

HOW SMARTLY ERLANG USES DISTRIBUTED COMPUTING

SEMINAR REPORT

Submitted by

MANDEEP SINGH

(3323 & T120224226)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

TE COMPUTER ENGINEERING



ARMY INSTITUTE OF TECHNOLOGY

2016-17



ARMY INSTITUTE OF TECHNOLOGY, PUNE

DEPARTMENT OF COMPUTER ENGINEERING

CERTIFICATE

This is to certify that the Seminar titled

HOW SMARTLY ERLANG USES DISTRIBUTED COMPUTING

has been prepared and presented by

MANDEEP SINGH

(T120224226)

Of Third Year (Computer Engineering)

in partial fulfillment of requirement for the award of

*Degree of Bachelor of engineering in computer Engineering under the University of Pune
during the academic year 2016-17*

SEMINAR GUIDE

Prof. S.R. Dhore

HEAD OF DEPARTMENT

Prof. S.R. Dhore

ACKNOWLEDGEMENT

To acknowledge and thank every individual who directly or indirectly contributed to this venture personally, it would need associate excessive quantity of time. We are deeply indebted to many individual, whose cooperation made this job easier.

We avail this opportunity to express our gratitude to our friends and our parents for their support and encouragement throughout project. We feel it is as a great pleasure to express our deep sense of profound thank to Prof S.R. Dhore, who guided us at every step and encouraged us to carry out research on the topic.

MANDEEP SINGH (3323)

ABSTRACT

The construction of computer systems consisting of more than one computer is becoming more common. The complexity of such systems is higher than single computer systems.

To do this we have augmented the functional concurrent programming language Erlang with constructs for distributed programming. Distributed programs written in Erlang typically combine techniques for symbolic functional programming with techniques for distributed programming. In contrast to traditional imperative languages Erlang does not need interface description languages to specify the format of inter-processor messages in a heterogeneous network. This considerably simplifies distributed programming. Distributed Erlang is currently being employed in several large software projects within the Ericsson group and large messaging platforms like WhatsApp.

Erlang is type-less in the same sense as traditional logic languages, uses pattern matching for variable binding and function selection, has explicit mechanisms to create concurrent processes and advanced facilities for error detection and recovery.

OTP (Open Telecom Platform) of Erlang is very strong. The implementation of Erlang makes simple primitives fast and scalable, and makes effective use of modern multicore environment, eliminating the need for more complex mechanisms.

Erlang's concurrency is built upon the simple primitives of process spawning and message passing, and its programming style is built on the assumption that these primitives have a low overhead.

The processes created in Erlang are very light-weight, so very less time is required for creating and destroying the processes. The process creation time is very less (1micro-sec for creating over 2500 processes) as compared to other languages like JAVA (takes 300 micro-sec for creating small no. Of processes).

Keywords: distributed programming, concurrency, process creation, fault tolerance, functional programming, pattern matching, message passing

TABLE OF CONTENTS

CHAPTER NO.	TITLE	PAGE NO
	CERTIFICATE	ii
	ACKNOWLEDGEMENT	iii
	ABSTRACT	iv
	TABLE OF CONTENTS	v
	LIST OF FIGURES	vii
1	INTRODUCTION	1
	1.1 Background and History of Erlang	1
	1.2 Motivation	2
2	LITERATURE SURVEY	4
3	ARCHITECTURE	5
	3.1 Problem Domain	5
	3.2 Concurrency Oriented Programming	6
	3.2.1 Characteristics of COPLs	7
	3.2.2 Process Isolation	7
	3.2.3 Message Passing	8
	3.2.4 Protocols	8
	3.3 System Requirements	8
	3.4 Language Requirements	9
	3.5 Library Requirements	9
	3.6 Application Libraries	10
4	ERLANG	11
	4.1 Overview	11

	4.2 Example	11
	4.3 Sequential Erlang	13
	4.3.1 Data Structures in Erlang	13
	4.3.2 Variables	13
	4.3.3 Records	13
	4.4 Concurrent Programming	14
	4.4.1 Register	15
	4.5 Process Links and monitors	15
	4.6 Distributed Programming	15
5	Programming Techniques in Erlang	16
	5.1 Understanding the Problem	16
	5.1.1 event	17
	5.1.2 client	17
	5.1.3 processes inside event server	17
	5.2 event module	19
	5.3 The event server	19
	5.3.1 Hide your messages	22
	5.4 Test drive of program	23
6	PROGRAMMING FAULT TOLERANT SYSTEMS	25
7	OPEN TELECOM PLATFORM(OTP)	27
8	RESULTS	29
9	CONCLUSION	31
10	FUTURE EXPANSIONS	32
	REFERENCES	33

LIST OF FIGURES

Fig No.	Description	Page No.
1	Erlang program for calculation of factorial of number.	12
2	Output of factorial program	13
3	client and event server	17
4	functionalities between client and event server	18
5	execution of event module	20
6	event server module functionality	20
7	init function of event server	21
8	Message Handling on event server	21
9	Find event in orddict on event server	22
10	Terminate the event on event server	22
11	eveserv module	22
12	subscribe function of event server	23
13	add event to event server	23
14	output shell of event server	24
15	OTP System Architecture	28
16	Process creation time	29
17	Message passing time	30

CHAPTER - 1

INTRODUCTION

How will we tend to program systems that runs perfect within the presence of computer code errors, giant systems usually face errors that should be corrected at runtime of system, the system is often expected to behave during a cheap manner? This is the main issue that which programming language should be used for such large systems for reliable and fault tolerant performance.

Some of these essential necessities are consummated by Erlang artificial language et al. are consummated by the libraries written in Erlang. along the language and libraries facilitate in building a reliable package systems that work dependably in presence of package errors. Erlang is a functional language which belongs to the family of message passing languages. It is concurrent process-based language having strong isolation between processes. Erlang programming model makes extensive use of fail-fast process. In most of the programming languages, the resources are shared due to which a fault is propagated from one thread to another, and damage the internal consistency of system, but this is not the case with Erlang.

