Process Models for High Performance Telecom Systems

S T A F F A N L U N D S T R Ö M

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-97

Abstract

The process model is a keystone in server architectures. With process models we mean how and when operating system processes are created, managed and terminated at application level. A fine-grained number of processes provide fault isolation and concurrent service of incoming requests. However, fine-grained processes often lead to deteriorated performance at the same time as concurrency also can be attained through multithreading or asynchronous I/O.

In this thesis, some selected process models are analyzed and evaluated for usage in telecom systems. Modern telecom systems have requirements on real-time performance for multimedia service and constant availability for emergency calls. It is shown that models with few and long-lived processes perform better in terms of capacity and latency, but that the difference is fairly low on the telecom server platform TSP.

Processmodeller för högpresterande telekomsystem
Examensarbete

Sammanfattning

Processmodellen är en fundamental byggsten i serverarkitekturer. Med processmodeller menar vi hur och när operativsystemsprocesser skapas, hanteras och termineras på applikationsnivå. Många och små processer innebär en liten feldomän och semiparallell hantering av inkommande förfrågningar till servern. Men många och små processer leder också ofta till försämrad prestanda samtidigt som samma nivå av parallellitet kan uppnås genom multitrådning eller asynkron I/O.

I detta examensarbete analyseras och utvärderas några utvalda processmodeller med avseende på prestanda och tillämpbarhet på telekomsystem. Moderna telekomsystem måste klara av multimediatjänster med realtidsprestanda och konstant tillgänglighet för nödsamtal. Det visas att modeller med få och långlivade processer presterar bäst, men att skillnaderna är tämligen små på telekomplattformen TSP.

Acknowledgements

I would like to thank Ericsson AB, Staffan Pernler, Bernth Gustavsson and Ulf Lundström for giving me this opportunity, Per Nilsson for industry supervision, Daniel Lind for TSP expertise, Mikhail Soloviev and Martin Regner for lab support and Robert Rönngren for examination. As it is my last course, I would also like to thank KTH for providing me an excellent education.

Last but not the least I would like to thank my family for supporting me during all these years.

Staffan Lundström
Stockholm, 25th November 2005

Contents

Chapter 1

Introduction

Telecom systems of today are more and more built on software, with fewer features implemented in specialized hardware. This means much higher requirements on software architectural issues, to ensure cost efficient solutions capable of handling huge amounts of subscribers and traffic on clusters consisting of commercial off-the-shelf processors. In response to the merger of the cellular infrastructure with the packet-switched Internet, a framework called the IP Multimedia Subsystem is defined in the third generation of mobile systems [40]. The framework is designed as a network of loosely coupled components. Proxy servers take a central role, since they route messages between mobile end users. By having end-to-end messages passing through several nodes in the core network, latency is at the risk to increase. Signaling, which used to be based on numbers, is replaced by text-based protocols such as SIP, which puts performance even more in focus.

These changes result in high performance requirements on network products, while traditional requirements such as robustness at the same time should be maintained. Server architectural issues are of highest importance.

1.1 Goal

The goal of this Master's thesis is to study and evaluate how process models in a SIP proxy server affect performance, robustness, and complexity. The assumption is that a process model with long-lived processes performs better than a process model which restarts its processes. The analysis should result in a recommendation on the best way forward for a converged network product by Ericsson, henceforth referred to as *"the product"*.

1.2 Scope

The thesis scope is limited to solutions on server process models, which means that product functionality and other technical issues related to functionality are excluded from the thesis. The measurements are restricted to the telecom server platform TSP. Improvements of TSP itself are not considered.

1.3 Comparison to Related Work

There are in particular two differences between this work and much related work. This work is based on TSP whereas most other research projects as well as commercially developed servers assume Unix like behavior of the operating system. Process creation costs are high on Unix, thus giving strong motivations for implementing pooling architectures (described in Section 4.3). It is not obvious that TSP's characteristics lead to the same conclusions. The other important difference is the availability requirements. This server is intended for telecommunications. The availability requirements in telecom are far higher than in ordinary data communications, due to the fact that telecom networks serve as emergency infrastructure. This requirement affects architectural design choices.

1.4 Overview of Document

Chapter 1 presents a longer problem statement as well as the goal and scope of the thesis. Chapter 2 introduces the Session Initiation Protocol, which is the application protocol that this proxy server (the product) operates on. Chapter 3 is on the IP Multimedia Subsystem, which define the framework and the usage context of the product. Chapter 4 is an essential chapter about server architectures, presenting the process modeling problem and examining different approaches to process modeling. Chapter 5 describes a server platform, specialized for the telecom environment, called the Telecom Server Platform. Chapter 6 discusses solutions to the process modeling problem and presents tests on prototypes. Chapter 7 analyzes results from the tests and draws conclusions about the relative performance of the process models. Chapter 8 summarizes analytical and experimental conclusions, and gives a recommendation on ways forward for *the product.* Appendix A contains abbreviations that are used in the document. Appendices C, D and E contain test data.

Chapter 2

Session Initiation Protocol

This chapter, as well as the following chapters, presumes elementary knowledge of data communications, for example the layers in the Internet model and fundamental concepts like client/server architectures. For the reader lacking this background, Forouzan [2] gives a good introduction to data communications and the TCP/IP protocol suite.

Client/server architecture denotes a paradigm, where clients send request messages to servers and servers respond with response messages. Servers and clients are entities in a network. In this thesis, the most important network entity is the *proxy server*, or the proxy in short form. A proxy is both a server and client. It is defined by Internet Engineering Task Force [39] as "an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients".

Servers and clients operate on protocols in the application layer. The *Session Initiation Protocol* (SIP) is a protocol in the application layer. SIP has its origins in the Internet community and is developed by the Internet standards body, the Internet Engineering Task Force (IETF) [39]. The TCP/IP protocols, as well as other so called Internet protocols, are standardized by the IETF, which assures consistent design goals, reuse of components and interoperability between different protocols and network layers. RFC 3261 [3] is the specification document of SIP. Besides RFC 3261 there are a number of extensions to SIP defined in other RFCs or Internet drafts.

SIP can operate over several different transport protocols. SIP has been chosen as the session control protocol for the IMS domain of 3G [1]. IMS is described in Chapter 3.

2.1 Functionality

SIP is designed to establish, modify and terminate multimedia sessions with one or more participants. Typically in this context a multimedia session is a telephone call, but SIP is independent of the type of multimedia session. The media are determined by a session description protocol such as the Session Description Protocol (SDP) [4]. The purpose of SIP is to deliver the session description to the right user(s) at their current location. Possible multimedia sessions apart from audio calls are videoconferences, shared whiteboards, gaming sessions et cetera.

SIP supports five aspects of session establishment and handling [3]:

- User location: determination of which end system to communicate with;
- User capabilities: determination of the media and media parameters;
- User availability: determination of the willingness of the called party to engage in the invited session;
- Call setup: establishment of call parameters at both the called and the calling party;
- Call handling: termination and transfer of calls.

SIP can invite parties to both unicast and multicast sessions and the initiator does not need to be part of the session to which it is inviting. Both persons and machines can participate. Sessions can be modified whenever the users want, i.e. media and participants can be added to an existing session by delivering new session descriptions.

2.1.1 User Moblity

SIP must support a mechanism to locate and identify the called party in order to deliver a session description. A session protocol that is designed for telecom purposes must also allow user mobility in the localization mechanism. User mobility means seamless movement between different networks during and between sessions without changing identity, interrupting the session or losing the localization ability.

The IP-address is not enough for user identification because of two reasons: First, an IP host is not the same as user identification, since an IP host can be used by more than one user and a user can move between different computers and networks. Second, an IP address is related to a specific network and is not designed to be mobile.

The user identification defined in SIP is called SIP Uniform Resource Identifier (SIP URI). URI is a general concept defined in [5] for referencing anything that has identity. The SIP URI format resembles a mailto URI [12], which has the format `user@host`. The user part is a user name or a telephone number. The host part is a domain name or a numeric network address. Additionally, SIP URIs can contain a number of parameters, separated by semi-colons. Figure 2.1 shows two examples of SIP URIs belonging to a user called Anna Nilsson. Let us say that the first URI in the example is defined to be her *Public URI* (the Public URI is also referred to as the Address of Record (AoR) [3]). The Public URI points to a domain with a location service that can map her Public URI to another URI (see the paragraph about location service on page 6). It could for example be mapped to her university URI or cellular URI depending on where she is and how she wants to communicate. All addresses can easily be distilled into the single Public URI, which she can print on her business card.

```
sip:anna.nilsson@domain.com
sip:anna@kth.se; transport=tcp
```

Figure 2.1. Examples of SIP URIs

2.2 SIP Entities

A network architecture supporting SIP communication requires a set of entities for routing purposes. These network entities can be viewed as logical functions and not necessarily identifiable as physical nodes. It is not unusual that one physical entity behaves as several logical entities. This is typically true for complex servers, whose behavior depends on the particular situation. Putting "SIP" in front of a network entity name (e.g. SIP server) emphasizes that the network entity specifically is capable of handling the SIP protocol.

A *server* is "any network element that receives requests in order to service them and sends back responses to those requests" [3]. The response accepts, rejects or redirects the request. A SIP server receives SIP requests and returns SIP responses. Server is a general term, which among others covers proxy servers, user agent servers, redirect servers, and registrars.

A *client* is "any network element that sends SIP requests and receives SIP responses" [3]. User agent clients and proxy servers are clients.

A *user agent* (UA) is an endpoint, that is an application that interacts with the user. Sessions are typically established between user agents. The calling user agent —behaving in this situation as a client— initiates the SIP request. The called user agent —behaving in this situation as a server— contacts the user when a SIP request is received and returns a SIP response on behalf of the user.

A *proxy server* is "an intermediary program that acts as both a server and a client for the purpose of making requests on behalf of other clients" [3]. In other words, it receives a SIP message from a user agent or from another proxy and routes it toward its destination. A proxy interprets, and, if necessary, modifies a request message before forwarding it.

A *redirect server* is a server that accepts a SIP request, maps the address into zero or more new addresses and redirects the client to these addresses with a redirection response message (see categorization of response messages in Table 2.2 on page 7). A redirection server does not forward SIP requests, as a proxy server does, and it is not an endpoint as a user agent is.

A *registrar* is a server that registers a user's location. It accepts SIP REGISTER messages from users and updates its domain's location server with the user's current location. The registrar functionality is usually co-located with a proxy or a redirect server and acts as a front-end to the location server.

A *location server* offers information about a callee's possible location(s). Location servers are not SIP entities because they do not communicate using SIP. Hence, they are usually co-located with a registrar. Since it is not a SIP entity, it is clearer in a SIP context to talk about a location service instead of a location server. Proxy and redirect servers use the location service.

2.3 Protocol Format

The SIP protocol is text-based and SIP messages are human readable. The benefits of text-based protocols are ease of understanding, ease of debugging and ease of implementation. The messages do not have to be interpreted by a network analyzer tool for human understanding. The drawback is inefficient use of bandwidth; it yields longer messages than its binary encoded counterparts.

HTTP [25] is a well-known text-based predecessor to SIP. The SIP message format is based on HTTP. Both SIP and HTTP employ a request/response model. Clients send requests and servers send back responses. A message consists of a start-line, one or more header fields, an empty line indicating the end of the headers and an optional message-body (see general description in Figure 2.2 and actual examples in Appendix C). The start line is referred to as the request line for a request message and the status line for a response message. Apart from the start line the formats are the same for a request and a response message.

```
Generic-message = start line
                  *message-header
                  carriage-return line-feed
                  [ message-body ]
```

Figure 2.2. SIP message format described in Augmented Backus-Naur form

The request line of a *request message* begins with a method name, followed by a Request-URI and the protocol version (SIP/2.0). The method name specifies the type of request. Table 2.1 on page 7 lists all currently defined method names and their meanings.

The start line of a *response message* is called the status line, which consists of the protocol version, a 3-digit status code and a textual explanation to the status code. Figure 2.3 on page 7 shows an example of a status line. The status code is categorized into six different groups, each starting with a new number. Table 2.2 on page 7 lists and describes the categories.

Method name	Meaning
ACK	Acknowledge the establishment of a session
BYE	Terminate a session
CANCEL	Cancel a pending request
INFO	Transport PSTN telephony signaling or other application layer information
INVITE	Establish a session
NOTIFY	Notify the user agent about a particular event
OPTIONS	Query a server about its capabilities
PRACK	Acknowledge the reception of a provisional response
PUBLISH	Upload event state information to a server
REGISTER	Map a Public URI with the current location of the user
SUBSCRIBE	Request notification about a particular event
UPDATE	Modify some characteristics of a session (like a repeated INVITE but it has no impact on the dialog state)
MESSAGE	Carry an instant message (stand alone in contrast to the session model)
REFER	Instruct the recipient to request a resource

Table 2.1. All the defined SIP request methods (by August 11, 2005) [39]. INVITE, ACK, OPTIONS, BYE, CANCEL and REGISTER are core SIP methods [3]. INFO [13], PRACK [14], SUBSCRIBE and NOTIFY [15], UPDATE [16], MESSAGE [17], REFER [18], PUBLISH [19] belong to SIP extensions.

```
SIP/2.0 180 Ringing
```

Figure 2.3. Example of a status-line in a provisional response

Status code range	Category	Meaning
100-199	Provisional	Indicates progress with request, but is not a final response
200-299	Success	Action successfully received, understood and accepted
300-399	Redirection	Further action needs to be taken by callee to complete the request
400-499	Client error	Bad syntax in request or request cannot be fulfilled at this server
500-599	Server error	Apparently valid request but server failure
600-699	Global failure	Request cannot be fulfilled at any server

Table 2.2. Status code categories [3]. The status codes are used in SIP response messages.

2.3.1 Header Fields

Header fields follow the start line for both requests and responses. Some fields are mandatory and some are optional. A header field consists of the header's name, a colon and the value. Some header types can occur in several entries or in one entry with the values comma separated. The mandatory and most important header fields are `To`, `From`, `Cseq`, `Call-ID`, `Max-Forwards` and `Via`. Section 2.4.2 on page 11 covers the three headers `Contact`, `Record-Route` and `Route`. A full list of SIP headers is given in Appendix B on page 61.

The `To` header (see such a header in Figure 2.4) contains the destination address of the request. Proxy servers on the path to the destination do not utilize this field. The purpose is solely for human usage and end filtering based on recipient address. The `tag` parameter is a random identification number used to distinguish between different user agents with the same URI.

```
To: Thomas A. Watson <sip:Thomas.Watson@bell.com>;tag=1232
```

Figure 2.4. Example of a header field

The `From` header contains the originator address of the request. The purpose is the same as for the `To` header.

The `Call-ID` header uniquely identifies a particular SIP message exchange. `Call-ID` in registration messages is interpreted differently. All registrations from a user agent client should have the same `Call-ID` in order to detect the arrival of unordered REGISTER request.

The `CSeq` (Command Sequence) header contains a sequence number and the method name of the request, for example `CSeq: 3211 INVITE`. The `CSeq` value identifies a request with its response. It enables ordering of the messages within a single call (the call is identified by the `Call-ID`).

