10th International Erlang/OTP User Conference

Stockholm, October 21, 2004

Proceedings

http://www.erlang.se/euc/04/

ERICSSON

Erlang Training and Consulting

corelatus

Mobile Arts

Sjöland & Thyselius

ERLANG

synapse mobile networks s.a.

Erlang/OTP User Conference 2004

Conference Programme

08.30 *Registration.*

Session I

09.00 **Building ground support equipments with Erlang.**
Jouni Rynö, Finnish Meteorological Institute.

09.30 **SERVAL: a VLAN software switch developed in Erlang.**
Alejandro García Castro, Igalia, and Juan José Sánchez Penas, University of A Coruña.

10.00 **An Erlang WTLS implementation.**
Johan Blom and Göran Oettinger, Mobile Arts.

10.30 *Coffee.*

Session II

11.00 **Learning Erlang and developing a SIP server/stack with 30k potential users.**
Fredrik Thulin, Stockholm university.

11.30 **Messaging with Erlang and Jabber.**
Mickaël Rémond, `http://www.erlang-projects.org/`

12.00 **Synapse DMC, liberating the mobile internet!**
Per Bergqvist, Synap.se.

12.30 *Lunch.*

Session III

14.00 **Dialyzer (DIscrepancy AnaLYZer of ERlang programs).**
Tobias Lindahl and Kostis Sagonas, Uppsala university.

14.30 **In the need of a design... reverse engineering Erlang software.**
Thomas Arts and Cecilia Holmqvist, IT university of Göteborg.

15.00 **Erlang's exception handling revisited.**
Richard Carlsson, University of Uppsala, Björn Gustavsson and Patrik Nyblom, Ericsson.

15.30 *Coffee.*

Session IV

16.00 **ErlGuten.**
Joe Armstrong, SICS.

16.30 **Proposal for an Erlang foundation.**
Mickaël Rémond, `http://www.erlang-projects.org/`

17.10 **Erlang/OTP R10B.**
Kenneth Lundin, OTP team, Ericsson.

17.30 *Close followed by bus transport to an ErLounge in downtown Stockholm!*

Demonstrations (during intermissions)

Mickaël Rémond demonstrates the **Jabber.**
Tobias Lindahl demonstrates the **Dialyzer.**
Tony Rogvall demonstrates **SSH** implemented in Erlang.

1(12)

Building GSEs with Erlang

Jouni Rynö, FMI / Space research

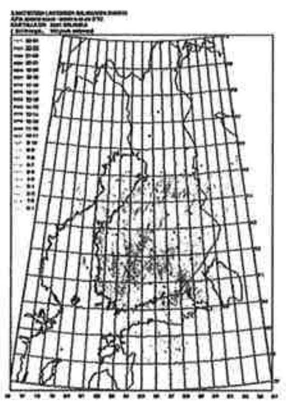
- history of Erlang at Finnish Meteorological Institute (FMI)
- CIDA and COSIMA instruments
- Ground Support Equipment (GSE)
  - **real-time telecommands (TC) and telemetry (TM)**
  - **offline telemetry (DB)**
  - **online telemetry (WWW)**

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 1

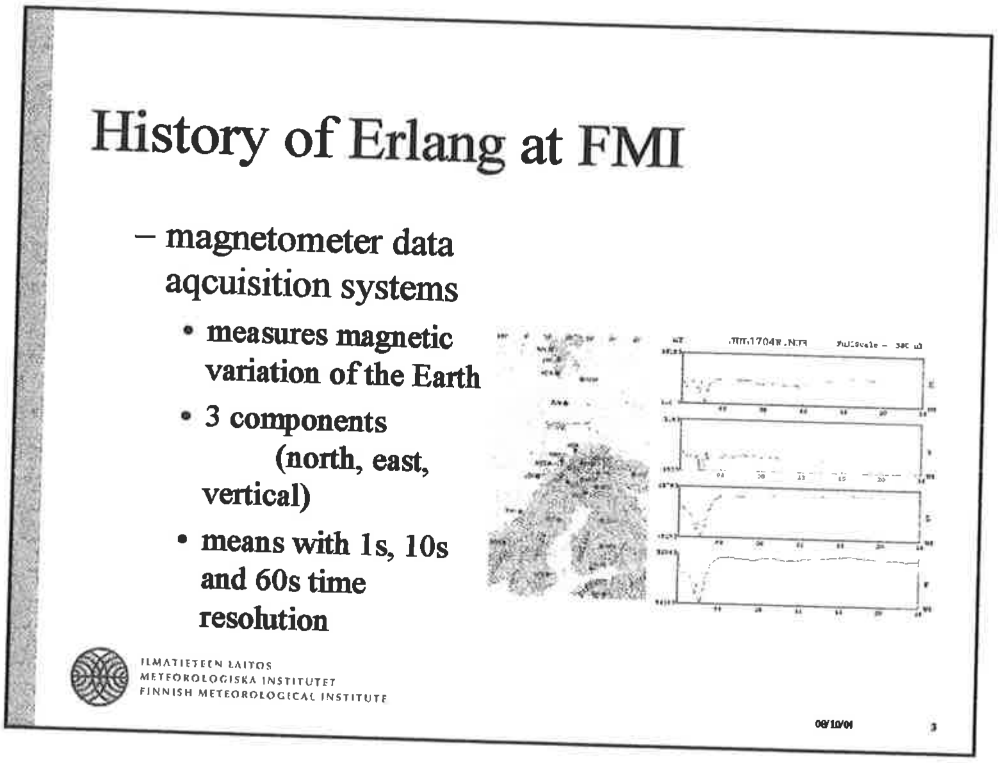
History of Erlang at FMI

- a small world
  - **a colleague had been a colleague of Joe in 1980's**
  - **and the colleague's brother was working at Ericsson**
- a broken lightning location system in June 1995
  - **from learning Erlang to a realtime graphical lightning display in 3 weeks...**



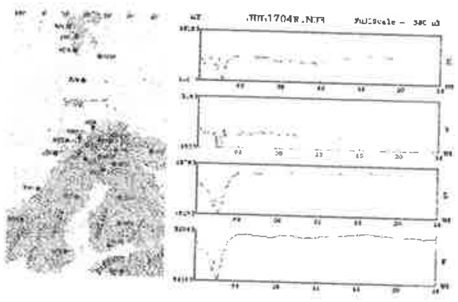
ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 2



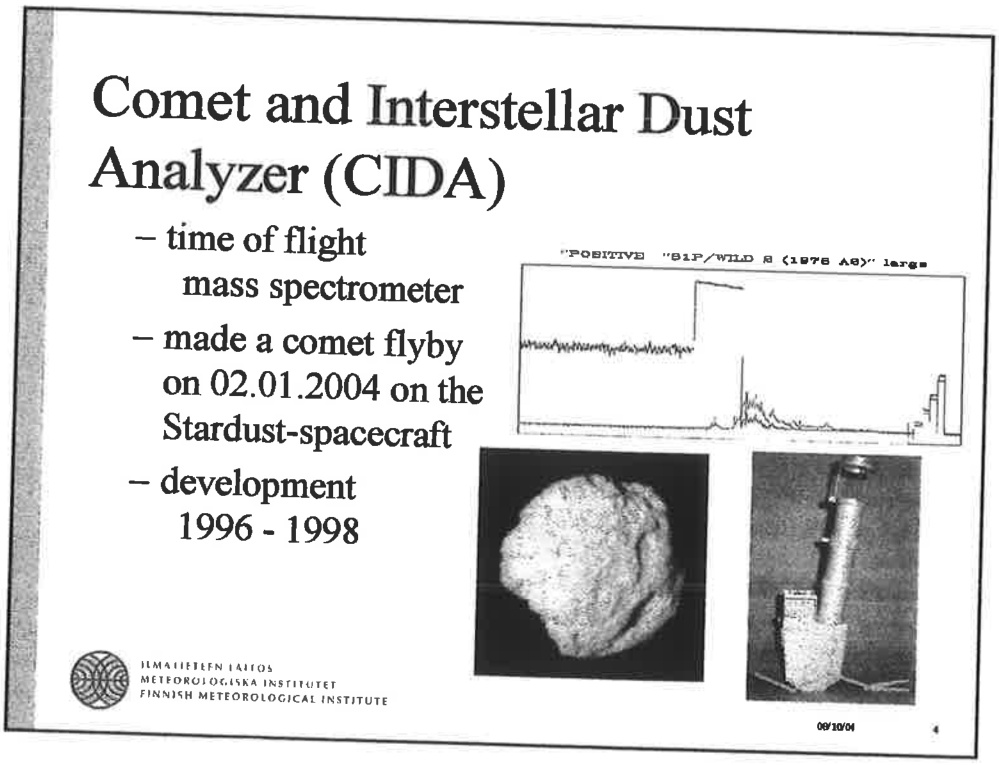
History of Erlang at FMI

- magnetometer data aqcuisition systems
  - measures magnetic variation of the Earth
  - 3 components (north, east, vertical)
  - means with 1s, 10s and 60s time resolution



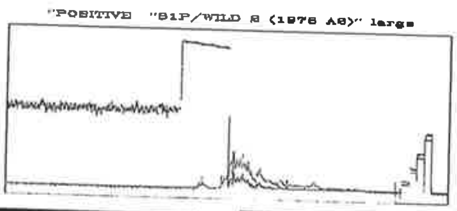
ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

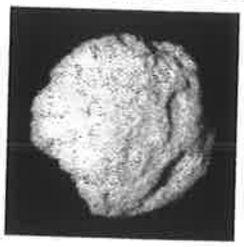
08/10/04 3

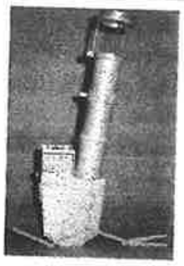


Comet and Interstellar Dust Analyzer (CIDA)

- time of flight mass spectrometer
- made a comet flyby on 02.01.2004 on the Stardust-spacecraft
- development 1996 - 1998





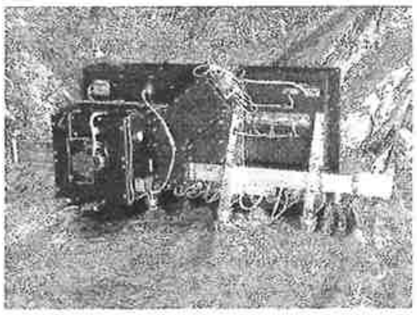


ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

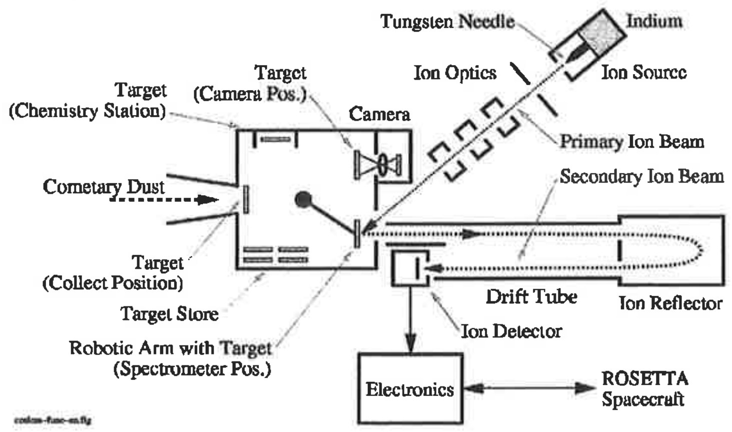
08/10/04 4

Cometary Secondary Ion Mass Analyzer (COSIMA)

- mass spectrometer
- onboard Rosetta, ESA's spacecraft to study the comet 67 P/Churyumov-Gerasimenko
- measurements 2014 - 2015

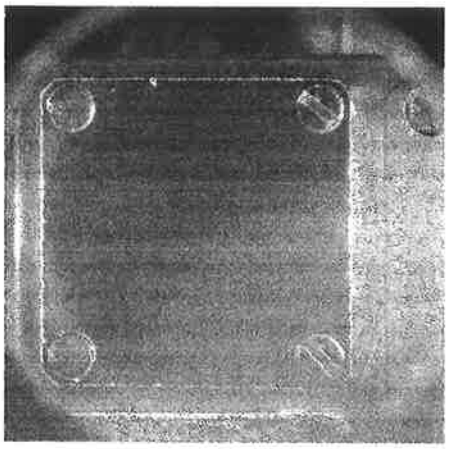


Cometary Secondary Ion Mass Analyzer (COSIMA)



Cometary Secondary Ion Mass Analyzer (COSIMA)

- 72 dust targets, each 10*10 mm size
- camera to detect 10-100 µm size particles
- motor system to move targets around with 4 µm accuracy
- 80 µm In ion beam

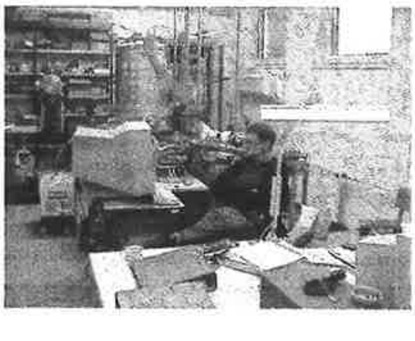


ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 7

Ground Support Equipment

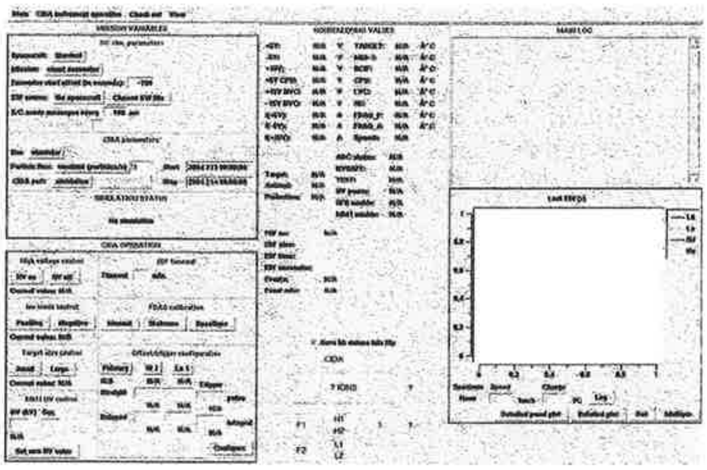
- simulates S/C electrical interface during the development
- provides telecommand generation and telemetry analysis
- as in the Erlang book!



ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 8

User interface



User interface 2

- user interface written in Tcl/Tk
  - button/menu interface fine for simple and often used interactive operations
  - complicated commanding needs a scripting language
  - BLT gives nice 2D graphs
- Erlang talks with Tcl via modified ewish

H/W interface

- selfmade PCI-card for S/C interface(linux driver with IDL c-server)
- power supply
- analog to digital PCI-card for current, temperature and vacuum monitoring

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

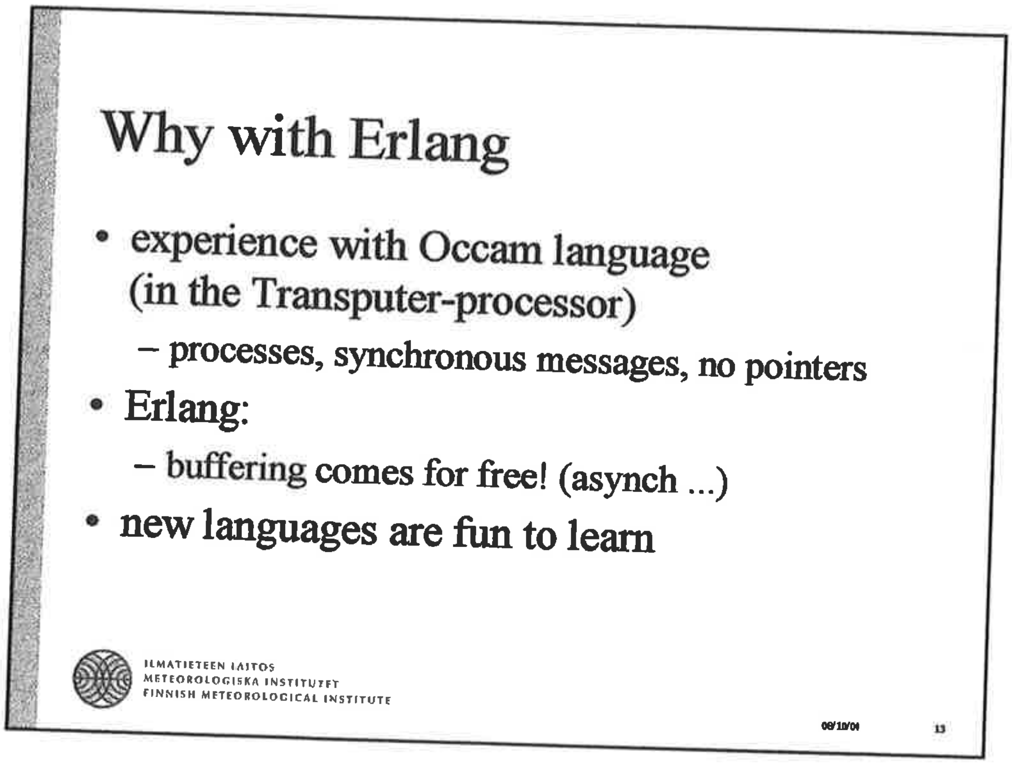
08/10/04 11

Why not Erlang

- 1996:
  - no binary syntax
  - no hex format
  - not open source
  - not sure about language future
- 2004:
  - few know about the language

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 12

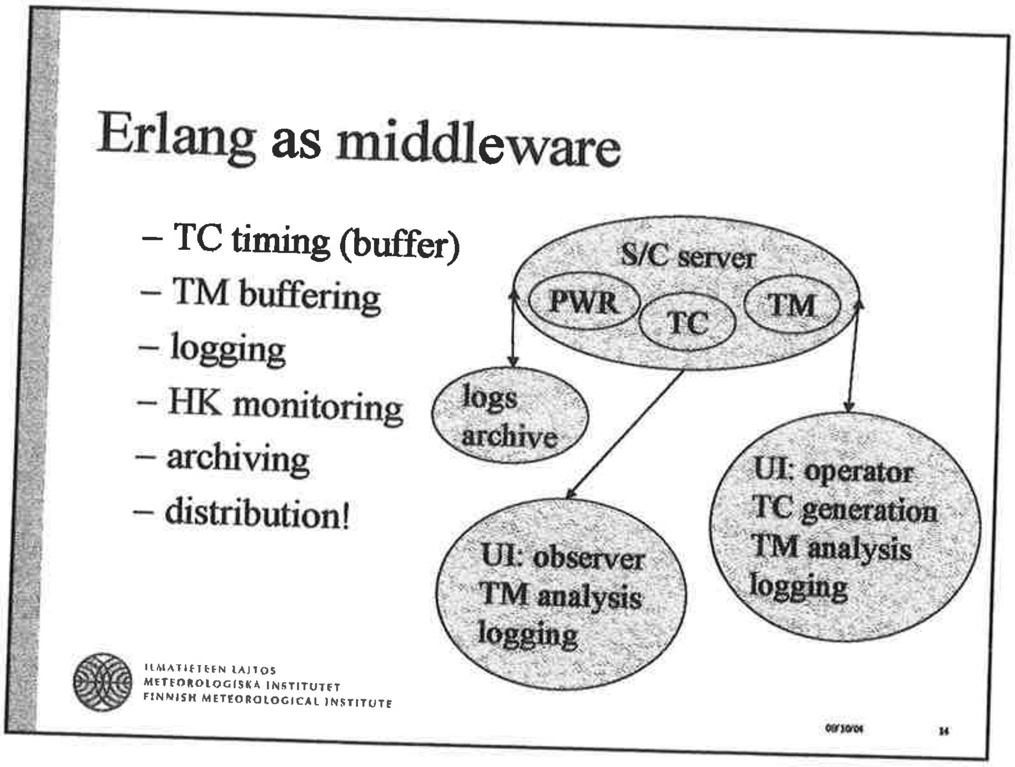


Why with Erlang

- experience with Occam language (in the Transputer-processor)
  - processes, synchronous messages, no pointers
- Erlang:
  - buffering comes for free! (asynch ...)
- new languages are fun to learn

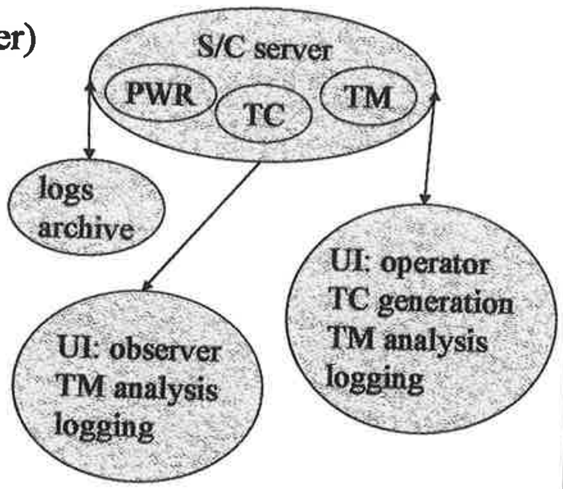
ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 13



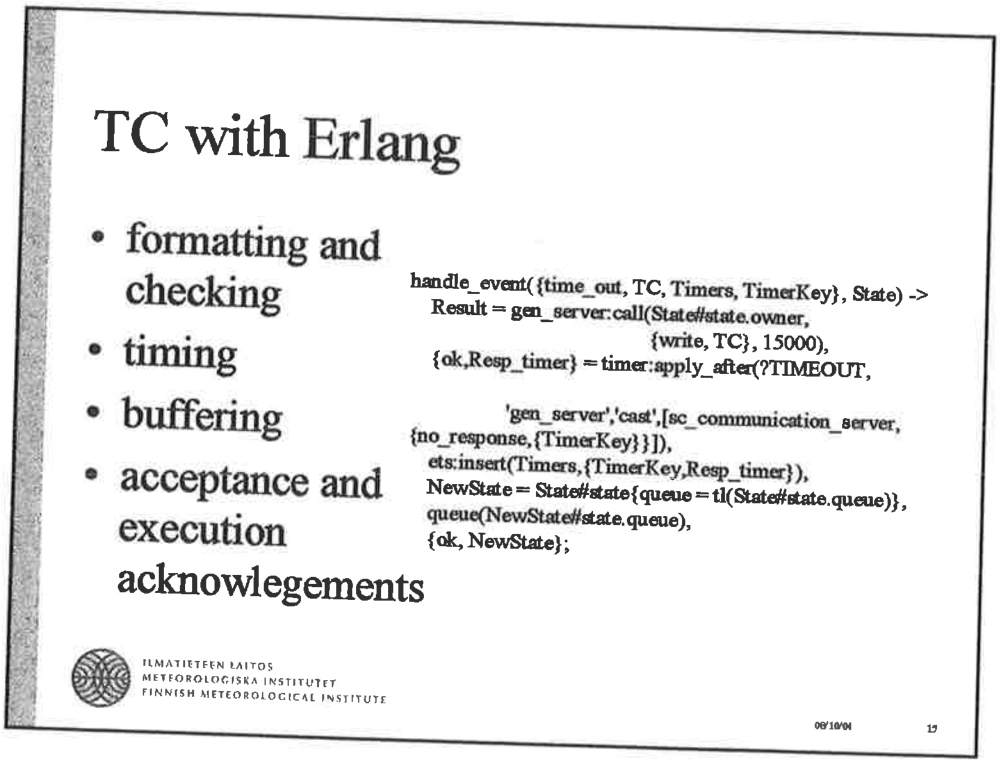
Erlang as middleware

- TC timing (buffer)
- TM buffering
- logging
- HK monitoring
- archiving
- distribution!