1.1 Background and History of Erlang

- 1982-85
 - Experiments with programming of medium victimization > twenty completely different languages. Conclusion: The language should be a awfully high level symbolic language so as to attain productivity gains! (Leaves US with: Lisp, Prolog, Parlog ...)
- 1985-86
 - Experiments with Lisp, Prolog, Parlog etc. Conclusion: The language must contain primitives for concurrency and error recovery, and the execution model must not have back-tracking
- 1988
 - ACS/Dunder Phase 1. Prototype construction of PABX functionality by external users Erlang escapes from the lab!
- 1989
 - ACS/Dunder Phase 2. Reconstruction of 1/10 of the complete MD-110 system. **Results:** >> 10 times greater gains in efficiency at construction compared with construction in PLEX! Further experiments with a fast implementation of Erlang.

- 1990
Erlang is presented at ISS'90, which results in several new users, e.g. Bellcore.
- 1991
Fast implementation of Erlang is released to users. Erlang is represented at Telecom'91. More functionality such as ASN1 - Compiler, graphical interface etc.
- 1992
A lot of new users, e.g. several RACE projects. Erlang is ported to VxWorks, PC, Macintosh etc. Three applications using Erlang are presented at ISS'92. The two first product projects using Erlang are started.
- 1993
- Distribution is extra to Erlang, that makes it doable to run a uniform Erlang system on a heterogeneous hardware. call to sell implementations Erlang outwardly. Separate organization in Ericsson began to maintain and support Erlang implementations and Erlang Tools.
- 1994-96
Only then was the language deemed mature enough to use in major projects with hundreds of developers, including Ericsson's broadband, GPRS, and ATM switching solutions. In conjunction with these projects, the OTP framework was developed and released in 1996. OTP provides a framework to structure Erlang systems, offering robustness and fault tolerance together with a set of tools and libraries.
- 1997-99
Ericsson made the decision to release Erlang as open source in December 1998 using the EPL license, a derivative of the Mozilla Public License. This was done with no budget or press releases, nor with the help of the corporate marketing department. In January 1999, the erlang.org site had about 36,000 page impressions.

1.2 Motivation

Distribution is additional to telephone unit, that creates it achievable to run the same telephone unit system on a heterogeneous hardware. decision to sell implementations telephone unit externally. Separate organization in Ericsson began to take care of and support telephone unit implementations and telephone unit Tools. Erlang highly supports hot swapping which means an engineer can change or modify the code without stopping the system.

Erlang has simple (most of the times) syntax, has inbuilt database, which is mnesia. Mnesia is a relational type database which exists to support erlang.

CHAPTER - 2

LITERATURE SURVEY

In a paper – “Distributed programming in Erlang” presented by Claes Wikstrom, he discussed about how Erlang is currently being employed in large sub-systems by Ericsson and other message platform based companies. The paper also defines Erlang, how it is different from other programming languages. Since the construction of large fault tolerant system is a extremely complex task, the system contains multiple cooperating CPUs, so we need a language that supports concurrency and distributed processing.

It is easier to implement distributed applications in Erlang rather than in other programming languages due to high level of abstraction in Erlang code. Erlang has a strong inbuilt standard library which contains functions which supports concurrency like spawn () etc.

In one other paper - “The Erlang Approach to Concurrent System Development” presented by Michael J. Lutz, he discusses about how the locking approaches like locks, semaphores and condition variables in languages like C and JAVA are error prone and how this problem can be solved by a functional programming language like Erlang because this language has immutable state. Erlang with roots of Prolog is used to develop robust, concurrent, fault tolerant communication switches by Ericsson Ltd (31ms downtime per year).

The paper “Concurrency Oriented Programming in Erlang” presented by one of the creator of Erlang, Joe Armstrong discusses about how the sequential programming languages created a notion that writing concurrent programs is very difficult task, this is not due to nature of concurrency itself, but rather to the way in which concurrency is implemented in these languages. This paper has short overview of concurrent programming followed by brief tutorial introduction to Erlang.

In many languages, the concurrency model of language is same as the concurrency model of the underlying operating system. This means that concurrency program for JAVA will change for different OS, and the operating system hardly supports any light weight processes, neither they allow large number of processes to occur concurrently. Joe Armstrong argues that concurrency should be the property of programming language and not the operating system.

CHAPTER - 3

ARCHITECTURE

This chapter specifies how to design a fault tolerant system.

3.1 Problem Domain

Earlier Erlang was developed for Telecom Switching Networks, however now its use is extended to modern messaging systems like WhatsApp and facebook. those structures are anticipated to function forever without any downtime. These are the basic necessities for telecom systems:

1. The system must be able to handle huge number of concurrent processes.
2. Actions must be performed at a certain point in a time or within a fix interval of time.
3. Systems may be distributed over several computers or several nodes at different places.
4. The system has functionality of controlling complex hardware.
5. The system exhibits complex functionality such as, feature interaction.
6. The system must be in continuous operation over years.
7. Software maintenance (reconfiguration, etc.) should be performed without stopping the system.
8. These are stringent qualities, and reliability requirements.
9. Fault tolerance both to hardware failures, and software errors, must be provided.

These requirements can be further expanded for messaging platforms as done by Joe Armstrong in his thesis as follows:

1. Concurrency: If we talk about a messaging platform, large number of people interact over a short interval of time, so it is very necessary that the system should be highly concurrent, and it should be able to handle this number of concurrent activities.
2. Soft real-time: In messaging services, the messages need to be delivered to specific user if receiver is online, so that time of sending a message and receiving back acknowledgment of delivered message should be real-time with minimum delay, these should be specifically monitored with the help of timers.

3. Distributed: The system should be designed in such a way that it is easily possible to transfer system from single node to multi node.
4. Hardware Interaction: The device drivers should be efficiently written and context switching between device drivers should be efficient.
5. Complex functionality: The functionality is complex in the sense that we may need to update the system or extent features of systems, this all should be done without stopping or rebooting the system.
6. Continuous operation: These systems are designed for continuous operation over a large span of time, so the operations like hardware and software maintenance should be done without stopping the system.
7. Quality requirements: The system must run in presence of errors and faults.
8. Fault Tolerance: Inside the system, the faults may generate, but the service of system should be acceptable even in presence of faults.

3.2 Concurrency Oriented Programming

The system that we tend to built, should be built up with concurrency as a vital half, thus we named it concurrency familiarized programming to distinguish it from different programming designs.

In Concurrency familiarized Programming the coincident structure of the program ought to follow the coincident structure of the appliance. it's notably suited to programming applications that model or move with the world.

