VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH A IMPLEMENTACE GENERICKÉ ÚSTŘEDNY VOIP S VYUŽITÍM FUNKCIONÁLNÍHO PROGRAMOVÁNÍ

DESIGN AND IMPLEMENTATION OF GENERIC VOIP EXCHANGE USING FUNCTIONAL PROGRAMMING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MAREK KIDOŇ
AUTHOR

VEDOUCÍ PRÁCE Ing. PETR MATOUŠEK, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt
Výtah (abstrakt) práce v českém jazyce.

Abstract
Výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova
Klíčová slova v českém jazyce.

Keywords
Klíčová slova v anglickém jazyce.

Citace
Marek Kidoň: Design and Implementation of Generic VoIP Exchange Using Functional Programming, bakalářská práce, Brno, FIT VUT v Brně, 2014

Design and Implementation of Generic VoIP Exchange Using Functional Programming

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana …

. . . .
Marek Kidoň
May 7, 2014

Poděkování

Zde je možné uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc.

Contents

Chapter 1

Prologue

With rapid development of information technology in last century a real-time commmunication such as a voice need to be transfered across new media. The Internet. A wide sort of new communication technologies were developed from new protocols to dedicated telephony hardware. Some of these technologies will be covered in later chapters especially the communication protocols. Similarly to traditional public switched telephone network (PSTN) a basic demand on telephony devices remains. End user still needs his hardware desktop phone whose basic behavior remains the same. Similarly telephone exchange behaviour remains the same apart from the underlying transport layer.

A wide variety of session establishmet protocols exist these days. We can take SIP or h.323 protocols as examples. Although they are quite different to each other, they share some common behaviour. They were built to provide signaling among exchange devices but more importantly among exchange devices and end devices. And end devices does not differ among different signaling protocols. They always consist of an earphone and a dial. When caller initiates a call remote part need to be somehow advertized usually with a ring and caller knows that remote phone is ringing when hears some sort of tone in his earphone. This behaviour does not change even with wider network bandwiths nor with modern smart phones. Every single signaling protocol was build upon these facts and tough, they share some common behaviour. This thesis focuses on extraction this common behaviour outside each implemented protocol and build them upon these generic protocol. We will also evaluate advantages and disadvantages of this approach. As a practical result VoIP exchange will be developed using these techniques.

Although functional programming is not common these days, it brings many advantages compare to the more traditional sequential approach to constructing programs. Building software VoIP exchange is a complex task and it requires right tool to be done without sacrificing any of the core aspect such system should have. High level of abstraction, strong typing and other constructs may show its full potential in building such system. Erlang is an industrial quality functional language which was designed to be used inside telecommunication systems. It was selected for its natural pedigree and other characteristics such as high-reliability and soft-realtime properties. Erlang will be closely covered later chapters.

Chapter 2

Internet telephony

The term internet telephony refers to provisioning of a communication services such as voice, fax or text over public internet rather than using dedicated public switched telephone network (PSTN). In spite of need to resolve issues like jitter and frequent high latency, this approach inevitably came on to the light with best-effort internet approach and dependent technologies. Steps involved in establishing successful call and corresponding principes remains the same:

- signaling employs session control and signaling protocols are used to set up or tear down a call
- channel setup
- converting analog signal (voice) to its digital representation for transport over digital ling
- encoding media using codecs to optimize the stream

Traffic in PSTN is transfererd over circuit-switched network. On the other hand, internet covers wide variety of transport technologies where the most common end technologies are based on packet-switched networks (Ethernet + Internet protocol). Since internet was designed as a packet-switched best-effort delivery network it does not incorporate network based mechanism to ensure packets were delivered. As a negative consequence a jitter or high latency may appear. Although widely used reliable TCP exist to outweight some problems, it is not suitable to be used in real-time voice transmission and does not outweight all problems. Network routers may classify network traffic traffic and process several network streams differently thus ensuring VoIP media traffic will be manipulated faster than regular traffic without low latency demands.

Chapter 3

ISO/OSI session and application layer protocols

ISO/OSI session layer is layer where signaling protocols take their place. Their job is to establish a media session among 2 or more participants. Steps involved in establishing media session are:

- localization of endpoint
- contact remote endpoints and determine willingness to communicate,
- exchange of media session information
- tear down media sessions

Media session can be almost anything from a voice, presence, text messages or fax. Examples of signaling protocols are SIP or H.323.

Signaling protocol communication flow copies a phone device behaviour.

3.1 H.323

H.323 is a signaling standard developed by Internet Telecommunication Union, Telecommunications Standard Sector (ITU-T). It is a set of protocols encapsulated inside a single standard. It contains protocol for call create and tear down, exchange of media session information. Some of them are listed below:

- H.225 is responsible for call initiation and Registration, Admission, Status (RAS) signaling
- H.245 encapsulates logic for call handling, including gateway option exchange such as codec.

[4]

H.323 network device architecture consist of the following network devices:

- terminal, end end point device such as terminal software running on PC, or a phone
- gateway is a device responsible for call routing, between networks and end device localization

- gatekeeper is unique device to H.323, it monitors network and in case of high network load wont enable additional calls, thus preventing network overload
- multipoint control unit (MCU) a device responsible for a conference signaling

3.2 SIP

SIP or session initiation protocol is signaling protocol developed by IETF, Multi-Party Multimedia Session Control Working Group. First version was 1.0 and was submitted as an Internet Draft in 1997. Since significant changes were done to improve the protocol, version 2.0 was submitted as an Internet Draft in 1998. In 1999 protocol reached the Proposed Standard level and is described in RFC 2543 []. In the following years several SIP extending documents were published.

From more technical point of view, SIP is a text protocol which makes it well readable. It is based on HTTP protocol from which inherits client-server model and use of URI's. From SMTP SIP borrows header-style (headers such as From, To, etc... are in both protocols and have similar meaning). SIP communication is based on Request-Response mechanims. Where Request points from UAC to UAS and Responses are directed vice-versa. Good Request examples are INVITE (a join request) or BYE (request for session termination). Response message uses HTTP response codes to indicate request consequences.

SIP protocol intelligence is distributed across different network devices. Every network device that is part of SIP signalization topology is called a User Agent (UA). UA are further divided into a quite complex hierarchy. A picture below serves as an topology example. Otherwise SIP network devices are similar to network devices mentioned in H.323 secion.[9]

- User Agent Clients, an end user device such as a
  - a soft-phone running on a PC
  - a hardware phone
- User Agent Servers
  - Proxy Server is responsible for routing among networks
  - Registrar Server provides registration point and localization of clients

3.3 LCPCPv1

Is a simple signaling protocol developed by a former company Siemens Enterprise Communications. In contrast with previously mentioned protocols, it is much simpler. LCPCP stands for Low-Cost Phone Control Protocols, but important is the phrase Low-Cost. End devices do not hold any signaling logic at all, because that would increase their build cost. Instead they are completely controlled by LCP exchange. It is binary protocol with client-server, request-response architecture and only a few basic messages. Such a concept is then reflected in protocol messages. They are much more low-level. LCPCPv1 device state could be described by 2 states. IDLE and ESTABLISHED.Device begins in state IDLE and transitions to state ESTABISHED as soon as successfuly associates to exchange. As a part of association process, information about device available hardware such as information about display, keyboard, possible codecs and others is exchanged. Responsibility for these devices is then taken by exchange. This is reflected in LCPCPv1 protocol which contains messages such as DSP_TEXT_CMD (to set a text on particular display) or RTP_OPEN_CMD (to open RTP port) to operate client devices. On the other hand, client tells exchange which keyboard button were pressed using KBD_DOWN_IND or KBD_DOWN_IND messages. Complete protocol definition is publicly available.