S/C server
PWR
TC
TM
logs archive
UI: observer TM analysis logging
UI: operator TC generation TM analysis logging

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 14



TC with Erlang

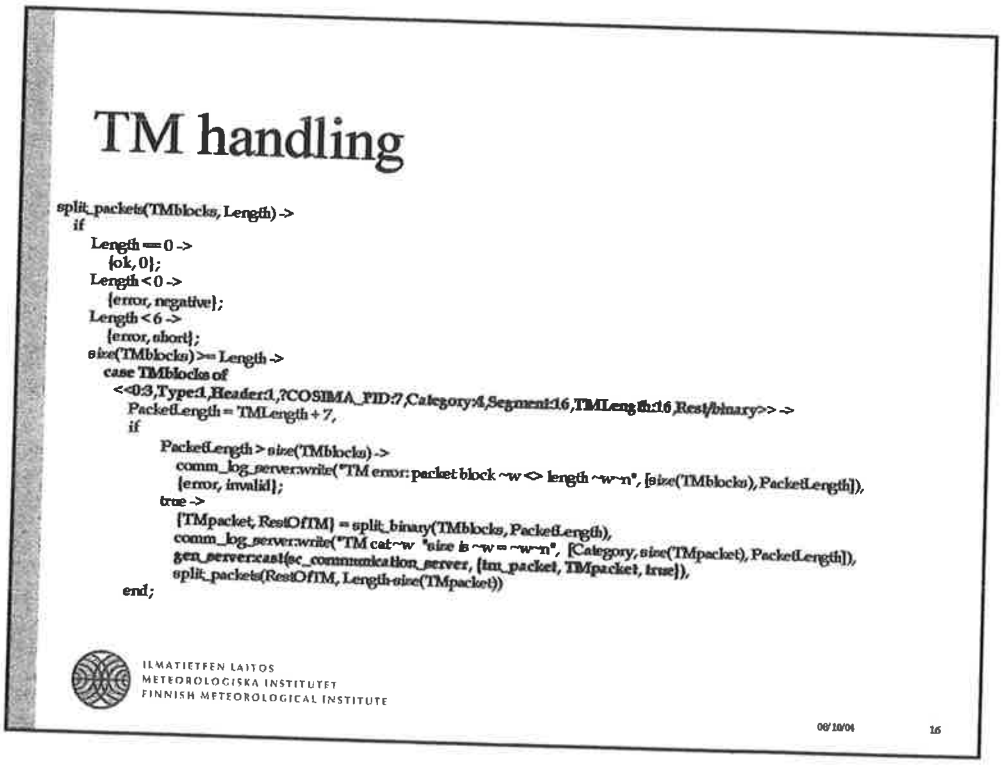
- formatting and checking
- timing
- buffering
- acceptance and execution acknowlegements

```
handle_event({time_out, TC, Timers, TimerKey}, State) ->
    Result = gen_server:call(State#state.owner,
                             {write, TC}, 15000),
   {ok,Resp_timer} = timer:apply_after(?TIMEOUT,

            'gen_server','cast',[sc_communication_server,
{no_response,{TimerKey}}]),
   ets:insert(Timers,{TimerKey,Resp_timer}),
   NewState = State#state{queue = tl(State#state.queue)},
   queue(NewState#state.queue),
   {ok, NewState};
```

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 15

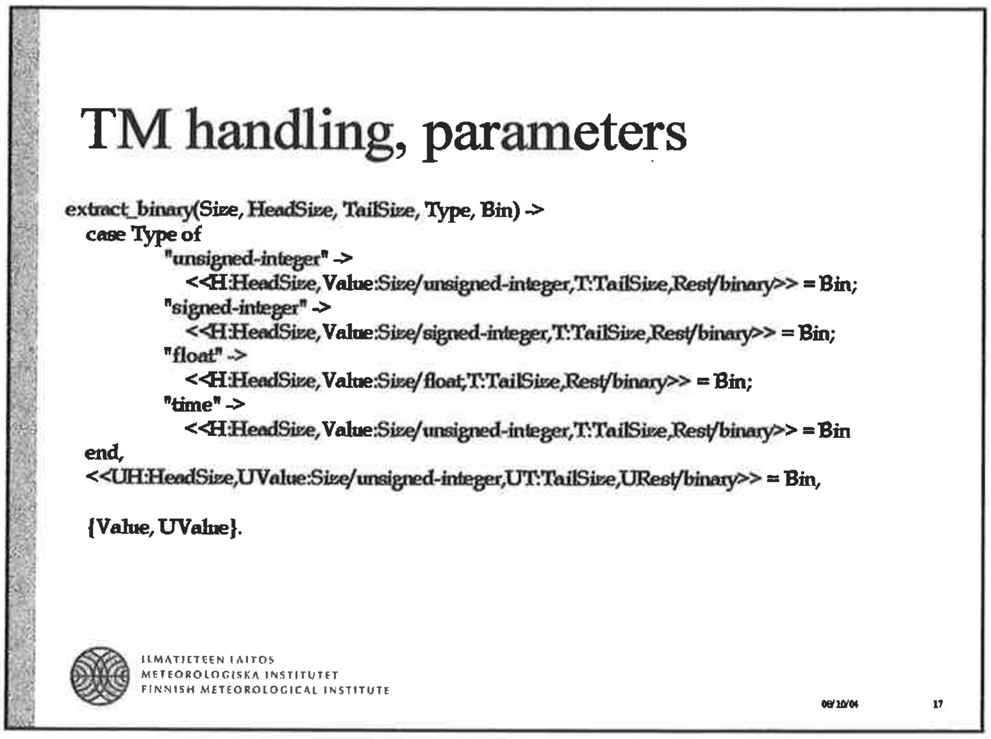


TM handling

```
split_packets(TMblocks, Length) ->
  if
    Length == 0 ->
      {ok, 0};
    Length < 0 ->
      {error, negative};
    Length < 6 ->
      {error, short};
    size(TMblocks) >= Length ->
      case TMblocks of
        <<0:3,Type:1,Header:1,?COSIMA_PID:7,Category:4,Segment:16,TMLength:16,Rest/binary>> ->
          PacketLength = TMLength + 7,
          if
              PacketLength > size(TMblocks) ->
                comm_log_server:write("TM error: packet block ~w <> length ~w~n", [size(TMblocks), PacketLength]),
                {error, invalid};
              true ->
                {TMpacket, RestOfTM} = split_binary(TMblocks, PacketLength),
                comm_log_server:write("TM cat~w  "size is ~w = ~w~n", [Category, size(TMpacket), PacketLength]),
                gen_server:cast(sc_communication_server, {tm_packet, TMpacket, true}),
                split_packets(RestOfTM, Length-size(TMpacket))
        end;
```

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 16



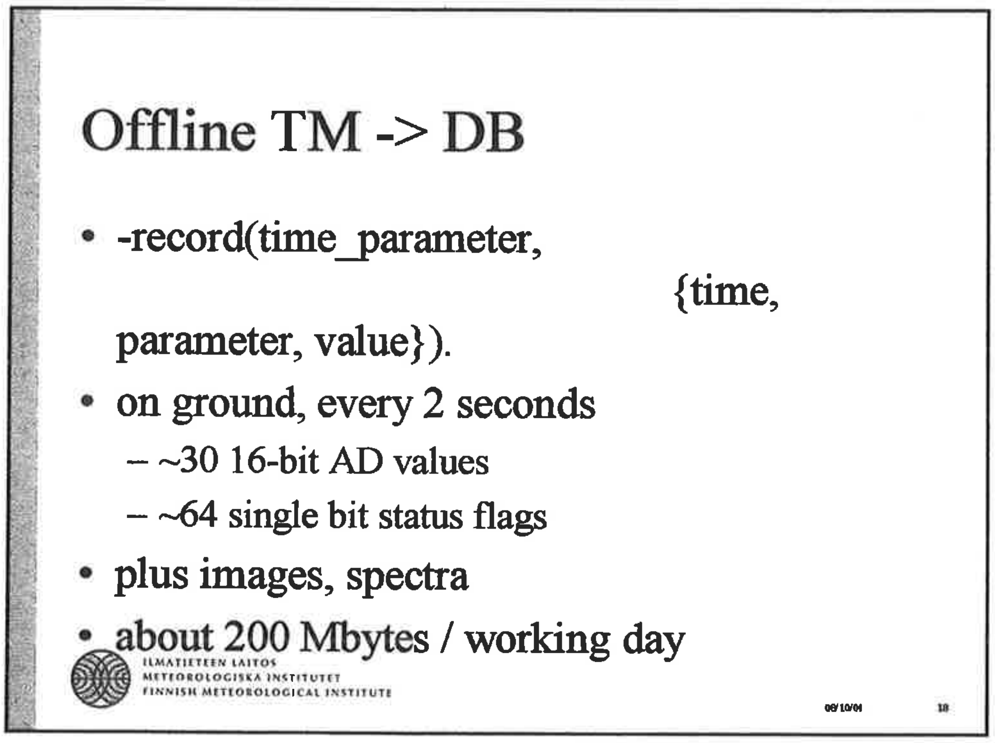
TM handling, parameters

```
extract_binary(Size, HeadSize, TailSize, Type, Bin) ->
  case Type of
          "unsigned-integer" ->
             <<H:HeadSize,Value:Size/unsigned-integer,T:TailSize,Rest/binary>> = Bin;
          "signed-integer" ->
             <<H:HeadSize,Value:Size/signed-integer,T:TailSize,Rest/binary>> = Bin;
          "float" ->
             <<H:HeadSize,Value:Size/float,T:TailSize,Rest/binary>> = Bin;
          "time" ->
             <<H:HeadSize,Value:Size/unsigned-integer,T:TailSize,Rest/binary>> = Bin
  end,
  <<UH:HeadSize,UValue:Size/unsigned-integer,UT:TailSize,URest/binary>> = Bin,

  {Value, UValue}.
```

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 17



Offline TM -> DB

- -record(time_parameter, {time, parameter, value}).
- on ground, every 2 seconds
  - ~30 16-bit AD values
  - ~64 single bit status flags
- plus images, spectra
- about 200 Mbytes / working day

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 18

Mnesia

- 200 Mbyte / day -> disk_only_copies
- for faster searching, use fragmented tables with a special hash module
  - key is MJD2000, decimal day since 01.01.2000
  - key_to_frag_number(State, Key) when record(State, hash_state) ->
    IntKey = trunc(Key).

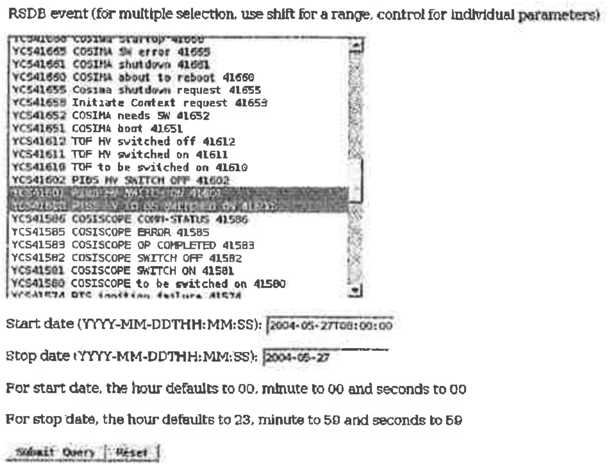
ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

Online access -> yaws

- DB frontend for the scientists
  - housekeeping data
  - events, images, spectra
  - target history
  - instrument status
- document repository
- wiki

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

Yaws



RSDB event (for multiple selection, use shift for a range, control for individual parameters)

```
YCS41666 COSIMA startup 41666
YCS41665 COSIMA SW error 41665
YCS41661 COSIMA shutdown 41661
YCS41660 COSIMA about to reboot 41660
YCS41655 Cosima shutdown request 41655
YCS41653 Initiate Context request 41653
YCS41652 COSIMA needs SW 41652
YCS41651 COSIMA boot 41651
YCS41612 TOF HV switched off 41612
YCS41611 TOF HV switched on 41611
YCS41610 TOF to be switched on 41610
YCS41602 PIDS HV SWITCH OFF 41602
YCS41601 PIDS HV SWITCH ON 41601
YCS41600 PIDS HV to be switched on 41600
YCS41586 COSISCOPE COMM-STATUS 41586
YCS41585 COSISCOPE ERROR 41585
YCS41583 COSISCOPE OP COMPLETED 41583
YCS41582 COSISCOPE SWITCH OFF 41582
YCS41581 COSISCOPE SWITCH ON 41581
YCS41580 COSISCOPE to be switched on 41580
YCS41574 RTS ignition failure 41574
```

Start date (YYYY-MM-DDTHH:MM:SS): 2004-05-27T08:00:00

Stop date (YYYY-MM-DDTHH:MM:SS): 2004-05-27

For start date, the hour defaults to 00, minute to 00 and seconds to 00

For stop date, the hour defaults to 23, minute to 59 and seconds to 59

Submit Query | Reset

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

08/10/04 21

Yaws

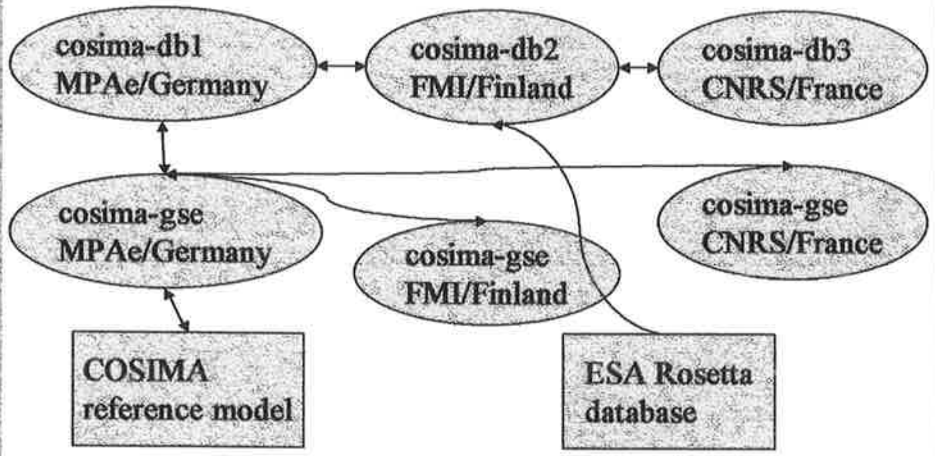
- define queries to the DB

```
Selection = [{{time_parameter,'$1','$2','$3'},
      [{'<',{const,1608.00},'$1'},
       {'=<','$1',{const,1609.00}},
       {'=:=','$2',{const,"NCSA1100"}},
       {'orelse',{'=:=',{const,4.15650e+4},'$3'},
                 {'=:=',{const,4.16000e+4},'$3'},
                 {'=:=',{const,4.16010e+4},'$3'}}],
      ['$_']}]
```

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

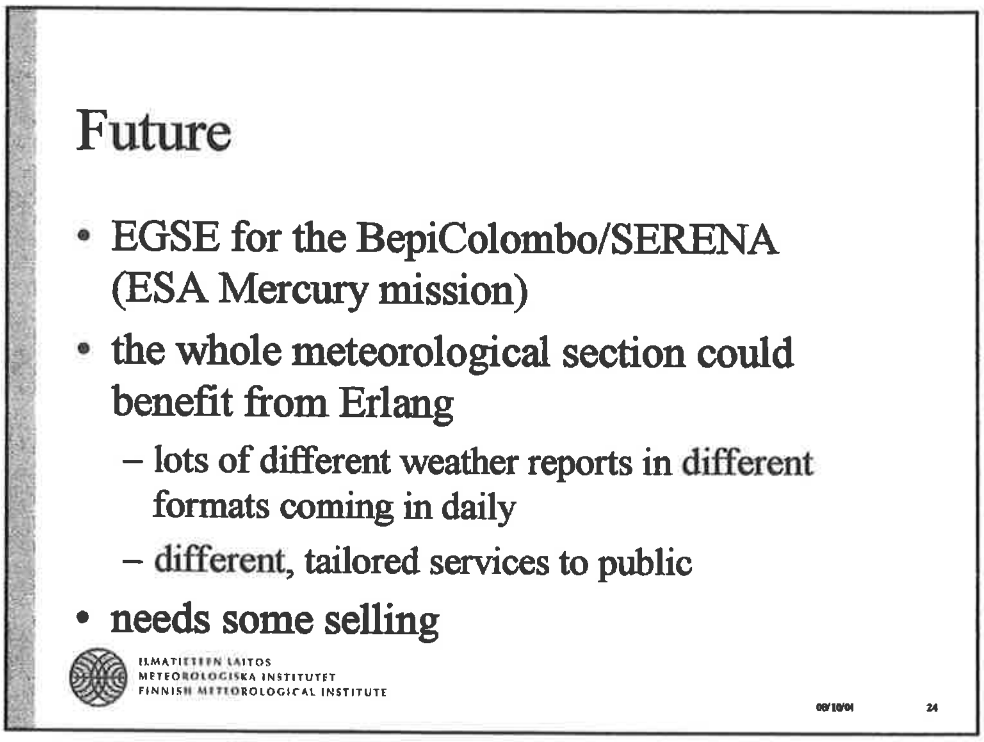
08/10/04 22

2005: COSIMA net nodes



ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

06/10/04 23



Future

- EGSE for the BepiColombo/SERENA (ESA Mercury mission)
- the whole meteorological section could benefit from Erlang
  - lots of different weather reports in different formats coming in daily
  - different, tailored services to public
- needs some selling

ILMATIETEEN LAITOS
METEOROLOGISKA INSTITUTET
FINNISH METEOROLOGICAL INSTITUTE

06/10/04 24

1(16)

SERVAL: an Internet software *VLAN* switch developed in Erlang *

Alejandro García Castro, Francisco Javier Morán Rúa
Igalia Software Engineering
Gutenberg, 34B 2°, Polígono de A Grela – 15008 A Corunha
e-mail: {acastro,jmoran}@igalia.com

Juan José Sánchez Penas
University of Corunha, Computer Science Department
Campus de Elvinha – 15071 A Corunha
e-mail: juanjo@dc.fi.udc.es

Abstract

There are situations in which it is very interesting to connect a machine to a different *Local Area Network* from the one its network card is actually connected to. Some network applications require our local host to be virtually connected to a remote *Local Area Network*. This article describes a proposal to develop a software system that emulates the operation of a switch, allowing to create *Virtual Local Area Networks* over the Internet, that completes the current similar solutions. We have created a prototype developed with Erlang/OTP using a client/server architecture and we are working on the integration with the Operating Systems using virtual network interfaces. Erlang is very suitable to face the main issues of this system: performance, communications and fault tolerance. We have accomplished performance and functional tests to assess the suitability of the designed system using the prototype. The paper will explain the current results of the research and describe future work.

1 Introduction

SERVAL is a research and development project whose aim is to assess the feasibility of a system for creating *Virtual Local Area Networks* (*VLANs*) using a software server. The main goal of this software is to provide a way to set up *VLANs* between computers, no matter their location or the connection they use to access the network. For this purpose, we have designed a system that emulates the operation of a hardware switch. The Operating System in the client does not have a regular network interface, but a special program that, acting as a virtual interface, communicates with the SERVAL server. The clients can connect to *VLANs* defined inside the server, which works as a software switch forwarding the messages between the clients in the same *Virtual Local Area Network* (*VLAN*).

Nowadays the solutions available to link two *Local Area Networks* (*LANs*) do not provide some features that would be desirable in some cases. The main technologies for connecting remote *LANs* currently are: *Virtual Private Networks* (*VPNs*) and *VLANs*. The applications we have in mind as interesting examples to be implemented on top of this technologies range from mobility solutions to file sharing.

The main issues of this project that we have to face are:

- Client/server architecture: we have to support this kind of architecture because the connection from local range IP addresses is a requirement, and therefore P2P solutions would not satisfy our needs.
- Performance: the emulation would have no sense if we do not have suitable latency and throughput. The system should be able to overcome stress situations. Scalability is also an important feature.

*Partially supported by Xunta de Galicia - PGIDITSIN0313E

- Operating System integration: the interfaces to our switch in the client side must be virtual network interfaces. User space applications would use these interfaces as the regular ones. Our target Operating Systems are GNU/Linux and Microsoft Windows.

- Communications security: this kind of systems should assure their communications, because the traffic goes through an unsafe medium.

We have designed a system following these requirements: a client/server architecture aimed to solve the main risks we have detected. We have decided to use Erlang/OTP [1] as the development environment because its features fit very well with the project goals.

In this research we want to face and measure the main risks we can see to develop a system like the one we describe. The paper will explain the current results of the research and describe future work.

In Section 2, the current alternative solutions for connecting remote *LANs* are described, and their advantages and disadvantages discussed; the motivations for the project are also presented. In Section 3, the main goals and system requirements for the research and development project are explained. After that, Section 4 introduces the system architecture, leaving for Section 5 the details related with the use of Erlang inside the system. Current status of the project, including some performance tests, is presented in Section 6, before concluding in Section 7.

2 State of the art and motivation

Nowadays technology brings us some options to connect remote *LANs*, but these applications do not provide us features that are very interesting in some environments.

We could use *Virtual Private Networks* (*VPNs*) to create a virtual connection between remote *LANs* communicating over possible untrusted networks. With this technology we can communicate remote networks emulating neighbor networks with a router in the middle. But using this kind of technology we can not transmit non-routed traffic between the network (therefore, local area protocols cannot be used). Besides, we need to configure a router to send out the packages. An example of this kind of software is FreeS/Wan [2], which implements IPSec [3], an standard protocol for encrypting IP traffic between two networks connected by two IPSec gateways.

Another option would be the use of *Virtual Local Area Networks* (*VLANs*) connecting *Local Area Networks* that are physically separated, enabling non-routed traffic between networks (and therefore local area protocols). *Virtual Local Area Networks* are normally implemented using the 802.1Q [4] protocol, which sends layer two traffic with *Virtual Local Area Network* information to define *Local Area Networks* using ports of different switches. The fact of being able to communicate any kind of traffic between the networks, would simplify some regular tasks when we share resources and will also enable the use of applications that communicate with each other using protocols like Rendezvous or SMB. The main limitation of this kind of solutions is that, nowadays, we can only deploy a system like that if we have control over all the physical switches placed between the host and the network we want to connect it to; besides, all the intermediate machines should have that feature implemented. We cannot forget that a telecommunications company, even controlling all the hardware of the network, can not touch the configuration of their machines dynamically in a safe way, because any mistake would spoil all the traffic of the network. We also have to remark that changing the configuration of the machine is not simple, and a trained technical assistance would be needed.

The system we propose in this paper can emulate this *VLANs* behavior using a scalable, distributed, TCP/IP server that acts as a software switch. The clients would run programs that simulate logical network cards connected to the software switch.

We have taken some features of both *VPN* and *VLAN* systems to define the main goals of the project. The security of *VPNs* is a very important feature, because communications are performed through an unsafe medium. Authentication and authorization are a important issues that needs to be solved properly. Hardware switches are designed to handle a great flow of packages, therefore the system has to be ready to manage heavy stress conditions. The performance is another main issue we have to face and specifically the system scalability. We should consider a group of thousands of clients that define a group of *VLANs* trying to send their discovery messages through the switch, the latency has to be correctly handled. In any case, the use of this kind of systems must be well designed because the amount of traffic that the local protocols produce can be very large.

Some of the standards and well-known technologies we are using or considering for the project are:

- *Local Area Network* technologies: Ethernet, Token Ring, etc. We also should review systems to manage this kind of traffic, congestion management.
- *Virtual Local Area Network* technologies: 802.1Q [4], 802.1D [5] and current hardware that supports it.
- TCP/IP transport protocols: UDP and TCP. We want to do research about which one should be more appropriate.
- Application level protocols and their encryption systems: SSL and TSL [6].
- Network interface emulation, both for GNU/Linux and Microsoft Windows.

We have discussed and proposed some applications of the SERVAL technology together with *R, Cable e Telecomunicacións de Galicia, S.A.*, to learn more about possible use cases and increase the knowledge about the system requirements. The idea was to find out some applications in which the advantages of the technology would make them specially interesting for our project:

- Virtual corporative *LANs* among several physical networks in a company, with a simple and flexible configuration. Enterprises could define communications between their headquarters easily, they could even work from home, using their personal Internet connection, but accessing the network and resources of the company. This is very interesting for mobility: if a worker is in a different place than the rest of the company, he can still connect to the network and develop his tasks or access to a document that he has in his account.
- File sharing: users could easily create a private network between them to transmit information. They just have to use standard local area network protocols and they could share resources and services.
- Games through Internet that could only be used before in a *LAN* environment. This is an interesting product for a telecommunications provider company like, because the easiest way to play in a network is using machine and software discovery protocols that only work in a *LAN*. The tool to manage the connections to the *VLANs* can be an easy interface that allows a regular user to connect to this networks and play network games the same way he/she is used to do in local environments. The entertainment industry is nowadays an important part of the telecommunication business.

3 Project goals

The main goal of this research is to assess the feasibility of the use of the current technologies for building a software system able to create and control *VLANs*. We have agreed some functional requirements that the developed system should fulfill:

- Client/server architecture: the basic architecture of the system should have these two layers. We need this kind of architectural design because of the current Internet connectivity, there are a lot of conditions where the clients are in a network with local IP address range. In these cases we need this kind of architecture to assure the connection between the hosts, because P2P technologies would not adapt correctly to our needs. The flexibility and control that the server application provide us seems to be an interesting feature for the system. Anyway, we have also considered the *peer-to-peer* (P2P) architecture, there are some conditions where the P2P could be a good solution. Therefore, our main goal is the client/server architecture but we will consider the way to adapt the system to a P2P deployment.
- GNU/Linux and Microsoft Windows link layer integration: it is an important point for the system usability. Due to these requirements

we need a multiplatform development environment, able to produce software to be run in both GNU/Linux and Microsoft Windows. The perfect integration of the client with the Operating System lets us use the regular facilities and programs with SERVAL transparently. The Operating System will detect a new network interface that will use to transmit traffic like any other interface of the system. The integration with current local area technologies like Ethernet will also be a very interesting target because of the facilities that include. This is one of the main risks of the project, especially in the Microsoft Windows environments. In GNU/Linux we already have a virtual Ethernet driver (TAP) that lets us redirect the traffic to a user space application. Microsoft Windows environments are closed and the assess of the solution is not so easy, we will have to develop a system virtual driver.

