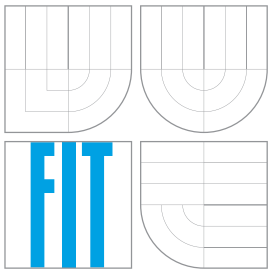
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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
AUTHOR

MAREK KIDOŇ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR MATOUŠEK, Ph.D.

BRNO 2014

Abstrakt

Tato práce se zabývá návrhem a implementací generické ústředny. Při implementaci byl použit jazyk Erlang, především díky jeho příznivým vlastnostem vzhledem k implementaci telefoního software. Práce také pojednává o deklarativním programování, jazyku Erlang a jeho konkrétních vlastnostech, výhodách a jeho nevýhodách. Na několika jednoduchých příkladech je pak úkázána funkčnost výsledné aplikace, která se jmenuje Genericka Ústředna. V závěru práce jsou shrnuty dosažené výsledky a návrh Generické Ústředny je porovnán s návrhem ústředny Asterisk.

Abstract

This thesis deals with the design and implementation of generic exchange. As an implementation language was selected Erlang mainly because of its positive features with respect to a telephony software. Thesis also talks about declarative programming, Erlang language, its particular properties, advantages and disadvantages. The resulting software is called the Generic Exchange and its funcionallity is demonstrated on a few simple examples. In the end of the thesis are summarized the results and Generic Exchange design is compared with the design of Asterisk exchange.

Klíčová slova

Internetová telefonie, VoIP, generická ústředna, Erlang, funkcionální programování

Keywords

Internet telephony, VoIP, generic exchange, Erlang, functional programming

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 Ing. Petra Matouška, Ph.D.

.....
Marek Kidoň
May 19, 2014

Poděkování

Rád bych poděkoval panu Ing. Petru Matouškovi, Ph.D. za metodické vedení a nespočet rad, které mi poskytl a Ing. Janu Wozniakovi za implementaci LCPCPv1 stacku a za poskytnuté rady.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Prologue

With rapid development of information technology in last century a real-time communication such as a voice needed to be transferred across new media. The Internet. A wide sort of new communication technologies were developed starting with communication protocols to dedicated telephony hardware. Some of these technologies will be covered in later chapters especially the communication protocols. Similarly to the 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 establishment protocols exist these days. We can take *SIP*[8] or *H.323*[12] protocol family 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. End devices in concept do not differ among arbitrary signaling protocols. They always consist of an earphone, a dial and possibly a display. When caller initiates a call remote part need to be somehow advertised usually with a ring and caller knows that the remote phone is ringing when he hears some sort of tone in his earphone. This behaviour does not change neither with wider network bandwidths 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 of this common behaviour outside each signaling protocol and encapsulate it inside a 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, compare to the more traditional sequential approach, it brings many advantages and new features to he field of large telephony systems. Building software VoIP exchange is a complex task and it requires right tools 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-real time properties. Erlang will be closely covered later chapters.

Chapter 2

Internet telephony

The term Internet telephony refers to a provisioning of a communication services such as a voice, a fax or a 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:

1. Signaling employs session control and signaling protocols are used to set up or tear down a call,
2. channel setup,
3. converting analog signal (voice) to its digital representation for transport over digital link,
4. encoding media using codecs to optimize the stream.

Traffic in *PSTN* is transferred over circuit-switched network. On the other hand, the Internet covers wide variety of transport technologies where the most common end technologies are based on a packet-switched networks (Ethernet + Internet protocol). Since the 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 out-weight 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 and process several network streams differently, thus ensuring VoIP media traffic will be manipulated faster than regular traffic without low latency demands. ISO/OSI session layer is a network 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 an endpoint usually represented by a hardware phone or a softphore,
- contacting remote endpoints and determine willingness to communicate,
- exchange of media session information,
- teardown of media sessions,

Media session can be almost anything from a voice, a presence, a text messages or a fax. Examples of signaling protocols are *SIP* or *H.323* which are currently in use in a majority of VoIP systems.

2.1 H.323

H.323 [12, 2] is a signaling standard developed by the Internet Telecommunication Union, Telecommunications Standard Sector (ITU-T). It is a set of protocols encapsulated inside a single standard. It contains several protocols for call initiation and tear down, exchange of media session information. Some of them are listed below:

- *H.225*[10] is responsible for call initiation and Registration, Admission, Status (*RAS*) signaling.
- *H.245*[11] encapsulates logic for call handling, including gateway option exchange such as codec.

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

- A terminal, an end point device such as terminal software running on PC, or a hardware phone,
- a gateway, a device responsible for call routing between networks and end device localization,
- a gatekeeper is unique device to *H.323*, it monitors network and in case of high network load won't enable additional calls, thus preventing network overload,
- a multi-point control unit (*MCU*), a device responsible for a conference signaling.

2.2 SIP

SIP [8, 16] 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[7]. 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 (and possibly more) readable compare to the *H.323*. 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 mechanism. Where request points from *UAC* to *UAS* and responses are directed vice versa. A Good request examples are an `INVITE` (a join request) or a `BYE` (request for session termination). Response message uses *HTTP* response codes to indicate request consequences which took their place on the UAS side.

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*s are further divided into a quite complex hierarchy. Otherwise *SIP* network devices are similar to network devices mentioned in the *H.323* (2.1) section.

- 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.

2.3 LCPCPv1

Is a simple signaling protocol developed by a former company Siemens Enterprise Communications. In contrast with previously mentioned protocols, it is much simpler. *LCPCPv1* stands for Low-Cost Phone Control Protocol, but important is the phrase Low-Cost. The end devices do not hold any important signaling logic at all, because that would increase their build cost. Instead they are completely controlled by a *LCPCPv1* exchange, they are associated with.

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 oriented. *LCPCPv1* device state could be described by two states. The `IDLE` state and the `ESTABLISHED` state. Device begins in the `IDLE` state and transitions to the `ESTABISHED` state as soon as successfully registeres to the exchange. As a part of association process, information about end device's available hardware such as information about a display, a keyboard, supported codecs and others is exchanged. Responsibility for these devices is then taken by exchange. *LCPCPv1* client does not know neither what text is in its diplay nor what media sessions are opened. Everything is handled by the 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[13].

For better understanding I will examine a few simple examples. Imagine a caller wants to establish a media session (probably a voice) with a *LCPCPv1* callee. Since *LCPCPv1* client does not maintain an information about active media sessions, exchange does. When an arbitrary media session initiation arrives from caller at the exchange, it decides whenever *LCPCPv1* client may answer the call. Suppose it may. To maintain proper phone behaviour, exchange tells the *LCPCPv1* client to open a port and to be prepared to receive a media session. It then replies to the caller that client is ringing and plays a ringing tone to a callee utilizing the *RTP* protocol. Observe the following figure 2.1 :

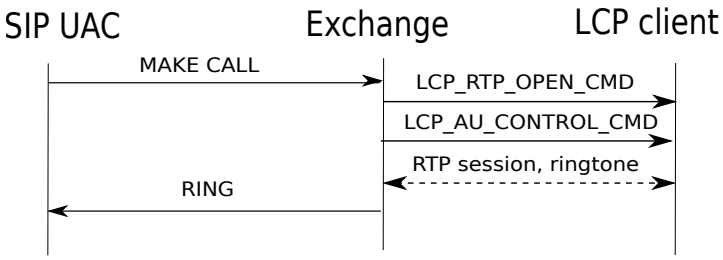


Figure 2.1: Message sequence chart of LCPCPv1 behaviour

2.4 RTP

RTP[9] is an application layer protocol for real-time transfer of data streams. It is considered a standard for transporting voice, video and other media in packet-switched networks. It is one of the key elements of VoIP.It works 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 the *RTP* and the *RTCP*) could be initiated, a session description need to be negotiated between end-points. And thats 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 transferred as a data part of signaling protocol message.

2.5 Summary

All above mentioned protocol (apart from *LCPCPv1*) are widely used in today's world. They managed to fight their place into the VoIP foreground and persist there for almost two decades now. This fact tells us something about their time overleaping design and robustness. Thus we cannot simply omit them and some of them we will be incorporated in our exchange.

Chapter 3

Declarative programming paradigm

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

- What computer program goals are,
- how the program should reach given goals.