3.4 RTP

Is an application layer protocol for real-time transfer of data streams. It is considered a standard for transporting voice or video media in packet-switched networks. It is one of key elements of VoIP.It wokrks in pair with Real-time Transport Control Protocol (RTCP) where RTP carries media streams and RTCP carries flow information, creates statistics and optimizes the RTP streams. Since VoIP works upon packet-switched network RTP includes mechanisms to compensate jitter, messages received out of sequence.

Before media session (handled by RTP and RTCP) can be initiated, a session description need to be negotiated. That where Session Description Protocol (SDP) takes its place. SDP is standard format for describing media initialization parameters. It holds necessary information for initiating media session like end device ip address, RTP and RTCP ports and a codec. It is usually transfered as a data part of signaling protcol message.

Chapter 4

Declarative programming paradigm

Is a programming paradigm that describes computer program logic without describing its control flow. In much more common imperative paradigm one often describes:

- what computer program goals are
- how the program should reach given goals

Program run is devided into small steps. For example in language C single steps are separated using a semicon character. Program flow is defined as modifiing its internal state using steps.

On the other hand, when talking about declarative paradigm, we mark program state as undefined. Of course programmer is accustomed to programming language evaluation strategy otherwise he would not be able to build program properly. In declarative languages programer only specifies:

- what computer program goals are

Declarative languages could be devided into following subsets:

- functional
- logical
- hardware definition languages
- DDL and DML such as SQL

4.1 Functional programming

Is a subset of declarative paradigm in which programs are constructed using functions. Functional languages (languages built upon functional programming) are basically separated into 2 groups:

- purely functinal languages such as Haskell, Lisp,...
- partially functional such as Erlang, Clojure,...

Functions in terms of functional programming are much more mathematicaly oriented compare to imperative languages. When an argument is supplied to a function thus calling it, the function will never ever return different value when called again with the same argument. Consequence of this approach is immutable data, often lack of variables and undefined program state. Rather than loop recursion is used. Its formal basis is a lambda-calcul, a formal system used in mathematical logic. It was developed in 1930's by Alonzo Church to support his solution of the Entschiedung problem. Later was used as a formal basis for functional programming.

4.1.1 Evaluation strategy

Evaluation strategy refers to an approach of evaluating function arguments. Function arguments can be evaluated in different manners as we will show later. Evaluation strategy has a huge impact on how whole program is evaluated, on program speed and coding style requirements. From evaluation point important aspects are:

- when are function arguments evaluated
- what value is passed to the function

Call-by-Value

Call-by-value is the most common approach to evaluating function arguments. It is used by language such as a C or Erlang. When function is called, their arguments are first evaluated and the resulting value is then bound to variable inside function body. This is usually done by copying value to new memory segment. When new value is assigned to function parameter, the original value remains unchanged due to the assignment to different memory region.

Call-by-Reference

As a counterpart to the Call-by-Value approach, Call-by-Reference does not need to copy memory segments to pass parameters to a function. It rather passes reference. This approach is in some form or another implemented in most languages. Typically languages use Call-by-Value as default evaluation strategy but often support special syntax for Call-by-Reference approach. C programming language is an perfect example offering Call-by-Reference explicitly by introducing the pointer concept.

Call-by-Name

A major drawback of previous evaluation mechanisms were in cases when function argument was never used in function. In spite of argument was never used, it was always evaluated befare the function was called. Such approach wastes CPU resources and makes overall program run slower. A reasonable improvement could be to evaluate arguments only in cases when argument is used inside function body. And it is the point of Call-by-Name evaluation. Drawback is that if argument is used multiple times, it is also multiple-times evaluated.

Call-by-Need

Call-by-Need, also called the Lazy evaluation delays argument evaluation until it is truly needed. Improvments to Call-by-Name is that arguments are evaluated only once. Such improvement significantly speeds-up overall program run, but since operation order becomes indeterminate it is hard to combine with imperative features such as exception handling. It is main reason why this approach is not widely used. Haskell programming language adopted Lazy evaluation and makes following construct possible.[8]

Listing 4.1: map function example in Haskell programming language

```
main = [0 .. ] !! 6
```

In preceeding example is taken sixth element of list. But since our list is infinite in other than lazy evaluation concept program would stuck in endless loop.

4.1.2 Type systems

If type is a property that can be assigned to various programming construct (for example a variable, expression or function are programming constructs), then type system is a pack of rules that assign a type to a construct. Its main purpose is to reduce bugs by defining interfaces between various parts of computer program. These parts when connected (for example function call) can be checked for consistency. This check can be performed statically at compile time or dynamically at run time. Various types can be declared implicitly, explicitly or infered.

- statically typed usually require explicit type declaration, type consistency can be checked at compile time
- dynamically typed languages do not require explicit type declarations, their consistency is checked at program run

Type inference is an action which performs compiler of statically typed language (such as a Haskell) before checking program for type consistence. Most significant drawback of this approach is that programmer doesn't need to explicitly specify type when declaring constructs. Compiler can infer declaration types from its construct.

Another important aspect of type system is wheter it is strong or weak.

- weakly typed languages
- strongly typed

Chapter 5

Erlang programming language

Erlang is a functional programming language with high emphasis on high realiability and heavy concurrency. These two main features go hand to hand with support for distribution applications and fault tolerance. It is well suited for applications whose main characteristics are:

- extremly reliable
- distributed
- soft real-time
- concurrent

5.1 High reliability

High reliability is one of Erlang key factors. Although it will not solve all problems, by inheriting different but simple approaches to handle errors and exceptions and using well designed and robust but very general library modules, Erlang will make your job easier and in most cases more native.

5.1.1 Fault tolerace

Is a 'keep it running' approach to error handling. It means that whenewer a part of system (process, group of processes) crashes, we will let the rest of the system alive. The Erlang VM will tell us, when the crash happened and why it did it. It is programmer's responsibility to ensure that the crash will not affect correct system behavior in global. We can take our VoIP switch as an example. If user invokes conference feature which is not implemented and system crashes in local scale user lost his call but in global scale, simultaneous calls should not be affected and switch should be up and running, ready to process any new requests.

Supervision tree is a concept trough which fault tolerance could be reached. Every process in such a model is either a worker or a supervisor. Worker job could be any routine work you can imagine starting from reading a file to a database server. On the other hand supervisor only job is to monitor other processes, workers or other supervisors and if any supervising process goes wrong it will restart it, stop all processes that are being supervised, etc…In general such action is called a restart strategy.

Each process in Erlang system should have its supervisor. Process diagram will mostly result in a tree where only process without supervision will be a root.