- Performance: it is also a very important risk of the research, because we have to accomplish some minimum results in order to be able to properly emulate a *LAN*. The latency of the server is an important issue to solve: we have to reach a minimum latency even under stress conditions. The throughput should also be optimized, because we could loose too much bandwidth using the system. Regarding this subject we have to consider the scalability of the system, which should be the best solution for being able to handle a lot of concurrent users at the same time. We also have to think about the transport layer protocol used for the transmission: UDP or TCP; it seems that depending on the concrete conditions, one alternative could be better than the other, so this feature should be configurable.

- Fault tolerance: the system should be designed to continue working in the presence of software or hardware failures. The server could be deployed in a cluster of computers, and the design of processes and protocols should take into account any possible problem derived from any kind of error.

- Security: one of the most important features that the system should accomplish is to secure the connections; even the performance could be penalized in some conditions. Authentication, authorization and encryption should be added to the system, and the connections should be assured to avoid security problems. There are applications of the system where security is not an important issue but when we want to transmit sensitive information we must assure the communication.

- Heterogeneous *LAN* protocol encapsulation: regarding the type of protocols we should support at least Rendezvous and SMB. Anyway, it is very interesting to support other network technologies, being the optimal solution that the virtual interface had no difference with the rest of the network interfaces of the system. If we completely implement the virtual Ethernet driver we can accomplish this goal.

4 Software and hardware architecture

When we thought about building a system that emulates *VLANs* in a *WAN* environment, we had to decide the global system architecture we were going to use. It should be noticed, before going on with the detailed description, that, by *global architecture*, we mean both the hardware and software skeletons of the system.

The first option we evaluated was whether a *peer-to-peer* approach would be interesting. With the use of a *peer-to-peer* a approach, the scenario would be the following one:

Each user of the system would have in his computer a client of the SERVAL system installed. This client would negotiate a connection with another user having also installed the SERVAL program. Once the connection between the clients was established, they would be connected at link layer level, being this medium Ethernet compatible. Therefore, the Operating System of each computer would see the other one as if it was in its *LAN* with all the advantages this fact has.

This scenario looks fairly attractive but has several serious drawbacks that made us reject it as a general target architecture. The main disadvantage is that if both clients are in private networks, with private network IP addresses, then the *peer-to-peer* connection would not be feasible.

Another drawback is that this skeleton is not good if we want to emulate a lot of machines in the same *LAN*. This is so because the program in each client would have to open a *peer-to-peer* connection with each computer belonging to the same *VLAN*. This would cause a serious network overhead. For example, if we wanted to have a hundred users in a *VLAN*, the number of connections which would be necessary in order to open in each computer is one hundred and, altogether, they would be up to ten thousand. Besides, this skeleton would not be persistent and each client needs to know the addresses of all the other clients we want to include in the same emulated *LAN*.

A second architectural possibility to solve the system scalability problem with the number of connections would be a bus structure. This can be viewed as an improvement of the *peer-to-peer* approach. In the bus architecture approach, the programs would be joined forming a line. Hence, in this one-dimensional structure, clients would propagate messages in order they reach their final destination.

The bus architecture, as we can see, reduces the number of opened connections because each SERVAL program only has to be linked with two more users and the configuration complexity is lower than the *peer-to-peer*. However, it still has two important drawbacks: the connection among private IP clients, which is not possible, and how a client can belong to several *VLANs*. Another problem is the different latency of the messages depending on the position of the clients involved in the communication. Therefore, the bus architecture is not suitable to fulfill our requirements either.

After having discarded the previous system architectures, we came to the conclusion that the one we were looking for, more adapted to the project needs, was the client/server model.

In the client/server model the entities taking part are the following ones:

Server The server will be a program running as a daemon in an Internet accessible host. This program, the SERVAL server, will listen for connections coming from the users of the system. Its function is similar to the function a hardware switch has in a *LAN*. It will be able to group users in different independent sets and will be the mediator among the clients.

When a user wants to send a message to another client, he has to send it first to the server and, then, the server forwards it to its final destination. The fact that the server can group users in sets can be compared to the *VLANs* existing in some hardware switches.

Client The clients in this architecture will be the users who want to connect to our system. They will have to install in their computers the client program to access the virtual switch and through it, the rest of clients.

The user Operating System, accessing the SERVAL client, will be able to see the other clients in his groups as if they were in the same *Local Area Network*.

With the client/server model, the drawbacks and limitations existing for a P2P or bus architecture are overcome. In order to communicate the client and the server, operations and messages similar to the ones used in the link layer protocols were defined.

The messages interchanged between the client and the server are shown in Figure 1. The client can connect and disconnect to the server, ask for the list of available *VLANs*, and join or leave one of them. Other messages allow the client to send a message to a given *VLAN* (and therefore to all the users connected to it).

Other important messages are the `addressOfClientRequest` and its answer from the server side: the `addressOfClientResponse`. They were created to emulate the ARP Ethernet link layer protocol and are used when at network level a client wants to talk to another.

At network level clients communicate with each other knowing their network level address. For instance, if we are using TCP/IP knowing their IP. However, at link level it is not enough with the IP address to contact with the destination but it is necessary to know the link layer address of the next hop towards the destination as well. Therefore, in SERVAL we use the two former messages, `addresssOfClientRequest` and `addressOfClientResponse`, to find out the SERVAL link layer address of a client knowing its network address.

Finally, the global architecture chosen has to be both fault tolerant and scalable. By fault tolerance we mean the system should be resistant to a partial system crash, being able to overcome the situation and continue the normal operation. By scalability we mean that if the system requirements grow, and the number of clients connected is higher than the initially expected, new resources can still be added to the server in order to increase its performance and fulfil the new requirements.

With this goals in mind, we have extended the client/server model to be a distributed system. So, in the final architecture, instead of having only a node of the SERVAL program running in a host, we have several nodes which collaborate with each other: they will detect a node failure and restart it if possible, the clients of a crashed node will be moved to another node of the cluster, and so on.

5 Implementation of the system using Erlang

As explained in the previous section, we have to build three software artefacts:

- SERVAL server. It is the program which plays the role of a hardware switch in a real *VLAN* environment. It has to be run in an IP accessible from all clients. Its function is to manage all the operations related to both the *VLANs* and client management. For instance, it accepts input connections from clients, creates *VLANs*, routes messages among clients etc.
- SERVAL client. It is the software which clients must use to access the server. It has two parts:
  - User land adapter It is the program which receives Ethernet frames from the virtual Ethernet driver and maps them to messages of the communications protocol. Next, these messages are sent to server.

    It has also an interface to receive requests directly from user. For example, requests to join *VLANs*, to abandon them etc.
  - Virtual Ethernet driver. It is an Ethernet driver which implements a virtual network card. It controls the communications with the Ethernet Operating System kernel API and the user land adapter.

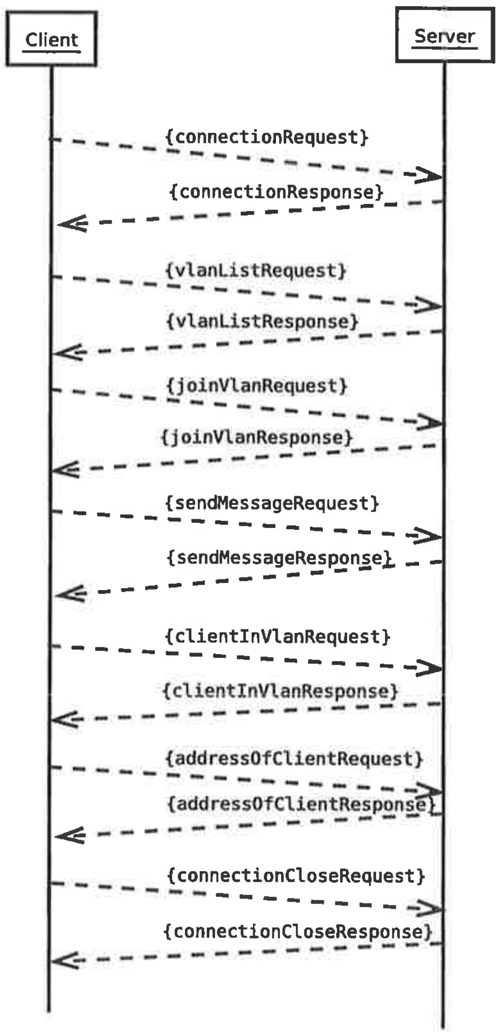


Figure 1: Messages interchanged between client and server

7

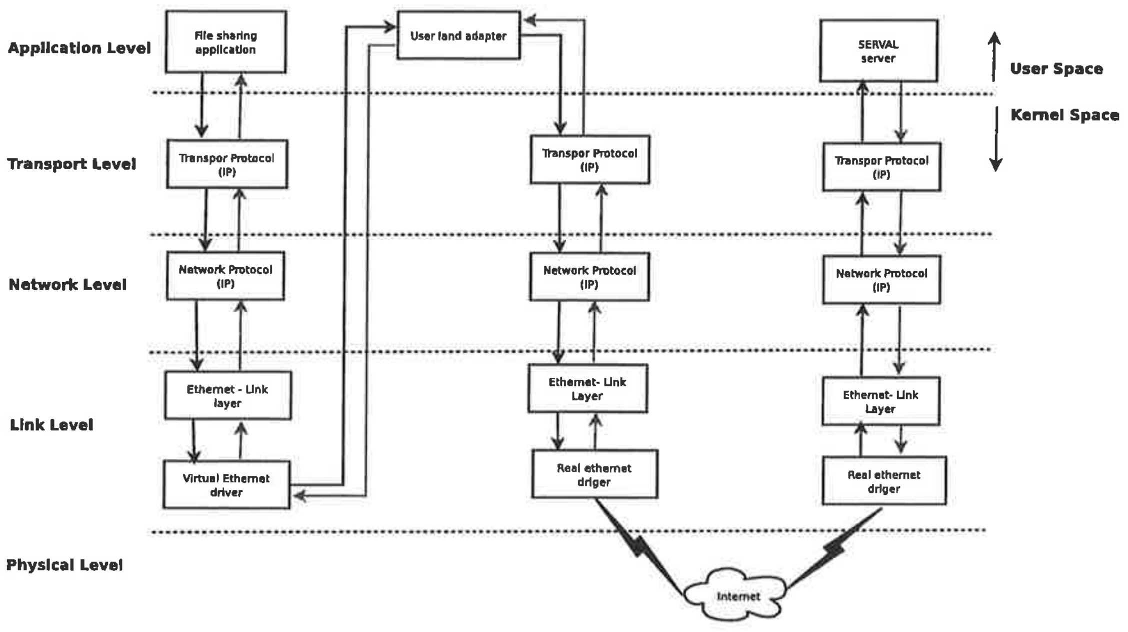


Figure 2: Relation between the client and Ethernet driver

A diagram which summarizes the relation between the client agent developed and the virtual Ethernet driver is shown in Figure 2. It shows how an end user application running in a computer connected to a server would use the client and the communication protocol stack. The messages sent by the application level program are encapsulated into a transport level protocol. Then, the transport protocol is encapsulated into a network level one. After this, the network level datagrams are converted into Ethernet frames which, next, are sent through the virtual Ethernet interface. This virtual interface delivers the frames to the user land adapter which, finally, maps them to the protocol messages used to communicate witch the server. They are encapsulated through the protocol stack again and are sent by a real network interface.

The reverse path is followed by the messages delivered to the user agent by the real network interface after the demultiplexing which happens on ascending the communication protocol stack.

In the rest of this section, we detail the Erlang implementation of both the server and the client agent.

5.1 Server implementation in Erlang

The process structure of the multinode switch can be observed in Figure 3. They are represented the process classes and the relations existing among them. The relations we emphasize are two:

- *Creation link relation.* A process class A has a creation link relation with a process class B when processes of class A create processes of class B. This relation type is shown by the continuous lines.
- *State link relation.* A process class A has a state link relation with a process of class B when in the state of processes of class A is stored the process identifier or the registered name of processes of class B. This relation type is represented as dotted lines.

The task each process class carries out is explained to understand the Erlang implementation of the virtual *VLAN* switch accurately.

serval_app This process class is an Erlang application behavior. It has been created to start/stop the server.

serval_sup This process class is an Erlang supervisor behavior. It supervises the *serval_server_logger*, *serval_tcp_port_manager* and *serval_udp_port_manager* process classes.

serval_server_logger It is an Erlang generic server behavior. It has the mission of logging all the information sent by the other processes existing in a node.

serval_tcp_port_manager This one is another generic server. It listens on a TCP port waiting for incoming connections. Every time it receives a connection request it spawns a new process called *serval_connection_manager*.

serval_udp_port_manager It is also implemented as a generic server and its function is to listen on an UDP port waiting for incoming messages. When it receives the first message coming from a source socket (source IP, source port) spawns another process, **serval_connection_manager**, for that connection.

serval_connection_manager It is a supervisor and is entrusted with the task of coordinating and supervising the *serval_connection_communications_tcp*, *serval_connection_communications_udp* and *serval_connection_operation* process classes.

serval_connection_operation It is a generic server. This process carries out the operations associated with the messages sent by the client.

serval_connection_communications_udp This process class does the sending of messages from the virtual switch to the SERVAL user agents. The messages which arrive from the clients to the server are received in UDP communications by the **serval_udp_port_manager**.

serval_connection_communication_tcp This process class does both the receiving of the messages which arrive to the server from clients and the delivery of the messages sent by the virtual switch to the user agents.

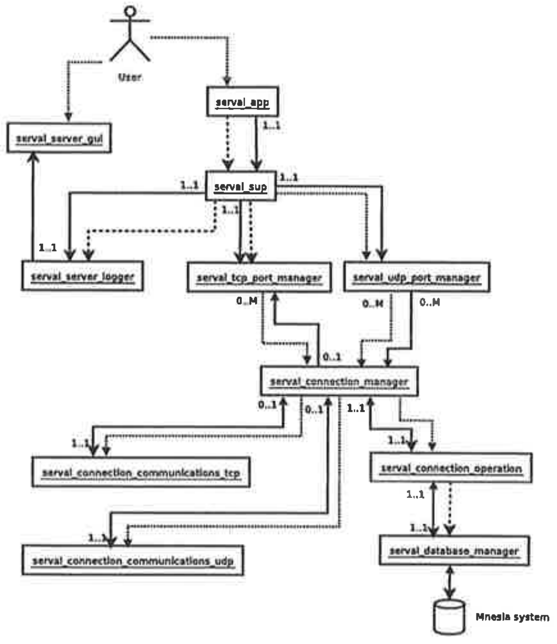


Figure 3: Link process class diagram

5.2 Client implementation

Regarding the user agent implemented in Erlang, the process diagram of the design can be observed in the Figure 4.

The process classes which take part in the user agent are:

serval_test_client_gui This process control the interface which is used to make requests to the server and to send and receive messages from *VLANs*.

It is the process which talks to the graphics system and receives all the events from it when users click in the interface widgets.

serval_test_client Processes of this process are created by the *serval_test_client_gui* when the user requests a connection with the server.

It is the process entrusted with the task of sending messages to the SERVAL server and receiving from the Internet all the information sent to the client.

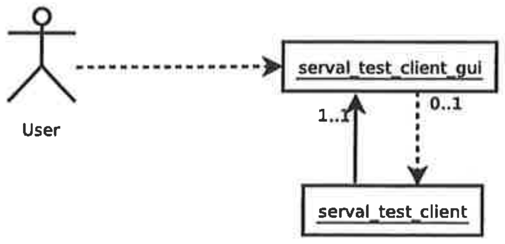


Figure 4: SERVAL user agent link process class diagram

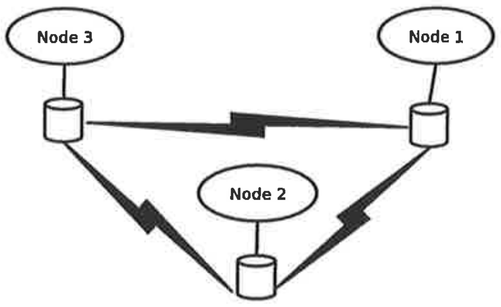


Figure 5: SERVAL server cluster with 3 nodes

5.3 Storing *VLANs* information in Mnesia

In Section 4, it was stated that the project goal was to build a scalable, concurrent and fault tolerant system.

To achieve this target, we decided to make our virtual switch a multiple node software. We mean that in our architecture we have several instances of the SERVAL application which can attend client requests. Therefore, clients can connect to any of the nodes of the server, and independently of the node they access they see each other and the same *Virtual Local Area Networks*. In other words, this means that a client connected to a node A has to be able to send a message to another client connected to a node B if both belong to the same *VLAN*.

In this multinode architecture, then, it is necessary to have communication among nodes to share *VLANs* information. We decided to use Mnesia, a distributed database management system included in Erlang/OTP. With this distributed database we can access information of the existing *VLANs* and clients connected to each of them from any of the nodes of the system.

The use of Mnesia has big advantages for this project. Thanks to its distribution capabilities, synchronization among nodes is done automatically by this DBMS and it is transparent to SERVAL. In this way, as we rely on Mnesia, synchronization of the virtual switch nodes is done without overloading the code with synchronization tasks. This idea is represented in Figure 5, which shows the situation in which there are three nodes running belonging to the SERVAL cluster. It can be observed how each of the nodes has a local copy of the distributed Mnesia tables and how the communication among nodes is done using the database.

5.4 ASN1 Erlang compiler for communication protocol implementation

We decided to use the *Abstract System Notation One - ASN1* for the definition, transmission and encapsulation of our internal, client/server, communications protocol.

ASN1 is formal language for abstract description of the messages interchanged in communications protocols, with independence of the programming language chosen and the data memory representation. ASN1 is a standard since 1984 and, consequently, its codification framework is mature and it has been used successfully in a lot of different scenarios.

In the project we have used the Erlang ASN1 compiler. This compiler in very useful because it generates coding and decoding functions which can be directly used by Erlang programs.

5.5 Process collaboration scenarios

In this section we describe several collaboration scenarios with the aim to understand more accurately how the SERVAL system works.

5.5.1 SERVAL server starting

When the SERVAL server is started, it is launched a process of class *serval_app*. Next, the *serval_app* spawns three more process of class *serval_server_logger*, *serval_tcp_port_manager* and *serval_udp_port_manager* respectively.

Our SERVAL server has support to maintain communications with the clients using as transport protocol TCP or UDP. This is the reason why, when the SERVAL server is launched, the processes *serval_tcp_port_manager* and *serval_udp_port_manager* are created. The first listens for input connections in a TCP port and the second listens for packets in an UDP port.

5.5.2 SERVAL client connection request to SERVAL server

We are going to explain what happens when a client requests a TCP connection with the SERVAL server.

First an *open_port* TCP message is received by the process *serval_tcp_port_manager*. Second, this process creates a process *serval_connection_manager* and, then, this last one spawns two more processes, a *serval_connection_communications_tcp* and a *serval_connection_operation*.

With these three processes we have the structure to manage all the operations related to the client has requested the connection.

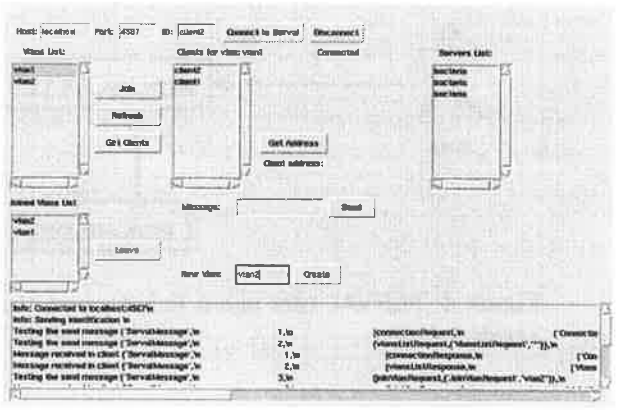
5.5.3 Message interchange between processes belonging to the same *VLAN*

When a client A wants to send a message to another B, first, A has to find out B client identifier knowing its network level address. In order to get B client identifier, it sends the SERVAL server a `adressOfClientRequest`.

All clients in the same *VLANs* that A, receive this `addressOfClientRequest`. The one whose network level address matches the one included in the message, that's to say B, sends back to A an `addressOfClientResponse` with its client identifier.

Next, A sends the SERVAL server a `sendMessageRequest` with the data and the B client identifier as destination address. The SERVAL server checks that B is in the same *VLANs* and, if this condition is true, delivers B the message. After this, sends A a `sendMessageResponse`.

We can observe all this behaviour in the figure 6.



```
Info: Connected to localhost:4567\n
Info: Sending identification \n
Testing the send message {'ServalMessage',\n        1,\n       {connectionRequest,\n          {'Connectio
Testing the send message {'ServalMessage',\n        2,\n       {vlansListRequest,{'VlansListRequest',""}},\n
Message received in client {'ServalMessage',\n       1,\n        {connectionResponse,\n             {'Con
Message received in client {'ServalMessage',\n       2,\n        {vlansListResponse,\n              {'Vlans
Testing the send message {'ServalMessage',\n        3,\n       {joinVlanRequest,{'JoinVlanRequest',"vlan2"}},\n
```

Figure 7: Client screenshot

6 Current status of the research

6.1 The prototype GUI

In Figure 7, the graphical user interface developed for the prototype user agent can be seen.

In the GUI window of the example, the client is connected to a SERVAL server which is listening for input connections in port 4567. The user is connected to the virtual switch through a client with address *client1*. We can see that there are two *VLANs* created, *vlan1* and *vlan2*, and that *Client1* joined both. Finally, we can observe that *vlan1* contains two clients with the address *client1* and *client2*.

6.2 Testing the system

Two key features of the system, as already explained, are performance and fault tolerance. Every design and new characteristic we add to the system is developed thinking of what impact it will have over these two variables. But we are not only think in the impact of the modifications but trying to measure and to check the system with the new additions. In this subsection, some performance measurements and fault tolerance tests that we have carried out are described.

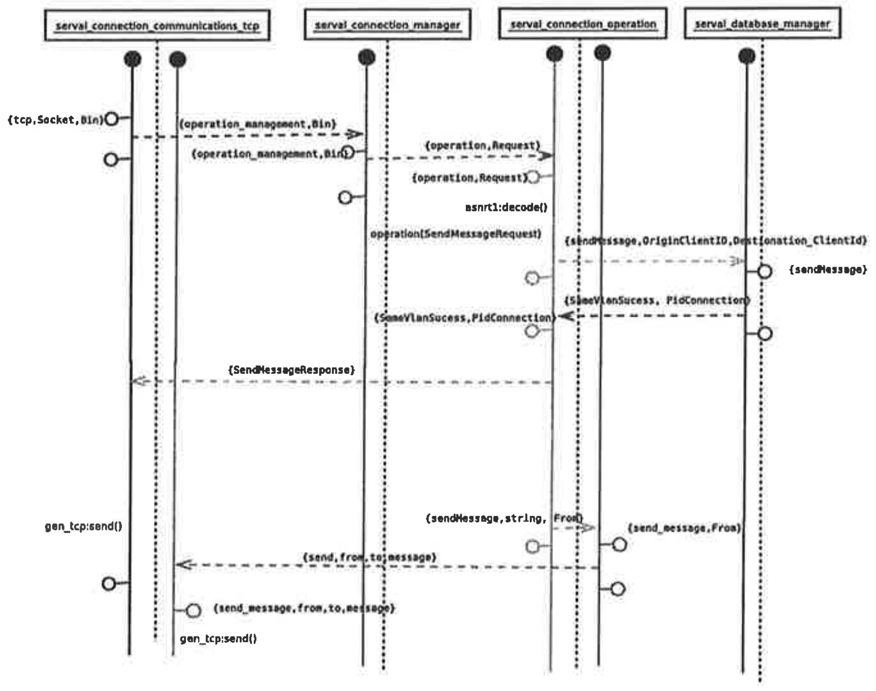

serval_connection_communications_tcp
serval_connection_manager
serval_connection_operation
serval_database_manager
{tcp,Socket,Bin}
{operation_management,Bin}
{operation_management,Bin}
{operation,Request}
{operation,Request}
asnrt1:decode()
operation(SendMessageRequest)
{sendMessage,OriginClientID,Destionation_ClientId}
{sendMessage}
{SameVlanSucess, PidConnection}
{SameVlanSucess,PidConnection}
{SendMessageResponse}
gen_tcp:send()
{sendMessage,string, From}
{send_message,From}
{send,from,to,message}
{send_message,from,to,message}
gen_tcp:send()

Figure 6: Message interchange between processes belonging to the same *VLANs*

6.2.1 Performance tests

Due to the real time nature of the system, and the amount of concurrent users that should potentially be able to handle, a good performance is an essential requirement for the system.

