor utilization at the proxy server (0 to 1)
- Success Rate (SR) of SIP transactions at the UAC
- Number of Request Transmissions (RT) for all transactions, including unsuccessful transactions at the UAC
- Final Response Delay (fRpD) at the UAC
- Pending Transactions (PT) at the UAC

A client transaction is denoted as pending as long as no final response has been received, that is, as long as the corresponding state machine (depicted in figure 2.6) is in the state *Trying*. The simulation run lasts for 1000 (simulation-)seconds.

Problem Description

Figure 4.2 shows the number of new transactions as well as the number of retransmissions for 400 simulated seconds. At t=50, an overloading peak of 120 new transactions per second is introduced for a duration of 10 s. This short peak leads to a situation where many user agents re-transmit their requests according to the SIP standard because no response is received in time and they assume the messages lost. This leads to a huge increase of the load that is injected to the proxy server. After reducing the arrival rate to a value below the capacity limit of the proxy server

the user agents still re-transmit many requests, introducing unnecessary load to the system. Such a situation is also called congestion collapse and is for example also illustrated in [28]. A SIP system's congestion collapse is a severe situation because

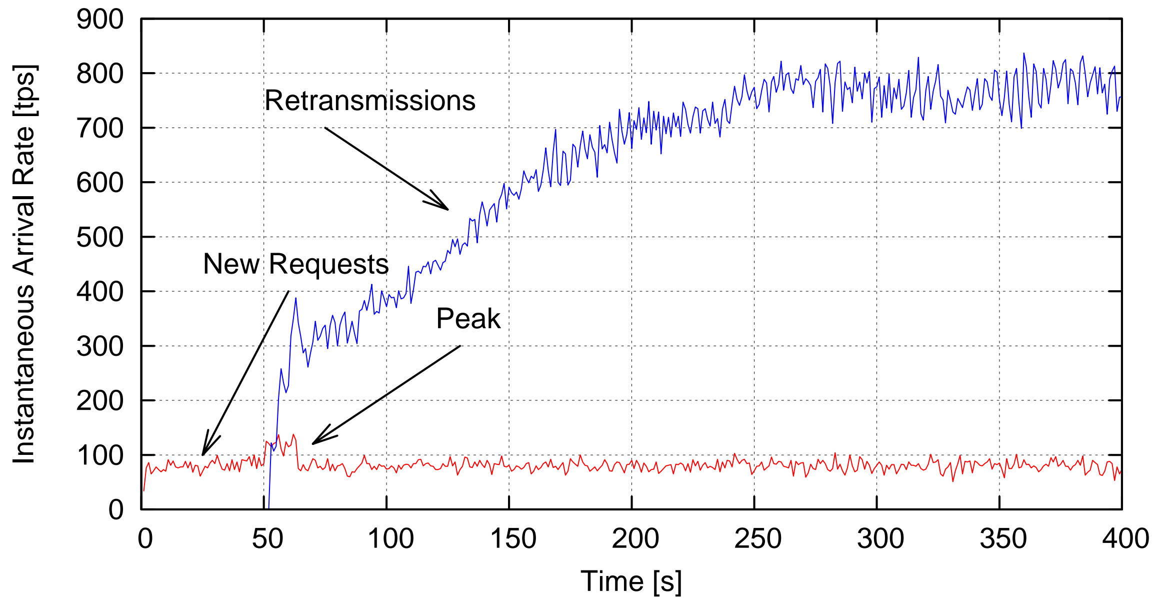


Figure 4.2.: Arrival Rate for new Requests (red) and Re-transmissions (blue)

the involved entities cannot recover by themselves and a considerable amount of input reduction is necessary in order to recover. As the goal of this thesis is to develop performance improvements for SIP it is essential that a measurement metric is able to indicate such effects. The following sections introduce the QoSg values Success Rate (SR), Request Transmits (RT) and Final Response Delay (fRpD) and the processor utilization of the proxy server and the number of Pending Transactions (PT).

Success Rate

Figure 4.3 shows the success rate (SR) for the above configuration. Following the short peak, the system runs into a congestion collapse, and only a few transactions are handled successfully. The rest times out.

Total Number of Transmissions per Transaction

Figure 4.4 depicts the transmission count per transaction until the UAC receives the first final response. It demonstrates that the system is flooded by retransmissions and that after about 70 seconds, all requests are sent multiple times. This figure

highlights the huge amount of redundant messages that are sent to the proxy server, compared to normal operation where all requests are just sent a single time.

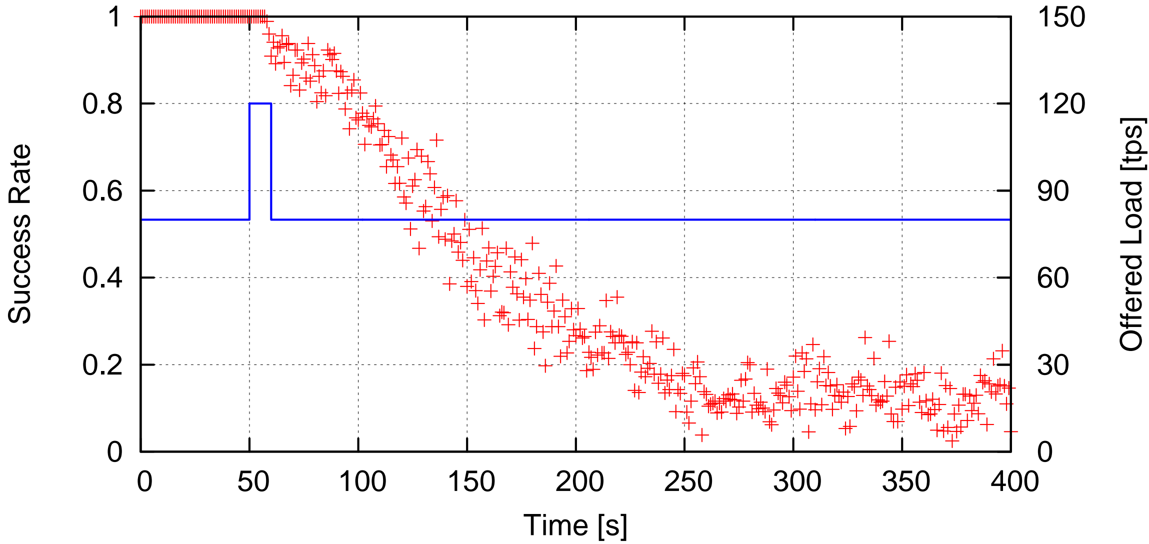


Figure 4.3.: Success Rate (red crosses) and Offered Load (blue line)

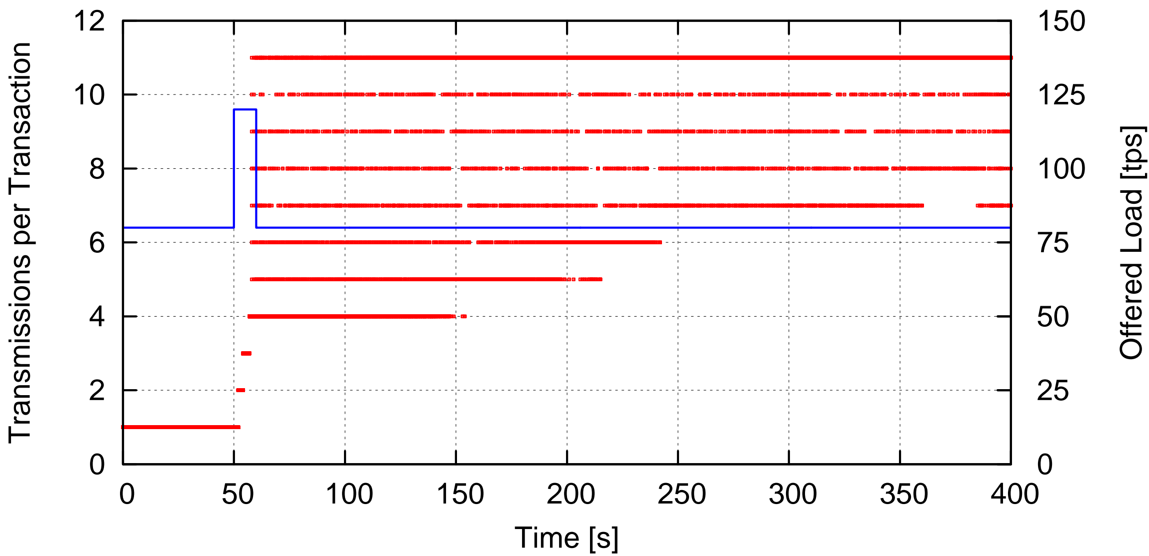


Figure 4.4.: Transmissions per Transaction (red dots) and Offered Load (blue line)

Final Response Delay

Figure 4.5 shows the final Response Delay (fRpD) in seconds in a scatter plot, where each dot corresponds to a single transaction. Observe that the fRpD increases up to

the maximum time of 31.5 seconds. Due to the static re-transmission timer configuration of the UAC, the dots merge to lines and each of these lines corresponds to a single re-transmission instance. This means that the lowest line belongs to transactions that have received a response to the first transmission, the second lowest line to transactions that have received a response to the first retransmission and so on. This supports the assumption that re-transmissions are unnecessary in situations where a proxy server is congested.

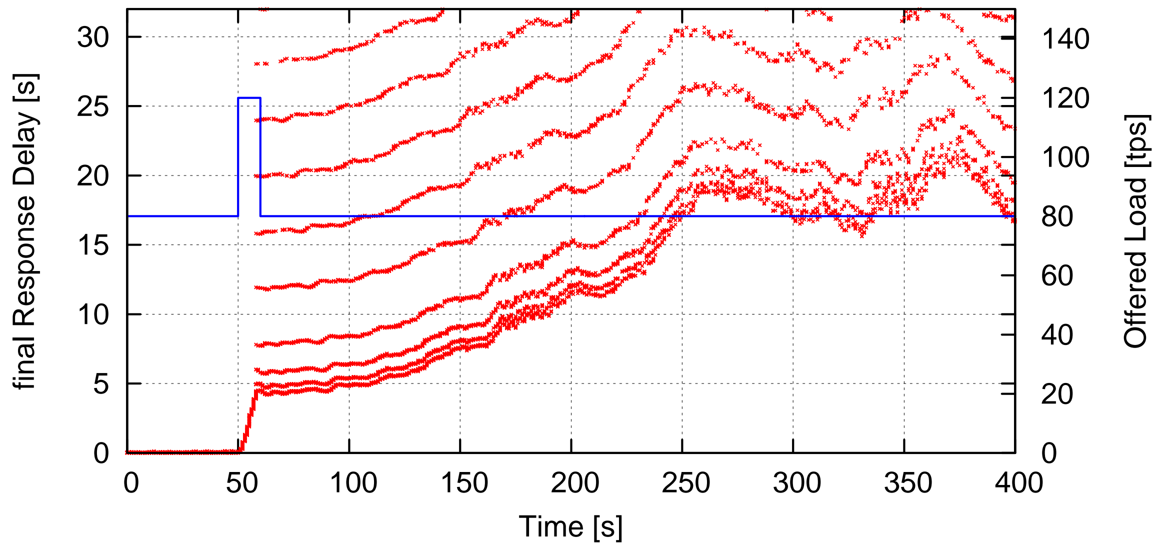


Figure 4.5.: Final Response Delay (red dots) and Offered Load (blue line)

Proxy Processor Utilization

Figure 4.6 shows the processor utilization. The processor of the proxy throughout remains at 100% utilization after the injected peak load. It is obvious that the proxy server is processing re-transmissions and therefore not able to handle all incoming messages.

Collapse Analysis

To analyze the collapse in more detail the same scenario has been simulated with multiple different initial seed values in order to gain statistically proper results. The expectation is that the system has the same behavior for each run: a fast collapse after injecting a short congesting peak load. The simulation has been replicated 500 times and the distribution of the success rate after the system is in a steady state

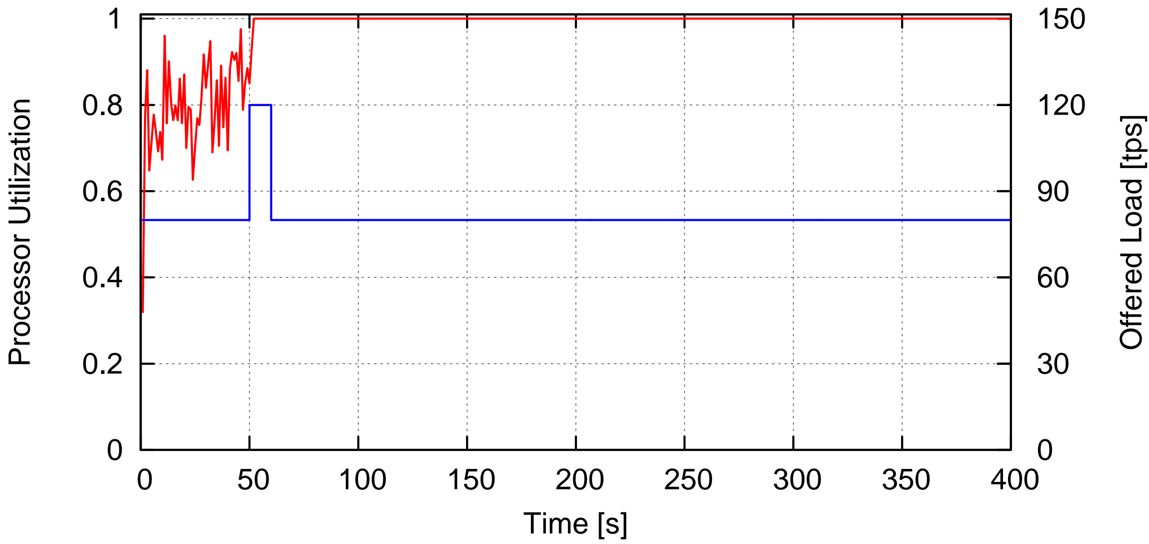


Figure 4.6.: Processor Utilization (top red line) and Offered Load (bottom blue line)

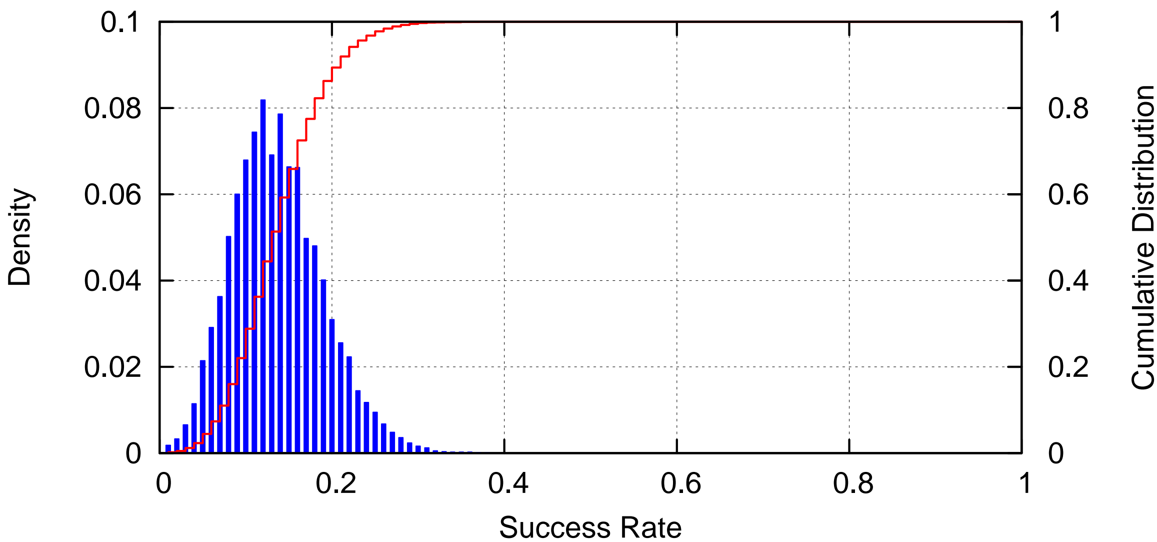


Figure 4.7.: Success Rate Distribution

(simulation time: 600-700 s) has been calculated. Figure 4.7 presents the distribution of the success rate. Observe that the success rate for all 500 runs is between 0 and 0.35, that is, the collapse happened on all simulation runs. This verifies the assumption that a collapse for the simulated system is unavoidable.

The previous section has shown that the re-transmission algorithm for SIP using UDP can be a problem if a system is under heavy load. It creates a huge load of redundant messages and the system can not recover by itself. The next section analyzes the

behavior of SIP systems in more detail and describes the characteristics of a proxy in various load situations.

4.1.2. SIP Node Load Characteristics

This section analyzes the characteristics that SIP nodes exhibit when they get faced with varying load. For this reason the same system has been simulated three times. First almost idle with only a few transactions per second (10 tps) so that the buffers are always empty. Second with high load that is just below the capacity limit (90 tps), and third with capacity exceeding load (120 tps). It is shown that the QoSg metrics fRpD and RT reflect exactly the load situation a proxy server is currently in and that the resulting information is a viable input for a congestion detection and avoidance system. Parts of the following sections have been previously published in [14].

Idle System

Figure 4.8 depicts the final Response Delay of a simulation over 400 s. It is visible that the value stays consistently lower than 100 ms during the entire simulation period. Figure 4.9 shows the density and distribution of the same values. The plots illustrate that the majority of transactions (about 90%) are processed within

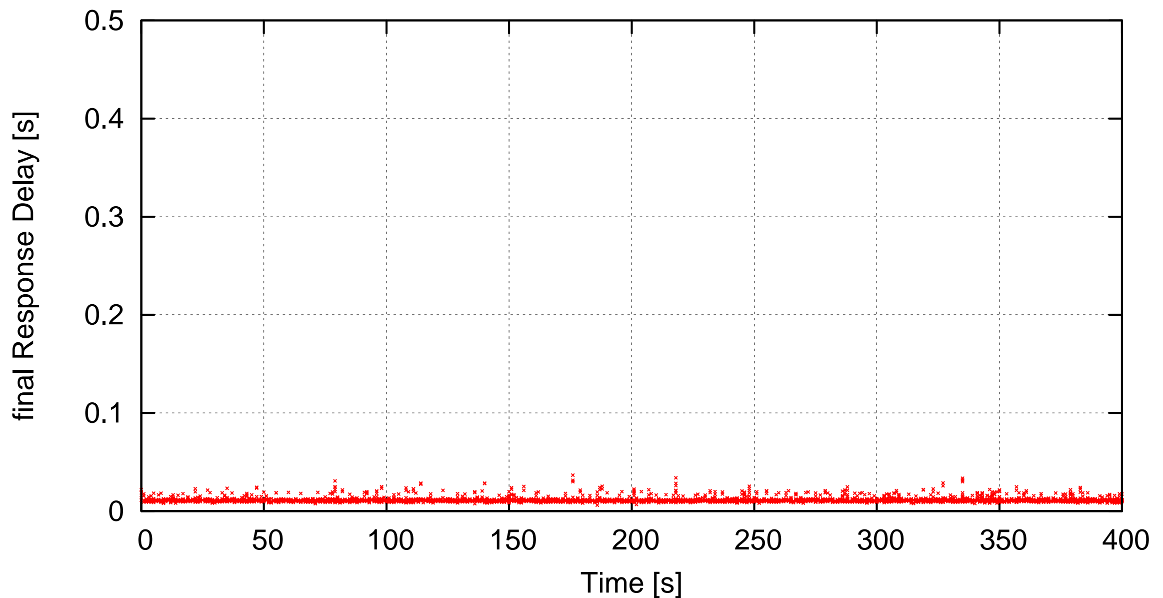


Figure 4.8.: Final Response Delay - Idle System

approximately 15 ms, caused by processing at the proxy. About 10% of the requests or responses arrive at the proxy whilst the proxy currently processes another request, therefore being subject to additional queuing delay. This effect results in the long tail of the cumulative distribution function reaching values up to 40 ms. Figure

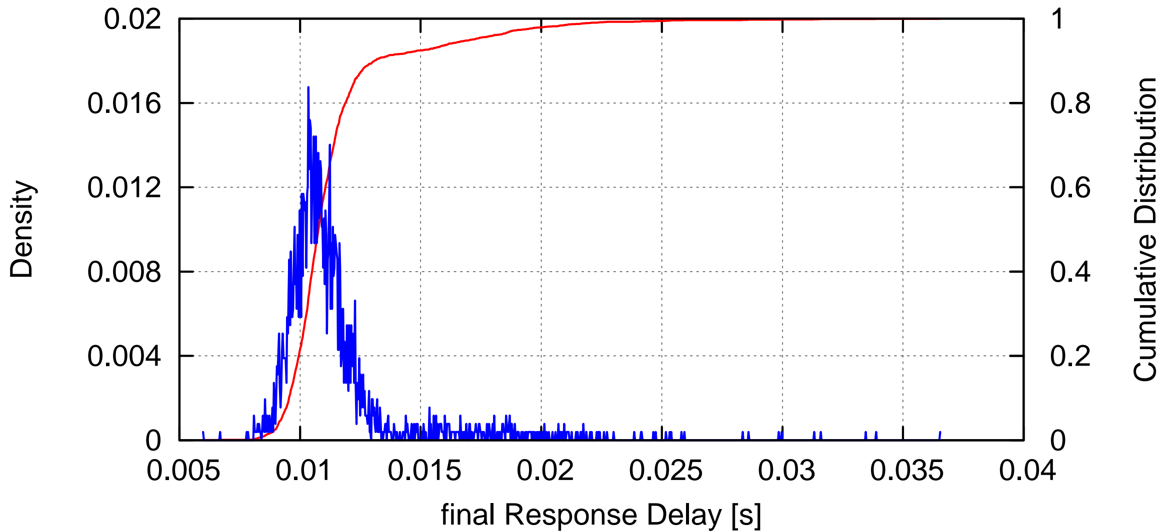


Figure 4.9.: Final Response Delay Distribution - Idle System

4.10 demonstrates that only one single transmission is necessary for each transaction. This maps to the figures of the fRpD metric and demonstrates that the system is not congested.

Highly Loaded System

If the traffic rate is increased towards high load, the fRpD metric exhibits additional variation effects that are observable in figure 4.11. Figure 4.12 depicts the distribution of the data of figure 4.11, illustrating that the high load leads to a significant change in the delay distribution. This behavior originates from the queueing mechanisms that have to manage high load inside the proxy server; the inter-arrival times are not discrete but poisson distributed and therefore the queues in the server are not always empty. This leads to more requests that have to wait until the processor is free and able to handle the request. About 90% of the transactions have their final response within 150 ms and the remaining 10% have a maximum of 350 ms. As the value of the fRpD keeps consistenldy below 500 ms, no request has to be retransmitted and therefore the RT metric does not show any difference to figure 4.10.

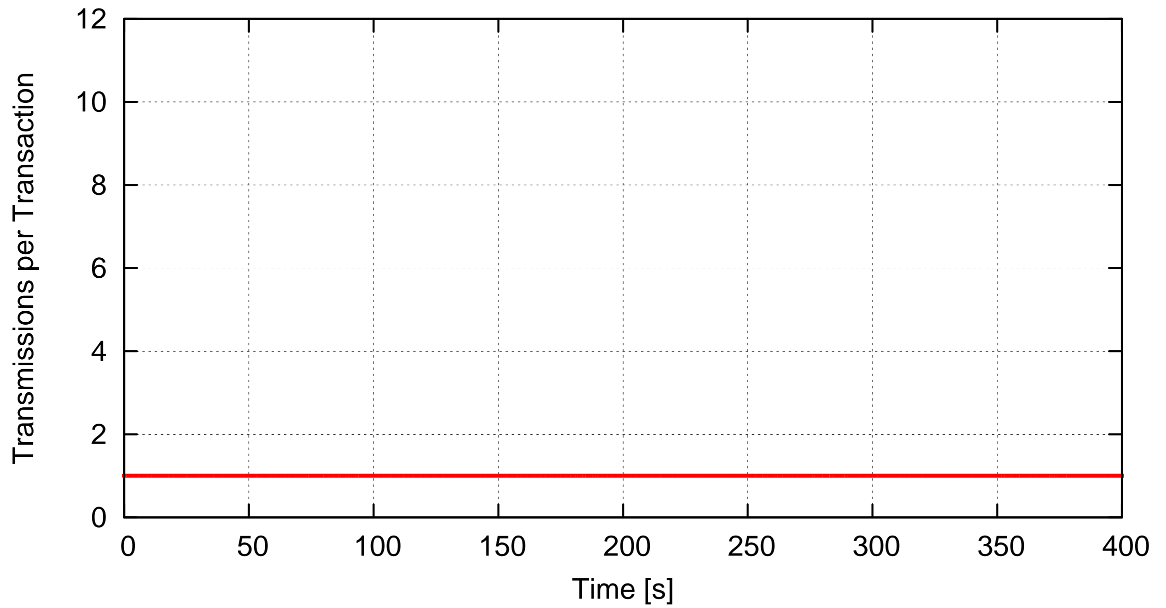


Figure 4.10.: Request Transmissions - Idle System

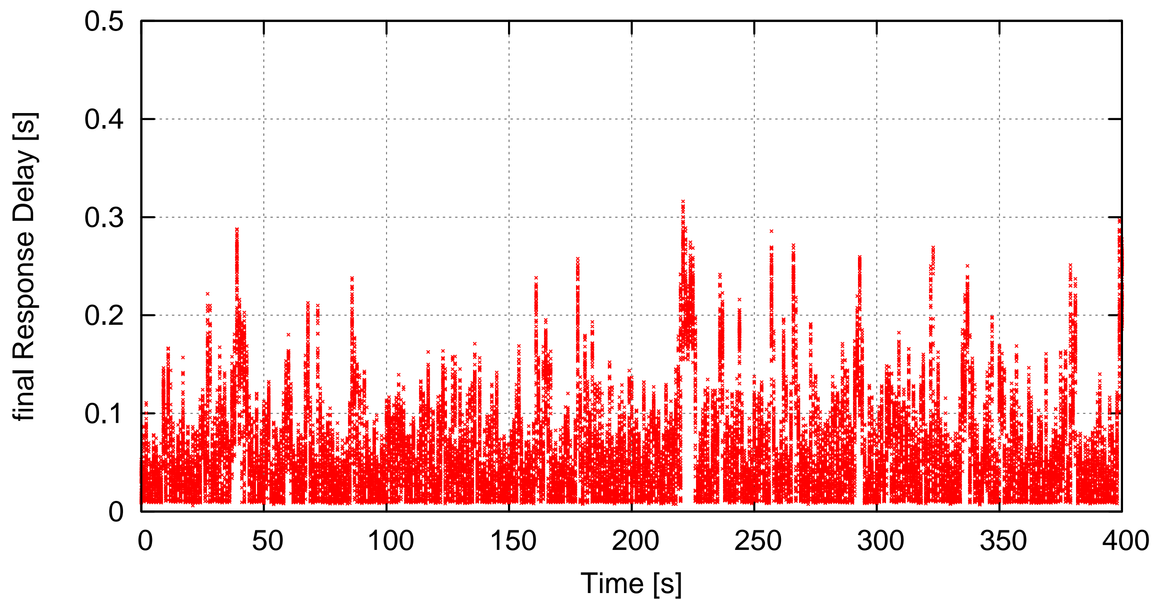


Figure 4.11.: Final Response Delay - High Load - No Congestion

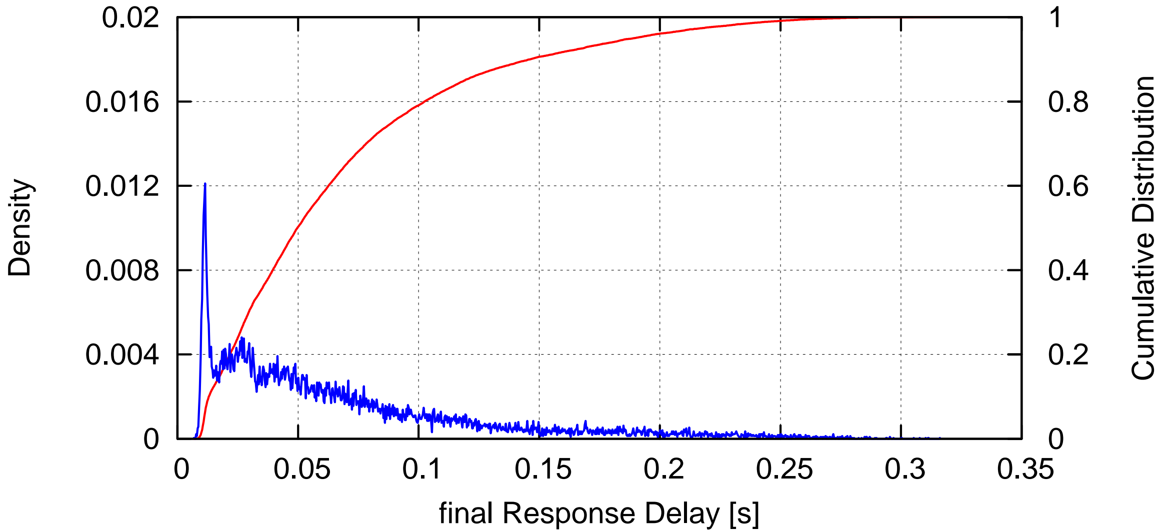


Figure 4.12.: Final Response Delay Distribution - High Load - No Congestion

Overloaded System

Figure 4.13 shows the measured fRpD for a system that has been offered more load than its capacity allows. At the beginning, the fRpD keeps low, but as soon as the systems buffers are filled, messages are dropped and therefore re-transmitted. Starting with $t = 120$ s all transactions are re-transmitted 11 times. This creates a huge load and the proxy server is only able to parse the messages, but has no free resources to route and forward the requests. Due to this reason, no requests are successful from this time onwards and no fRpD values are calculated anymore. Figure 4.14 shows for each transaction the number of requests that have been sent until a final response has ben received. It depicts that starting with $t = 120$ s all requests have to be sent the maximum number of 11 times.

These simulations indicate that the fRpD and RT metrics reflect the load state of a proxy server and the metrics therefore meet the prerequisites for congestion detection. This means that these metrics are viable candidates to be used for implicit congestion detection. Because SIP client transactions make use of timers to re-transmit requests in order to ensure reliable transmission, the effort of recording these values is acceptable.

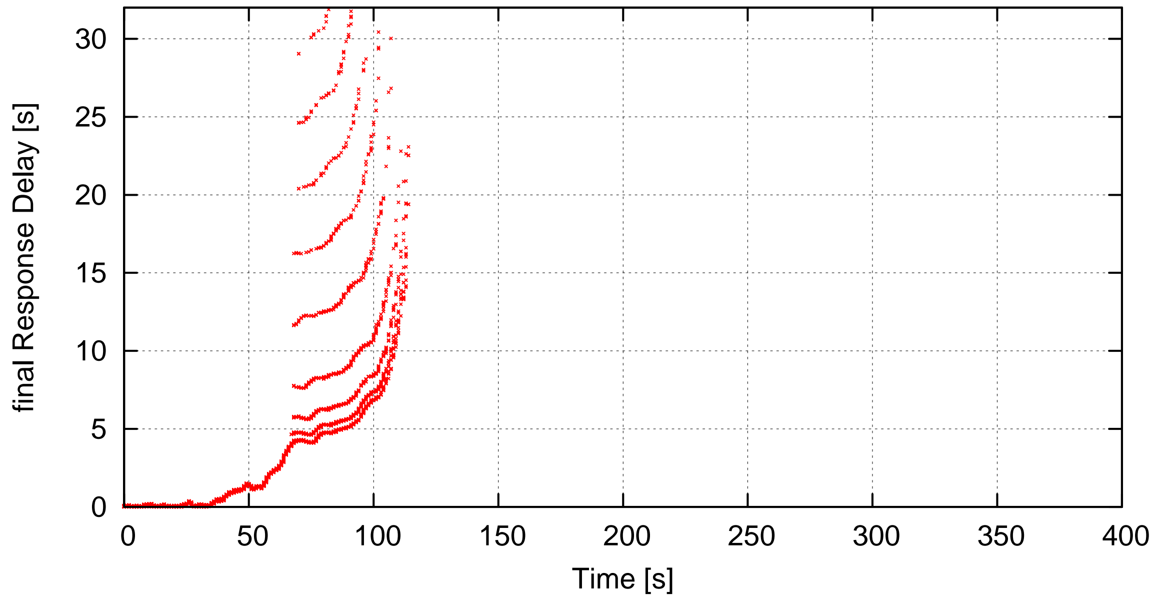


Figure 4.13.: final Response Delay - Congestion

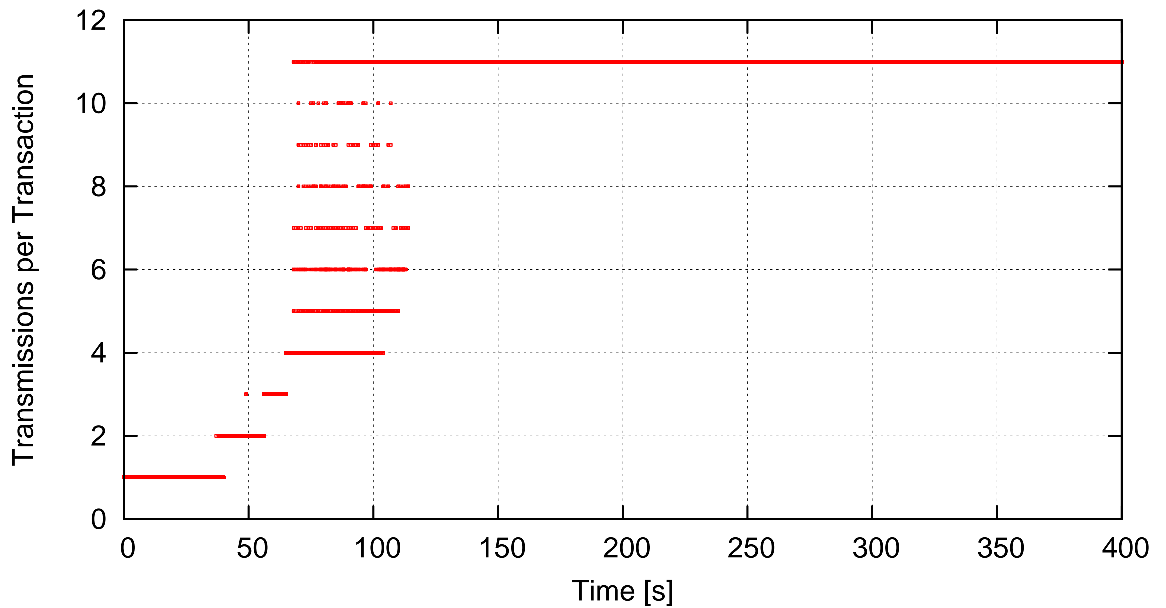


Figure 4.14.: Transmissions per Transaction - Congestion

4.2. Implicit Load and Congestion Detection

This section shows that an imminent collapse can be detected in advance by continuous measurements of implicitly measurable or computable values. These values can then be used to reduce the injected load to a downstream proxy, eventually avoiding a collapse. Because of the lack of a suitable overload control mechanism of SIP itself, the SIP Overload Control Working Group (SOC WG) of the IETF and the ETSI are both currently working on models and design considerations for SIP overload control mechanisms. Their focus is mainly on explicit overload control [21]. However the SOC WG mentions implicit overload control in [27]. They state that *"the idea of implicit overload control is that senders should try to sense overload of a downstream neighbor even if there is no explicit overload control feedback. It avoids an overloaded server, which has become unable to generate overload control feedback, from being overwhelmed with requests"* [27]. The huge advantage of implicit overload control is therefore that a system is able to detect overload of a downstream server non regarding of its support of any explicit mechanism. However, the SOC WG does not investigate on the sensing method and therefore this area needs further comprehensive research. The first step is essentially to analyze and document SIP server behavior in different load situations. This has been done and demonstrated in [14] as well as in section 4.1. This information can be used in live SIP proxies to match the current behavior of a downstream server with the documented pattern and therefore infer on its load situation.