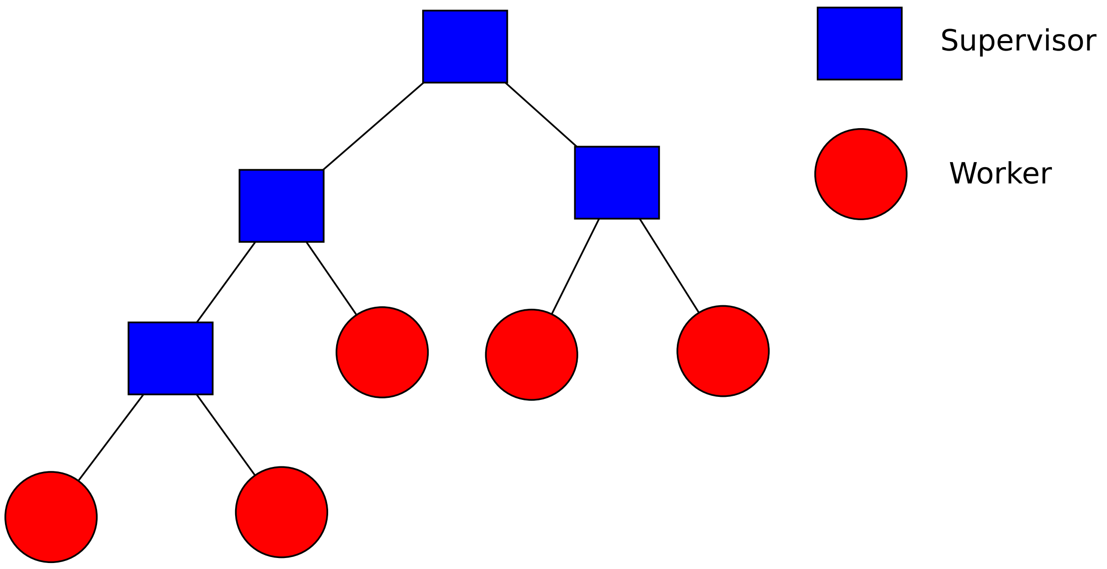


Figure 5.1: Example supervision tree

5.1.2 Hot-code swaping

In addition to fault tolerance, when a bug is found but system does not crash of that fault we need to fix our software and provide our feature to customers. In manner of high reliability and high up-time such a fix needs to be done when system is running. Again our VoIP switch can serve as an example. When bug in conferrence system is reported, hot-code swaping allows us to fix source code, recompile selected source files and insert fixed code into a running systems. [3] [6]

5.2 Massive concurrency

Any programming language that features concurrency inherits one of the following models:

- Shared state concurrency. This model use shared state to communicate among processes. Such a state could be represented by a shared piece of memory that processes can access. It is usually very fast but especially in complex systems it requires a lot of synchronization overhead to prevent race-conditions. This synchronisation becomes extremly hard in distributed environment where network faults and other downsides come onto light. It is also harder to proof correct. Shared state concurrency is the approach that almost every programming language use these days.

- Message passing concurrency is model where does not exist any shared state among processes. Instead processes communicate by message passing. Messages could be synchronous or asynchronous and reliable or unreliable. It depends on implementation. Advantages of this approach is that one does not need mutual exclusion since every process has its own state, message passing thus serves as a synchronisation mechanism. On the other hand it could be and usually is slower than shared state concurrency.

Erlang uses message passing concurrency. In case of Erlang it is a model that fits needs for distribution, and fault tolerance trough which high reliability is implemented.

As an addition to MPC, its processes are lightweight and fast. Erlang does not use OS process/thread capabilities as a backend for its concurrency model. Instead it uses own threads running separately in VM completely indepdendent of the underlying OS.This approach results in extremly fast threads.

5.3 Erlang history and philosophy

Erlang as a language was developed in 1986 in Ericson Computer Science Laboratory. Later the laboratory was to Ellemtel company and development continued. Initial motivation which later lead to Erlang was *to make something like PLEX, to run on ordinary hardware, only better*[1] where PLEX is Ericson proprietary language developed to run on AXE platform. Erlang was highly influenced by PLEX design. Members of the laboratory started implementing exchanges in every possible language that could run general purpose machine and operating system (4.2 BSD UNIX on Vax/11750 at the time) and compare results. They tried Ada, Concurrent Euclid, PFL, LPL0, Frames and CLU. The results were pretty simple:

- smaller language is prefered over large and complex language
- good concurency support is essential
- Logic programming was considered best alongside Functional programming which seemed to have minor issues

Later experimenting with prolog continued. They developed an meta-interpreter which was rapidly expanding. Next message-passing concurrency was added. This could be considered first Erlang implementation. Written in Prolog. Erlang was popularity was slowly growing in Ericson and was selected as implementation language for a few projects. Later they wanted to leave Prolog experimental implementation and make a stable one. After several exotic attemts such as cross compilation to other languages they implemented Erlang virtual machine in C. This implementation was 70 times faster then its Prolog predecessor.

5.4 Why Erlang

- fast, highly parallel and
- good concurency support is essential
- Logic programming was considered best alongside Functional programming which seemed to have minor issues

5.5 Erlang type system

Erlang is dynamically typed language. Erlang compiler will not complain on evident errors, instead every error will be caught at run time and it is programmer responsibility to handle this. This is often understood as one of major Erlang drawbacks. Reasons for this is Erlang uniqueness. Erlang features highly advanced concepts such as hot-code swaping and message passing concurrency which makes difficult to check for type correctness. As an example, it is hard to check types for code that does not exist yet (hot-code swaping). A number of project tried to create comprehensive type system for Erlang, but none of them really succeeded. Deeper study of this problem is considered beyond this paper but as a final conclusion I will quote Joe Armstrong: *It seems like it should be 'easy'-and indeed, a few weeks programming can make a type system that handles 95% of the language. Several man-years of work [by some of the brightest minds in computer science] have gone into trying to fix up the other 5%-but this is really difficult.*[2]

Fortunatelly for Erlang engineers a pack of tools which resulted from research at Uppsala University exist to fill this gap.

5.5.1 TypEr

Is a tool for automatic type inference. TypEr will check specified types againts inferred and check for inconsistencies. Specifying function parameter and return types has has yet another important consequence. It makes source code understanding significantly easier.

5.5.2 Dializer

Is an acronym for A DIscrepancy AnaLYZer for ERlang programs. It extends Typer and performs static analysis on Erlang programs. Its major function is to reveal software discrepancies such as: unreachable code, obvious type errors, redundant tests etc...

5.6 Erlang evaluation strategy

Call-By-Value is Erlang evaluation strategy as described in earlier chapters.

5.7 Open Telecom Platform

OTP is a set of design patters and general behaviours packed in compact library module. It is aware of all concepts mentioned earlier such as distributed programming or hot-code swaping thus making it insdustrial-ready standard for building fast, reliable and distributed applications. While it is not necessary to embed OTP into your system, it is highly advised to do so for at least 2 very good reasons:

- it is safer than reinventing already invented
- it makes source code much more readable because OTP general patterns are visible at first sight as shown in listing 7.1.

Listing 5.1: erlang behaviours

```
-module(factorial).
-behaviour(gen_server).
-export(...).

start_link() ->
```

```
...

init() ->
...
```

5.8 Behaviours

A behaviour is a formalization of common patterns. For example all supervising processes are similar, the only difference among them is what is their subject of supervision and how should they react when some of them crashes. Everything else is just a generic part, which is constant across all supervisors. What Erlang behaviour do, is encapsulating this generic part into a module and let programmer decide what the specific part will be. In terms of our supervisors: what the supervised processes will be and what should supervisor do when some of them crashes.