The `Max-Forwards` header sets the maximum number of hops that a request can transit. Each proxy it passes will decrement the value by one. If the `Max-Forwards` value reaches zero before the request reaches its destination, it will be rejected with a 483 Too Many Hops error response.

The `Via` header field registers the user agent client (UAC) and all the traversed proxies. The first `Via` value —the value of the UAC— identify the location where the response is to be sent. The other `Via` entries are used to ensure that the response traverses the same proxies as the request did but in the opposite direction. Routing loops can also be detected with the `Via` field.

2.3.2 Message Body

The message body is transmitted end to end, which means that the proxies do not need to parse the body. This property makes the message body uninteresting in this thesis. Nota bene, if the message body is encrypted, then the proxies cannot interpret the message body even if they wanted to.

The message body is separated from the header fields by an empty line. A SIP message can carry any type of body due to MIME encoding [7, 8, 9, 10, 11].

2.4 Message Flow

The reader should now be familiar with the fundamentals of the protocol format as well as the routing entities in a SIP network. With the basics in place it is time to introduce *transactions* and *dialogs*. They describe how SIP messages are combined and processed in practice to achieve various results.

2.4.1 Transaction

The general message exchange pattern in SIP is a request message from one user agent to another user agent, followed by a response message in the other direction. The request/response exchange is called a transaction and is identified by having the same `CSeq` header value in all messages. A transaction is terminated by a final response but it can have several provisional responses in between.

The most important transaction is the INVITE transaction. The INVITE transaction is used to initiate sessions. The flow chart in Figure 2.5 assumes that a person A wants to call a person B:

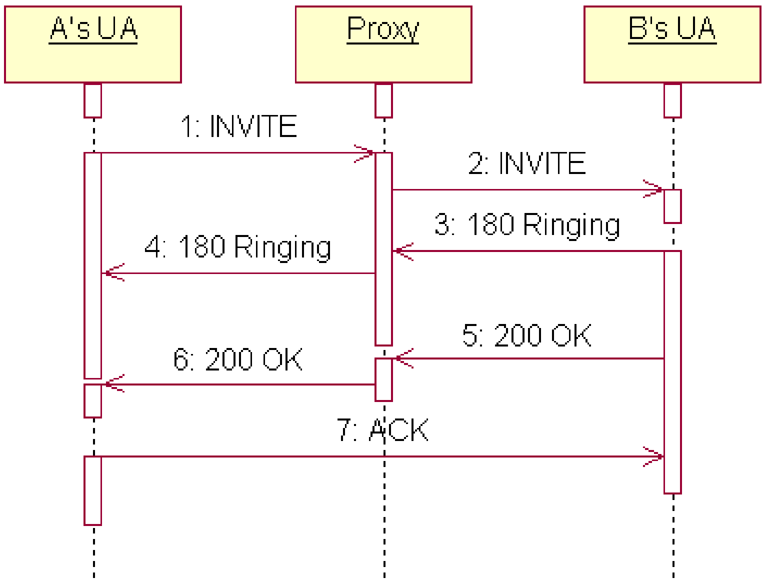

Figure 2.5. INVITE-ACK transaction

- First, A's UA sends an INVITE request to B (1).
- This message is proxied to B's UA (2).

- B's UA responds with a 180 Ringing message (3-4), which is a provisional response.
- After B has accepted the call, B's UA sends the final response 200 OK (5).
- Finally, A sends an ACK request (7) to B in order to acknowledge the final response.
- The call session is established; A and B can talk.

There are some things to notice in this transaction. The 180 Ringing response is provisional, which means that the transaction still is unfinished. But the 200 OK message is a final response and, according to the definition of a transaction, the transaction is completed. Why is there an ACK request followed by no response? The reason to the ACK request is that the INVITE-ACK sequence is a special case of a transaction. This so called three-way handshake is defined to be two transactions, where the ACK request constitutes a transaction of its own but always follows an INVITE transaction. Normally a client expects a fast response from a server on a request. But an INVITE response can take some time due to natural limitations; it takes time for a person to answer the phone, especially if he is doing something else. The ACK ensures the callee that the caller still is on the line.

The BYE transaction terminates a session. The transaction, which is illustrated in Figure 2.6, is also a good example of a normal transaction:

- The BYE transaction begins with a BYE request (1-2).
- A successful session termination ends with a 200 OK response (3-4).

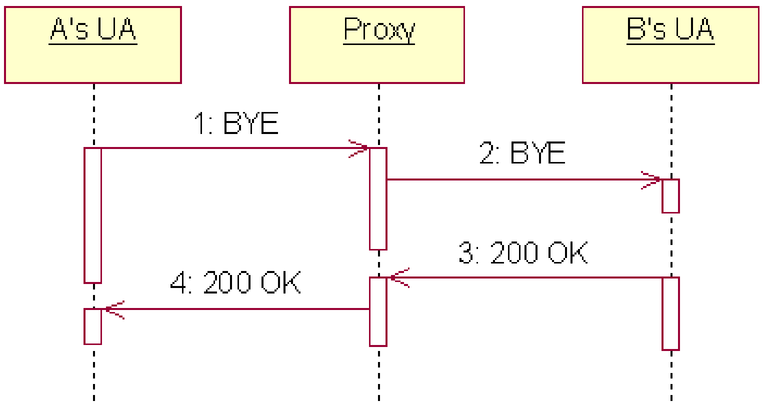

Figure 2.6. BYE transaction

2.4.2 Dialog

The INVITE-ACK transaction and the BYE transaction are related to each other, because both are needed to fulfill a session instance. The first transaction creates a session and the second terminates the session. A session instance is uniquely identified by the combination of the `To`, `From` and `Call-ID` values. It is referred to as a dialog (in early RFCs referred to as a call leg [3]). Except from the headers mentioned in Section 2.3.1 on page 8 there are especially three header fields that are interesting within a dialog: `Contact`, `Record-Route` and `Route`. The `Contact` header field is employed in the callee's response message to inform the caller of a direct route to the callee. While the `Via` header field tells the callee where to send the response, the `Contact` header informs the caller where to send future requests. When a dialog is established, all messages within a dialog follow the same path and it can, if desired, be direct communication from user agent to user agent, since all information needed for this purpose is contained in the `Via` and `Contact` headers. But a proxy might want to stay in the signaling path during the dialog in order to fulfill various control and route mechanisms. In order to stay in the signaling path, there is a header field called `Record-Route`, which the proxy inserts with its own address as value. The user agent carries out the routing directive by inserting the header `Route` with the proxy address as value in subsequent requests.

Chapter 3

IP Multimedia Subsystem

The *IP Multimedia Subsystem* (IMS) is an initiative to merge the cellular infrastructure with the Internet protocols. The motivation of IMS, a standardization by 3GPP [1], is to provide methods for charging, integration of different services, and quality of service. IMS entities use SIP for communication. SIP identification is a subset of IMS identification.

3.1 Motivation

The cellular world reached two billion users worldwide on the 17th of September in 2005 [41]. The second generation of mobile networks (2G) covers virtually all parts of the world where people live and services include telephone calls and simple text messages (SMS). 2G terminals can act as a modem to transmit IP packets over a circuit, which allows a limited access to the Internet services.

In the third generation of mobile networks (3G) there are two domains: the circuit-switched domain and the packet-switched domain. The circuit-switched domain is an evolution of 2G, optimized for voice and video transport. The packet-switched domain use native packet-switched technologies to perform data communications, which enables an Internet access with much higher bandwidth than the access in 2G.

But what is the motivation of IMS? The vision of the IMS is to offer Internet services using ubiquitous cellular technologies, i.e. cellular access available everywhere to web, email, instant messaging, presence, shared whiteboards, VoIP (Voice over IP), videoconferencing et cetera; real-time multimedia services provided across roaming boundaries and different access technologies. IMS is designed to be a robust system merging the Internet protocols with the cellular world. However, in the packet-switched domain of 3G everything on the Internet can be accessed without the need of introducing IMS. Still there are reasons for developing IMS. Camarillo and García-Martín [27] mention three reasons: charging, integration of different services and quality of service (QoS).

3.1.1 Charging

The Internet architecture lacks a good charging mechanism for services, except that operators can charge for a connection without the possibility of separating different types of data. With IMS, operators can charge differently depending on the content and follow the business model they prefer. Application vendors get built-in mechanisms for charging.

3.1.2 Integrated Services

IMS defines standard interfaces based on IETF protocols to service developers. Services using these components can interoperate and be integrated to new services. A voice application and a video application can be combined into a videoconferencing tool and an application can use presence information to enhance the functionality. Furthermore, a subscriber will be able to execute his services in foreign networks and not only in the home network.

3.1.3 Quality of Service

The packet-switched domain provides best effort quality because it relies on the Internet infrastructure. In contrast, the 2G mobile networks offer QoS and allocated resources for a call. An objective of IMS is to offer QoS over IP and allow the operators to differentiate their services offer instead of only selling a bit pipe to Internet independent of the data type.

3.2 Standardization

The standardization of 3G is a collaborative effort between different standardization bodies. The IETF provides the foundation for IMS, which is the protocol specifications of the so called Internet protocols. The 3rd Generation Partnership Project (3GPP) specifies IMS, the architectural framework [40].

According to Camarillo and Garcia-Martin [27] the IMS framework has been designed to meet the following requirements:

- Support for establishing IP Multimedia Sessions.
- Support for a mechanism to negotiate Quality of Service
- Support for interworking the Internet and circuit-switched networks.
- Support for roaming.
- Support for strong control imposed by the operator with respect to the services delivered to the end-user.
- Support for rapid service creation without a standardization process.

- Support for access technology independence

To understand IMS you both need to understand the underlying protocols (such as SIP) and the integrating architectural framework.

3.3 Architecture

There are several important protocols that form the basis of the IMS architecture. SIP is the session control protocol. Authentication, Authorization, and Accounting (AAA) are performed by a protocol called Diameter [20] (its predecessor RADIUS [21] is widely used today when a user connects to his/her Internet Service Provider). Other important protocols in IMS are H.248 [22], RTP (Real-Time Transport Protocol) [23], RTPC (RTP Control Protocol) [23], and COPS (Common Open Policy Service) [24].

The cellular infrastructure can be categorized into *home* and *visited* networks. A user is in the home network, when he is within the area covered by his operator's infrastructure. When he leaves the area, for example he goes abroad or leaves the town, then he either will not be able to use the phone at all or he will enter a visited network where another operator provides the infrastructure. The latter alternative is organized by a roaming agreement between the operators; the user is a "paying visitor" in the network.

A network can also be divided into access networks and core network. The access network can for example be WLAN, ADSL or radio link. IMS is supposed to be independent of the access technology. The user connects to the core network through the access network using an IMS terminal, referred to as *User Equipment* (UE). IMS implements a SIP infrastructure and from SIPs perspective the UE is perceived as a SIP UA.

IMS follows a server/client approach and there are several entities in the core network. 3GPP does not specify any network nodes in the IMS architecture. Rather it specifies *functions* that are accessed through standardized interfaces. Functions can be implemented as separate nodes, but not necessarily. The earlier mentioned converged network product by Ericsson (*the product*) is an example of a function/node in the architecture. Here follows a brief description of some IMS entities:

3.3.1 Control/Session Call Function

The *Control/Session Call Function* (CSCF) is one of the most essential functions in IMS. It is from SIP perspective basically a SIP proxy and a SIP registrar. CSCF is divided into three logical nodes: the *Proxy-CSCF* (P-CSCF), the *Interrogating-CSCF* (I-CSCF), and the *Serving-CSCF* (S-CSCF). The P-CSCF is the public interface to an IMS network. One of the main functions of the S-CSCF is to provide SIP routing services. The I-CSCF is a lightweight SIP proxy server, whose main task is to find the right S-CSCF.

3.3.2 Home Subscriber Server

The *Home Subscriber Server* (HSS) is a user database. It contains persistent user information and is used for example by the CSCF to retrieve information. Data include location information, security information, user profile information and the S-CSCF allocated to a particular user.

If the network contains more than one HSS, then you need a function called SLF (Subscriber Location Function). SLF is simply a database that maps a user's address to the HSS where the user's information is stored. The HSS and the SLF are not SIP entities and communicate via the Diameter protocol. It corresponds to the Location Server described in Section 2.2 on page 6.

3.3.3 Application Server

The *Application Server* (AS) hosts and executes applications and services. AS takes the role either as a SIP UA, a SIP B2BUA (Back-to-Back User Agent, i.e. a concatenation of two UAs), a SIP redirect server or a SIP proxy server depending on the service. AS communicates via the Serving-CSCF.

The AS can be located in a foreign network. It may interface the HSS but only if it is located in the home network.

3.3.4 Media Resource Function

The *Media Resource Function* (MRF) takes care of media-related tasks in the home network. Some tasks are transcoding between different codecs, playing announcements, analyzing media, obtaining statistics and mixing media streams. MRF is partitioned into the MRF Controller (MRFC) and the MRF Processor (MRFP). MRFC acts as a SIP UA and interfaces the S-CSCF. MRFP manages the media-related functions and is controlled by MRFC.

3.3.5 PSTN/Circuit-Switched Gateway

The *Public Switched Telephone Network/Circuit-Switched Gateway* (PSTN/CS) is the interface towards to the old Public Switched Telephone Network and other circuit-switched networks. This enables IMS terminals to communicate with PSTN terminals.

The PSTN/CS gateway is divided into three functions: the Signaling Gateway (SGW), Media Gateway Control Function (MGCF), and the Media Gateway (MGW).

3.3.6 Breakout Gateway Control Function

The *Breakout Gateway Control Function* (BGCF) is a SIP server and routes based on telephone numbers. The BGCF selects a PSTN/CS gateway when an IMS user wants to initiate a session with a PSTN user. If there is no suitable PSTN/CS gateway in the network it selects another network.

3.4 Identification

The discussion about identification started in Section 2.1.1 when discussing SIP. SIP identification can be described as a subcomponent of the identification mechanism in IMS. In IMS subscribers are identified by the *Public User Identity* (Public ID). A Public ID can either be a SIP URI or a TEL URL [6] (A URL is a subset of a URI [5]). A TEL URL (see Figure 3.1) represents a phone number and is needed for backward-compatibility with PSTN terminals that only can handle digits. Depending on the context a Public User Identity can be represented in abbreviated formats, for example a TEL URL in local format, long-distance format or the absolute international format.

```
tel:+46-705-619-508
```

Figure 3.1. Example of a TEL URL in international format

Each subscriber is also assigned a *Private User Identity* (Private ID). The Public User Identity can be extracted from a SIP message but the Private User Identity is not sent over the network. Its purpose is for subscription identification and authentication. A subscriber has one or more Private User Identities, and each Private User Identity is associated with one or more Public User Identities. A Public ID can also be associated with more than one Private ID. However, the normal case is one Private ID per subscriber and at least two associated Public IDs - one SIP URI and one TEL URL. The HSS stores the Private User Identity and the associated Public User Identities.