4.2.1. Simulation Scenario

The simulation uses the models of User Agent and Proxy that have been described in sections 3.3.1 and 3.3.2. Since the focus of this section is to describe mechanisms that can detect an imminent congestion or collapse, the same simple simulation scenario as described in section 4.1 has been used. It basically consists of an aggregated User Agent Client, an aggregated User Agent Server, and a Proxy in between and is depicted in figure 4.15. The server has been configured to handle up to 100 SIP transactions per second and the links between the nodes are characterized by 0.3 ms delay and zero packet loss rate. If such a system comes into a short overload situation, many user agents have to re-transmit their requests, because the proxy server is not able to forward them accordingly. This leads to a huge increase of the incoming load. Reducing the arrival rate of new requests to the value prior to the the overload is not

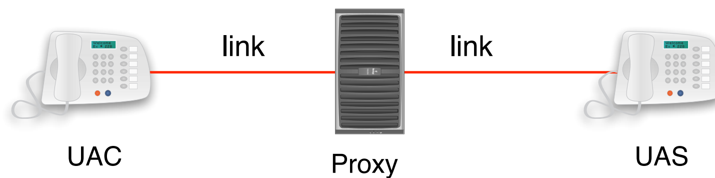


Figure 4.15.: Simulation Scenario

sufficient to recover from the collapse. This is mainly described in section 4.1.

The following section 4.2.2 describes a mechanism that uses the QoSg metric fRpD [25] as sensor and section 4.2.3 introduces a new technique that calculates an anomaly indicator for better congestion detection.

4.2.2. Delay-based Congestion Detection

This section introduces an algorithm for congestion detection (CD) based on the fRpD metric that has been defined in [25]. The fRpD is the time span between sending a SIP request until a final SIP response is received by the UAC.

The first simulation creates a specific number of new requests per second, starting from one up to 100 requests per second and logs the fRpD for all transactions. Then the distribution of the values is calculated to analyze it for the various load situations. Figure 4.16 depicts the distribution of the fRpD values of the simulated system in different load situations; the system is configured with a capacity of 100 tps and each simulation run lasts for 400 (simulation-) seconds. It is apparent, the higher the injected load, the longer takes the processing of requests. This is a well-known behavior and is caused by the queueing inside the proxy server and the Poisson arrival process of new SIP requests. When using UDP, there is no possibility to detect whether a message got lost and a retransmission is triggered by specific timers whose characteristics are described in chapter 2.1. However, under heavy load this approach introduces even more load to the system, because delayed messages may be wrongly interpreted as lost, and based on this, retransmissions are triggered. In the worst case, this behavior may further evolve into a collapse of the whole system and, once collapsed it is hard for the system to recover, even if the load starts shrinking (this fact is discussed in more detail for instance in [28]).

Therefore, the author proposes an algorithm which takes the fRpD values of the last

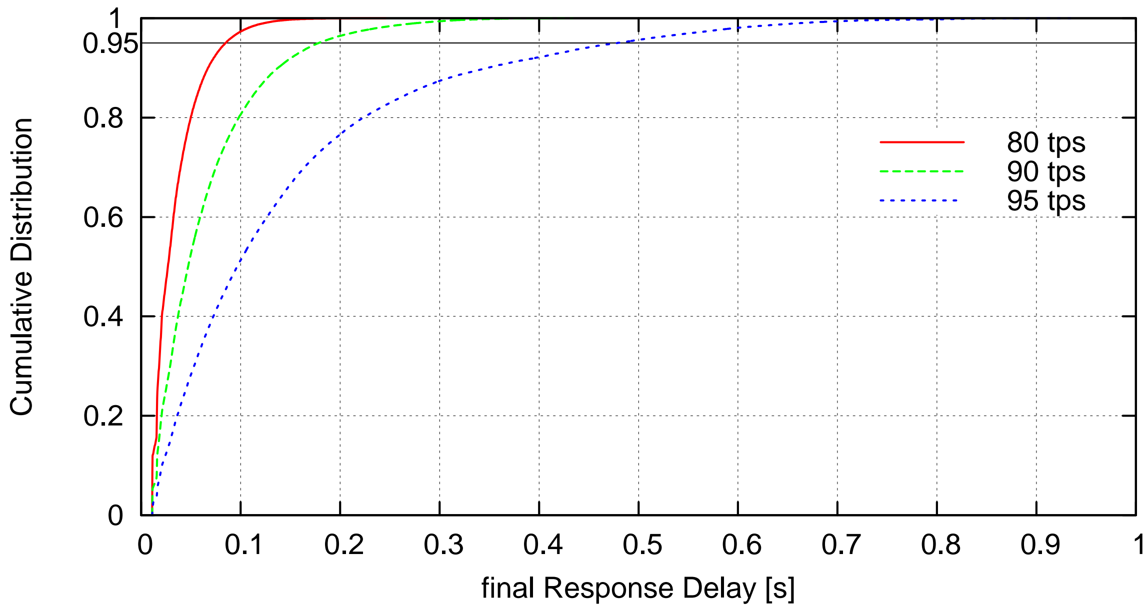


Figure 4.16.: Final Response Delay Distribution for multiple Load Settings

m seconds and calculates their 95% percentile[1]. A percentile value is used in order to cope with crashing clients and with runaway values. Due to the fact that the SIP standard requires a message to be re-transmitted after T1 (0.5 s) and that a system which is not overloaded does not need re-transmissions (or at least only a few) it can be assumed that a system is in low-load if 95% of all transactions are completed within T1. Figure 4.17 depicts the 95% percentile against the injected load and shows that the value increases only moderately at low-load. The fRpD rises with increasing load until it exceeds the 0.5 s label that is defined as low-load limit. Beyond this value the curve increases more steeply. Obviously the simulated system has that limit at 95 tps. The new algorithm which the author proposes basically consists of two parts, a measurement part that continuously measures the fRpD values of all outgoing requests and a calculation and decision part that takes the measured values every w seconds, and calculates a trigger value to decide if a situation may lead to performance problems. That trigger value is the 95% percentile of the measured fRpD values of the last m seconds. If it is above the limit of t, then a congestion is assumed. The principle of the algorithm is shown in figure 4.18 whereas P_{95} means the 95% percentile and the parameter values have been set to $m = 5$, $w = 1$ and $t = 0.5$. A system configured with these values re-calculates its performance values for the

[1]that means that 95% of all transactions need this value or below to complete successfully

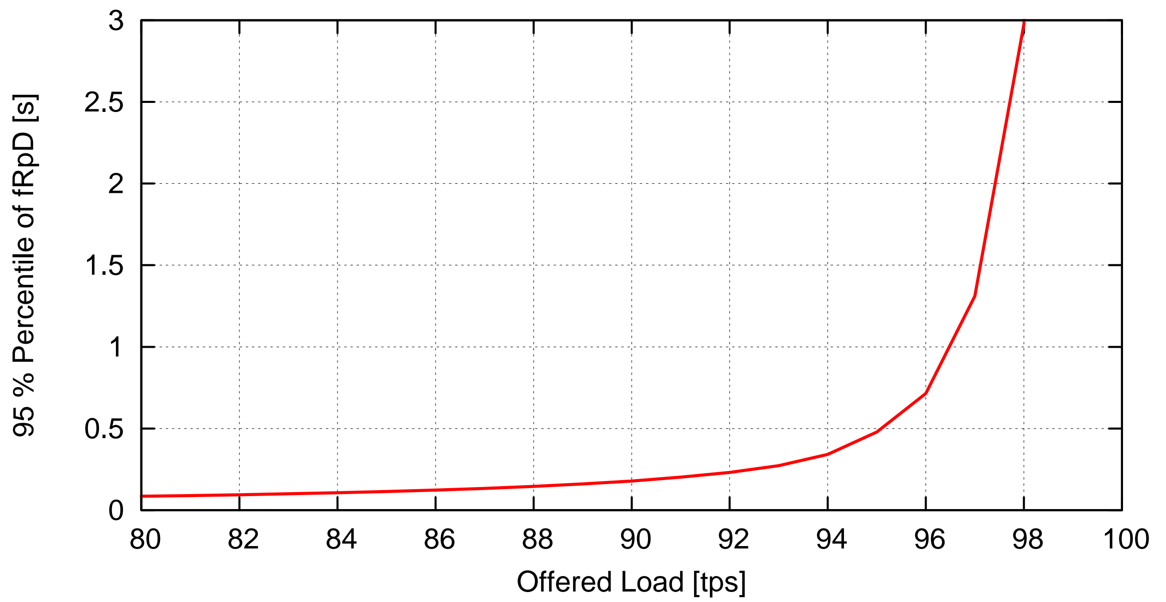


Figure 4.17.: 95% percentile of fRpD depending on Load

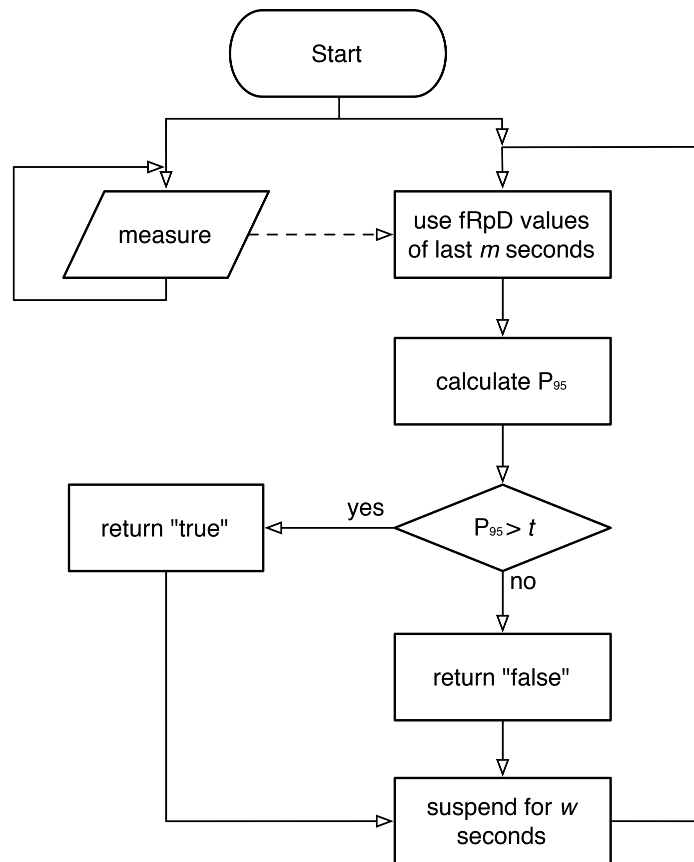


Figure 4.18.: Delay-based Congestion Detection Algorithm

last 5 s on a per-second basis. Different values may result in better performance but an empirical simulation study has shown that the above stated dimensions lead to a stable, non-oscillating system.

If a P_{95} value above t is detected, the mechanism returns “true” to the congestion handling module that is then able to reduce the outgoing load accordingly.

4.2.3. Pending-Transaction-based Detection

This section presents a new mechanism to detect an imminent collapse implicitly. This mechanism, depicted in figure 4.19, collects data that is present in any proxy

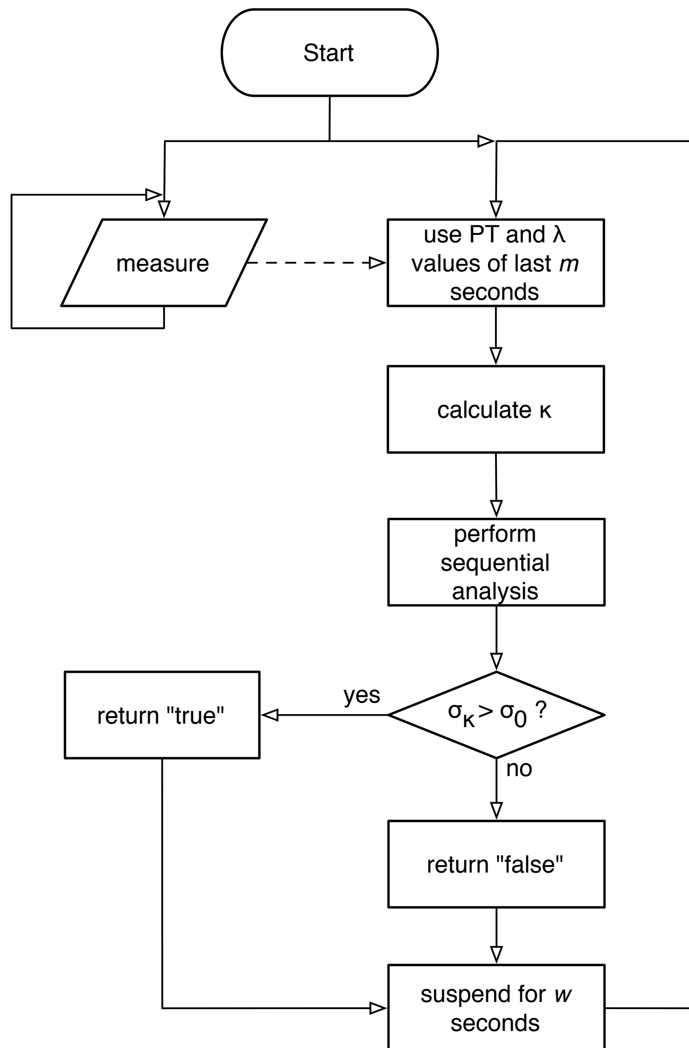


Figure 4.19.: Pending-Transaction-based Congestion Detection Algorithm

server, calculates statistics and based on that it decides if a collapse is imminent or not. It uses the metric Pending Transactions (PT) that observes the number of transactions that did not receive a response yet and the number of newly sent requests, denoted as λ. Then it calculates an anomaly indicator denoted as κ every second (or more often as configured) that is described later in this section. This value changes considerably for a collapse situation in comparison to a normal situation and a sliding statistical test can be used as detection trigger. The algorithm has been published in [15] and [24].

Data collection and Anomaly Indicator Calculation

In [24] it is demonstrated that the number of PT in a lightly loaded system correlates with the injected load and the fRpD.

$$\bar{P} = \bar{\lambda} \cdot D_f \cdot c_1 + c_2 \tag{4.1}$$

Whereas $\bar{P}$ represents the mean number of pending transactions, $\bar{\lambda}$ the mean arrival rate, D_f the fRpD, and c_1 and c_2 system dependent empirical constant correcting factors. Figure 4.20 depicts that correlation by means of simulation results of multiple

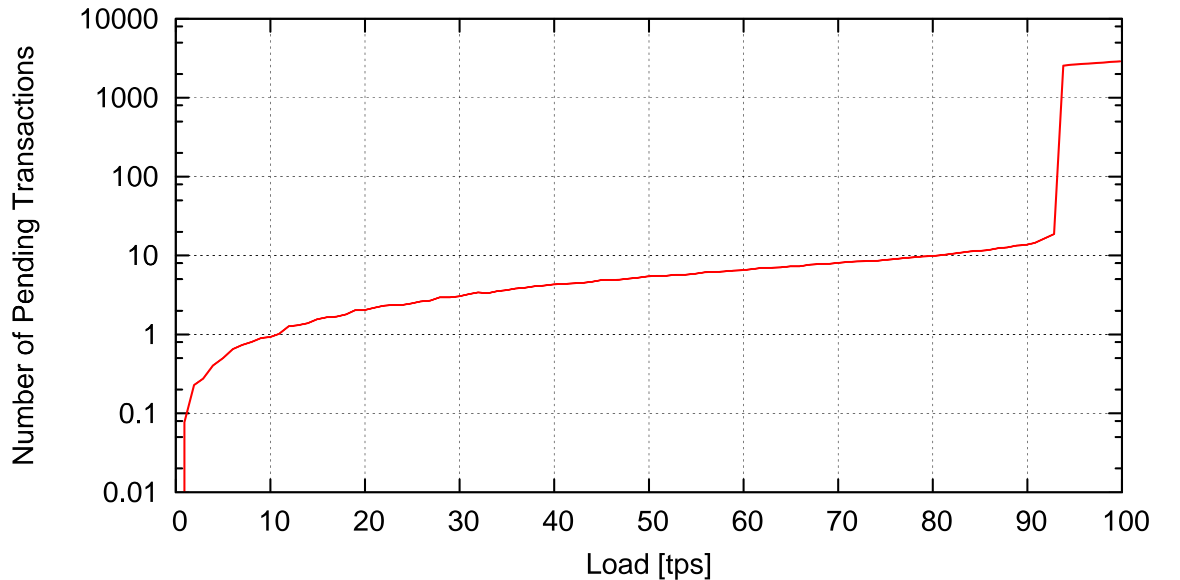


Figure 4.20.: Number of Pending Transactions depending on the injected Load

load situations on a logarithmic scale. The dependency between the load and the number of pending transactions is linear in the ergodic state (load equal to 1..92).

Above (load $>$ 92), the number of pending transactions shows a severe increase, that is, the correlation is no more linear. For the non-overloaded state, it can be assumed that the fRpD is independent of the injected load. For simplicity reasons it is also assumed that D_f is close to constant, so that the delay variation is very small. Following that, it can be stated that the number of PT correlates linearly with the injected load.

$$c_3 = D_f \cdot c_1 \tag{4.2}$$

$$\bar{P} = \bar{\lambda} \cdot c_3 + c_2 \tag{4.3}$$

As the exact value of the variables c_1 and c_2 that combine these values is unknown, and as these values are system dependent, the first derivations of them with respect to time is calculated as:

$$\frac{d\bar{P}}{dt} = \frac{d\bar{\lambda} \cdot c_3}{dt} \tag{4.4}$$

That means that changes of the load situation within a given time frame directly correlate with changes of the number of pending transactions; yielding

$$\kappa = \frac{\bar{P}'}{\bar{\lambda}'} = c_3 \tag{4.5}$$

is constant, because the correlation of the load and the number of pending transactions is linear. The new factor κ is introduced and called anomaly indicator as it is intended to indicate abnormal situations like shown in section 4.1.1. If the system is overloaded, that is, more transactions are sent to a proxy server as its capacity allows, the fRpD is no more independent of the injected load because the queues in the servers get full and some requests must probably be retransmitted. It can therefore be stated that κ is no more constant because the number of pending transactions increases dramatically. To proof this assumption multiple load situations on a two-proxy system have been simulated that logged the load and PT values. An overloading peak has been injected at $t = 50$ s that creates congestion and a collapse like already presented in section 4.1.1. As infinitely small time frames can not be logged, the values must be discretised for the simulation.

$$\frac{d\bar{P}}{dt} = \frac{\Delta\bar{P}}{\Delta t} \tag{4.6}$$

$$\frac{d\bar{\lambda}}{dt} = \frac{\Delta\bar{\lambda}}{\Delta t} \tag{4.7}$$

The simulation calculates $\Delta\bar{P}$ and $\Delta\bar{\lambda}$ each simulated second as well as the absolute value of κ. Figure 4.21 shows that $|\kappa|$ varies for our system between 0 and 2 if it is in a non-overloaded state ($t = 0..50$). That variation is explainable due to fluctuations in the fRpD that come from Poisson-distributed inter-arrival times of new requests. After introducing an overloading peak at $t = 50$ s, κ increases significantly and can therefore perfectly be used as indicator for congestion in downstream components without specific knowledge of their capacity limits. The heavy fluctuations that appear during the collapse are explainable due to non-constant inter-arrival times of new requests. However, as only the first increase of κ is interesting for a congestion detection system, that fluctuation is not studied further.

The following section analyzes the anomaly indicator for various load situations. Main aim is to get a statistical basis for statistical tests to determine the current load situation in a downstream component.

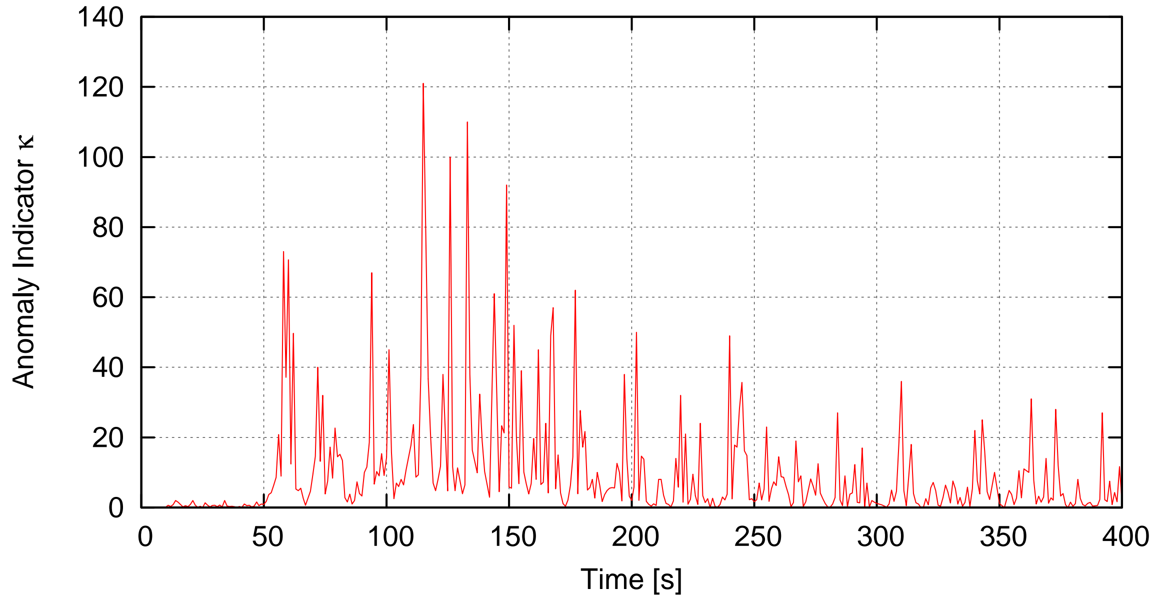


Figure 4.21.: Anomaly Indicator

Data Analysis for Statistical Tests

This section presents the distribution of the κ values for situations when the downstream component is not overloaded as well as for overloaded situations. This has to be done in order for the statistical test that is used later to be able to distinguish between normal and possibly dangerous situations.

For this purpose a two-proxy system has been simulated (see figure 4.22) where proxy 1 has 10 times more processing capacity than proxy 2 in order to overload proxy 2 and to allow proxy 1 to detect that. First the κ values of the system have been analyzed

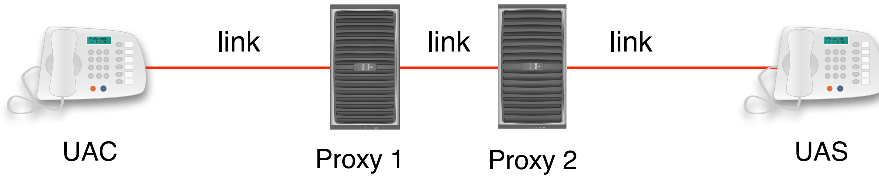


Figure 4.22.: Simulation Scenario for Data Analysis

with a load that uses 80% of the capacity of proxy 2. Second the load is increased above the capacity of proxy 2 (100 tps) to determine the difference.
Figure 4.23 shows the κ values of the non-congested system. The data collection

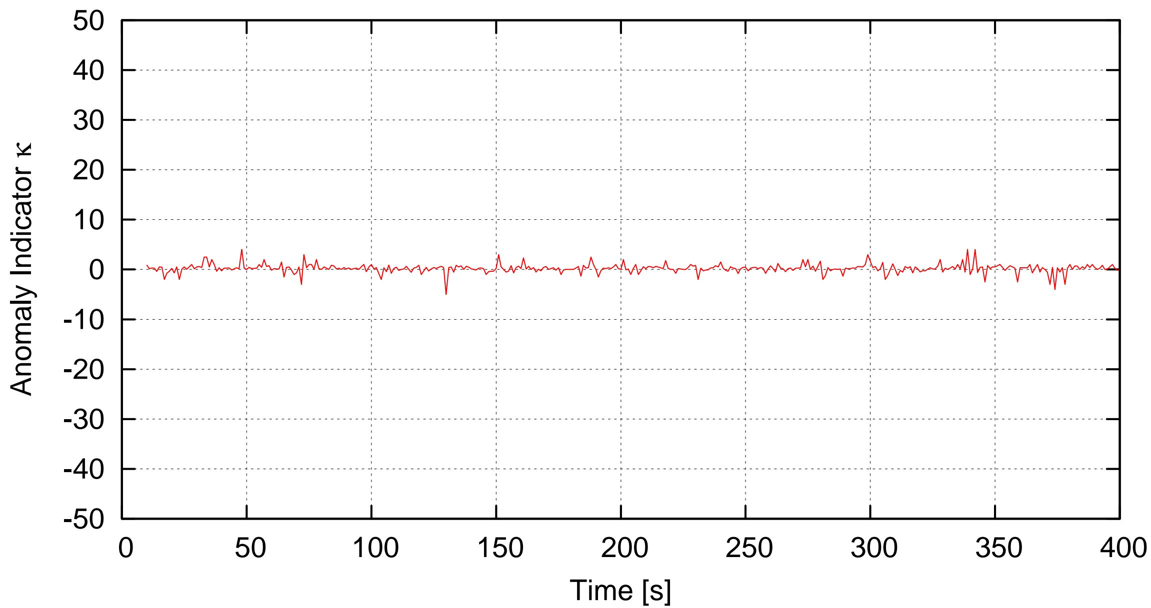


Figure 4.23.: Anomaly Indicator κ in Non-Congested System State

starts at $t = 10$ s in order to allow the system to come into a steady state and beyond that point the value stays within the interval $[-5, 4]$, has a mean value of $\mu = 0.056$ and a standard deviation of $\sigma = 0.962$. The requirement of a normally distributed population of the sequential analysis test that is used later in this section necessitates a statistical test that proves normal distribution. Figure 4.24 shows the distribution of the collected values as well as a normal distribution curve using the previously calculated mean and standard deviation values. The calculated normal distribution

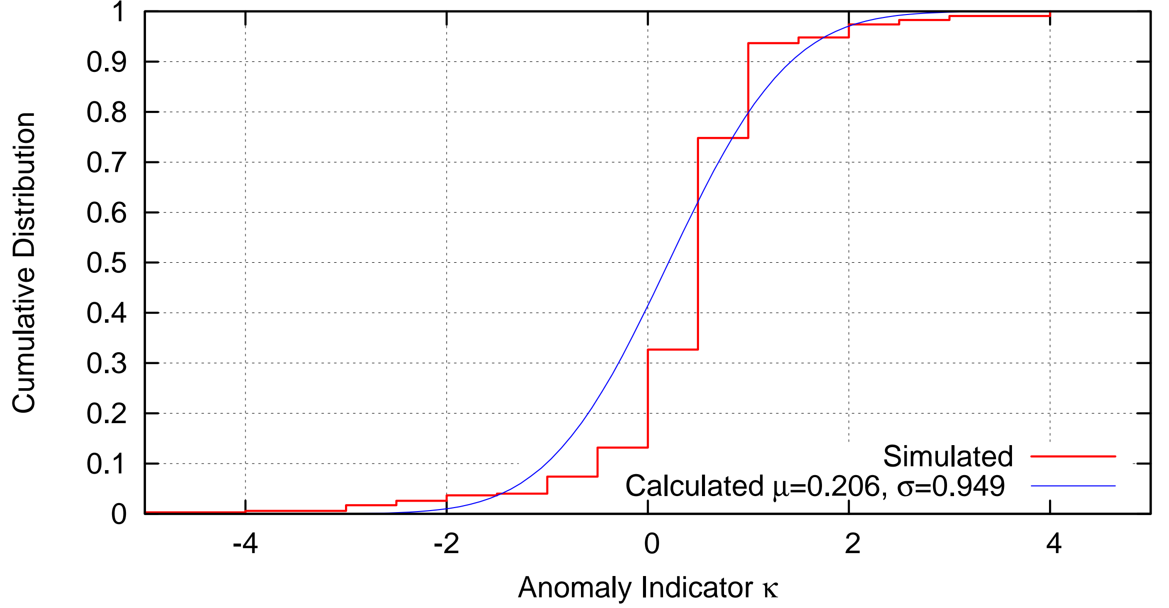


Figure 4.24.: κ Distribution in Non-Congested System State

curve does not fit perfectly to the simulated values, but the assumption that the values are normal distributed can not be rejected. A Kolmogorov-Smirnov-Test has been applied with the null hypothesis H_0 "κ values are normally distributed" (4.8) against H_1 "κ values are not normally distributed" (4.9).

$$H_0 : F^o(\kappa) = F^e(\kappa) \tag{4.8}$$

$$H_1 : F^o(\kappa) \neq F^e(\kappa) \tag{4.9}$$

The distribution of simulated (observed) values is hereby denoted as $F^o(\kappa)$ and of calculated (expected) values as $F^e(\kappa)$. For this test, the κ values have been sorted into 16 bins and are listed in table 4.1. The test requires to find the largest absolute difference between the simulated and calculated cumulative distribution functions. The observation yields to a value of 0.294. The table of critical values for the Kolmogorov-Smirnov-Test A.2 lists a value of 0.2947 for a significance level of 0.05. The observed maximum value is smaller than the critical value and that means that the null hypothesis cannot be rejected and the values can be assumed to be normally distributed.

For the overloaded situation, the load has been increased at $t = 150$ to 120 transactions per second. During the overload situation, the number of pending transactions

Table 4.1.: Observed and Expected absolute Differences of Simulated and Calculated Cumulative Distribution Functions of a Non-Congested System

i	κ_i	$F^o(\kappa_i)$	$F^e(\kappa_i)$	$\lvert F^e(\kappa_i) - F^o(\kappa_i)\rvert$	$\lvert F^e(\kappa_i) - F^o(\kappa_{i-1})\rvert$
1	-5.0	0.003	0.000	0.003	0.000
2	-4.0	0.006	0.000	0.006	0.003
3	-3.0	0.017	0.000	0.017	0.006
4	-2.5	0.026	0.002	0.024	0.015
5	-2.0	0.037	0.010	0.027	0.016
6	-1.5	0.040	0.036	0.004	0.001
7	-1.0	0.074	0.102	0.028	0.062
8	-0.5	0.132	0.228	0.096	0.154
9	0.0	0.328	0.414	0.086	0.282
10	0.5	0.748	0.622	0.126	**0.294**
11	1.0	0.937	0.799	0.138	0.051
12	1.5	0.948	0.914	0.034	0.023
13	2.0	0.974	0.971	0.003	0.023
14	2.5	0.983	0.992	0.009	0.018
15	3.0	0.991	0.998	0.007	0.015
16	4.0	1.000	1.000	0.000	0.009

increases significantly up to a certain value that is limited by the current load times the transaction timeout timer F (default: $64 \cdot T1 = 32s$). In the present simulation scenario, this value is equal to $\frac{120trans.}{s} \cdot 32s = 3840$ transactions. Figure 4.25 shows the number of pending transactions against time and that the value increases starting with $t = 150$ until $t = 220$. As the test has to detect the collapse process as early as possible (that is, before the maximum number of pending transactions is reached) the distribution of the anomaly indicator value κ has to be calculated only during the period $t = 150..220$ because it is different from the distribution when the number of pending transactions has reached is maximum value. In figure 4.26 the κ values are shown during the period $t = 150..200$. The variance of the value increases significantly in comparison to the non-congested state whereas the mean value approximately stays within the same dimension. The characteristic values amount to $\mu = -2.407$ and $\sigma = 20.392$. Figure 4.27 shows the distribution of κ for the interval $t = 150..200$ and a normal distribution curve using the calculated values for a congested system. The curves for the simulated and calculated values lead also to the

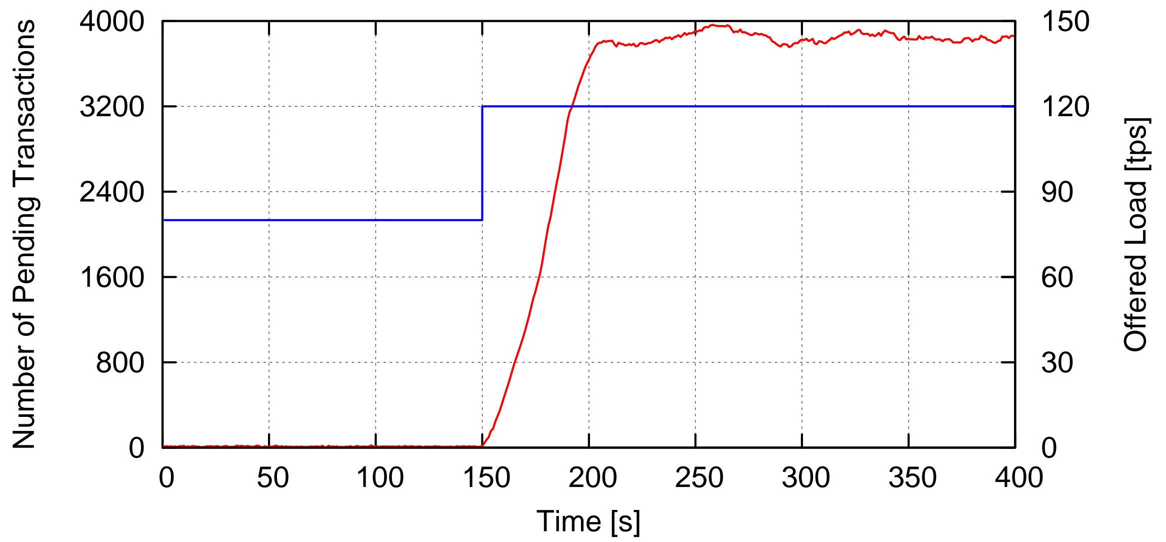


Figure 4.25.: Number of Pending Transactions during Overload

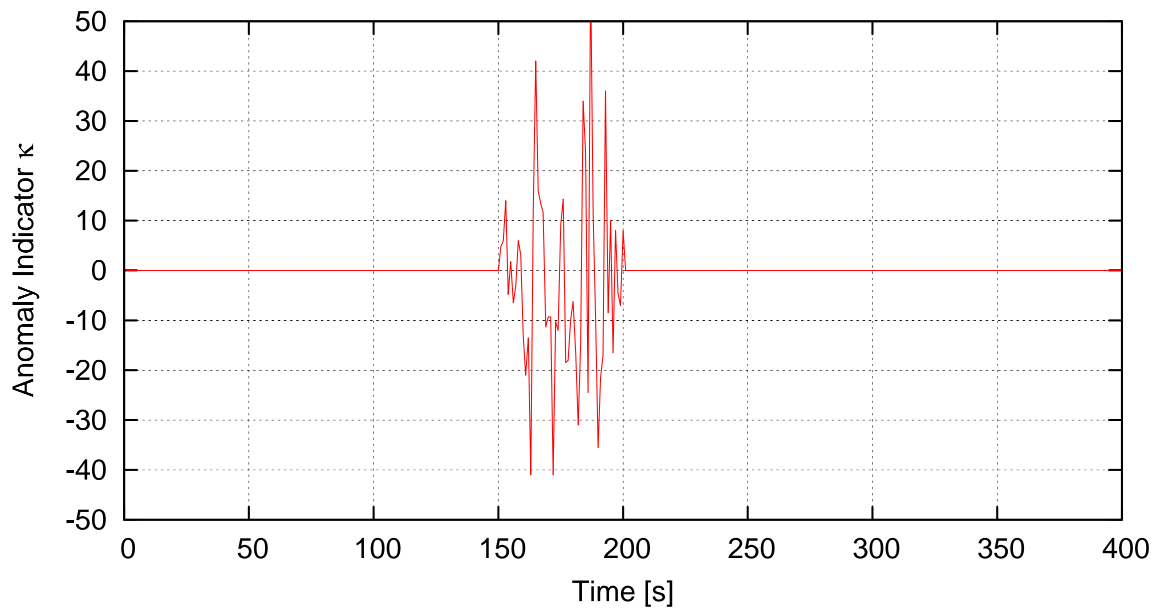


Figure 4.26.: Anomaly Indicator κ in Congested System State during $t = 150..200$

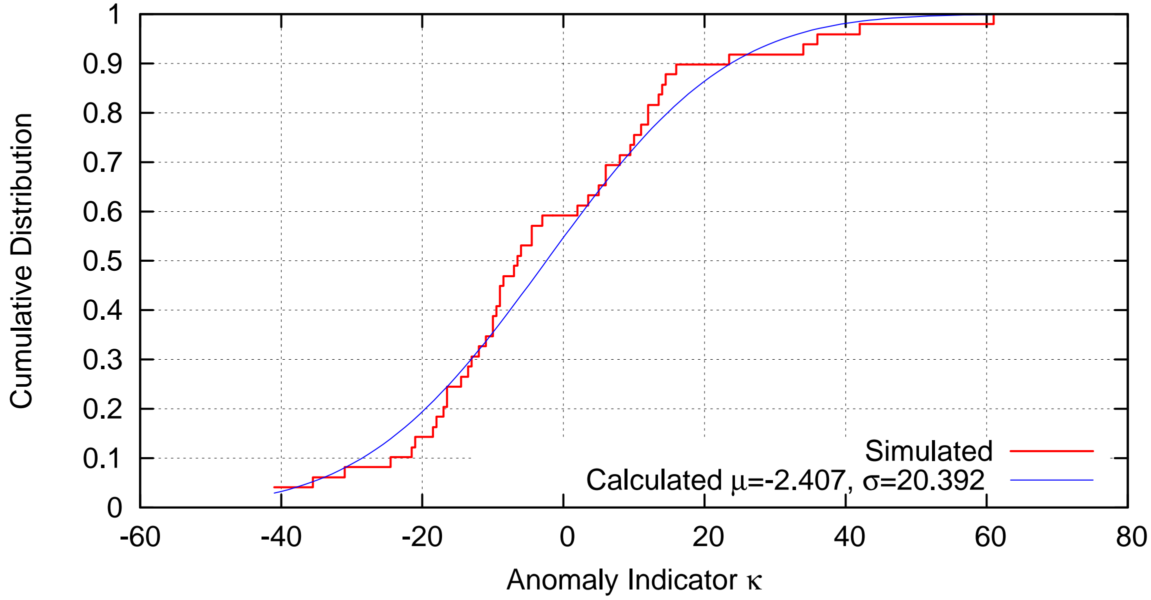


Figure 4.27.: κ Distribution in Congested System State

assumption of a normal distributed κ and a Kolmogorov-Smirnov-Test has been done for these values as well whose results follow. The values for the congested system are between -41..61 and have been sorted into 42 bins whose cumulative values are shown in table 4.2 (the table has been cropped and only shows the values between -7 and 9.5 as the maximum appears at $\kappa = -4.5$; the full table is in appendix A in table A.2).