A Erlang behaviour consist of 2 parts:

- generic module
- specific module

When using a behaviour, programmer oblige to implement function call-backs in specific module. Such call-backs are then called from generic module.[7]

5.8.1 Generic servers

Is a behaviour where client-server relation is defined. This model is usually defined by a single server implemented by fulfilling gen server behaviour and almost any number of clients. As an example, server could handle some kind of resource and clients query server for resource share. Messages could be synchronous or asynchronous. It is important to note that client and server role is not reserved for a concrete process. Clients can in different situations behave like servers and vice versa.

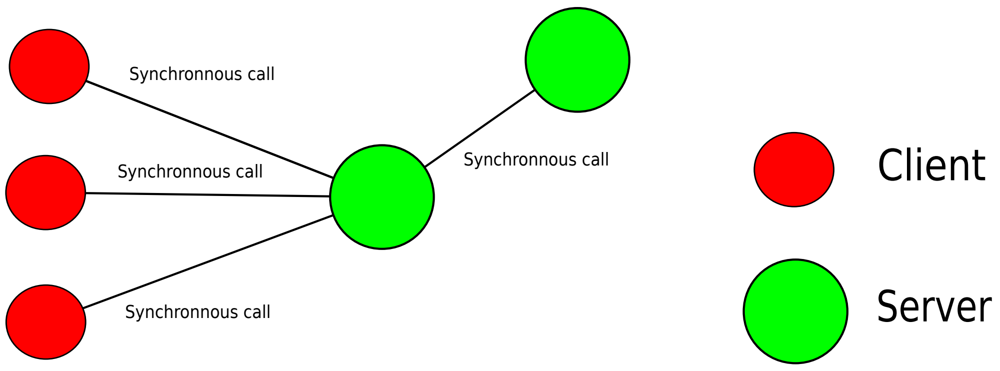


Figure 5.2: Example gen server behaviour hierarchy

5.8.2 Finite state machines

As described in Erlang/OTP official documentation. A finite state machine is a relation of the following form:

$State(S) x Event(E) \rightarrow Actions(A), State(S')$

If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.[5]

This behaviour as defined in gen_fsm behaviour is suitable for modeling any system that could be described with finite state machines. Such examples could from something really simple like a door which would consist of a few states and events (door could be in either open or closed state, and will react to events like open and close) to complex protocol stack. Later our generic protocol will be described using this behaviour making it simple, readable and easily extendible.

5.8.3 Event handlers

Event handler behaviour (gen_event) consist of one event handling manager and an arbitary number of event handlers which could be added or removed dynamically. Since event manager is generic, programmer only need is to implement specific event handler which could be inserted into event manager to interact with surrounding world.

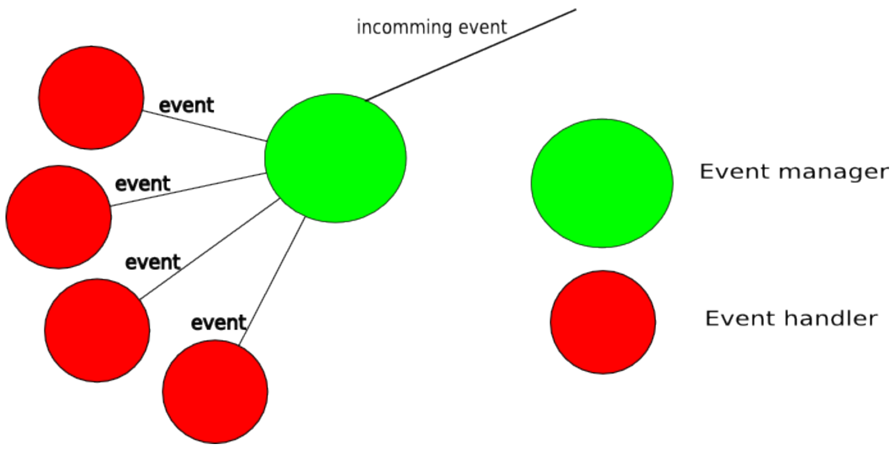


Figure 5.3: Example gen_event behaviour hierarchy

5.8.4 Applications

TODO

Chapter 6

Generic VoIP Exchange

The initial motivation hidden behind the Generic Exchange is to have exchange capable of handling various signaling protocols at once. This would allow us to create media sessions among clients working on the top of different signaling protocols. The Generic Exchange is style of exchange design that would easily allow us these features. This chapter will take us trough the design and implementation of such Generic Exchange.

A Generic Exchange on the highest level of abstraction consist of 2 basic parts. The first part is the generic generic core and the second are protocol specific peripherals, wrapped under the 'specific part' acronym. We will cover the Generic core in the first place, put some examples of how it works and then we will cover the specific peripherals.

6.1 Generic VoIP Exchange Core

The initial problem we need to solve is where to place the border where the generic part should start and what functionality it should cover. I decided to have the smallest possible amount of funcionlity in the core. The reason for this is that whenewer a new protocol is added into the generic exchange we do not want do modify the core at all. The core should persist and live without a change for as long time as possible. In ideal case, forever. Thus it will only consist of a generic protocol and a supporting database which will all necessary information about registrations and running dialogs.

6.1.1 Generic Protocol

A network protocol in general is an Agreement of information exchange in distributed networking[?]. Although this definition says what network protocols are and what is their purpose, it is too abstract, not really helping us in any way. A lower level definition describes network protocols as a finite state machines. Such technical definition nicely fits into our Erlang/OTP environment especially the finite state machine behaviour (gen_fsm). In terms of generic exchange, a main part of a core should be generic finite state machine. It is the most crucial part, theoretically shared by an arbitary number of specific protocol (SIP, H.323, LCPv1). Since every protocol is different designing errorless, shared, generic logic is very difficult.

Our job at this point is to develop a generic protocol using finite state machine as a description technique and Erlang/OTP gen_fsm behaviours as the right tool to implement it. Every signaling protocol is very complex. Designing a generic protocol for such a vast set of states is beyond the scope of this thesis. Instead, a few sample but significant use cases will be selected, and upon these state set a generic protocol will be developed. A final set of use cases:

- successfull registration both SIP and LCPCPv1
- simple call from SIP to SIP client
- simple call from SIP to LCPCPv1 client
- call reject
- call cancel
- proper handling of out of transaction messages

Generic protocol design could be handled several ways. Do we want a simple generic protocol but for the price of leaving a small (or maybe a bigger) piece of functionality in the specific part? Or do we want more complex generic protocol with specific part without any logic? Or some mix of previous?