3.5 Traffic flow through the product

All traffic in an IMS network passes through *the product*. Traffic in telecommunications is called a “call”, which corresponds to a SIP dialog. A call can be described by a half call model, where a half call divides a call into an originating half and a terminating half. The caller’s messages pass through his or her operator’s product in the originating side of a call. The product then forwards the messages to the product at the terminating side, which serves the callee. It can be the same physical node. The product handles two types of traffic flows: registration traffic and session traffic.

The registration traffic is essentially SIP REGISTER transactions. It involves only the originating half of the network because its purpose is to register the location of a UE and not to communicate with another UE. The traffic goes to the product and back to the UE.

The session traffic is all other traffic and it consists of full calls from one UE to another UE as described above. Then both the originating and terminating half is involved in the traffic.

Chapter 4

Server Architecture

Basic knowledge in operating systems is assumed in this and the following chapters. For a good coverage of this topic, see Tannenbaum [26].

There are several ways to design a server. The simplest way is to handle all requests sequentially, but all sophisticated models service several requests concurrently. Concurrency can be achieved by using multiple processes, multiple threads or asynchronous input/output operations. Server performance is measured in terms of serviced jobs per time unit and latency time per job.

This chapter presents several server architectures and defines optimal server characteristics. Many innovations in server architecture have originally been intended for web servers or other specific types of servers. Hence, it does not follow that all ideas are directly applicable on *the product* and other SIP proxy servers built on TSP.

4.1 Characteristics

A server can be thought of as a pipeline, whose input is a flow of requests and output a flow of responses. A server cluster can be thought of as several pipelines in parallel. A server is well-conditioned if it has a performance similar to a pipeline: When the load increases the delivered throughput should increase proportionally until the maximum capacity of the server is reached. Load above maximum capacity should not degrade the throughput, only increase the response time due to queuing. The response time, or the latency, should be roughly constant under light load. Inevitably, the number of requests per time unit to a server is far greater than the number of serving processors, even if a server cluster is employed. This means that the requests can never be handled truly in parallel. Due to this limiting factor, the response time of a well-conditioned service should increase linearly with the number of clients. The impact of a load increase should be equal or policy-determined for the clients.

These are ideal characteristics. A typical experience is a degradation of throughput as load increases and a dramatic increase in latency. In the worst case the server more or less stops functioning when the load is above capacity.

From an architectural point of view there are no big differences between a proxy server and a regular server. Both employ a client/server model, where the client sends a request message to a server and the server services the request and returns a response message. The difference is that a proxy server does not generate the response itself; instead it performs some work on the request and then forwards it to the next node, which can be another proxy server or a server. The proxy server may also handle the response message. If the proxy server handles the response message and the proxy server is stateful, which means that it maintains a state in order to know the context when servicing a request or a response, then it needs to keep a transaction state for a relatively long time between the reqest and response message. This can affect the architecture, since it may be necessary to dispatch both the request and the response to the same process in the operating system.

The general steps involved in serving a request start with the request receipt. The server reads a network socket and parses the incoming message. The processing steps depend on the functional behavior of the server, but they typically involve reads/writes from disks or databases, and possibly network communication with other nodes. Finally, the server sends a response message. The response should be dispatched as soon as logically possible in order to reduce latency. Cleanup tasks can be taken care of afterwards.

4.2 Single-Process Serialized Server Architecture

The *single-process serialized server architecture* (SPS) [32] is the naïve approach to server architecture. The model sequentially accepts a request and services it to the end before taking on the next request. The sequential model is easy to implement and requires only one process and one thread of execution. But the model is unacceptable because any model must be dimensioned to handle a massive amount of requests in parallel and this model processes them sequentially. Different users make requests independently of each other and do not expect to wait in a queue before they can make a call. Even though the actual execution in a CPU is serialized, this model is still unacceptable in comparison to a concurrent model due to two reasons: First, a long job can block all other jobs for a long time, thus giving short jobs poor and unreliable latency. Second, the CPU is not utilized efficiently. A normal job involves I/O operations, during which the CPU is idle. I/O (input/output) refers to operations that transfer data into or out from the primary memory of a computer to/from peripheral devices. If no other jobs are scheduled during these wait states, then CPU-time is wasted.

SPS is only acceptable as long as scheduling predictability is not important, as is the case with low-priority background tasks. However, most server applications, including SIP applications, are oriented around processing large numbers of short-lived tasks that are triggered by external events. The desire to efficiently handle parallel tasks lead to hardware clusters and concurrent models. Concurrency is accomplished by using processes, threads and/or asynchronous I/O, which requires support by the operating system. The more sophisticated models utilize a balanced combination of these techniques. Most performance optimizations either reduce initialization time or exploit cache effects.

4.3 Multi-Process Server Architecture

The *multi-process server architecture* (MP) [34, 32] assigns a process for sequentially executing the basic steps associated with serving a client request. Concurrent request handling is achieved by utilizing multiple processes at the same time. The operating system transparently takes care of the scheduling according to some policy.

The simplest way to implement a MP model is to have a master process with an infinite while-loop. The process accepts new connections in the loop and for each new incoming request the master process creates a new process to which it dispatches the job. The master process is free to listen for new requests and start new worker processes without having to wait for a worker process to finish. The worker process exits itself upon completion of the job.

In a Unix context this model is called the *forking model*, because the system call for creating a new process is called "fork". The benefit of the forking model is the simplicity of the master process, which is a good guarantee for robustness. Furthermore, a process is the smallest fault domain in an operating system. If a worker process fails due to some software error, only one request is affected. If all requests were handled by one process, then all requests would be lost in a process failure. Another benefit of the forking model is that small memory leaks are not a problem. The concept of an operating system process comprises a private address space, which is assigned by the OS kernel. Since the worker processes are extremely short-lived, there is no risk that bad memory management by the programmer leads to significant memory leaks by accumulation. When a process exits, the address space is automatically garbage collected.

The first web servers, the CERN httpd and the NCSA httpd, were forking Unix servers [32]. Despite its simplicity, the forking model is considered obsolete for web servers due to performance reasons [32]. Creating a new process takes considerable time and in the forking model this processing time grows proportional with the number of requests.

The first breakthrough in high performance web server engineering was the introduction of the *pooling model* [32]. The multi-process pooling model assigns, as in the forking model, a process to each request. The improvement is a pre-creation of a fixed number of processes at server startup time, which forms a pool of long-lived worker processes. The master process assigns a job to a process in the pool, which performs the job and returns to the pool upon completion. A queue could be used to synchronize the distribution of jobs to free worker processes, but the model is independent of the specific data structure, hence the more general term pool is employed instead of queue. If there are more incoming requests than available processes, the requests will simply be queued in the master process. The free web server Apache employed the process pooling model in its initial version 1.x [36].

The benefit of the pooling model compared to the forking model is apparent. Reuse of a process eliminates system calls and the associated process initialization and termination overheads. Since this procedure repeats for each request, the performance benefit is considerable. The introduction of long-lived processes may require a method to deal with memory leaks, because often servers are implemented using programming languages that lack automatic garbage collectors due to requirements on predictability of performance. Memory leaks can be handled in two ways [32]: by limiting the lifetime of a worker process or by using memory pools. Limitation of a worker process lifetime means that after a specific time or a specific number of handled requests a new process with fresh memory replaces the old process. Memory pools means that specialized dynamic memory management routines are employed instead of standard routines like `malloc()` and `free()` or `new()` and `delete()`. Since a process in the pool is regarded as a new process, the dynamic memory management routine could simply have a method that marks all dynamic memory it controls as free. It could be implemented as a wrapper to `malloc()`, which makes one large request to `malloc()` at the beginning of the job and one invocation to `free()` at the end of the job. Pointers are used to keep track of where the allocated memory starts and ends. Several memory pools can be used if needed. No time-consuming freeing of memory to the OS is necessary.

4.4 Multi-Threaded Server Architecture

The *multi-threaded server architecture* (MT) [34, 32] is similar to MP. The conceptual difference between processes and threads is that threads share memory and other resources. A process is a container of one or more threads. The execution state is associated with the thread, and multiple threads yield concurrency. Similar to the forking model one thread is spawned per job and exited upon completion. Let us denote it the *spawning model.* The spawning model can be enhanced to a *thread pooling model* in the same way as the forking model. It even allows an optimization, where the workers read directly from the socket, which makes the master thread redundant. This is not easily implemented for processes because processes do not share sockets. MT requires OS support for kernel threads in order to effeciently schedule runnable threads [34]. It is not impossible to share memory between processes.

Except from the conceptual difference there are very important performance and robustness differences between multiple processes with one thread each and one process with multiple threads. Threads are more lightweight, i.e. starting and managing a thread is less resource consuming than it is for a process. Using a MT architecture yields improved performance in comparison to MP. But processes are, as earlier pointed out, the smallest fault isolation domain, so if one thread in a process crashes, then this often results in that the whole process and, consequently, all its threads go down. A MP architecture improves robustness in comparison to MT. It is a trade-off between performance and robustness.

It is in many cases easier to perform optimizations in a MT architecture due to the shared memory. If several jobs request the same data, then the information only needs to be read once to the shared heap memory. However, shared memory together with concurrency brings in a more difficult concurrent programming model. The programmer must ensure the consistency of shared data, because the system does not guarantee a specific instruction order of two concurrent kernel threads. The kernel can change the executing kernel thread between any machine instructions. Consistency is achieved through various synchronization mechanisms, e.g. condition variables, mutexes or monitors, which bundle sequences of instructions into atomic operations.

Two cases when MT is more manageable for optimizations than MP are information gathering and application-level caching. They are similar types of tasks that frequently occur. The MP model must gather information via time-consuming IPC operations. It also leads to multiple caches instead of one due to non-shared data, which imply more misses and inefficient memory consumption. The MT model can be implemented with a single cache and a global space for information data, but accesses/updates must be synchronized, which can lead to lock contention. The SPED architecture, as we will see in the next section, needs neither IPC nor synchronization to share information.

The MP and MT models can be combined into process pools, where each process contains a pool of prespawned threads. Apache version 2.x [36] is based on this architecture.

4.5 Single-Process Event-Driven Server Architecture

The *single-process event-driven server architecture* (SPED) [34, 32] uses a single process and a single thread of control to process multiple jobs. The key is to use non-blocking system calls to perform I/O operations. The process is asynchronously notified about the completion of disk and network operations as these operations often are time-consuming. The CPU can overlap operations instead of having to wait idle. SPED is often designed as a state machine, where each job performs several steps. The steps are interleaved with steps associated to other jobs.

Concurrency is already achieved in MP and MT, but the benefit of SPED is that the overhead of context switching and thread synchronization is avoided. The memory requirements are smaller, since it has only one process and one stack. However, the OS must provide true asynchronous support for all I/O operations, which is not always the case [34].

[35] compared a simple MT server, having one pre-allocated thread per task, with an event-driven of the server, having only one thread (and one process) to handle all jobs. Each task consisted of a 8 KB read from a disk file. It always read

the same file, so the data was always in the buffer cache. The experiment yielded large performance variations. For the MT version, throughput reached its peak at approximately 10 concurrent threads (one thread per task), having a throughput of approximately 25,000 tasks per second. The throughput started to degrade substantially after reaching 64 concurrent threads. Response time approached infinite as the number of concurrent threads increased to 1,000 and beyond. The server collapsed in other words. For the event-driven version, throughput reached a peak value above 30,000 tasks per second, when having 64 tasks in pipeline. The major difference was that the throughput remained constant at almost maximum throughput as load gradually increased to 1,000,000 tasks in pipeline. Response time increased only linearly as the number of tasks increased, which equals the optimal latency behavior described earlier. A conclusion from this test is that event-driven architectures scale better. The management and context-switching overhead of threads and processes grows as there are more threads and processes to manage.

The Harvest and Squid proxy servers employ the SPED architecture [32] as well as the Zeus web server [38].

4.6 Asymmetric Multi-Process Event-Driven Server Architecture

The *asymmetric multi-process event-driven server architecture* (AMPED) [34] combines SPED with MP or MT. In general it behaves like SPED, but dedicated helper processes (or threads) handle blocking calls that cannot be re-architected asynchronously. The processes communicate via an interprocess communication (IPC) channel and task completion is notified back via the IPC as any other asynchronous event. IPC between the server and the helper processes implies an extra cost, but when the alternative is a blocked server it is better to put work on dedicated processes.

The Netscape Enterprise Server employs the AMPED model [32, 37].

4.7 Staged Event-Driven Server Architecture

The *staged event-driven server architecture* (SEDA) [33, 35] uses *stages* as the fundamental unit of processing. Applications are programmed as a network of event-driven stages connected by explicit queues. A stage is a self-contained application component consisting of a group of operations and private data. Conceptually, a stage resembles a class in an object-oriented language, but an object is a data representation, which for example functions, threads, or stages act on, while a stage is a control abstraction used to organize work. An operation is an asynchronous procedure call, i.e. invocation, execution and reply are decoupled. Stages have scheduling autonomy over its operation, which allows it to control the order and concurrency with which its operations execute. An operation executes sequentially, non-preemptible and can invoke any number of synchronous and asynchronous operations. The programmer designs operations to which the system dispatches asynchronous events. An operation must relinquish the given control over the execution thread in order to let other asynchronous operations execute. Asynchronous operations are triggered by events, such as network messages, completed disk operations, internal calls from asynchronous operations, timed-out timers, synchronizations, system updates, etc.

Continuing on the pipeline metaphor in the beginning of Section 4.1, the pipeline is divided into several stages, where each stage solves a subtask of the job. The pipeline does not have to be linear. The approach can be generalized to a finite state automaton. The orthogonal difference between SEDA and the earlier described architectures is that SEDA is flow-centric, not resource-centric.

The MP model encapsulates a job in a process, which performs all steps. A stage handles only one step but for a batch of jobs. The former model gives an easy abstraction and fault isolation. The latter model tries to leverage on hardware mechanisms such as caches, TLBs, and branch predictors. Processor speed and parallelism has improved rapidly, while the memory access time has only improved slowly during the last decades [33]. The previously mentioned mechanisms are attempts to alleviate this performance gap, by storing data that are likely to be re-used in fast media. All these mechanisms assume that programs exhibit locality, i.e. previously executed code is likely to repeat and contiguous data are likely to be accessed. But server software displays less spatial and temporal locality due to the few loops and the short period of execution before another process or thread is scheduled [33]. Studies referenced by [33] have found that online database systems perform only a tenth of its peak potential, because of high cache miss rates. The goal of SEDA is to schedule flows of similar operations sequentially instead of interleaving diverse operations belonging to different stages. Larus improved server throughput by 20% and reduced L2 cache misses by 50% using SEDA compared to MT [33].

Few operating systems have built-in support for stages. Almost all OSes are designed with the process construct in order to provide a virtual machine view, which hides the concurrency with other processes from the programmer's view. Stages can of course be designed within one process or one process can correspond to one stage. The first design is fragile and the second model risks that IPC consumes all gains.

Chapter 5

Telecom Server Platform

The product is implemented on the *Telecom Server Platform* (TSP) [30, 28, 29, 31]. In order to model an efficient proxy server it is essential to understand the platform. This chapter describes TSP.