As the number of bins is larger than 40, the critical value is calculated as

$$z = \frac{1.22}{\sqrt{n}} = \frac{1.22}{\sqrt{42}} = 0.188 \tag{4.10}$$

The observed maximum value of 0.112 is smaller than the critical value and this means that the null hypothesis cannot be rejected and the values can be assumed to be normally distributed.

This section has shown that the κ value is normal distributed in the two interesting load situations and that the variance increases significantly in an overload situation. That leads to the following section which explains how a sequential analysis test is applied so that a congestion situation can be detected.

Table 4.2.: Observed and Expected absolute Differences of Simulated and Calculated Cumulative Distribution Functions of the Anomaly Indicator κ of a Congested System (cropped)

i	κ_i	$F^o(\kappa_i)$	$F^e(\kappa_i)$	$\lvert F^e(\kappa_i) - F^o(\kappa_i)\rvert$	$\lvert F^e(\kappa_i) - F^o(\kappa_{i-1})\rvert$
20	-7.0	0.490	0.411	0.079	0.058
21	-6.5	0.510	0.420	0.090	0.070
22	-6.0	0.531	0.430	0.101	0.080
23	-4.5	0.571	0.459	**0.112**	0.072
24	-3.0	0.592	0.488	0.104	0.083
25	2.0	0.612	0.586	0.026	0.006
26	3.5	0.633	0.614	0.019	0.002
27	5.0	0.653	0.642	0.011	0.009
28	6.0	0.694	0.660	0.034	0.007
29	8.0	0.714	0.695	0.019	0.001
30	9.5	0.735	0.720	0.015	0.006

Sliding Sequential Analysis for κ

The previous sections have shown that the κ values of a non-congested and a congested system differ significantly from each other. The idea behind this section is to use the sequential analysis test developed by A. Wald and described in [59] to differentiate between a normal and an overload situation. This test represents *a method of statistical inference whose characteristic feature is that the number of observations required by the procedure is not determined in advance of the experiment* [59]. These criteria suit a collapse detection system perfectly as in general the point in time of collapse is not known and so the number of necessary observations of the value κ for a statistical test is unknown in advance.

The characteristics of the value κ for various load situations have already been presented in the previous section and it appears that the mean values of a congested and non-congested system are approximately in the same order of magnitude whereas the standard deviations differ significantly. Because of that, the test presented in chapter 8 of [59] is appropriate for the present case. It has been developed for *testing that the standard deviation of a normal distribution does not exceed a given value*. If the κ values are tested continuously with this procedure, the test should detect overload situations.

The sequential test procedure works as follows. The smaller the standard deviation σ_κ of κ, the lower the probability that the system is in a congested state. Thus, according to [59] a value σ'_κ can be defined as a limit such that the system is considered as congested if $\sigma_\kappa > \sigma'_\kappa$ and as not congested if $\sigma_\kappa \leqq \sigma'_\kappa$. This can be formulated as hypothesis H_0 that the system is not congested:

$$H_0 : \sigma_\kappa \leqq \sigma'_\kappa \tag{4.11}$$

and hypothesis H_1 that the system is congested:

$$H_1 : \sigma_\kappa > \sigma'_\kappa \tag{4.12}$$

The test procedure defines a sampling plan for testing the hypothesis that the system is not congested.

Two values $\sigma_{\kappa,0}$ and $\sigma_{\kappa,1}$ (whereas $\sigma_{\kappa,0} < \sigma'_\kappa$ and $\sigma_{\kappa,1} > \sigma'_\kappa$) are specified to classify the system state. For the simulated system, the previously gathered standard deviation values for a non-congested and a congested system can be used as $\sigma_{\kappa,0}$ and $\sigma_{\kappa,1}$ respectively. If the system state is classified as congested if $\sigma_\kappa \leqq \sigma_{\kappa,0}$ this is considered an error of the first kind and if the system is classified as non-congested if $\sigma_\kappa \geqq \sigma_{\kappa,1}$ this is considered as an error of the second kind. The probability of an error of the first kind is denoted as α and should not exceed a small value whenever $\sigma_\kappa \leqq \sigma_{\kappa,0}$ and the probability of an error of the second kind is denoted as β and should not exceed a small value whenever $\sigma_\kappa \geqq \sigma_{\kappa,1}$. For each iteration of the sequential test (denoted by integer number m), the acceptance number

$$a_m = \frac{2 \log \frac{\beta}{1-\alpha}}{\frac{1}{\sigma^2_{\kappa,0}} - \frac{1}{\sigma^2_{\kappa,1}}} + m \cdot \frac{\log \frac{\sigma^2_{\kappa,1}}{\sigma^2_{\kappa,0}}}{\frac{1}{\sigma^2_{\kappa,0}} - \frac{1}{\sigma^2_{\kappa,1}}}, \forall m \geq 1 \tag{4.13}$$

and the rejection number

$$r_m = \frac{2 \log \frac{1-\beta}{\alpha}}{\frac{1}{\sigma^2_{\kappa,0}} - \frac{1}{\sigma^2_{\kappa,1}}} + m \cdot \frac{\log \frac{\sigma^2_{\kappa,1}}{\sigma^2_{\kappa,0}}}{\frac{1}{\sigma^2_{\kappa,0}} - \frac{1}{\sigma^2_{\kappa,1}}}, \forall m \geq 1 \tag{4.14}$$

is calculated. At each iteration step the sum

$$\xi_m = \sum_{i=1}^{m} (\kappa_i - \mu)^2 \tag{4.15}$$

has to be calculated and the inspection is continued as long as $a_m < \xi_m < r_m$. As soon as $\xi_m \leqq a_m$ the iteration is terminated and the system declared as non-congested and if $\xi_m \geqq r_m$ the iteration is terminated and the system declared as congested.
This calculation has to be done as frequently as possible to provide a fast response time if a severe situation happens. However, exceedingly frequent calculations may use many resources and therefore a tradeoff of response time and resource consumption must be found. As a starting point, the simulation calculates these values every 100 ms, that is, 10 observations per second.
It is worth noting that the error of the first kind is less critical than an error of the second kind as it is generally more critical if a congestion is not detected (because a congestion situation can result in more severe situations as presented in section 4.1) as if a normal situation is wrongly interpreted as congestion. Because of this, the probability of an error of the first kind α has been set to 0.1 and β, the probability of an error of the second kind to 0.05.
In order to prove these assumptions, the sequential test has been implemented in the simulator IBKsim and the same two-proxy system as shown in the beginning of this section has been used. As result the simulator saves the simulation time to a file when it possibly detects a congestion. First a non-overloaded system with constant load is analyzed and second a system that is firstly not congested and after some time an overloading peak is introduced. The simulations last 2000 seconds and the

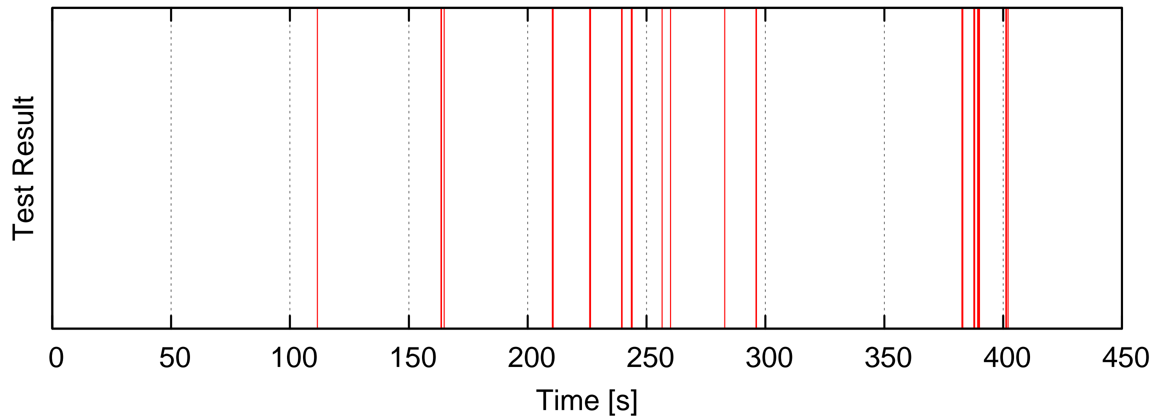


Figure 4.28.: Results of the Sequential Analysis Test for Non-Congested System

injected load for the first simulation is approximately 80% of the capacity of the second, weaker proxy. The calculations of κ and the sequential test start at $t = 150$ s and that leads to $(2000\ s - 150\ s) * \frac{10}{s} = 18500$ calculations. The resulting file lists 229 lines, however, the second proxy server was never overloaded due to the injected

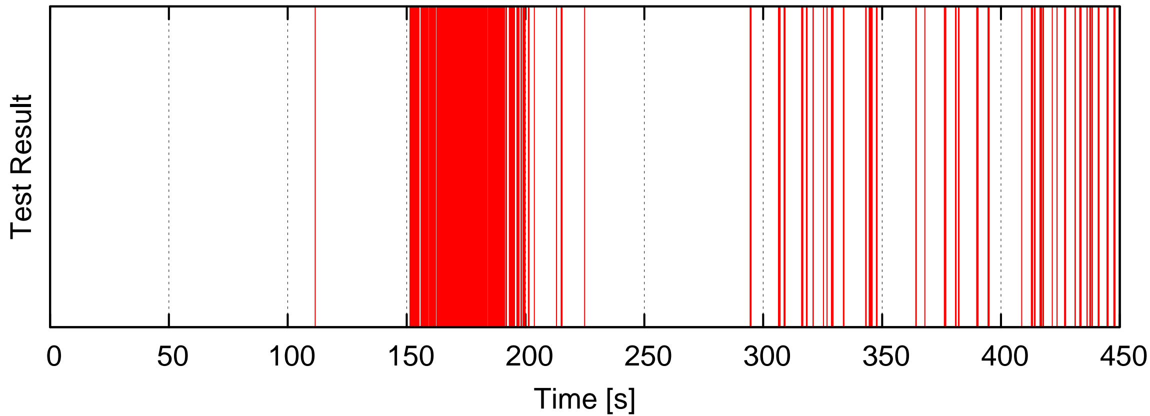


Figure 4.29.: Results of the Sequential Analysis Test for Congested System

load and so it is obvious that all of these lines are false alarms, that are about 1.2% of all calculations. Figure 4.28 depicts the test results by means of impulses, whereas each impulse corresponds to a single detection. The second simulation introduces overload of approximately 120% of the proxy's capacity at $t = 150$ s. Figure 4.29 depicts the results of the test for this situation and it appears that it responds at the time when the overload situation starts.

Analysis of the mechanism

Because of the false alarms in a non-congested situation presented in the last section, it is now of interest how the number of false alarms depends on the load. In order to analyze this, various load situations have been simulated starting from 1 transaction per second up to 100 and the number of alarms has been logged. Because of the processing capacity of 100 transactions per second of proxy 2 and poisson distributed arrival times of new transactions, a total system capacity limit of 92 transactions per second has already been identified (see also figure 4.20). That means that any alarms given by the sequential analysis test is a false alarm if the injected load is below that limit. Again, the simulation duration for each load situation is 2000 seconds and the logging starts at $t = 100$ s. Figure 4.30 shows the number of alarms (% of total calculations) against injected load. Below the capacity limit of proxy 2 the number of alarms stays below the previously defined 5% error threshold level. We conclude, the proportion of false alarms is acceptable and the proposed method works as expected. Thereafter, several methods can be applied to reduce the outgoing load such that the overloaded system can recover from its collapse. These methods are presented

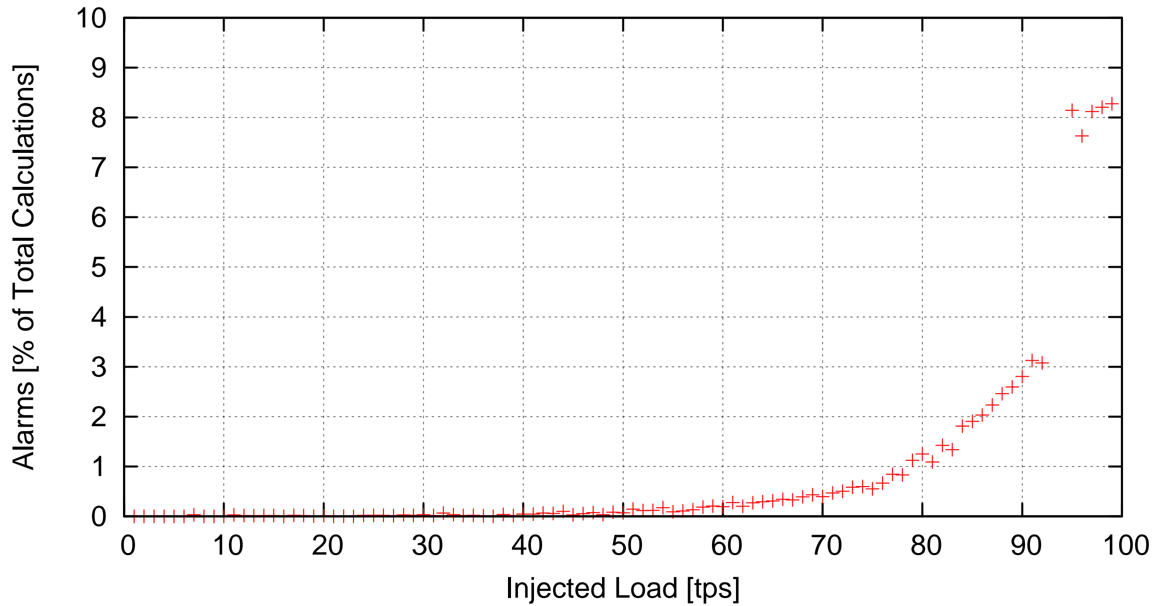


Figure 4.30.: Number of Alarms (% of Total Number of Calculations) against injected Load

in the following sections. One method is to increase the re-transmission timer for newly arriving requests to give the overloaded system sufficient time to process them, as presented in section 4.3.2 or in [16]. Another approach is to limit the number of pending transactions to a certain value and to reject additional requests as presented in section 4.3.3, as well as in [15] and [24].

4.3. Congestion Handling

The previous section has shown that implicit detection is possible either by means of measuring a delay that occurs when sending a new request until a response has been received or by calculating an anomaly indicator that uses the number of outgoing new- and the number of pending requests. The presented mechanisms provide the information that a congestion situation is currently imminent or not and can act subsequently as a trigger for the following congestion handling mechanisms. As SIP systems generally generate revenues for an operator, it is of major interest to accept as many requests as possible. Therefore, a so-called lossless mode to reduce load in a congestion situation is always preferable because it tries to cope with congestion without rejecting requests. If that is not sufficient, a lossy mode has to be applied, where some requests have to be rejected in order to prevent a congestion collapse.

The following section 4.3.1 shows the simulation model that has been used for this research. Section 4.3.2 presents a lossless mode that analyzes and proposes settings for various configurations of SIP re-transmission timers. Section 4.3.3 presents the results of applying a rate-limiter by means of restricting the maximum number of allowed pending transactions, a lossy mode.

4.3.1. Simulation Scenario

The simulation uses again the same models of user agent and proxy that have been described in sections 3.3.1 and 3.3.2. The scenario basically consists of an aggregated UAC, an aggregated UAS, and two proxies in between as depicted in figure 4.43. Proxy 1 has ten times more processing capacity than proxy 2 and therefore the second proxy is a bottleneck which has to be detected by the first.

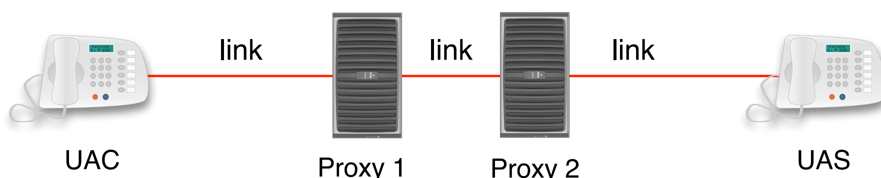


Figure 4.31.: Simulation Scenario

4.3.2. Adaptive Timer Manipulation

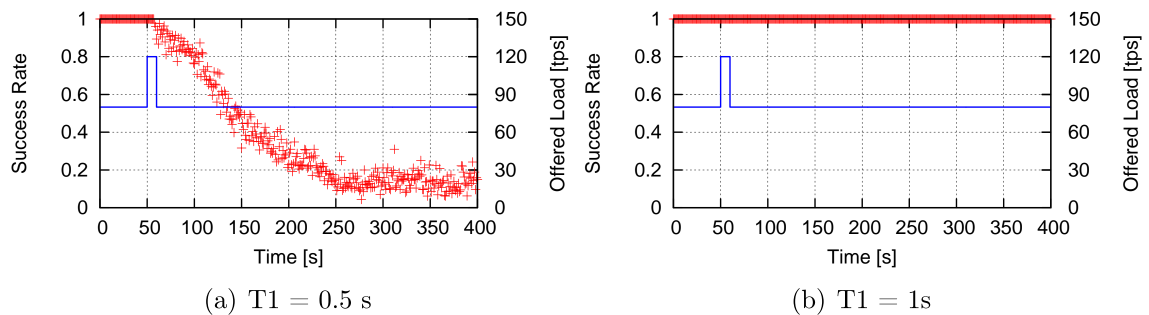
This section presents a detailed impact analysis of SIP timers onto the functionality of a system and its capabilities to recover from overload situations. The results of the simulations demonstrate that message retransmissions originating from a minor short-term overload can force a system into a deterministic congestion collapse when using the timer T1 settings as recommended by RFC 3261 [53]. A recovery from this severe system overload situation is highly difficult or impossible, even if the system load is reduced substantially afterwards [28]. Performance evaluations of various parameters show that an increase of T1's value significantly enhances the stability, robustness and the ability of the system to handle overload. The resulting increase of response times is relatively small, while overall system responsiveness can even improve in some cases.

Analysis of Multiple Timer Configurations

The following sections show the results of evaluations of the system's success rate, the total number of transmissions per transaction that is necessary for all transactions including unsuccessful ones, the resulting fRpD as well as the processor utilization. The retransmission timer T1 has been set to the default value recommended in RFC 3261 [53] and then the same scenario has been simulated with a timer T1 value equal to 1 s. A background arrival rate of 80 messages per second is introduced when starting the simulation. At $t = 50$ s, the arrival rate is increased to 120 messages per second for a duration of 10 s. That causes congestion in the system as the arrival rate clearly exceeds the capacity of proxy 2 of 100 messages per second.

Success Rate

Figure 4.32(a) shows the success rate for T1 equal to 0.5 s. After the short peak, the system runs into a congestion collapse, and only a few transactions are handled successfully. For a T1 value of 1 s depicted in figure 4.32(b), the proxy can process all requests.



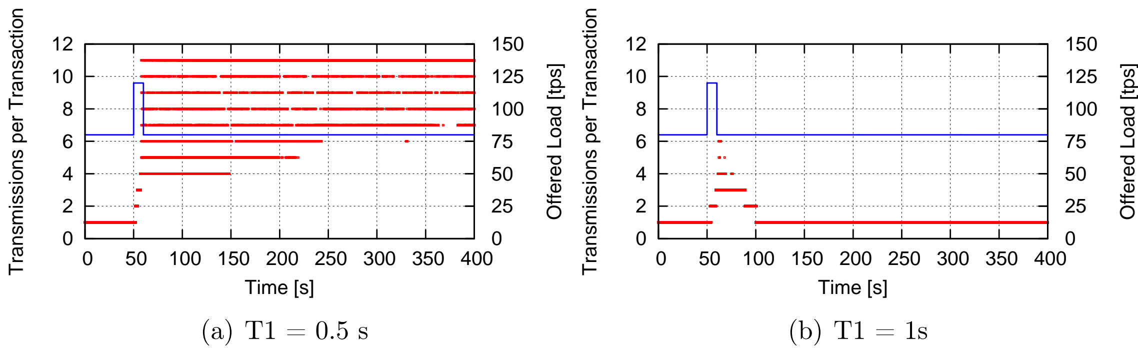
(a) T1 = 0.5 s (b) T1 = 1s

Figure 4.32.: Success Rate (red crosses) and Offered Load (blue line)

Total Number of Transmissions per Transaction

Figure 4.33 shows the number of transmissions for each transaction that has been sent until the UAC receives the first response. It illustrates that for a T1 value of 0.5 s, the system is flooded by retransmissions and that after about 70 seconds, all requests are sent multiple times. Increasing T1 to 1 s changes this behavior to a regular high-load situation as with this setting the system as a whole has more time to process the requests before they get re-transmitted. During the peak load and

some time afterwards a few requests have to be retransmitted. However, after the peak has disappeared, all requests are processed within the first transmission and no re-transmissions are necessary anymore, so that the system returns to a stable state.



(a) T1 = 0.5 s

(b) T1 = 1s

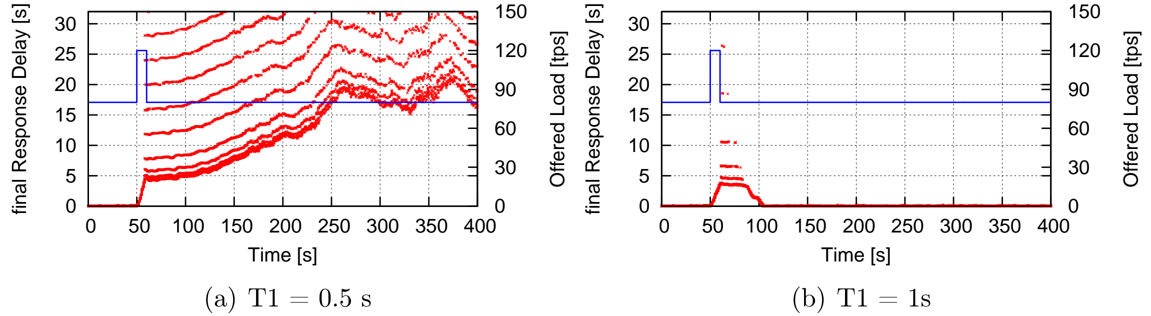
Figure 4.33.: Total Number of Transmissions per Transaction (red dots) and Offered Load (blue line)

Final Response Delay

Figure 4.34 shows the final response delay in seconds in a scatter plot, where each dot corresponds to a single transaction. It can be observed that for T1 = 0.5 s, the fRpD increases up to the maximum time of 31.5 seconds. Due to the static re-transmission timer configuration of the UAC, the dots merge to lines and each of the lines corresponds to a single re-transmission instance. That means the lowest line belongs to transactions that have received a response to the first transmission of the initial request, the second lowest line to transactions that have received a response to the second transmission and so on. This supports the assumption that re-transmissions are unnecessary in situations where a proxy server is congested. For T1 equal to 1 s, the fRpD increases during the appearance of the peak but decreases afterwards and consistently stays at the previous level.

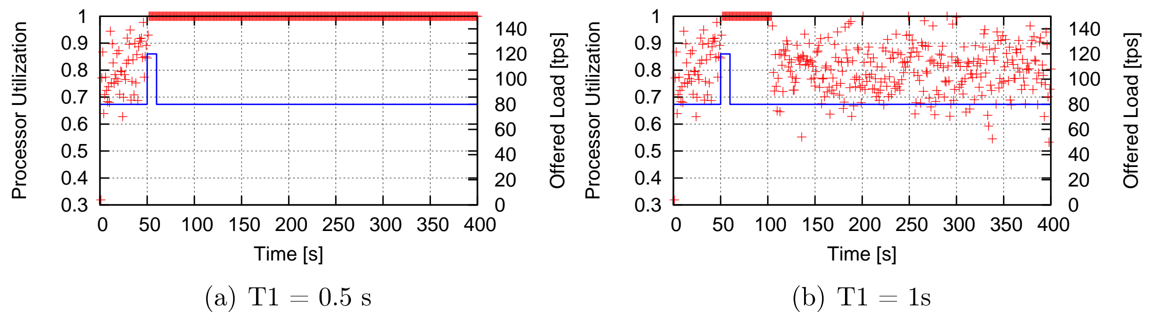
Proxy Processor Utilization

Figure 4.35 shows the processor utilization of the proxy server. It can be observed that for T1 equal to 0.5 s, the processor of the proxy remains at 100% utilization after the injected peak load throughout. Increasing T1 to 1 s yields a processor utilization that decreases afterwards and consistently stays at the previous level.



(a) T1 = 0.5 s (b) T1 = 1s

Figure 4.34.: Final Response Delay (red dots) and Offered Load (blue line)

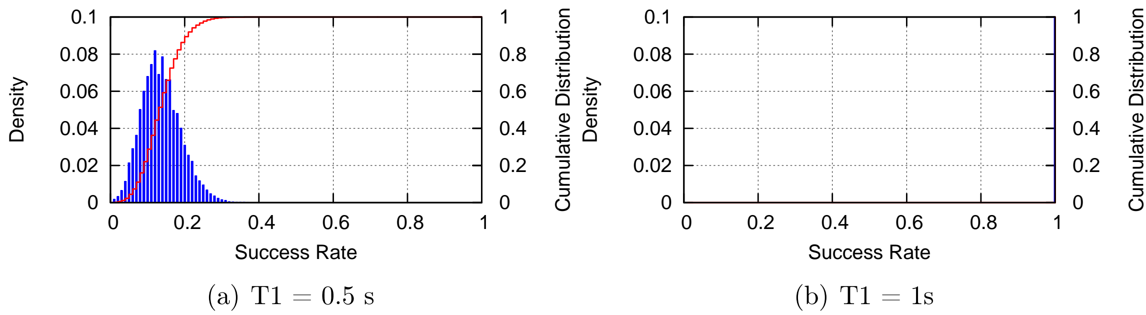


(a) T1 = 0.5 s (b) T1 = 1s

Figure 4.35.: Processor Utilization (red crosses) and Offered Load (blue line)

Collapse Analysis

To analyze the collapse in more detail, the same scenario has been simulated with multiple different initial seed values and the system is expected to have the same behavior for each run: a fast collapse after injecting a short congesting peak load. Five hundred runs have been simulated and the distribution of the success rate after the system is in a steady state (simulation time: 600-700 s) has been analyzed. Figure 4.36 presents the distribution of the success rate. It can be observed that for T1 = 0.5 s the success rate for all 500 runs is between 0 and 0.35, that is, in none of the simulation runs the system could fully recover from the load peak. This verifies the assumption that a collapse for this simulated system is unavoidable with a T1 setting of 0.5 s. For a T1 setting equal to 1 s, none of the systems has collapsed within 700 seconds of simulation time and the distribution function shows a single peak at SR=1.



(a) T1 = 0.5 s (b) T1 = 1s

Figure 4.36.: Success Rate Distribution

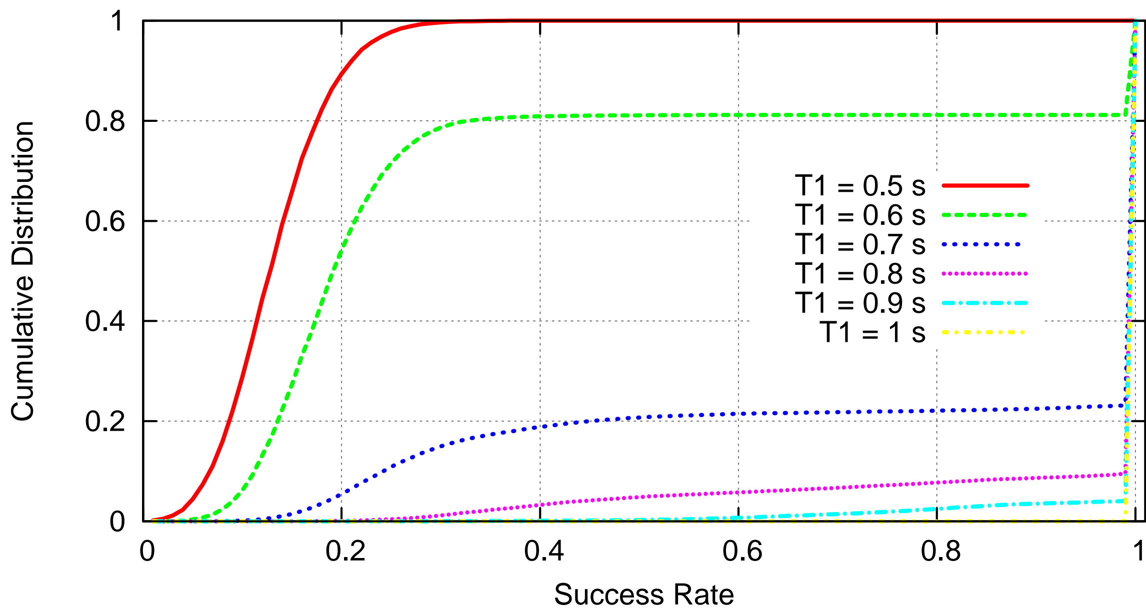
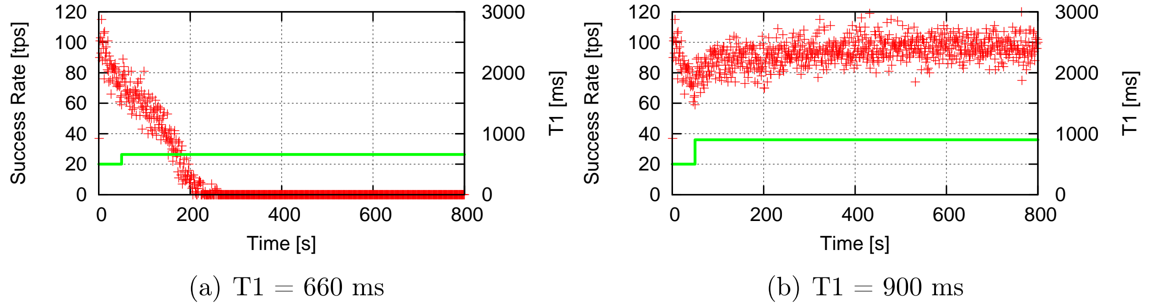


Figure 4.37.: Success Rate Distribution for multiple T1 settings

Adaptive Timer Manipulation

Summarizing the results presented so far, it can be concluded that increasing T1 from 0.5 s to 1 s results in more stability as none of the simulation runs has collapsed. In [16], we have derived a more precise value of T1 achieving this effect. Figure 4.37 shows the distribution of the success rate for multiple T1 settings. It can be observed that for a T1 of 0.5 s all runs are fully collapsed after the simulated 700 s, whereas, increasing T1 decreases the probability that a single run collapses.

However,too large values of T1 affect negatively the fRpD of messages lost on the link. For this reason we suggest to dynamically increase T1 after detecting an imminent



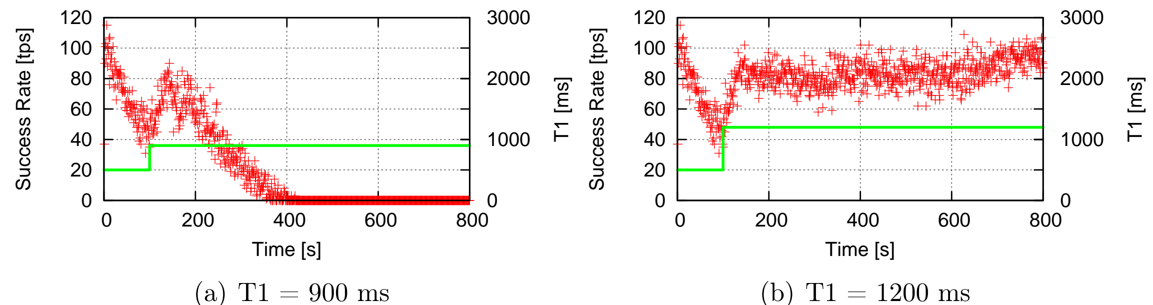
(a) T1 = 660 ms (b) T1 = 900 ms

Figure 4.38.: Success Rate when changing T1 after 50 s (red crosses) and T1 Setting (green line)

collapse. We show that the level of increasing T1 such that the system fully recovers highly depends on the already reached diminution of the success rate. This is also illustrated to some detail in [16].