The word concurrency refers to sets of events that happen at the same time. Thee world is coincident, and consists of an over-sized range of events, out of which many events can happen at the same time.

When we perform a straightforward action, like driving a automobile on a express-way, we take care of the actual fact that there could also be several hundreds of cars inside our immediate atmosphere, nevertheless we are able to perform the complicated task of driving an automobile, and avoiding of these potential hazards while not even pondering it. If we didn't have the power to research and predict the result of the many concurrent events we would live in great danger, then tasks like driving a automobile would not be possible. The actual fact that we do things that need to process huge amounts of parallel data suggests that we are equipped

with perceptual mechanisms which permit us to intuitively perceive concurrency while not consciously pondering it. Once it involves creating by mental acts things suddenly become inverted. Programming a consecutive chain of activities is viewed the norm, and in some sense is assumed of as being simple, whereas programming collections of coincident activities is avoided the maximum amount as do-able, and is mostly perceived as being troublesome.

3.2.1 Characteristics of Concurrency Oriented Programming Languages (COPLs)

1. COPLs must support processes. A process can be thought as a self contained virtual machine.
2. Processes operating on same computer must be strongly separated. A fault in one process should not adversely affect processing of another process, unless such process is explicitly programmed.
3. Each process must be identified by unique identifier which is called as Pid of the process.
4. There should be no shared state between processes. Process interact by sending messages. If you know Pid of the process, then you can send message to it.
5. Message passing is assumed to be unreliable with no guarantee of delivery.
6. It should be possible for one process to detect failure in another process. We should also know the reason of failure.

3.2.2 Process Isolation

The notion of isolation is central to understanding COP, and for the development of fault-tolerant software system. Two processes running on machine should work in manner such that they're running on two physically totally different machines.

Isolation have several consequences:

1. Processes have “share nothing” semantics. This is obvious as they are assumed to be running on two different machines.
2. The only way to pass messages between processes is through Message Passing.

3. Isolation implies message passing is asynchronous. If process communication is synchronous then a software error in receiver of message could indefinitely block the sender of message destroying the property of isolation.
4. Since nothing is shared, everything necessary to distributed computation must be copied. The only way to know if the message is delivered to receiver process is by sending a confirmation message back(often called acknowledgement of message).

3.2.3 Message Passing

Message Passing strictly obeys the following rules:

1. Message passing is assumed to be atomic which means a message is either delivered completely or not at all.
2. Message passing between a pair of processes is assumed to be ordered meaning that if a sequence of messages is sent and received between any pair of processes then the messages will be received in the same order they were sent.
3. Messages should not contain pointers to data structures contained within processes—they are only differentiated by Pids.

3.2.4 Protocols

Isolation of parts, and message passing between parts, is architecturally decent for safeguarding a system from consequences of a computer error, however it's not decent to specify the behavior of a system, nor, within the event of some reasonably failure to work out that element has unsuccessful.

To complete our programming model, we thus add an additional issue. Not solely can we want utterly isolated parts that communicate solely by message passing, but additionally we want to specify that communication protocols that are used between each part of components that communicate with one another.

3.3 System Requirements

To support a CO style of programming, and to make a system that can satisfy the requirements of a telecoms system we arrive at a set of requirements for the essential characteristics of a system.

The essential requirements are:

1. Concurrency — Our system must support concurrency. The machine effort required to form or destroy a synchronous method ought to be terribly tiny, and there ought to be no penalty for making massive numbers of synchronous processes.
2. Error encapsulation — Errors occurring in one process must not be able to effect working of other processes inside the system.
3. Fault detection — It must be possible to detect faults both locally and remotely.
4. Code upgrade — There should be methods inside the system which can be used to upgrade the code without stopping the system.
5. Stable storage — we need to store data in a manner which is not affected by system fault.

3.4 Language Requirements

The programming language which we use to program the system must have:

- Encapsulation primitives — there should be variety of mechanisms for limiting the consequences of error. The processes should be isolated in a way that working of one process don't damage any functionality of other process.
- Concurrency — the language should support a light-weight mechanism to form parallel process, and to send messages between the processes. Context switching between processes, and message passing, should be economical.
- Location transparency — If we know the Pid of a process then we should be able to send a message to the process.
- Dynamic code upgrade — It should be possible to dynamically change code in a running system.

3.5 Library Requirements

Language is not everything—a number of things are provided in the accompanying system libraries. The essential set of libraries routines must provide:

- Stable storage — the storage which is stable and not effected by any crash.

- Device drivers — these must provide a mechanism for communication with the different protocols outside the world.
- Code upgrade — this allows us to upgrade code in a running system without stopping.
- Infrastructure — for starting, and stopping the system, logging errors etc.

3.6 Application Libraries

Stable storage etc isn't provided as a language primitive in Erlang, however it is provided within the basic Erlang libraries. Having such libraries may be a pre-condition for writing any advanced application software system.

The OTP libraries give us an entire set of style patterns (called behaviors) for building fault-tolerant applications. The behaviors which are most popular are:

- supervisor — a supervision model.
- gen_server — a behaviour for implementing client-server applications.
- gen_event — a behavior used for implementing event handling software.
- gen_fsm — a behaviour used for implementing finite state machines.

CHAPTER - 4

ERLANG

Erlang belongs to the category of Message-oriented languages — message oriented languages give concurrency within the style of parallel processes. There are not any shared objects in any message-oriented language. Instead all interaction between processes is achieved by sending and receiving messages.