5.1 Design and Characteristics

TSP provides server functionality but it is separated from the gateway functionality unlike Ericsson's telephone switch AXE. TSP is an Ericsson technology, but it is to some extent based on commonly available components and open standards. TSP includes a processor cluster, two operating systems, clusterware, a distributed object oriented database and an associated development environment. TSP furthermore includes a run time environment for Java and a CORBA-compliant object request broker. Communication with the external world is IP based.

The hardware consists of commercial off-the-shelf components, e.g. Intel Pentium processors. The two operating systems in TSP are Linux and Dicos —the latter is an Ericsson developed operating system. Dicos is intended for real-time and mission critical tasks. Subsequently, the normal traffic flow is handled by Dicos. Linux takes care of host operation and maintenance (O&M) as well as providing a standard programming environment for integration of third-party software. Since the thesis only concerns traffic processing, the description of TSP is focused on the Dicos part.

Key design goals of TSP have been:

- Reliability
- Scalability
- Real-time operation
- Openness

The strongest emphasis has been put on reliability. Existing telecommunication systems already provide a reliable and almost constantly available emergency infrastructure and new products should be able to match their predecessors' working.

Scalability means that operators should be able to add new processors to the cluster and get an equivalent increase in capacity without having to replace the old hardware.

Real-time operation is a necessary feature of a telephone conversation. In telecommunications soft real-time performance is sufficient [28], which means that performance is specified in terms of statistics, e.g. a certain operation is required to perform 90% of the time within a certain time limit, in contrast to hard time limits.

Openness in TSP means

- Non-specialized hardware: to always benefit from the latest commercially available hardware technology;
- Standard programming languages: to have access to a large developer community;
- Interoperability: to communicate with external systems using standard protocols;
- Compatibility: to run third-party software on standard operating systems.

5.2 Process Types

There are two main categories of process types in TSP: static and dynamic. Static processes are created when the system is started or when the process is installed. They are also recreated after a failure. Dynamic processes are created when they are addressed by another process and do not restart after failure. Generally static processes are intended for continuously running functionality and dynamic processes are intended for short-lived tasks.

There are five subgroups of static and dynamic processes in total:

- Static central
- Static load shared
- Dynamic load shared
- Dynamic keyed
- DB-keyed

These are called *process categories.* A specification of a process belonging to a process category results in a *process type.* One or more *instances* of a process type are created at run-time. A distinction is made between *logical* and *physical* instances. A physical instance is the actual instance on a processor. A logical instance is a virtual, application level addressable view. A logical process instance can be mapped to several processors with one physical instance on each of these processors sharing the load of the logical instance. A logical instance can be represented by several physical instances over time, for example due to a process crash.

A *static central process* is only instantiated once in the system per process type. It is not load shared because there is only one physical instance at a time and it is addressed by the process name in the process type specification.

A *static load shared process* is only instantiated once in the system per process type and has one physical process instance on every processor specified to be involved in the distribution.

A *dynamic load shared process* is load shared like the static counterpart, but has as many logical instances as the application developer creates.

A *dynamic keyed process* is distributed accorded to a key value, which gives a possibility to co-locate it with database objects. A `keyToDU()` method is used to map a specific process key to a specific processor. The distribution mechanism is described in Section 5.5 on page 30. Dynamic keyed processes are not load shared, thus there is only one physical instance per logical instance at a time. If dynamic keyed process is started with an already existing key, there is a call for the existing process instead of the creation of a new process instance.

A *DB-keyed process* is a special type primarily designed for hardware supervision. It belongs to the static process category and it does not use load sharing. It is distributed according to a database object key value. If the database object is deleted, the process is automatically deleted. The DB-keyed processes are mostly used by TSP itself.

5.3 Interprocess Communication

Processes do not share memory but can communicate over something called dialogs, which is a high level form of interprocess communication. A dialog is a communication protocol between two objects. The two objects are called parties and handle the communication. Each communicating process also has a proxy object to represent the other party. A remote operation is simply done by a method call on the proxy object.

A dialog between two process types must be specified before it can be set up, just like a process type must be specified before a process can be instantiated. A dialog is per definition set up by the initiating party. The other party is the accepting party. Beside the objects that are parties in the dialog, two setup objects are required for initializing the underlying communication protocol. The dialog specification states which operations are possible to invoke on a process, rather than which operations a process can invoke on another process. A remote operation can return a value, but does not need to.

Instantiation of dynamic processes are always initiated by existing processes and never by the system. A dynamic process is created by addressing a non-existent accepting party at dialog setup.

5.4 Database

An integral part of the TSP architecture is a distributed object-oriented real-time database, stored entirely in the primary memory of the processors. Data in a process is considered to be volatile whereas data in the database is considered to be persistent, due to replication and other safety mechanisms.

Data is abstracted by *persistent objects types* (POTs). POTs have attributes that represent the persistently stored data and methods to manipulate the data. Instances of POTs, data objects, are accessed either by a unique primary key or by a reference from an attribute of a data object.

An interesting aspect is that, even if interprocess communication and databases are effectively different concepts, both allow processes to communicate and exchange data.

5.5 Distribution

The distribution goals in a distributed system are to balance the processor load and to minimize the interprocessor communication. The distribution mechanism of TSP is as follows: The application developer groups process instances into a controllable number of *distribution units* (DUs), which in run-time are distributed by TSP to processors. There are two levels of mapping before a process instance is mapped to a distribution unit. The first mapping is from process type to distribution unit type. The process type specification includes a declaration to state which distribution unit type it belongs to, which means that the mapping is determined at compile-time.

The second level of mapping is trivial for all process categories, except for dynamic keyed processes, because in those cases there should only be one distribution unit instance per distribution unit type. A dynamic keyed processes is assigned to a distribution unit type, which has between one and 1024 distribution units. The application developer implements a `keyToDU()` method, which at run-time on basis of the process key determines which distribution unit this process instance should be placed in. The distribution units can be allocated to different processors, but the application developer does not control which processor within a given pool that TSP allocates a particular distribution unit on. The key feature is that by allocating a database object to the same distribution unit type and the same distribution unit as the process, the application developer knows that the process will be co-located with the data without a priori having to know which processor TSP will distribute the distribution unit on.

A site-specific configuration file associates the physical processors with logical distribution pools. A processor can be part of more than one pool. Likewise, distribution unit types are associated with the pools. Within a pool it is up to TSP to map the distribution units containing processes, to the processors in accordance with the distribution goals.

5.6 Program and Development Environment

Programs for TSP are written in C, C++ or Java. But first the processes, dialogs and the persistent object types are defined in the proprietary specification language Delos. Delos generates C/C++ or Java skeleton files. The developer adds application-specific code in these skeleton files. Normally the developer structures the code so that only parts of the application-specific code are contained in the pre-generated files. The rest is written in separate C, C++ or Java files. TSP services are presented through an API, which is further described in [29].

Understanding the associated development environment is central for application development. The development environment runs on Unix and a description is given in [31]. The environment includes tools for configuration, building, debugging and simulation.

5.7 Execution Paradigm

All execution within a process occurs as the execution of serialized callback functions [28]. Execution is triggered by outside events —operations on dialogs are asynchronous— and the programmer must adapt to the inherently asynchronous paradigm. For example, a scenario with an infinite while loop that listens on a socket for messages is a forbidden programming style in TSP, because the method never releases the control of the execution thread. If the execution thread of the process never is released, then no other events scheduled for this process gets execution time. The correct programming style in this case lets the system, asynchronously on incoming messages, invoke a method, which receives and handles the message. By the same reason, blocking calls should be used with great care in order not to starve other calls to the process.

Since all callback functions run serialized, the difficulties with synchronization and consistent states involved in concurrent programming are more or less hidden from TSP application developers. Avoiding concurrent programming and concealing kernel multithreading is a deliberate design choice by the TSP architects, who have valued a forced safe programming practice higher than programmer flexibility and control [28].

Processes can execute on four priority levels: high, normal, low and background. Traffic applications run on normal priority and maintenance on low. Hardware servicing processes can run on high priority, if needed. The background priority level is intended for audits and hardware diagnostic tests. The scheduling policy is as follows : the highest priority process that is ready to execute is allowed to execute for at most one time slice (about two milliseconds) until it becomes blocked, idle, or its time slice expires. If its time slice expires, the process is placed last in the queue of its priority level. This procedure is then repeated with the highest priority process that is ready to execute.

In order to uphold soft real-time performance TSP has a load-regulation mechanism that permits applications to reject parts of the traffic operations. Another feature to sustain real-time performance is an interruptible OS kernel in Dicos. To reduce kernel complexity the number of operations that is allowed to run in the kernel is restricted.

Chapter 6

Prototypes and Tests

This chapter describes the current process model and discusses how the process models presented in Chapter 4 can be applied to *the product*. Finally three models are implemented and tested.

6.1 Current Process Model

The current process model of the product creates one dynamic keyed process per half call. In other words, one process encapsulates one transaction. The INVITE and ACK transactions are counted as one transaction by Ericsson in contrast to the standardized terminology (see discussion about the INVITE and ACK transactions in Section 2.4.1 on page 10). In order to ensure that a new process is created for each request a unique process key must be given in the process instantiation. The current process key is designed to guarantee transaction uniqueness and is a concatenation of Public User ID, `From`, `To`, `Call-ID`, `CSeq`, and something called Port Portion. All fields can be determined based on information contained in a SIP message. The SIP header fields are extracted using a SIP parser. The procedure of calculating the Public User ID depends on several factors such as originating or terminating call, request method name, etc. For a response message, the Public User ID is extracted from the `Via` header. The Public User ID is hashed to a number in the process key.

The process instances are mapped on a distribution unit (DU) in accordance with the distribution mechanism in TSP. The `keyToDU()` method extracts the Public User ID from the process key and takes it modulus the total number of DUs. In short, the same user is always placed in the same DU. Since the database objects related to a specific user also are mapped to a DU based on the Public User ID, a user's database objects and process instances will always be co-located on the same processor. This reasoning presupposes that the designer during the Delos specification phase has mapped the process type (PT) and the database object type (POT) to the same distribution unit type (DUT).

The tasks involved in a transaction are encapsulated in state machine logic. The capsules executing sub-functions are controlled by a mediator function. Some tasks are distributed on static load shared processes. In those cases one logical instance —physically replicated on every processor— takes care of all transactions in the system. These tasks include SIP state monitoring, accounting, configuration, network initiated user requests, as well as other functions. Distribution to other processes implies asynchronous interprocess communication. The typical processing of a half call also involves network communication with other functions, such as the HSS, BGCF, MRFC, and AS (described in Section 3.3 about IMS on page 15). The network communication is asynchronous and possibly between physically differently located nodes. These delays are longer than those caused by asynchronous IPC. Asynchronous delays do not consume CPU time, but they affect the response time of a transaction.

6.2 Process Model Discussion

The current product process model is similar to the forking model described in Section 4.3 on page 21. It is extremely stable because a process crash will only affect one transaction and potential problems of memory leaks are avoided by the short-lived processes. All information that lives longer than a transaction such as dialog states must be stored in the database. The very high degree of persistent storage additionally strengthens the robustness of the solution, but on the other hand it has a performance cost due to database accesses and updates. The forking model is considered obsolete for web servers due to the high initialization costs. It must be stressed that most research assumes the time it takes to start a process in Unix, which is very costly. TSP is designed for rapid process creation [29]. Upgrades are easily performed with short-lived dynamic processes but can be tricky with long-lived dynamic processes. Static processes are always long-lived and therefore during upgrade a new instance is started in parallel with the existing one and given an address reference to the existing instance. Dynamic processes on the other hand are simply asked by the system to terminate within a given time frame. This is not a problem for short-lived dynamic processes because they would anyway naturally terminate within this time. But for long-lived dynamic processes, this means that it needs to store its states to the database before terminating.

The TSP documentation [29] recommends a "resource centric model", which means that all traffic handling covering one "resource" is confined into one dynamic process instance. The motivation for this principle is that the recovery domains should be kept as small as possible. The current process model in the product has a small recovery domain, but I think it would be more natural to model a single user as a resource confined in a process instead of a transaction. Two observations suggest performance improvements by this change. The first observation is that half calls related to one user access partly the same user data. The second observation is that one user performs several transactions during a limited time frame. One call implies at least three transactions. A more advanced user behavior implies more transactions. But the product should be dimensioned for about one million subscribers, and it is not possible to have one million processes allocated in the system at the same time. The problem can be solved by having a timer on each inactive process, which sends a termination signal to the process after time-out. The next time a message arrives the process is re-started and data is fetched from the database. Whether this is a good solution depends on a number of factors. The total available memory and the required memory per process determine the maximum number of processes that is possible to have at a given time. The maximum number of processes in the system at a given time and the traffic model give the maximum lifetime of a process, i.e. the timer value. The traffic model and the value of the timer statistically determine the share of processes that needs to be restarted per transaction. If this value is low, this model could be efficient. If employed, then Public User ID can be used as the process key in a "user centric model". A possible optimization is to have a centralized timer in a static process instead of one timer per process. All user processes register their termination time and the static process notifies when it is time to terminate. The static process can delay a termination if there is enough memory or force an earlier termination if necessary.

TSP does not provide multithreading so any MT model can be discarded immediately for the product. One of the performance strengths of multithreading is the shared heap memory. A user centric process model could still share user data in the heap between different transactions, thus reducing database reads and writes. However, it may still be necessary to write to the database for backup reasons, but the number of reads should at least be decreased.

TSP implements an event-driven execution model, which seems like a good choice with the literature study in server architecture in mind. But unlike SPED and AMPED the currently employed event-driven model is a mixture with MP. Hence, weaknesses from MP such as process management overhead and cleaned caches from context switches are not avoided. When a process only handles one transaction there are no queuing events to handle when the process becomes idle, which leads to a context switch. It may seem strange to employ an event-driven execution model and at the same time encourage a resource centric process encapsulation. It is important to remember that robustness is chosen before performance by the telecom industry if a choice is necessary. The rigorous stability requirements disqualify SPED, MT and any other single process models.

The pooling multi-process model can be implemented as a minor change to the current process model. In this case, the fault isolation domain is unchanged and process initialization costs are saved. The number of processes is fixed. An additional operation in the TSP dialog between the master process and a worker process needs to be defined. Instead of terminating a worker process upon completion the worker process should notify the master process via this dialog operation that it is ready to receive new jobs. There must be one master process per processor and, since a master process must be able to dispatch a job to any DU in order to achieve co-location with database objects, each master process must manage one pool per DU. Hence there are as many pools per DU as there are processors in the cluster. A dynamic keyed process per DU that manages the pool would reduce the number to one pool per DU. The mapping of a job to a process is dynamic within a pool, because any process in the correct pool can handle any assigned job. No modifications related to product functionality are necessary in the product. But this fact also means that there are no potential improvements concerning the implementation of functionality.