Figure 4.38 depicts the case of increasing the value of T1 after 50 s, showing success rates as a scatter plot as well as the value of T1 as step function. Note that we observed that our previously demonstrated system (we have randomly chosen one out of our 5000 different runs) did not collapse if T1 is set to 660 ms. From figure 4.38(a), we conclude that a T1 value of 660 ms is no longer sufficient as soon as the collapsing process has already started. Instead, a much higher T1 value (e.g. 900 ms as shown in figure 4.38(b)) is required for the recovery.

The situation gets even worse if the T1 increase is delayed further. Figure 4.39 depicts the situation where T1 is kept at 500 ms during the first 100 s. In this case, increasing T1 to 900 ms turns out to be insufficient now, only a T1 value of 1200 ms leads to a satisfying result. Finally, from figure 4.39(b) we observe that in this case,



(a) T1 = 900 ms (b) T1 = 1200 ms

Figure 4.39.: Success Rate when changing T1 after 100 s (red crosses) and T1 Setting (green line)

the convergence to full recovery takes much longer than with figure 4.38(b) (and is in fact only reached beyond the 800 s simulation time used in our figures throughout).

Extension: Signaling Timer Modifications

As shown in the previous section, increasing the timer T1 adequately may prevent the entire system from collapse during overload situations. Of course, from there we may not conclude that this timer should be set to an arbitrarily large value, as on the other hand, low values of T1 are necessary for keeping the overall fRpD low in the case of link loss. In order to address this trade-off properly, we propose the introduction of an additional SIP-parameter in the Via-header in order to signal the currently optimal value of T1.
In a recent IETF draft on SIP overload control [21], Gurbani et al. introduce parameters that enable an overloaded server to signal its current state and a desired reduction of incoming requests to the upstream neighbor. To this end, five additional parameters to the Via header of SIP-responses are proposed as summarized in listing 4.1:

```
oc = "oc" [EQUAL 0-100]
oc-validity = "oc_validity" [EQUAL delta-ms]
oc-accept = "oc_accept"
oc-port = "oc_port"
oc-seq = (1*DIGIT) / (1*DIGIT "." 1*DIGIT)
```

Listing 4.1: Proposed Via Header Extensions

Here, `oc` (acronym for "overload control") defines the desired traffic reduction over a timespan defined by `oc-validity` (in ms). The third parameter `oc_accept` indicates the downstream neighbour that this SIP instance is accepting and understanding overload control messages in the presented form. The optional parameter `oc_port` specifies an upstream neighbour to apply the overload control only for subsequent requests sent to the specified port. `oc_seq` defines (optionally) a sequence number associated with the `oc` parameter.
As the proposed parameters are only referring to inter-server-communication, based on our simulation results we therefore suggest to add the following two new parameters (listing 4.2). `oc-t1` signals the modified T1 and `oc-t1-validity` corresponds to the validity timespan, over which this new T1 is valid.

```
oc-t1 = "oc_t1" [EQUAL new-T1]
oc-t1-validity = "oc_t1_validity" [EQUAL delta-ms]
```

Listing 4.2: Extensions to modify T1

Note that T1 is reset to its system-wide default value (usually 500 ms) as soon as the end of this validity timespan is reached. Moreover, we propose to extend the scope of these parameters beyond the original request-dropping `oc`-signalling in order to be able to modify also the behavior of the user agent clients. With this technique, we can reduce the load created on a border node without perceptible reducing the quality of the services from user perspective. We furthermore argue that in the case that the upstream neighbor is another SIP-proxy, modifying T1 (by adding the `oc-t1`-parameter) first and dropping messages with the `oc` parameter only if the modified T1 is not solving the overload situation is in general preferable to request dropping only as proposed originally in [21].

As an instructive example, a `200 OK`-response of the high-loaded node `reg.atl.com` answering a REGISTER request which uses the proposed extensions is presented in the following listing 4.3:

```
reg.atl.com -> proxy.atl.com

SIP/2.0 200 OK
Via: SIP/2.0/UDP proxy.atl.com:5060;
  branch=z9hG4bKnvzhgd4;
  oc-t1=1000;
  oc-t1-validity=60000
Via: SIP/2.0/UDP alicespc.atl.com:5060;
  branch=z9hG4bKnashds7;
  received=192.0.2.4
To: Alice <sip:alice@atl.com>;tag=249334
From: Alice <sip:alice@atl.com>;tag=45a4e33
Call-ID: 843817637684230@998sdasdh09
CSeq: 1826 REGISTER
Contact: <sip:alice@192.0.2.4>
Expires: 180
```

Listing 4.3: 200 OK-Response with T1 Modification Headers

In the response in listing 4.3, the overload control information is added to the topmost Via-header (as proposed in [21]). The `oc-t1` is set to `1000` and the `oc-t1-validity` is set to `60000`. These values result in a modification of the T1-timer of the `proxy.atl.com`-host to 1 second for a duration of 60 seconds for all requests sent to the overloaded host `reg.atl.com`.

Summarizing, we conclude that the integration of the presented additional parameters into the proposed overload-control-draft [21] provides a viable, cheap and efficient solution for the manipulation of the SIP retransmission timer.

4.3.3. Rate-Limiting

This section introduces a lossy mode method in order to reduce the outgoing load. For this purpose a limiter module that restricts the number of pending outgoing transactions[2] has been developed. That module limits the outgoing load by responding with *503 Service Unavailable* to requests that must be dropped in order to avoid the collapse. In order to detect an overload situation, the delay-based congestion detection (presented in section 4.2.2) has been used.
A basic load of 80 tps is offered to this system and after 50 s additional load (+40 tps) is injected that leads to a congestion collapse at the second proxy. The goodput of this simulation is depicted in figure 4.40. It appears that the value increases temporarily but as soon as the buffers at the proxy server are full, the system collapses and the goodput decreases down to about 15 msg/s (blue dots). On the other hand, the pink dotted curve shows the same results but with activated CD system. The number of successfully processed messages stays high and the system does not collapse. Finally,

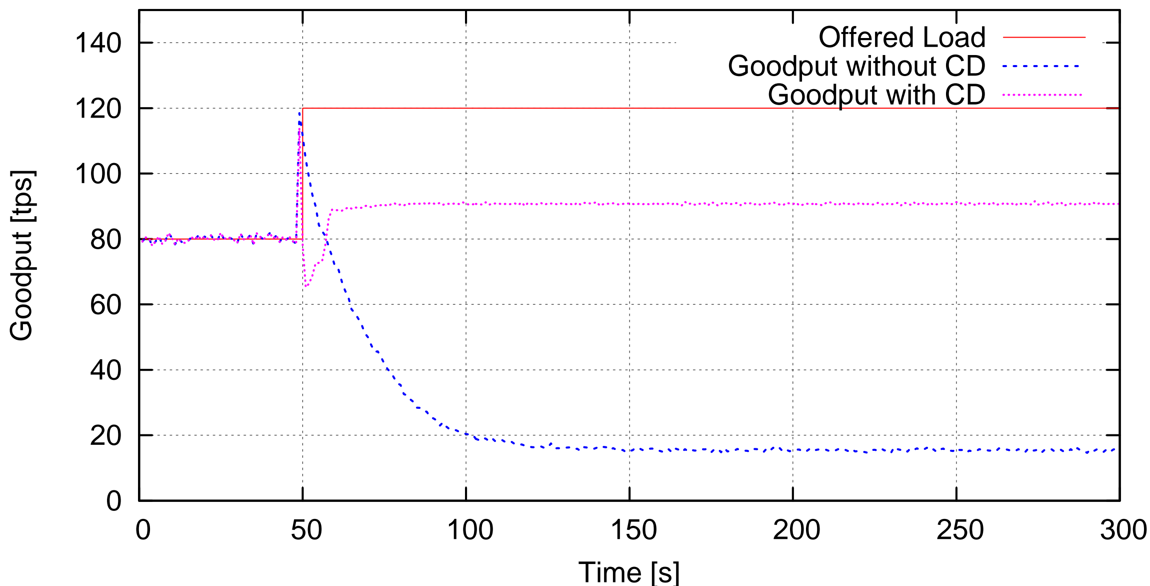


Figure 4.40.: Goodput with and without Congestion Detection

figure 4.41 shows the distribution of the fRpD across the whole simulation time. Without CD, the system collapses and the fRpD is distributed across all possible values 0..32 and a 95% percentile of 28.7 seconds. That leads to one or more retransmissions of nearly all transactions that cause additional load which after all is

[2]a transaction is pending when a request has been sent but no response has been received yet

unnecessary. With CD, 95% of the transactions are handled within 0.355 s[3] and only a few need to be re-transmitted. In the next step the mechanism has been analyzed

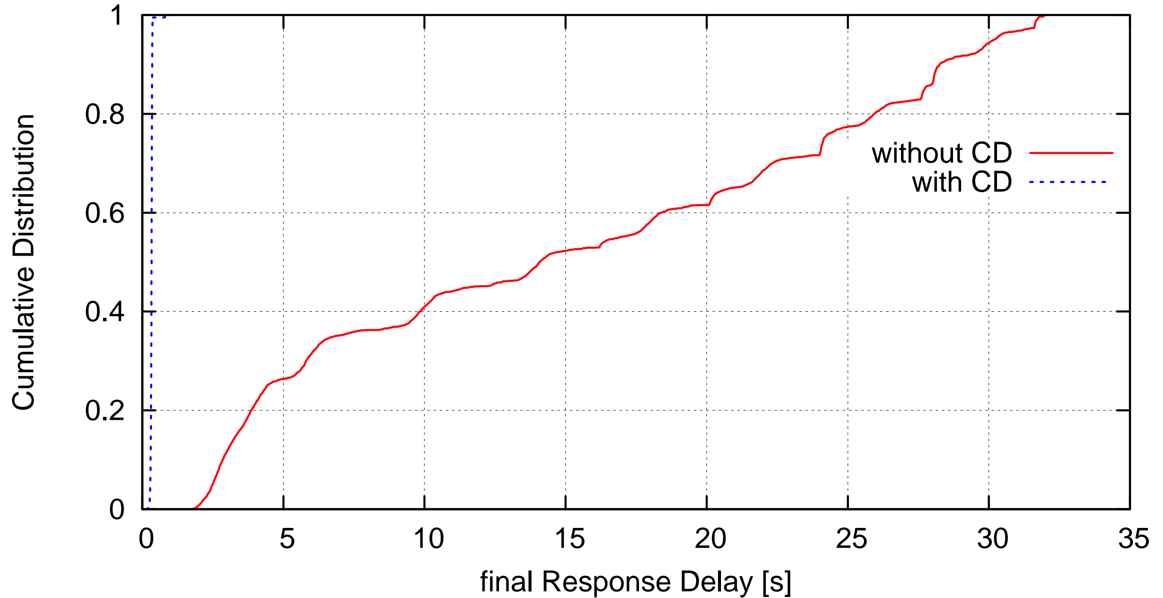


Figure 4.41.: Final Response Delay Distribution with and without Congestion Detection

for all load situations including overload. The focus for that is on steady state to get feedback on the quality of the algorithm in long-lasting situations. Figure 4.42 shows the goodput of the system without and with the implementation depending on the injected load. It appears that the system without CD is no longer able to handle the upcoming load of requests and collapses starting from 85 msg/s. Enabling CD allows a constant goodput up to the capacity limit and in overload situations as well.
These results demonstrate that limiting the number of pending transactions is a viable way to cope with a detected congestion. It allows to limit the outgoing load to a value that the next downstream host is able to handle and thus prevents congestion reliably.

4.4. Offloading

The previous sections have shown that congestion can be detected implicitly by analyzing information such as the final response delay or the number of pending transactions and how to cope with such situations in order to keep the throughput high

[3]this calculation does not differentiate between successful and rejected transactions

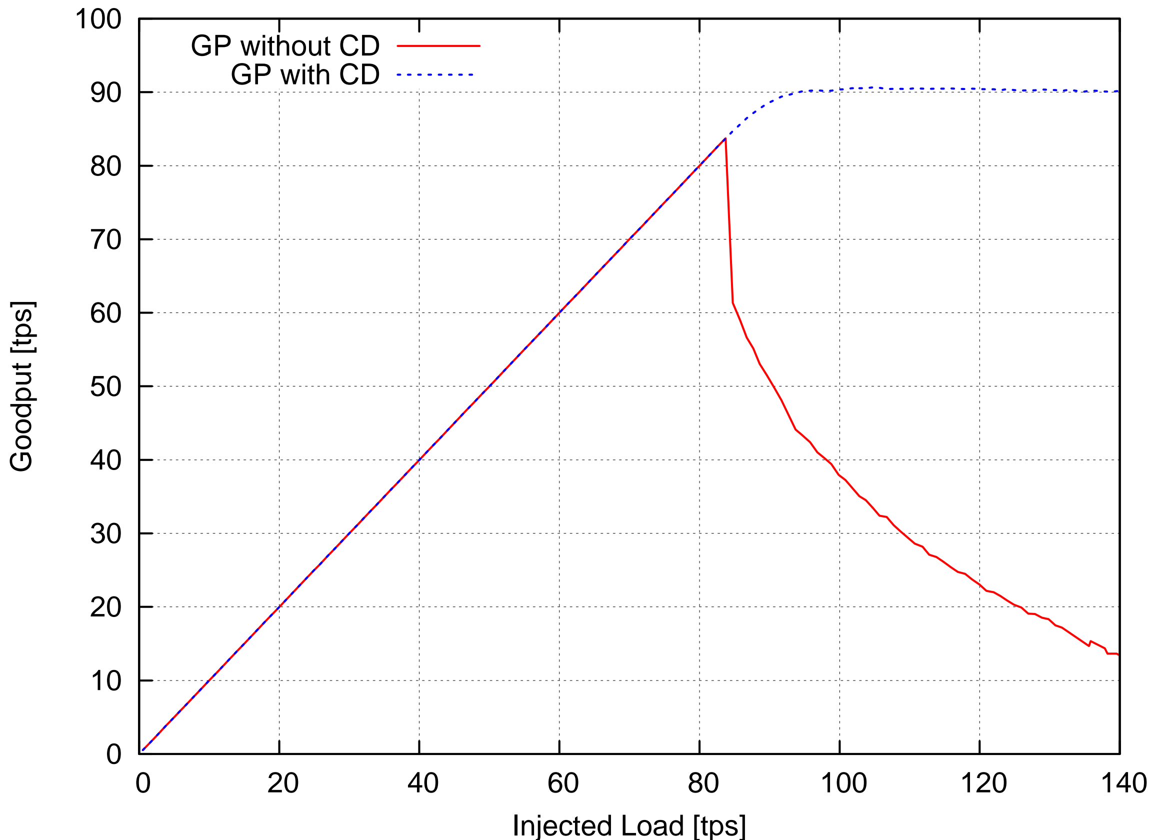


Figure 4.42.: Goodput against Injected Load

and the fraction of rejected requests low. However, these mechanism do not change anything of the implementation of SIP in the application software of the proxy server and it is interesting how optimizations in that area alter the behavior in congestion situations. One approach is for example, offloading some parts of SIP message processing inside the proxy server, especially parsing, to some specialized hardware or software. Another option is to profile the proxy server, to analyze where which modules consume most of the processing time and to optimize that specific part.

Offloading has been a controversial issue in the last decade, especially if it comes to TCP offloading [4]. The central idea behind it is to relieve the processor from protocol stack processing, and even encryption has been discussed, because it is an expensive feature in terms of computational effort [42]. But, as TCP is a highly complex protocol, and because of various other reasons [42], TCP offloading is implemented only rarely.

However, in the case of SIP, offloading of some protocol processing parts could be beneficial. SIP has been designed as a text-based protocol which enhances human

readability, but is detrimental in terms of processing efficiency. In [63], the authors have profiled the processing of two popular SIP proxy implementations, SER [33] and JSIP [49] and found that parsing takes up to 40 % of CPU time. They also implemented and tested a SIP Offload Engine (SOE) that transforms SIP messages into a binary format in a front-end and passes these binary encoded messages to the stack. Due to their research, these changes improve throughput by 100% and reduce the response time in overload situations.

This section uses this concept to verify if that approach changes anything to the earlier presented overload and congestion situations and potentially if SIP offloading can support in avoiding a congestion collapse.

4.4.1. Simulation Scenario

The simulation uses the same models of user agent and proxy that have been described in sections 3.3.1 and 3.3.2. The same simulation scenario as described in previous section 4.2.3 has been used, and is only described here in brief. It consists of an aggregated user agent client, an aggregated user agent server, and two proxies in between and is depicted in figure 4.43. Proxy 1 has ten times more processing

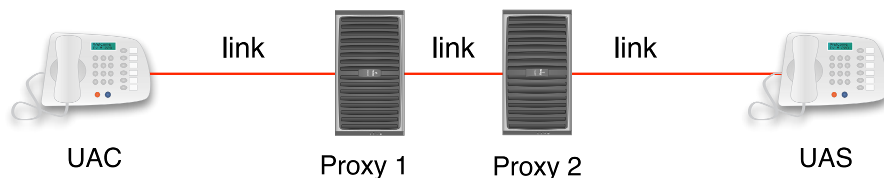
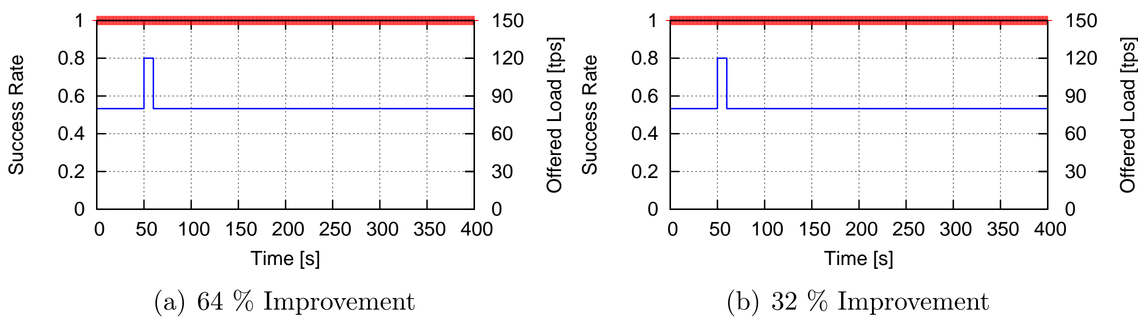


Figure 4.43.: Simulation Scenario

capacity than proxy 2 and therefore the second proxy is a bottleneck. To simulate the SIP offload engine the percentage of processor time that the parsing part needs is decreased, because the parsing of binary messages is more effective. According to [63], the amount of that reduction is 36 % for SER and 64 % for JSIP. Previously the model used in this thesis assumed that 20 % of CPU time is used for parsing and the following sections test the effects of decreasing that amount by these values (12.8 % and 7.2 % respectively).

4.4.2. Results

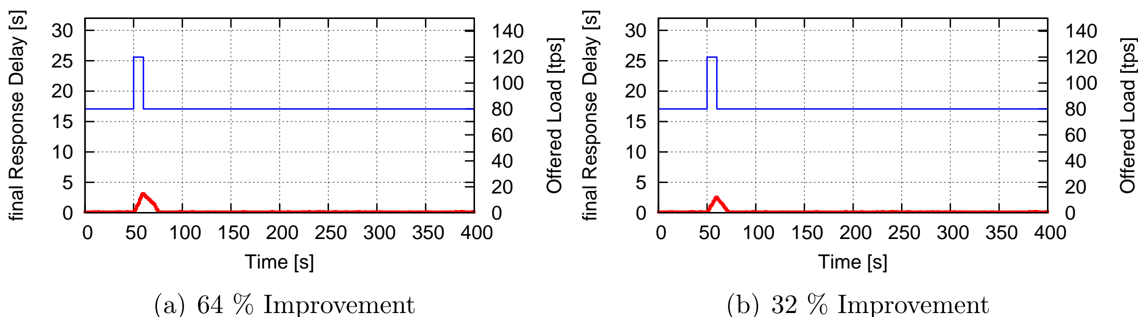
The same load patterns as used in section 4.1.1 are also used in this section. The system is started with 80 new transactions per second that are sent to the proxies, and after 50 s, an overloading peak of 120 new transactions is introduced for a duration of 10 s. That leads in the system tested in section 4.1.1 to a congestion collapse that can heavily recover by itself. Figure 4.44 shows the success rate with implemented SOE for both improvement factors. All transactions are handled successfully within the simulated time. Figure 4.45 shows the fRpD for the same scenario and that it in-



(a) 64 % Improvement (b) 32 % Improvement

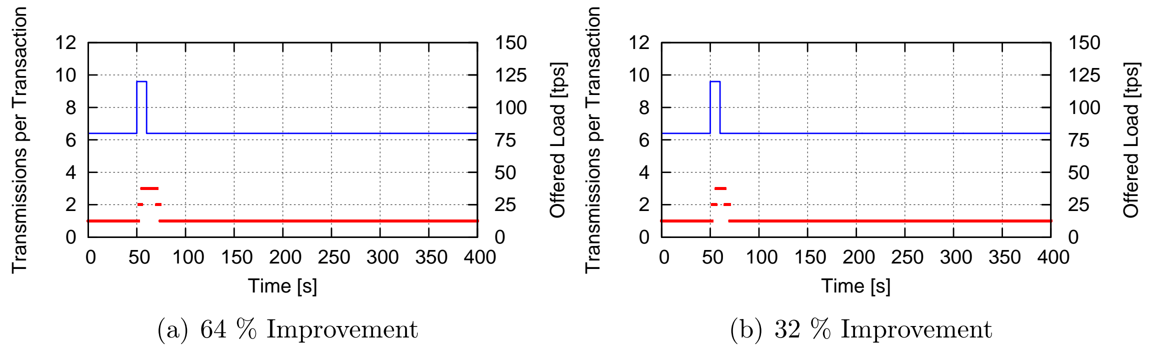
Figure 4.44.: Success Rate with SOE Improvement (red crosses) and Offered Load (blue line)

creases slightly for the duration of the overloading peak but decreases to its previous level afterwards. In figure 4.46 the total number of transmissions per transaction is shown and it is obvious that during the overloading peak the number increases for all transactions but drops to one transmission afterwards.



(a) 64 % Improvement (b) 32 % Improvement

Figure 4.45.: Final Response Delay with SOE Improvement (red dots) and Offered Load (blue line)



(a) 64 % Improvement

(b) 32 % Improvement

Figure 4.46.: Total Number of Transmissions per Transaction with SOE Improvement (red dots) and Offered Load (blue line)

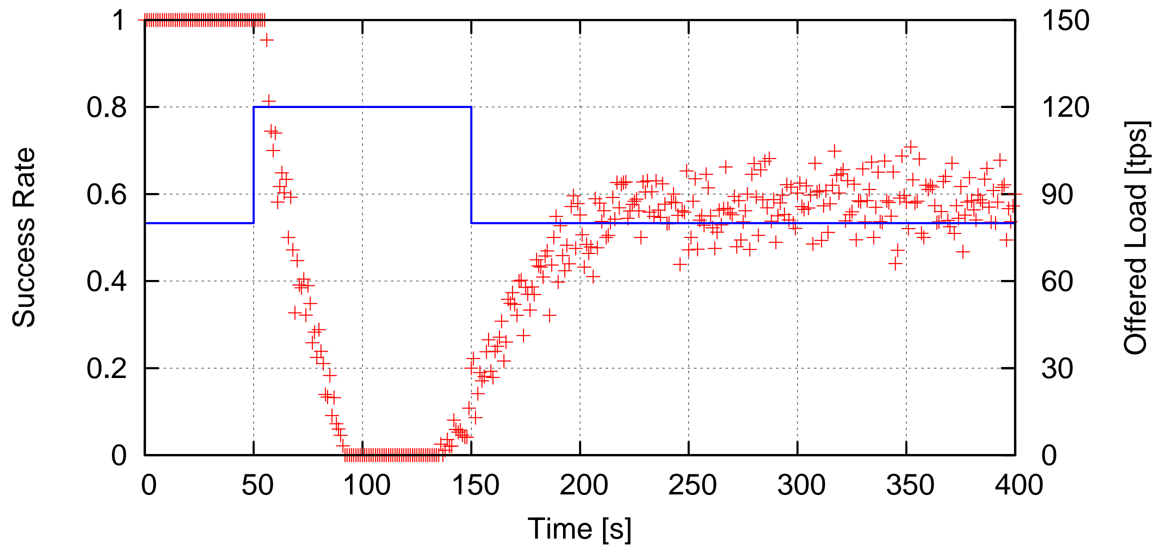


Figure 4.47.: Success Rate with SOE and prolonged peak (red crosses) and Offered Load (blue line)

The figures show that improving the efficiency of parsing significantly changes the behavior of the SIP proxy as it never reaches a congestion collapse for the simulated scenario and thus leads to considerable stability enhancement. However, increasing the background load to 90 new transactions per second and prolonging the peak to 100 seconds (instead of 10 seconds) as well as increasing the peak load to 150 new transactions per second again leads to a congestion collapse from which the system cannot recover by itself. Figure 4.47 shows the success rate of that simulation. While the peak load is active, the success rate decreases down to zero (that means also that all transactions are transmitted the maximum number of 11 times) and as soon as the

load returns to its previous level of 90 new transactions per second the SR increases to about 0.6. This means that still many transactions are re-transmitted 11 times even if the load is below the capacity limit of the proxy server.
It can be concluded that SOE is indeed able to increase the performance of a SIP system and to decrease the probability of a congestion collapse, but in more extreme situations better methods for overload protection are necessary. That finding is also in accordance with the results of [63].

4.5. Application Scenarios

The previous section has shown that congestion can appear in high-load situations and that appropriate countermeasures can improve the goodput and the time to complete recovery. This section presents real-world scenarios that may lead to such negative effects and how the previously introduced congestion detection mechanisms affect the overall situation. The examinations of this section use short-time period effects as source of high load that can appear in several situations:

- partial infrastructure outage
- "call-now" lotteries
- public holidays, especially New Year's Eve
- begin, midday and end of working days

During or after these events many user agents try to send messages to other user agents or servers, therefore causing overload in scenarios demonstrated below. As first scenario, the standard presence service has been chosen because it creates already a huge number of messages in normal operation, leading to a flood of messages if many users update the presence state of their buddy list within a short time period. The second scenario is the registration service that requires by design two consecutive requests to be sent to a registrar server. This may also lead to a huge amount of messages if many users register their UAs in a short time period.

4.5.1. Presence

The presence service distributes information about the state of a user to a list of buddies. For this purpose it uses the event notification framework extension for

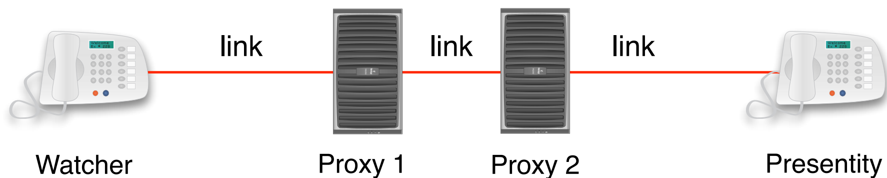


Figure 4.48.: Simulation Scenario for Presence Service

SIP [50] and the event state publication mechanism, defined in [44]. To demonstrate the performance problem of this service in a worst-case scenario, the point-to-point presence has been chosen, which sends presence messages directly to other user agents (see also section 3.3.5).

Simulation Scenario

The scenario used for this simulation, depicted in figure 4.48, has two proxy servers where *proxy 2* plays the role of a bottleneck and a watcher that subscribes to the presence state of the presentity. The watcher and the presentity represent a variable number of users and each watcher has to subscribe to the presence state of its presentities. The presentities notify all subscribed watchers on any presence status update.

For performance evaluations an estimation of the number of messages that the watcher and the presentity exchange is presented as follows. The number of messages exchanged between watchers and presentities per day depends on the number of users n_u, the number of buddies in the user's buddy lists n_b and the number of presence updates a day n_p. This causes $n = n_u \cdot n_b \cdot n_p$ messages a day for updating the presence state (hereby neglecting messages for maintaining the subscriptions). The authors of [30] assume that a domain has 40.000 watchers that have 4 presentities in their buddy list and the presentities change their state 3 times per hour. Then the total number of messages n that are sent due to state changes amounts to 11.520.000 per day or 133 per second. However, new services that use the SIP presence extension such as location could cause more presence state updates. This estimation shows that the presence service is ineffective if used in the way standardized by [50] and [44]. Anyhow, these values are exaggerated to be directly used by simulations because the same effects can be shown with lower amounts. Because of huge resource requirements of large simulation scenarios, the author has chosen to simulate a smaller amount of

messages and SIP nodes.
However, the following examinations of the presence service use a mandatory re-subscription to the presence state because of a partly infrastructure failure of an operator. Due to that failure situation, a part of the clients have to re-subscribe to the presence state of the users in their buddy list as soon as the failed part of the infrastructure is re-started and they therefore create the following messages.

- One SUBSCRIBE message to each presentity in the list
- One NOTIFY message sent from each presentity to subscribed watchers

To stay consistent with the values chosen in sections 4.1.1, the following assumptions have been made for the simulations of this section. The processing capacity of *proxy 2* is set to 100 transactions per second and background traffic generates 80 transactions per second. To create an overloading situation like in section 4.1.1, the number of watchers that have to re-subscribe is set to 50 with 4 presentities each. All watchers send the new subscription within 10 seconds after re-start of the failed part. This leads to 200 SUBSCRIBE messages that are sent within 10 seconds or 20 per second plus one NOTIFY for each SUBSCRIBE that has reached the presentity, totaling 40 additional messages per second. Thirty seconds after the simulation starts, the watchers start to subscribe to the presentities and 250 seconds after the presentities have received a subscription message, they change their presence state. If a watcher does not receive a notification within 90 seconds after initial sending of the subscription message, it re-sends the subscription request.

System without Protection

This section presents the behavior of the previously described setting for a SIP system that has no protection against congestion. At the beginning, the watchers start an initial subscription to the presence state of the presentities during a period of 10 seconds. Subsequently, the presentities send notifications to their watchers whenever they change their presence state.
Figure 4.49 shows the number of active subscriptions against time for that system. A subscription is considered as active as soon as the watcher receives a NOTIFY message. After the start of the presence service the number of active subscriptions increases until all 200 subscriptions are active. The processor utilization of *Proxy 2* is shown in figure 4.50 and stays at about 0.8 before the presence service starts because of the background load. As soon as the presence service starts, the processor

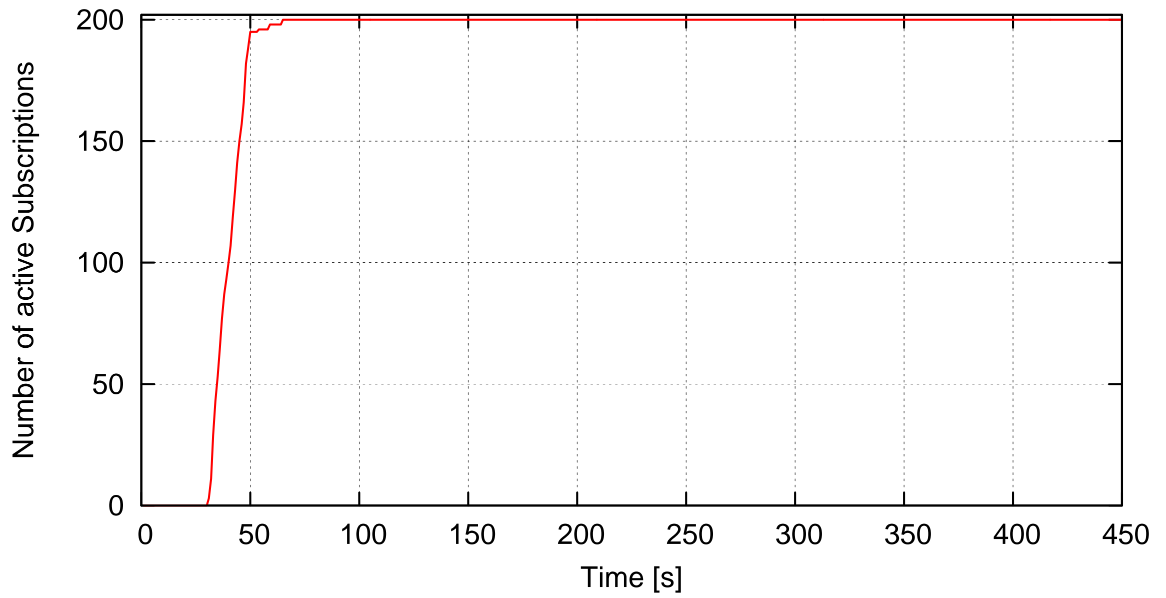


Figure 4.49.: Number of Active Subscriptions for a System without Protection

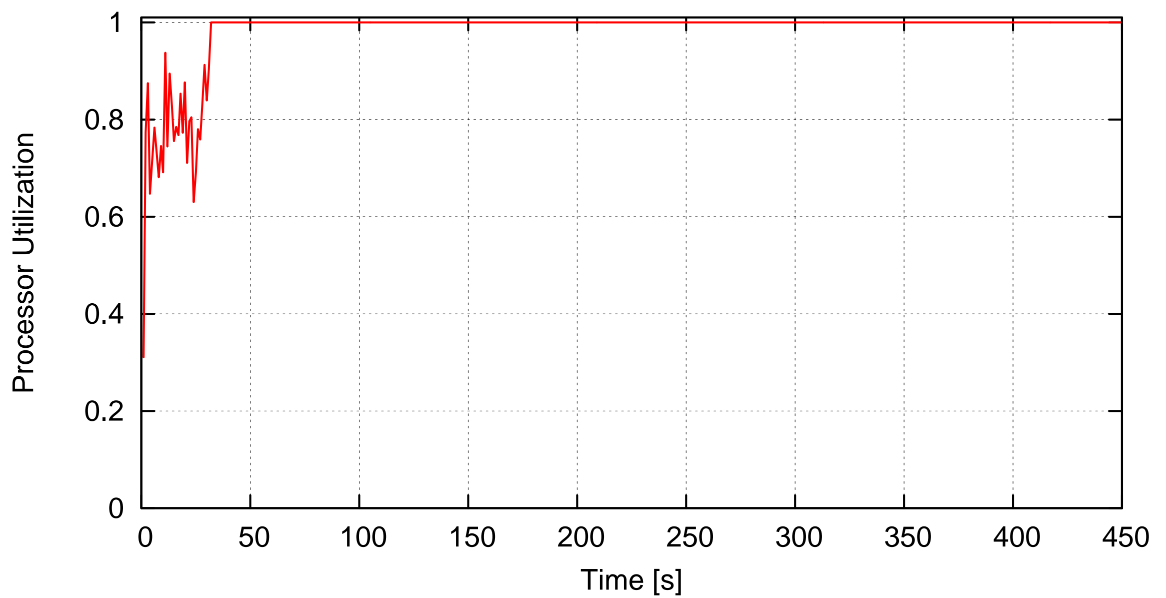


Figure 4.50.: Proxy 2 Processor Utilization for a System without Protection

utilization increases to 100 % and stays at this value even though the sending of new subscriptions ends with $t = 40$ s. The value does not decrease back to the previous level because the system has run into a congestion collapse. Therefore, all transactions are re-transmitted because they do not receive a final response within 500 ms even though the overloading peak already disappeared[4].