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

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

- what computer program goals are

Declarative languages could be divided into following subsets:

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

3.1 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 functional languages such as Haskell, Lisp,...

- partially functional such as Erlang, Clojure,...

Functions in terms of functional programming are much more mathematically 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 a loop, recursion is used. Its formal basis is a lambda-calculus, 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.

3.1.1 Evaluation strategy

Evaluation strategy refers to an approach of evaluating function arguments. Function arguments can be evaluated in different manner. 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 languages 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 a new memory segment. When new value is assigned to function parameter, the original value remains unchanged due to the assignment to a different memory region.

```
int add(int x, int y)
{
   return x + y;
}
```

Listing 3.1: Call-by-Value example in C

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.

```
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;

   return;
}
```

Listing 3.2: Call-by-Reference example in C

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 before 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. Improvements 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[17].

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

Listing 3.3: Example of a map function in Haskell

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

3.1.2 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 by utilizing a 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 inferred.

- 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.

3.2 Summary

Declarative programming paradigm offers a new approach to quite conservative field of programming languages. Although they live with us for many quite a long time now, they have been continually overlooked in the field of large software systems. Erlang programming language utilized functional programming paradigm and together they offer a new approach to difficult tasks like massive concurrency, high reliability and real-time properties.

Chapter 4

Programming using Erlang

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

- extremely reliable,
- distributed,
- soft real-time,
- concurrent.

Although it is also effective in fast prototyping, it does not degrade it on '*just a scripting language*' level. With all above mentioned advantages it is suitable primarily for large software and especially for telecommunication systems.

4.1 High reliability

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

4.1.1 Fault tolerance

Erlang utilized a 'keep it running' approach to error handling. It means that whenever a part of a 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 happened. It is programmer's responsibility to ensure that the crash will not affect correct system behavior in global scope. We can take our VoIP exchange 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 the exchange should be still 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's only job is to monitor other processes, workers or other supervisors and if any supervised process goes wrong it will restart it, stop all supervised processes, 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 the only process without supervision will be a root. In the figure 4.1 is a resulting supervision tree of the Generic Exchange

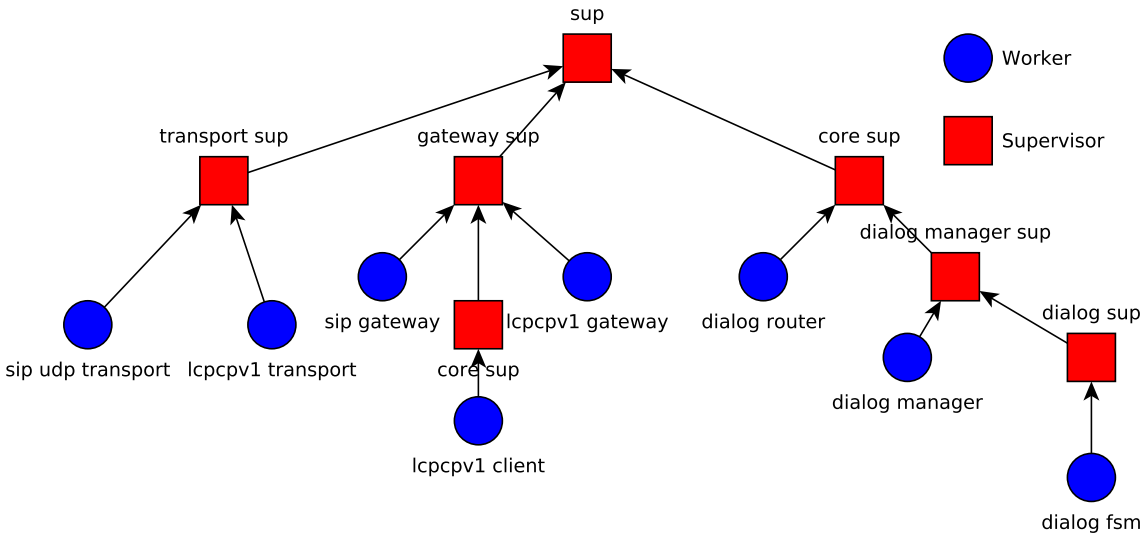


Figure 4.1: Generic Exchange supervision tree

4.1.2 Hot-code swapping

In addition to fault tolerance, when a bug is found, but our system does not crash from 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 conference system is reported, hot-code swapping allows us to fix source code, recompile selected source files and insert fixed code into the running systems [15, 4]. The Generic Exchange does not support hot-code swapping.

4.2 Massive concurrency

Almost every programming language that features a concurrency inherits one of the following models:

- Shared state concurrency (SSC). This model uses a 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 synchronization becomes extremely hard in distributed environment where network faults and other downsides come onto light. It is also harder to proof correct. Shared state concurrency is an approach, almost every programming language use these days.

- Message passing concurrency (MPC) 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 synchronization 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 perfectly fits the need 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 back-end for its concurrency model. Instead it uses own threads running separately in VM completely independent of the underlying OS.This approach results in extremely fast threads.

4.3 Erlang history and philosophy

Erlang as a language was developed in 1986 in Ericson Computer Science Laboratory. Later the laboratory moved 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 on a 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:

1. Smaller language is preferred over large and complex language.
2. Good concurrency support is essential.
3. Logic programming was considered best alongside Functional programming which seemed to have minor issues

Later experimenting with Prolog continued. They developed a meta-interpreter which was rapidly expanding. Next, message-passing concurrency was added. This could be considered the first Erlang implementation. Written in Prolog. Erlang popularity was slowly growing inside 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 attempts such as cross compilation to other languages they implemented Erlang virtual machine in C. This implementation was 70 times faster then its Prolog predecessor.

4.4 Why Erlang

Erlang was selected because among other functional languages, it offers message passing concurrency and a good support for utilizing it in large software projects. For example there is a Haskell with message passing concurrency model implemented as well but lacks a framework for simple application development on the top of it. Also in Erlang it is not difficult to start developing arbitrary project, because it is not purely functional and

developer does not need to dump all of his imperative habbits. A long development and a various large project from VoIP were written in it, so it can be considered well tested *'in the wild'* as well.

4.5 Erlang type system and static analysis

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's 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 swapping 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 swapping). Numerous projects 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.*[14]

Fortunately for Erlang engineers a pack of tools which resulted from research at Uppsala University exist to fill this gap. They are based on so called *'success typing'*.

4.5.1 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.

4.5.2 Dialyzer

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...

4.5.3 Result after applying static analysis to Generic Exchange source code

Static analysis was run over resulting source code (which was partially, explicitly typed), and I was more than surprised how many errors I have created. Many of them were hidden and hard to discover with the basic 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 being compiled. Rather, I used it to check the code once in few hours. It took about 15 minutes on average laptop to compute the results and the response I have been given was almost always surprising 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 under 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 ?'. The listing 4.1 shows a few discrepancies of Generic Exchange taken from static analysis result.

```
generic_exchange_lcp_gateway.erl:106: The variable Error can never match since
    previous clauses completely covered the type 'true'
generic_exchange_sip_gateway.erl:107: Function handle_specific/3 will never be
    called
generic_exchange_sip_generic.erl:7: Function sip_to_generic/3 has no local return
generic_exchange_sip_generic.erl:262: Function extract_body_from_generic/1 will
    never be called
generic_exchange_sip_generic.erl:287: Function generic_type_to_sip_class/2 will
    never be called
generic_exchange_sip_message_factory.erl:7: Function method_not_allowed/1 has no
    local return
generic_dialog_fsm.erl:258: The pattern [{DSIP, DSPort, _Opts} | _] can never match
    the type [{binary()} | {binary(),binary()},...]
```

Listing 4.1: Generic Exchange static analysis result

4.6 Open Telecom Platform

Open Telecom Platform (OTP) is a set of design patters and general behaviours packed in a compact library module. It is aware of all concepts mentioned earlier such as a distributed programming or a hot-code swapping thus making it industrial-grade 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:

1. It is safer than reinventing already invented.
2. It makes source code much more readable because OTP general patterns are visible at first sight as shown in the following listing 4.2.