A change to user encapsulation and a change to pooling are both potential optimizations. However, it is complex to combine pooling and user encapsulation. User encapsulation implies that a process does not return to the pool before it has timed-out. Half calls related to a user with an active process can not be directed to any process in the pool. Instead it must be dispatched to the correct active process. Moreover, it can be discussed whether these two changes are enough for a dramatic impact. A solution that still takes advantage of these benefits, but also enables other optimizations is a model with multiple users per process and multiple processes (MUMP). Mapping all users to one process is not possible due to stability reasons, but a fine-tuned number of users per process can meet both robustness and performance requirements. As in the pooling model, the number of processes is fixed and processes are long-lived. Unlike the pooling model, the mapping of a user to a process is static. In MUMP a Public User ID is always mapped to the same process key (and a process key is always mapped to the same DU). Because the number of users in the system is greater than the number of processes, several Public User IDs map to the same process key. The process has a data structure, e.g. a hashmap, to distinguish between different user objects. The initialization costs are removed as in the pooling model and more concurrent jobs per process lead to fewer context switches. MUMP does not scale in the sense that a subscriber increase leads to an unchanged number of users per process. The number of processes can be a configuration parameter. However, memory consumption is likely to be an issue if user and state information is cached. Suppose there are X processes and each process maximally is assigned M bytes. Furthermore, assume that there are Y users per process and that the amount of memory Y users need if they cache registration state and dialog state is more than M bytes. If the process allocates more than M bytes the process stay at this size because dynamic memory management functions do rarely give back memory to the OS before the process terminates and these processes are long-lived. Hence, there must either be a mechanism that makes sure that the limit M never is exceeded or a mechanism that releases memory to the OS. Having such mechanisms adds complexity.

The staged event-driven model can, as stated previously, be implemented with all stages in one process or with one stage per process. The first method does not meet the robustness requirements and the second method increases latency due to IPC. However, SEDA introduces some interesting ideas on localization and it might be possible to increase performance by sending a batch of jobs over a dialog at the same time instead of one by one. The question is whether there are jobs to the same process frequently enough to win time on buffering.

6.3 Prototypes

On basis of the background study and the discussion above I choose to implement prototypes of three models for measurements and comparison. The three models I chose to implement were a forking MP model, a SPED model and a pooling MP model. The prototypes are designed for one traffic processor on TSP, but all prototypes can be extended to handle a cluster.

The reason to why I chose to implement a forking MP model was because it resembles the current process model and in order to recommend any changes I must be able to measure it against the standard. The implemented model is an extreme case because it creates a new dynamic process for each message. The prototype is hereafter referred to as the *dynamic prototype*.

I implemented a SPED model because it is the contrary case to the dynamic prototype in regard to the number of processes. The prototype dispatches all messages to the same process. The prototype is hereafter referred to as the *asynchronous prototype* because it only relies on asynchrony for concurrency.

I chose to implement a pooling MP model because it improves the currently employed model without modifying the logic. The prototype is hereafter referred to as the *pooling prototype*.

The reason to why I chose these models in favor of MUMP and AMPED was because the benefits are more clearly observable in these extreme cases. Observation of benefits arising from implementation of functionality requires a full implementation of the product, which is beyond the scope of this thesis. Hence, the prototypes are restricted to the process models and they do not implement a SIP parser nor the extensive logic related to the product. However, as stated previously, combinatory models like MUMP and AMPED are interesting when considering a real implementation. The SEDA model was rejected, because it is too dissimilar from the current solution to motivate the implementation costs.

The target environment for the prototypes is the Dicos OS. The prototypes are developed in the TSP development environment on Linux mentioned in Section 5.6. The processes and dialogs are specified in Delos, the TSP definition language, and all other code is written in C++. All prototypes use a static central process as master process. The worker process(es) are dynamic keyed processes. The process keys are simply a number starting from zero. The master process receives incoming messages and a worker process forwards the message to the recipient. There is a dialog between the master process and each worker process and it is used by the master process to dispatch an incoming message to a worker process. In all models the server specific behavior should be performed in the worker process, but as stated previously these prototypes do not perform this step. All three prototypes receive and send SIP messages encapsulated in UDP datagrams. The code related to the socket management notifies the master process asynchronously about incoming messages.

The dynamic prototype creates one dynamic keyed process and dialog per message. The dialog and process are shutdown after completion. The pooling prototype creates 100 dynamic keyed processes and dialogs initially and puts process identifiers in a ready queue. The master processes picks worker processes from the queue and the worker processes subsequently signals via the dialog to be put back in the ready queue. The asynchronous prototype creates one dialog and dynamic keyed process initially to which all messages are dispatched.

6.4 Tests

The goals of the tests are to quantify the process management costs in order to evaluate differences between the models. The first test ensures the correct functionality of the prototypes. The second test measures capacity affecting factors. The third test measures latency originating from the prototypes.

6.4.1 Test model

The test model is as follows: A half call model is employed, which means that there is only one proxy server instead of one per user agent as in the full call model. The test model employs three nodes in total: one user agent client (UAC), one user agent server (UAS) and one proxy server (the prototype). A (half) call consists of one INVITE transaction, one ACK transaction and one BYE transaction. No provisional responses are sent and in total one call corresponds to five messages. The UAC initiates all transactions. There is a holding time of 2000 ms between the ACK transaction and the BYE transaction. The test model is illustrated in Figure 6.1.

The traffic generator *SIPp* (release 1.0) [42] acts as UAC and UAS. SIPp is started with two instances on the same node so the UAC and the UAS have the same IP address. The network analyzer tool *Ethereal* (release 0.10.12) [43] is used to capture SIP communication. The prototypes are employed on TSP 5.1 - MCP 6 with one traffic processor activated. The program TelORB Manager is used for TSP management, abortion identification and surveillance of memory usage. The program TelORB Viewer shows CPU usage per processor. I used the commands *pts*, *ps* (process status) and *cs* (capsule status) for collecting measurement information.

Before progressing with the tests, let us more carefully define a couple of terms. *Load* is defined as the number of calls per second a server is burdened with from an external point of view. Load does not take a server's capability into consideration. *Capacity* is the maximum load a server is capable of handling. *Throughput* is the number of handled calls that a server outputs per second. The throughput is less than or equal to the capacity of the system. When being in a stable state, the throughput equals the load. Load, capacity and throughput can also be defined in terms of SIP messages per second instead of (half) calls per second. Since a half call corresponds to a deterministic number of SIP messages in this test model, it is only a matter of measurement unit.

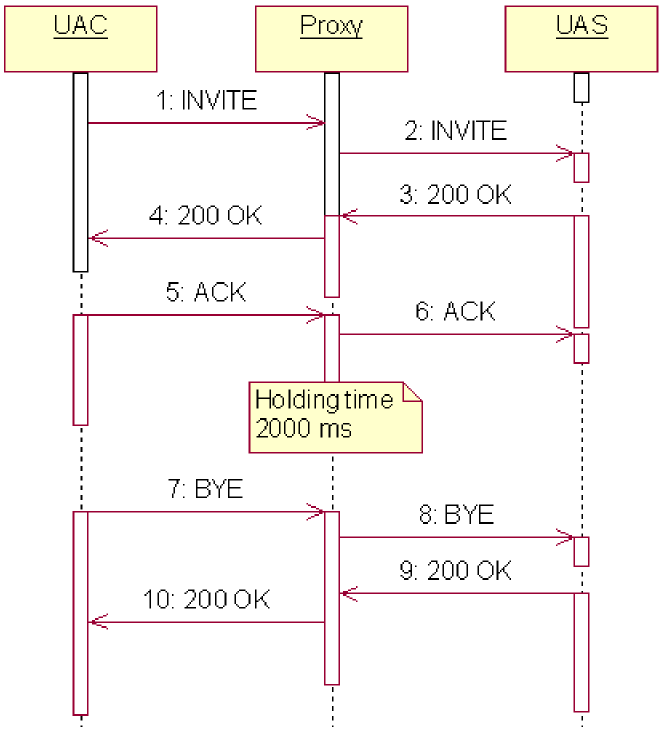

Figure 6.1. Test model

6.4.2 Test 1: Functionality

The purpose of the functionality test is to ensure that the test environment is correctly set up and that the prototypes work properly. The goal of this test is to perform one successful half call, i.e. five messages. No measurements are performed. Figure 6.2 shows the port configuration used for this (and the following) tests. The prototypes send messages from an ephemeral port. SIPp sends from the same port it is listening to. The traffic generated by one half call is captured by Ethereal and presented in Appendix C.

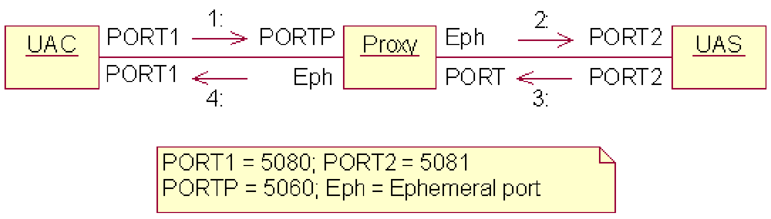
UAC
PORT1
1:
PORTP
Proxy
Eph
2:
PORT2
UAS
PORT1
4:
Eph
PORT
3:
PORT2
PORT1 = 5080; PORT2 = 5081
PORTP = 5060; Eph = Ephemeral port

Figure 6.2. Port configuration

6.4.3 Test 2: Capacity

There are several factors that potentially limit the capacity of a server, for example lack of bandwidth, memory or CPU capacity. Compliance with quality specifications might also reduce server capacity. For example, if a certain load leads to latency above a specified maximum value, then the maximum allowed load must be restricted to a value less than this load. If IP packets are dropped or the rate of incoming messages are greater than the rate of outgoing (proxied) messages then obviously the load is above maximum capacity. A server that gracefully handles overload rejects some fraction of the incoming calls in order to equalize the rate of accepted incoming messages with the rate of outgoing messages. Preferably new calls, i.e. INVITE requests, are rejected and ongoing calls are prioritized.

The goals of these capacity tests were to measure

- how CPU usage varies as a function of call rate for each prototype
- the CPU processing time of a call for each prototype

The less CPU processing time per call and the less CPU usage required for solving a task, the better server performance. I did not do any memory measurements, because the prototypes did not store any user data or state information.

CPU utilization can be measured as percentage of CPU capacity. Closely correlated to CPU utilization is CPU load, which can be measured in form of a scalar, representing the number of running and runnable processes during a given time period (i.e. the sum of the run queue length and the number of currently running processes). Operating systems providing this scalar value usually denote it *load average*. While the optimal CPU usage is 100%, the optimal load average is 1. If the CPU load is 100%, then no CPU time is wasted. If the load average is less than 1, then no processes are stalled waiting for CPU time. Stalled processes in ready state affect performance negatively, but they are not the only waiting state affecting the performance of a system. Processes that await I/O, i.e. processes in "sleeping" state or "blocked" state, potentially affect performance too. To distinguish these two bottlenecks, a process whose performance to complete a computation principally is determined by the speed of I/O operations to memory is generally referred to as I/O bound. A process whose performance principally is determined by the speed of the CPU is generally referred to as CPU bound.

I measured the CPU usage for each prototype by starting with a call rate of 100 half calls per second and noting the value in TelORB Viewer. Then I increased the half call rate by 100 and repeated the procedure until the half call rate reached 1000. In the dynamic prototype the CPU usage increased by approximately 5 percentage points per increased hundred in half call rate. The corresponding value for the pooling prototype and the asynchronous prototype was 1 percentage point (see Figure 6.3). The number of concurrent calls increased and stabilized at a higher value every time I increased the call rate. But at 500 calls per second and beyond, the increase was indefinite.

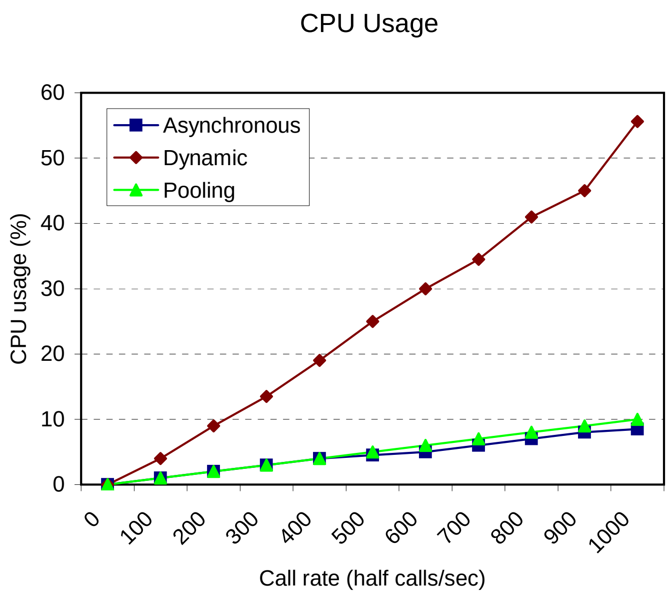

Figure 6.3. CPU usage as a function of call rate

Since the CPU computation time per job is independent of call rate (the linear curve in Figure 6.3 confirms this theoretical anticipation), I only systematically measured the computation time at 300 half calls per second for each prototype. The calculations use Equation 6.1.

$$\text{CPU time per call} = \frac{\text{CPU time}}{\text{Number of calls}} \tag{6.1}$$

I measured CPU time for approximately 500,000 calls. The pts command provided the accumulated CPU time of the two process types associated with the prototypes. The arithmetic mean values are displayed in Table 6.1, while complete test data is presented in Appendix D. Table 6.1 is analyzed in the next chapter.

Prototype	CPU time (ms)
Asynchronous	0.10
Dynamic	0.39
Pooling	0.11

Table 6.1. CPU time per call: The arithmetic mean value measured in millisec per half call.

6.4.4 Test 3: Latency

The goal of the third test is to empirically identify any variations in latency between the process models. Latency in this context is the difference between the time the proxy receives a SIP message and the time it sends that SIP message. In order words, the interesting value is the contribution to latency by the proxy server and not the total round-trip time.

I measured latency using the time stamps provided by Ethereal, which shows when messages are received and sent. After generating around 10,000 calls per call rate for each prototype, I analyzed the Ethereal time stamps using Product Latency Analyzing Utility v3.0, a proprietary program by Ericsson. Product Latency Analyzing Utility v3.0 calculates various values such as mean values and medians. Figure 6.4 shows the 75th percentile values (remember from Chapter 5 about TSP that the telecommunication industry uses statistical goals). Figure 6.5 shows the 75th percentile values but only for the INVITE requests. Figure 6.6 shows latency values for the 75th percentile but only for the BYE requests. Complete test results are presented in Appendix E.