4.1 Overview

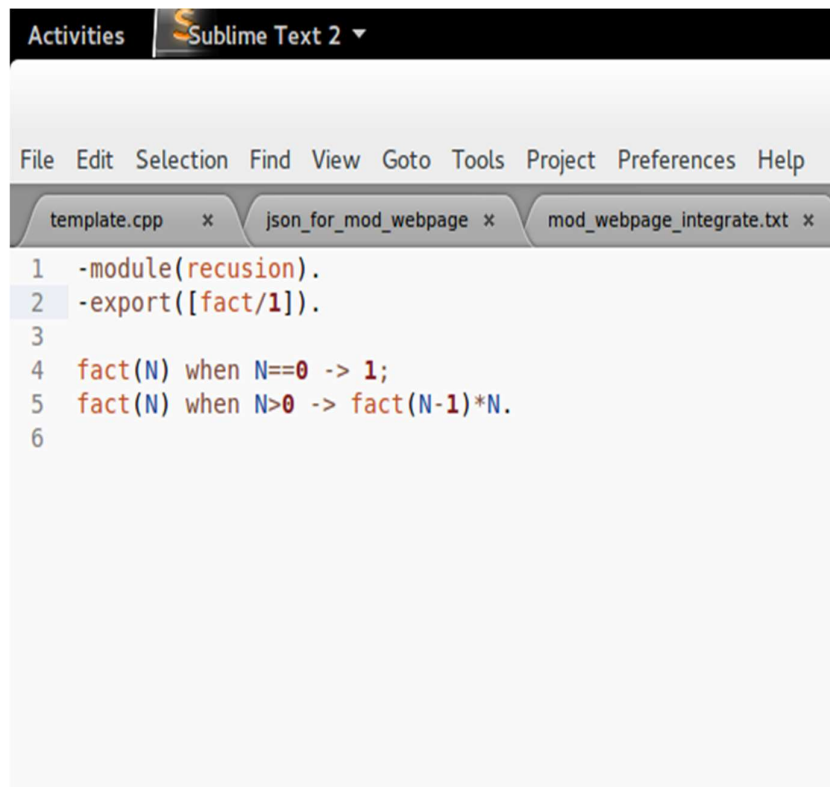
The Erlang view of the world can be summarized in the following statements:

- Everything in Erlang is process.
- Processes are strongly isolated.
- Processes creation and destruction is light weight operation.
- Message Passing is the only way for processes to interact.
- Processes are uniquely identified by their Pids.
- If you know Pid of the process, you can send message to it.
- Processes share no resources.
- Processes do what they are supposed to do or fail.

Viewed as a concurrent language, Erlang is very simple. Since there are no shared data structures, no monitors or synchronized methods etc there is very little to learn.

4.2 Example

Fig. shows a simple erlang program.



```
1 -module(recusion).
2 -export([fact/1]).
3
4 fact(N) when N==0 -> 1;
5 fact(N) when N>0 -> fact(N-1)*N.
6
```

Fig 1 Erlang program for calculation of factorial of number.

The program has following structure:

1. The program starts with a module definition (line 1) followed by export and input declarations and then by a number of functions.
2. The export declaration (line 2) says that the function fact/1 is to be exported from this module. The notation fact/1 means the function called fact which has one argument. The only functions which can be called from outside the module are those which are contained in the export list.
3. Line 4 and line 5 contains actual implementation of program, fact/1 is a recursive function which has base condition as when $N=0$, return 1 otherwise make a recursive call $\text{fact}(N)=\text{fact}(N-1)*N$.

```
Activities | Terminal ▾ Thu 4:32 PM
mandeep@mandeep-Inspiron-5447: ~
File Edit View Search Terminal Help
mandeep@mandeep-Inspiron-5447:~$ erl
Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.2 (abort with ^G)
1> cd("/home/mandeep/erlang").
/home/mandeep/erlang
ok
2> c(recursion).
{ok,recursion}
3> recursion:fact(5).
120
4> recursion:fact(10).
3628800
5> f().
ok
6> q().
ok
7> mandeep@mandeep-Inspiron-5447:~$ □
```

Fig 2. Output terminal of program recursion.erl

4.3 Sequential Erlang

4.3.1 Data Structures in Language

Like in any language, erlang also has variety of supported data structures namely Integers, Atoms, Floats, References, Binaries, Pids, Ports, Funs, Tuples, Lists, Strings, Records, etc,

4.3.2 Variables

Variables in Erlang are sequences of characters starting with a upper case letter and followed by a sequence of letters or characters or the “_” character. Variables in Erlang are either unbound, meaning they have no value, or bound, meaning that they have a value. Once a variable has been bound the value can never be changed.

4.3.3 Records

Records provide a method for associating a name with a particular element in a tuple. The problem with tuples is that when the number of elements in a tuple becomes large, it is difficult to remember which element in the tuple means what. The records are like structs in C.

4.4 Concurrent Programming

In Erlang, creation of parallel process is achieved by evaluating the spawn primitive. The expression

```
Pid = spawn(F)
```

Where F is a fun of arity zero creates a parallel process which evaluates F. Spawn returns a process identifier (Pid) which can be used to access the newly created process. The syntax `Pid ! Msg` sends the message Msg to Pid. The message can be received using the receive primitive, with the following syntax:

```
receive
```

```
Msg1 [when Guard1] ->
```

```
Expr_seq1;
```

```
Msg2 [when Guard2] ->
```

```
Expr_seq2;
```

```
...
```

```
MsgN [when GuardN] ->
```

```
Expr_seqN;
```

```
...
```

```
[; after TimeOutTime ->
```

```
Timeout_Expr_seq]
```

```
end
```

Msg1...MsgN are patterns. The patterns could also be followed by non-mandatory guards. once a message arrives at a process it's place into a mailbox belonging to that process. The next time

the process evaluates a receive statement the system can look within the mailbox and take a look at to match the primary item within the mailbox with the set of patterns contained within the current receive statement.

4.4.1 register

When we send a message to a method, we'd like to grasp the name of the process. this can be very secure, however somewhat inconvenient since all processes that wish to send a message to a given process ought to somehow get the name of that process. The expression:

Register (Name, Pid)

creates a global process, and associates the atom Name with the process identifier Pid. Thereafter, messages sent by evaluating Name ! Msg are sent to the process Pid.

4.5 Process Links and monitors

When one method within the system dies we might like alternative processes within the system to be told. There are two ways of doing this. we will use either a process link or a process monitor. Process links are used to cluster sets of processes in such the way that if a error happens in anyone of the processes then all the processes within the cluster get killed. Process monitors enable individual processes to monitor alternative processes within the system.

4.6 Distributed Programming

Erlang programs is simply ported from a uni-processor to a set of processors. Every complete and self-contained Erlang system is termed as node. A distributed application is developed and tested by running all the nodes within the application on one processor. Once the appliance works completely on different nodes that were assigned to a similar processor is moved to different nodes in an exceedingly network of distributed processors. With the exception of timing order all operations within the distributed system must precisely work the same approach as they worked within the single-node of system. Two primitives are needed for distributed processing:

- spawn(Fun,Node) — spawns a function Fun on the remote node Node.
- monitor(Node) — is used for monitoring the behaviour of an entire node.

monitor is analogous to link, the difference being that the controlled object is an entire node instead of a process.