We have concluded that the performance in SERVAL can be measured by the two following variables:

- **Operation latency**. The operation latency is the time it takes an operation to be completed, since it is ordered until it is finished. The lower the operation latency, the higher the system performance. As final goal, we are specially interested in reducing the message latency of a message sent from a client to another through the server.

- **Throughput**. The throughput of a system is the number of operation requests it can handle in a period of time. A higher throughput means that the system is able to carry out a lot of operations concurrently. We should also consider the bandwidth reduction caused by use of our internal protocol. We increase the size of the packages internally, because we need to store some extra information, we should keep them small to enhance the use of the medium. Currently we do not consider this increment of the size of the packages a problem.

In order to make these measurements we have developed a special module for creating performance graphics. It is call *serval_gnuplot* and uses the GPL application *gnuplot*.

With this module carried out performance tests in both the client and server side. We will describe two examples in which we have obtained performance improvements after analyzing the operation latency and the system throughput.

The get_vlans message This protocol message has a latency which grows with the number of *VLANs*. So, the high latency of this message blocks another messages in the server queue, which cannot be delivered by the server quickly.

After some performance tests, the Mnesia query for getting all the *VLANs* created in the server, was pointed out as the system bottleneck for this operation.

In figure 8 we can see the time process the *get_vlans* with different number of *VLANs* created in the server.

We have represented also the *VLANs* number against the *get_vlans* latency. The graphichs obtained can be seen in figure 9.

The impact in the latency of the messages that the number of *VLANs* created in the system has, can be easily observed.

The solution we chose to solve this problem was the use of a memory cache to speed up the request operation. So that, with this cache we were able to rise up the throughput of the virtual switch and, besides, we could drop the latency of the *get_vlans* message.

The `adressOfClientRequest` message After analyzing the system operations performed during the message interchange between the clients of the system, we detected that the number of *addressOfClientRequest* sent by the client agents was too high.

Many of these *addressOfClientsRequests* messages, however, were asking for the address of the same client. Hence we implemented a cache in client side to store the mapping between network addresses and the SERVAL identifiers.

Doing this we got a double improvement. First, we reduced the number of *addressOfClientsRequests* messages sent to the SERVAL switch dropping its average load; and second, we decreased the latency of this message. In Figure 10, we can observe how with the client address cache the number of *get_addr* messages received in the server is less than the number of the other message types.

6.2.2 Fault tolerance

To have fault tolerance features was already stated in Section 3 as an essential goal for the project.

There are different strategies which can be followed to create a fault tolerant system. In our case, the concurrency and distribution properties of Erlang have allowed us to build a robust recovery system based on the multiple node server described in Section 4.

The idea is that if a node crashes the system has mechanisms for letting the rest of nodes continue the work that was being carried out by the crashing one. This can be easily explained introducing some situations and how they are overcome:

One node crash If a node crashes, the recovery mechanism consists mainly in the reconnection engine implemented in the clients.

The description of this engine is as follows: each client receives from the server an address and port list with all the nodes belonging to the cluster as answer to the connection message. Therefore, when a client detects the node is connected to is unreachable, then it requests the connection with another of the nodes. This second node is obtained from the node list that, as mentioned, is stored in the client state.

When the server receives a connection request, it uses Mnesia to check if the client was connected through another node before. We consult Mnesia because we use a distributed table which registers each client connected to SERVAL. The information we record for each client is:

- The link layer SERVAL address. We call it *client identifier* as well.
- The process identifier of the SERVAL process we use to manage the client.

Taking this into account, we query the former Mnesia table to find out if there is a row with the same link layer address that the one of client is requesting the connection:

- If we get zero rows this means that the client is not doing a reconnection because the node it was acceding has crashed. We record the information for the the new client in a new row.
- If we get one row this fact means that the client is doing a reconnection. Hence, we have to update the row obtained with the new process identifier of the process created in the new access node to manage the connection.

Client crash This scenario describes which recovery actions are performed when a unsuitable client disconnection takes place, and the message

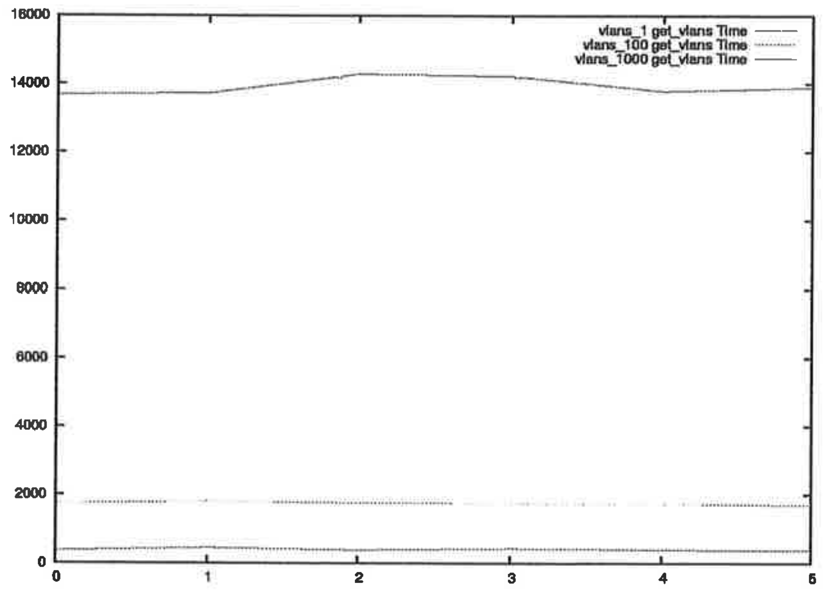


Figure 8: Time to process the get_vlan message depending on the *VLANs* existing in the server

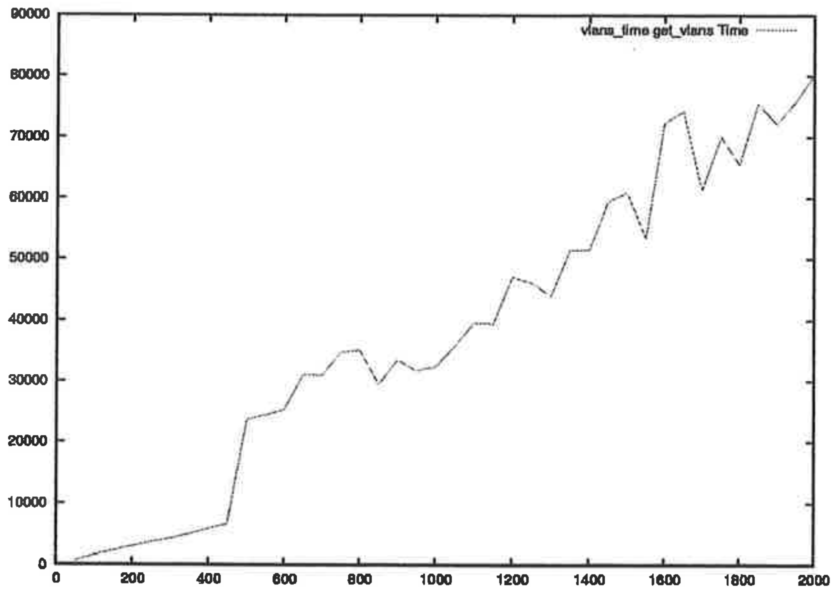


Figure 9: *VLANs* number against *get_vlans* latency

14

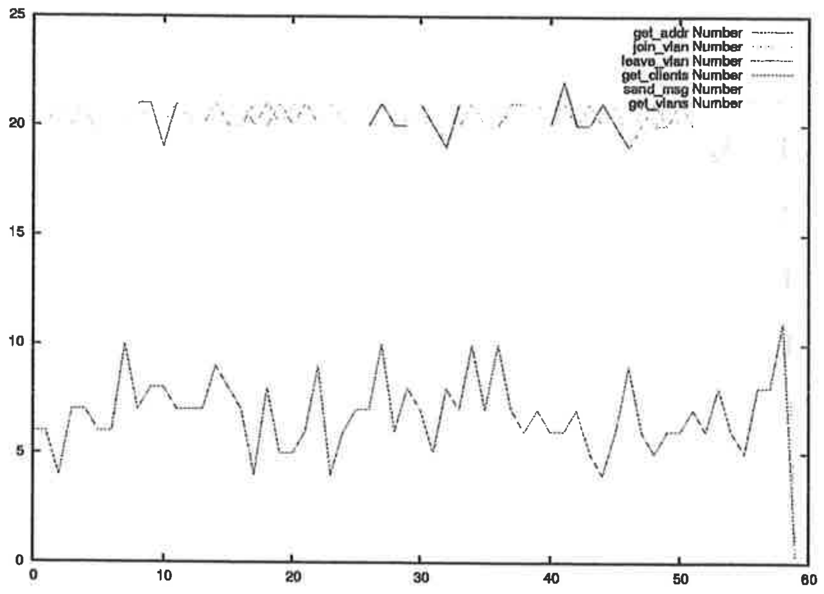


Figure 10: Number of each message type received in the server

`connectionCloseRequest` is not sent by the client to the server.

The client crash can be caused by a software failure, but the message could be also lost due to some network problem. When this happens, the processes responsible for the client management are kept alive in the node the client was connected to. Therefore, in this situation, if another client sends a message to this crashed one, in the switch cluster the client is detected as connected. Because of this, the switch sends the message, though it will never arrive to its final destination, because it is down (we will explain why we do not matter this lost of messages).

Next, we are going to describe the strategy we decided to implement when the former situation happens. Each client incorporates a *keep alive* mechanism which sends the server a message at regular periods of time. Each server node has a process which monitors the receiving of the keep alive messages coming from clients. If this monitoring process detects that a client is not sending the keep alive messages and it has not requested the connection close, then it kills the management processes for this client. Besides, the monitoring process deletes from Mnesia the information related to the *VLANs* the crashed client was connected to.

As we can observe, there is a period of time between a client crashes until this fact is detected in the server. If a message is sent to the crashed client during this period of time, the messages do not reach the destination. We could implement acknowledgement messages to guarantee they always arrive to its final destination. However, we do not wish a too heavy protocol because we are emulating a link layer scenario. Therefore, these failures rarely happen and will be detected by upper layers in the communication protocol suite used.

6.2.3 Future work

Nowadays, we have in mind to work on several things, some of which are:

Congestion control protocol We want the virtual server switch nodes to monitor its internal work load. So, to succeed in this task we are implementing the processes necessary to measure this magnitude. Besides, it is necessary to extend the link layer protocol in order it incorporates the messages needed for congestion control.

SERVAL clients are being also improved with the ability to process the congestion control information sent by the server. So, if they detect that the work load of the node they are accessing to is too high, they can decide to ask another node with lower load lodge them.

Access Control List For many real environments it is very important to be able to give and revoke permissions to clients to do certain operations or to access certain resources in the virtual switch.

For instance, we can wish to let users with a certain profile create *VLANs* in our server and forbid this operation to other users groups. Another example, is the possibility to join a *VLAN*. A server administrator may wish to have a predefined set of *VLANs* created in the virtual switch. He may wish to have the control to authorize or deny the access to each *VLAN* following a per user or per group policy. All this can be done implementing the ACL engine and we are hands on.

Ethernet driver In the section (4) we saw that one the components of the architecture is the interface with the link layer of the communication protocol kernel stack.

We are working in the strategy to implement this virtual Ethernet driver. He are also assessing the possibility to use the TAP device driver. The TAP is a Virtual Ethernet network device. It was designed as low level kernel support for Ethernet tunnelling. It provides to user-land application two interfaces:

- `/dev/tapX` character device
- `tapX` virtual Ethernet interface.

The user-land SERVAL Erlang client could use this device `/dev/tapX` to write Ethernet frames which will be received by the kernel. On the other hand, each Ethernet frame wrote by the kernel to the `tapX` interface will be received by the SERVAL client by reading the `/dev/tapX` device file.

Secure communications We are emulating a virtual switch, so our virtual operation environment is a *VLAN*. In a *LAN* the Ethernet frames don't leave the network limits so the security policy can be relaxed if we rely on our *VLAN* users.

However, although the SERVAL server virtual environment is a *VLAN*, the real environment is a *WAN*. As a consequence, all our link layer traffic is going to cross through the Internet and will be exposed to be sniffed by everybody. So we are implementing SSL support in client and server side to cipher communications.

7 Conclusions and future work

In this paper we have explained the main motivations and goals of the SERVAL project, the designed solution proposed to achieve them and the current results and work that we are developing. Through the paper we have established that Erlang is a suitable technology for this project, where network communications, performance and fault tolerance are the main requirements.

We think that our project can be interesting and applicable in a lot of real and diverse scenarios - pointed out in the paper - in which emulating *Virtual Local Area Networks* over Internet is a good solution. SERVAL *Virtual Local Area Networks* management will allow a more flexible way of designing network topologies.

We can transmit that nowadays we are satisfied with the results we are obtaining and with the future of the project. In fact, we are encouraged with having a real and almost complete SERVAL prototype in the near future that will be the first step to build an actual system.

8 Acknowledgements

The authors want to thank Alberto García González, José Juán González Alonso, Iago Toral Quiroga, Adrian Otero Vila and Ángel Vidal, and in general the whole Igalia company, for their collaboration in the development of the SERVAL project.

References

[1] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming*

in Erlang, 2nd edition. Prentice Hall International, 1996.

[2] The FreeS/WAN Project. Ipsec gnu/linux implementation, 2004. http://www.freeswan.org/.

[3] The Internet Engineering Task Force (IETF). Ip security protocol (ipsec), 2004. http://www.ietf.org/html.charters/ipsec-charter.html.

[4] IEEE Project 802.2 Working Group. IEEE standard for local and metropolitan area networks: Virtual bridged local area networks. Technical report, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 1998.

[5] IEEE Project 802.2 Working Group. IEEE standard for information technology–telecommunications and information exchange between systems–ieee standard for local and metropolitan area networks–common specifications–media access control (mac) bridges. Technical Report ISO/IEC 15802-3:1998, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 1998.

[6] T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, IETF - (Internet Engineering Task Force), January 1999.

[7] W. Stallings. *Local Networks.* Macmillan Publishing Company, New York, 3d edition, 1990.

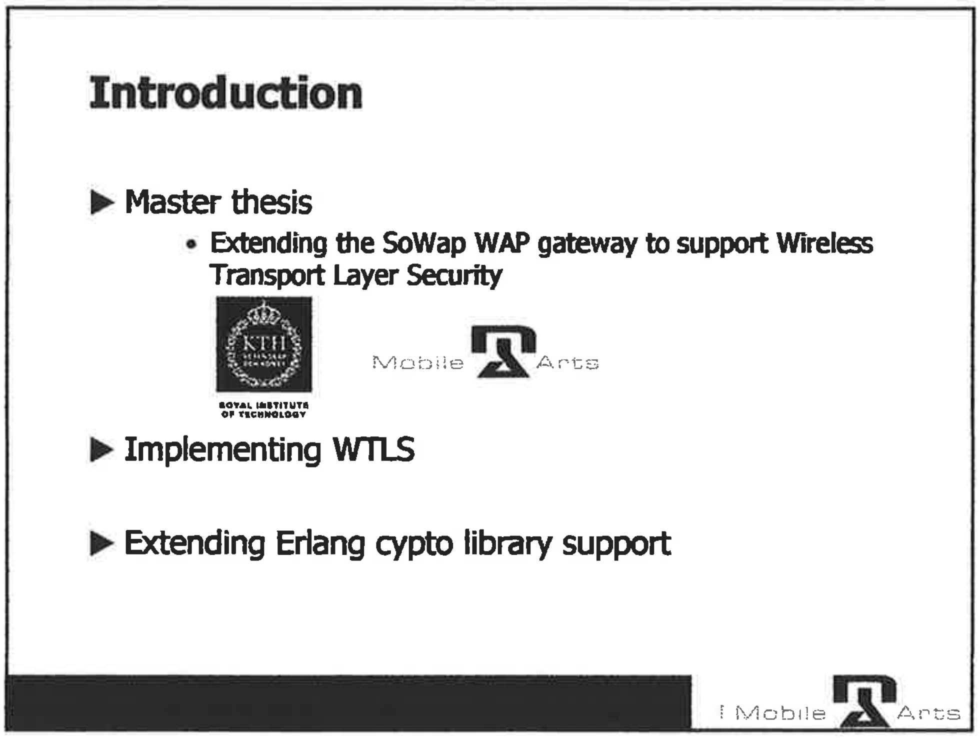
[8] IEEE Project 802.2 Working Group. IEEE standard for local and metropolitan area networks: Overview and architecture. Technical Report IEEE 802-2001, Institute of Electrical and Electronics Engineers, 3 Park Avenue, New York, NY 10016-5997, 2001.

[9] K. Pitt, D.; Sy. Address-based and non-address-based routing schemes for interconnected local area networks. *Computer Science Press*, 1986.

Mobile Arts

An Erlang WTLS Implementation

Erlang User Conference 2004



Introduction

- Master thesis
  - Extending the SoWap WAP gateway to support Wireless Transport Layer Security

KTH ROYAL INSTITUTE OF TECHNOLOGY

Mobile Arts

- Implementing WTLS
- Extending Erlang cypto library support

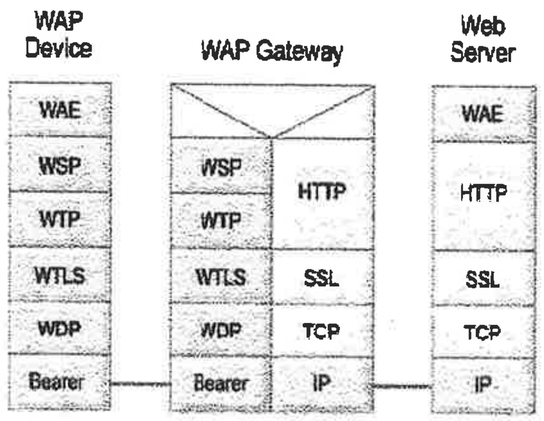
Mobile Arts

Background

- ▶ WAP – Wireless Application Protocol
- ▶ SoWap – Erlang Open Source WAP Gateway
  - History
  - Future

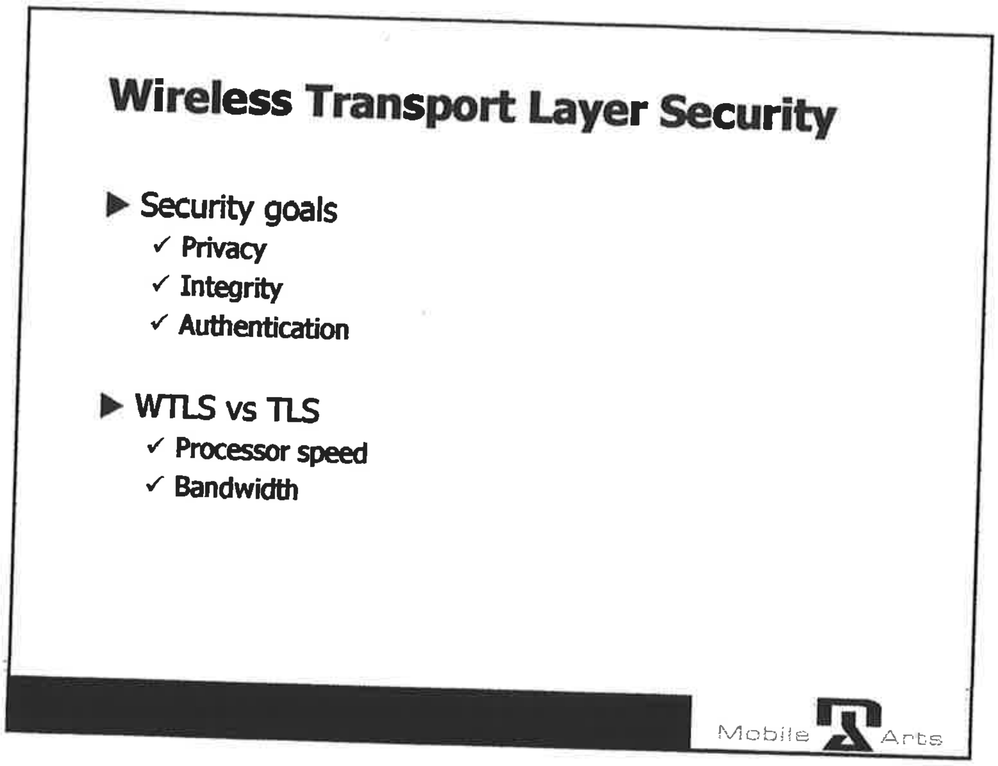
Mobile Arts

WAP Gateway



Mobile Arts

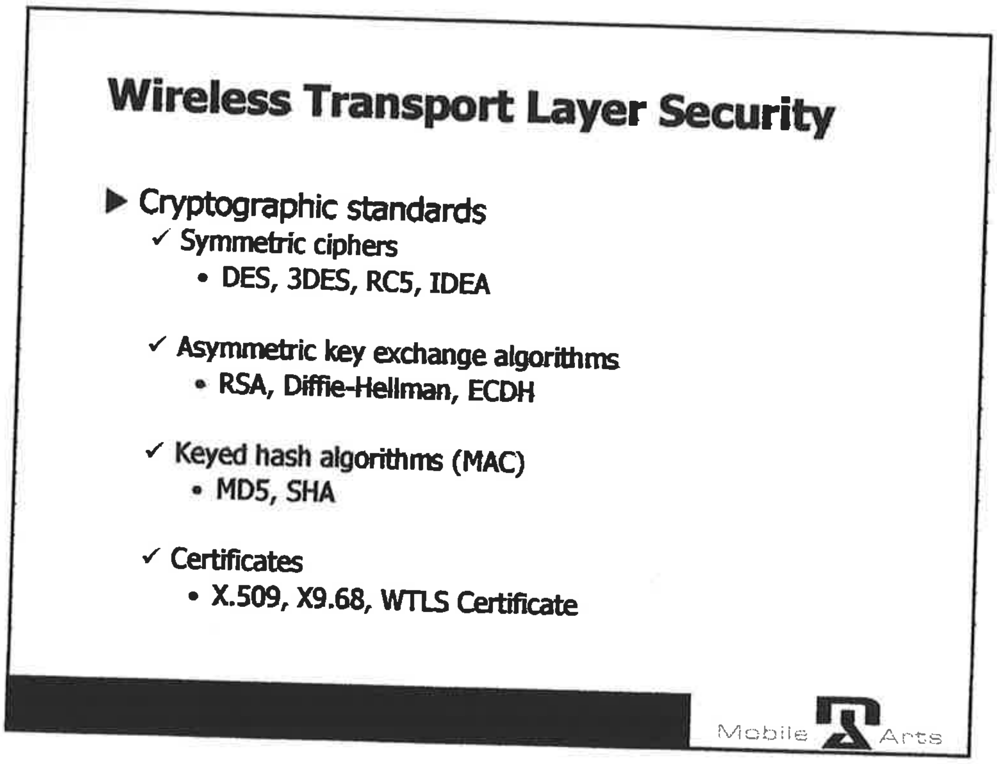
3



Wireless Transport Layer Security

- ▶ Security goals
  - ✓ Privacy
  - ✓ Integrity
  - ✓ Authentication
- ▶ WTLS vs TLS
  - ✓ Processor speed
  - ✓ Bandwidth

Mobile Arts



Wireless Transport Layer Security

- ▶ Cryptographic standards
  - ✓ Symmetric ciphers
    - DES, 3DES, RC5, IDEA
  - ✓ Asymmetric key exchange algorithms
    - RSA, Diffie-Hellman, ECDH
  - ✓ Keyed hash algorithms (MAC)
    - MD5, SHA
  - ✓ Certificates
    - X.509, X9.68, WTLS Certificate