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

start_link() ->
...

init() ->
...
```

Listing 4.2: Erlang/OTP behavioursi

4.7 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:

- a generic module,

- a 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[5].

4.7.1 Generic servers

Is a behaviour where client-server relation is defined. This model is usually defined by a single server implemented by fulfilling the `gen_server` behaviour and almost any number of clients. As an example, a server could handle some kind of resource and clients may query the 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. The following figure 4.2 displays a part of Generic Exchange utilizing `gen_server` behaviour.

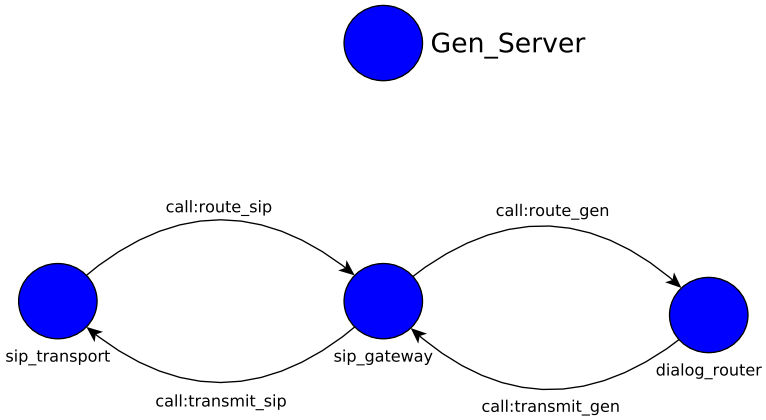


Figure 4.2: Example of `gen_server` behaviviour as present in Generic Exchange

4.7.2 Finite state machines

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

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

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'[3].

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 extendable.

4.7.3 Event handlers

Event handler behaviour (`gen_event`) consist of one event handling manager and an arbitrary 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.

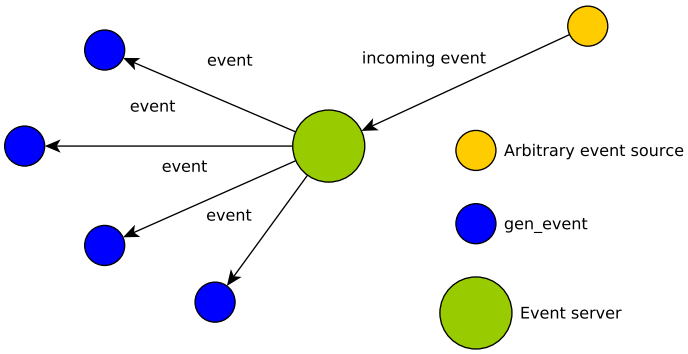


Figure 4.3: Example of `gen_event` behaviour hierarchy

4.7.4 Applications

Once Erlang application exists, a developer might usually need to safelly start and stop the application. Starting an application is not a big deal, but once application is running and starts numerous processes, stopping these processes one by one a big deal becomes. Also after developers software becomes more popular, a requirement to use it in other projects might get raised. The application behaviour encapsulates both. It provides a way to safely start, stop the application and a general API to use it as sub-application of larger projects.

Both the `LCPCPv1 transport` and the `SIP transport` which are part of the Generic Exchange were written using external libraries in form of the Erlang applications. The Generic Exchange itself fulfills the application behaviour too.

Chapter 5

Design and implementation of 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 the Generic Exchange.

The Generic Exchange on the highest level of abstraction consist of 2 basic parts. The first part is the 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. See the following figure 5.1.

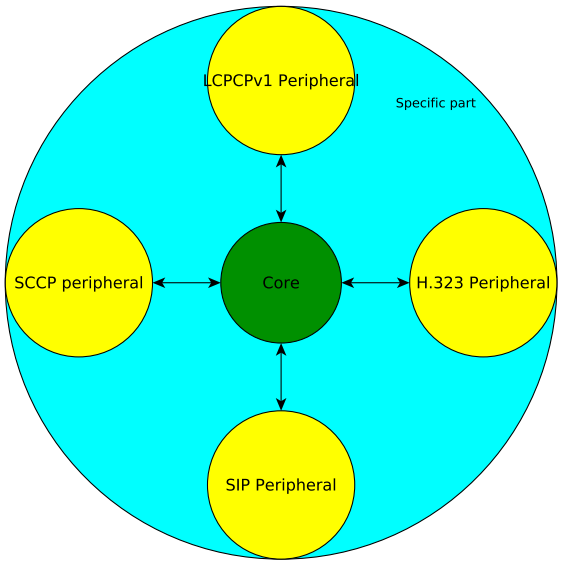


Figure 5.1: Generic Exchange high-level design

5.1 Generic VoIP Exchange Core

Basically a Generic Core is a high level component, shared by the peripherals. It encapsulates a general functionality shared by all peripherals into a single package and providing a simple way to access it. The Core should be as persistent and durable as possible and live without a change for as long time as possible. To achieve this goal, it is important to encapsulate as few components as possible. In case of maintenance or when a new signaling protocol is added to the Generic Exchange we do not want to change the Core at all. An additional effort and time is usually costly and lot of future funds could be wasted because of bad design decision.

Every VoIP exchange built upon arbitrary VoIP protocol performs routing. It sustains an information storage with information related to associated clients and their network location. It is also capable of performing a DNS look-up for foreign domain discovery. All these features are independent of underlying VoIP protocols and it is exactly the scope of functionality Generic Exchange Core encapsulates.