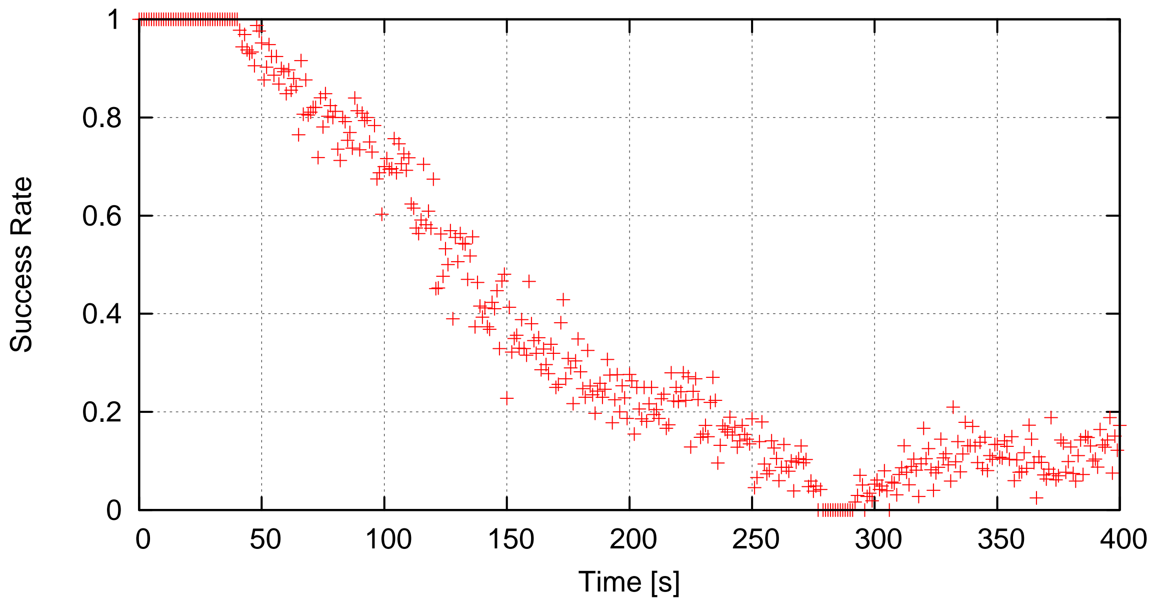


Figure 4.51.: Success Rate for Background Traffic for a System without Protection

Figure 4.51 shows the success rate for the background traffic that decreases immediately after the start of the presence service and that reflects perfectly the congestion collapse situation. At $t = 280$ s the success rate decreases further because the presentities change their presence state and notify their watchers. This creates additional traffic that further increases the load on *proxy 2*. Totally, 33.3% of all requests sent during the overload situation ($t > 30$ s) have been successful, that means, have received a *200 OK* response.
Figure 4.52 shows the total number of transmissions per transaction, illustrating that during the re-subscription of the watchers all transactions are sent multiple times. This creates a huge unnecessary load on the proxy server that leads to a congestion collapse, the system being unable to recover by itself from this situation.
Summarizing, this system suffers from a congestion collapse because a partly failure of the infrastructure has resulted in a re-subscription of the presence state of the

[4] compare section 4.1.1

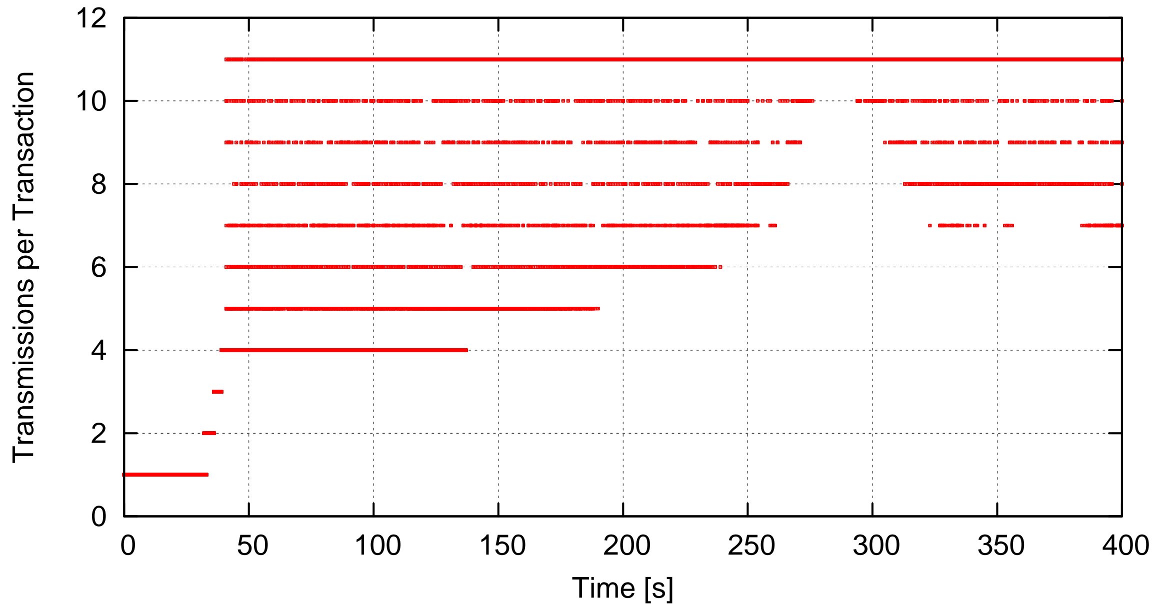


Figure 4.52.: Total Number of Transmissions per Transaction for a System without Protection

watchers. The system is not able to recover from this state and further load caused by status updates worsens the situation.

Delay-Based Detection

The previous section has shown that a system without congestion detection suffers from unnecessary load and runs into a congestion collapse caused by presence subscriptions. This section shows that implicit congestion detection as presented in section 4.2.2 avoids unnecessary load and leads to immediate recovery from overload. The number of active subscriptions, shown in figure 4.53 increases in three steps. The first step represents subscriptions that receive a notification within the first 10 seconds. However, due to rejects that occur because of overload, that is not the case for all. The second step are subscriptions that have not been successful in the first try and therefore re-send the initial subscription, whereas the third step corresponds to subscriptions that have been rejected in the second try. Eventually all 200 subscriptions are active.
The resource-usage of the delay-based detection system is lower than for the system without any protection, as shown in figure 4.54 that depicts the processor utilization. As soon as the presence service starts (simulation time $>$ 30 s), the values increase

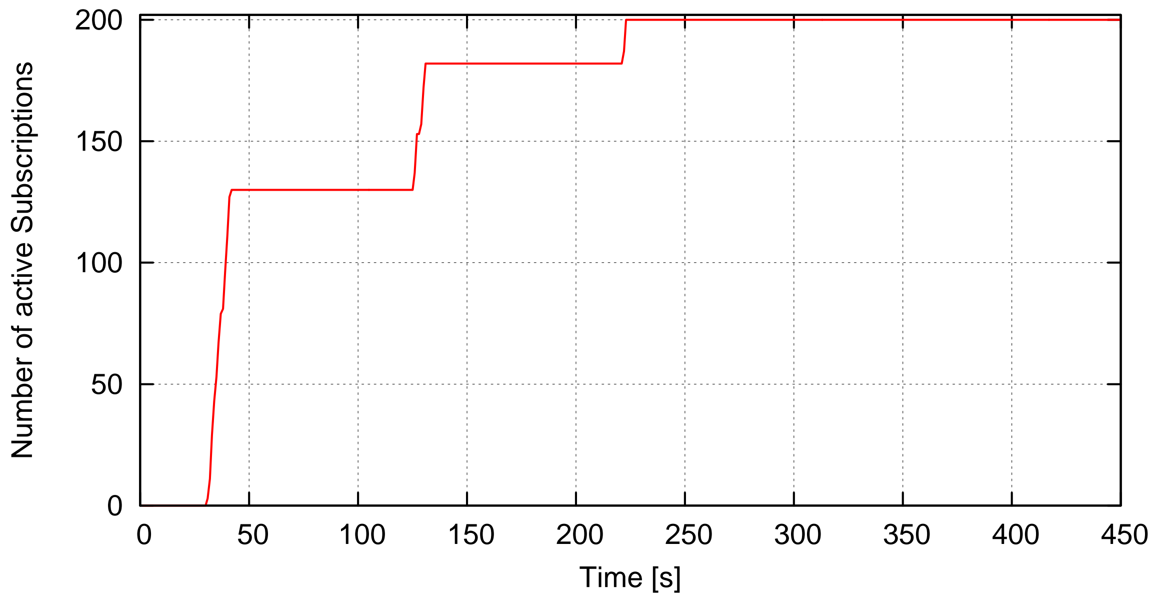


Figure 4.53.: Number of Active Subscriptions for Delay-based Detection

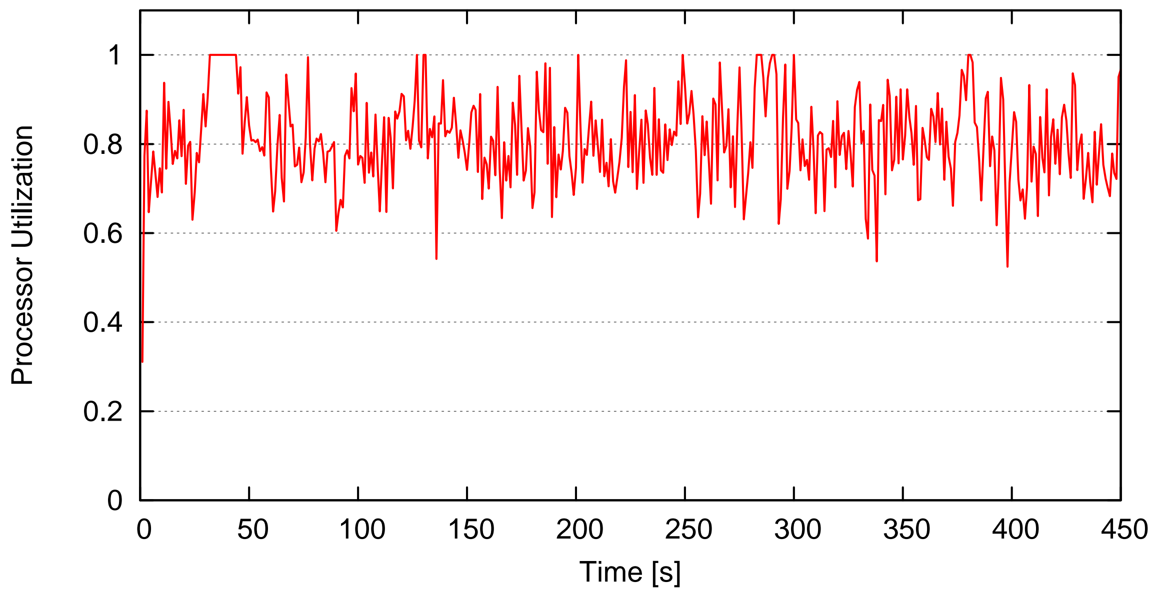


Figure 4.54.: Proxy 2 Processor Utilization for Delay-based Detection

up to the maximum and decrease to the previous level immediately after the initial subscription messages have been sent.

For the background traffic the situation is considerably enhanced as the success rate shown in figure 4.55 stays consistently at 100 %. Note that a transaction is considered

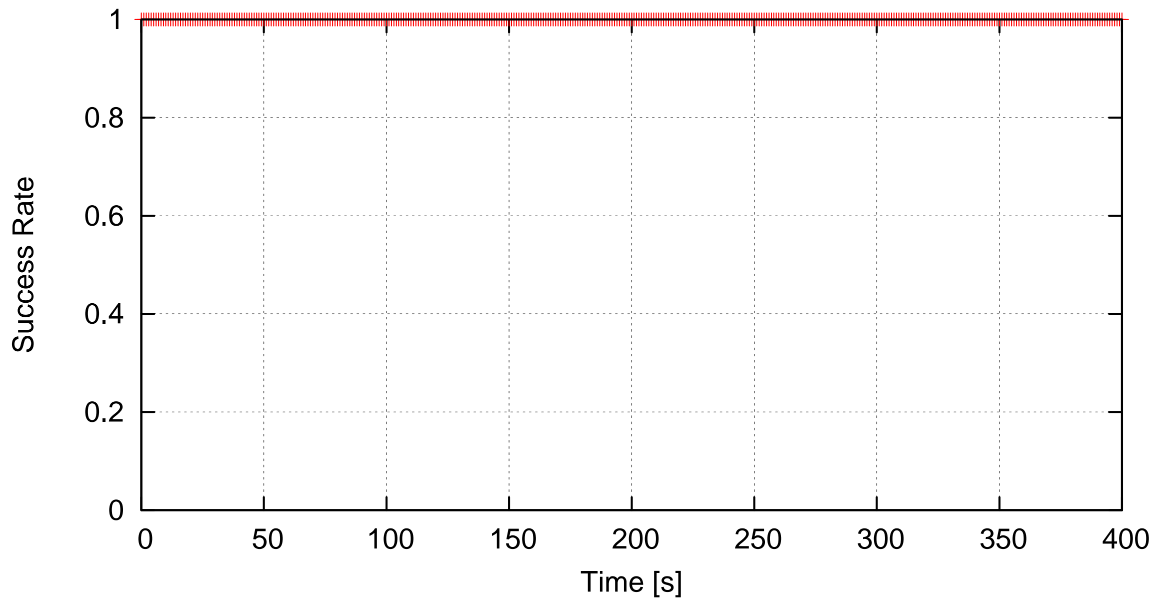


Figure 4.55.: Success Rate for Background Traffic for Delay-based Detection

to be successful if it has received a final response, that means a *200 OK* response as well as all other response codes except *1xx*. Because a part of the incoming messages have to be rejected because of overload, *Proxy 1* sends *503 Service Unavailable* responses to that part. A UAC that receives a 503 response to a request completes indeed the transaction, but the service request is not fulfilled in this case. For that reason the ratio of successful service requests (that is, transactions that received a 200 OK) to unsuccessful service requests (that is, transactions that received a 503 Service Unavailable) has been calculated. For this system using delay-based detection this ratio amounts to 99.1% (compared to 33.3% for the system without any protection).

The total number of transmissions per transaction is shown in figure 4.56. Only a few transactions need to be re-transmitted, which is a major enhancement in comparison to the system without any protection. Less unnecessary load is created and a congestion collapse is avoided. The system remains fully functional and nearly all service requests are handled positively, that is, all requested presence state subscriptions are active and 99.1% of the background traffic is handled.

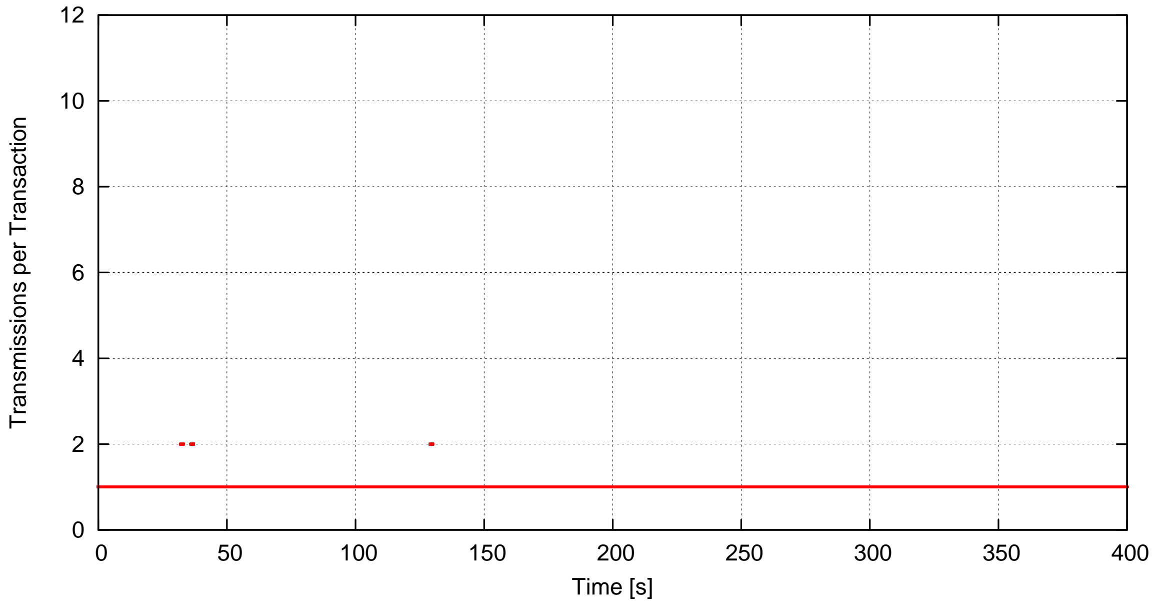


Figure 4.56.: Total Number of Transmissions per Transaction for Delay-based Detection

Pending-Transaction-based Detection

The application of the anomaly indicator onto the presence-service shows that pending-transaction-based detection can avoid a congestion collapse too, and that the system immediately recovers from this overload situation. Figure 4.57 shows again the number of active subscriptions that increases in two steps up to the number of 200 subscriptions. After 130 seconds, 100% of all possible subscriptions are active, which is faster than the delay-based system, that needs 100 seconds more. The processor utilization (figure 4.58) increases up to the maximum during the initial subscription but decreases immediately afterwards to the value caused by the background load. The success rate of background traffic stays at 100% throughout the whole simulation duration and 97.8% of all transactions receive a 200 OK response. The total number of transmissions per transaction shows a similar shape than for the system with delay-based detection. Only a few transactions need to be re-transmitted. Summarizing, the activation of the pending-transaction-based detection avoids a congestion collapse and significantly reduces unnecessary traffic.

At a glance, enabling any of the two presented implicit congestion detection mechanisms enhance the overall situation in a system that is flooded by a huge load of messages and reduces the number of redundant re-transmissions. The delay-based

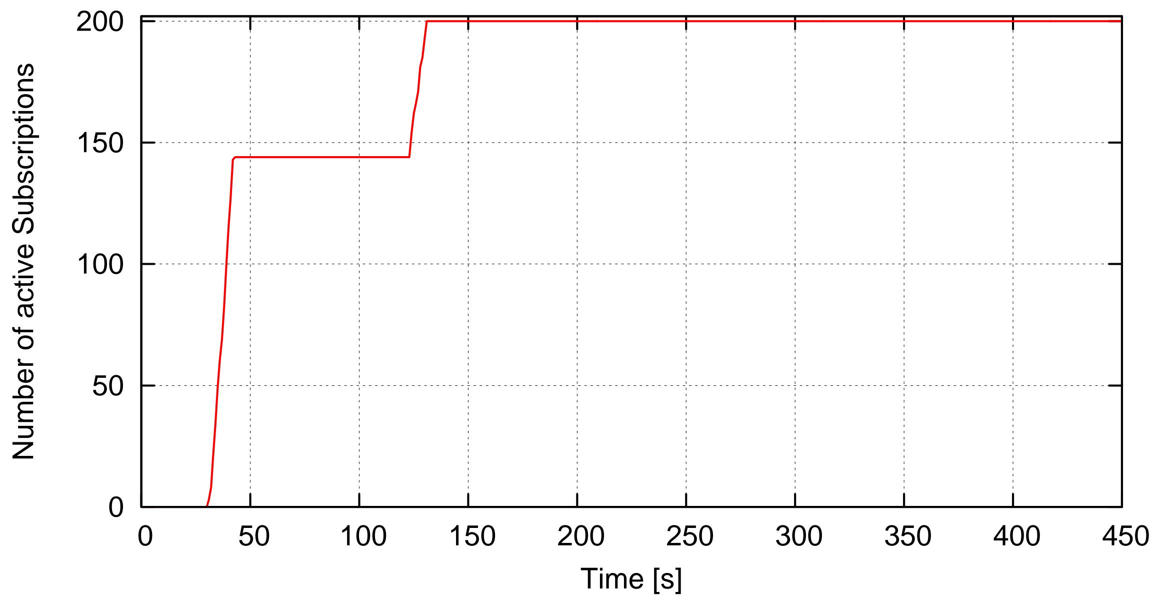


Figure 4.57.: Number of active Subscriptions for Pending-Transaction-based Detection

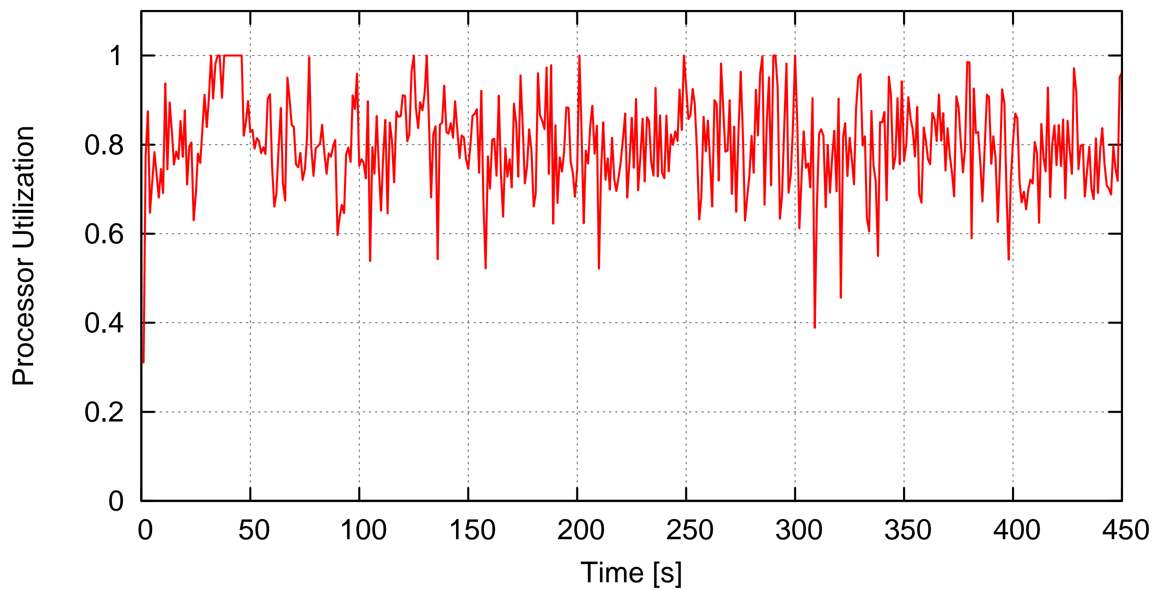


Figure 4.58.: Proxy 2 Processor Utilization for Pending-Transaction-based Detection

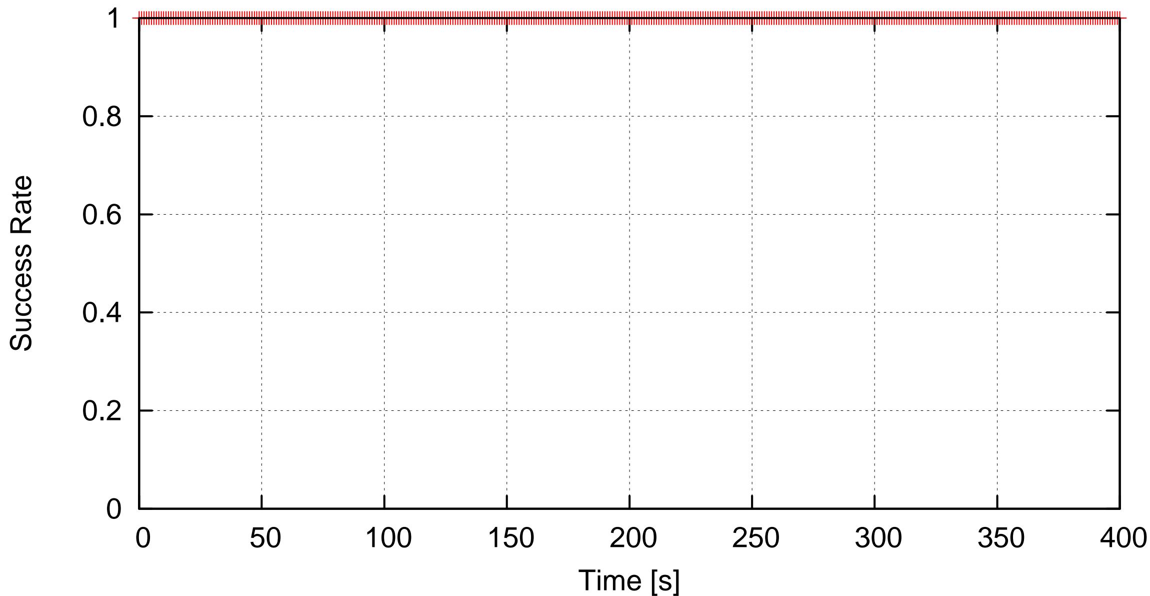


Figure 4.59.: Success Rate for Background Traffic for Pending-Transaction-based Detection

detection on one hand requires more time until all subscriptions are active but handles slightly more background traffic. The pending-transaction-based detection on the other hand, allows overall a faster subscription with the drawback of a slightly lower success rate than the delay-based mechanism.

4.5.2. Register

The SIP registration is necessary if a UA shall be reachable using a generic URI (for example sip:egger@tuwien.ac.at) rather than a plain IP address (see also section 2.1.1). Typically, a SIP registrar challenges the UAC for credentials after receiving a registration request by sending the nonce, a unique number that is only used once. The UAC uses this number to encrypt the credentials and sends a second registration request to the registrar that contains this encrypted response. If the credentials are correct, the registrar sends a 200 OK message back to the UAC that contains a time that denotes when that registration expires. The UAC has to re-register within that time in order to keep the registration active. These consecutive requests may create an overloading peak that repeats every time when the clients refresh their registration. This bulk re-registration overload is analyzed in this section. An application specific optimization that avoids this re-register synchronization can be

achieved by randomization of expiration time. Because an operator can not trust that the user's clients support random expiration time, the registrar must send a expiration time to the client that lies between a pre-set minimum and the client-requested value or a pre-set maximum.
The following sections present the number of active registrations throughout the simulation time. A registration is considered active after the client has received a 200 OK message to the second registration request (the one including credentials) and within the agreed registration interval.

Simulation Scenario

The registration scenario consists of a UAC that represents several users, two proxies, where *proxy 2* plays as in the previous section the role of a bottleneck and a Registrar that stores the registrations of the UACs (Figure 4.61). Again, a background load of 80 transactions per second is generated and *proxy2* has a processing capacity of 100 transactions per second. To simulate the same load pattern as previously, the UAC represents 200 clients that register themselves within 10 seconds 30 seconds after the simulation started. This results in additional 40 transactions per second, that are 20 initial registration messages per second and 20 registration messages per second that

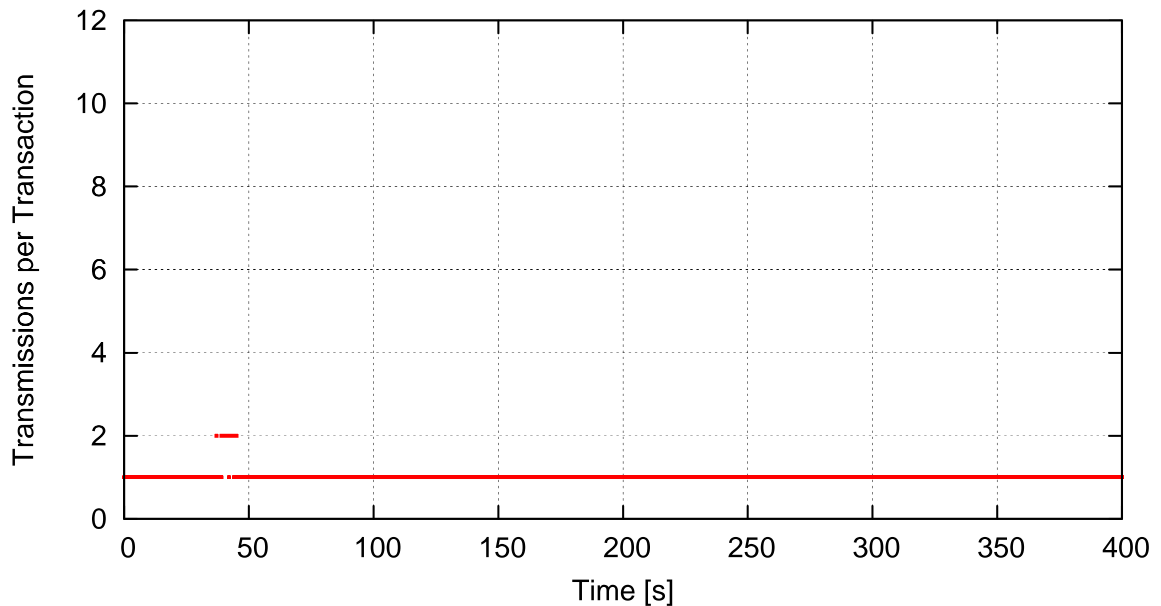


Figure 4.60.: Total Number of Transmissions per Transaction for Pending-Transaction-based Detection

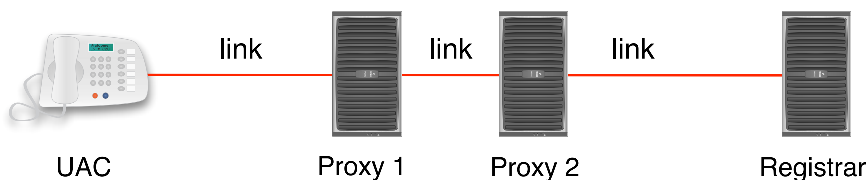


Figure 4.61.: Simulation Scenario for Registration Service

contain credentials. In order to present the effects of re-registration, the expiration time is set statically to 300 seconds[5]. If a registration is unsuccessful either because no response has been received or if expired, the UAC retries after 10 seconds. The total simulation time is expanded to 1000 seconds in order to present multiple expirations and re-registrations. The registrar is configurable to send a random expiration time to the client, that is distributed uniformly over a pre-set minimum of 5 seconds and a maximum of 300 seconds. The minimum value of 5 seconds has been chosen because the SIP standard [53] defines a *Timer K* for the Non-Invite state machine. This timer has been designed to let the client state machine wait for potential response re-transmissions and after this timer expires, the state machine can be deleted. In order to save resources on the machine running the SIP client it is required to wait until the previous state machine is destroyed before creating a new one.

System without Protection

This section illustrates the behavior of a system without any protection that suffers from a high number of clients that register within a short period. A static expiration time is configured for the scenario before the registrar uses a random expiration time.

Static Expiration Time

The number of active registrations, shown in figure 4.62 increases to the maximum of 200, but the additional traffic is enough to bring the system into a congestion collapse. At $t = 330$ s, the clients start to re-register because the expiration time is reached. Due to the occurrence of the congestion collapse situation no attempts succeed and eventually no registrations are active. The system remains in the congestion collapse situation and is not able to recover by itself. At $t = 810$ s, a few clients manage to

[5]A major austrian operator uses this value in a commercial SIP deployment

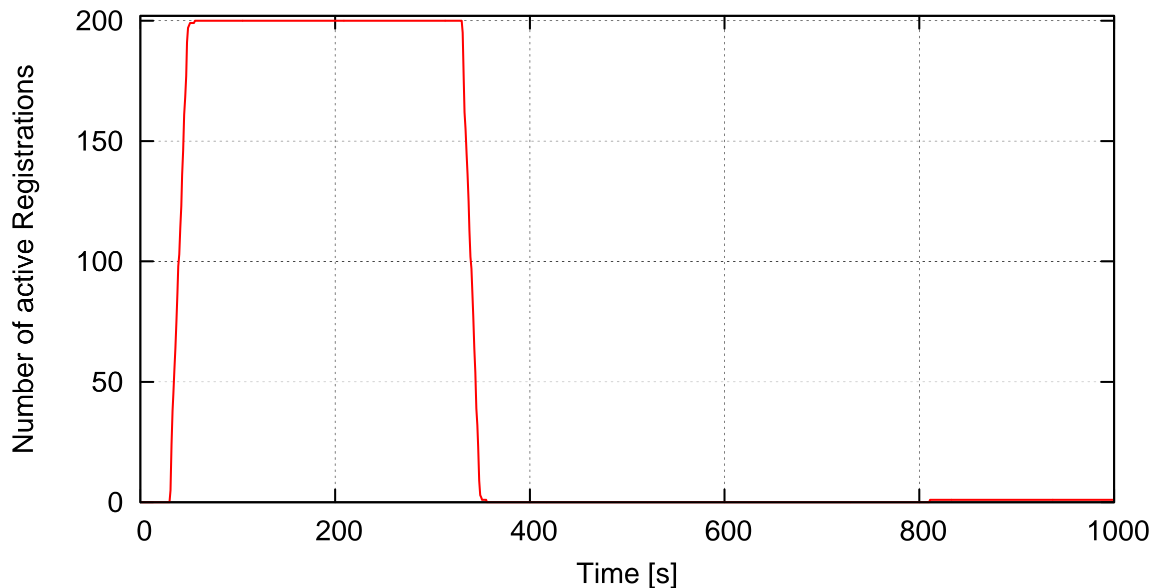


Figure 4.62.: Number of Active Registrations for a System with Static Expiration Time and without Protection

activate their registrations, but 300 seconds later they expire because the congestion situation is still present.

Random Expiration Time

Randomization of the expiration time appears to relieve the system partly because the re-registration attempts do not appear as a single bulk but uniformly distributed over a specific time frame. Simulating the same scenario but including randomized expiration times gives the results depicted in figure 4.63. The graph does not show any improvement because the collapse is already caused by the initial registration requests and a randomization of the expiration time simply results in earlier expiry of some registrations and is thus no enhancement in such a situation.