Since we want a Generic core to persist for a long time without a change, a smaller protocol will be better. Let the peripherals handle more logic. Adding new complex peripheral is simpler and easier than implementing a simple peripheral and making changes in the Core. When unapropriate changes in Core are made, we could possibly damage the exchange. Also, in the second case our generic protocol would not be really generic. It would be generic until we decided to add a new specific protocol. In such a case we would need to add not only the specific part but we would need to redesign our Generic protocol again. It would be a huge and pointless waste of time and resources cause by wrong design decision. Instead what we want is to design a Generic protocol which is durable enough to survive adding a new specific protocol. Such a demand leaves as with the first option only. Generic protocol was designed by taking in account all above mentioned use cases and general phone behaviour. The following finite state machine was found:

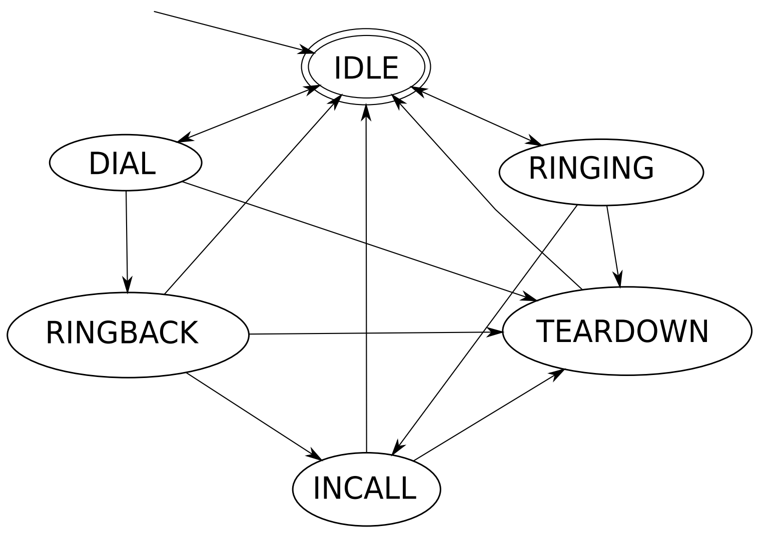


Figure 6.1: generic protocol

The above finite state machine displays a Generoc protocol and all possible transitions that could be taken during signaling. Transition conditions are not shown yet, becase it would make whole protocol hard to understand, they will be shown in a particular use case later. When a signaling transaction is started, the exchange creates 2 of this finite state machines. 1 for every part of transaction (in our case client and server transaction). The reason why, there are 2 transactions is for cases where 1 transaction wont suffice such as when user calls itself. Concept will be explained in more detail later. The following figure shows generic protocol message structure as present in my system.

Listing 6.1: generic protocol message structure

```
-record(generic_msg, {
        type                :: generic_msg_method(),
        target              :: generic_client_identifier(),
        caller              :: generic_dialog_party_identifier(),
        callee              :: generic_dialog_party_identifier(),
        upstreamRoute       :: route_path(),
        downstreamRoute     :: route_path(),
        routeToRecord       :: route_path(),
        sequenceNum         :: integer() ,
        specificProtocol    :: any(),
        timeToLive              :: non_neg_integer(),
        receivedOn              :: term()
}).
```

The finite state machine transitions from a state to a state as a reaction to generic messages. Mssages are structured into headers, these headers were designed to be sufficient and to contain only the necessary information. Particulary the routing information and a message type. Protocol specific headers not present in generic message are dumped in translation process. Lets take a closer look at the message structure now and we will demonstrate functionality on an real-world example.

header field name	example value	description
type	make_call	Message type identifier see 6.1.1 for details.
target	«1016»	Information about delivery destination.
caller	«1017», call_id, caller_part_id	Take a closer look at 6.1.1
callee	«1016», call_id, callee_part_id	Take a closer look at 6.1.1
upstreamRoute	[ ]	Basic routing information storage 6.1.1
downstreamRoute	[{{127,0,0,1}, 5070}]	Basic routing information storage 6.1.1
routeToRecord	[ ]	Information about a new persistent route 6.1.1
sequenceNum	3047	A traditional sequence number information
specificProtocol	[ ]	Specific protocol information 6.1.3
timeToLive	70	TTL information, prevents routing loops
receivedOn	127,0,0,1, 5060	An interface where packet was received

Table 6.1: Generic protocol message example

Generic message type header field

There are currently six generic message types implemented inside the exchange. The following table shows their name and meaning:

generic message type	meaning
associate	ask exchange to associate the client
make_call	ask the remote party to start media session
accept	positive response to a particular request
reject	negative response to a particular request
ring	ringing status indication
teardown	ask the remote party to end media session

Table 6.2: Generic protocol type header field values

Generic message caller and callee header fields

These header pair is very importatnt. It holds informatino about the direction of a message. Since generic messages couldnt be separated into request and responses but instead, a caller and callee information holds the information about the direction. It also deterministicaly identifies part of the dialog. This is especially important when message is being delivered to caller and callee dialogs which are represented by finite state machines.

Generic message downstreamRoute and upstreamRoute header field

A simple routing information storage. If present, exchange does not decide what the next hop will be, but instead the information is extracted from this header.

Generic message routeToRecord header field

This header field holds information about a new route. The intent behind this field is to once build a route and then use it all over again in future messages.

Generic message specificProtocol header field

This header holds all non-mandatory information about specific protocol. This information is not used by the generic part but may be used later by the specific part. A cleaner solution could be to save this information on the specific part of the exchange and retrieve it when needed. This solution wasnt implemented since it is more complicated and additional logic and storage space would be needed at the specific part.

6.1.2 Generic Exchange Core components

Dialog router

This process is an entry point to the Exchange Core. It has access to all necessary parts of the core. Its main responsibility is to lookup a dialog pair, one for caller part one for callee part, and distribute incomming message to both dialogs. Special case is when dialog does not exist. It then creates a new dialog pair. Since Dialog router is not responsible for dialog creation, it closely cooperates with Dialog manager which holds responsibility upon dialog creation.

Dialog Manager

As mentioned earlier, this process is responsible for dialog databse, dialog creation, dialog destruction and dialog lookup. It is used by both Dialog router and Dialog finite state machines.

Dialog finite state machine

Probably the most important process in whole Exchange. Every instance of this finite state machine represents a single dialog. The dialog could be either caller or callee. The dialog itself performs message routing and checks the state of the dialog. It interacts with Dialog Manager in case it finished its job. Whenever a dialog is stoped (reached the final state IDLE) it asks to remove the dialog from dialog table handled by Dialog manager and then it terminates.

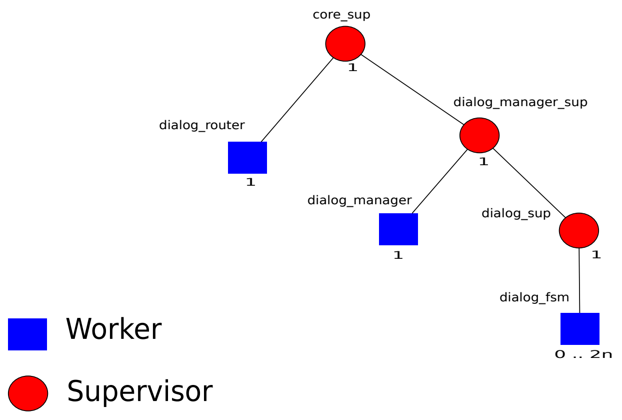


Figure 6.2: Generic Core process diagram

6.1.3 Generic exchange core function principes and examples

So finally, exchange core is constructed from the following parts: a dialog manager and a supervisor. These 2 processes take care of creation and correctness of dialog database. Dialog database is represented by Erlang table storage, working on similar principes as DBMS.Next important part is the dialog router. Whenever a generic message arrives at exchange core, it performs a lookup into a dialog databse and gets a pair of generic protocols represented by finite state machine as described earlier. Next is performed resolution which dialog belongs to particular part of dialog. As a final step, generic messages is tagged with on of following atoms: {fromRP, fromTU} indicating whenever the messages arrived from caller or callee and forwarded to a correct dialog. If dialog router looks up a dialog and there is no dialog found, it creates new pair of dialogs, and then works as described earlier. When a dialog receives a message and no proper transition can be performes, it indicates that invalid messages was received. In case of unassociated client, exchange still works as described earlier but performs asociation check explicitly in IDLE state. The following example explains how Generic protocol works, what messages are present in the dialog and how the Generic protocol behaves. (figure 6.1.3).