CHAPTER - 5

PROGRAMMING TECHNIQUES IN ERLANG

The previous chapter was about Erlang, but not about how to program in Erlang. This chapter is about Erlang programming techniques.

5.1 Understanding a Problem

The software designed will allow us to do the following:

Add an event. Events contain a deadline (the time to warn at), an event name and a description.

- Show a warning when the time limit exceeded.
- Cancel an event by name.
- No persistent disk storage.
- Given that we have no persistent storage, we have to be able to update the code while it is running.
- The interaction with the software will be done via the command line.

Here's the structure of the program

Where the client, event server and x, y and z are all processes. Here's what each of them can do:

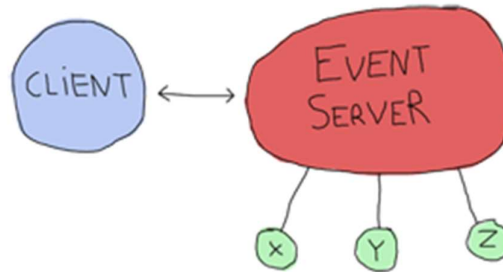


Fig. 3 client and event server

5.1.1 Event

- Accepts subscriptions from client.
- Send notifications from event processes from the subscribers.
- Accepts messages to add events.

- Can accept messages to cancel an event and kill the event processes
- Can be terminated by a client.

5.1.2 Client

- Subscribes to the event server and receive notifications as messages.
- Asks the server to add an event with all its details
- Asks the server to cancel an event
- Monitors the server (to notify if server goes down)
- Shuts down the event server if needed

5.1.3 x, y and z:

- Represent a notification waiting to ignite.
- Send a message to the event server when certain time limit is reached.
- Receive a cancellation request and terminate.

Here the software is written in this report, and it's assumed only one user will run it.

- Here's a more complex graph with all the possible messages:

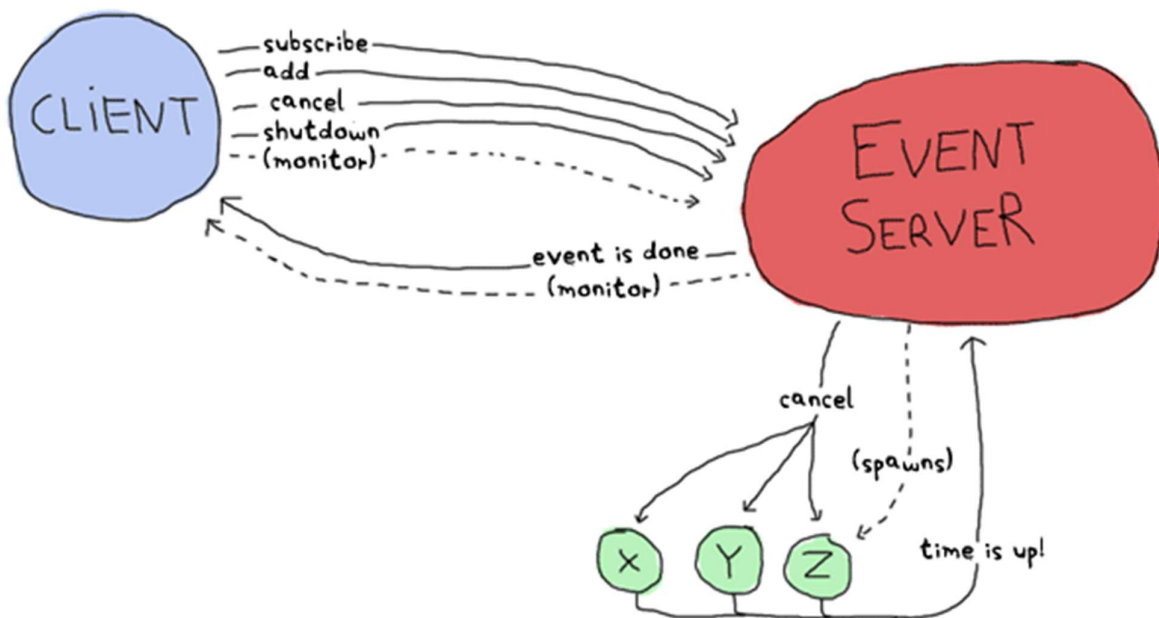


Fig. 4 Functionalities between client and event server

This represents each process we'll have. It should be noted that using one process per event to be reminded of is going to be overkill and one can't do this in real world applications. So we can use functions like `timer:send_after/2-3` to avoid spawning too many processes.

Project Directory Structure in Erlang

`ebin/`

`include/`

`priv/`

`src/`

The `ebin/` directory is where files will go once they are compiled. The `include/` directory is used to store `.hrl` files that are to be included by other applications; the private `.hrl` files are usually just kept inside the `src/` directory. The `priv/` directory is used for executables that might have to interact with Erlang, such as specific drivers and whatnot. Then the last one is the `src/` directory, where all `.erl` files stay.

In standard Erlang projects, this directory structure can vary a little. A `conf/` directory can be added for specific configuration files, `doc/` for documentation and `lib/` for third party libraries required for your application to run.

5.2 An event module

Get into the `src/` directory and start an `event.erl` module, which will implement the `x`, `y` and `z` events. How the addressing works, whether we use references or names, etc. Most messages will be wrapped under the form `{Pid, Message, Reference}`, where `Pid` is the sender and `Ref` is a unique message identifier to help know what reply came from who. If we were to send many messages before looking for replies, we would not know what reply went with what message without a reference.

```

17> c(event).
{ok,event}
18> f().
ok
19> event:start("Event", 0).
<0.103.0>
20> flush().
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).

```

Fig 5 execution of event module

5.3 The event server

```

-module(evserv).
-compile(export_all).

loop(State) ->
    receive
        {Pid, MsgRef, {subscribe, Client}} ->
            ...
        {Pid, MsgRef, {add, Name, Description, TimeOut}} ->
            ...
        {Pid, MsgRef, {cancel, Name}} ->
            ...
        {done, Name} ->
            ...
        shutdown ->
            ...
        {'DOWN', Ref, process, _Pid, _Reason} ->
            ...
        code_change ->
            ...
        Unknown ->
            io:format("Unknown message: ~p~n",[Unknown]),
            loop(State)
    end.