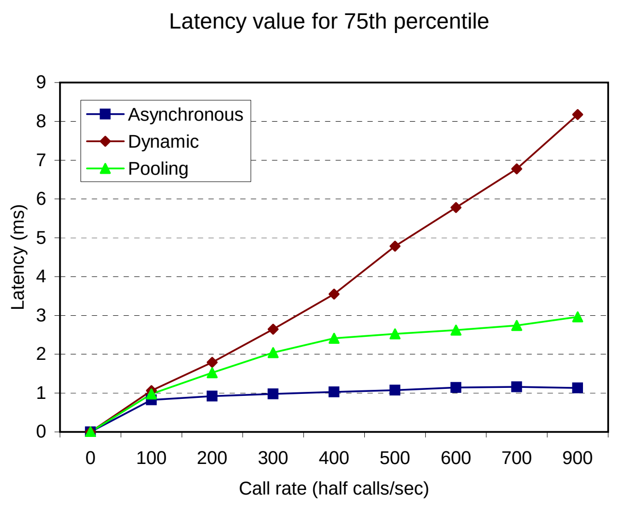

Figure 6.4. Latency for the 75th percentile of all SIP messages

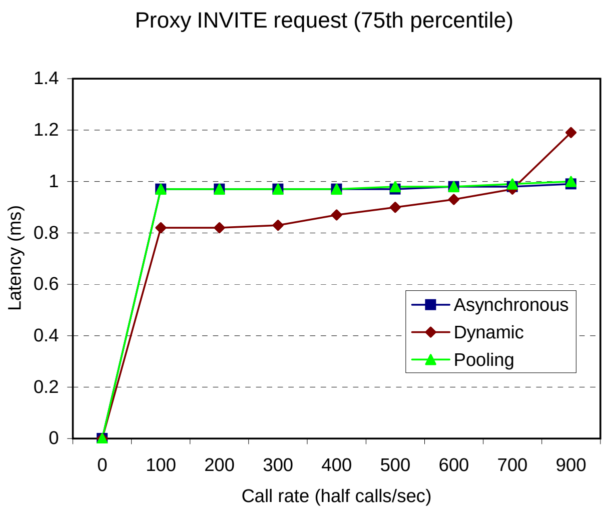

Figure 6.5. Latency for the 75th percentile of all INVITE requests

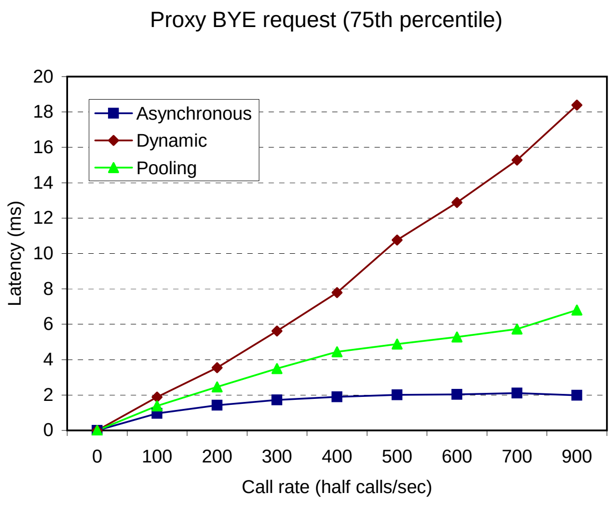

Figure 6.6. Latency for the 75th percentile of all BYE requests

Chapter 7

Test Analysis

This chapter analyzes and evaluates the tests described and the diagrams presented in Section 6.4. All data from the measurements is enclosed in Appendix C to Appendix E.

7.1 Functionality Test

Everything worked correctly and expectedly in the functionality test. Note in the printouts in Appendix C that the proxy did not modify the SIP messages as a complete SIP proxy product would do (in for example the `Via` and `Record-Route` headers).

7.2 Capacity Test

The first question raised by the capacity tests is why we lost packets at call rates greater than 400 half calls per second. To understand whether this was due to deficiencies in the proxy I analyzed a case with 500 generated half calls per second. Totally 10,000 calls were initiated. If no packets were lost this should imply 50,000 SIP messages and 100,000 UDP datagrams. The actual number of UDP datagrams that went in and out from proxy server was 99,490 datagrams based on Ethereal data. The UAS received 10,000 INVITE requests (see Figure 7.1), hence there were no losses there. The UAC received 10,000 200 OK responses, hence there were neither no losses there. But the UAS only received 9,982 of 10,000 ACK request, hence there was a packet loss somewhere. The UAS only received 9,745 of 10,000 BYE requests and the UAC received all 9,745 200 OK responses. In total, we lost 273 packets. Did we lose them before, inside or after the proxy node? If we lost them after the proxy then the total number messages into and out from the proxy should be $10,000 * 2 * 4 + 9,745 * 2 = 99,490$. This number is exactly the same number as the verified number of packets that actually was captured by Ethereal. Hence, no packets were rejected by the proxy. The messages were lost after the proxy and probably the packets were dropped by the UAS's network interface card

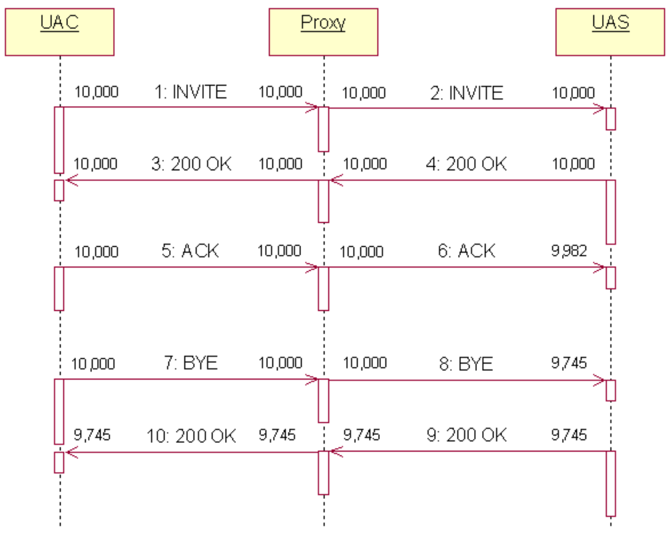

Figure 7.1. Illustration of where packets were lost

(NIC). If I had had more client computers I could probably have pushed the call rate further.

The computation time per job of the pooling prototype and the asynchronous prototype were approximately equal (see Table 6.1). The asynchronous prototype performed slightly better. The relative gain between the asynchronous prototype and the dynamic prototype was large, about 75% less computation time for the asynchronous prototype. However, the gain was only about 0.3 ms in absolute terms.

When not taking potential improvements associated with functionality into account, the improvement with long-lived processes is not good enough to alone motivate a redesign of the product. The overall computation time goal is 15 ms and satisfactory improvements are around 0.5 ms. On the other hand, there are some potential optimizations in the product implementation using long-lived processes, for example reducing the number of database transactions. If resources are not insufficient, dialogs to static helper processes can be open all the time, additionally improving performance.

7.3 Latency Test

The latency test gave some strange results. The prototypes were designed to treat all messages exactly equally, but still there were big discrepancies between INVITE transactions and BYE transactions, especially for the dynamic prototype. If we instead of percentile values look at mean values and standard deviations (see Appendix E), then we can see that the differences can not alone be explained by random errors and bad precision. According to the Ethereal FAQ [44], "Ethereal gets time stamps from libpcap/WinPcap, and libpcap/WinPcap get them from the OS kernel, so Ethereal is at the mercy of the time stamping code in the OS for time stamps". Furthermore, the FAQ states that Intel x86 processors, starting from the Pentium Pro, have a Time Stamp Counter (TSC) register from which the OS kernel get high-precision time stamps (1 μs resolution). The precision of the time stamps can not be the reason to the discrepancies.

But how is the accuracy? Are there any systematic errors in the measurements? All measurements are performed exactly the same, so there are no apparent explanations for having a systematic error in one measurement but not in another measurement. According to the preceding citation from the Ethereal FAQ, the packets were time stamped by the OS and not by the NIC. Hence, the NIC had to deliver and notify the CPU about a packet before the measurement started. This means that there could have been buffering behaviors that disturbed the accuracy. However, these delays should have been independent of process model. Remember that we only were interested in measuring the delay inside the proxy. This value should anyway be applicable for relative comparison as long as all prototypes were measured with the same start and end points. Buffering after the start point can be more troublesome in terms of individual values, but the measuring method, which took averages of multiple samples, should weigh up that.

After some consideration, the probable explanation to this discrepancy phenomenon is that the INVITE messages were sent continuously by the user agent program SIPp, while it apparently buffered the BYE requests and sent it in one chunk. The latter mode gave greater latency values because the messages arrived in one chunk and had to be queued.

To sum up, these measurements are most likely correct, but it could be interesting to measure all messages with the continuous sending mode or, preferably, by using many independent user agents in order to get a better message distribution.

Latency, which is the sum of the queuing time and the computation time, showed quite constant values for the asynchronous prototype. The latency of the dynamic prototype increased linearly and the latency of the pooling prototype was somewhere in between. While having in mind that an unrealistic message distribution might have disturbed the results, the same method always measured significant differences between the prototypes. The latency results speak for the asynchronous prototype or the pooling prototype.

7.4 Conclusions

The results from the tests show that a process model that avoids process creations and terminations performs better, but not drastically better in absolute numbers. As previously mentioned in Section 6.2, this feature can partly be achieved by having one user per process with timer and entirely by implementing pooling of the current transaction based solution or by implementing MUMP. The first two solutions are quite simple. Pooling leads to the fewest number of code changes, but it does not lead to any other potential improvements. User based encapsulation results in clean and nice modeling, and is therefore recommendable. MUMP has the greatest performance potential, but it must be carefully modeled and dimensioned. There is a risk of having to introduce mechanisms for controlling various issues such as memory consumption, leading to performance costs and degraded robustness due to complex code.

Chapter 8

Conclusion

This chapter summarizes conclusions from the Master's thesis. It gives a recommendation on appropriate ways forward for *the product* and points out some future work.

8.1 Summary

Knowledge about how process models differ in terms of performance, memory usage, robustness and complexity from a platform perspective is important when designing the architecture. However, the process model is only a part of the full server architecture and advantages and disadvantages have to be put in relation to the overall picture. Only a complete design can take all aspects into account. The following list summarizes important conclusions about process models for high performance telecom system based on TSP:

- Usage of multiple worker processes is the only way to achieve an inherently stable process model.
- Event-driven process models with asynchronous I/O operations are typically very efficient.
- The test results show that process models that avoid TSP process startup and termination as well as TSP dialog setup and shutdown perform better than process models that create new processes. The gain is approximately 0.3 ms in computation time per half call, assuming the comparison model creates a new process per SIP message. This improvement does not take any other optimizations enabled by long-lived processes into account.
- The test results show that process models with few processes and no startup time reduce latency considerably when many messages arrive at the same time.
- Long-lived processes affect memory consumption even when they are idle, since the memory never is released by the process. Assuming each worker

process require approximately the same maximum amount of memory and that they never release any memory to the OS, the number of worker processes must be less than the available memory for worker processes divided by the maximum amount of required memory per worker process.

- Process models with short-lived processes tend to be difficult to optimize in performance, since data is short-lived and the amount of initializations is difficult to reduce. However, short-lived processes typically yield less complex process models than long-lived processes.

A design of the product should take the following conclusions into account:

- Process models with multithreading are not realizable, because TSP does not support an interface to kernel threads at application level.
- A multi-process model must be employed in order to fulfill the robustness requirements of the product.
- The multiple worker processes should be designed as TSP dynamic keyed processes.
- An event-driven model must be employed, because TSP's execution paradigm is event-driven.
- A user centric process encapsulation is more natural than a transaction centric encapsulation from a modeling perspective.
- The current process model enforces database backups at transaction level, but a user level process model can also implement database writes at transaction level.
- A transaction centric process encapsulation leads to short-lived processes, if the processes are not reused by the pooling method.
- A process model that avoids process creations/terminations and dialog setups/shutdowns performs better, however not drastically better at low call rates.

Pooling is simple to add to the current transaction centric solution. It does not lead to any other potential improvements, but it is the most cost-efficient way to implement long-lived processes, since it requires only minor modifications of already fully functioning code.

The model with multiple users per process has the best potential, but each process must limit its memory consumption, leading either to complex code and overhead costs or to little space for caching of state and user data. The memory resources required for the product must be calculated, given its functionalities and traffic model, before it can be considered realizable.

One user per process is simpler than many users per process, but leads to more changes to the current solution than transaction centric pooling. From a memory perspective, the model can easily terminate inactive processes if memory consumption is too high. A negative characteristic is that the model must terminate its processes immediately if the set of active users is greater than the set of processes that fits into the memory. The process lifetime is the shortest during busy hours when you want the performance to be at its best. Section future work briefly outlines what data that is needed for estimating the average process lifetime value during busy hour.

Transaction centric pooling as well as multiple users per process can eliminate process startup and termination costs. The one user per process model can partly accomplish this feature. The best process model of these three to proceed with for the product is a decision between acceptable implementation costs and potential performance benefits. If implementation time is valued high, pooling is the better alternative. Otherwise the feasibility of the two user centric models should be calculated from a memory perspective before any of these models are considered appropriate for the product. Such a calculation must take product functionality and traffic models into account, which is beyond the scope of this thesis. However, if there is enough memory available, the performance potential is greater for these two models.

8.2 Future Work

The rate of process creations that can be avoided in the one user per process model can be calculated if a traffic model is assumed, a TSP system is assumed and the memory consumption per process is estimated. A traffic model typically includes the total number of users, the user penetration per service, the usage rate for a service per user during busy hour, the average number of transactions per second generated when using a specified service and so on. One must also statistically decide how the transactions are related to the user set, because the efficiency of the model depends on rate of transactions that originate from a user with an already allocated process in the system. Such calculations as well as other memory calculations that involve product functionality are left as future work.