Delay-Based Detection

The same scenario has been simulated with activated delay-based congestion detection. The number of active registrations is plotted in figure 4.64. The value increases up to the maximum of 200 possible active registrations and stays at this value throughout the entire simulation time. The system does not fall into a congestion collapse and all transactions are handled successfully. An expiration time randomiza-

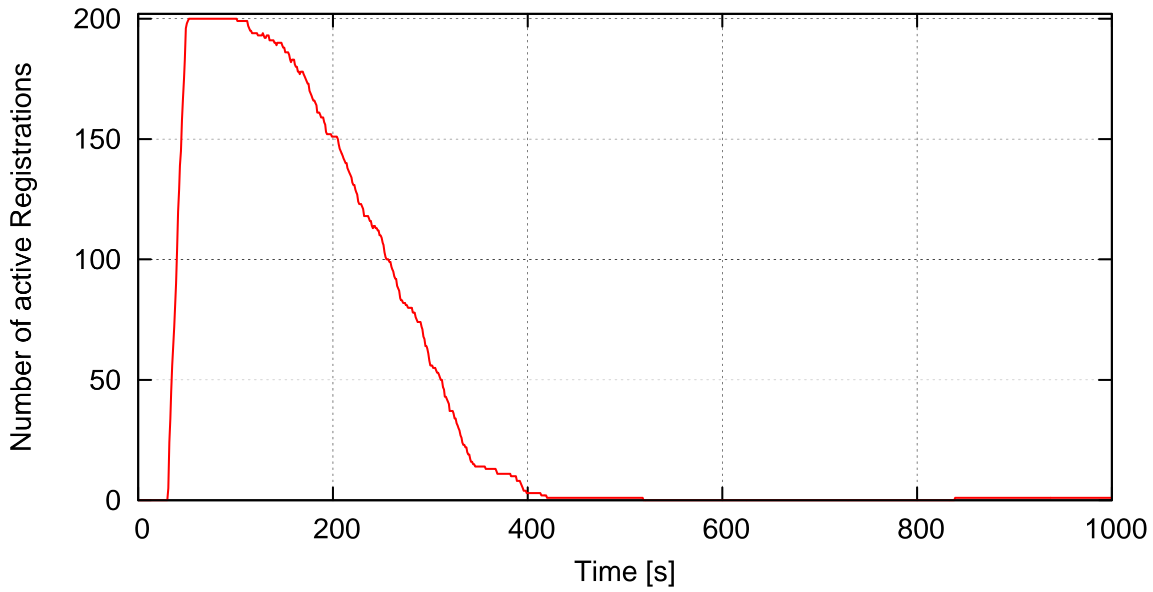


Figure 4.63.: Number of Active Registrations for a System without Protection and Randomized Expiration Time

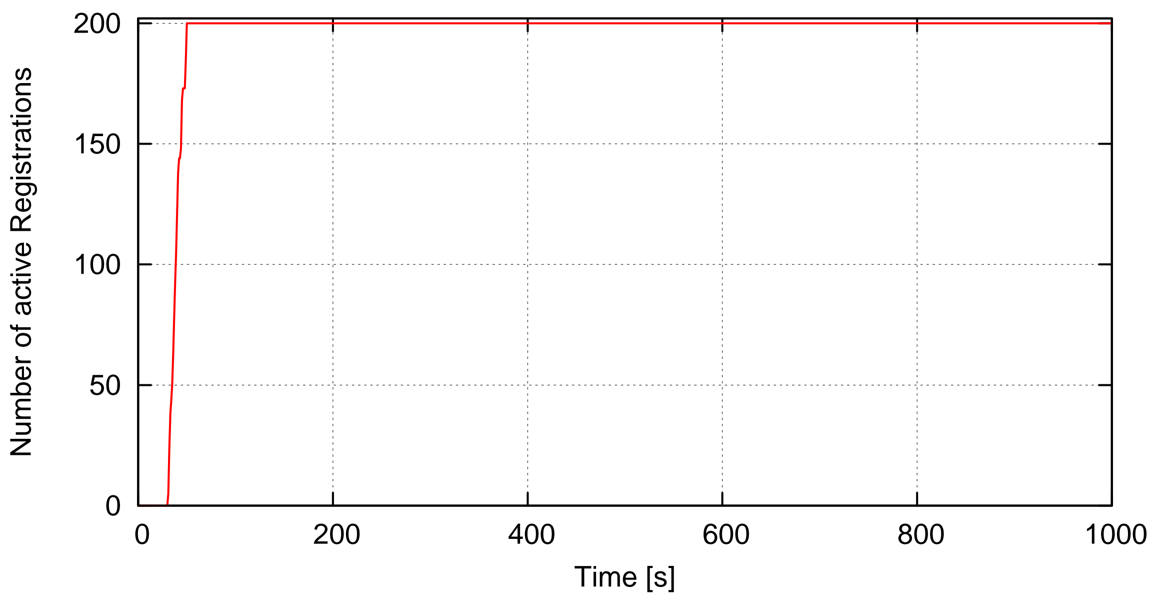


Figure 4.64.: Number of Active Registrations for Delay-based Detection

tion does not enhance the performance because all registrations are already active. Concluding, the activation of delay-based detection prohibits a congestion collapse and enables the system to handle more clients.

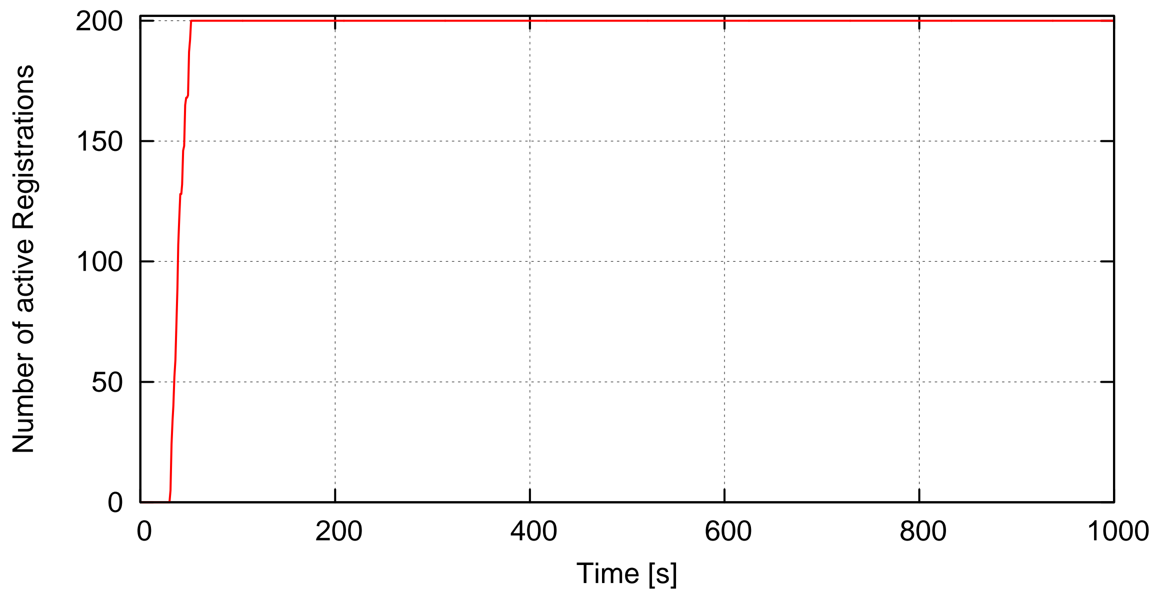


Figure 4.65.: Number of Active Registrations for Pending-Transaction-based Detection

Pending-Transaction-based Detection

In this section the pending-transaction-based detection mechanism has been tested for the registration service. Figure 4.65 shows that the number of active registrations increases up to the maximum of 200 and stays at this value throughout the entire simulation duration. Additional expiration time randomization does not yield any improvements as all possible registrations are continuously active. For the pending-transaction-based detection the same conclusion as for the delay-based detection can be drawn. The system does not fall into a congestion collapse and the system can handle all clients as required.

4.6. Summary and Strategy Recommendations

This section has shown that the new simulation environment that has been built for this thesis is able to reproduce known performance drawbacks of SIP. It has demonstrated that a congested SIP system leads to multiple redundant re-transmissions of requests and that results in additional load on that system. Two approaches have been developed. The first one, a delay-based congestion detection mechanism, which tracks the fRpD of all SIP transactions that occur within a certain interval and uses

the 95% percentile of these values as trigger for subsequent actions. If the percentile is above 500 ms, a system is considered to be congested, because it is assumed that a non-congested system does not require any re-transmissions. The second approach uses the number of pending transactions and the number of sent requests and calculates a so-called Anomaly Indicator that is as well used as trigger for counteractions. This value is calculated as the quotient of the first derivation with respect to time of these numbers. Counteractions are partitioned into so-called lossless modes and lossy modes. Lossless modes try to give the system sufficient time to handle the upcoming load without rejecting any requests. Lossy modes reject a part of the incoming requests in order to reduce the load on a congested system. It has further been shown that offloading cannot completely prevent a congestion collapse because it just shifts the point of load where the system flips into collapse. Scenarios for common applications have shown that overload in SIP systems can evolve out of typical usage.

From the results of this chapter can be concluded that it is worth to implement an implicit congestion detection mechanism. They are resource-saving and particularly do not require any implementation on systems that are closed source and thus immutable. However, in case of huge traffic and delay variations, implicit congestion detection mechanisms might operate unreliably. In these scenarios we recommend to use it as a trigger for lossless congestion handling and avoid lossy operation.

5. Measurement-based Verification

The last sections have shown that an improvement of the SIP signaling performance can be achieved by means of different methods. Measurements of implicitly available values and appropriate adaption of the outgoing rate of new transactions or of re-transmission timers can result in a major improvement. However, these results have been gained through simulation of the protocol and are therefore highly dependent of the simulation scenario configuration and the quality of the used simulation models. These configurations and models have been built to the best of the author's knowledge and belief, but in order to enhance the quality of the results, the simulated mechanisms have been implemented in a physical hardware SIP proxy server and thoroughly tested. The following sections show the set-up of the scenario for the measurements and the congestion detection mechanisms that uses the final Response Delay as well as the newly developed anomaly indicator as a trigger. Main goal of this section is therefore to reproduce the simulation results that have been presented in section 4.2 in a real world system.

5.1. Measurement Set-Up

For the measurements a Java-based SIP proxy has been used that has been implemented at the Institute of Telecommunications at the Vienna University of Technology. This proxy has been chosen because it is designed for easy integration of new overload detection mechanisms, either explicit or implicit and because of freely available source code. It has modules implemented for explicit overload signaling standardized by the IETF in [21] and an implicit mechanism published in [24]. The implementation is not designed for commercial deployment but rather as a prototype, which facilitates easy integration of various congestion detection methods.

The scenario used for the following measurements follow the simulation scenarios presented in earlier sections. Detection of congestion in a downstream proxy is the goal and therefore a two-proxy scenario where the second proxy plays the role of a

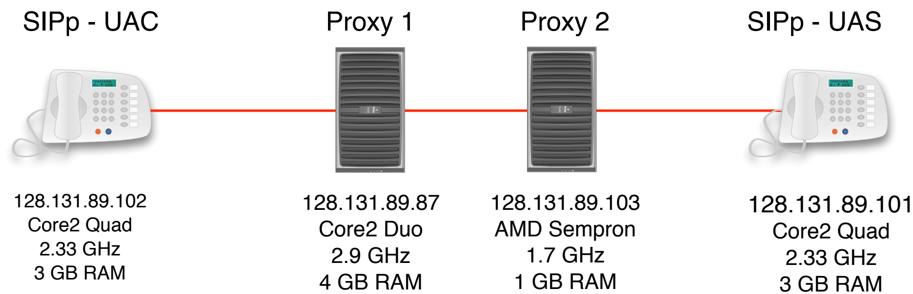


Figure 5.1.: Measurement Set-Up

bottleneck and congested entity has been chosen. Figure 5.1 shows an overview of the measurement set-up. The *UAC* message-generation is done on a Intel Core2 Quad using the industry-standard tool SIPp [31]. This tool generates new SIP messages at a configurable rate on a best-effort basis with no configurable arrival process. That issue is shortly discussed in section 5.3.1. SIPp complies to the SIP standard and implements the required state machines for INVITE and non-INVITE transactions. *Proxy 1* acts as outbound proxy for the UAC, runs on an Intel Core2 Duo and executes the congestion detection mechanisms. *Proxy 2* is the inbound proxy of the UAS and the bottleneck as its CPU can easily be outperformed by both the UAC's and proxy 1's CPUs. The SIPp *UAS* runs on an identical machine like the SIPp UAC and has therefore well enough processing capacity to outperform proxy 2. This set-up ensures that proxy 2 acts as bottleneck which has to be detected and handled accordingly to the implemented detection and control mechanism by proxy 1. All machines are connected by Gbit Ethernet interfaces and network in order to have sufficient transmission capacity and to focus on effects that appear because of congestion in the overloaded proxy. In order to remove any transient effects when starting and stopping the measurement, the first 60 seconds as well as the last 60 seconds are omitted for all statistical evaluations.

5.2. Measurement Results

This section presents foremost the measured behavior of an unprotected system and then the improvements that are gained by means of the previously developed implicit congestion detection and handling mechanisms.

5.2.1. Unprotected System

This section describes how an unprotected system reacts to overload. First, a constant load of a specific rate is generated and the behavior is documented in the following section, before the transient behavior is described for a short peak load scenario.

Constant Load

The UAC generates loads from 600 transactions per second (tps) up to 800 tps and the final Response Delay (fRpD) is measured as well as the success rate and the total number of transmissions per transaction. Figure 5.2 shows the progress of the fRpD for a load of 600 tps over a period of 300 s. This load does not overload Proxy2.

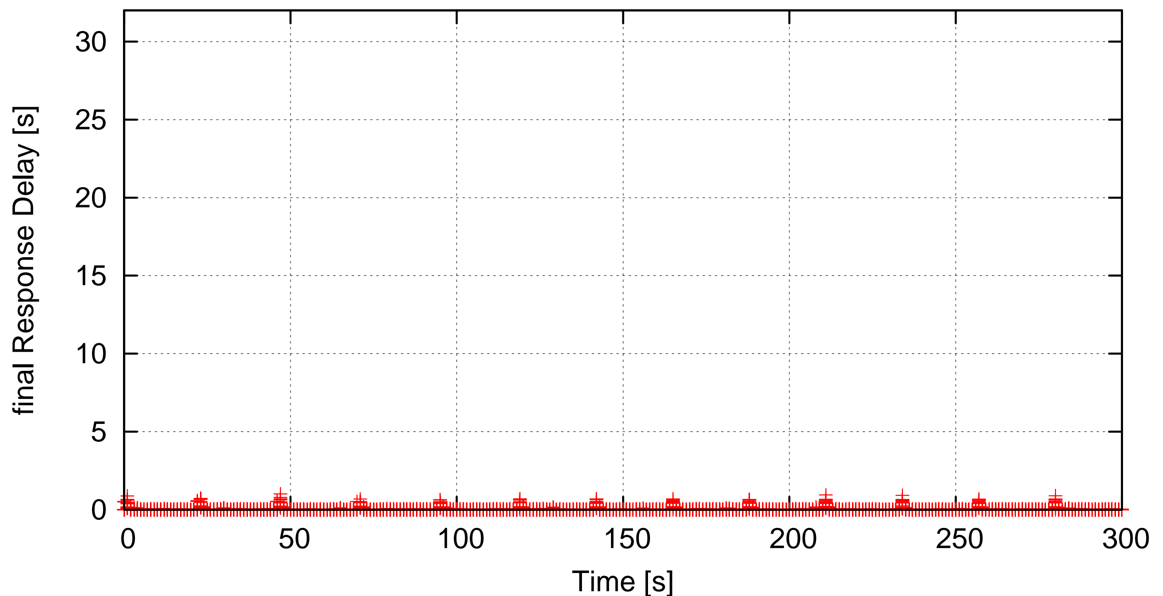


Figure 5.2.: fRpD Progress for 600 tps

The fRpD stays consistently below 500 ms (average 23.7 ms) and no request has to be re-transmitted. The periodic peaks are caused by the garbage collector of the Java implementation of the proxy server[1]. Increasing the load towards the processing capacity changes the characteristics of the fRpD as shown in figure 5.3 for a load of 744 tps. The peaks that are caused by the Java garbage collection increase because the process has to handle more data. This results in several re-transmissions that are, however, harmless for the system. Completely different is the behavior in an overload

[1] This behavior described to some detail in section 5.3.2

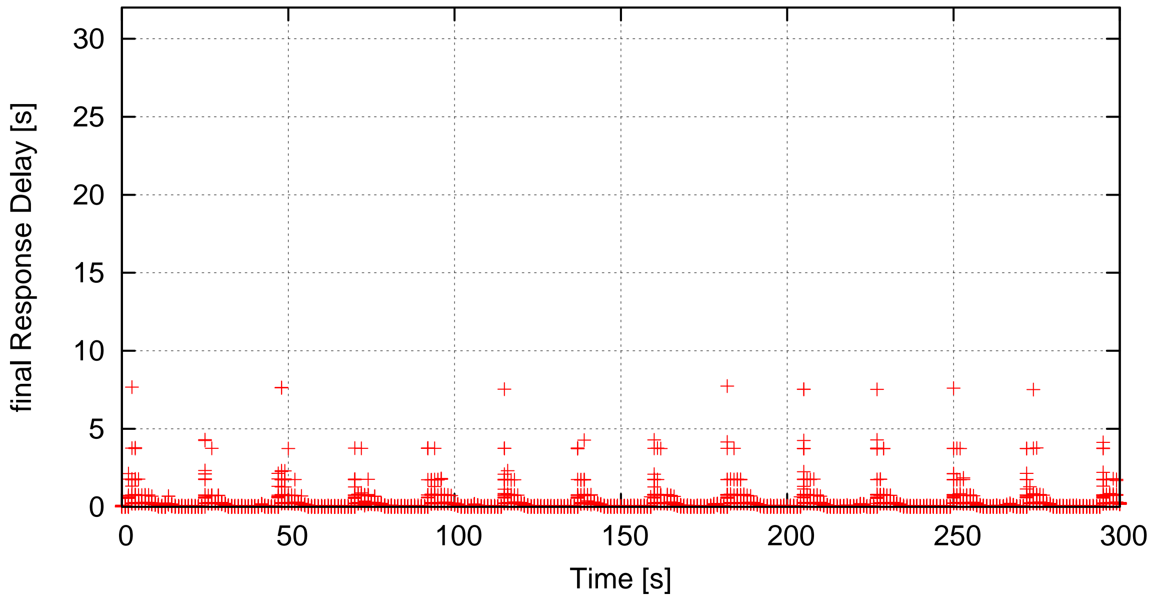


Figure 5.3.: fRpD Progress for 744 tps

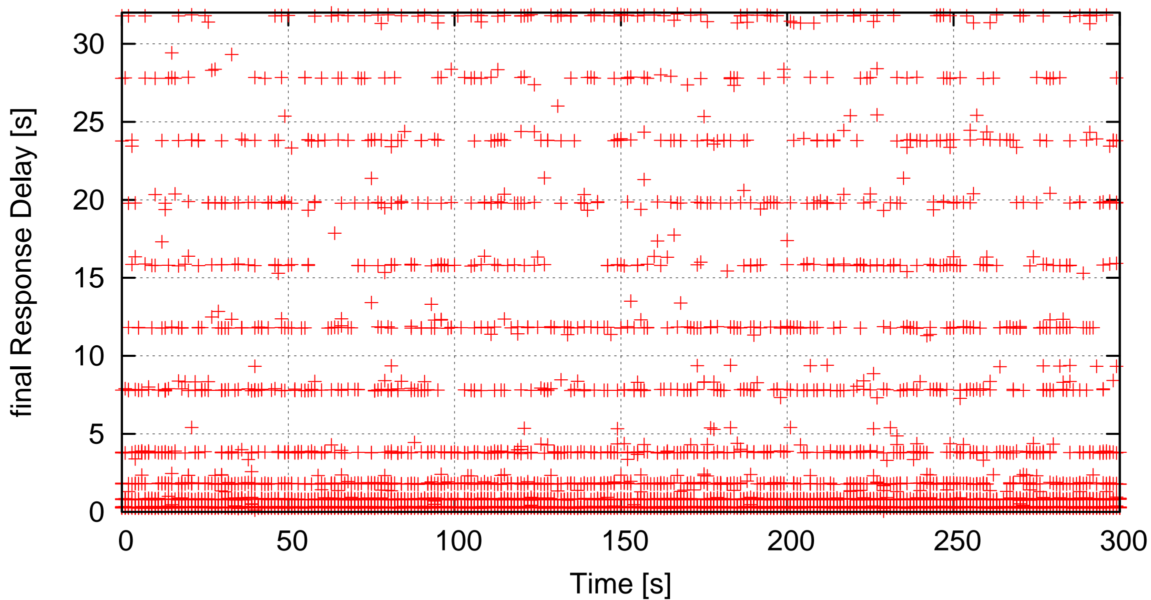


Figure 5.4.: fRpD Progress for 850 tps

situation (850 tps); many requests have to be re-transmitted and that results in a fRpD plot that is shown in figure 5.4.

Here the fRpD values are widely distributed and the re-transmissions are clearly visible through the grouping of some values into layers. The behavior that is documented here is completely consistent with the simulated results presented in section

4.1.1. Summarizing, the goodput, calculated as the number of successful requests (that is, requests that received a 200 OK response) per second against the injected load is plotted in figure 5.5. The goodput increases linearly with the injected load up

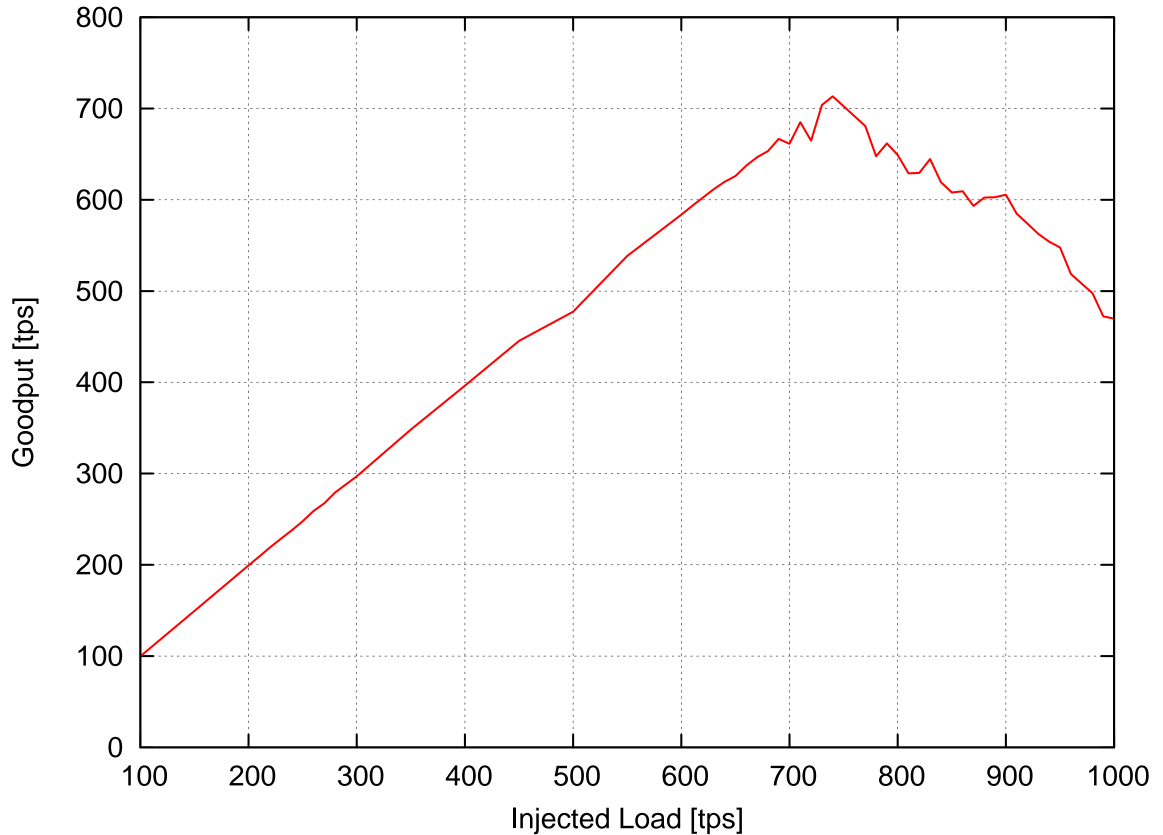


Figure 5.5.: Goodput against Injected Load: no Congestion Detection

to a number of 749 tps. Above this value, the goodput decreases because the proxy server has reached its processing limit and many requests have to be re-transmitted. These re-transmissions need additional processing time and therefore decrease the remaining CPU time for new requests as well as the overall goodput.

Peak Load

This section describes a more dynamic scenario. Starting, a constant non-overloading rate of 700 tps is generated and after 30 seconds, an overloading peak of additional 200 tps is injected. Figure 5.6 shows the fRpD of this scenario. The peak load is injected starting from $t = 30$ s until $t = 90$ s and during that time, the fRpD rises and shows the same behavior as documented for overload situations in the previous section. After the load decreases back to its previous level of 700 tps, many requests must

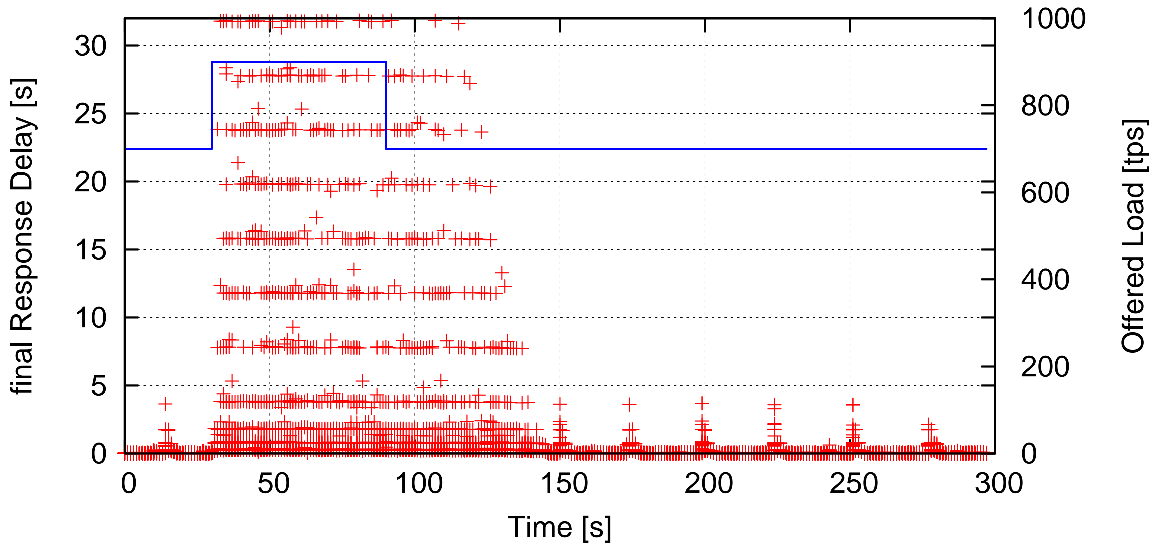


Figure 5.6.: fRpD Progress (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection

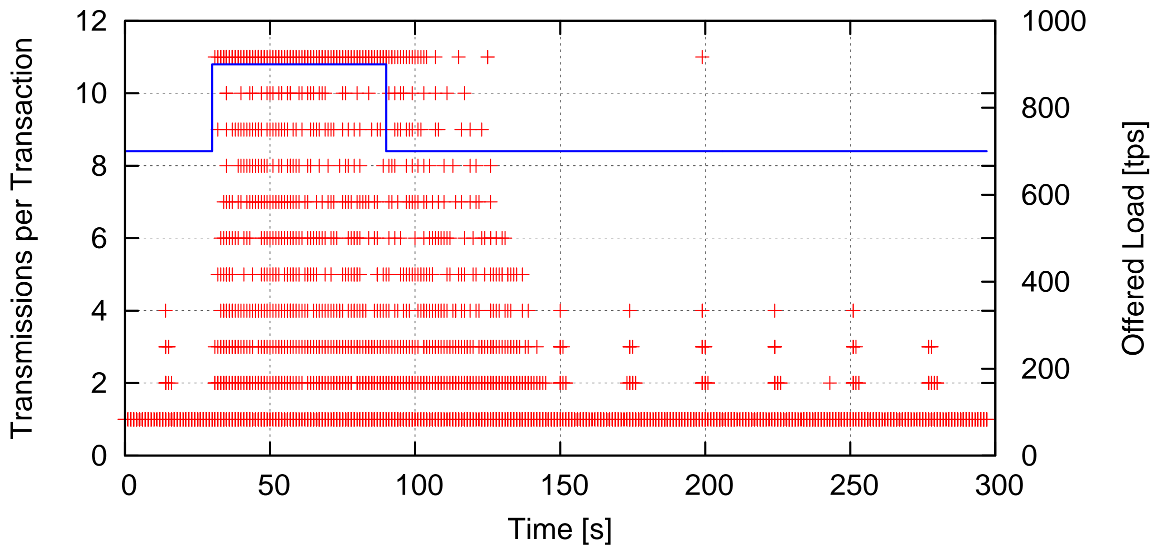


Figure 5.7.: Total Number of Request Transmissions per Transaction (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection

still be re-transmitted and the system needs additional 55 s until it has fully recovered. This is also demonstrated in figure 5.7 that shows the total number of transmissions per transaction. During the peak load and 55 s after the peak has disappeared, many requests are re-transmitted what creates additional load that is unnecessary

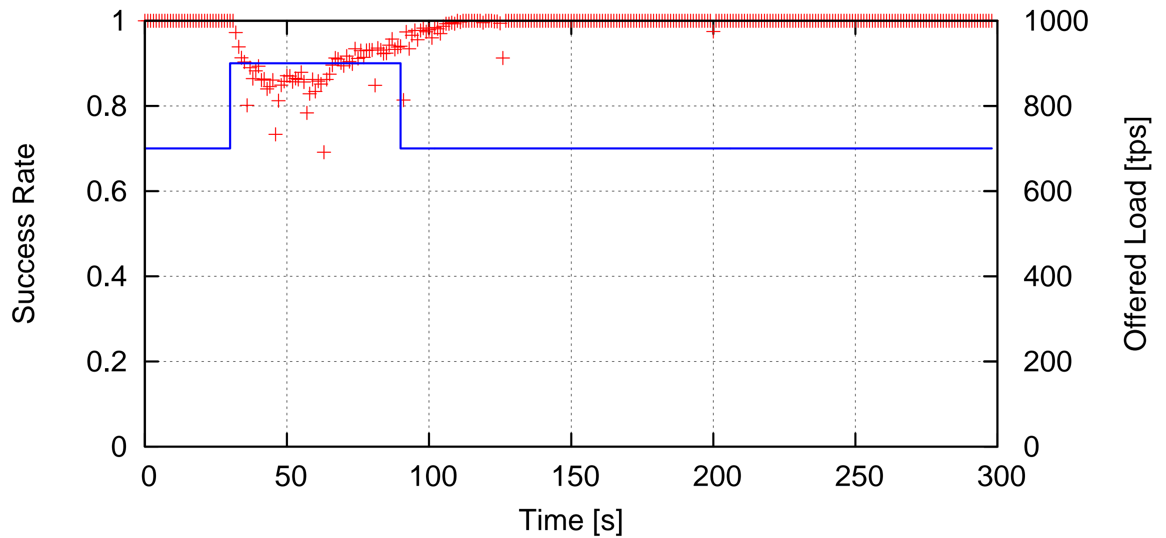


Figure 5.8.: Success Rate (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection

after all on a system that is already overloaded. The success rate, depicted in figure 5.8 shrinks to nearly 80% during the peak and also indicates that the system needs approximately 55 s until it has fully recovered.
Previous sections have shown that an unprotected system is prone to congestion and that re-transmissions cause additional load on an already overloaded system. This leads to a shrinking goodput with increasing load and unprotected systems need a long time to recover, even if a peak appears only for a short time. During that recovery time, many requests still time out and do thus not generate revenues for the system's operator. The following sections show how the congestion detection mechanisms developed throughout this thesis and tested by means of simulations perform in a physical environment.

5.2.2. Delay-based Detection

Section 4.2.2 has shown that the fRpD is dependent on the injected load; the higher the load, the longer it takes processing the requests. For evaluation of the fRpD values, rates of 10 tps up to 1000 tps in steps of 1 tps have been measured. Each measurement runs for 600 s to have sufficient transactions for statistical analysis, the first and last 60 seconds are omitted. Figure 5.9 shows the distribution of the fRpD of all messages for rates of 600, 700 and 730 tps and that similar as for the simulation the delay increases with increasing load. As the delay-based detection mechanism de-

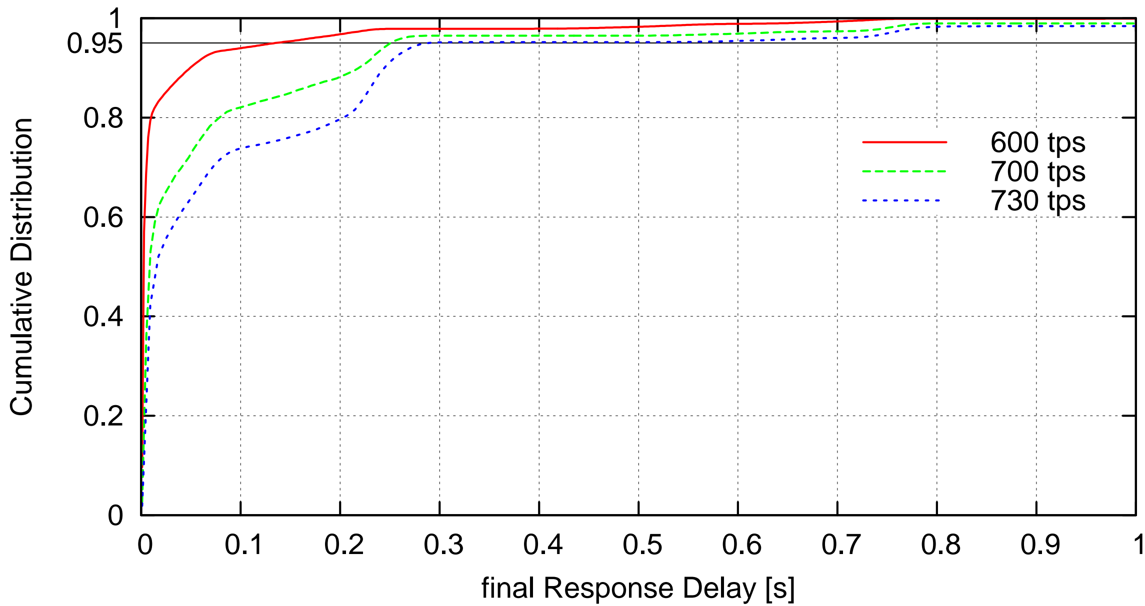


Figure 5.9.: Distribution of fRpD for the Measurement of various Load Amounts

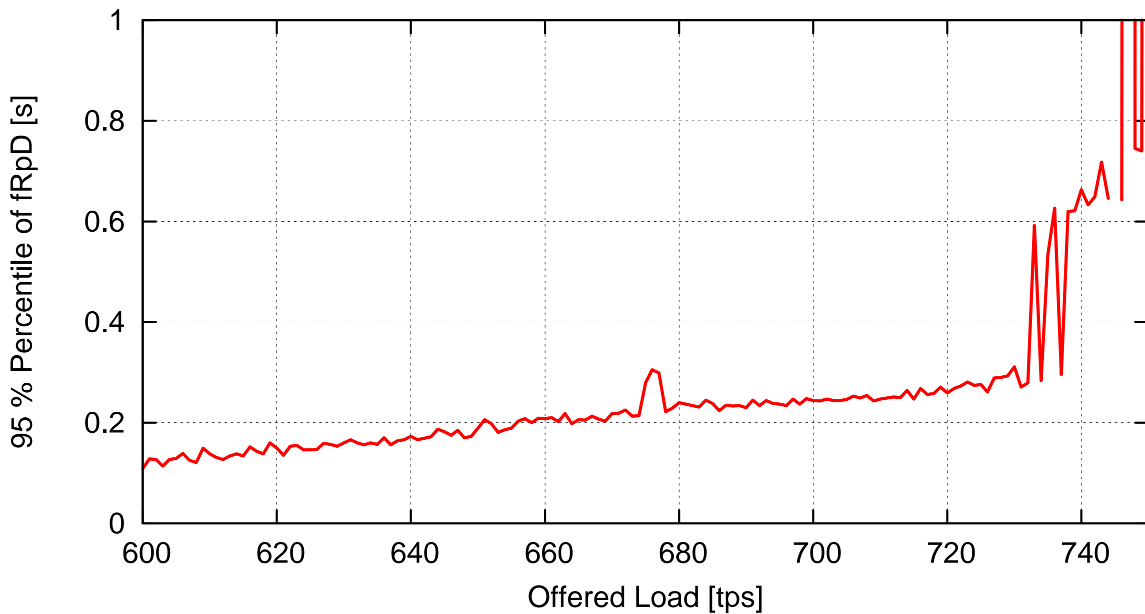


Figure 5.10.: 95% Percentile of fRpD Depending on Load

veloped in section 4.2.2 uses the 95% percentile of the fRpD for congestion detection, this value is calculated for the measurements as well. Figure 5.10 shows the 95% percentile of the fRpD for the range of 600..750 tps. The fRpD increases with increasing load similar to the simulation results, but in a different scale and behavior. Because

of the nearly deterministic arrival process of the used message generator SIPp, the processing capacity of the proxy server appears as a steep increase of the fRpD at a rate of 733 msg/s.[2] This value is slightly lower than the value of 749 tps detected in the previous section because some requests are successful after re-transmission. However, the goal of this delay-based detection mechanism is to prevent unnecessary re-transmissions and therefore the desired limit is found where the 95% percentile is just below 0.5 s. This section starts with analyzing the behavior of the delay-based mechanism using a constant load and later shows the peak load scenario as for the unprotected system.