The main part of the Core is the Generic Protocol. It is represented by a finite state machine and maintains the functionality of state-full proxy. It is also used for routing purposes. When a peripheral performs routing, it has to fill the Generic Protocol message and pass it to the Core. It means that every peripheral present in Generic Exchange has to be able to deterministically translate its message into the Generic Protocol message. This feature serves as an interface to the Generic Core. After Core receives a message it performs a look-up to associate incoming message with a existing dialog pair (We will explain later why a single dialog wont suffice). If succeeds the message is passed to a dialog pair if not, a new dialog pair is created. Dialog pair then performs an appropriate action which can be either routing or answer back to sender. New generic message is then passed to corresponding peripheral which sends it to the network. See figure 5.2.

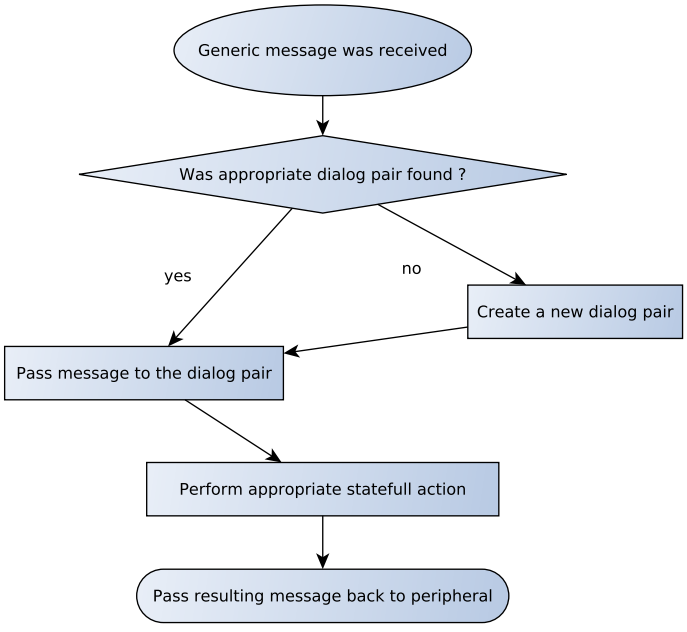


Figure 5.2: Generic Core data flow diagram

5.1.1 Generic Core process strcuture

The following process diagram shows a Generic Exchange Core sub-tree:

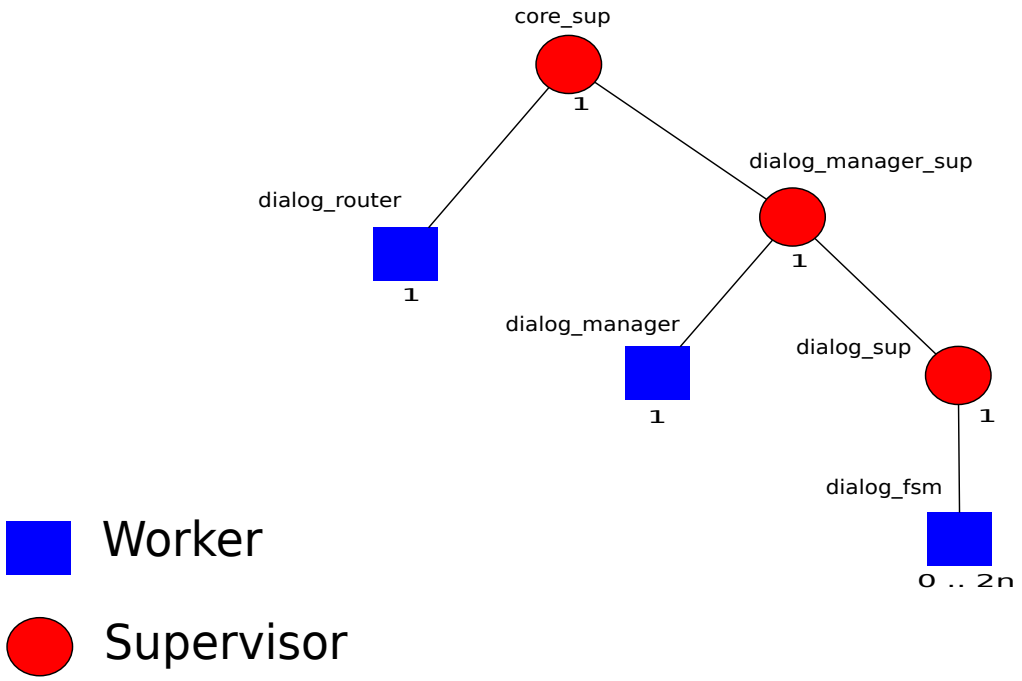


Figure 5.3: Generic Core process diagram

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 look-up a dialog pair, one for caller part one for callee part, and distribute incoming 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 database, dialog creation, dialog destruction and dialog look-up. Dialog database is represented by Erlang table storage, based on relation data model. 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 stopped (reached the final state IDLE) it asks to remove the dialog from dialog table handled by Dialog manager and then it terminates.

5.1.2 Generic Protocol

A network protocol in general is an Agreement of information exchange in distributed networking[6]. On a lower level, network protocols may be implemented (and they often are) as a finite state machines. Such technical feature nicely fits into our Erlang/OTP environment especially the finite state machine behaviour (`gen_fsm`). In terms of Generic Exchange Core, the most important part should be the Generic Protocol. It is the most crucial part, theoretically shared by an arbitrary number of specific protocol (*SIP*, *H.323*, *LCPCPv1*) maintaining features of a statefull proxy. Since every protocol is different designing error-less, shared, generic logic is difficult task.

Our job at this point is to develop a generic protocol using finite state machine as a description technique and Erlang/OTP `gen_fsm` behaviour 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 use cases a generic protocol will be developed. A final set of use cases:

use case	**SIP to SIP**	**SIP to LCPCPv1**
registration	yes	yes
establishing simple call	yes	yes
call reject	yes	yes
call teardown	yes	yes
proper handling of various errors	yes	yes

Table 5.1: Generic Protocol use case scope

Generic protocol design could be handled in 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 unappropriate changes in Core are made, we could possibly damage whole 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 caused by a wrong design decision. Instead what we want is to design a Generic protocol which is durable enough to survive an adding of 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 the general phone behaviour. In the figure 5.4 is Generic Protocol finite state machine as present in my system:

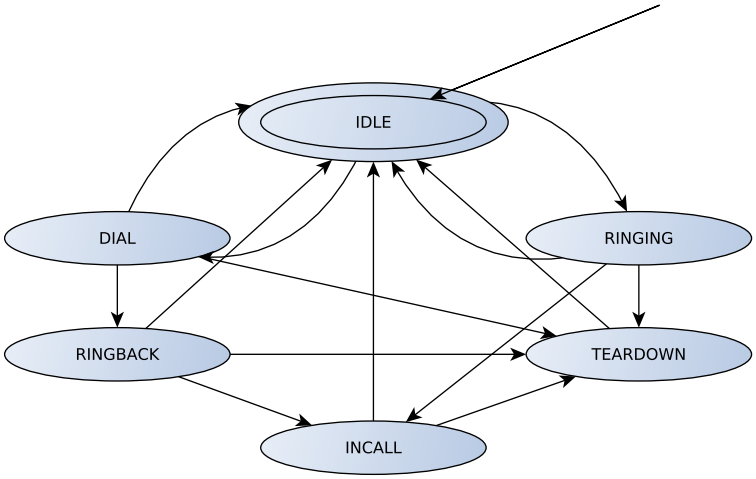


Figure 5.4: Generic Protocol

The above finite state machine displays a Generic protocol and all possible transitions that could be taken during signaling. Events are not shown yet, because 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 two of this finite state machines. One for every part of transaction (in our case client and server transaction). The reason why, there are two transactions is for cases where one transaction wont suffice such as when user calls itself.

Now lets observe Generic Protocol Message structure (listing 5.2).