Mobile Arts

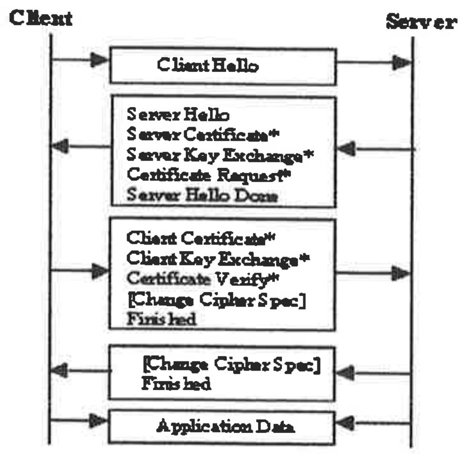
4

WTLS Handshake

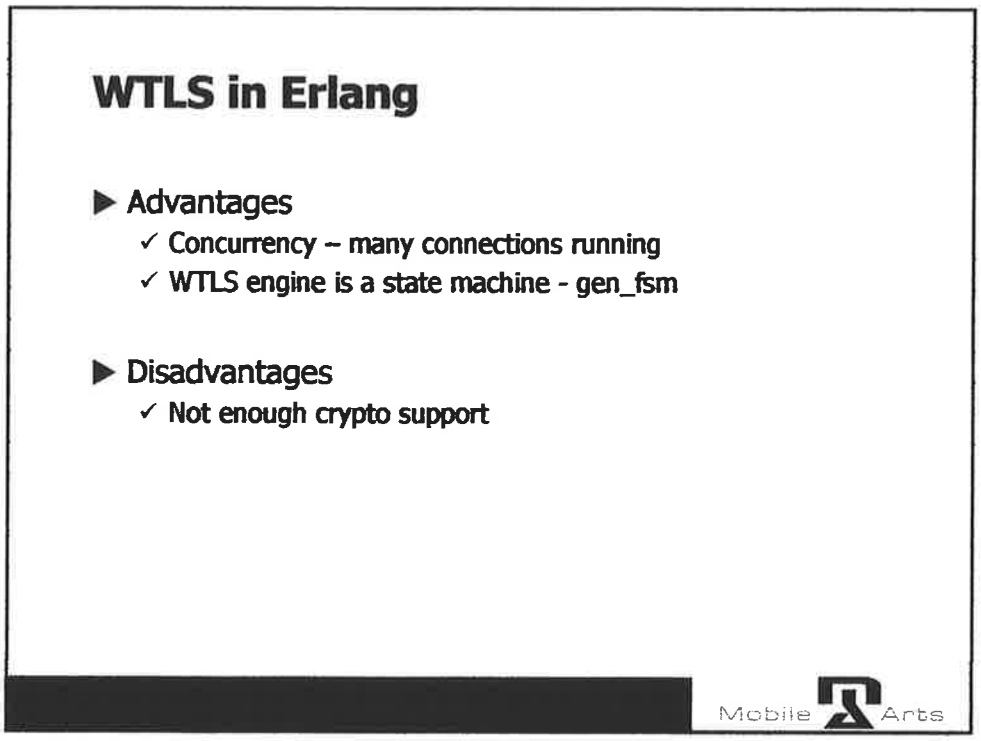
- ▶ Negotiate security algorithms
- ▶ Exchange random values and settings
- ▶ Exchange certificates
- ▶ Calculate secret

Mobile Arts

WTLS Handshake



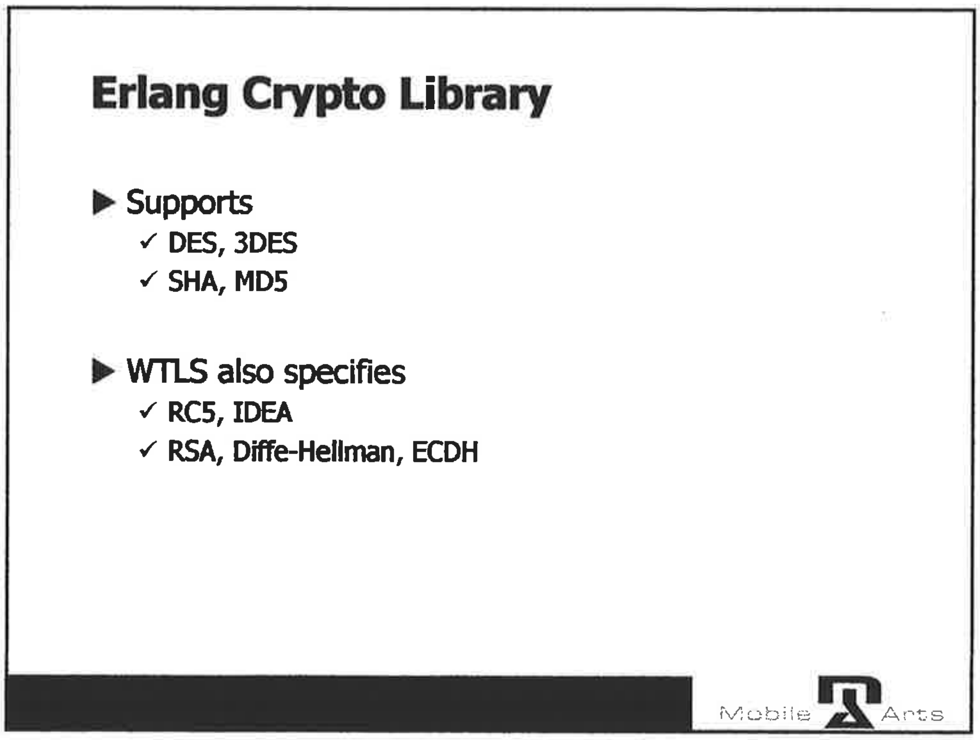
Mobile Arts



WTLS in Erlang

- ▶ Advantages
  - ✓ Concurrency – many connections running
  - ✓ WTLS engine is a state machine - gen_fsm
- ▶ Disadvantages
  - ✓ Not enough crypto support

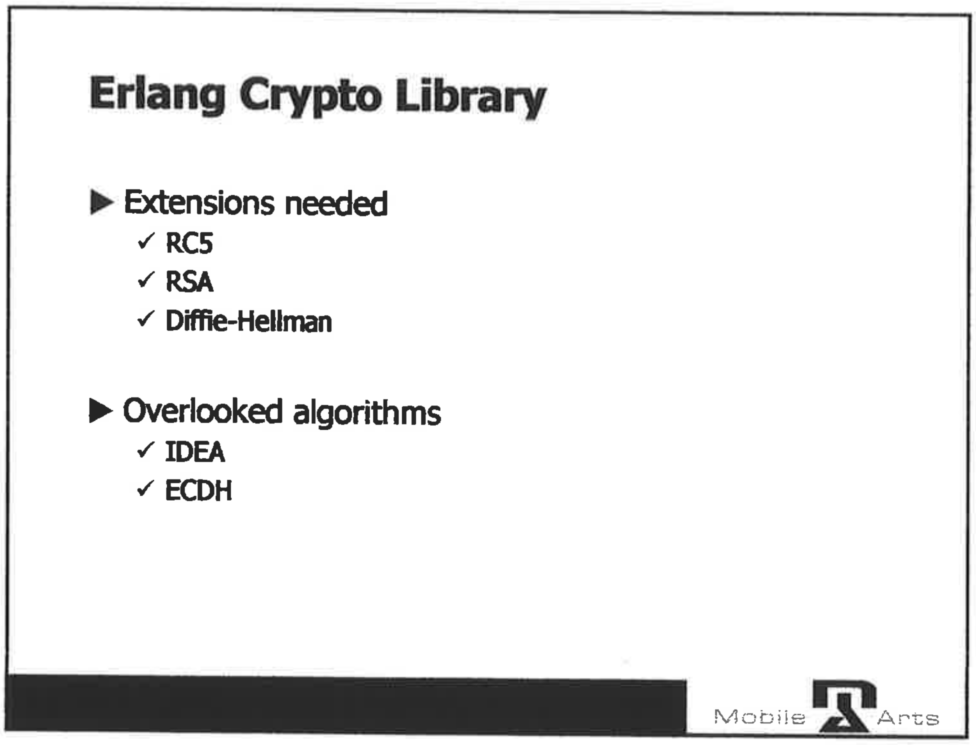
Mobile Arts



Erlang Crypto Library

- ▶ Supports
  - ✓ DES, 3DES
  - ✓ SHA, MD5
- ▶ WTLS also specifies
  - ✓ RC5, IDEA
  - ✓ RSA, Diffe-Hellman, ECDH

Mobile Arts

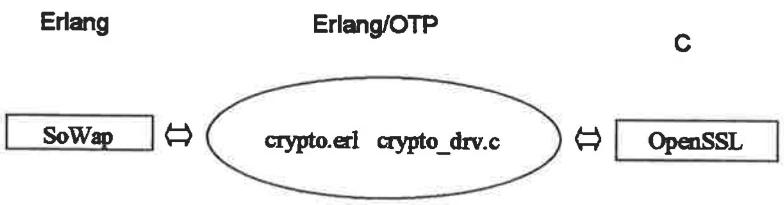


Erlang Crypto Library

- ▶ Extensions needed
  - ✓ RC5
  - ✓ RSA
  - ✓ Diffie-Hellman
- ▶ Overlooked algorithms
  - ✓ IDEA
  - ✓ ECDH

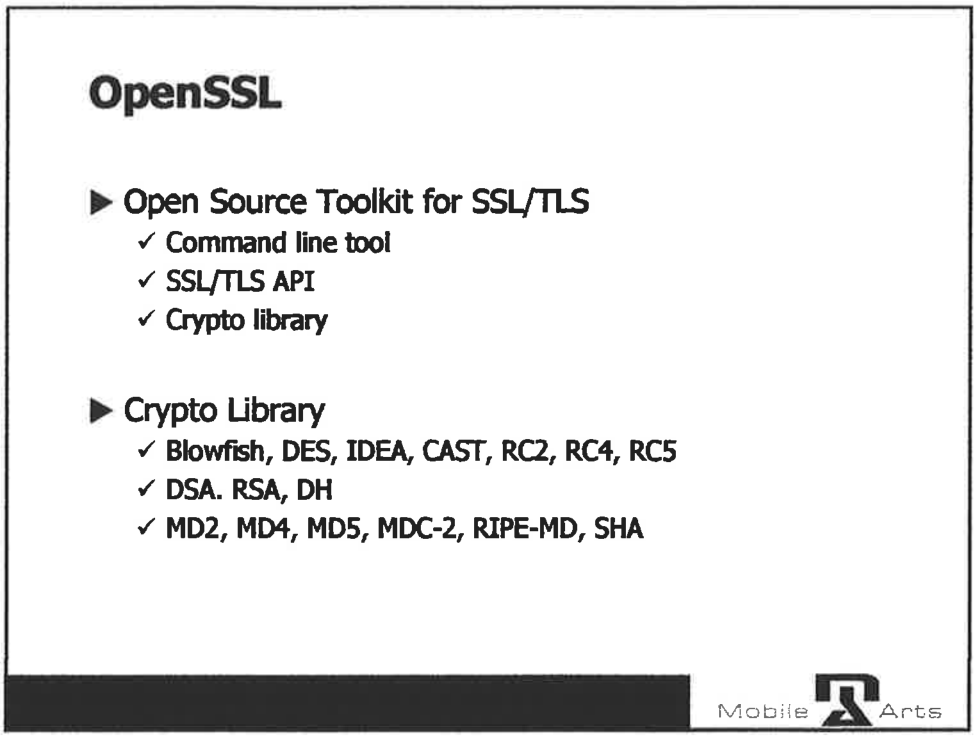
Mobile Arts

Erlang Crypto Library



Mobile Arts

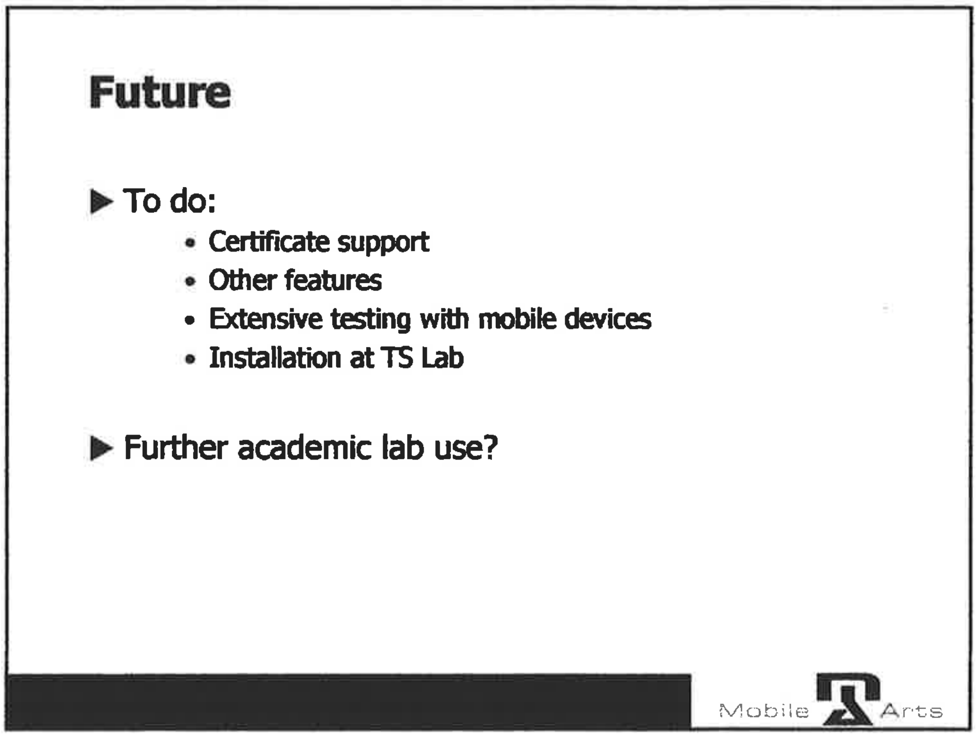
7



OpenSSL

- ▶ Open Source Toolkit for SSL/TLS
  - ✓ Command line tool
  - ✓ SSL/TLS API
  - ✓ Crypto library
- ▶ Crypto Library
  - ✓ Blowfish, DES, IDEA, CAST, RC2, RC4, RC5
  - ✓ DSA. RSA, DH
  - ✓ MD2, MD4, MD5, MDC-2, RIPE-MD, SHA

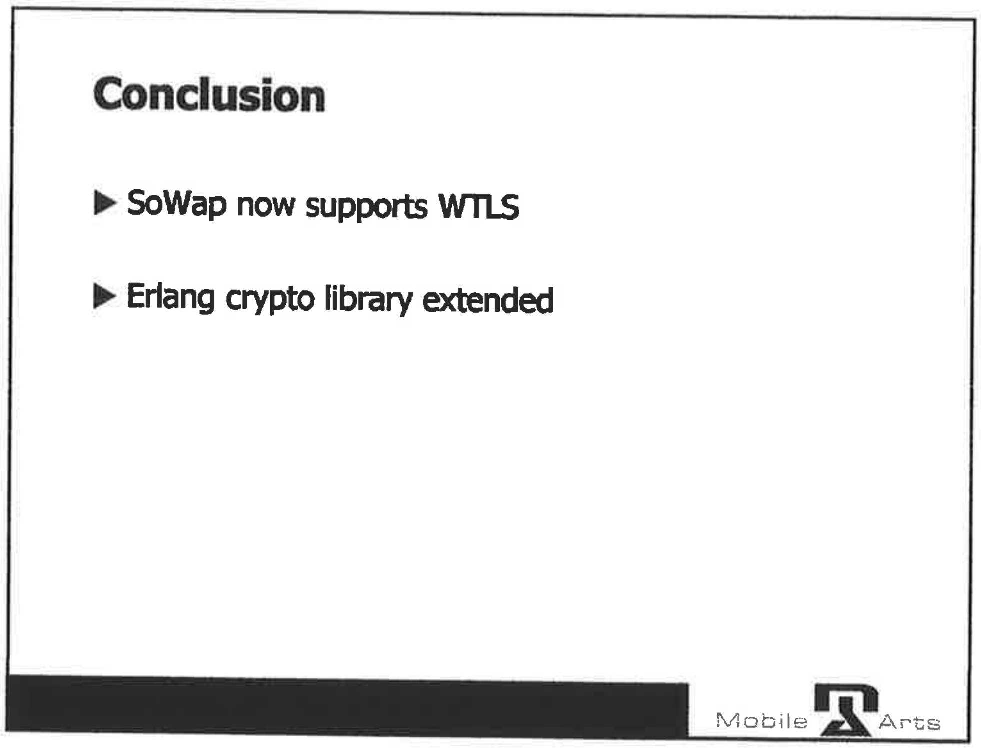
Mobile Arts



Future

- ▶ To do:
  - Certificate support
  - Other features
  - Extensive testing with mobile devices
  - Installation at TS Lab
- ▶ Further academic lab use?

Mobile Arts



Conclusion

▶ SoWap now supports WTLS

▶ Erlang crypto library extended

Mobile Arts

Learning Erlang and developing a SIP server/stack with 30k potential users

Fredrik Thulin <sip:ft@it.su.se>
Enheten för IT och media
Stockholms universitet

EUC2004

1(7)

Stockholm university telephony in 1997 :

- Ericsson MD110
- 5000 subscribers

2000 :

- Ericsson MD110 – 4200 subscribers
- Cisco CallManager – 800 subscribers

- Early VoIP adopters, but still no open standard

SIP

- Session Initiation Protocol
- IETF proposed standard (RFC3261, 2, 3, 4 and 5)
- Does not care about audio/video/whatever
- Instant messaging and presence
- Parsing is hard, even though (?) it's plain text
- Transactions are complex
- Few open source implementations as to date

Implementation

- Magnus Ahltorp, KTH made a couple of implementations in Perl, Python, C, ...
- Erlang implementation was the most viable one

- First Yxa snapshot released 2003-10-07

Personal Erlang experience

- No prior knowledge about functional programming
- Hard time understanding some syntax (strings are lists, [H | T] = "string")
- Assign once
- ...

Plans

- Build distributed SIP servers
  - for routing
  - for students (free VoIP/SIP-service)
  - for basic call service? evaluating.
- Distributed policy control (rate limiting etc.)
- Pluggable authentication modules
- External event logging with call context

Project

- http://www.stacken.kth.se/projekt/yxa/

Messaging with Erlang and Jabber

Erlang User Conference '04

21st. October 2004

Mickaël Rémond <mickael.remond@erlang-fr.org>

ERLANG projects

www.erlang-projects.org

1 (13)

What are XMPP and Jabber ?

- XMPP stands for eXtensible Messaging & Presence Protocol
- XMPP is a generic and extensible messaging protocol based on XML. It is now an IETF standard
- Jabber is an Instant Messaging protocol that rely on XMPP
- Very active community:
  - Several server implementations of XMPP and Jabber
  - Several client software and library

ERLANG

20

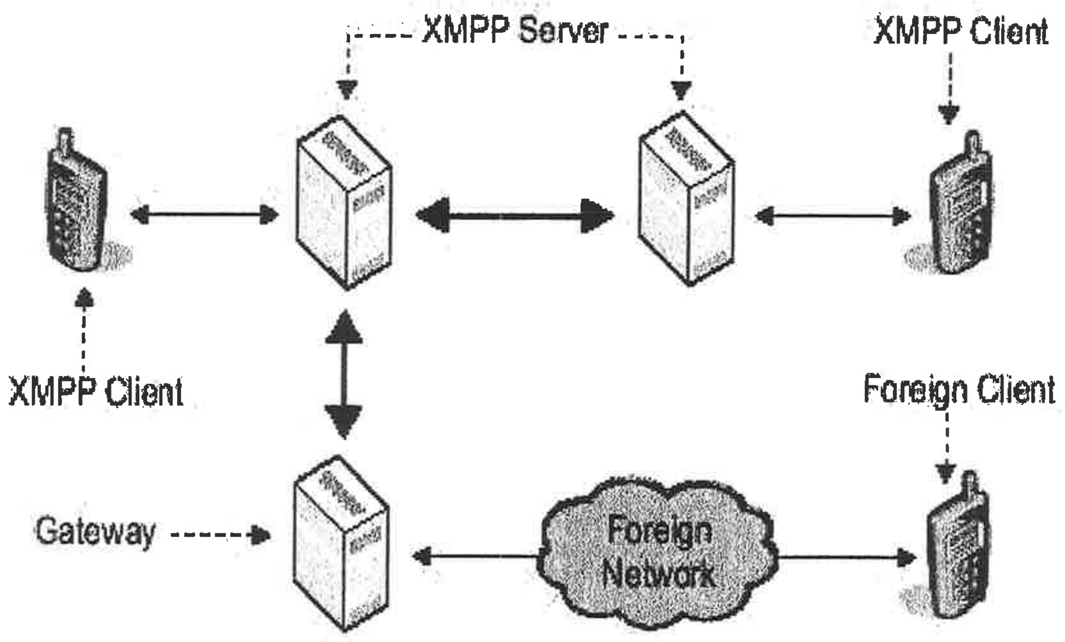
Fun things to do with XMPP and Erlang

- XMPP and Jabber servers are massively concurrent: a Jabber server must handle huge community of users.
  - A Jabber server in Erlang makes sense to handle massive concurrency (plus it's fun and challenging).
  - It can prove the reliability and ability of Erlang to handle concurrency
- XMPP server protocol is build around a complete XML API for client to server and server to server communications:
  - Developping Erlang software agent that plug on the bus is easy: Erlang expressiveness.
  - It allows to use the bus as a mediation layer between Erlang and non-Erlang software (Kind of web service but more simple and powerful)
  - Several levels of interaction for Erlang extension: Plug-in in the XMPP bus with a service protocol or XMPP client connected to the bus (client protocol)

ERLANG

XMPP bus design

- XMPP rely on a naturally distributed architecture
  - Includes server to server communication
  - Includes gateways to various other protocols



ERLANG

4

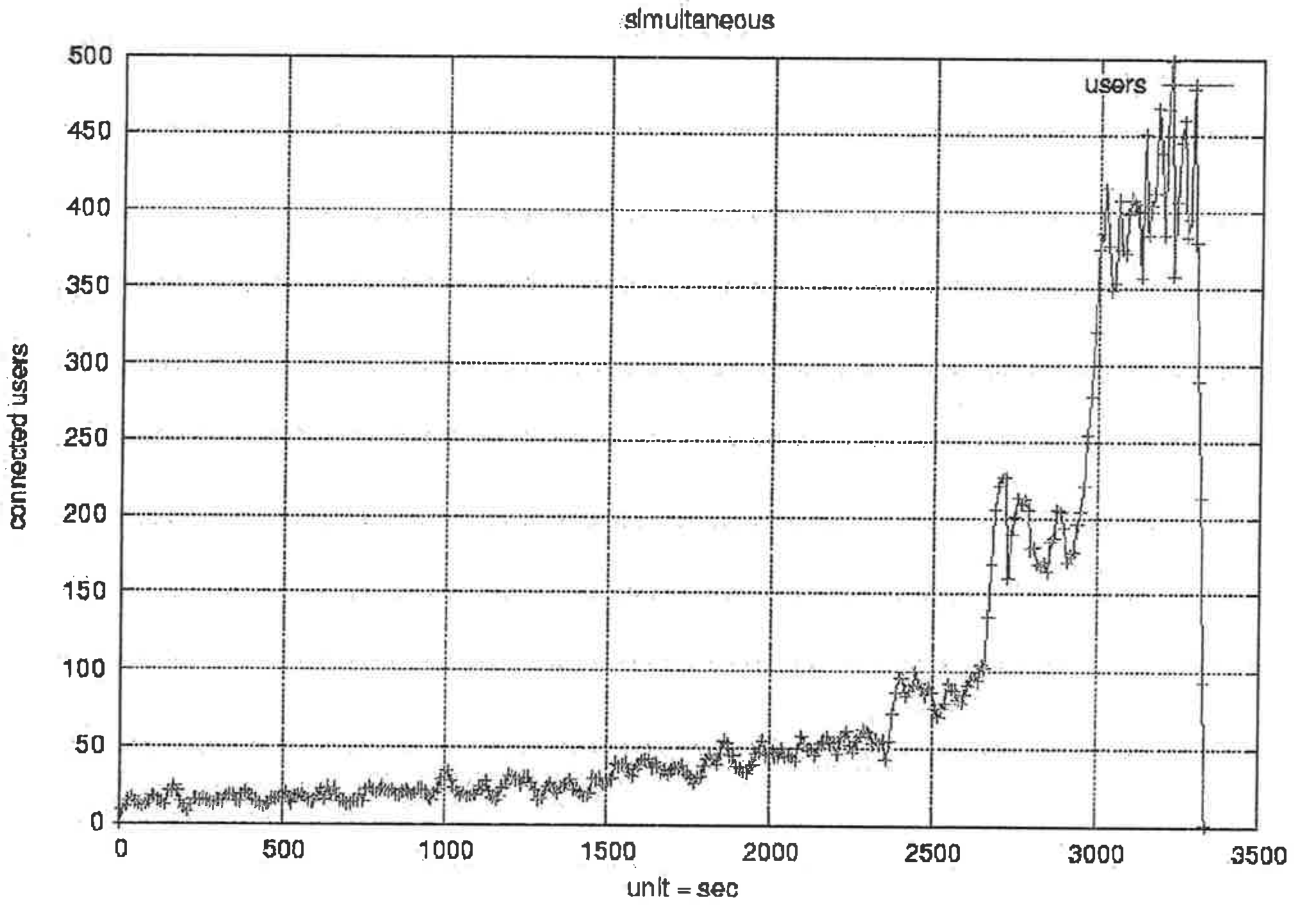
ejabberd

- ejabberd is an Erlang-based XMPP server implementation.
- It has been designed to support clustering, fault-tolerance and high-availability.
- It supports many features and extensions of the Jabber protocol:
  - Built-in Multi-User Chat service
  - Distributed database (Mnesia)
  - Built-in IRC transport
  - Built-in Publish-Subscribe service
  - Support for LDAP authentification
  - Service discovery
- It is more scalable than the most used open source implementation (Jabberd1.4 and Jabber2).