At first a pair of dialogs, is created in state IDLE.The red part stands for a caller initiation sequence and the green one stands for the callee initiation sequence. After both dialogs hit the INCALL state, the caller decides to teardown the media session. Callee

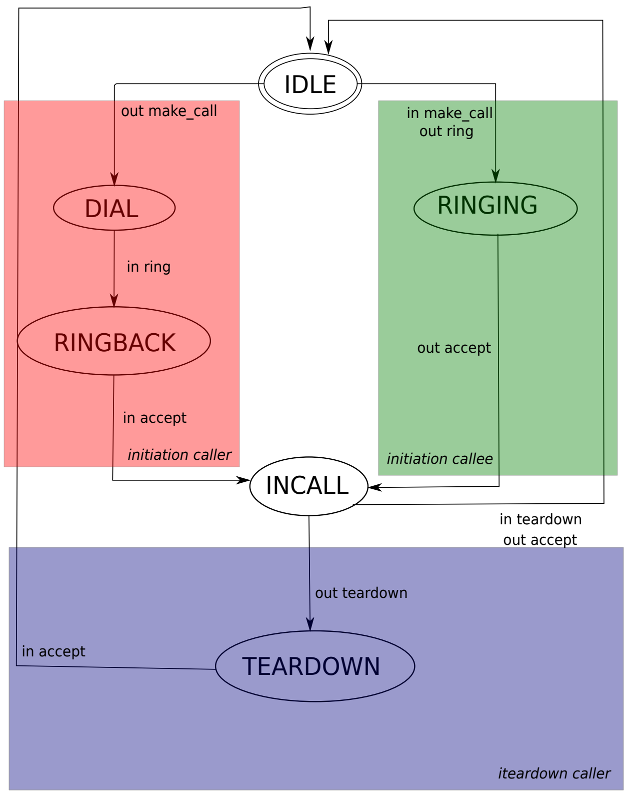


Figure 6.3: Generic exchange design

immediatelly transtitions to the IDLE final state, the caller takes the teardown sequence represented by blue mark.

6.2 General VoIP Exchange peripherals

Generic Exchange peripherals also called the specific part, is a part of the exchange, that is handling all specific signlaing protocols. There can be 1 to n peripherals, each handling different signaling protocol. Each peripheral is responsible for receiving packet from network and its parsing, taking care of a minimal context and passing it to generic core. For example, whenewer is signaling packet received from network, it is parsed, next the peripheral looks at minimal context of current message and if necessary, responds to sender with apropriate response. Then message is deterministically translated into generic message as described in section 6.1.1 and passes the message to the exchange core. When a generic message is received from the core, same actions are performed in reverse order. There is currently implemented SIP in full scope as mentioned earlier in section 6.1.1 and LCP, implemented partially.

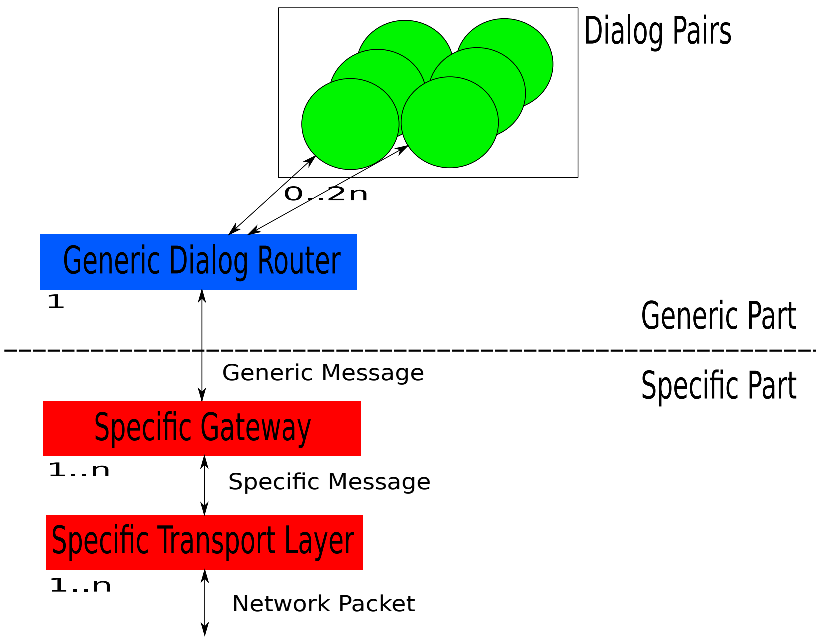


Figure 6.4: Generic exchange design

6.2.1 SIP peripheral

Basically it consist of 2 processes. The first one is a transport. Its responsibility is to receive and send packet over a socket. Since we use UDP sockets, its an UDP transport. It is used by the second process which is an SIP gateway. Its job is to deterministically translate SIP message into Generic message and vice versa. It also responsible to handle signaling communication which is beyond generic protocol scope.

SIP 3-way handshake

Such situation happens every INVITE transaction. Since INVITE transaction is a 3-way handshake and generic protocol does not understand concept of 3-way handshake.

SIP/CANCEL vs SIP/BYE

Generic protocol is does not recognise difference between BYE and CANCEL messages. They transfer the same information but in different context CANCEL is used to tear-down media session before media session is established, BYE is used after. Translation of generic_protocol/teardown message into SIP/CANCEL or SIP/BYE is performed using context information that is held in specific part of generic message but can only be understood by SIP gateway.

Contacts header

SIP Contact header holds information about where client could be physically reached. It is usefull in cases when clients do not want to use exchange for signaling once they exchanged Contact information. Simple and extremly usefull concept but absolutely unusable in case when different signaling protocols are used. The problem is overcomed by forcing Contact information of exchange to force both clients to communicate over exchange rather than peer-to-peer.

6.2.2 LCP peripheral

TODO

6.3 Summary

In this section I will try to summarize all important factors of the exchange and both parts will be put together to show its functionality as a package. The following diagram shows the supervision tree as it is implemented in the generic exchange. After reading the first two sections of this chapter, the diagram should be self-explanatory.