```
-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()
}).
```

Listing 5.1: Generic Protocol message structure

The finite state machine transitions from a state to a state as a reaction to Generic Messages. Messages are structured into fields, these fields were designed to be sufficient and to contain only the necessary information. Particularly the routing information and a message type. Protocol specific information which are not present in Generic Message are dumped during translation process. Lets take a closer look at the message structure (listing 5.2) now and we will demonstrate functionality on the example (5.5) later.

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 5.2: Generic Protocol Message type field values

Generic message caller and callee header fields

These pair of fields is very important. It holds information about the direction of a message. Since generic messages couldn't be separated into request and responses but instead, a caller and callee information holds the information about the direction. It also deterministically 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 value is present, exchange does not decide what the next network 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 idea behind this field is, to build a route once 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. Example of carried values is SDP.

5.1.3 Example

At the end of this section, I will put an example which I hope will clarify all underlying principes. First the following diagram (in the figure 5.5 shows a media session initiation and teardown use case. Then example of Generic Protocol Message will be shown (in the table 5.3).

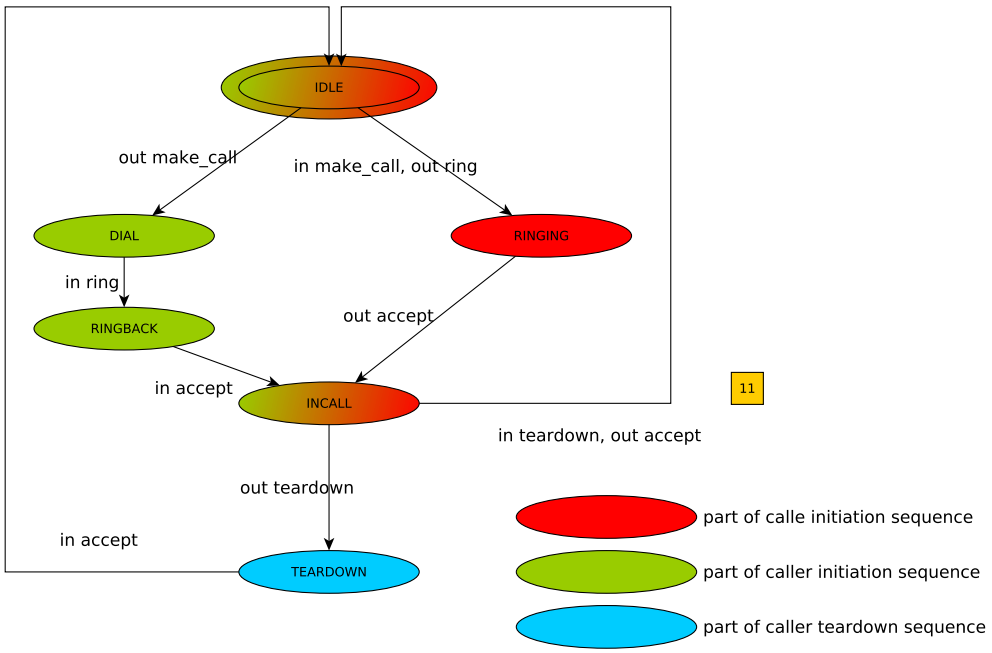


Figure 5.5: Generic Protocol use case

When a `make_call` message arrives at the Core, the Generic Exchange performs a look-up for existing dialog pair. And since it is not found the Core will then create a new dialog pair. At first a pair of dialogs, is created in state `IDLE`.The green part stands for a caller initiation sequence and the red one stands for the callee initiation sequence. After both dialogs hit the `INCALL` state, the caller decides to teardown the media session. Callee immediately transtions to the `IDLE` final state, the caller goes trough the teardown sequence represented by a blue mark.

field name	**example value**	**description**
type	make_call	message type identifier
target	«1016»	Information about delivery destination.
caller	{«1017», call_id, caller_part_id }	Take a closer below
callee	{«1016», call_id, callee_part_id }	Take a closer below 5.1.2
upstreamRoute	[ ]	routing information storage
downstreamRoute	[{{127,0,0,1}, 5070}]	routing information storage
routeToRecord	[ ]	information about a new persistent route
sequenceNum	3047	a sequence number
specificProtocol	[ ]	specific protocol information
timeToLive	70	TTL information, prevents routing loops
receivedOn	{{127,0,0,1}, 5060}	An interface where packet was received

Table 5.3: Generic Protocol Message example

5.2 General VoIP Exchange peripherals

Generic Exchange peripherals also called the specific part, is a part of Generic Exchange, which is handling all specific signaling protocols. There can be *one to n* peripherals, each handling different signaling protocol. Each peripheral is responsible for receiving packet from network, its parsing, taking care of a minimal context and passing it to Generic Core.

The minimal context is all necessary specific protocol functionality which is not implemented inside the Core (eg. *SIP* `100` response). The context is different across different specific protocols. It is saved in the specific part and the peripheral have to take into account the context to maintain proper functionality. During the design phase we decided to leave Generic Core as smallest as possible and let the peripherals handle more functionality. The context is a part of a peripheral where this out of Generic Protocol functioanlity takes its place. The following diagram (5.6) explains how an arbitrary peripheral works.

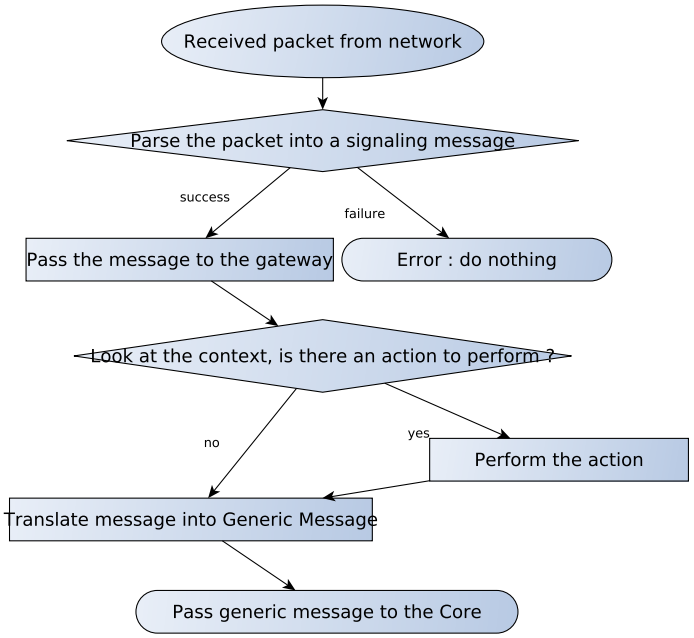


Figure 5.6: Generic Exchange peripheral data flow diagram

All peripherals consist of the following parts, each represented by a single process:

1. A `transport`, which is responsible for receiving or sending a packet from or to socket, and parsing or serializing of signaling messages,
2. a `gateway`, which is responsible for handling the minimal context and translation of signaling message into the Generic Message and vice versa.

5.2.1 SIP peripheral

SIP peripheral refers to a peripheral handling the *SIP* protocol. To build this peripheral I used the `NKSIP`[1] Erlang framework. Instead implementing own parser I used an existing one from this library. This section mainly explains how the *SIP* peripheral works and what challanges I faced during the implementation.

[1] `NKSIP`, Erlang *SIP* application server : http://kalta.github.io/nksip/docs/v0.3.0/nksip/index.html

SIP to Generic Protocol translation

For every peripheral it is crucial to be able to deterministically translate its message into Generic Protocol message. Otherwise it would not be able to use the Generic Exchange Core and perform routing or providing the state-full proxy services. The following table explains how *SIP* requests and responses match Generic Protocol message types.

SIP request / response	Generic Protocol Message type
Register	associate
Invite	make_call
200 OK	accept
603 Decline	reject
180	ring
Bye / Cancel	teardown

Table 5.4: *SIP* request / response to Generic Protocol message type

For a full picture the following table shows how particular *SIP* header fields are translated into Generic Protocol fields.

SIP headers	GP header fields	Note
Via	downstreamRoute	branch parametr is saved, transport information is dumped
Max-Forwards	timeToLive	Value remains unchanged
From	caller	tag and user are saved, domain is dumped
To	callee	tag and user are saved, domain is dumped
Call_ID	caller, callee	is translated as part of both caller and callee
Content-Length	-	body length is newly calculated
Content-Type	-	information is retrieved from specificProtocol
SIP Body	specificProtocol	Usually SDP is saved and remains unchanged

Table 5.5: SIP request / response to Generic Protocol message headers

Since the Generic Protocol handles only the basic signaling behaviour, the following out of Generic Protocol scope situations need to be handled in the *SIP* peripheral. Please note that this approach is in complete harmony with initial design where this logic should be handled here and not in the Generic Protocol. The reasons were examined in the beginning of this chapter.

SIP 3-way handshake

Such situation happens every time `INVITE` transaction is routed trough the Generic Exchange. Since `INVITE` transaction is a 3-way handshake and generic protocol does not understand concept of 3-way handshake. The result of this that a `SIP ACK` request is dropped every time it reaches the peripheral. In addition to this, when peripheral receives

a Generic Protocol `accept` message in context of media session initiation, it immediately generates a `SIP ACK` message. The context is saved in Generic Protocol `specificProtocol` header field.

SIP/CANCEL vs SIP/BYE

Generic protocol is does not recognize difference between `BYE` and `CANCEL` messages. They transfer the same information but in different context the `CANCEL` is used to teardown media session before media session is established, the `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 the *SIP* gateway.

Contacts header

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

5.2.2 LCPCPv1 peripheral

This peripheral rapidly differs from the *SIP* one. The *LCPCPv1* client do not maintain its internal state on a sufficient level. It means that every single important information about the client need to be handled by the Generic Exchange. This creates an additional stress on LCPCPv1 Generic Exchange peripheral. On the other hand, since the client state is maintained on the exchange, there is no need to translate from any specific protocol. We just create the Generic Message based on current client state and pass it to the Generic Exchange Core. Please note that only working feature is a registration at this time. As an implementation support I used `elcpcp`[2] library for parsing and network access.

To properly maintain states of associated clients, this peripheral features 2 additional components to the, `transport`, `gateway` duo:

1. `LCPCPv1 client` supervisor, part of Erlang supervision tree,
2. `LCPCPv1 client` which is represented by Erlang `gen_fsm` behaviour and maintaints state of each individial *LCPCPv1* end-device.

5.3 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 real operation of exchange is by my opinion best understood on a message sequence chart and its actual messages. 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. It is similar to a

[2] `ELCPCP` on Github : https://github.com/Tr1p0d/elcpcp

traditional MSC diagrams as present in the *SIP* rfc. It is enriched by the Generic Protocol translation and the Generic Protocol Core. Now it should be easy to understand how the Generic Exchange works.

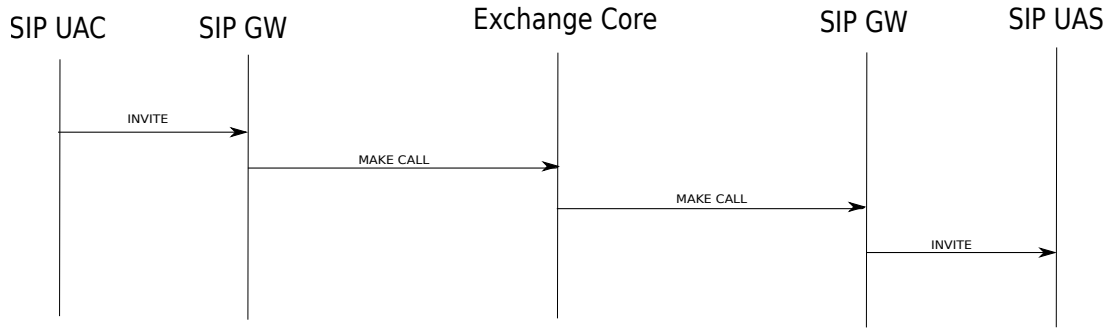


Figure 5.7: *SIP* to *SIP* Initial Invite message sequence chart

Caller at `10.0.0.139:5070` asks the Generic Exchange at `10.0.0.139:5060` to localize the callee and transmit the following *SIP* message (in the Listing 5.2) to him. At first, the incoming packet is received by `SIP transport`, parsed and saved into a *SIP* message which is internally represented by a record.