ERLANG

Benchmarks: how does ejabberd perform ?

- Jabber benchmarks realized with the **Tsunami** benchmarking tool.
- Comparison between ejabberd and classical Jabber implementation.



ERLANG

9

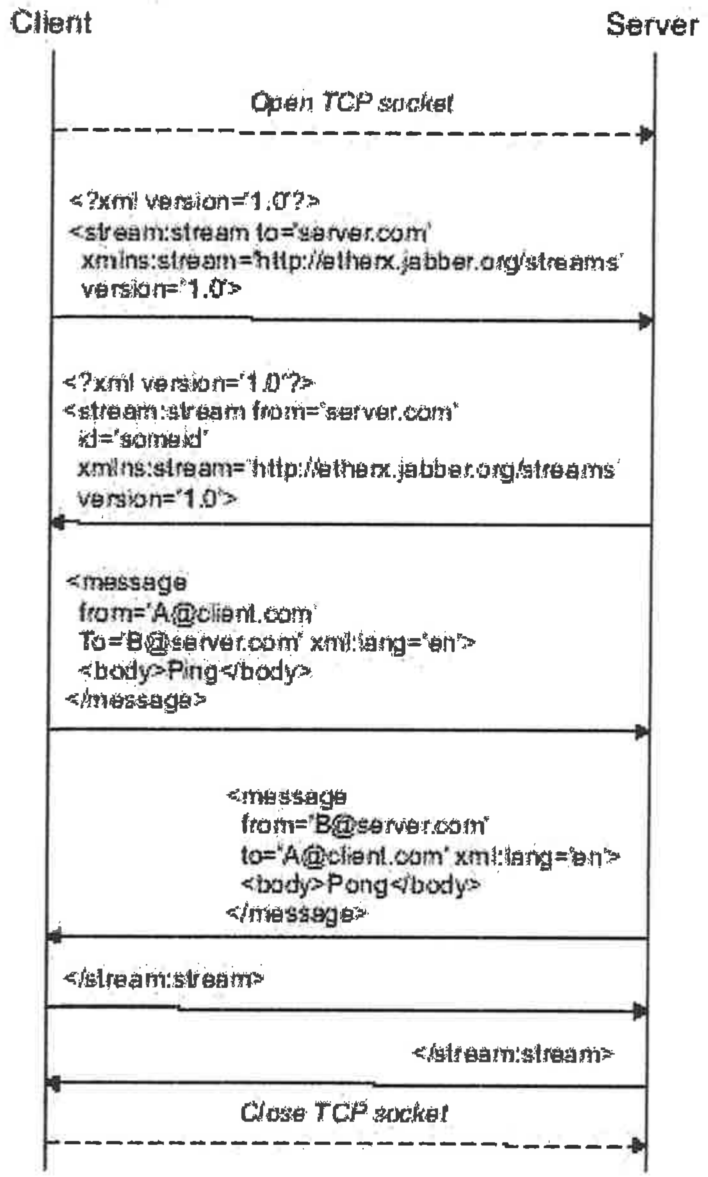
Fun evolutions and ideas

- **SIP / XMPP gateway**: Running together ejabberd and yxa to share user base and presence information.
-

ERLANG

What does the XMPP protocol looks like ?

- **Interleaved XML streams**: Client and serverstreams form an XML document.
- First level tag: **<stream>**
- Three types of second levels tags:
  - **message**: asynchronous communications
  - **iq**: Synchronous communications
  - **presence**: presence and status data



```
<?xml version='1.0'?>
<stream:stream to='server.com'
  xmlns:stream='http://etherx.jabber.org/streams'
  version='1.0'>
```

```
<?xml version='1.0'?>
<stream:stream from='server.com'
  id='someid'
  xmlns:stream='http://etherx.jabber.org/streams'
  version='1.0'>
```

```
<message
  from='A@client.com'
  To='B@server.com' xml:lang='en'>
  <body>Ping</body>
</message>
```

```
<message
  from='B@server.com'
  to='A@client.com' xml:lang='en'>
  <body>Pong</body>
</message>
```

```
</stream:stream>
```

```
</stream:stream>
```

∞

J-EAI: an XMPP based integration tool

- J-EAI is an **Enterprise Application Integration** tool.
- It is intended to **control and organize** data streams in a given information system.
- It allow **transformation**, **routing**, **queueing** of all the data exchanges between application in a company.

ERLANG

Jabberlang: Helper library to write XMPP client in Erlang

- Jabberlang is a client library to write XMPP client in Erlang.
- It is implemented as an Erlang behaviour.
- It allow to write XMPP services in Erlang.
- Can be used for inter Erlang programs communication (term to binary <-> binary_to_term)

Examples:

ERLANG

Using XMPP for Erlang distribution ?

- Using XMPP for Erlang distribution could allow to develop distributed applications running through the Internet.
- This approach can solve some security aspects: Rosters configuration can decide if two process are allowed to exchanged messages.
- SSL is supported.
- Performance penalty: only relevant for non heavy-loaded critical message passing applications.
- Application design in such a way could switch to Erlang standard distribution to get more performance.

ERLANG

References

- http://www.jabber.org
- http://www.jabber.org/press/2004-10-04.php
- http://ejabberd.jabberstudio.org/
- http://www.erlang-projects.org/

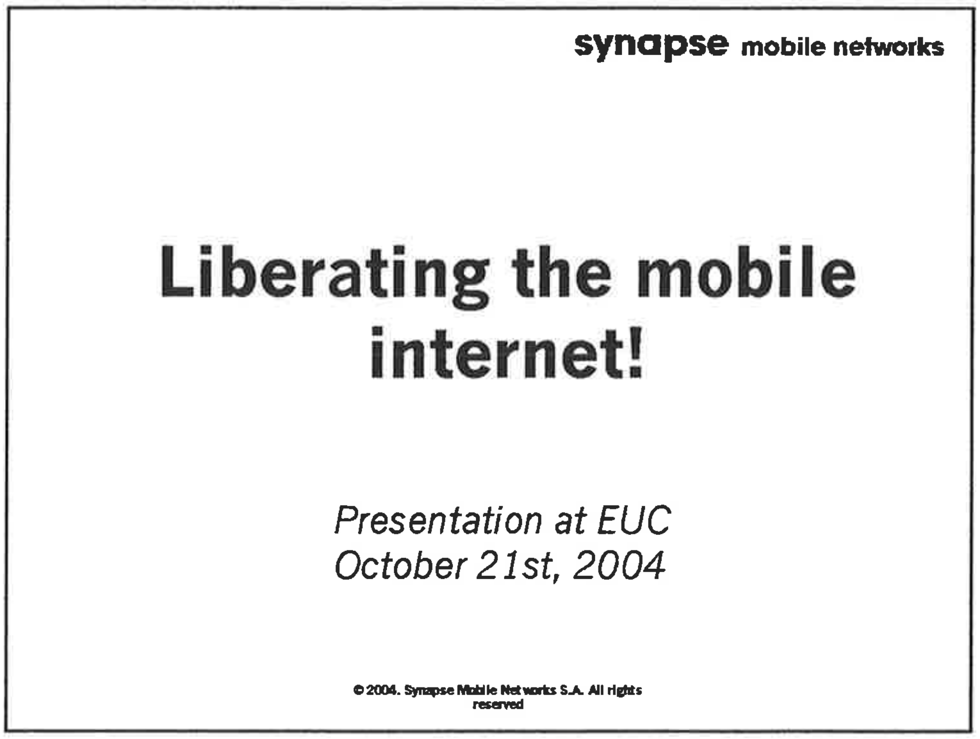
ERLANG

Messaging with Erlang and Jabber

Questions

ERLANG

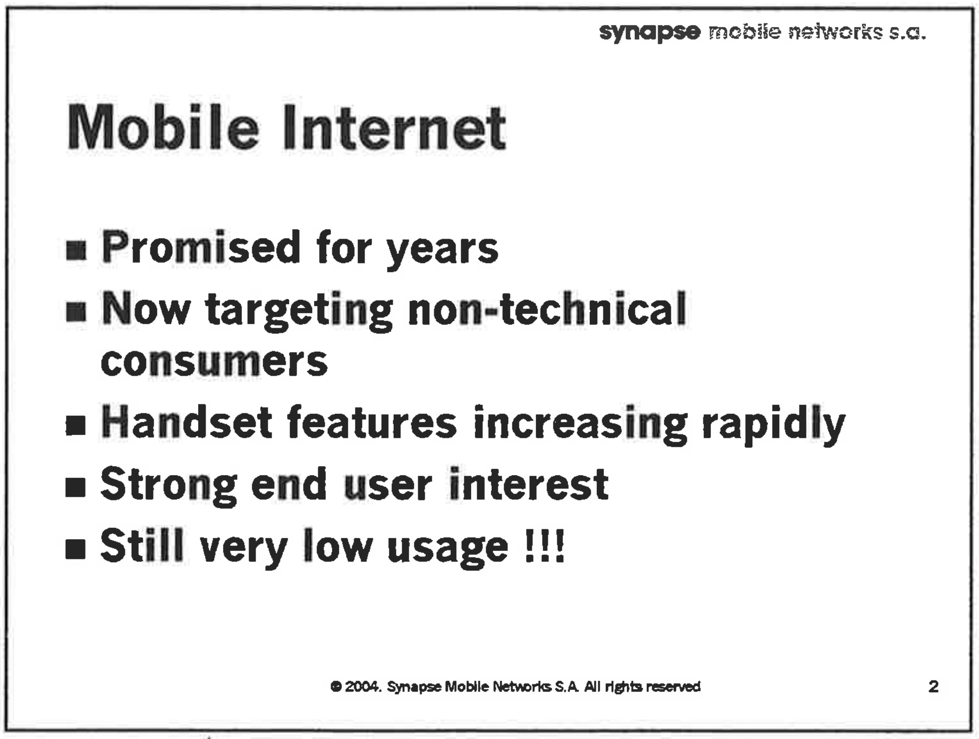
1 (13)



synapse mobile networks

Liberating the mobile internet!

Presentation at EUC
October 21st, 2004

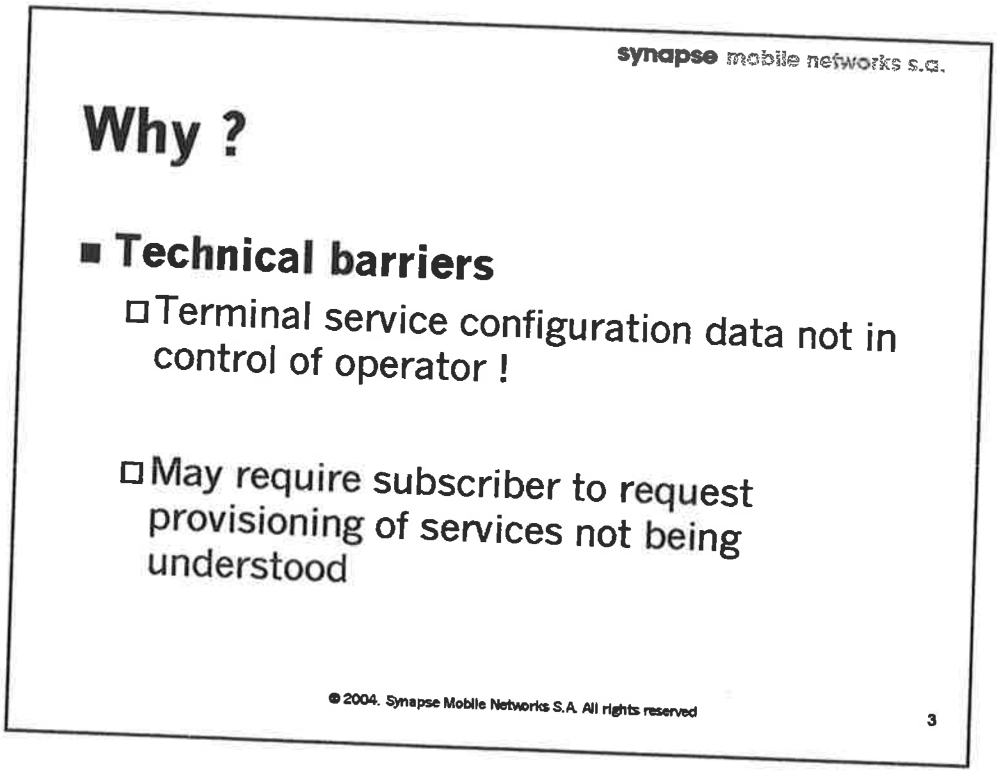


synapse mobile networks s.a.

Mobile Internet

- Promised for years
- Now targeting non-technical consumers
- Handset features increasing rapidly
- Strong end user interest
- Still very low usage !!!

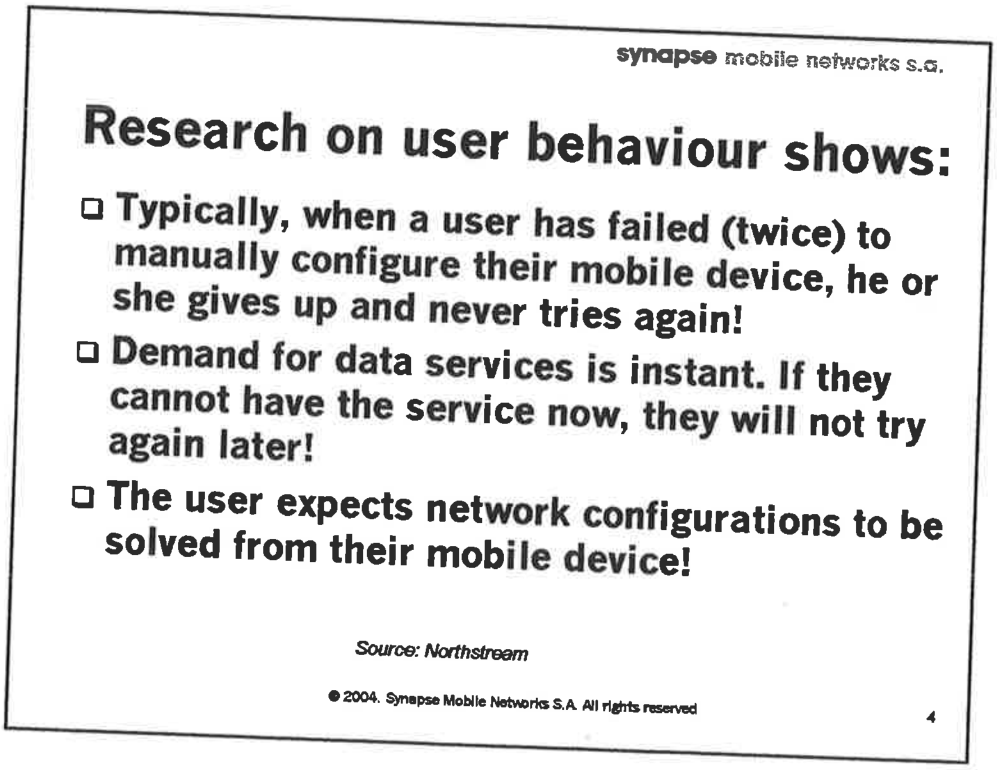
2



synapse mobile networks s.a.

Why ?

- **Technical barriers**
  - Terminal service configuration data not in control of operator !
  - May require subscriber to request provisioning of services not being understood



synapse mobile networks s.a.

Research on user behaviour shows:

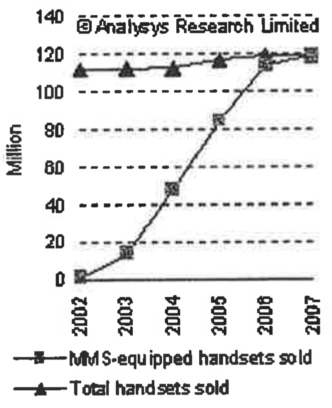
- **Typically, when a user has failed (twice) to manually configure their mobile device, he or she gives up and never tries again!**
- **Demand for data services is instant. If they cannot have the service now, they will not try again later!**
- **The user expects network configurations to be solved from their mobile device!**

Source: Northstream

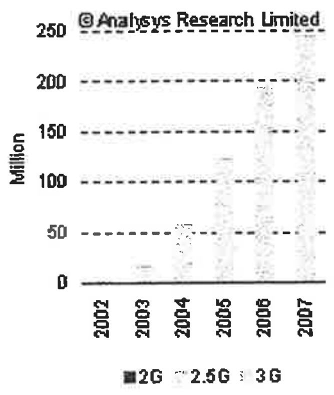
3

synapse mobile networks s.a.

MMS forecast for Western Europe



Yearly sales of MMS phones.

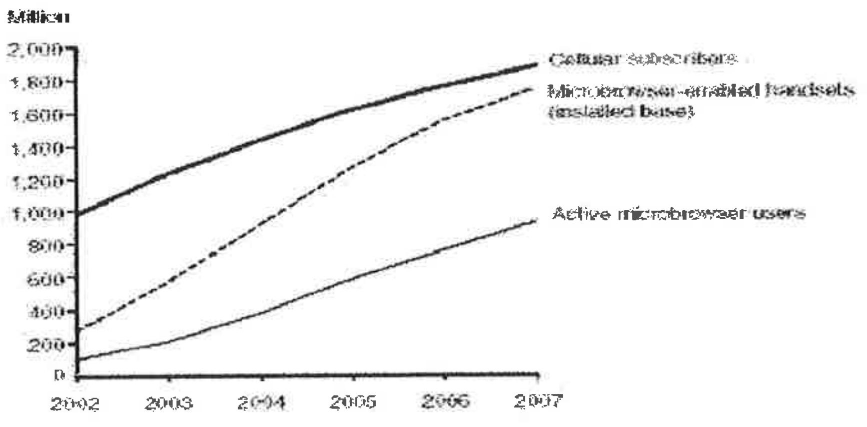


Total no. of MMS phones.

Source: Analysis Research Ltd

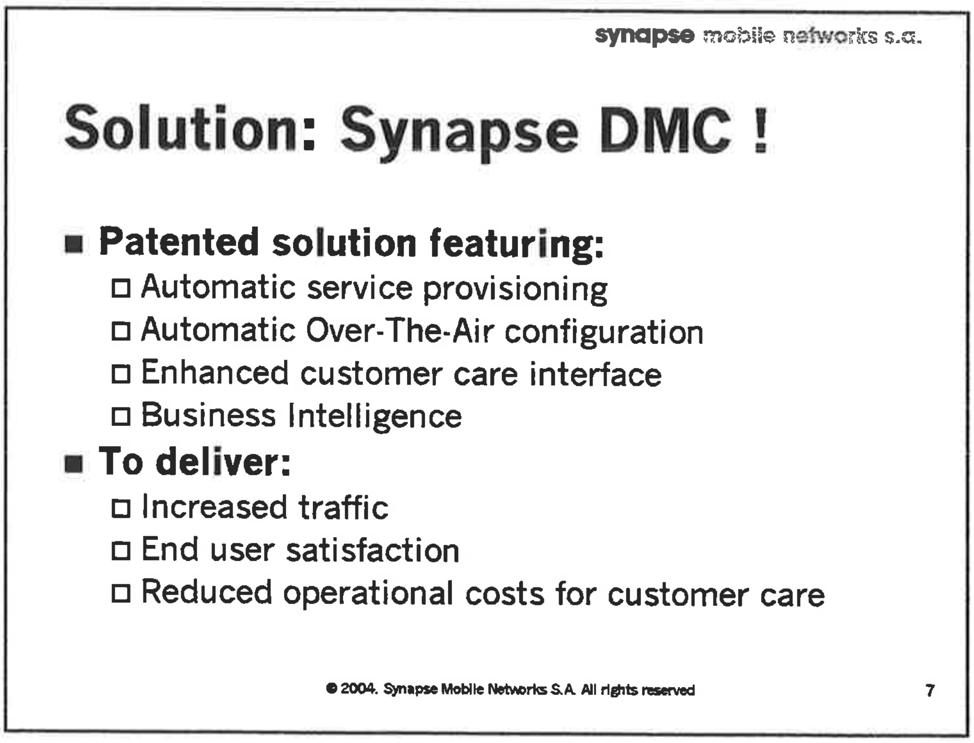
synapse mobile networks s.a.

Gap of un-configured devices



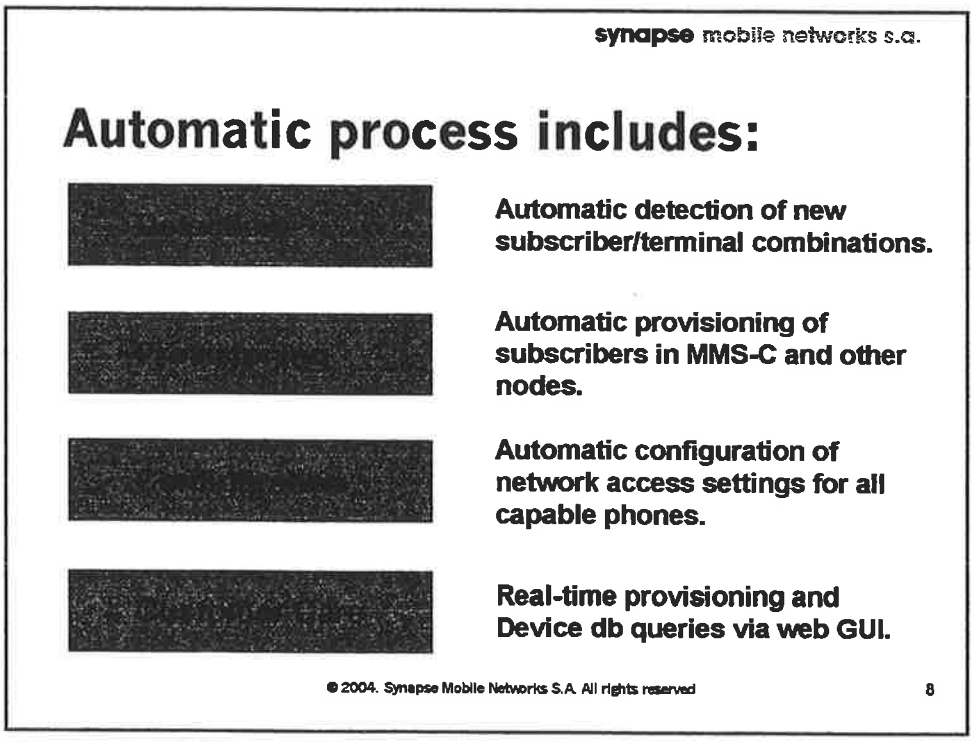
Source: Ovum / Ovum Forecasts: Global Wireless Markets 2002-2006/Chapter 6)

4



Solution: Synapse DMC !