Constant Load

The rise of the delay with the injected load is a pre-requisite that a delay-based congestion detection mechanism can be applied because the delay increases significantly when the processing capacity is reached. As next step, the delay-based congestion detection mechanism has been implemented as described in section 4.2.2. The system measures the fRpD values continuously and takes the values of the past 5 seconds for the calculation of the 95% percentile. Again, measurements for rates of 50 up to 1000 tps have been started and the goodput has been calculated.
Figure 5.11 (solid red line) shows the goodput against injected load for this measurement. The values increase with increasing load. The dashed line shows the same curve for an unprotected system as presented previously. In overload situations (loads above 733 tps), the goodput of the CD system clearly exceeds the unprotected system. This figure shows that the system with delay-based congestion detection performs better in overload situations, since it maintains the goodput close to the system's processing capacity.

Peak Load

As for the system without protection, a constant non-overloading rate of 700 tps is generated and after 30 seconds, an overloading peak of additional 200 tps is injected. The delay-based congestion detection mechanism indicates the behavior shown by means of the fRpD progress in figure 5.12. The fRpD increases for some transactions during the peak load, but decreases rapidly after the peak disappears. The total

[2]the capacity is defined as the point where the 95 % percentile of the fRpD is just below 0.5 s, in order to prevent a congestion collapse because of too many re-transmissions

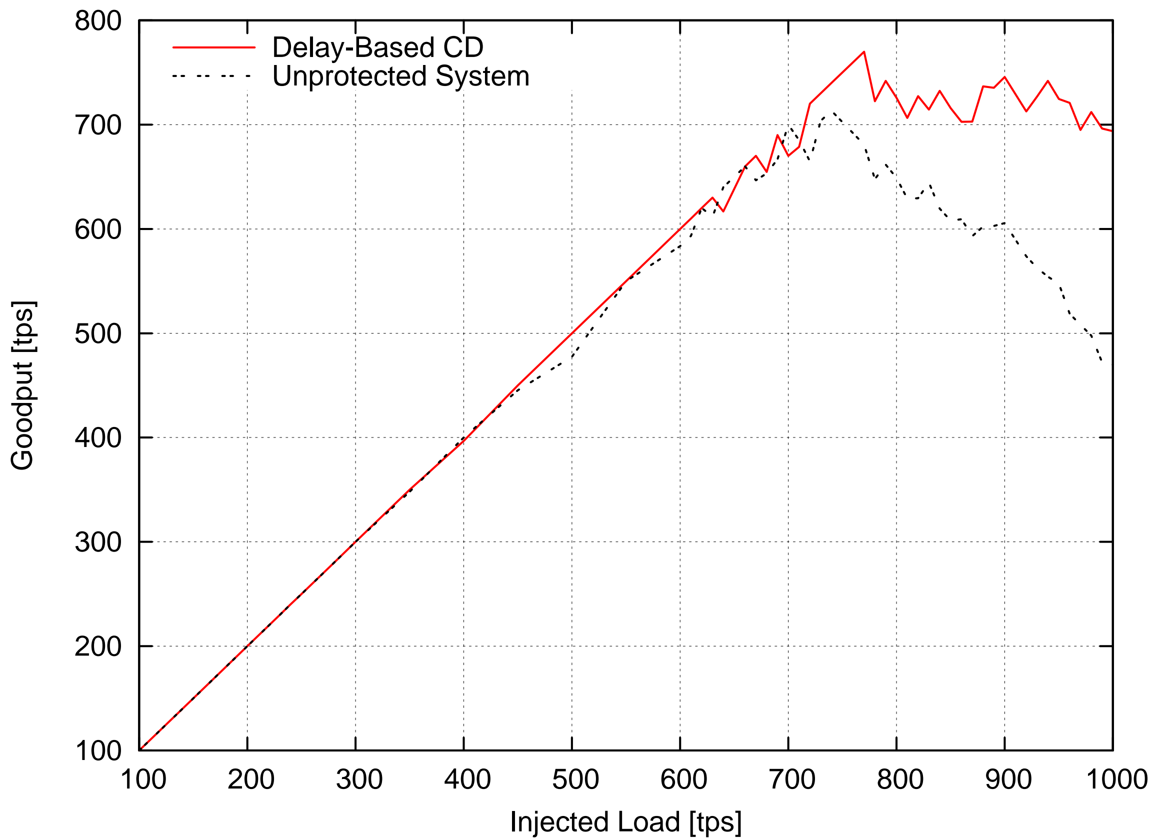


Figure 5.11.: Goodput against Injected Load: Delay-based Congestion Detection Compared to Unprotected System

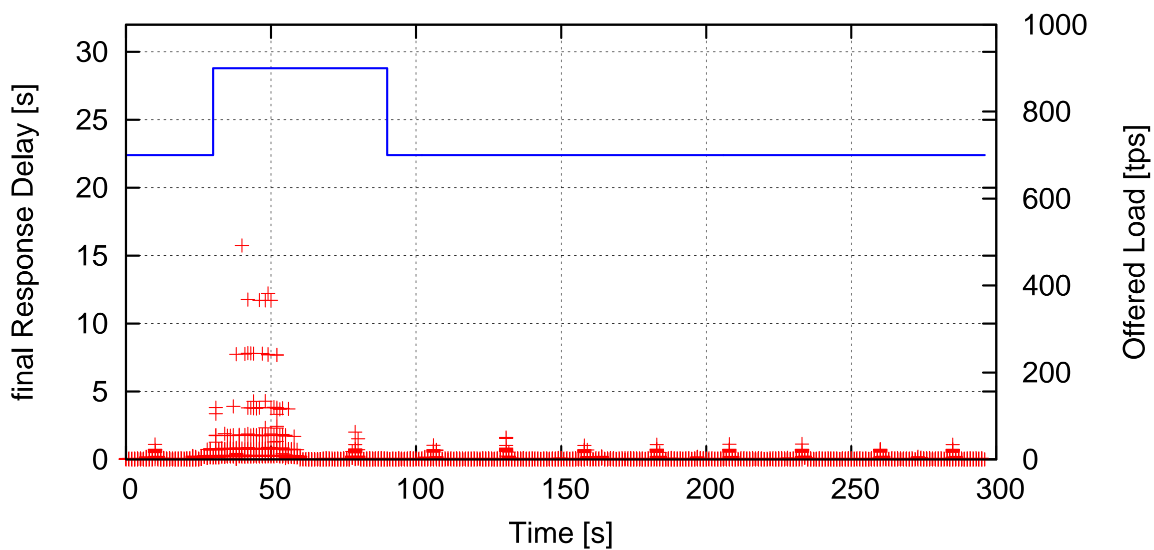


Figure 5.12.: fRpD Progress (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection

number of transmissions per transaction points out that some transactions are re-transmitted during the peak period and that afterwards the characteristics return to normal, that is, nearly all transactions succeed after a single transmission. Figure

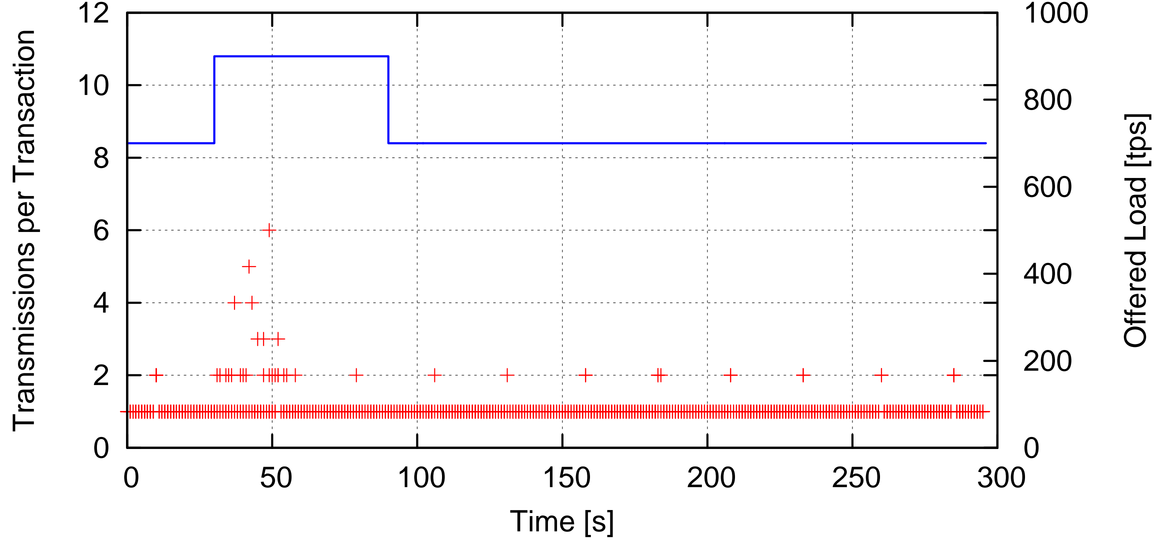


Figure 5.13.: Total Number of Request Transmissions per Transaction (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection

5.14 shows the success rate for the delay-based congestion detection that decreases only slightly and is up at 100 % immediately after the peak has disappeared.
Summarizing, the delay-based congestion detection mechanism does not completely prevent any re-transmissions during an overloading peak but does significantly reduce the time that the system requires until it has completely recovered. Additionally, the success rate during the overload situation is higher than without any protection.

5.2.3. Pending-Transaction-based Detection

This section presents the measurement results of the implementation of the pending-transaction-based congestion detection mechanism presented in section 4.2.3. That section has shown that an anomaly indicator, calculated from the number of currently pending transactions and the current load can indicate a congestion situation. The same algorithm as for the simulation has been implemented for the real-world measurements. The anomaly indicator κ is calculated as the quotient of the derivations of the load and the number of pending transactions with respect to time. The proxy calculates these values once per second and figure 5.15 shows the progress of

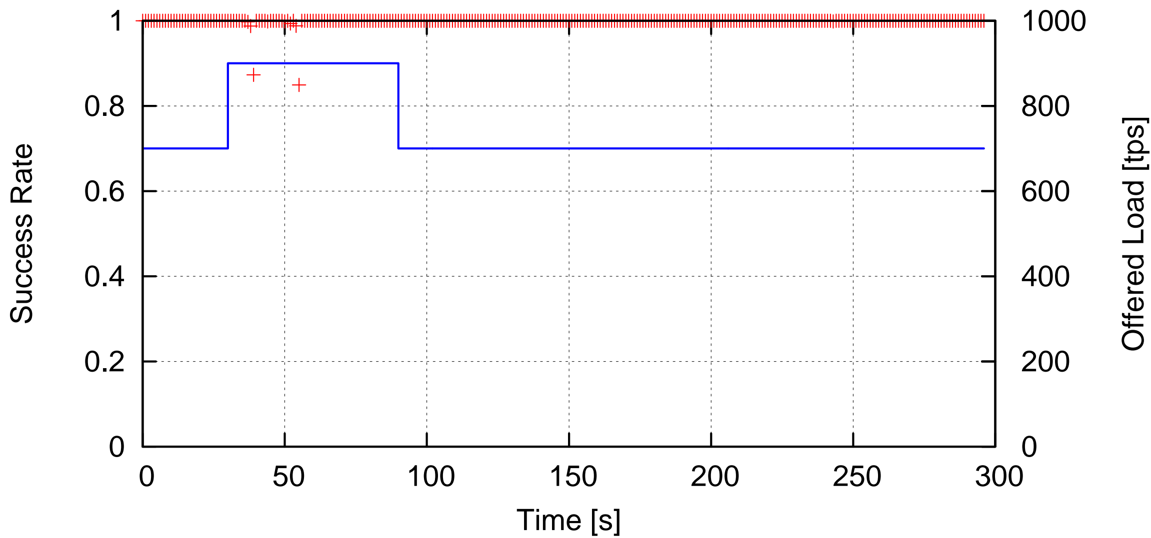


Figure 5.14.: Success Rate (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection

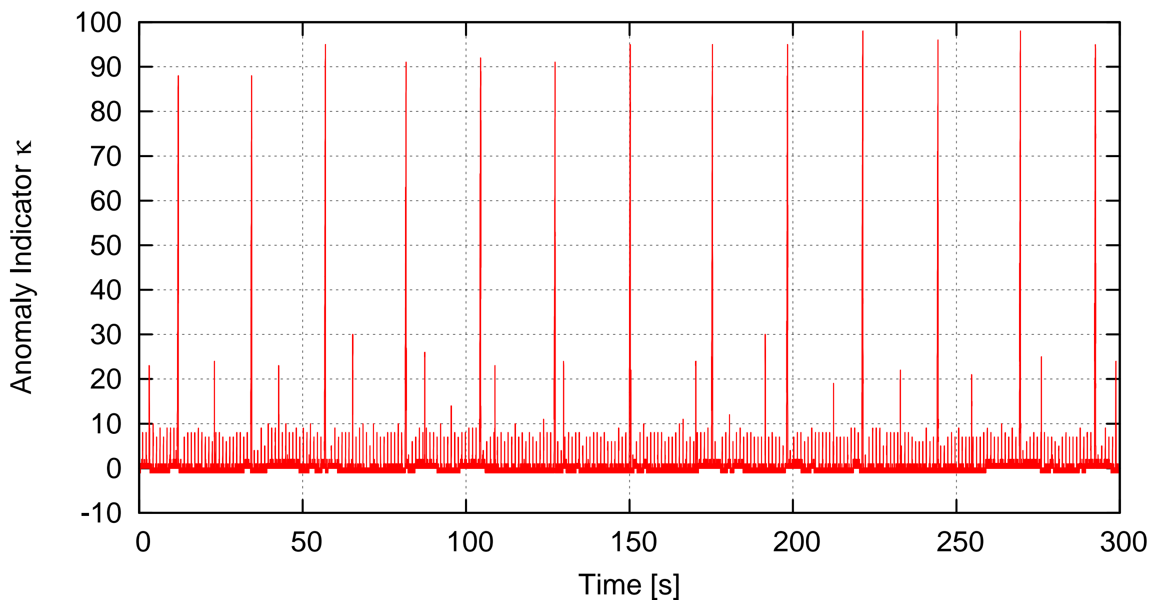


Figure 5.15.: Anomaly Indicator κ in Non-congested System State

the anomaly indicator κ for a non-congested system that is loaded by 300 tps. The value fluctuates between -1 and 10 with some peaks that occur due to the Java garbage collector[3]. Characteristic values amount to $\mu = 0.589$ and $\sigma = 6.780$, that is, within a similar order of magnitude as the simulations. The next measurement starts

[3]see section 5.3.2

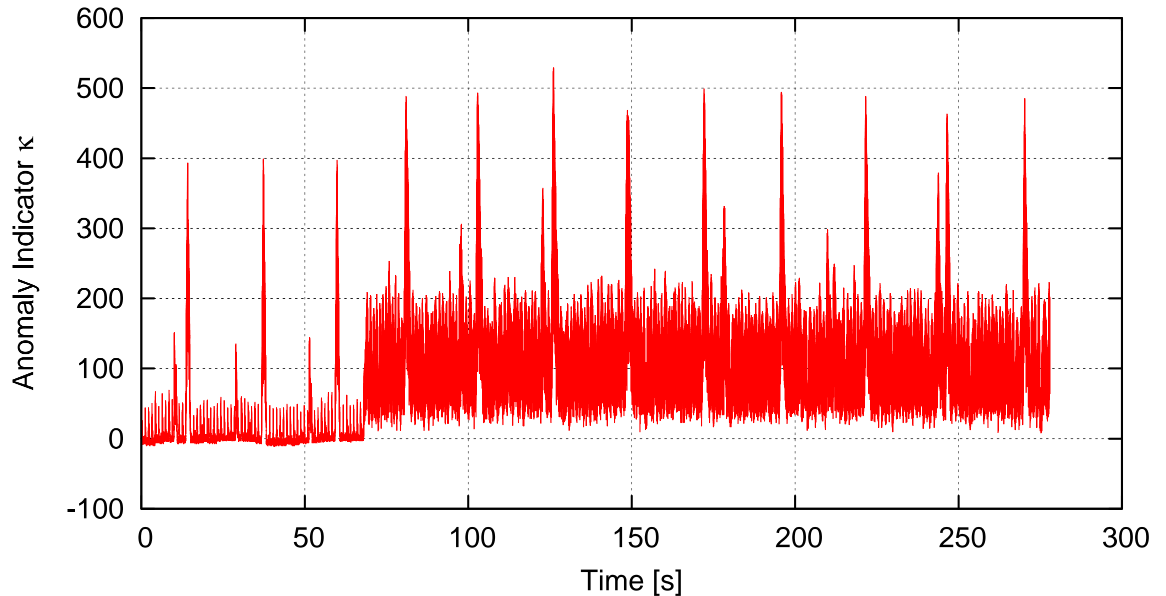


Figure 5.16.: Anomaly Indicator κ in congested System State

with a basic load of 700 tps and injects additional 200 tps after 60 seconds. The progress of the measured anomaly indicator value is shown in figure 5.16. κ increases significantly when the overload is injected to characteristic values of $\mu = 100.067$ and $\sigma = 83.097$. This is different to the simulation where the mean value stayed within the same range as for the non-congested state but the standard deviation has increased. The significant change of the mean value of the measured κ value implies that the sequential test that has been applied for the simulations (*Testing that the standard deviation of a normal distribution does not exceed a given value*) is not applicable for the measurement. Abraham Wald describes in [59] various sequential tests and the test *testing that the mean of a normal distribution with known standard deviation does not fall a given value* is appropriate and has been used and implemented for the measurements (see chapter 7 in [59]).

Constant Load

The measurement applies the same scenarios as the previous section for rates of 100 to 1000 tps and as result, the goodput against the injected load is shown in figure 5.17. The values increase with increasing load and stay at the maximum processing capacity of approximately 733 tps also in overload situations as the solid red line in figure 5.17 shows. The dashed line shows again the behavior of the system without

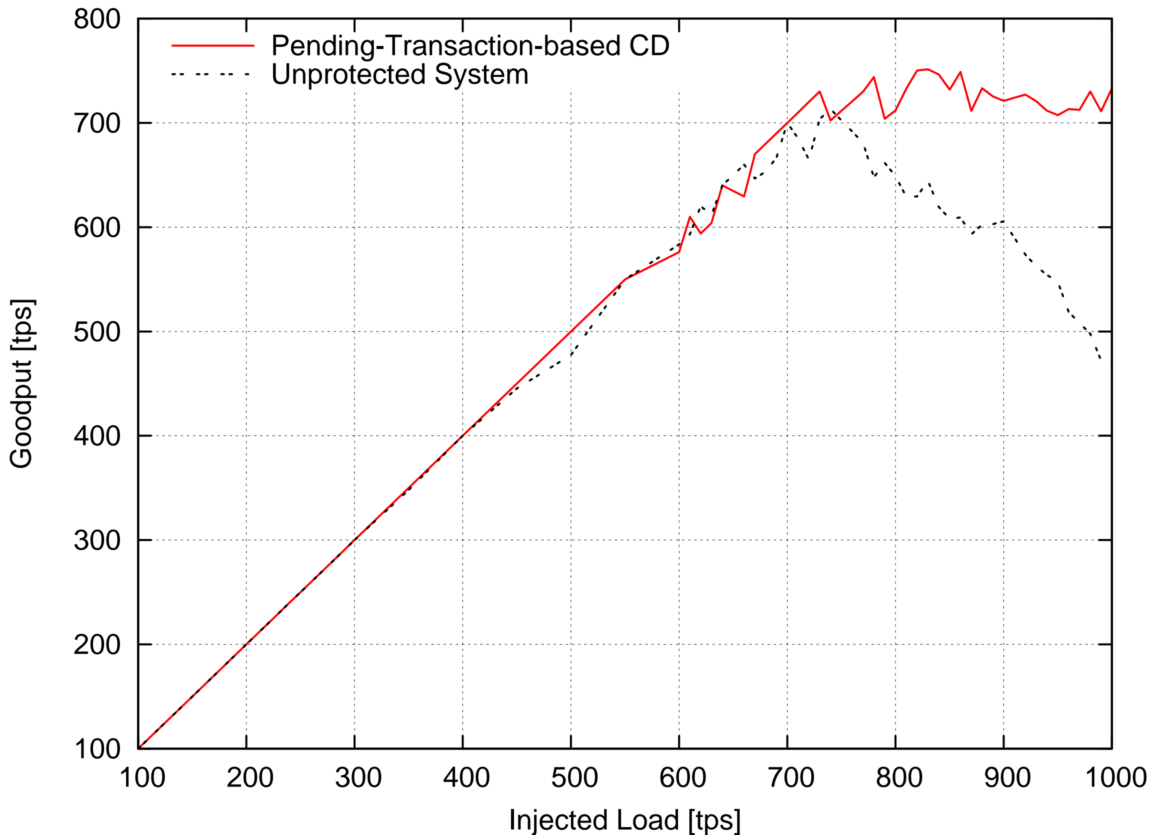


Figure 5.17.: Goodput against Injected Load: Pending-Transaction-based Congestion Detection

congestion detection. In non-overload situations the unprotected system performs slightly better because of the Java garbage collector that leads to some peaks in the anomaly detector κ as shown in figure 5.15 and therefore some requests are rejected mistakenly (considered to be false positives). Figure 5.18 shows the percentage of rejections of requests because of the Java garbage collector depending on the injected load and that this value stays below 2% what is acceptable because a limit of 10% has been defined for errors of the first kind.

Peak Load

Again, SIPp generates a background load of 700 tps and after 30 seconds of operation an overloading peak of additional 200 tps and 30 seconds duration. Figure 5.19 shows the fRpD values measured for this set-up. The values stay low almost permanently, only a small number of transactions last longer. Only a few requests need to be re-transmitted (see figure 5.20) and thus the overloaded system is not flooded by

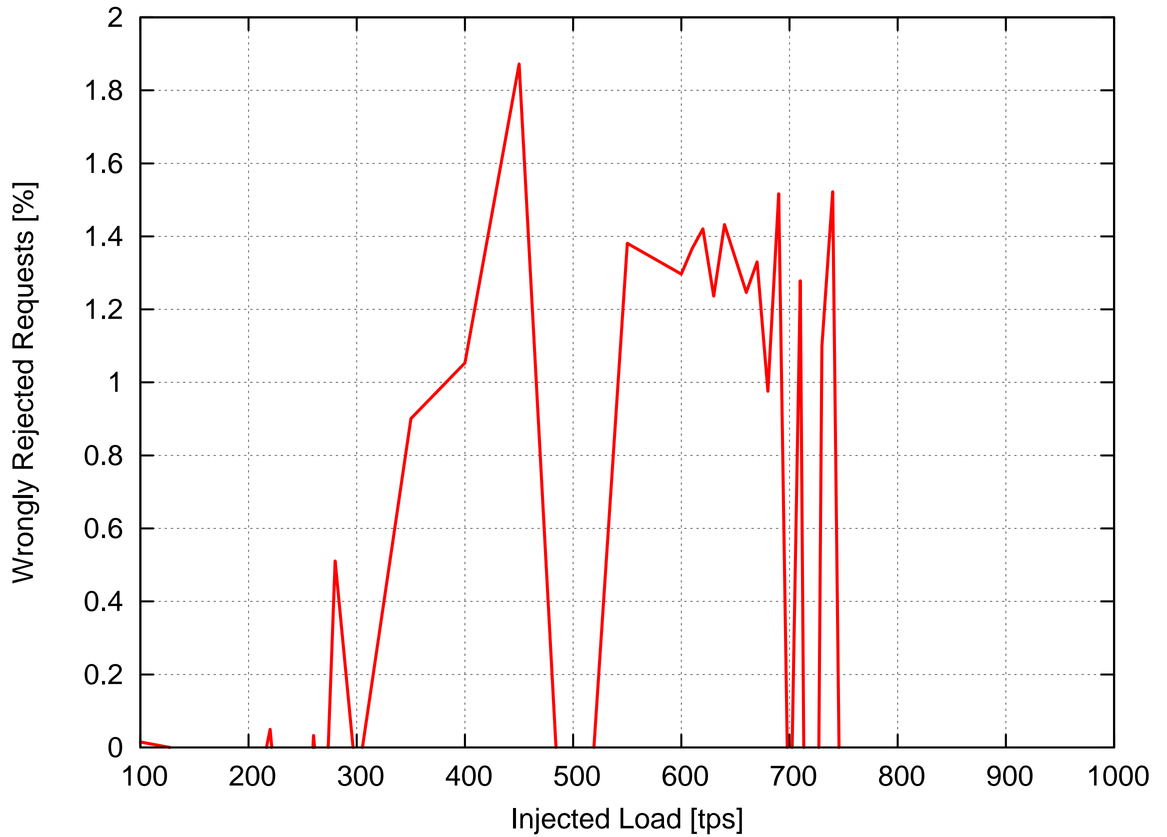


Figure 5.18.: Mistakenly Rejected Requests against Injected Load: Pending-Transaction-based Congestion Detection

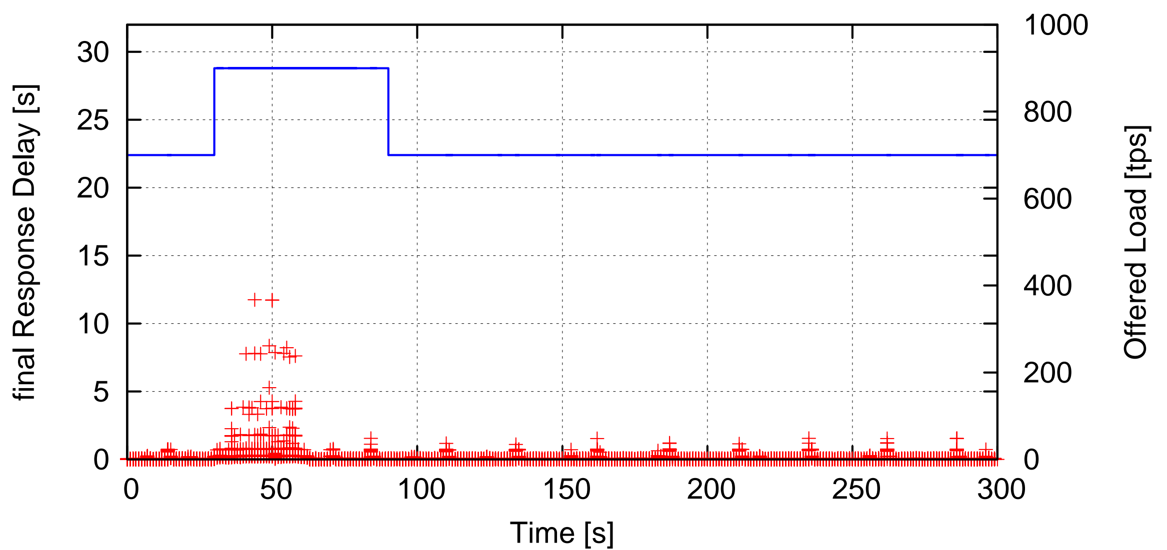


Figure 5.19.: fRpD Progress for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection

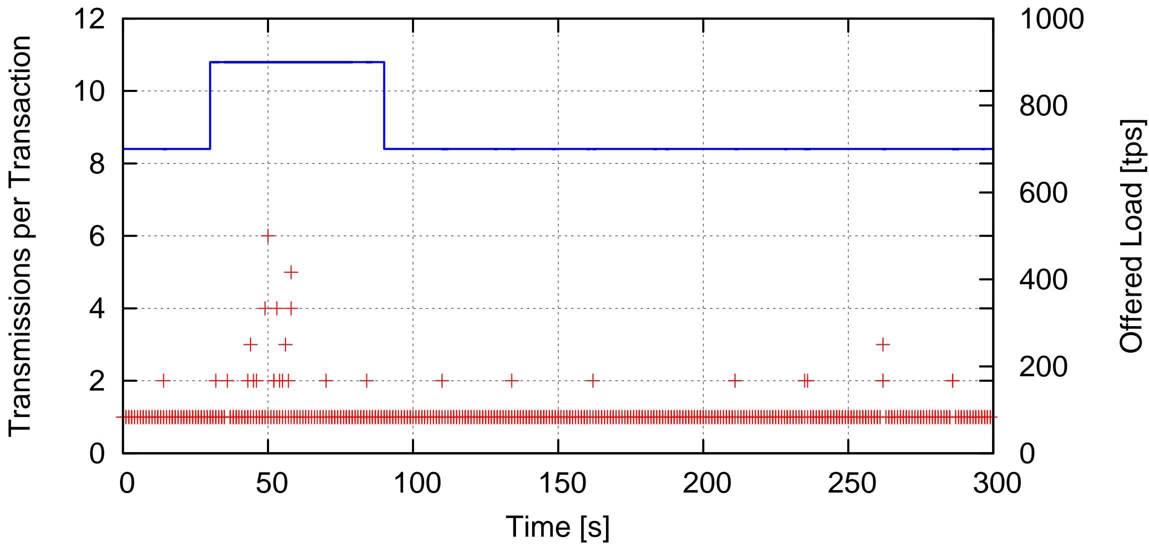


Figure 5.20.: Total Number of Request Transmissions per Transaction for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection

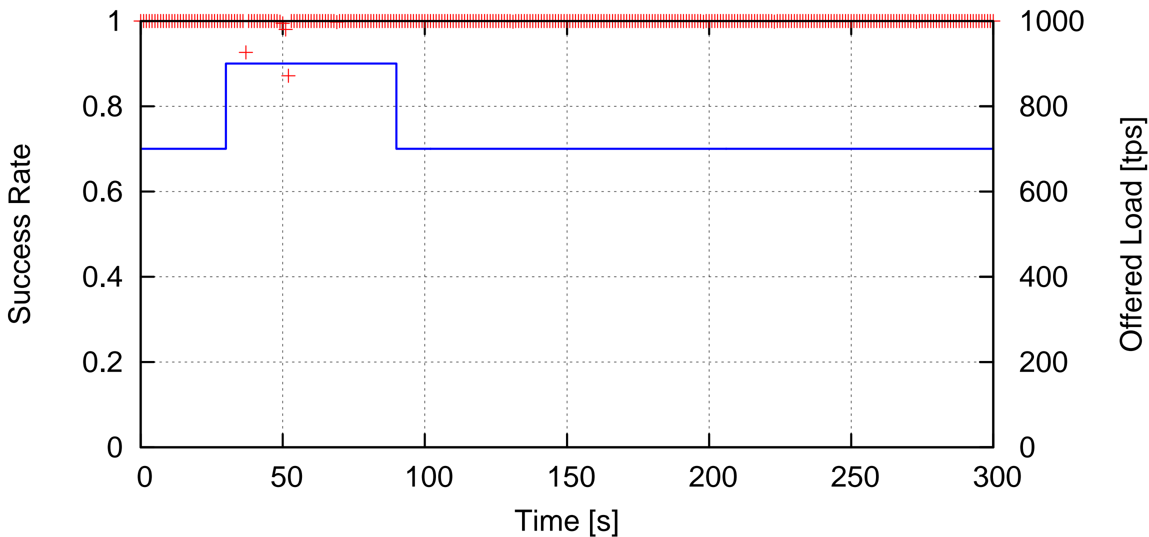


Figure 5.21.: Success Rate for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection

re-transmissions throughout the entire measurement duration. Similar is the plot of the success rate, shown in figure 5.21, only few measurements show a success rate below 100%.

Recapitulating, the pending-transaction-based congestion detection is able to recognize overload situations and to react adequately, leading to a fast recovery after the

overload has disappeared.
Summarizing, the measurements in real systems confirm the simulative results in that both approaches, the delay-based as well as the pending-transaction-based approach lead to a better overall performance of an overloaded SIP system. Figure 5.22 shows a goodput comparison of both approaches. However, it is impossible to derive a recommendation from this graph as both curves show a very similar shape.

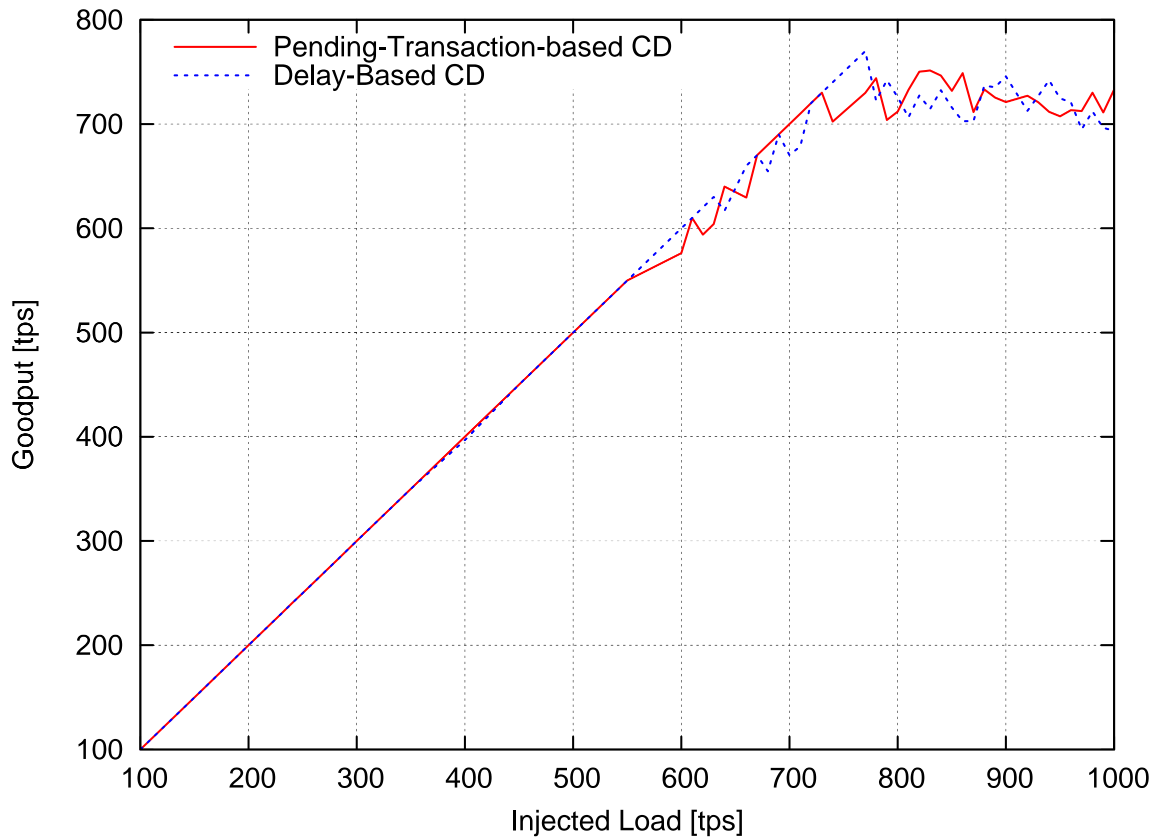


Figure 5.22.: Goodput against Injected Load: Comparison of Pending-Transaction-based and Delay-based Congestion Detection

5.3. Limiting Factors

A series of factors limit the representativeness of performance measurements for the real system. This section discusses these shortcomings. First discussed is the undefined arrival process of SIPp and second the influence of the Java garbage collector to the delay.