```

Fig. 6 Event server module functionality

Now, the server will need to keep two things in its state: a list of subscribing clients and a list of all the event processes it spawned. If you have noticed, the protocol says that when an event is done, the event server should receive {done, Name}, but send {done, Name, Description}. The idea here is to have as little traffic as necessary and only have the event processes care about what is strictly necessary.

```

init() ->
  %% Loading events from a static file could be done here.
  %% You would need to pass an argument to init telling where the
  %% resource to find the events is. Then load it from here.
  %% Another option is to just pass the events straight to the server
  %% through this function.
  loop(#state{events=orddict:new(),
              clients=orddict:new()}).

```

Fig 7. Init function of eveserv.erl

The first message is the one about subscriptions. We want to keep a list of all subscribers because when an event is done, we have to notify them. Also, the protocol above mentions we should monitor them. It makes sense because we don't want to hold onto crashed clients and send useless messages for no reason. Anyway, it should look like this:

```

{Pid, MsgRef, {subscribe, Client}} ->
  Ref = erlang:monitor(process, Client),
  NewClients = orddict:store(Ref, Client, S#state.clients),
  Pid ! {MsgRef, ok},
  loop(S#state{clients=NewClients});

```

Fig 8. Message handling

So what this section of loop/1 does is start a monitor, and store the client info in the orddict under the key *Ref*. The reason for this is simple: the only other time we'll need to fetch the client ID will be if we receive a monitor's EXIT message, which will contain the reference (which will let us get rid of the orddict's entry).

The next message to care about is the one where we add events. Now, it is possible to return an error status. The only validation we'll do is check the timestamps we accept. While it's easy to subscribe to the `{{Year, Month, Day}, {Hour, Minute, Seconds}}` layout, we have to make sure we don't do things like accept events on February 29 when we're not in a leap year, or any other date that doesn't exist

The next message defined in our protocol is the one where we cancel an event. Canceling an event never fails on the client side, so the code is simpler there. Just check whether the event is in the process' state record. If it is, use the `event:cancel/1` function we defined to kill it and send ok. If it's not found, just tell the user everything went right anyway -- the event is not running and that's what the user wanted.

```

{Pid, MsgRef, {cancel, Name}} ->
    Events = case orddict:find(Name, S#state.events) of
        {ok, E} ->
            event:cancel(E#event.pid),
            orddict:erase(Name, S#state.events);
        error ->
            S#state.events
    end,
    Pid ! {MsgRef, ok},
    loop(S#state{events=Events});

```

Fig 9. Find event in orddict

That should be it for most of the loop code. What's left is the set different status messages: clients going down, shutdown, code upgrades, etc. Here they come:

```

shutdown ->
    exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
    loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
    ?MODULE:loop(S);
Unknown ->
    io:format("Unknown message: ~p~n", [Unknown]),
    loop(S)

```

Fig 10. Terminate the event

The first case (shutdown) is pretty explicit. You get the kill message, let the process die. If you wanted to save state to disk, that could be a possible place to do it. If you wanted safer save/exit semantics, this could be done on every add, cancel or done message. Loading events from disk could then be done in the init function, spawning them as they come.

5.3.1 Hide your messages

Hiding messages! If you expect people to build on your code and processes, you must hide the messages in interface functions. Here's what we used for the eveserv module:

```

start() ->
    register(?MODULE, Pid=spawn(?MODULE, init, [])),
    Pid.

start_link() ->
    register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
    Pid.

terminate() ->
    ?MODULE ! shutdown.

```

Fig 11. Eveserv module

I decided to register the server module because, for now, we should only have one running at a time. If you were to expand the reminder use to support many users, it would be a good idea to instead register the names with the global module, or the gproc library.

The first message we wrote is the next we should abstract away: how to subscribe. The little protocol or specification I wrote above called for a monitor, so this one is added there. At any point, if the reference returned by the subscribe message is in a DOWN message, the client will know the server has gone down.

```
subscribe(Pid) ->
  Ref = erlang:monitor(process, whereis(?MODULE)),
  ?MODULE ! {self(), Ref, {subscribe, Pid}},
  receive
    {Ref, ok} ->
      {ok, Ref};
    {'DOWN', Ref, process, _Pid, Reason} ->
      {error, Reason}
  after 5000 ->
    {error, timeout}
  end.
```

Fig 12. Subscribe function of eveserv.erl

The next one is the event adding:

```
add_event(Name, Description, TimeOut) ->
  Ref = make_ref(),
  ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
  receive
    {Ref, Msg} -> Msg
  after 5000 ->
    {error, timeout}
  end.
```

Fig 13 add event to server

5.4 A test drive

Now, by going in your command line and running `erl -make`, the files should all be compiled and put inside the `ebin/` directory for you. Start the Erlang shell by doing `erl -pa ebin`. The `-pa <directory>` option tells the Erlang VM to add that path to the places it can look in for modules

```
1> evserv:start().
<0.34.0>
2> evserv:subscribe(self()).
{ok, #Ref<0.0.0.31>}
3> evserv:add_event("Hey there", "test", FutureDateTime).
ok
4> evserv:listen(5).
[]
5> evserv:cancel("Hey there").
ok
6> evserv:add_event("Hey there2", "test", NextMinuteDateTime).
ok
7> evserv:listen(2000).
[{done, "Hey there2", "test"}]
```

Fig 14 output terminal of event server

CHAPTER – 6

PROGRAMMING FAULT TOLERANT SYSTEM

This is the central question that how do we design a fault tolerant system, a quotation from a thesis which defines a fault tolerant system

To design and build a fault-tolerant system, you must understand how the system should work, how it might fail, and what kinds of errors can occur. Error detection is an essential component of fault tolerance. That is, if you know an error has occurred, you might be able to tolerate it by replacing the offending component, using an alternative means of computation, or raising an exception. However, you want to avoid adding unnecessary complexity to enable fault tolerance because that complexity could result in a less reliable system. —

Dugan quoted in Voas.