- **Patented solution featuring:**
  - ☐ Automatic service provisioning
  - ☐ Automatic Over-The-Air configuration
  - ☐ Enhanced customer care interface
  - ☐ Business Intelligence
- **To deliver:**
  - ☐ Increased traffic
  - ☐ End user satisfaction
  - ☐ Reduced operational costs for customer care



Automatic process includes:

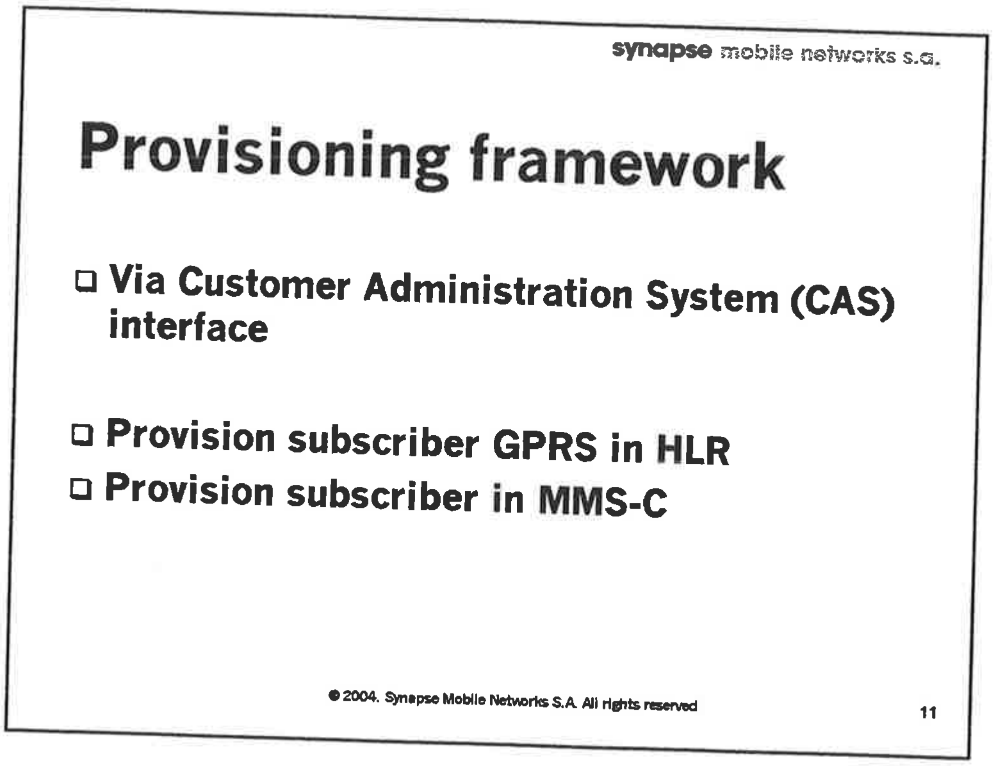
[illegible]	**Automatic detection of new subscriber/terminal combinations.**
[illegible]	**Automatic provisioning of subscribers in MMS-C and other nodes.**
[illegible]	**Automatic configuration of network access settings for all capable phones.**
[illegible]	**Real-time provisioning and Device db queries via web GUI.**

Device Detection

- Detect new combinations of subscriber devices
- Identify device capabilities
- Device database with
  - > 6000 device models
  - > 1300 MMS capable device models
- Device alias management

Automatic detection options

- Vendor independent monitoring of network traffic
  - Intelligent monitoring probes (A & D interfaces)
- Concentrated methods
  - Monitoring of extended MSC – EIR (F interface)
  - Integrated extended EIR (active)
- Vendor specific MSC/HLR triggers (when available)
  - Ericsson
  - Nokia
- Via CDR file processing

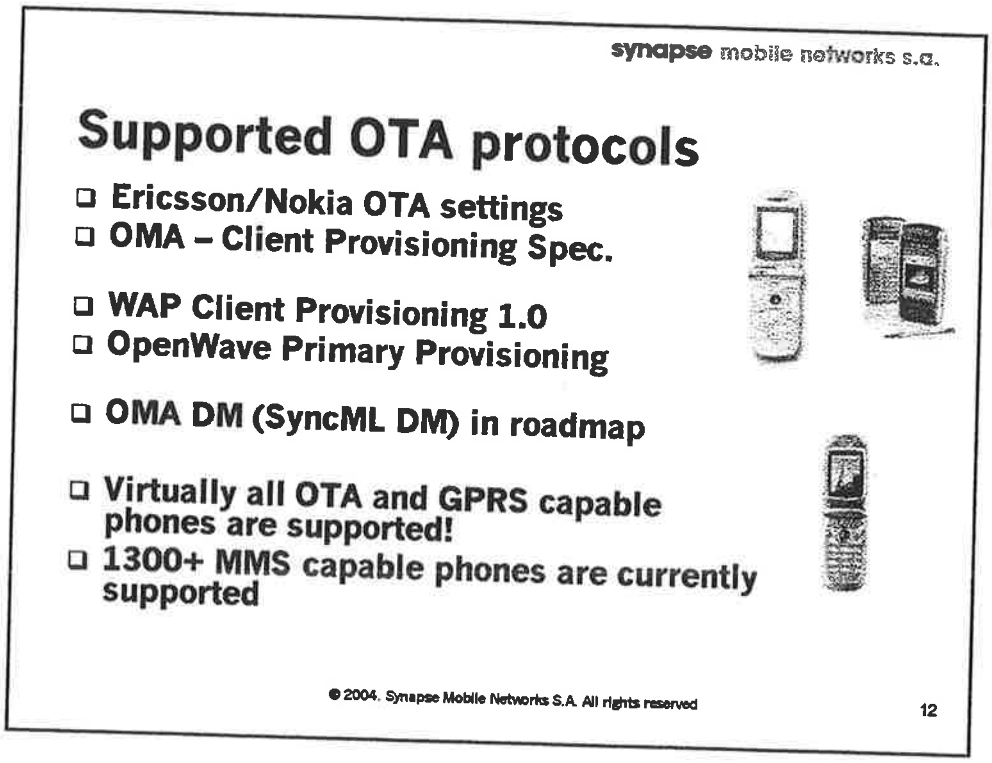


synapse mobile networks s.a.

Provisioning framework

- Via Customer Administration System (CAS) interface
- Provision subscriber GPRS in HLR
- Provision subscriber in MMS-C

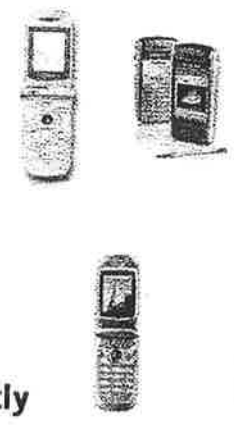
11



synapse mobile networks s.a.

Supported OTA protocols

- Ericsson/Nokia OTA settings
- OMA – Client Provisioning Spec.
- WAP Client Provisioning 1.0
- OpenWave Primary Provisioning
- OMA DM (SyncML DM) in roadmap
- Virtually all OTA and GPRS capable phones are supported!
- 1300+ MMS capable phones are currently supported



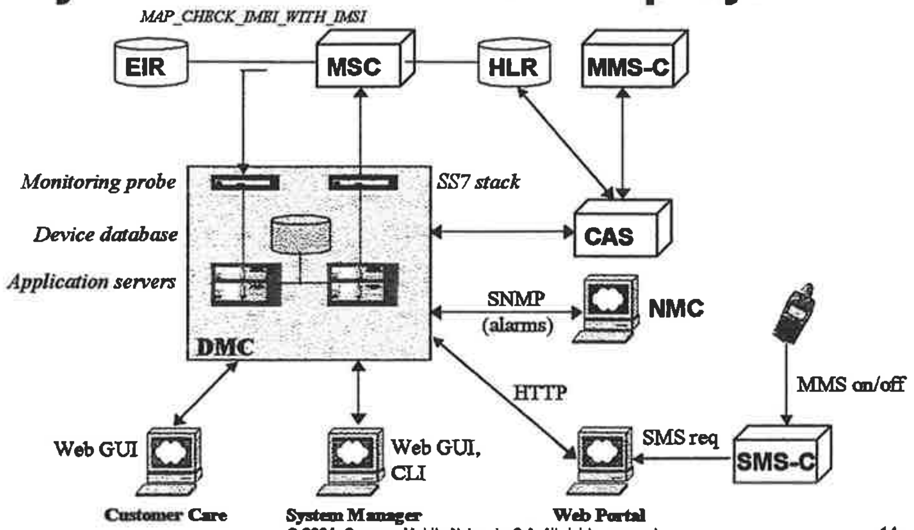
12

7

Automatic MMS provisioning example:

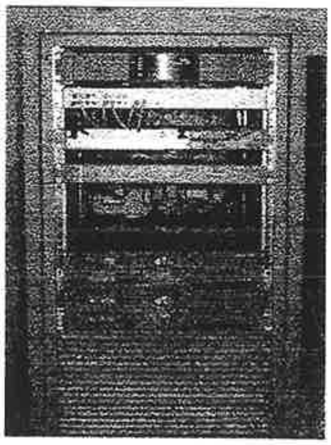
- Detect MMS capable terminal devices
- Check HLR subscriber status (Operator barring and MSISDN, GPRS status)
- Check for pre-paid subscribers
- Check MMS user status
- Provision subscriber in MMS-C and HLR
- Send out notification (if requested)
- Send out device settings
- Send out Welcome MMS message

System overview – CYTA project



DMC 2.0 installation example

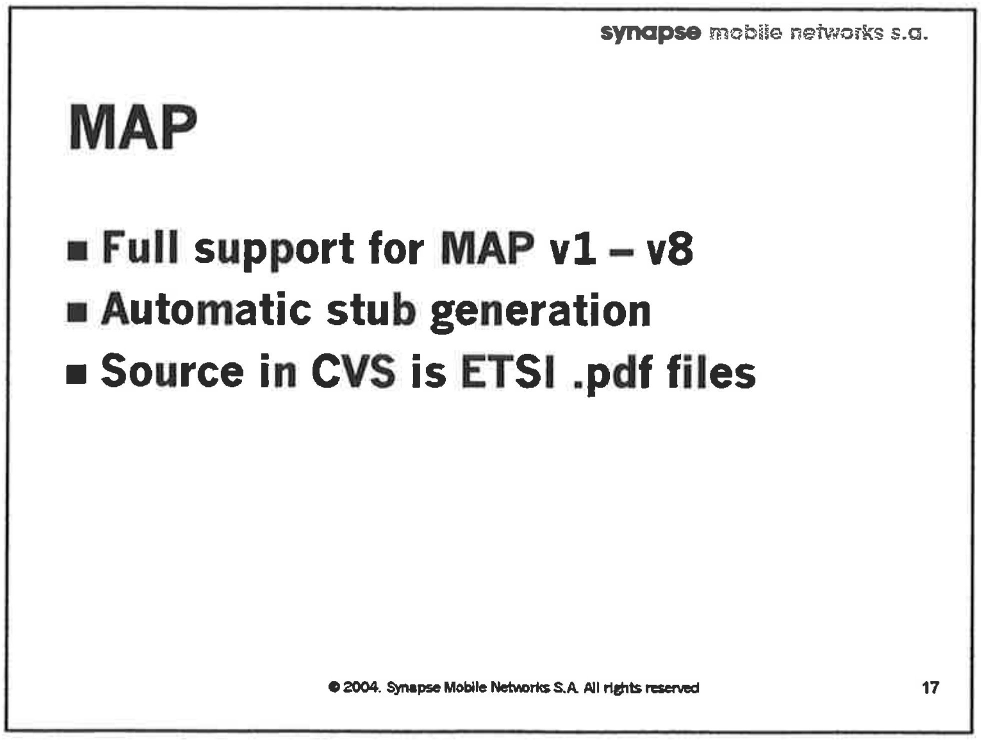
- Firewall
- Ethernet switch
- SS7 stack
- Corelatus GTH probe
- 2 x Sun Fire V240



System environment

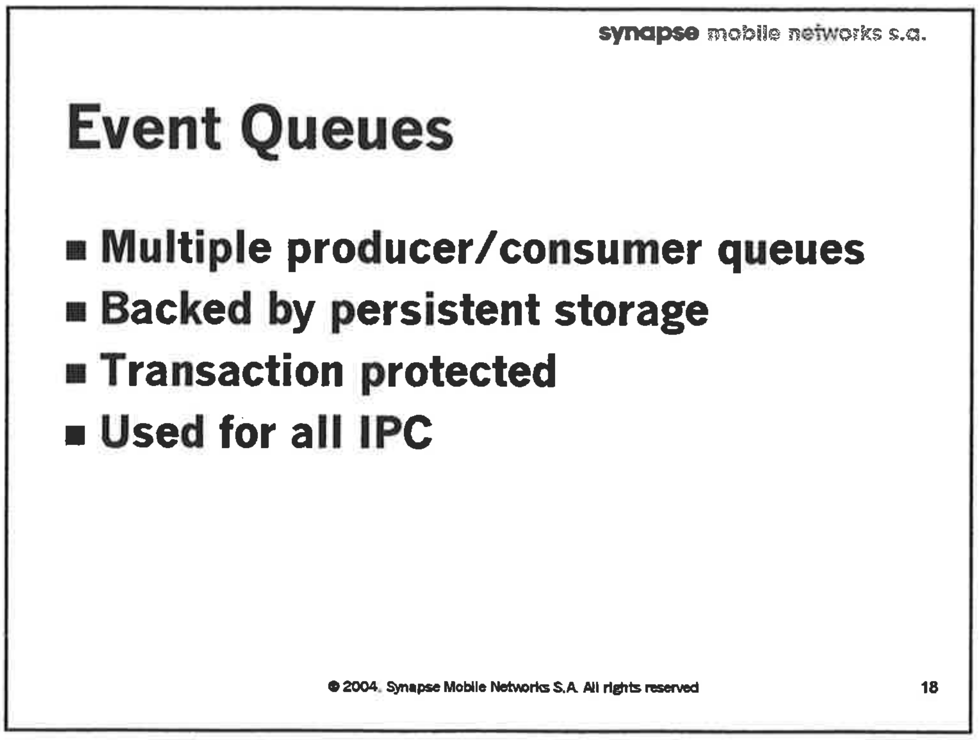
- OTP R9C-0 design base
- Unimind cluster
- Solaris 8 (and Linux)
- TietoEnator Portable SS7
- Synapse 3GPP/GSM MAP
- Rapid Installer

9



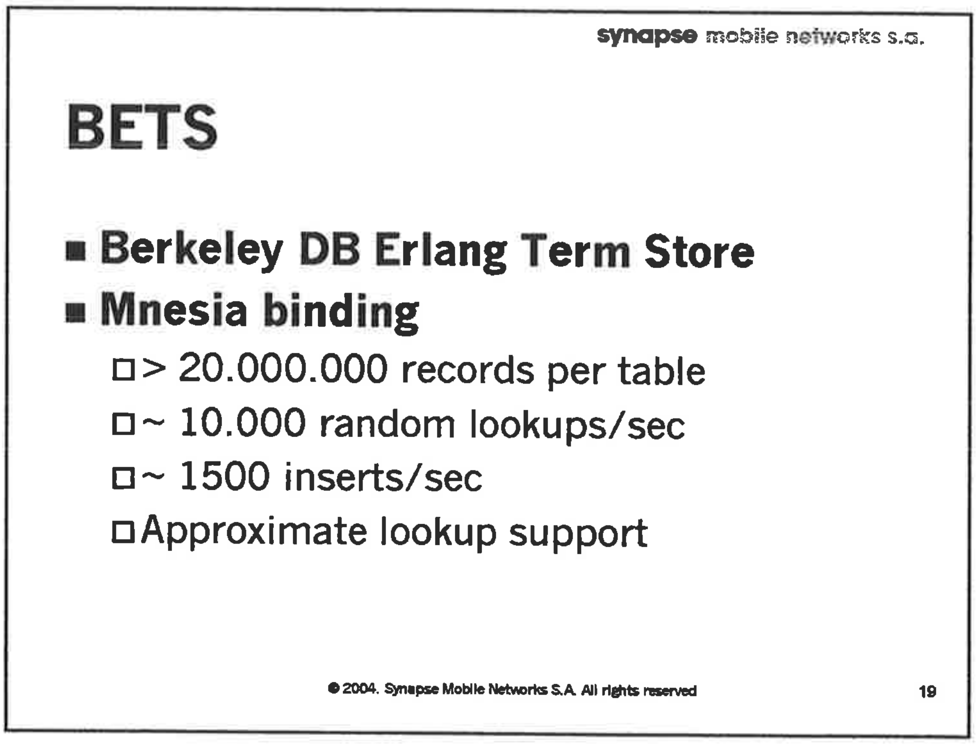
MAP

- Full support for MAP v1 – v8
- Automatic stub generation
- Source in CVS is ETSI .pdf files



Event Queues

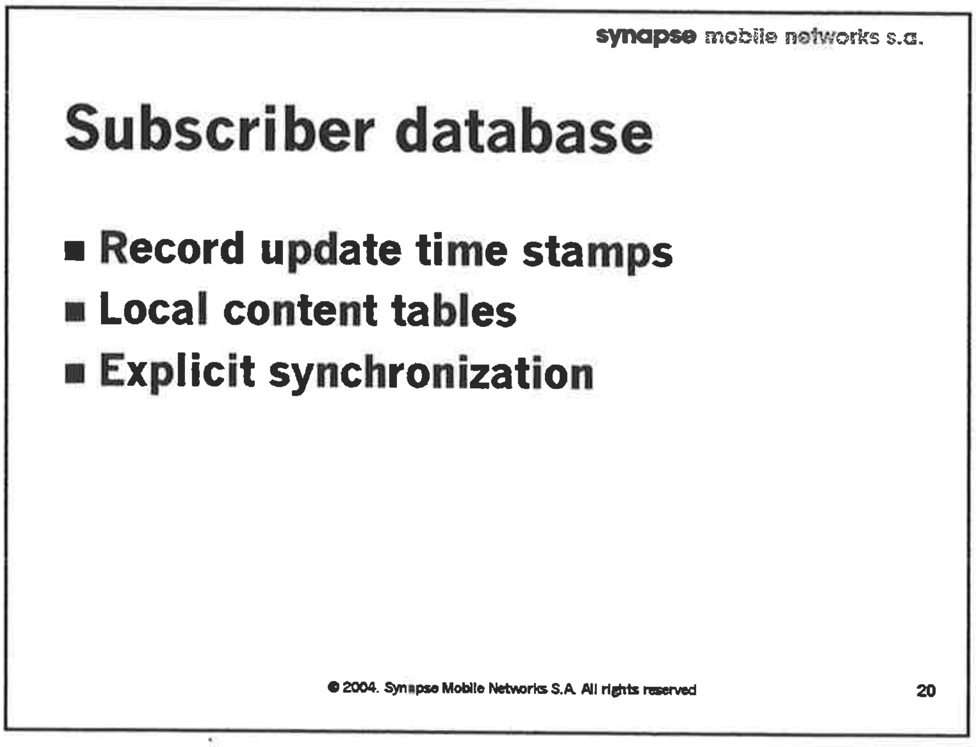
- Multiple producer/consumer queues
- Backed by persistent storage
- Transaction protected
- Used for all IPC



synapse mobile networks s.a.

BETS

- Berkeley DB Erlang Term Store
- Mnesia binding
  - > 20.000.000 records per table
  - ~ 10.000 random lookups/sec
  - ~ 1500 inserts/sec
  - Approximate lookup support

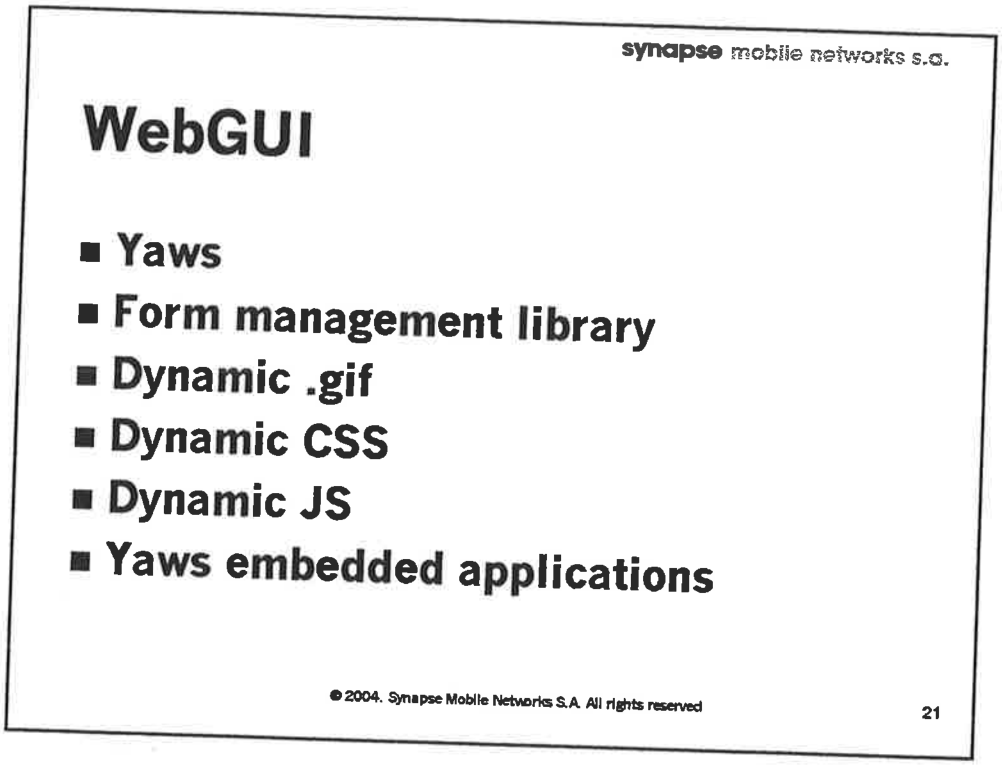


synapse mobile networks s.a.

Subscriber database

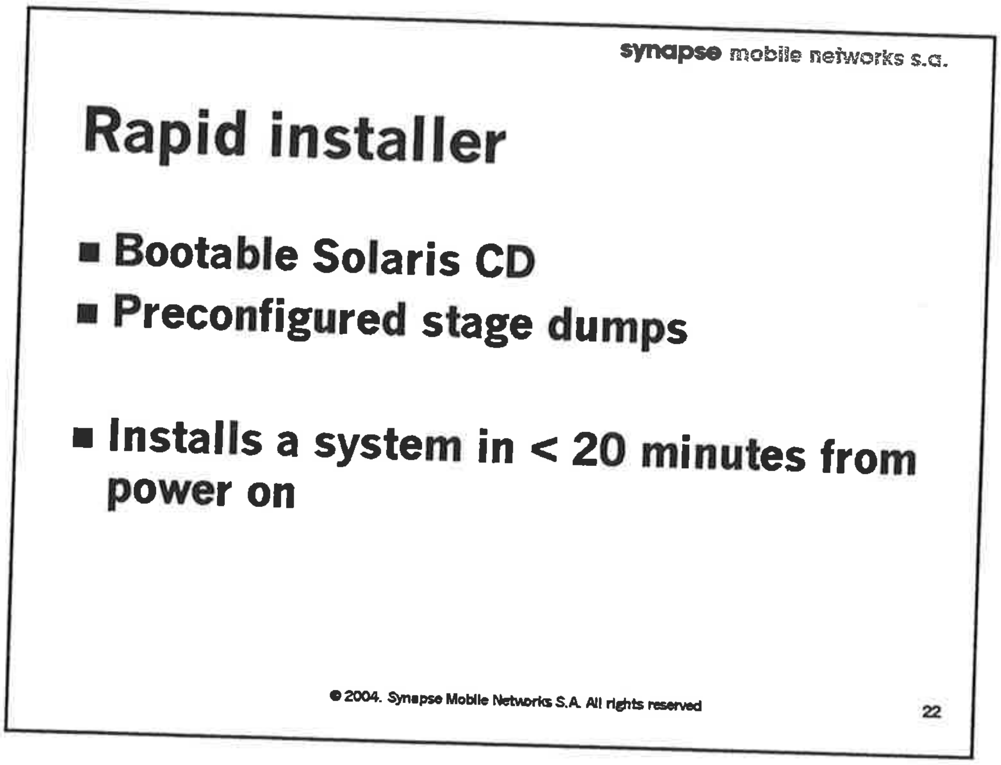
- Record update time stamps
- Local content tables
- Explicit synchronization

11


synapse mobile networks s.a.

WebGUI

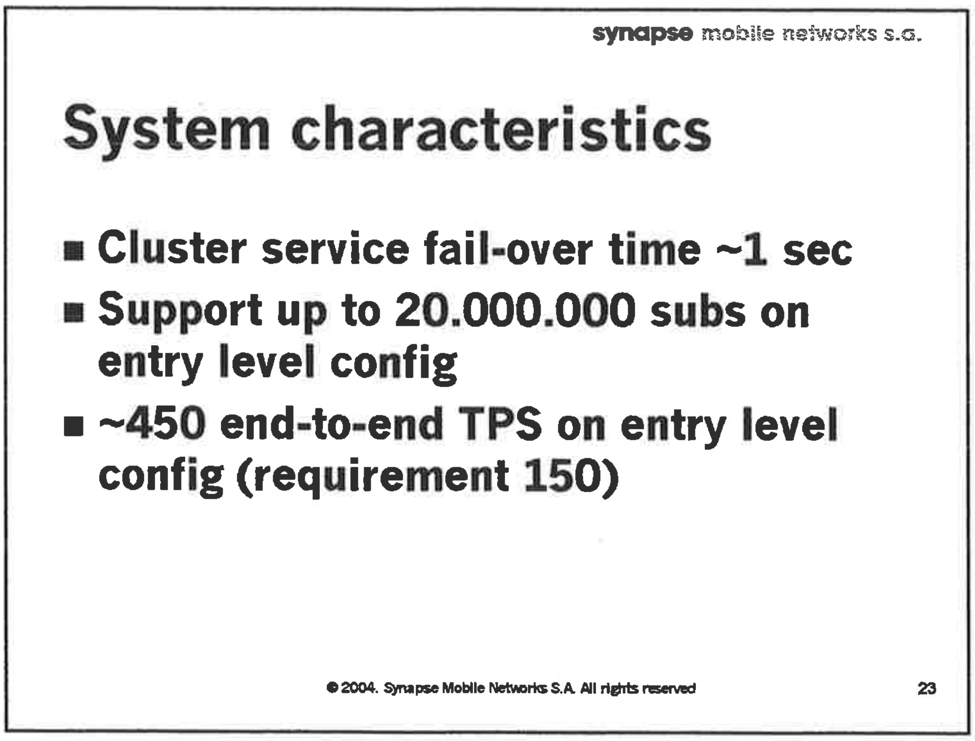
- Yaws
- Form management library
- Dynamic .gif
- Dynamic CSS
- Dynamic JS
- Yaws embedded applications


synapse mobile networks s.a.