Bibliography

[1] *IP Multimedia Subsystem (IMS); Stage 2*, 3GPP TS 23.228, 2005

[2] B. A. Forouzan, *Data Communications and Networking*, 3rd edition, McGraw-Hill, New York, USA, 2002

[3] J. Rosenberg et al., *SIP: Session Initiation Protocol*, RFC 3261, IETF, 2002

[4] M. Handley and V. Jacobson, *SDP: Session Description Protocol*, RFC 2327, IETF, 1998

[5] T. Berners-Lee et al., *Uniform Resource Identifiers (URI): Generic Syntax*, RFC 2396, IETF, 1998

[6] A. Vaha-Sipila, *URLs for Telephone Calls*, RFC 2806, IETF, 2000

[7] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, RFC 2045, IETF, 1996

[8] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, RFC 2046, IETF, 1996

[9] K. Moore, *Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text*, RFC 2047, IETF, 1996

[10] N. Freed, J. Klensin and J. Postel, *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*, RFC 2048, IETF, 1996

[11] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*, RFC 2049, IETF, 1996

[12] P. Hoffman, L. Masinter and J. Zawinski, *The mailto URL scheme*, RFC 2368, IETF, 1998

[13] S. Donovan, *The SIP INFO Method*, RFC 2976, IETF, 2000

[14] J. Rosenberg and H. Schulzrinne, *Reliability of Provisional Responses in the Session Initiation Protocol (SIP)*, RFC 3262, IETF, 2002

[15] A. B. Roach, *Session Initiation Protocol (SIP)-Specific Event Notification*, RFC 3265, IETF, 2002

[16] J. Rosenberg, *The Session Initiation Protocol (SIP) UPDATE Method*, RFC 3311, IETF, 2002

[17] B. Campbell et al., *Session Initiation Protocol (SIP) Extension for Instant Messaging*, RFC 3428, IETF, 2002

[18] R. Sparks, *The Session Initiation Protocol (SIP) Refer Method*, RFC 3515, IETF, 2003

[19] A. Niemi, *Session Initiation Protocol (SIP) Extension for Event State Publication*, RFC 3903, IETF, 2004

[20] P. Calhoun et al., *Diameter Base Protocol*, RFC 3588, IETF, 2003

[21] C. Rigney et al., *Remote Authentication Dial In User Service (RADIUS)*, RFC 2865, IETF, 2000

[22] *ITU-T Recommendation H.248*, ITU, 2000

[23] H. Schulzrinne et al., *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550, IETF, 2003

[24] D. Durham et al., *The COPS (Common Open Policy Service Protocol*, RFC 2748, IETF, 2000

[25] T. Berners-Lee and D. Connolly, *Hypertext Markup Language — 2.0*, RFC 1866, IETF, 1995

[26] A. S. Tannenbaum, *Modern Operating Systems*, 2nd edition, Prentice Hall, New Jersey, USA, 2001

[27] G. Camarillo and M. A. García-Martín, *The 3G IP Multimedia Subsystem (IMS)*, John Wiley & Sons, England, 2004

[28] L. Hennert and A. Larruy, *TelORB - The distributed communications operating system*, Ericsson Review No. 3, 1999, http://www.telorb.com (2005-10-13)

[29] *TSP Application Development and Troubleshooting Environment (TADE) 5-MCP 5100.6.* Ericsson AB, 2005

[30] *TSP Overview*, Global Services, Ericsson AB, 2003

[31] *TSP Software Implementation*, Ericsson AB, 2002

[32] A. Luotonen, *Web proxy servers*, Prentice Hall, New Jersey, USA, 1998

[33] J. R. Larus and M. Parkes, *Using cohort scheduling to enhance server performance*, Technical Report MSR-TR-2001-39, Microsoft Research, 2001

[34] V. S. Pai, P. Druchsel, and W. Zwaenepoel, *Flash: An Efficient and Portable Web Server*, Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, California, 1999

[35] M. Welsh, D. Culler, and E. Brewer, *SEDA: An Architecture for Well-Conditioned Scalable Internet Services*, Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Alberta, Canada, 2001

[36] *The Apache web server*, Apache Software Foundation, http://www.apache.org (2005-11-13)

[37] *Netscape Enterprise Server*, Netscape Corporation, http://home.netscape.com/enterprise/v3.6/index.html (2005-11-13)

[38] *Zeus Web Server*, Zeus Technology, http://www.zeus.co.uk (2005-11-13)

[39] *Internet Engineering Task Force*, http://www.ietf.org (2005-11-13)

[40] *3rd Generation Partnership Project*, http://www.3gpp.org/ (2005-11-13)

[41] *2 miljarder mobilanvändare i världen*, 2005-09-17, TT, Svenska Dagbladet, http://www.svd.se/dynamiskt/naringsliv/did_10563620.asp (2005-09-18)

[42] *SIPp*, Hewlett Packard, http://sipp.sourceforge.net/ (2005-11-01)

[43] *Ethereal*, http://www.ethereal.com (2005-11-01)

[44] *Ethereal FAQ*, http://www.ethereal.com/faq.html (2005-11-08)

Appendix A

Abbreviations

2G	Second Generation of Mobile Systems
3G	Third Generation of Mobile Systems
3GPP	Third Generation Partnership Project
AAA	Authentication, Authorization, and Accounting
ADSL	Asymmetric Digital Subscriber Line
AMPED	Asymmetric Multi-Process Event-Driven Server Architecture
ANSI	American National Standards Institute
AS	Application Server
B2BUA	Back-to-Back User Agent
BGCF	Breakout Gateway Control Function
COPS	Common Open Policy Service
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSCF	Call/Session Control Function
DNS	Domain Name System
DU	Distribution Unit
DUT	Distribution Unit Type
FAQ	Frequently Asked Questions
GSM	Global System for Mobile Communications
HSS	Home Subscriber Server
HTTP	Hypertext Transfer Protocol
I-CSCF	Interrogating-Call/Session Control Function
I/O	Input/Output
ID	Identity
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IP	Internet Protocol

IPC	Interprocess Communication
ISP	Internet Service Provider
ITU	International Telecommunication Union
L2	Level-2 CPU Cache
MGCF	Media Gateway Control Function
MGW	Media Gateway
MIME	Multipurpose Internet Mail Extensions
MP	Multi-Process Server Architecture
MRF	Media Resource Function
MRFC	MRF Controller
MRFP	MRF Processor
MT	Multi-Thread Server Architecture
MUMP	Multiple Users Per Process Multi-Process Server Architecture
NCSA	National Center for Supercomputing Applications
NIC	Network Interface Card
O&M	Operation and Maintenance
OS	Operating System
P-CSCF	Proxy-Call/Session Control Function
POT	Persistent Object Type
PSTN	Public Switched Telephone Network
PSTN/CS	Public Switched Telephone Network/Circuit-Switched Gateway
PT	Process Type
QoS	Quality of Service
RFC	Request for Comments
RTP	Real-Time Transport Protocol
RTPC	RTP Control Protocol
S-CSCF	Serving-Call/Session Control Function
SEDA	Staged Event-Driven Server Architecture
SDP	Session Description Protocol
SGW	Signaling Gateway
SIP	Session Initiation Protocol
SLF	Subscriber Location Function
SMS	Small Message Service
SMTP	Simple Mail Transfer Protocol
SPED	Single-Process Event-Driven Server Architecture
SPS	Single-Process Serialized Server Architecture
SS7	Signaling System no. 7
TCP	Transmission Control Protocol
TEL URL	Telephone URL

TLB	Translation Lookaside Buffer
TSC	Time Stamp Counter
TSP	Telecom Server Platform
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
UE	User Equipment
URI	Universal Resource Identifier
URL	Universal Resource Locator
VoIP	Voice over IP
WLAN	Wireless Local Area Network

Appendix B

SIP Headers

SIP headers			
Accept	Content-encoding	Max-forwards	Route
Accept-encoding	Content-language	MIME-version	Server
Accept-language	Content-length	Organization	Subject
Alert-info	Content-type	Priority	Supported
Allow	CSeq	Proxy-authenticate	Timestamp
Also	Date	Proxy-authorization	To
Authorization	Encryption	Proxy-require	Unsupported
Call-ID	Error-info	Record-route	User-agent
Call-info	Expires	Require	Via
Contact	From	Response-key	Warning
Content-disposition	In-reply-to	WWW-authenticate	Retry-after

Table B.1. SIP headers

Appendix C

Test data: Functionality

SIP data generated by one half call. A half call corresponds in the test model to an INVITE transaction, an ACK transaction and a BYE transaction between a UAC, a proxy server and a UAS.

```
Frame 1 (605 bytes on wire, 605 bytes captured)
Ethernet II, Src: 00:04:96:15:22:30, Dst: 00:20:ce:c9:ad:0c
Internet Protocol, Src Addr: 10.50.99.200 (10.50.99.200),
Dst Addr: 10.50.100.99 (10.50.100.99)
User Datagram Protocol, Src Port: 5080, Dst Port: 5060
Session Initiation Protocol
    Request-Line: INVITE sip:sipphone10001@cscf99.lab SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-0
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 INVITE
        Contact:  <sip:sipphone10001@10.50.99.200:5062>
        Max-Forwards: 70
        Subject: Performance Test
        Content-Type: application/sdp
        Content-Length:  187
    Message body
```

Figure C.1. INVITE request from UAC to Proxy

```
Frame 2 (605 bytes on wire, 605 bytes captured)
Ethernet II, Src: 00:20:ce:c9:ac:90, Dst: 00:04:96:15:22:30
Internet Protocol, Src Addr: 10.50.100.99 (10.50.100.99),
Dst Addr: 10.50.99.200 (10.50.99.200)
User Datagram Protocol, Src Port: 65480, Dst Port: 5081
Session Initiation Protocol
    Request-Line: INVITE sip:sipphone10001@cscf99.lab SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-0
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 INVITE
        Contact:  <sip:sipphone10001@10.50.99.200:5062>
        Max-Forwards: 70
        Subject: Performance Test
        Content-Type: application/sdp
        Content-Length:  187
    Message body
```

Figure C.2. INVITE request from Proxy to UAS

```
Frame 3 (334 bytes on wire, 334 bytes captured)
Ethernet II, Src: 00:04:96:15:22:30, Dst: 00:20:ce:c9:ad:0c
Internet Protocol, Src Addr: 10.50.99.200 (10.50.99.200),
Dst Addr: 10.50.100.99 (10.50.100.99)
User Datagram Protocol, Src Port: 5081, Dst Port: 5060
Session Initiation Protocol
    Status-Line: SIP/2.0 200 OK
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-0
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 INVITE
        Contact: <sip:sipphone10001@10.50.99.200:5081>
        Content-Type: application/sdp
```

Figure C.3. 200 OK response from UAS to Proxy

```
Frame 4 (334 bytes on wire, 334 bytes captured)
Ethernet II, Src: 00:20:ce:c9:ad:0c, Dst: 00:04:96:15:22:30
Internet Protocol, Src Addr: 10.50.100.99 (10.50.100.99),
Dst Addr: 10.50.99.200 (10.50.99.200)
User Datagram Protocol, Src Port: 65479, Dst Port: 5080
Session Initiation Protocol
    Status-Line: SIP/2.0 200 OK
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-0
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 INVITE
        Contact: <sip:sipphone10001@10.50.99.200:5081>
        Content-Type: application/sdp
```

Figure C.4. 200 OK response from Proxy to UAC

```
Frame 5 (371 bytes on wire, 371 bytes captured)
Ethernet II, Src: 00:04:96:15:22:30, Dst: 00:20:ce:c9:ad:0c
Internet Protocol, Src Addr: 10.50.99.200 (10.50.99.200),
Dst Addr: 10.50.100.99 (10.50.100.99)
User Datagram Protocol, Src Port: 5080, Dst Port: 5060
Session Initiation Protocol
    Request-Line: ACK sip:sipphone10001@10.50.99.200:5081 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-9
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 ACK
        Max-Forwards: 70
        Route:  <sip:sipphone10001@10.50.99.200:5081>
        Content-Length: 0
```

Figure C.5. ACK request from UAC to Proxy

```
Frame 6 (371 bytes on wire, 371 bytes captured)
Ethernet II, Src: 00:20:ce:c9:ac:90, Dst: 00:04:96:15:22:30
Internet Protocol, Src Addr: 10.50.100.99 (10.50.100.99),
Dst Addr: 10.50.99.200 (10.50.99.200)
User Datagram Protocol, Src Port: 65478, Dst Port: 5081
Session Initiation Protocol
    Request-Line: ACK sip:sipphone10001@10.50.99.200:5081 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-9
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 5 ACK
        Max-Forwards: 70
        Route:  <sip:sipphone10001@10.50.99.200:5081>
        Content-Length: 0
    Message body
```

Figure C.6. ACK request from Proxy to UAS

```
Frame 7 (372 bytes on wire, 372 bytes captured)
Ethernet II, Src: 00:04:96:15:22:30, Dst: 00:20:ce:c9:ad:0c
Internet Protocol, Src Addr: 10.50.99.200 (10.50.99.200),
Dst Addr: 10.50.100.99 (10.50.100.99)
User Datagram Protocol, Src Port: 5080, Dst Port: 5060
Session Initiation Protocol
    Request-Line: BYE sip:sipphone10001@10.50.99.200:5081 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-12
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 6 BYE
        Max-Forwards: 70
        Route:  <sip:sipphone10001@10.50.99.200:5081>
        Content-Length: 0
```

Figure C.7. BYE request from UAC to Proxy

```
Frame 8 (372 bytes on wire, 372 bytes captured)
Ethernet II, Src: 00:20:ce:c9:ad:0c, Dst: 00:04:96:15:22:30
Internet Protocol, Src Addr: 10.50.100.99 (10.50.100.99),
Dst Addr: 10.50.99.200 (10.50.99.200)
User Datagram Protocol, Src Port: 65477, Dst Port: 5081
Session Initiation Protocol
    Request-Line: BYE sip:sipphone10001@10.50.99.200:5081 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-12
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 6 BYE
        Max-Forwards: 70
        Route:  <sip:sipphone10001@10.50.99.200:5081>
        Content-Length: 0
    Message body
```

Figure C.8. BYE request from Proxy to UAS

```
Frame 9 (352 bytes on wire, 352 bytes captured)
Ethernet II, Src: 00:04:96:15:22:30, Dst: 00:20:ce:c9:ad:0c
Internet Protocol, Src Addr: 10.50.99.200 (10.50.99.200),
Dst Addr: 10.50.100.99 (10.50.100.99)
User Datagram Protocol, Src Port: 5081, Dst Port: 5060
Session Initiation Protocol
    Status-Line: SIP/2.0 200 OK
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-12
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 6 BYE
    Message body
```

Figure C.9. 200 OK response from UAS to Proxy

```
Frame 10 (352 bytes on wire, 352 bytes captured)
Ethernet II, Src: 00:20:ce:c9:ac:90, Dst: 00:04:96:15:22:30
Internet Protocol, Src Addr: 10.50.100.99 (10.50.100.99),
Dst Addr: 10.50.99.200 (10.50.99.200)
User Datagram Protocol, Src Port: 65476, Dst Port: 5080
Session Initiation Protocol
    Status-Line: SIP/2.0 200 OK
    Message Header
        Via: SIP/2.0/UDP 10.50.99.200:5080;branch=z9hG4bK-1-12
        From: <sip:sipphone10000@cscf99.lab>;tag=1
        To: <sip:sipphone10001@cscf99.lab>;tag=1-SIPP-UAS
        Call-ID: 1-31866@10.50.99.200
        CSeq: 6 BYE
    Message body
```

Figure C.10. 200 OK response from Proxy to UAC

Appendix D

Test data: Capacity

D.1 Asynchronous Prototype

Call rate (s^{-1})	CPU usage (%)
0	0
100	1
200	2
300	3
400	4
500	4.5
600	5
700	6
800	7
900	8
1000	8.5

Table D.1. CPU usage as a function of call rate

Process type	CPU time (ms)	Accum	Peak
10198	22125	1	1
10201	27111	1	1

Table D.2. Data retrieved from pts command. *Accum* is accumulated number of process instances. *Peak* is highest number of concurrently active process instances.

Successful calls	499921
Total messages	24997872
Total calls	499957.4
CPU time per call	0.10 ms

D.2 Dynamic Prototype

Call rate (s^{-1})	CPU usage (%)
0	0
100	4
200	9
300	13.5
400	19
500	25
600	30
700	34.5
800	41
900	45
1000	55.6

Table D.3. CPU usage as a function of call rate

Process type	CPU time (ms)	Accum	Peak
10198	53691	1	1
10201	139091	2500209	69

Table D.4. Data retrieved from pts command. *Accum* is accumulated number of process instances. *Peak* is highest number of concurrently active process instances.