```
Internet Protocol Version 4, Src: 10.0.0.139 , Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5070, Dst Port: 5060
Session Initiation Protocol (INVITE)
INVITE sip:1016@Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKnbjmsugd
  Max-Forwards: 70
  To: <sip:1016@Nest>
  From: "Marek" <sip:1017@Nest>;tag=vrbod
  Call-ID: ielycgdbsgodvue@Nest
  CSeq: 790 INVITE
  Contact: <sip:1017@10.0.0.139:5070>
  Content-Type: application/sdp
  Allow: INVITE,ACK,BYE,CANCEL,OPTIONS,PRACK,REFER,NOTIFY,...
  Supported: replaces,norefersub,100rel
  User-Agent: Twinkle/1.4.2
  Content-Length: 305
Message Body
  Session Description Protocol not shown
```

Listing 5.2: SIP registration request

SIP peripheral then translates the *SIP* message into Generic Message and passes it to the Generic Exchange Core. Please note that `SIP 100` response to the caller was omitted from the MSC.

```
type : make_call
target : <<1016@Nest>>
caller : {1017, ielycgdbsgodvue@Nest, <<vrbod>>}
calee : {1016, ielycgdbsgodvue@Nest, <<>>}
upstreamRoute : []
downStreamRoute : [{{10.0.0.139}, 5070, z9hG4bKnbjmsugd}]
routeToRecord : [],
sequenceNum : 790,
TTL : 70
specificHeaders=[<<sdp not shown>>]
```

Listing 5.3: Generic Protocol make_call request

Core performs a lookup and finds callee the 1016 in its association table. It then appends a route, decreases a TTL and sends (listing 5.4) the Generic Message back to the *SIP* peripheral.

```
type : make_call
target : <<1016@Nest>>
caller : {1017, ielycgdbsgodvue@Nest, <<vrbod>>}
calee : {1016, ielycgdbsgodvue@Nest, <<>>}
upstreamRoute : []
downStreamRoute : [{{10.0.0.139}, 5070}, z9hG4bKnbjmsugd},{{Nest},5060},1RWwgA}]
routeToRecord : [],
sequenceNum : 790,
TTL : 69
specificHeaders=[<<sdp not shown>>]
```

Listing 5.4: Routed Generic Protocol make_call request

After peripherheral receives the Generic Message alongside next hop information, it translates it back into the *SIP* message and puts it on the wire utilizing the `transport`.

```
nternet Protocol Version 4, Src: 10.0.0.139 (10.0.0.139), Dst: 10.0.0.140
    (10.0.0.140)
User Datagram Protocol, Src Port: 5060, Dst Port: 5060
Session Initiation Protocol (INVITE)
  INVITE sip:1016@Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKnbjmsugd
  Via: SIP/2.0/UDP Nest:5060;branch=1RWwgA
  From: <sip:1017@Nest>;tag=vrbod
  To: <sip:1016@Nest>;undefined
  Call-ID: ielycgdbsgodvue@Nest
  CSeq: 790 INVITE
  Max-Forwards: 69
  Content-Length: 305
  Contact: <sip:1017@10.0.0.139:5060>
  Content-Type: application/sdp
Message Body
  Session Description Protocol not shown
```

Listing 5.5: Routed *SIP* registration request

5.3.1 Erlang role during implementation

Apart from its significant difference compare to the 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.

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

Listing 5.6: Example of Erlang `map` function as present in generic_exchange_sip_generic.erl

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:

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

Listing 5.7: Example of Erlang `foldr` function as present in generic_exchange_sip_generic.erl

As a huge advantage I consider a combination of Erlang high-level properties combined with OTP and message-passing concurrency. Trough implementation phase I encountered numerous errors in design (that could be preceden by making deeper and more patient 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 a debugger. A huge role in this fact play Erlang atoms which help to identify software parts or messages nicely and easily.

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 truly important. I caught myself many times thinking of what do I want to express. The ratio of writing time to this 'thinking' time is by my judgement higher than in arbitrary imperative language. Consequence of this is larger amout if code that stay and will not be edited in the future.

Chapter 6

Scenarios and case studies

In this chapter we will observe a simple call example of 2 *SIP* clients. As example clients, a pair of SIP soft-phones were chosen. Twinkle soft phone implementation was selected, in version 1.4.2.

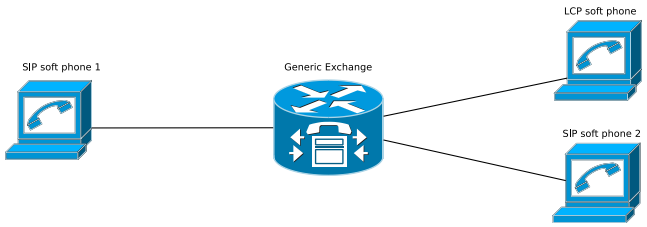


Figure 6.1: VoIP network configuration

6.1 SIP registration

In this section we will come trough an example of successful SIP registration and proper handling of method, which are not supported at this time. The registration process is initiated by a client. Client is configured with SIP registrar address and sends a registration request. You can see whole process in the figure 6.2. See the proper handling of the PUBLISH request which is not supported.

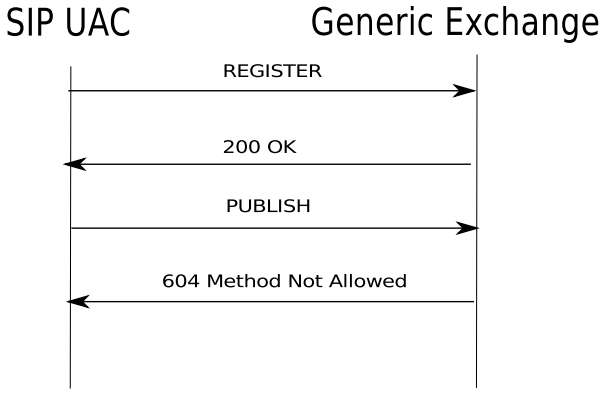


Figure 6.2: SIP registration message sequence chart

The exchange receives the following `SIP REGISTER` request from `10.0.0.139:5070`.

```
Internet Protocol Version 4, Src: 10.0.0.139 , Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5070, Dst Port: 5060
Session Initiation Protocol (REGISTER)
  REGISTER sip:Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKpwwgiysh
  Max-Forwards: 70
  To: "Marek" <sip:1017@Nest>
  From: "Marek" <sip:1017@Nest>;tag=bkdrf
  Call-ID: lfhyrvubwiyllyq@Nest
  CSeq: 83 REGISTER
  Contact: <sip:1017@10.0.0.139:5070>;expires=3600
  Allow: INVITE,ACK,BYE,CANCEL,OPTIONS,PRACK,REFER,NOTIFY,SUBSCRIBE,...
  User-Agent: Twinkle/1.4.2
  Content-Length: 0
```

Listing 6.1: SIP registration request

Currently there is no registration restriction implemented in the Generic Exchange. So there is no reason why exchange should reply with other response than 200.

```
Internet Protocol Version 4, Src: 10.0.0.139, Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5060, Dst Port: 5070
Session Initiation Protocol (200)
  SIP/2.0 200 OK
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKpwwgiysh
  From: <sip:1017@Nest>;tag=bkdrf
  To: <sip:1017@Nest>;tag=1dlNmj
  Call-ID: lfhyrvubwiyllyq@Nest
  CSeq: 83 REGISTER
  Max-Forwards: 69
  Content-Length: 0
  Contact: <sip:1017@10.0.0.139:5060>
  Expires: 3600
```

Listing 6.2: SIP registration response

Next a client attempts to publish its presence information using the `SIP Publish` request.