6.1 Programming Fault Tolerance

To make a system fault-tolerant we organise the software into a *hierarchy of tasks* that must be performed. The highest level task is to run the application according to some specification. If this task cannot be performed then the system will try to perform some simpler task. If the simpler task cannot be performed then the system will try to perform an even simpler task and so on. If the lowest level task in the system cannot be performed then the system will fail.

The distinctions between exceptions, errors and failures largely has to do with where in the system an abnormal event is detected, how it is handled and how it is interpreted. We trace what happens when an abnormal situation occurs in the system—this description is “bottom up” ie it starts at the point in time where the error is detected. As the tasks become simpler, the emphasis upon what operation is performed changes—we become more interested in protecting the system against damage than in offering full service. At all stages our goal is to offer an acceptable level of service though we become less ambitious when things start to fail.

- At the lowest level in the system the Erlang virtual machine detects an internal error—it detects a divide by zero condition, or a pattern matching error or something else. The important point about all of these detected conditions is that it is pointless to continue

evaluating code in the processes where the error occurred. Since the virtual machine emulator cannot continue, it does the only thing possible and throws an exception.

- At the next level, the exception may or may not be caught. The program fragment which traps the exception may or may not be able to correct the error which caused the exception. If the error is successfully corrected, then no damage is done and the process can resume as normal. If the error is caught, but cannot be corrected, yet another exception might be generated, which may or may not be trapped within the process where the exception occurred.
- The linked processes which receive these failure signals may or may not intercept and process these signals as if they were normal inter-process messages.

CHAPTER - 7

OPEN TELECOM PLATFORM (OTP)

The Open Telecom Platform(OTP) is a development system designed for building and running telecom applications. the OTP system is a so-called “middleware platform” designed to be run on top of a conventional operating system. The OTP system was developed internally at Ericsson. Most of the software was released into the public domain subject to the Erlang public license.

The OTP release include following components:

1. Compilers and development tools for Erlang.
2. Erlang run-time systems for a number of different target environments.
3. Libraries for a wide range of common applications.
4. A set of design patterns for implementing common behavioral patterns.
5. Educational material for learning how to use the system.
6. Extensive documentation.

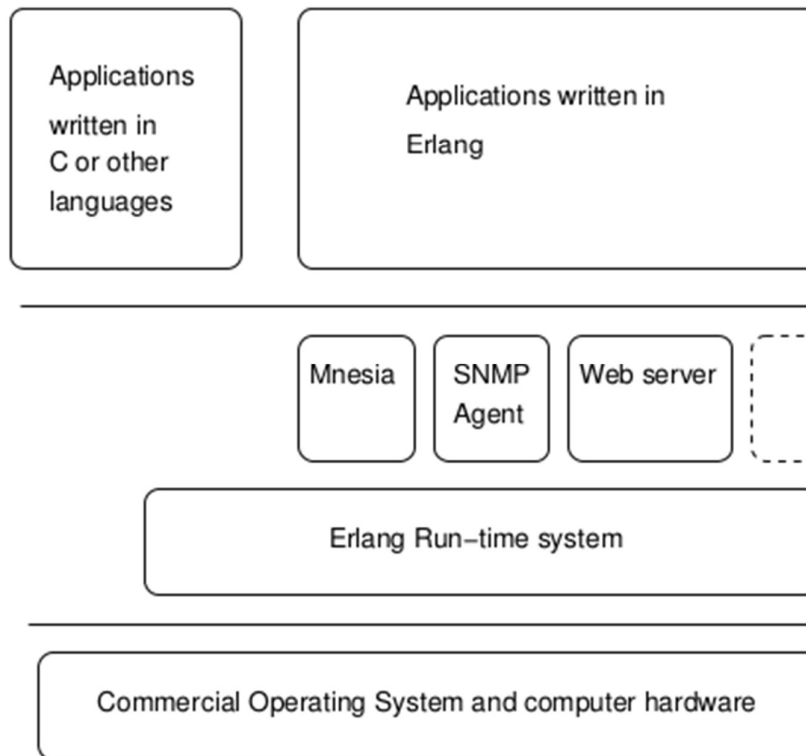


Fig 15 The OTP System Architecture

The Erlang run-time system is a virtual machine designed to run the intermediate code produced by the Erlang BEAM compiler. It also provides run-time support services for a native code Erlang compiler. The Erlang BEAM compiler replaced the original JAM compiler in 1998. The BEAM compiler compiles Erlang code into sequences of instructions for a 32-bit word threaded interpreter. The original JAM machine was a non-threaded byte code interpreter. For additional efficiency Erlang programs can be compiled to native code using the HIPE compiler developed at the University of Uppsala. Both interpreted BEAM and compiled code can be freely intermixed at a module level, ie, entire modules can be compiled to either BEAM or HIPE code, but code within an individual module cannot be intermixed. Both the beam and HIPE machines use common code in the Erlang run-time system for memory management, input/output, process management, and garbage collection etc.

CHAPTER – 8

RESULTS

A comparison between Erlang and other languages like JAVA, C# for time taken for process creation and message passing:

The process creation time in Erlang is very less as compared to other languages like C# and JAVA. The following graphs illustrates this:

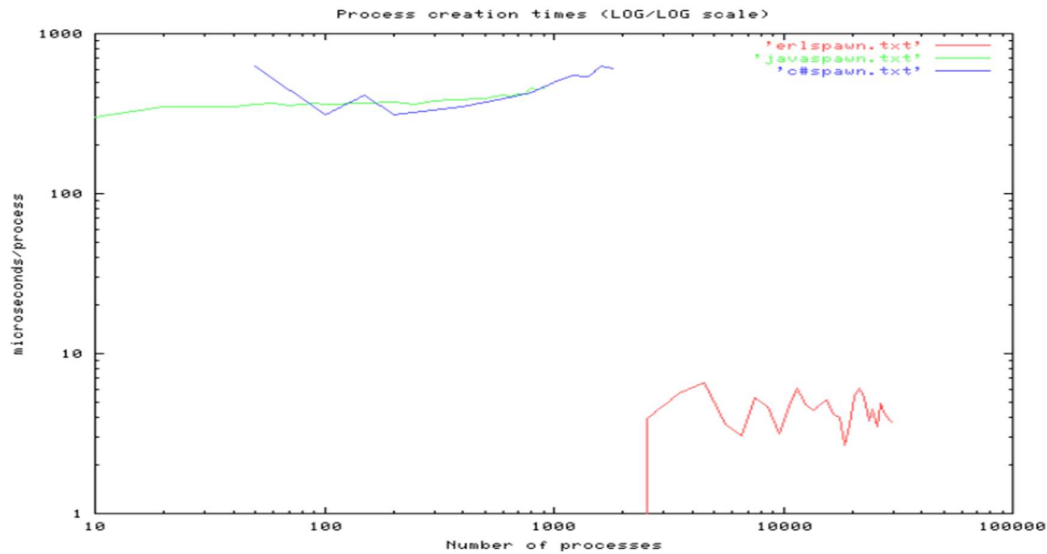


Fig. 16 Process Creation Time

Figure 16 shows the time needed to create the process as a function of total number of processes in system. It is observed that time taken to create an Erlang process is constant 1 us for up to 2,500 processes, thereafter it increases to 3 us for up to 30,000 processes. The performance of JAVA and C# is even worse, for small number of processes, it took about 300 us.

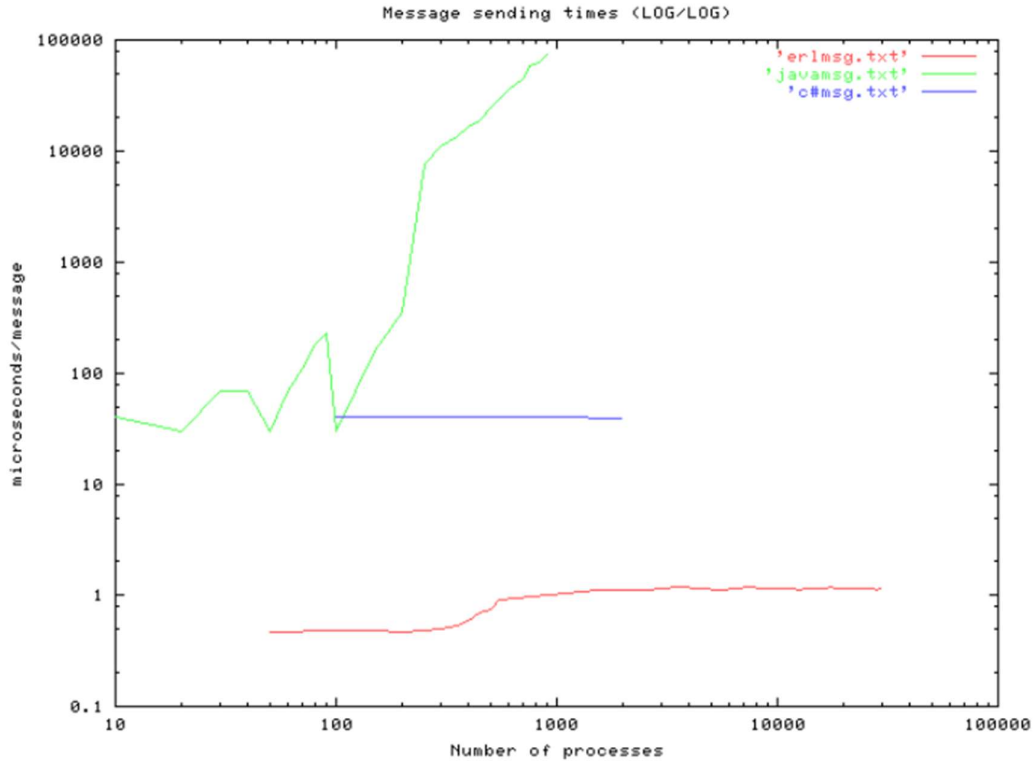


Fig 17. Message Passing Time

Figure 17 shows time to send the simple message between two concurrent processes running on the same machine as a function of total number of processes. We see that up to 30,000 processes the time to send message between two Erlang processes is about 0.8 us. For C#, it takes 50 us (For maximum 1800 processes) and for JAVA it's even worse it took 50 us for maximum of 100 processes.

CHAPTER - 9

CONCLUSION

- That Erlang and the associated technology works. Many people have argued that languages like Erlang cannot be used for full-scale industrial software development. Work on the AXD301 and on the Nortel range of products shows that Erlang is an appropriate language for these types of product development. The fact that not only are these products successful, but also that they are market leaders in their respective niches is also significant for example, WhatsApp, Ericsson, Facebook Messenger, etc.
- That programming with light-weight processes and no shared memory works in practice and can be used to produce complex large-scale industrial software.
- That it is possible to construct systems that behave in a reasonable manner in the presence of software errors.

CHAPTER – 10

FUTURE EXPANSIONS

In the earlier stage of Erlang, Erlang was developed by Ericsson for their purpose of switching networks, but the use of Erlang is extended for messaging platforms. It is extensively used by chatting platforms like WhatsApp, Facebook Messenger, Viber, etc. The complete backend of these softwares is implemented in Erlang and its application libraries and these are the market pioneers in their fields.

REFERENCES

- [1] Joe Armstrong (2003) Making reliable distributed systems in the presence of software errors: A Dissertation submitted to the Royal Institute of Technology.
- [2] Joe Armstrong. (2003) Concurrency Oriented Programming in Erlang : Distributed Systems Laboratory Swedish Institute of Computer Science.
- [3] Claes Wikstrom. Distributed Programming in Erlang: Computer Science Laboratory Ellemtel Telecommunications Science Laboratory.
- [4] Michael J. Lutz. The Erlang Approach to Concurrent System Development : Rochester Institute of Technology
- [5] Ghossoon M.W. Al-Saadoon, A Platform to Develop a Secure Instant Messaging Using Jabber Protocol, School of Computer and Communication Engineering, University Malaysia Perlis
- [6] Simon Fowler, An Erlang Implementation of Multiparty Session Actors, The University of Edinburgh Edinburgh, UK
- [7] Kenji RIKITAKE, Application Security of Erlang Concurrent System, Network Security Incident Response Group (NSIRG), NICT, Japan
- [8] Fred Hebert, Learn you some erlang for great good, <http://learnyousomeerlang.com/content>
- [9] Erlang official documentation, <http://erlang.org/doc/>
- [10] Jefferey Voas, Fault Tolerance (2001), IEEE Software