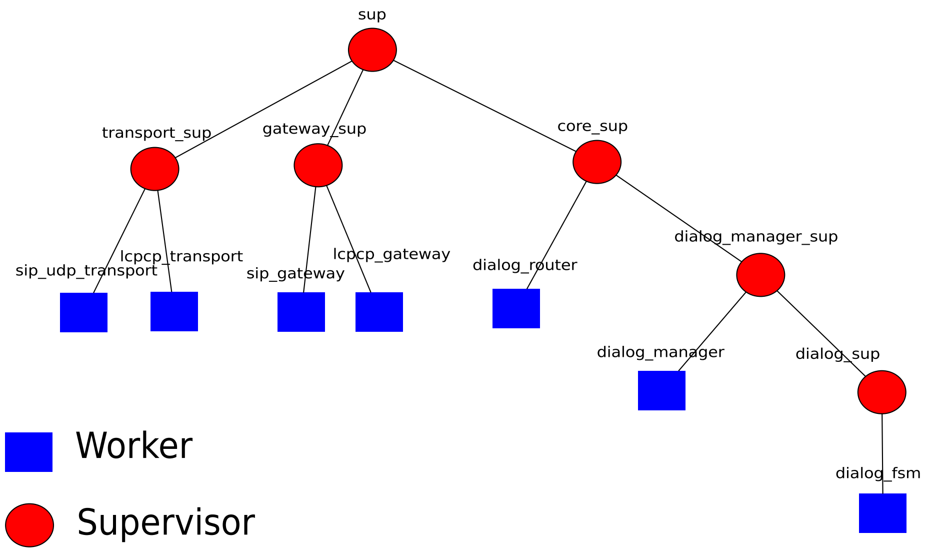


Figure 6.5: Generic exchange supervision tree

The following table sumarizes the overal count of active processes during exchange run. The reason for 'n' in worker count is that it depends on number of running dialogs and associated clients.

process type	count
supervisor	6
worker	7-n

Table 6.3: Generic Exchange process diagram metrics

The live working of exchange is by my opinion best understood on a data flow diagram and corresponding message sequence chart. It could be easily understood, how single messages are translated to generic protocol and after routing is performed, how it is then translated back to specific protocol. Consider the following dialog initiation sequence.

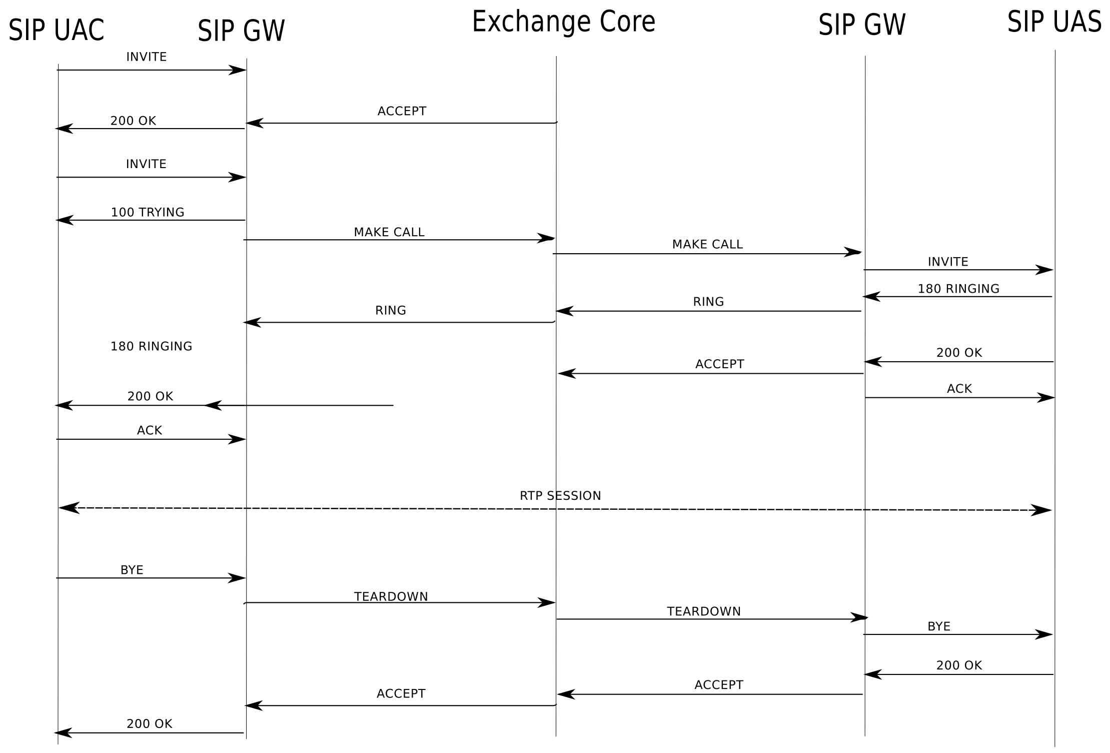
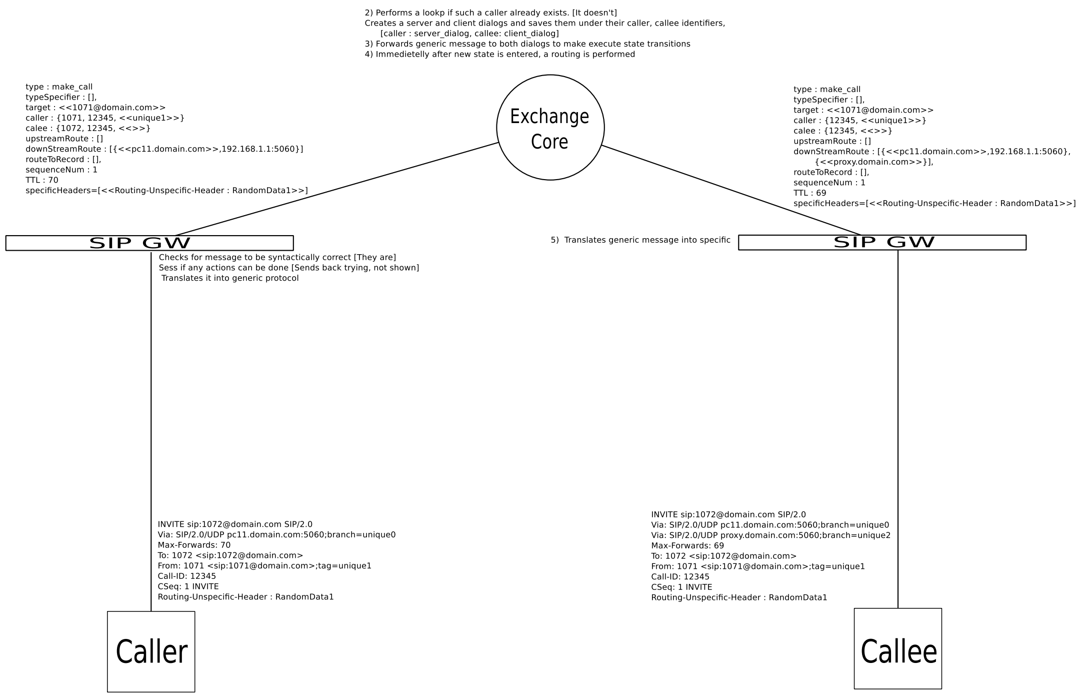