```
Internet Protocol Version 4, Src: 10.0.0.139 , Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5070, Dst Port: 5060
Session Initiation Protocol (PUBLISH)
  PUBLISH sip:1017@Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKtblwdnip
  Max-Forwards: 70
  To: "Marek" <sip:1017@Nest>
  From: "Marek" <sip:1017@Nest>;tag=utwfg
  Call-ID: erqzrumxltojpvt@Nest
  CSeq: 258 PUBLISH
  Content-Type: application/pidf+xml
  Event: presence
  Expires: 3600
  User-Agent: Twinkle/1.4.2
  Content-Length: 183
Message Body
  eXtensible Markup Language body not shown
```

Listing 6.3: SIP publish request

Since Generic Exchange does not support this method nor its equivalent is implemented in the Generic Protocol, a `405 Method Not Allowed` response is replied back to the client. This behaviour is in complete harmony with initial design.

```
Internet Protocol Version 4, Src: 10.0.0.139 , Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5060, Dst Port: 507)
Session Initiation Protocol (405)
  SIP/2.0 405 Method not allowed
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKtblwdnip
  From: "Marek" <sip:1017@Nest>;tag=utwfg
  To: "Marek" <sip:1017@Nest>
  Call-ID: erqzrumxltojpvt@Nest
  CSeq: 258 PUBLISH
  Max-Forwards: 70
  Content-Length: 183
  Content-Type: application/pidf+xml
  Expires: 3600
  Event: presence
  User-Agent: Twinkle/1.4.2
Message Body
  eXtensible Markup Language
```

Listing 6.4: SIP publish response

6.2 LCP registration

Next we will see the registration of *LCPCPv1* client (in the figure 6.3). There will not be any messages displayed since *LCPCPv1* is a binary protocol and messages are not easily readable. Instead, I will explain each message and the values that it carries.

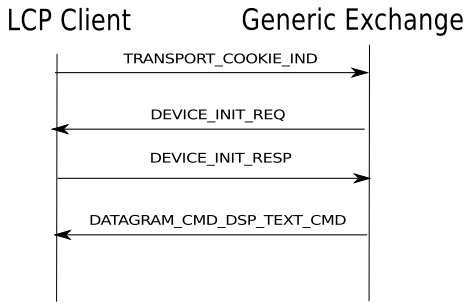


Figure 6.3: LCP registration message sequence chart

LCPCPv1 client initiates whole process by sending the `TRANSPORT_COOKIE_IND` message. The cookie is a secret value that serves as a registration protection. The cookie is supplied to *LCPCPv1* softphone on the command line. On the exchange part the value is hardcoded and is `cookie`. Also the `IP` the address and port of LCPCPv1 exchange service is hardcoded in the softphone and it is `127.0.0.1:4066`. The source code of the softphone is not at our disposal.

Generic Exchange responds with a `DEVICE_INIT_REQ`, asking client to associate. This message does not hold any important values.

Client as expected returns a `DEVICE_INIT_RESP` message which indicates that client is now associated. The message carries some basic information about the client devices such as the display, speaker or codecs. Now an association is completed.

Since LCPCPv1 client does not hold any information about itself, it is a good practice to inform the user that his LCPCPv1 device is registered. Generic Exchange then additionally sets a text on a client's display (as shown in the figure 6.4) utilizing the `DATAGRAM_CMD_DSP_TEXT_CMD` message about successfull association. The message carries information about display, message and where to place the message on the display.

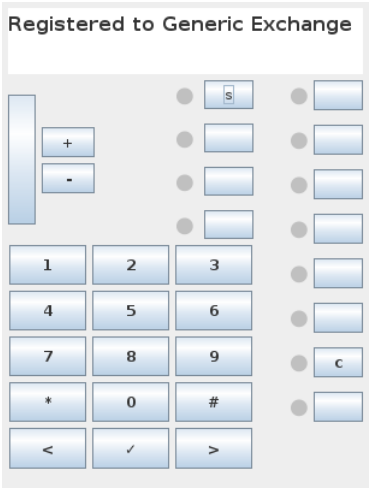


Figure 6.4: LCPCPv1 softphone after successfull registration

6.3 SIP to SIP simple call

The following message sequence chart represents a typical SIP to SIP call flow. Since the number of messages in this use case is quite large, only a few sample but important messages will be picked and displayed. A major difference compare to the traditional SIP Invite message sequence is the order of SIP `ACK` requests. Since Generic Protocol does not have concept of three-way handshake. The `ACK` are either dumped at arrival or generated after 200 response in INVITE context is retrieved. This behaviour is in perfectly with the design as explained earlier.

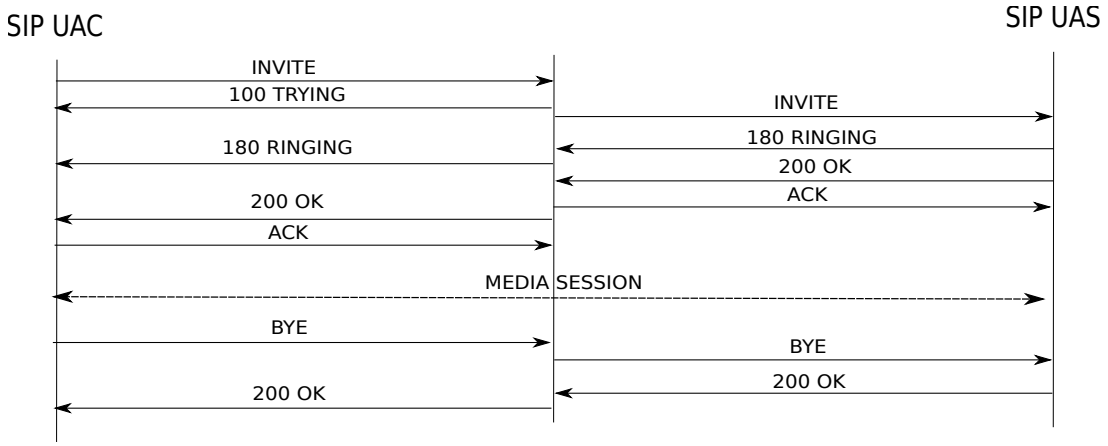


Figure 6.5: SIP to SIP simple call utilizing Generic Exchange

The SIP `INVITE` request displayed as received from the network at the SIP peripheral.

```
Internet Protocol Version 4, Src: 10.0.0.139 , Dst: 10.0.0.139
User Datagram Protocol, Src Port: 5070, Dst Port: 5060
Session Initiation Protocol (INVITE)
INVITE sip:1016@Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKnbjmsugd
  Max-Forwards: 70
  To: <sip:1016@Nest>
  From: "Marek" <sip:1017@Nest>;tag=vrbod
  Call-ID: ielycgdbsgodvue@Nest
  CSeq: 790 INVITE
  Contact: <sip:1017@10.0.0.139:5070>
  Content-Type: application/sdp
  Allow: INVITE,ACK,BYE,CANCEL,OPTIONS,PRACK,REFER,NOTIFY,...
  Supported: replaces,norefersub,100rel
  User-Agent: Twinkle/1.4.2
  Content-Length: 305
Message Body
  Session Description Protocol not shown
```

Listing 6.5: SIP registration request

The SIP `Invite` request after routing was performed. See that the `Via` header was added and `Contact` header field was changed. See section 5.2.1 for further information.