Successful calls	500038
Total messages	2500212
Total calls	500042.4
CPU time per call	0.39 ms

D.3 Pooling Prototype

Call rate (s^{-1})	CPU usage (%)
0	0
100	1
200	2
300	3
400	4
500	5
600	6
700	7
800	8
900	9
1000	10

Table D.5. CPU usage as a function of call rate

Process type	CPU time (ms)	Accum	Peak
10198	24787	1	1
10201	30684	100	100

Table D.6. Data retrieved from pts command. *Accum* is accumulated number of process instances. *Peak* is highest number of concurrently active process instances.

Successful calls	499921
Total messages	2499786
Total calls	499957.2
CPU time per call	0.11 ms

Appendix E

Test data: Latency

Min, mean, max, median, standard deviation and percentile values are all measured in *milliseconds*. Call rate is in *half calls per second*.

E.1 Asynchronous Prototype

Call rate	100	200	300	400	500	600	700	900
Min	0.77	0	0	0.05	0	0.01	0.57	0.14
Mean	0.95	0.96	0.96	0.96	0.96	0.97	0.97	0.99
Max	1.46	2.35	2.28	1.77	1.86	1.83	1.79	3.62
Median	0.96	0.96	0.96	0.96	0.96	0.97	0.97	0.97
Std dev	0.01	0.04	0.05	0.03	0.04	0.04	0.04	0.08
Count	10000	50104	10000	9994	10000	10000	10000	10000
75% <=	0.97	0.97	0.97	0.97	0.97	0.98	0.98	0.99
95% <=	0.97	0.97	0.98	0.98	1	1.01	1.02	1.11
99% <=	0.98	1.02	1	1.01	1.05	1.08	1.11	1.24

Table E.1. PROXY INVITE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.49	0	0	0.05	0	0	0	0
Mean	0.7	0.71	0.72	0.72	0.73	0.73	0.72	0.72
Max	1.34	2.38	2.05	2.55	2.56	2.47	3.08	2.82
Median	0.7	0.71	0.71	0.71	0.71	0.71	0.71	0.71
Std dev	0.02	0.07	0.11	0.14	0.16	0.15	0.14	0.13
Count	10000	50104	10000	9994	10000	10000	10000	10000
75% <=	0.71	0.72	0.71	0.72	0.71	0.72	0.71	0.72
95% <=	0.72	0.72	0.72	0.73	0.74	0.74	0.74	0.76
99% <=	0.73	1.07	1.51	1.72	1.72	1.7	1.52	1.39

Table E.2. PROXY 2xx INVITE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0	0	0	0	0	0	0	0.01
Mean	0.74	0.74	0.74	0.75	0.74	0.75	0.75	0.74
Max	1.91	2.36	2.49	2	2.52	2.15	1.91	1.95
Median	0.74	0.74	0.73	0.74	0.74	0.74	0.74	0.74
Std dev	0.05	0.06	0.07	0.07	0.07	0.07	0.08	0.08
Count	10000	50104	10000	9994	10000	10000	10000	10000
75% <=	0.75	0.75	0.74	0.75	0.75	0.75	0.75	0.75
95% <=	0.75	0.76	0.75	0.76	0.76	0.77	0.77	0.77
99% <=	0.98	1.04	0.89	1.16	1.08	1.03	1.2	1.05

Table E.3. PROXY ACK REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.68	0.41	0.34	0.71	0.67	0.04	0.7	0.49
Mean	0.89	1.28	1.5	1.65	1.68	1.69	1.71	1.62
Max	1.41	2.03	2.14	2.6	2.66	2.85	3.24	3.54
Median	0.89	1.34	1.58	1.71	1.67	1.64	1.63	1.45
Std dev	0.1	0.2	0.28	0.34	0.41	0.45	0.51	0.55
Count	10000	49674	10000	9954	9745	9236	8440	7577
75% <=	0.96	1.42	1.72	1.9	2.01	2.03	2.11	1.98
95% <=	1.05	1.5	1.83	2.12	2.33	2.48	2.59	2.72
99% <=	1.11	1.61	1.94	2.26	2.46	2.64	2.83	3.15

Table E.4. PROXY BYE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.01	0.02	0.02	0.01	0.11	0.05	0.07	0.02
Mean	0.71	0.68	0.69	0.75	0.77	0.83	0.87	0.95
Max	1.59	1.67	1.92	2.48	2.66	2.84	3.17	3.53
Median	0.72	0.69	0.56	0.51	0.49	0.49	0.47	0.51
Std dev	0.04	0.2	0.33	0.46	0.5	0.58	0.65	0.7
Count	10000	49674	10000	9954	9745	9236	8440	7577
75% <=	0.73	0.75	0.74	0.8	0.93	1.23	1.26	1.22
95% <=	0.75	1.1	1.47	1.77	1.89	2.12	2.38	2.59
99% <=	0.79	1.3	1.71	2.06	2.27	2.51	2.75	3.1

Table E.5. PROXY 2xx BYE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0.39	0.09	0.07	0.16	0.16	0.02	0.27	0.13
Mean	0.8	0.87	0.92	0.97	0.98	0.99	1	1
Max	1.54	2.16	2.18	2.28	2.45	2.43	2.64	3.09
Median	0.8	0.89	0.91	0.93	0.91	0.91	0.9	0.88
Std dev	0.04	0.11	0.17	0.21	0.24	0.26	0.28	0.31
Count	10000	49932	10000	9978	9898	9694	9376	9031
75% <=	0.82	0.92	0.98	1.03	1.07	1.14	1.16	1.13
95% <=	0.85	1.01	1.15	1.27	1.34	1.42	1.5	1.59
99% <=	0.92	1.21	1.41	1.64	1.72	1.79	1.88	1.99

Table E.6. SUMMARY ASYNCHRONOUS PROTOTYPE

E.2 Dynamic Prototype

Call rate	100	200	300	400	500	600	700	900
Min	0	0	0	0	0	0	0	0.01
Mean	0.83	0.87	0.97	1.08	1.42	1.73	2.25	3.51
Max	2.35	7.24	6.5	9.06	13.31	17.12	19.74	23.92
Median	0.82	0.82	0.82	0.84	0.84	0.86	0.89	1.1
Std dev	0.06	0.34	0.79	1.23	2.19	2.99	4.09	5.82
Count	9999	9998	9998	9934	9906	9705	9711	9996
75% <=	0.82	0.82	0.83	0.87	0.9	0.93	0.97	1.19
95% <=	0.83	0.84	0.93	1.04	6.39	11.35	14.59	18.89
99% <=	1.11	3.09	5.71	8.03	11.52	14.16	17.37	21.26

Table E.7. PROXY INVITE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0	0	0	0	0	0	0	0
Mean	0.63	0.66	0.72	0.8	0.99	1.11	1.32	2.03
Max	1.84	4.25	6.67	9.59	11.59	13.61	15.76	17.62
Median	0.62	0.62	0.62	0.62	0.63	0.63	0.63	0.73
Std dev	0.11	0.34	0.58	0.89	1.51	1.88	2.38	3.43
Count	9999	9998	9998	9934	9906	9705	9711	9996
75% <=	0.62	0.62	0.63	0.63	0.63	0.64	0.65	0.82
95% <=	0.63	0.63	0.66	0.73	4.9	5.8	7.15	11.51
99% <=	1.53	3.59	3.81	5.03	8.3	10.49	12.83	15.4

Table E.8. PROXY 2xx INVITE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0	0	0	0	0	0	0	0
Mean	0.67	0.7	0.74	0.79	0.88	0.97	1.07	1.44
Max	2.23	4.59	7.16	9.53	13.23	16.63	21.35	23.65
Median	0.65	0.65	0.65	0.65	0.65	0.66	0.66	0.77
Std dev	0.15	0.37	0.63	0.97	1.5	1.96	2.31	2.82
Count	9999	9998	9998	9934	9906	9705	9711	9996
75% <=	0.66	0.66	0.66	0.66	0.66	0.68	0.72	0.85
95% <=	0.66	0.66	0.67	0.68	0.81	0.99	1.52	3.17
99% <=	1.19	2.58	4.54	7.34	11.2	14.45	16.85	20.03

Table E.9. PROXY ACK REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.61	0.54	0.46	0.68	0.41	0.41	0.04	0.7
Mean	1.66	2.86	4.39	5.81	8.23	10	11.76	13.94
Max	2.67	4.45	7.17	10.43	13.4	16.33	20.26	22.94
Median	1.74	3.12	4.67	6.34	9.42	11.37	13.4	15.99
Std dev	0.35	0.83	1.51	2.22	3.12	3.89	4.67	5.48
Count	9998	9999	9999	9928	9703	9250	8967	8092
75% <=	1.88	3.54	5.62	7.79	10.76	12.89	15.28	18.39
95% <=	2.06	3.85	6.22	8.61	11.76	14.57	17.38	19.82
99% <=	2.16	4.04	6.42	8.94	12.33	15.84	18.21	21.22

Table E.10. PROXY BYE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.62	0.63	0.62	0.61	0.63	0.75	1.35	1.2
Mean	1.28	2.96	4.71	6.83	9.8	12.44	14.85	17.9
Max	1.98	4.64	7.2	10.47	13.48	17.13	23.27	24
Median	1.24	2.88	4.56	6.89	9.79	12.4	14.87	17.99
Std dev	0.17	0.48	0.9	1.28	1.45	1.84	2.29	2.57
Count	9998	9999	9999	9928	9703	9250	8967	8092
75% <=	1.33	3.32	5.48	7.81	10.98	13.75	16.26	19.62
95% <=	1.66	3.82	6.26	8.73	12.03	15.27	18.35	21.79
99% <=	1.79	4.03	6.48	9.06	12.61	16.17	19.27	22.84

Table E.11. PROXY 2xx BYE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0.25	0.23	0.22	0.26	0.21	0.23	0.28	0.38
Mean	1.01	1.61	2.31	3.06	4.26	5.25	6.25	7.76
Max	2.21	5.03	6.94	9.82	13	16.16	20.08	22.43
Median	1.01	1.62	2.26	3.07	4.27	5.18	6.09	7.32
Std dev	0.17	0.47	0.88	1.32	1.95	2.51	3.15	4.02
Count	9999	9998	9998	9932	9825	9523	9413	9234
75% <=	1.06	1.79	2.64	3.55	4.79	5.78	6.78	8.17
95% <=	1.17	1.96	2.95	3.96	7.18	9.6	11.8	15.04
99% <=	1.56	3.47	5.39	7.68	11.19	14.22	16.91	20.15

Table E.12. SUMMARY DYNAMIC PROTOTYPE

E.3 Pooling Prototype

Call rate	100	200	300	400	500	600	700	900
Min	0.91	0	0.03	0	0	0	0	0.01
Mean	0.97	0.99	1.05	1.03	1	1.01	1.01	1.05
Max	1.83	12.19	4.6	15.33	6.17	6.33	1.77	2.44
Median	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.98
Std dev	0.02	0.27	0.52	0.57	0.3	0.3	0.12	0.24
Count	10000	10000	10000	9999	10000	9998	10000	10000
75% <=	0.97	0.97	0.97	0.97	0.98	0.98	0.99	1
95% <=	0.97	0.99	0.99	1.01	1.07	1.23	1.39	1.79
99% <=	1.02	2.49	4.34	5.4	1.14	1.36	1.55	2.13

Table E.13. PROXY INVITE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.26	0	0	0	0	0	0	0.01
Mean	0.71	0.74	0.77	0.77	0.78	0.77	0.78	0.81
Max	1.25	4.12	4.04	8.21	4.86	10.31	6.02	6.65
Median	0.71	0.71	0.71	0.71	0.71	0.71	0.71	0.71
Std dev	0.03	0.23	0.37	0.45	0.44	0.46	0.46	0.36
Count	10000	10000	10000	9999	10000	9998	10000	10000
75% <=	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.74
95% <=	0.72	0.72	0.73	0.8	0.78	0.77	0.8	1.34
99% <=	0.93	2.65	3.49	3.67	4.04	3.53	3.75	2.36

Table E.14. PROXY 2xx INVITE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0.6	0	0	0	0	0	0.01	0.01
Mean	0.75	0.77	0.83	0.85	0.86	0.85	0.85	0.85
Max	1.74	5.88	4.38	8.88	6.15	6.27	5.77	5.37
Median	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
Std dev	0.08	0.24	0.44	0.66	0.65	0.64	0.64	0.61
Count	10000	10000	10000	9999	10000	9998	10000	10000
75% <=	0.75	0.75	0.79	0.76	0.79	0.78	0.78	0.78
95% <=	0.75	0.76	0.82	0.82	0.82	0.82	0.82	0.83
99% <=	1.04	2.05	3.88	5.17	5.18	5.19	5.15	5.04

Table E.15. PROXY ACK REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.38	0.05	0.75	0.06	0.75	0.75	0.75	0.72
Mean	1.21	1.96	2.73	3.44	3.86	4.25	4.65	5.38
Max	1.76	3.4	4.21	5.34	5.46	5.94	6.6	7.69
Median	1.28	2.12	2.76	3.54	4.22	4.91	5.36	6.19
Std dev	0.22	0.58	0.86	1.18	1.26	1.41	1.54	1.93
Count	10000	10000	10000	9641	8606	7883	7382	6434
75% <=	1.39	2.46	3.49	4.44	4.88	5.28	5.73	6.8
95% <=	1.49	2.66	3.91	4.98	5.19	5.5	6.04	7.21
99% <=	1.55	2.74	4.05	5.11	5.31	5.68	6.21	7.33

Table E.16. PROXY BYE REQUEST

Call rate	100	200	300	400	500	600	700	900
Min	0.68	0.11	0.73	0.27	0.74	0.73	0.68	1.24
Mean	0.95	2.48	3.91	4.75	4.9	4.97	5.05	5.18
Max	1.49	3.68	4.57	5.96	6.17	6.29	6.53	7.69
Median	0.93	2.61	4.03	4.96	5.04	5.14	5.1	5.01
Std dev	0.14	0.34	0.43	0.68	0.56	0.61	0.64	0.95
Count	10000	10000	10000	9641	8606	7883	7382	6434
75% <=	1.06	2.72	4.23	5.18	5.27	5.35	5.48	5.5
95% <=	1.2	2.82	4.39	5.51	5.6	5.63	5.91	7.02
99% <=	1.29	2.89	4.48	5.7	5.8	5.87	6.12	7.28

Table E.17. PROXY 2xx BYE RESPONSE

Call rate	100	200	300	400	500	600	700	900
Min	0.57	0.03	0.30	0.07	0.30	0.30	0.29	0.40
Mean	0.92	1.39	1.86	2.17	2.28	2.37	2.47	2.65
Max	1.61	5.85	4.36	8.74	5.76	7.03	5.34	5.97
Median	0.93	1.43	1.84	2.19	2.34	2.50	2.58	2.73
Std dev	0.10	0.33	0.52	0.71	0.64	0.68	0.68	0.82
Count	10000	10000	10000	9856	9442	9152	8953	8574
75% <=	0.98	1.52	2.04	2.41	2.53	2.62	2.74	2.96
95% <=	1.03	1.59	2.17	2.62	2.69	2.79	2.99	3.64
99% <=	1.17	2.56	4.05	5.01	4.29	4.33	4.56	4.83

Table E.18. SUMMARY POOLING PROTOTYPE