Figure 6.6: SIP to SIP simple call message sequence chart

To save space, we will only we will only show the SIP invite message and its flow through the exchange. At first, the incomming packet is received by SIP transport, parsed and saved into a sip message which is internally represented by a record. It is then passed to a SIP gateway which takes a look at the minimal context of a message and in case it is needed, it performs a specific action. In this case a 100 messages is replied to a sender. It then translates the SIP message into a generic and passes it to a generic core. After the core recieves a generic message, it looks on the caller and callee headers and passes the message to corresponding caller and callee dialog. The dialog may perform a transition into another state and performs a corresponding action. In this case a routing is performed. Next the generic message is passed to a specific gateway. Every dialog knows the PID of its gateway although it does not know which exactly it is. In our case it is the SIP gateway. It performs all the actions mentioned earlier in reverse order.



```
2) Performs a lookp if such a caller already exists. [It doesn't]
Creates a server and client dialogs and saves them under their caller, callee identifiers,
    [caller : server_dialog, callee: client_dialog]
3) Forwards generic message to both dialogs to make execute state transitions
4) Immedietelly after new state is entered, a routing is performed
```

```
type : make_call
typeSpecifier : [],
target : <<1071@domain.com>>
caller : {1071, 12345, <<unique1>>}
calee : {1072, 12345, <<>>}
upstreamRoute : []
downStreamRoute : [{<<pc11.domain.com>>,192.168.1.1:5060}]
routeToRecord : [],
sequenceNum : 1
TTL : 70
specificHeaders=[<<Routing-Unspecific-Header : RandomData1>>]
```

```
type : make_call
typeSpecifier : [],
target : <<1071@domain.com>>
caller : {12345, <<unique1>>}
calee : {12345, <<>>}
upstreamRoute : []
downStreamRoute : [{<<pc11.domain.com>>,192.168.1.1:5060},
        {<<proxy.domain.com>>}],
routeToRecord : [],
sequenceNum : 1
TTL : 69
specificHeaders=[<<Routing-Unspecific-Header : RandomData1>>]
```

```
Checks for message to be syntactically correct [They are]
Sess if any actions can be done [Sends back trying, not shown]
 Translates it into generic protocol
```

```
5)  Translates generic message into specific
```

```
INVITE sip:1072@domain.com SIP/2.0
Via: SIP/2.0/UDP pc11.domain.com:5060;branch=unique0
Max-Forwards: 70
To: 1072 <sip:1072@domain.com>
From: 1071 <sip:1071@domain.com>;tag=unique1
Call-ID: 12345
CSeq: 1 INVITE
Routing-Unspecific-Header : RandomData1
```

```
INVITE sip:1072@domain.com SIP/2.0
Via: SIP/2.0/UDP pc11.domain.com:5060;branch=unique0
Via: SIP/2.0/UDP proxy.domain.com:5060;branch=unique2
Max-Forwards: 69
To: 1072 <sip:1072@domain.com>
From: 1071 <sip:1071@domain.com>;tag=unique1
Call-ID: 12345
CSeq: 1 INVITE
Routing-Unspecific-Header : RandomData1
```

Figure 6.7: SIP to SIP session initiation data flow diagram

Chapter 7

Exchange in action

Chapter 8

Erlang role in implementation

Apart from its significant diference compare to imperative languages. It made some programming expressions much more readable. As an example the following code snippets were taken directly from exchange source code.

In the following example a translation from Generic protocol to SIP Via header filed is shown. The reverse is needed to maintain correct order of Vias which is reversed in Generic protocol.

Listing 8.1: map function example in generic_exchange_sip_generic.erl

```
vias=lists:reverse(lists:map(
        fun({Domain, Port, Opts}) ->
                #via{domain=Domain, port=Port, opts=Opts}
        end,
        DownstreamRoute)),
```

The following example takes care of translation of SIP Via header field into Generic protocol DownStreamroute which has similar meaning. Instead of complicated loop in arbitrary imperative language, use of foldr funcion makes reading and understanding easier:

Listing 8.2: foldr function example in generic_exchange_sip_generic.erl

```
downstreamRoute = lists:foldr(
        fun(Via, Acc) ->
                [{Via#via.domain, Via#via.port, Via#via.opts}|Acc]
        end,
        [], Vias),
```

As a huge advantage I consider a combination of Erlang high-level properties combined with OTP and message-passing concurency. Trough implementation phase I encountered numerous errors in design (that could be preceden by making deeper design) and other minor errors. Solving these errors never took me more than few minutes. In most cases it usually took me just a few seconds to figure out what's wrong. And trough whole implementation phase, I never used debugger. A huge part in this play Erlang atoms which help to identify software parts and messages nice and easy.

As a consequence of Erlang high-expression properties, implementation is really rapid. Instead of focusing on low-level constructions, I could spend my focus on what is trully important. I cought myself many times thinking of what do I want to express. The ratio of time writing time and this 'thinking' time is by my judgement higher than in arbitrary imperative language. Consequence of this is larger ammout if code that stay and will not be edited in the future.

At the end, the biggest advantage I discovered during the design phase.

Chapter 9

Automated software discrepancies discovery

From beginning Erlang was developed as a dynamically typed language. But in recent years with rapid development of strongly typed functional languages such as Haskell, software engineers realized it would be very usefull to have something similar in Erlang. Therefore a requirment for type checking and further static analysis was raised. There were several attempts to implement static type system into Erlang but all of them failed. It was due to Erlang unique features such as: dynamic code swaping, message passing concurrency and other.

9.1 Type system and type inference

In the first place, type inference tool was developed. Its main requirement was to work automaticly without any explicit guidance. It is called TyPEr and it uses success types mechanism to infer function type. Success typing is unfortunatelly beyond the scope of this book. Although typer and dialyzer can work without any explicit type annontations and infer function types by itself, programmer can explicitly annotate functions with type specification which is then used in type checking ant static analysis.

9.2 Static analysis

Dialyzer (a DIscrepancy AnaLYZer for ERlang programs) can identify software discrepancies such as obvious type errors, dead code and others.

9.3 Results after applying it to my source code

Static analysis was run over resulting source code, and I was more than surprised how many errors I have created. Many of them were hidden and hard to discover with unit tests I created. Some of them were more obvious. As a drawback, static analysis performed by Dialyzer takes quite a long time which makes it unsuitable to perform every time exchange is beeing compiled. Rather, I used it to check the code once in few hours. It took about 15 minutes on average laptop (place specification here if requested) to compute the results and the response it I have been given was almost always suprising and full of more or less obvious errors. It definitely helped me in order to produce more reliable code. Most errors it discovered were type errors in clauses that would match rarely usualy in error conditions, thus making them hard to discover. The second kind of errors were wrong type specifications. These are errors where type inference tool infers a function type and compares it with type I specified explicitly. These kind errors are priceless, because they help to discover crucial design flaws in early stages of implementation. They sort of work 'Are you sure this function work as you intended ?'.

Chapter 10

Conclusion

TODO

Bibliography

[1] Joe Armstrong. History of erlang.

[2] Joe Armstrong. Joe armstrongs's mailing list post on erlang type systems.

[3] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-600-5.

[4] David Bazala. *Telekomunikace a VoIP telefonie.* Technická literatura BEN, 2006. ISBN 80-7300-201-9.

[5] Ericsson_AB. Gen_fsm behaviour. `http://www.erlang.org/doc/design_principles/fsm.html`, 2013.

[6] Simon Thompson Francesco Cesarini. *Erlang Programming : A Concurrent Approach to Software Development.* O'Reilly Media, June 2009. 978-0-596-51818-9.

[7] Fred Herbert. *Learn You Some Erlang for Great Good!* No Starch Press, 2013. on-line version.

[8] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011. on-line version.

[9] Kevin Wallace. *VoIP bez předchozích znalostí.* Computer Press, 2006. ISBN 978-80-251-1458.

List of Figures

Listings