```
nternet Protocol Version 4, Src: 10.0.0.139 (10.0.0.139), Dst: 10.0.0.140
    (10.0.0.140)
User Datagram Protocol, Src Port: 5060, Dst Port: 5060
Session Initiation Protocol (INVITE)
  INVITE sip:1016@Nest SIP/2.0
  Via: SIP/2.0/UDP 10.0.0.139:5070;rport;branch=z9hG4bKnbjmsugd
  Via: SIP/2.0/UDP Nest:5060;branch=1RWwgA
  From: <sip:1017@Nest>;tag=vrbod
  To: <sip:1016@Nest>;undefined
  Call-ID: ielycgdbsgodvue@Nest
  CSeq: 790 INVITE
  Max-Forwards: 69
  Content-Length: 305
  Contact: <sip:1017@10.0.0.139:5060>
  Content-Type: application/sdp
Message Body
  Session Description Protocol not shown
```

Listing 6.6: Routed SIP registration request

6.4 Generic Exchange in contrast with Asterisk

An important part of this thesis was to compare Generic Exchange with already existing VoIP Exchange. I choose Asterisk as another implementation of the exchange software. Asterisk is an open source telephony switching and PBX service for Linux. It was created in 1999 by the Digium company. It us dual licensed under GPL[18] and proprietary software license to allow distribution of proprietary unpublished components. Asterisk was mainly chosen for its support of VoIP services regardless of underlying signaling protocol. Which is exactly what Generic Exchange does. This chapter will compare architectures of both systems.

Although features Asterisk holds, outnumbers Generic Exchange features by several categories, the initial principes in design could be compared. Since we already covered Generic Exchange design we will now cover some of important Asterisk design aspects and compare them. Asterisk represents a connection between arbitrary end point and the

exchange by something called the Channel. When an end point contacts Asterisk, the Channel is created. Each Channel consist of two parts. The first part is closely binded with the uderlying signaling technology used. The second part is more abstract one. It is mainly use to communicate with other Channels in Asterisk exchange. The first part is different for every underlying layer but the abstract one is always the same which allows exchange to connect phone call regardless of technologies in use.

The connection between two channel is called the Bridge. Since the Asterisk supports features like phone call recording, a voice or video media may flow trough the exchange. Bridge is a way to deliver a media between two channels. There are currently two methots of Bridging Channels:

1. A generic bridge, which is used regardless of underlying signaling technology, all media traffic flows trough the exchange. Although this is the most flexible method it is also the least efficient one.

2. A native bridge is used when bridged channels incorporate the same signaling technology. There is no need to use any abstraction at all. It is simpler and more efficient.

As a communication media within Asterisk exchange Frames are used. Frames flow trough bridges and they may carry a either a media or a signaling information. Here a close analogy with Erlang messages comes on to the light. Erlang utilizes a message passing concurrency which is built into its nature. But Asterisk does not features true message passing concurrency. In a heavily multi-threaded application, which Asterisk is, an additional layer of abstraction is added to the shared state concurrency model. Additional problem with message delivery synchronization need to be solved by using locks and semaphores. This opens a space for nasty errors. In Erlang we do not need to worry about any of these things which allows us to focus on what is truly important.

Although Asterisk is heavily multi-threaded application, it will never reach the parallelization level of an exchange written in Erlang due to the fact that it is using parallelization dependent on operating system threads. Operating system threads are much more heavyweight than Erlang threads. Although there is a abstraction layer build upon the operating system thread interface which after closer inspection strongly reminds me of a message passing concurrency concept, Erlang threads are still more easier to use. It is not only the threads by itself but the supporting message passing concurrency compliant debugger and others.

6.5 Summary

I can see an analogy in overall design of both Generic Exchange and Asterisk. Although they are quite different at the first sight, after a closer inspection they are similar. Asterisk also incorporates the concept of specific and generic parts. But instead of having a single generic core and a single gateway for every specific signaling protocol, it has a generic core and specific gateway for every currently active end device. This concept is one step ahead of Generic Exchange. Especially important is the decision which briding method will be taken. It has a huge effect on overall Asterisk performance but overall performance won't be highly impressive because the number of active threads during several hundreds active calls will be beyond the limits of any non-dedicated hardware.

Chapter 7

Conclusion

This text covered a design and implementation of the Generic Exchange. We discovered, that the Generic Exchange incorporates a high-level design which is well-thoughtout, efficient and quite elegant. In addition to this, the design of both the Generic Core and the peripherals was explained. In early chapters, we summarized the role of signaling protocols, we presented Erlang programming language, its framework Open Telecom Platform, declarative paradigm and how they fit into the Erlang concept. We sketched-out possible advantages of using Erlang and functional programming.

The Generic Exchange was successfully implemented, SIP is currently supported in full scope, LCPCPv1 is supported partially. Later during implementation phase, I realized the great importance of the design phase. Although the design was well-thoughtout, and was performed with patience and rigorousness, it was not perfect. I discovered a few minor errors and possible place in Generic Protocol for improvements. The implementation in semi-functional Erlang is truly rapid. It allows developer to focus on what is trully important and does not bother him with low-level programming expressions known from imperative languages. Developer then spends more time thinking about problem and not thinking of how a particular loop should look like. Thanks to the Dialyzer performing static analysis, software written in Erlang becomes safer, more reliable and durable. As contrasting VoIP exchange implementation was selected Asterisk. Well known and widely used VoIP exchange. Although its design is one step ahead of the Generic Exchange, its basic principes remains the same. The current Asterisk development team is trying to run in distributed environment over multiple nodes. This is another field where Erlang would be a great tool, since it has support of distributed computing in its very nature.

I'm glad that Unify s.r.o. provided me with this opportunity to write a Generic Exchange. It has been a great experience and I learned a lot. Personally I think the most important experience I gained during writing this thesis was the experience of designing such system. I (again) learned that patient design phase saves a lot of time in the oncomming implementation phase. My personal opinion on Erlang is that it is a tool capable of big things and I don't understand why it is not more widely used these days. It just saves so much time and effort...

Although the Generic Exchange was, by my opinion, designed pretty well, it needs a few extra hours (possibly days) to be perfect. Generic Protocol was in this case probably the weakest spot. Also for the Generic Exchange to be really useful, it would need more features to be added. Simple call is not really usefull to anyone. The implementation phase would be easier if a static analysis would be performed every time the exchange was compiled to see immediate results. This was not possible due to weak hardware, but question remains

if it would be possible at all especially in case of larger Erlang projects.

Apart from adding all missing features in the future, I would like to make Generic Exchange distributed. I think it will not be possible without redesigning whole Exchange. Although separating the exchange to one Generic Core and numerous specific parts is nice, it is still a centralized model. To make the exchange really distributed we would need a model where are numerous Generic Cores and numerous peripherals for each specific protocol. In ideal case creating both, the generic and the specific part for each active end point.

Bibliography

[1] JOE ARMSTRONG. *A History of Erlang.* ACM Press New York, 2007.

[2] DAVID BAZALA. *Telekomunikace a VoIP telefonie.* Technická literatura BEN CZ, 2006.

[3] ERICSON_AB. Erlang/OTP generic finite state machine behaviour. `http://www.erlang.org/doc/design_principles/fsm.html`, 2013.

[4] FRANCESCO CESARINI and SIMON THOMPSON. *Erlang Programming : A Concurrent Approach to Software Development.* O'Reilly Media, June 2009.

[5] FRED HERBERT. *Learn You Some Erlang for Great Good!* No Starch Press San Francisco, 2013.

[6] GERARD J. HOLZMANN. *Desing and Validation of Computer Protocols.* Prentice Hall USA, 1991.

[7] IETF NETWORK WORKING GROUP. SIP: Session Initiation Protocol. RFC 2543, 1999.

[8] IETF NETWORK WORKING GROUP. SIP: Session Initiation Protocol. RFC 3261, 2002.

[9] IETF NETWORK WORKING GROUP. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[10] ITU-T. Call signalling protocols and media stream packetization for packet-based multimedia communication systems. H. 225, 1996.

[11] ITU-T. Control protocol for multimedia communication. H. 245, 1996.

[12] ITU-T. Packet-based multimedia communications systems. H. 323, 1996.

[13] JAN WOZNIAK and ALEŠ KOCOUREK and TOMÁŠ KUKOSA. LCPCPv1 definition. Technical report, 2011.

[14] JOE ARMSTRONG. A post on erlang type system. Mailing List post.

[15] JOE ARMSTRONG. *Programming Erlang.* The Pragmatic Bookshelf USA, 2013.

[16] KEVIN WALLACE. *VoIP bez předchozích znalostí.* Computer Press CZ, 2006.

[17] MIRAN LIPOVAČA. *Learn You a Haskell for Great Good!* No Starch Press San Francisco, 2011.

[18] RICHARD STALLMAN. GNU General Public License, June 2007.