5.3.1. SIPp arrival Process

SIPp generates new messages at a configured rate on a best-effort basis and runs single-threaded. This means that there is no guarantee that a specific rate can be reached. Moreover, the arrival process is undefined. The solid red curve in figure 5.23 shows the distribution of measured inter-arrival times for a rate of 300 tps over a duration of 60 seconds. Because of the non-deterministic behavior of a standard

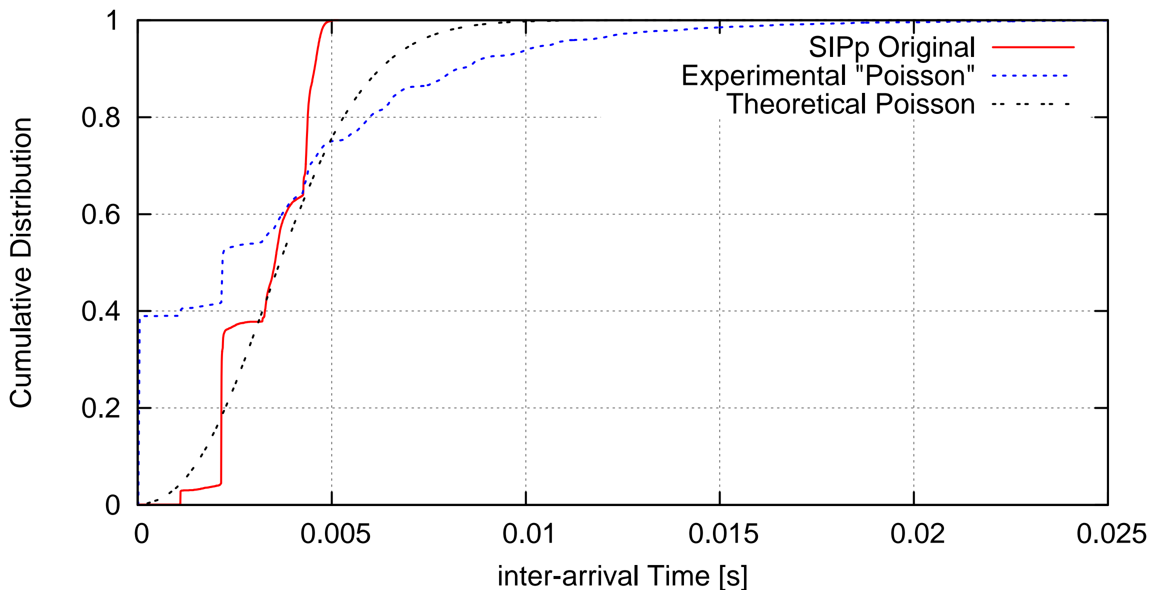


Figure 5.23.: Measurement Set-Up

Linux kernel, there is no guarantee that a specific process can use the CPU at a specified time. Therefore SIPp can not generate messages on a deterministic basis and also no specific inter-arrival distribution can be guaranteed. However, as the distribution range is small in comparison to the measured fRpD values[4], the arrival rate is considered to be nearly deterministic.

[4]inter-arrival time range: 3 ms, fRD average for 300 tps: 20 ms

Nevertheless an experiment to attempt to implement a Poisson arrival process in SIPp has been started. The blue dotted curve in figure 5.23 shows the distribution of the inter-arrival times of this implementation. The plot shows a steep increase to approximately 0.4 at low values, that is, nearly 40% of all messages are sent directly after each other and that does not correspond to a Poisson distribution (black dashed curve). For this reason and because SIPp is a standard industry tool that is used oftenly for performance checks, the author decided to use the original implementation of SIPp for the measurements.

5.3.2. Java Garbage Collector

During analysis of the measurement, especially of the fRpD values, peaks were recognizable that have been inexplicable at first sight. After deep examination, and because these peaks appear periodically, one thought has been that the Java garbage collector (GC) is responsible for that. Java allows to log the time stamps that state when the garbage collector has been activated and the amount of time it needed. Figure 5.24 shows these time stamps and the duration of the garbage collection. The

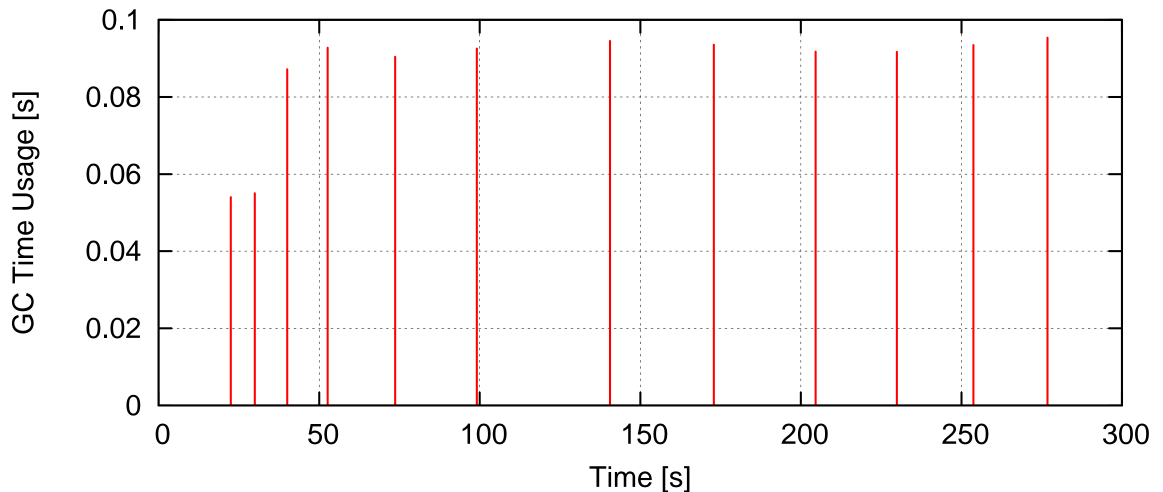


Figure 5.24.: Time stamps of Java garbage collection

GC becomes active in this configuration every 10 to 40 seconds, depending on the load of incoming SIP messages and thus on the amount of data that has to be handled by the proxy server. The process of garbage collection takes nearly 100 ms each and that is significant in comparison to average non-overload delay values of 23 ms and therefore visible in the fRpD measurements.

6. Summary and Conclusions

The current transition from circuit switched to packet switched telecommunication systems goes along with the rapidly increasing importance of SIP as signalling protocol. The most revolutionizing step is the large-scale deployment of LTE that does no longer provide circuit switched functionality at all. Thus, all currently available circuit switched services must be replaced by packet switched alternatives and the 3GPP has chosen SIP as main signaling protocol for the IMS. It is expected that this transitioning process also drives the deployment of new services, which will substantially increase the usage of SIP based systems. The increasing usage increases consequentially the amount of SIP traffic, and in high-load situations the probability of a congestion collapse of the whole system rises. The IETF and the ETSI are currently in the process of defining mechanisms to counteract overload situations. These mechanisms either define extensions for SIP that add extra data fields to SIP messages or specify an entirely new protocol in order to explicitly signal overload to upstream components. However, these approaches use resources on already overloaded hosts and that is in conflict with the goal to reduce load on that system. Implicit approaches are mechanisms that attempt to sense the load situation of a downstream component by continuously analyzing implicitly available values such as round-trip delay and are therefore better suited for this task.

In order to analyze implicit overload handling a new event-based simulation environment for SIP has been developed. Current known performance drawbacks of the protocol have been verified. One of these disadvantages is the re-transmission algorithm that may cause a huge flood of messages when a system comes into high-load and overload situations. SIP retransmits messages using static timers in order to cope with the unreliability of internet traffic. However, retransmissions are also triggered if messages are not lost, but delayed because of congestion. If the load on a system is already on a high level, all messages that have to be processed by that system are delayed. This causes a flood of redundant messages on an already excessively loaded system, eventually leading to a congestion collapse.

In order to counteract such situations, two novel implicit mechanisms have been developed. Both approaches add significantly to the stability of a system and prevent it from a potential collapse. The first approach assumes that no re-transmissions are necessary in a system that is not congested. It measures the final Response Delay of all transactions on an intermediate proxy server for a specific time and calculates their 95% percentile. If this is above 500 ms it assumes that the system is congested, because if the final Response Delay is above this values, re-transmissions are sent. The second approach is slightly more complex. It assumes that in a non-congested system the number of pending transactions directly correlates with the outgoing load and that the final Response Delay is more or less constant. The quotient of the first derivations of these values with respect to time must be more or less constant if this assumption is true. This value has been called *anomaly indicator*. The application of a statistical test called sequential analysis to continuously calculated anomaly indicator values has shown that it can reliably detect congestions.

The process of overload handling has been divided into two parts, *congestion detection* and *congestion handling*. Congestion detection is the continuous analysis of performance values in order to classify a system into non-congested or congested state. Performance values that have been used in this thesis are the QoSg defined metric final Response Delay and the number of pending transactions in combination with the number of sent requests. Congestion handling is the application of counteractions onto a system that has been classified as congested. These counteractions are divided into lossless-modes that increase the time a system has to handle the offered load without rejecting any requests, and lossy-modes that reject a part of the incoming requests. As lossless mode this thesis demonstrates that the adaptive manipulation of SIP re-transmission timers can act as tool for prevention of a congestion collapse caused by short-term overload. But, however, if the overload is more extensive, this approach can be insufficient. Rate-limiting is presented as lossy-mode which reduces the load on a congested downstream entity by rejecting a certain part of the incoming requests. In order to prove that the developed mechanisms do not only work in a simulated environment, they have been implemented in a SIP proxy and their potential has been analyzed. The measurements have verified that implicit congestion detection is possible for SIP. Application scenarios have been presented that represent examples for realistic overload-generating situations to show that the simulation scenarios map to real-world examples.

Implicit mechanisms as demonstrated in this thesis introduce only little overhead

and are able to react fast. However, it may be difficult to control situations when overload has huge variations caused by many entities throughout the network. Additionally it is important to mention that delay variations caused by various SIP message destinations may impede correct congestion detection using implicit mechanisms. Implicit overload controls represent a heuristic which by definition can not provide deterministic results. Finally, it is up to the operator of a network to choose its approach.

Important future research in this area, is the study of dynamic adaptation of the correction factors in order to enable the developed mechanisms to react upon more dynamic scenarios. As source for that extension, already existing long-term measurements of a mobile operator need to be used. The application of measured values enables to create more realistic, real-world oriented simulation scenarios. Therefore, the developed implicit congestion detection mechanisms can be adapted to fit also in more dynamic conditions. Furthermore, already existing implicit congestion detection mechanisms, like for example the various types developed for TCP, could be adapted in order to enhance the SIP performance.

Abbreviations

3GPP	Third Generation Partnership Project
AoR	Address of Record
CD	Congestion Detection
CPU	Central Processing Unit
CRLF	Carriage Return Line Feed
DNS	Domain Name System
DSL	Digital Subscriber Line
ETSI	European Telecommunications Standards Institute
fRpD	Final Response Delay
FCFS	First Come First Serve
GC	Garbage Collector
GOCAP	Generic Overload Control Activation Protocol
HSPA	High Speed Packet Access
HTTP	Hyper-Text Transfer Protocol
IETF	Internet Engineering Task Force
IID	Independent and Identically Distributed
IMS	IP Multimedia Subsystem
IP	Internet Protocol
IPPM	IP Performance Metrics
LTE	Long Term Evolution
MIME	Multipurpose Internet Mail Extensions
NGN	Next Generation Network
NOCA	NGN Overload Control Architecture
POTS	Plain Old Telephony Service
PMOL	Performance Metrics for Other Layers
PT	Pending Transactions
QoSg	Quality of Signaling
RFC	Request for Comments
RTT	Round Trip Time
RT	Request Transmits
SER	SIP Express Router
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SMS	Short Message Service

SOC	SIP Overload Control
SOE	SIP Offload Engine
SR	Success Rate
ST	Server Transaction
TCP	Transmission Control Protocol
TM	Transaction Manager
TU	Transaction User
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UTF-8	Unicode Transformation Format 8
VoIP	Voice over IP
WG	Working Group

A. Tables

Table A.1.: Observed and Expected absolute Differences of Simulated and Calculated Cumulative Distribution Functions of the Anomaly Indicator κ of a Congested System

i	α_i	$F^o(\alpha_i)$	$F^e(\alpha_i)$	$\lvert F^e(\alpha_i) - F^o(\alpha_i)\rvert$	$\lvert F^e(\alpha_i) - F^o(\alpha_{i-1})\rvert$
1	-41.0	0.041	0.029	0.012	0.029
2	-35.5	0.061	0.052	0.009	0.011
3	-31.0	0.082	0.080	0.002	0.019
4	-24.5	0.102	0.139	0.037	0.057
5	-21.5	0.122	0.175	0.053	0.073
6	-21.0	0.143	0.181	0.038	0.059
7	-18.5	0.163	0.215	0.052	0.072
8	-18.0	0.184	0.222	0.038	0.059
9	-17.0	0.204	0.237	0.033	0.053
10	-16.5	0.245	0.245	0.000	0.041
11	-14.5	0.265	0.277	0.012	0.032
12	-13.5	0.286	0.293	0.007	0.028
13	-13.0	0.306	0.302	0.004	0.016
14	-12.0	0.327	0.319	0.008	0.013
15	-11.0	0.347	0.337	0.010	0.010
16	-10.0	0.388	0.355	0.033	0.008
17	-9.5	0.408	0.364	0.044	0.024
18	-9.0	0.449	0.373	0.076	0.035
19	-8.5	0.469	0.383	0.086	0.066
Continued on next page					

i	α_i	$F^o(\alpha_i)$	$F^e(\alpha_i)$	$\lvert F^e(\alpha_i) - F^o(\alpha_i)\rvert$	$\lvert F^e(\alpha_i) - F^o(\alpha_{i-1})\rvert$
20	-7.0	0.490	0.411	0.079	0.058
21	-6.5	0.510	0.420	0.090	0.070
22	-6.0	0.531	0.430	0.101	0.080
23	-4.5	0.571	0.459	0.112	0.072
24	-3.0	0.592	0.488	0.104	0.083
25	2.0	0.612	0.586	0.026	0.006
26	3.5	0.633	0.614	0.019	0.002
27	5.0	0.653	0.642	0.011	0.009
28	6.0	0.694	0.660	0.034	0.007
29	8.0	0.714	0.695	0.019	0.001
30	9.5	0.735	0.720	0.015	0.006
31	10.0	0.755	0.729	0.026	0.006
32	11.0	0.776	0.745	0.031	0.010
33	12.0	0.816	0.760	0.056	0.016
34	13.5	0.837	0.782	0.055	0.034
35	14.0	0.857	0.789	0.068	0.048
36	14.5	0.878	0.796	0.082	0.061
37	16.0	0.898	0.817	0.081	0.061
38	23.5	0.918	0.898	0.020	0.000
39	34.0	0.939	0.963	0.024	0.045
40	36.0	0.959	0.970	0.011	0.031
41	42.0	0.980	0.985	0.005	0.026
42	61.0	1.000	0.999	0.001	0.019

Table A.2.: Critical Values for the Kolmogorov-Smirnov-Test [10]

	α				
n	0.1	0.05	0.025	0.01	0.005
1	0.9000	0.9500	0.9750	0.9900	0.9950
2	0.6838	0.7764	0.8419	0.9000	0.9293
3	0.5648	0.6360	0.7076	0.7846	0.8290
4	0.4927	0.5652	0.6239	0.6889	0.7342
5	0.4470	0.5094	0.5633	0.6272	0.6685
6	0.4104	0.4680	0.5193	0.5774	0.6166
7	0.3815	0.4361	0.4834	0.5384	0.5758
8	0.3583	0.4096	0.4543	0.5065	0.5418
9	0.3391	0.3875	0.4300	0.4796	0.5133
10	0.3226	0.3687	0.4092	0.4566	0.4889
11	0.3083	0.3524	0.3912	0.4367	0.4677
12	0.2958	0.3382	0.3754	0.4192	0.4490
13	0.2847	0.3255	0.3614	0.4036	0.4325
14	0.2748	0.3142	0.3489	0.3897	0.4176
15	0.2659	0.3040	0.3376	0.3771	0.4042

Continued on next page

	α				
n	0.1	0.05	0.025	0.01	0.005
16	0.2578	0.2947	0.3273	0.3657	0.3920
17	0.2504	0.2863	0.3180	0.3553	0.3809
18	0.2436	0.2785	0.3094	0.3457	0.3706
19	0.2373	0.2714	0.3014	0.3369	0.3612
20	0.2316	0.2647	0.2941	0.3287	0.3524
21	0.2262	0.2586	0.2872	0.3210	0.3443
22	0.2212	0.2528	0.2809	0.3139	0.3367
23	0.2165	0.2475	0.2749	0.3073	0.3295
24	0.2120	0.2424	0.2693	0.3010	0.3229
25	0.2079	0.2377	0.2640	0.2952	0.3166
26	0.2040	0.2332	0.2591	0.2896	0.3106
27	0.2003	0.2290	0.2544	0.2844	0.3050
28	0.1968	0.2250	0.2499	0.2794	0.2997
29	0.1935	0.2212	0.2457	0.2747	0.2947
30	0.1903	0.2176	0.2417	0.2702	0.2899
31	0.1873	0.2141	0.2379	0.2660	0.2853
32	0.1844	0.2108	0.2342	0.2619	0.2809
33	0.1817	0.2077	0.2308	0.2580	0.2768
34	0.1791	0.2047	0.2274	0.2543	0.2728
35	0.1766	0.2018	0.2242	0.2507	0.2690
36	0.1742	0.1991	0.2212	0.2473	0.2653
37	0.1719	0.1965	0.2183	0.2440	0.2618
38	0.1697	0.1939	0.2154	0.2409	0.2584
39	0.1675	0.1915	0.2127	0.2379	0.2552
40	0.1655	0.1891	0.2101	0.2349	0.252
Approx. for $n > 40$	$\frac{1.07}{\sqrt{n}}$	$\frac{1.22}{\sqrt{n}}$	$\frac{1.36}{\sqrt{n}}$	$\frac{1.52}{\sqrt{n}}$	$\frac{1.63}{\sqrt{n}}$

List of Figures

2.1. Forking Proxy 14
2.2. Redirection Server 15
2.3. Message Flow for SIP Registration 15
2.4. Session Scenario 18
2.5. Message flow for SIP MESSAGE 20
2.6. Non-INVITE Client Transaction (cf. [53]) 23
2.7. Non-INVITE Server Transaction (cf. [53]) 23
2.8. INVITE Client Transaction (cf. [53]) 24
2.9. INVITE Server Transaction (cf. [53]) 25
2.10. Final Response Delay Definition 26

3.1. Event-driven Flow Control (cf. [39]) 32
3.2. IBKsim Kernel 38
3.3. Logging Procedure 39
3.4. User Agent Client Transaction Manager 44
3.5. User Agent Server Transaction Manager 45
3.6. First-Come-First-Served Model 46
3.7. Request/Response distinguishing Model 47
3.8. Advanced Proxy Simulation Model 48
3.9. Transaction Relationships (cf. [53]) 50
3.10. Simulation environment: sending new request 51
3.11. Message Service 52
3.12. Register 52
3.13. Presence Service 53

4.1. Simulation Scenario 58
4.2. Arrival Rate for new Requests (red) and Re-transmissions (blue) . . . 59
4.3. Success Rate (red crosses) and Offered Load (blue line) 60
4.4. Transmissions per Transaction (red dots) and Offered Load (blue line) 60

4.5. Final Response Delay (red dots) and Offered Load (blue line) 61
4.6. Processor Utilization (top red line) and Offered Load (bottom blue line) 62
4.7. Success Rate Distribution 62
4.8. Final Response Delay - Idle System 63
4.9. Final Response Delay Distribution - Idle System 64
4.10. Request Transmissions - Idle System 65
4.11. Final Response Delay - High Load - No Congestion 65
4.12. Final Response Delay Distribution - High Load - No Congestion . . . 66
4.13. final Response Delay - Congestion 67
4.14. Transmissions per Transaction - Congestion 67
4.15. Simulation Scenario 69
4.16. Final Response Delay Distribution for multiple Load Settings 70
4.17. 95% percentile of fRpD depending on Load 71
4.18. Delay-based Congestion Detection Algorithm 71
4.19. Pending-Transaction-based Congestion Detection Algorithm 72
4.20. Number of Pending Transactions depending on the injected Load . . 73
4.21. Anomaly Indicator 75
4.22. Simulation Scenario for Data Analysis 76
4.23. Anomaly Indicator κ in Non-Congested System State 76
4.24. κ Distribution in Non-Congested System State 77
4.25. Number of Pending Transactions during Overload 79
4.26. Anomaly Indicator κ in Congested System State during $t = 150..200$. 79
4.27. κ Distribution in Congested System State 80
4.28. Results of the Sequential Analysis Test for Non-Congested System . . 83
4.29. Results of the Sequential Analysis Test for Congested System 84
4.30. Number of Alarms (% of Total Number of Calculations) against injected Load 85
4.31. Simulation Scenario 86
4.32. Success Rate (red crosses) and Offered Load (blue line) 87
4.33. Total Number of Transmissions per Transaction (red dots) and Offered Load (blue line) 88
4.34. Final Response Delay (red dots) and Offered Load (blue line) 89
4.35. Processor Utilization (red crosses) and Offered Load (blue line) . . . 89
4.36. Success Rate Distribution 90
4.37. Success Rate Distribution for multiple T1 settings 90

4.38. Success Rate when changing T1 after 50 s (red crosses) and T1 Setting (green line) . . . 91
4.39. Success Rate when changing T1 after 100 s (red crosses) and T1 Setting (green line) . . . 91
4.40. Goodput with and without Congestion Detection . . . 94
4.41. Final Response Delay Distribution with and without Congestion Detection . . . 95
4.42. Goodput against Injected Load . . . 96
4.43. Simulation Scenario . . . 97
4.44. Success Rate with SOE Improvement (red crosses) and Offered Load (blue line) . . . 98
4.45. Final Response Delay with SOE Improvement (red dots) and Offered Load (blue line) . . . 98
4.46. Total Number of Transmissions per Transaction with SOE Improvement (red dots) and Offered Load (blue line) . . . 99
4.47. Success Rate with SOE and prolonged peak (red crosses) and Offered Load (blue line) . . . 99
4.48. Simulation Scenario for Presence Service . . . 101
4.49. Number of Active Subscriptions for a System without Protection . . . 103
4.50. Proxy 2 Processor Utilization for a System without Protection . . . 103
4.51. Success Rate for Background Traffic for a System without Protection 104
4.52. Total Number of Transmissions per Transaction for a System without Protection . . . 105
4.53. Number of Active Subscriptions for Delay-based Detection . . . 106
4.54. Proxy 2 Processor Utilization for Delay-based Detection . . . 106
4.55. Success Rate for Background Traffic for Delay-based Detection . . . 107
4.56. Total Number of Transmissions per Transaction for Delay-based Detection . . . 108
4.57. Number of active Subscriptions for Pending-Transaction-based Detection 109
4.58. Proxy 2 Processor Utilization for Pending-Transaction-based Detection 109
4.59. Success Rate for Background Traffic for Pending-Transaction-based Detection . . . 110
4.60. Total Number of Transmissions per Transaction for Pending-Transaction-based Detection . . . 111
4.61. Simulation Scenario for Registration Service . . . 112

4.62. Number of Active Registrations for a System with Static Expiration Time and without Protection . . . 113
4.63. Number of Active Registrations for a System without Protection and Randomized Expiration Time . . . 114
4.64. Number of Active Registrations for Delay-based Detection . . . 114
4.65. Number of Active Registrations for Pending-Transaction-based Detection 115

5.1. Measurement Set-Up . . . 118
5.2. fRpD Progress for 600 tps . . . 119
5.3. fRpD Progress for 744 tps . . . 120
5.4. fRpD Progress for 850 tps . . . 120
5.5. Goodput against Injected Load: no Congestion Detection . . . 121
5.6. fRpD Progress (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection . . . 122
5.7. Total Number of Request Transmissions per Transaction (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection . . . 122
5.8. Success Rate (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): No Congestion Detection . . . 123
5.9. Distribution of fRpD for the Measurement of various Load Amounts . 124
5.10. 95% Percentile of fRpD Depending on Load . . . 124
5.11. Goodput against Injected Load: Delay-based Congestion Detection Compared to Unprotected System . . . 126
5.12. fRpD Progress (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection . . . 126
5.13. Total Number of Request Transmissions per Transaction (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection . . . 127
5.14. Success Rate (red crosses) for 700 tps Background Load and 900 tps Peak Load (blue line): Delay-based Congestion Detection . . . 128
5.15. Anomaly Indicator κ in Non-congested System State . . . 128
5.16. Anomaly Indicator κ in congested System State . . . 129
5.17. Goodput against Injected Load: Pending-Transaction-based Congestion Detection . . . 130
5.18. Mistakenly Rejected Requests against Injected Load: Pending-Transaction-based Congestion Detection . . . 131

5.19. fRpD Progress for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection 131
5.20. Total Number of Request Transmissions per Transaction for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection 132
5.21. Success Rate for 700 tps Background Load and 900 tps Peak Load: Pending-Transaction-based Congestion Detection 132
5.22. Goodput against Injected Load: Comparison of Pending-Transaction-based and Delay-based Congestion Detection 133
5.23. Measurement Set-Up 134
5.24. Time stamps of Java garbage collection 135

Bibliography

[1] 3GPP. High Speed Downlink Packet Access (HSDPA); Overall description; Stage 2. TS 25.308, 3rd Generation Partnership Project (3GPP), September 2009.

[2] 3GPP. 3rd Generation Partnership Project. http://www.3GPP.org, April 2012.

[3] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, Internet Engineering Task Force, September 2009.

[4] Boon S. Ang. An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing onto an i9 60RN-based iNIC. Technical report, January 2001.

[5] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. Diameter Base Protocol. RFC 3588, Internet Engineering Task Force, September 2003.

[6] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428, Internet Engineering Task Force, December 2002.

[7] J.S. Chase, A.J. Gallatin, and K.G. Yocum. End System Optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68 –74, April 2001.

[8] M. Davis, A. Phillips, and Y. Umaoka. BCP 47 Extension U. RFC 6067, Internet Engineering Task Force, December 2010.

[9] Luc Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[10] Josef Dutter, Peter Filzmoser, and Friedrich Leisch. Statistik und Wahrscheinlichkeitsrechnung. http://www.statistik.tuwien.ac.at/public/dutt/vorles/mb_wi_vt/mb_wi_vt.html, November 2011.

[11] Christoph Egger. A Charging Prototype for the IP Multimedia Subsystem. Master's thesis, Vienna University of Technology, July 2006.

[12] Christoph Egger, Marco Happenhofer, Joachim Fabini, and Peter Reichl. BIQINI: A Flow-based QoS Enforcement Architecture for NGN Services. In *Proceedings of the 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, Berlin, Germany, May 2010.

[13] Christoph Egger, Marco Happenhofer, Joachim Fabini, and Peter Reichl. Collapse Detection and Avoidance for SIP Architectures. In *Praxis der Informationsverarbeitung und Kommunikation*, volume 35, pages 91–99, May 2012.

[14] Christoph Egger, Marco Happenhofer, and Michael Hirschbichler. A Study of SIP Proxy Load Patterns. In *International Workshop on Measurements and Networking*, Capri Island, Italy, October 2011.

[15] Christoph Egger, Marco Happenhofer, and Peter Reichl. SIP Proxy High-Load Detection by continuous Analysis of Response Delay Values. In *The 19th International Conference on Software, Telecommunications and Computer Networks*, Split and Adriatic Islands, Croatia, September 2011.

[16] Christoph Egger, Michael Hirschbichler, and Peter Reichl. Enhancing SIP Performance by Dynamic Manipulation of Retransmission Timers. In *6ter Workshop Leistungs-, Zuverlaessigkeits- und Verlaesslichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen (MMBnet11)*, Hamburg, Germany, September 2011.

[17] ETSI. NGN Congestion and Overload Control; Part 2: Core GOCAP and NOCA Entity Behaviours. Technical Report ES 283039-2, Telecommunications and Internet converged Services and Protocols for Advanced Networking, January 2010.

[18] Joachim Fabini, Peter Reichl, Christoph Egger, Marco Happenhofer, Michael Hirschbichler, and Lukas Wallentin. Generic Access Network Emulation for NGN Testbeds. In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, Innsbruck, Austria, 2008.

[19] H. Fathi, S. S. Chakraborty, and R. Prasad. Optimization of SIP Session Setup Delay for VoIP in 3G Wireless Networks. *IEEE Transactions on Mobile Computing*, 5(9):1121–1132, September 2006.

[20] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, Internet Engineering Task Force, June 1999.

[21] V. Gurbani, V. Hilt, and H. Schulzrinne. Session Initiation Protocol (SIP) Overload Control. Internet-Draft draft-ietf-soc-overload-control-05, Internet Engineering Task Force, October 2011. Work in progress.

[22] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, Internet Engineering Task Force, July 2006.

[23] Marco Happenhofer. *Transaction-based Signaling Performance in Next Generation Networks*. PhD thesis, Vienna University of Technology, Institute of Telecommunications, March 2011.

[24] Marco Happenhofer and Christoph Egger. Implicit SIP Proxy Overload Detection Mechanism based on Response Behavior. In *The 19th International Conference on Software, Telecommunications and Computer Networks*, Split and Adriatic Islands, Croatia, September 2011.

[25] Marco Happenhofer, Christoph Egger, and Peter Reichl. Quality of Signaling: A new Concept for Evaluating the Performance of Non-INVITE SIP Transactions. In *Teletraffic Congress (ITC), 2010 22nd International*, September 2010.

[26] Marco Happenhofer and Peter Reichl. Quality of Signaling (QoSg) Metrics for Evaluating SIP Transaction Performance. In *The 18th International Conference on Software, Telecommunications and Computer Networks*, pages 270–274, Split and Adriatic Islands, Croatia, September 2010.

[27] V. Hilt, E. Noel, C. Shen, and A. Abdelal. Design Considerations for Session Initiation Protocol (SIP) Overload Control. RFC 6357, Internet Engineering Task Force, August 2011.

[28] Volker Hilt and Indra Widjaja. Controlling Overload in Networks of SIP Servers. In *2008 IEEE International Conference on Network Protocols*, pages 83–93, Orlando, FL, USA, October 2008.

[29] Yang Hong, Changcheng Huang, and James Yan. Analysis of SIP Retransmission Probability using a Markov-Modulated Poisson Process Model. In *Network*

Operations and Management Symposium (NOMS), 2010 IEEE, pages 179–186, Osaka, Japan, April 2010.

[30] A. Houri, E. Aoki, S. Parameswar, T. Rang, V. Sing, and H. Schulzrinne. Presence Interdomain Scaling Analysis for SIP/SIMPLE. Internet-Draft draft-ietf-simple-interdomain-scaling-analysis-08, Internet Engineering Task Force, 2009. Work in progress.

[31] HP. SIPp is a free Open Source Test Tool and Traffic Generator for the SIP Protocol. `http://sipp.sourceforge.net/`, January 2012.

[32] IETF. IP Performance Metrics (IPPM). `http://datatracker.ietf.org/wg/ippm/charter/`, March 2012.

[33] iptel. About SIP Express Router. `http://www.iptel.org/ser`, December 2011.

[34] Raj Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons, Inc., 1991.

[35] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) Basic Call Flow Examples. RFC 3665, Internet Engineering Task Force, December 2003.

[36] M. Jurvansuu, J. Prokkola, M. Hanski, and P. Perala. HSDPA Performance in Live Networks. In *IEEE International Conference on Communications*, pages 467–471, Glasgow, Scotland, June 2007.

[37] Alexander A. Kist and Richard J. Harris. SIP Signalling Delay In 3GPP. In *Sixth International Symposium on Communications Interworking of IFIP*, pages 13–16, Montreal, Canada, 2002.

[38] Stephen S. Lavenberg. Computer Performance Modeling Handbook. Academic Press, Inc., Orlando, FL, USA, 1983.

[39] Averill M. Law and David W. Kelton. Simulation Modelling and Analysis. McGraw-Hill Education - Europe, April 2000.

[40] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30, January 1998.

[41] P.V. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, Internet Engineering Task Force, November 1987.

[42] Jeffrey C. Mogul. TCP Offload is a Dumb Idea whose Time has Come. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, page 5, Berkeley, CA, USA, 2003. USENIX Association.

[43] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory: The Mathematics of Computer Performance Modeling*. Springer, May 1995.

[44] A. Niemi. Session Initiation Protocol (SIP) Extension for Event State Publication. RFC 3903, Internet Engineering Task Force, October 2004.

[45] E. Noel and C. R. Johnson. Novel Overload Controls for SIP Networks. In *21st International Teletraffic Congress*, Paris, France, 2009.

[46] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330, Internet Engineering Task Force, May 1998.

[47] J. Postel. User Datagram Protocol. RFC 0768, Internet Engineering Task Force, August 1980.

[48] J. Postel. Transmission Control Protocol. RFC 0793, Internet Engineering Task Force, September 1981.

[49] JSIP Project. JSIP: JAVA API for SIP Signaling. `http://jsip.java.net/`, December 2011.

[50] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265, Internet Engineering Task Force, June 2002.

[51] J. Rosenberg. Requirements for Management of Overload in the Session Initiation Protocol. RFC 5390, Internet Engineering Task Force, December 2008.

[52] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263, Internet Engineering Task Force, June 2002.

[53] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[54] J.H. Schuringa. *Mobile Routing Agents deployed in Resilient Packet Networks.* PhD thesis, Vienna University of Technology, Institute of Broadband Communications, 2004.

[55] Otto Spaniol and Simon Hoff. *Ereignisorientierte Simulation - Konzepte und Systemrealisierung.* Thomson Publishing, 1995.

[56] R. Sparks. Actions Addressing Identified Issues with the Session Initiation Protocol's (SIP) Non-INVITE Transaction. RFC 4320, Internet Engineering Task Force, January 2006.

[57] J. Stadler. *Advanced Models for SIP-based Next Generation Networks: Improvements in Performance and Functionality for SIP Network Components and Terminals.* PhD thesis, Vienna University of Technology, Institute of Communication Networks, 2003.

[58] R. Stewart. Stream Control Transmission Protocol. RFC 4960, Internet Engineering Task Force, September 2007.

[59] Abraham Wald. *Sequential Analysis.* John Wiley & Sons, Inc., June 1947.

[60] L. Wallentin, M. Happenhofer, C. Egger, and J. Fabini. XML meets Simulation: Concepts and Architecture of the IBKSim Network Simulator. In *Simulation News Europe*, volume 20, pages 16–20. Eurosim, 2010.

[61] Bang Wang, K. I. Pedersen, T. E. Kolding, and P. E. Mogensen. Performance of VoIP on HSDPA. In *61st IEEE Vehicular Technology Conference*, pages 2335–2339, Stockholm, Sweden, 2005.

[62] Hong Yang, Huang Changcheng, and J. Yan. Stability Condition for SIP Retransmission Mechanism: Analysis and Performance Evaluation. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 387–394, Ottawa, Canada, July 2010.

[63] Jia Zou, Wei Xue, Zhiyong Liang, Yixin Zhao, Bo Yang, and Ling Shao. SIP Parsing Offload: Design and Performance. In *IEEE Global Telecommunications Conference*, pages 2774–2779, Washington, DC, USA, November 2007.