Rapid installer

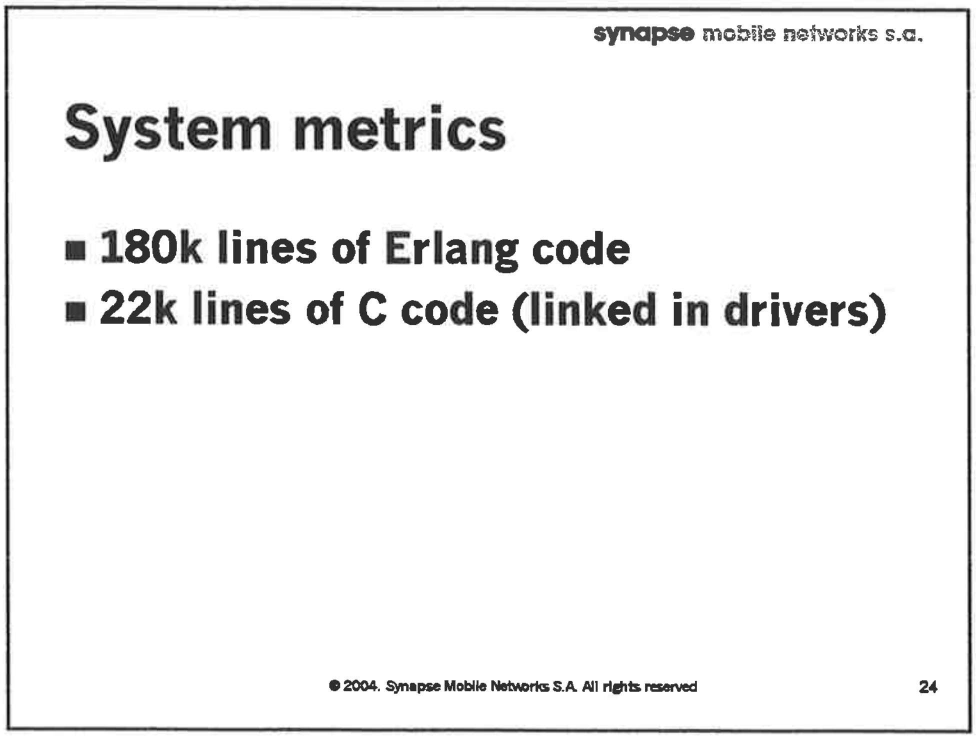
- Bootable Solaris CD
- Preconfigured stage dumps

- Installs a system in < 20 minutes from power on



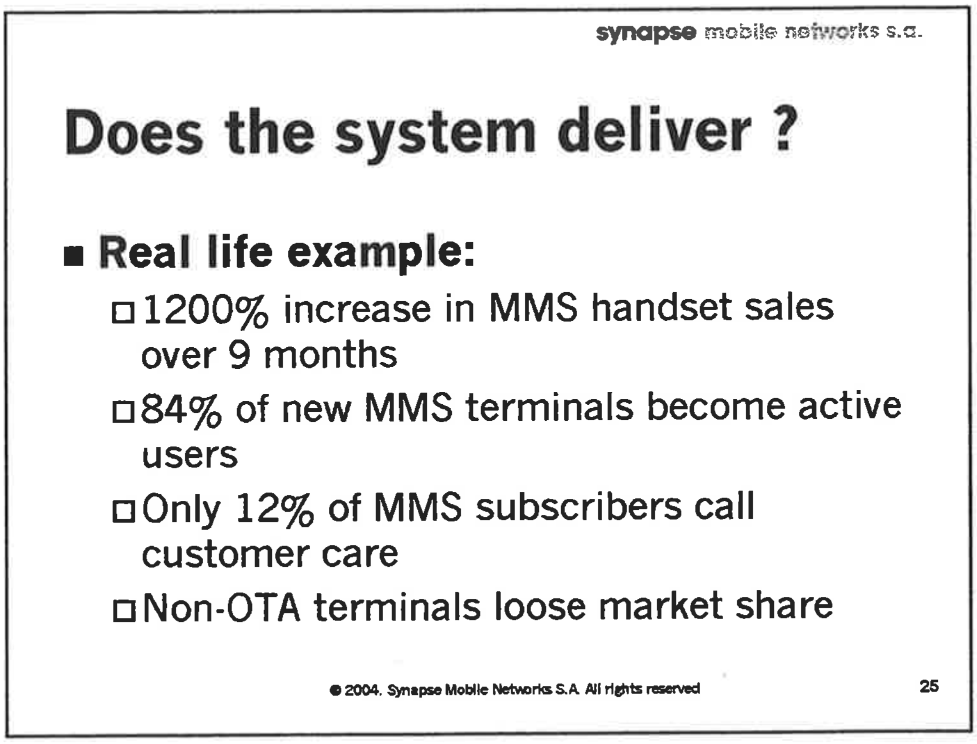
System characteristics

- Cluster service fail-over time ~1 sec
- Support up to 20.000.000 subs on entry level config
- ~450 end-to-end TPS on entry level config (requirement 150)



System metrics

- 180k lines of Erlang code
- 22k lines of C code (linked in drivers)

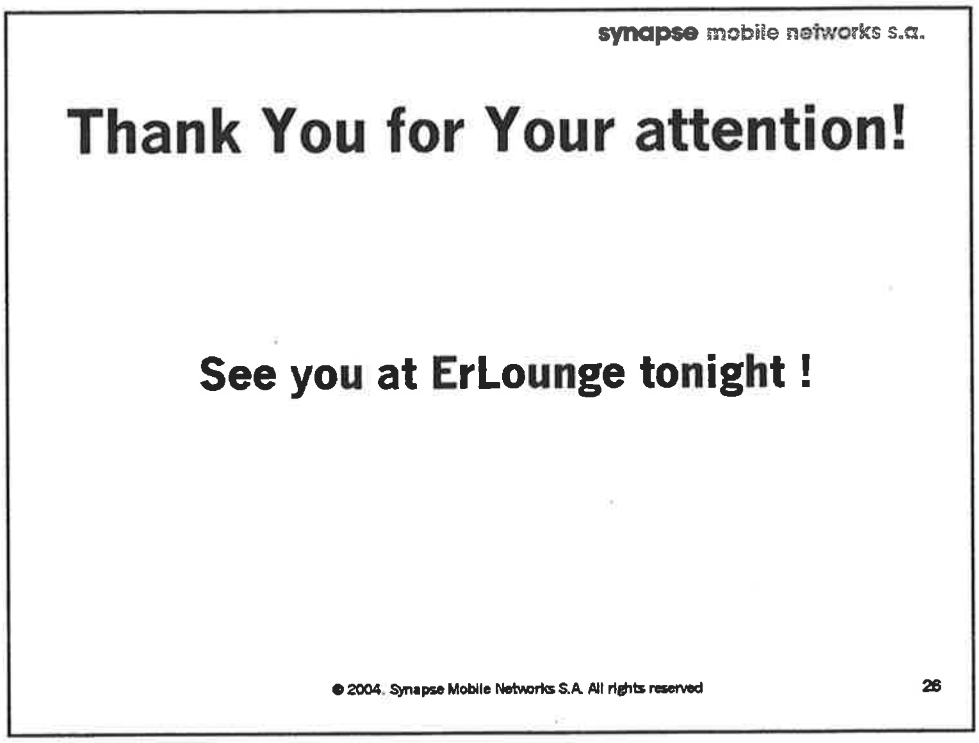


synapse mobile networks s.a.

Does the system deliver ?

- **Real life example:**
  - □ 1200% increase in MMS handset sales over 9 months
  - □ 84% of new MMS terminals become active users
  - □ Only 12% of MMS subscribers call customer care
  - □ Non-OTA terminals loose market share

25



synapse mobile networks s.a.

Thank You for Your attention!

See you at ErLounge tonight !

26

1(16)

Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story

Tobias Lindahl and Konstantinos Sagonas

Computing Science, Dept. of Information Technology, Uppsala University, Sweden
{Tobias.Lindahl,Konstantinos.Sagonas}@it.uu.se

Abstract. In safety-critical and high-reliability systems, software development and maintenance are costly endeavors. The cost can be reduced if software errors can be identified through automatic tools such as program analyzers and compile-time software checkers. To this effect, this paper describes the architecture and implementation of a software tool that uses lightweight static analysis to detect discrepancies (i.e., software defects such as exception-raising code or hidden failures) in large commercial telecom applications written in Erlang. Our tool, starting from virtual machine bytecode, discovers, tracks, and propagates type information which is often implicit in Erlang programs, and reports warnings when a variety of type errors and other software discrepancies are identified. Since the analysis currently starts from bytecode, it is completely automatic and does not rely on any user annotations. Moreover, it is effective in identifying software defects even in cases where source code is not available, and more specifically in legacy software which is often employed in high-reliability systems in operation, such as telecom switches. We have applied our tool to a handful of real-world applications, each consisting of several hundred thousand lines of code, and describe our experiences and the effectiveness of our techniques.

Keywords: Compile-time program checking, software development, software tools, defect detection, software quality assurance.

1 Introduction

All is fair in love and war, even trying to add a static type system in a dynamically typed programming language. Software development usually starts with love and passion for the process and its outcome, then passes through a long period of caring for (money making) software applications by simply trying to maintain them, but in the end it often becomes a war, the *war against software bugs*, that brings sorrow and pain to developers. In this war, the software defects will use all means available to them to remain in their favorite program. Fortunately, their primary weapon is concealment, and once identified, they are often relatively easy to kill.

In the context of statically typed programming languages, the type system aids the developer in the war against software bugs by automatically identifying type errors at compile time. Unfortunately, the price to pay for this victory is the compiler rejecting all programs that cannot be proved type-correct by the currently employed type system.

This starts another war, the *war against the type system*, which admittedly is a milder one. The only way for programmers to fight back in this war is to rewrite their programs. (Although occasionally the programming language developers help the programmers in fighting this war by designing a bigger weapon, i.e., a more refined type system).

Dynamically typed programming languages avoid getting into this second war. Instead, they adopt a more or less "anything goes" attitude by accepting all programs, and relying on type tests during runtime to prevent defects from fighting back in a fatal way. Sometimes these languages employ a less effective weapon than a static type system, namely a *soft type system*, which provides a limited form of type checking. To be effective, soft type systems often need guidance by manual annotations in the code. Soft typing will not reject any program, but will instead just inform the user that the program could not be proved type-correct. In the context of the dynamically typed programming language ERLANG, attempts have been made to develop such soft type systems, but so far none of them has gained much acceptance in the community. We believe the main reasons for this is the developers' reluctance to invest time (and money) in altering their already existing code and their habits (or personal preferences). We remark that this is not atypical: just think of other programming language communities like e.g., that of C.

Instead of devising a full-scale type checker that would need extensive code alterations in the form of type annotations to be effective, we pragmatically try to adapt our weapon's design to the programming style currently adhered to by ERLANG programmers. We have developed a lightweight type-based static analysis for finding *discrepancies* (i.e., software defects such as exception-raising code, hidden failures, or redundancies such as unreachable code) in programs without having to alter their source in any way. The analysis does not even need access to the source, since its starting point is virtual machine bytecode. However, the tool has been developed to be extensible in an incremental way (i.e., with the ability to take source code into account and benefit from various kinds of user annotations), once it has gained acceptance in its current form.

The actual tool, called DIALYZER,[1] allows its user to find discrepancies in ERLANG applications, based on information both from single modules and from an application-global level. It has so far been applied to programs consisting of several thousand lines of code from real-world telecom applications, and has been surprisingly effective in locating discrepancies in heavily used, well-tested code.

After briefly introducing the context of our work in the next section, the main part of the paper consists of a section which explains the rationale and main methods employed in the analysis (Sect. 3), followed by Sect. 4 which describes the architecture, effectiveness, and current and future status of DIALYZER. Section 5 reviews related work and finally this paper finishes in Sect. 6 with some concluding remarks.

2 The Context of our Work

The Erlang language and Erlang/OTP. ERLANG [1] is a strict, dynamically typed functional programming language with support for concurrency, communication, dis-

[1] DIALYZER: DIscrepancy AnaLYZer of ERlang programs. (From the Greek $\delta\iota\alpha\lambda\acute{\upsilon}\omega$: to dissolve, to break up something into its component parts.) System is freely available from `www.it.uu.se/research/group/hipe/dialyzer/`.

tribution and fault-tolerance. The language relies on automatic memory management. ERLANG's primary design goal was to ease the programming of soft real-time control systems commonly developed by the telecommunications (telecom) industry.

ERLANG's basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for objects (*records* in the ERLANG lingo) is supported, but the underlying implementation of records is the same as tuples. To allow efficient implementation of telecommunication protocols, ERLANG nowadays also includes a *binary* data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. There are no destructive assignments of variables or mutable data structures. Functions are defined as ordered sets of guarded clauses, and clause selection is done by pattern matching. In ERLANG, clause guards either succeed or silently fail, even if these guards are calls to builtins which would otherwise raise an exception if used in a non-guard context. Although there is a good reason for this behavior, this is a language "feature" which often makes clauses unreachable in a way that goes unnoticed by the programmer. ERLANG also provides a `catch/throw`-style exception mechanism, which is often used to protect applications from possible runtime exceptions. Alternatively, concurrent programs can employ so called *supervisors* which are processes that monitor other processes and are responsible for taking some appropriate clean-up action after a software failure.

Erlang/OTP is the standard implementation of the language. It combines ERLANG with the Open Telecom Platform (OTP) middleware. The resulting product, Erlang/OTP, is a library with standard components for telecommunications applications (an ASN.1 compiler, the Mnesia distributed database, servers, state machines, process monitors, tools for load balancing, etc.), standard interfaces such as CORBA and XML, and a variety of communication protocols (e.g., HTTP, FTP, SMTP, etc.).[2]

Erlang applications and real-world uses. The number of areas where ERLANG is actively used is increasing. However, its primary application area is still in large-scale embedded control systems developed by the telecom industry. The Erlang/OTP system has so far been used quite successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks, etc.) to develop software for large (several hundred thousand lines of code) commercial applications. These telecom products range from high-availability ATM servers, ADSL delivery systems, next-generation call centers, Internet servers, and other such networking equipment. Their software has often been developed by large programming teams and is nowadays deployed in systems which are currently in operation. Since these systems are expected to be robust and of high availability, a significant part of the development effort has been spent in their (automated) testing. On the other hand, more often than not, teams which are currently responsible for a particular product do not consist of the original program developers. This and the fact that the code size is large often make bug-hunting and software maintenance quite costly endeavors. Tools that aid this process are of course welcome.

Our involvement in Erlang and history of this work. We are members of the HiPE (High Performance Erlang) group and over the last years have been developing the

[2] Additional information about ERLANG and Erlang/OTP can be found at `www.erlang.org`.

HiPE native code compiler [10, 16]. The compiler is fully integrated in the open source Erlang/OTP system, and translates, in either a just-in-time (JIT) or ahead-of-time fashion, BEAM virtual machine bytecode to native machine code (currently UltraSPARC, x86, and AMD64). The system also extends the Erlang/OTP runtime system to support mixing interpreted and native code execution, at the granularity of individual functions.

One of the means for generating fast native code for a dynamically typed language is to statically eliminate as much as possible the (often unnecessary) overhead that type tests impose on runtime execution. During the last year or so, we have been experimenting with type inference and an aggressive type propagator, mainly for compiler optimization purposes. In our engagement on this task, we noticed that every now and then the compiler choked on pieces of ERLANG code that were obviously bogus (but for which the rather naïve bytecode compiler happily generated code). Since in the context of a JIT it does not really make much sense to stop compilation and complain to the user, and since it is a requirement of HiPE to preserve the observable behavior of the bytecode compiler, we decided to create a separate tool, the DIALYZER, that would statically analyze ERLANG (byte)code and report defects to its users. We report on the methods we use and the implementation of the tool below. However, we stress that the DIALYZER is not just a type checker or an aggressive type propagator.

3 Detecting Discrepancies through Lightweight Static Analysis

3.1 Desiderata

Before we describe the techniques used in DIALYZER, we enumerate the goals and requirements we set for its implementation before we embarked on it:

1. The methods used in DIALYZER should be *sound*: they should aim to maximize the number of reported discrepancies, but should not generate any false positives.
2. The tool should request minimal, preferably no, effort or guidance from its user. In particular, the user should not be *required* to do changes to existing code like providing type information, specifying pre- or post-conditions in functions, or having to write other such annotations. Instead the tool should be completely automated and able to analyze legacy ERLANG code that (quite often) no current developer is familiar with or willing to become so. On the other hand, if the user *chooses* to provide more information, the tool should be able to take it into consideration and improve the precision of the results of its analysis.
3. The tool should be able to do something reasonable even in cases where source code is not available, as e.g., could be the case in telecom switches under operation.
4. The analysis should be *fast* so that DIALYZER has a chance to become an integrated component of ERLANG development.

All these requirements were pragmatically motivated. The applications we had in mind as possible initial users of our tool are large-scale software systems which typically have been developed over a long period and have been tested extensively. This often creates the illusion that they are (almost) bug-free. If the tool reported to their maintainers 1,000 possible discrepancies the first time they use it, of which most are false alarms, quite possibly it would not be taken seriously and its use would be considered a waste of time and effort.[3] In short, what we were after for DIALYZER version 1.0 was to create a lightweight static analysis tool capable of locating discrepancies that are errors: i.e., software defects that are easy to inspect and are easily fixed by an appropriate correcting action.[4] We could relax these requirements only once the tool gained the developers' approval; more on this in Sect. 4.4.

Note that the 2nd requirement is quite strong. It should really be obvious, but it also implies that there are no changes to the underlying philosophy of the language: ERLANG is dynamically typed and there is nothing in our method that changes that.[5]

3.2 Local Analysis

To satisfy the requirement that the analysis is fast, the core of the method is an *intra-procedural, forward* dataflow analysis to determine the set of possible values of live variables at each program point using a *disjoint union of prime types*. The underlying type system itself is based on an extension of the Hindley-Milner static type discipline that incorporates recursive types and accommodates a limited form of union types without compromising its practical efficiency. In this respect, our type system is similar to that proposed by Wright and Cartwright for Soft Scheme [18].

The internal language of the analysis to which bytecode is translated, called Icode, is an idealized ERLANG assembly language with unlimited number of temporaries and an implicit stack. To allow for efficient dataflow analyses and to speed up the fixpoint computation which is required when loops are present, Icode is represented as a control-flow graph (CFG) which has been converted into static single assignment (SSA) form [3]. In Icode, most computations are expressed as function calls and all temporaries survive these. The function calls are divided into calls to primitive operations (primops), built-in functions (bifs), and user-defined functions. Furthermore, there are assignments and control flow operations, including switches, type tests, and comparisons. The remainder of this section describes the local analysis; in Sect. 3.3 we extend it by describing the handling of user-defined functions and by making it inter-modular.

Although ERLANG is a dynamically typed language, type information is present both explicitly and implicitly at the level of Icode. The explicit such information is in the form of type tests which can be translations of explicit type guards in the ERLANG source code, or tests which have been introduced by the compiler to guard unsafe primitive operations. The implicit type information is hidden in calls to primops such as in e.g. addition, which demands that both its operands are numbers. Note that non-trivial

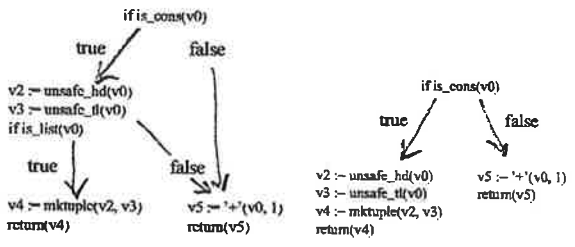
[3] This was not just a hunch; we had observed this attitude in the past. Apparently, we are not the only ones with such experiences and this attitude is not ERLANG-specific; see e.g. [5, Sect. 6].

[4] Despite the conservatism of the approach, we occasionally had hard time convincing developers that some of the discrepancies identified by the tool were indeed code that needed some correcting action. One reaction we got was essentially of the form: "*My program cannot have bugs. It has been used like that for years!*". Fortunately, the vast majority of our users were more open-minded.

[5] This sentence should *not* be interpreted as a religious statement showing our conviction on issues of programming language design; instead it simply re-enforces that we chose to follow a very pragmatic, down-to-earth approach.

```
test(X) ->
  case X of
    [H|T] when is_list(X) ->
      {H, T};
    _ ->
      X + 1
  end.
```

(a) ERLANG code



```
if is_cons(v0)

v2 := unsafe_hd(v0)
v3 := unsafe_tl(v0)
if is_list(v0)

v4 := mktuple(v2, v3)
return(v4)

v5 := '+'(v0, 1)
return(v5)
```

(b) Icode w/o optimization

```
if is_cons(v0)

v2 := unsafe_hd(v0)
v3 := unsafe_tl(v0)
v4 := mktuple(v2, v3)
return(v4)

v5 := '+'(v0, 1)
return(v5)
```

(c) Icode w optimization

Fig. 1. ERLANG code with a redundant type guard.

types for arguments and return values for all primops and bifs can be known *a priori* by the analyzer. These types can be propagated forward in the CFG to jump-start the discrepancy analysis. For example, if a call to addition succeeds, we know for sure that the return value must be a number. We also know that, from that point forward in the CFG the arguments must be numbers as well, or else the operation would have failed. Similarly, if an addition is reached and one of its arguments has a type which the analysis has already determined is not a number, then this is a program point where a discrepancy occurs.

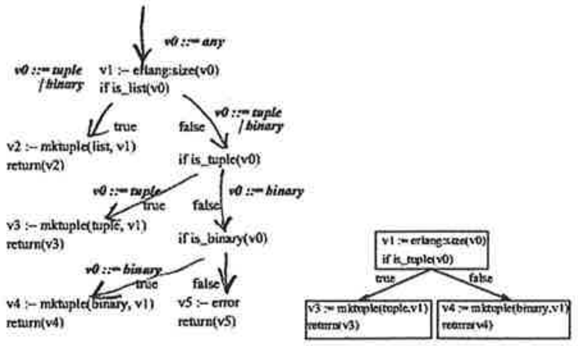
More specifically, the places where the analysis changes its knowledge about the types of variables are:

1. At the *definition point* of each variable.[6] At such a point, the assigned type depends on the operation on the right-hand side. If the return type of the operation is unknown, or if the operation statically can be determined to fail, the variable gets assigned the type *any* (the lattice's top) or *undefined* (its bottom), respectively.
2. At *splits* in the CFG, such as in nodes containing type tests and comparisons. The type propagated in the success branch is the *infimum* (the greatest lower bound in the lattice) of the incoming type and the type tested for. In the fail branch, the success type is subtracted from the incoming set of types.
3. At a point where a variable is used as an *argument* in a call to a primop or a bif with a known signature. The propagated type is the *infimum* of the incoming type and the demanded argument type for the call. If the call is used in a guard context, then this is a split in the CFG and the handling will be as in case 2 above.

When paths join in the CFG, the type information from all incoming edges is unioned, making the analysis *path-insensitive*. Moreover, when a path out of a basic block cannot be taken, the dead path is removed to simplify the control flow. In Fig. 1 the `is_list/1` guard in the first clause of the `case` statement can be removed since the pattern matching compiler has already determined that `X` is bound to a (possibly non-proper) list. This removal identifies a possible discrepancy in the code.

[6] Note that since Icode is on SSA form there can be only one definition point for each variable.

```
test(X) ->
  case size(X) of
    N when is_list(X) ->
      {list, N};
    N when is_tuple(X) ->
      {tuple, N};
    N when is_binary(X) ->
      {binary, N};
    _ ->
      error
  end.
```



(a) ERLANG code (b) Type-annotated Icode (c) Icode w optimization

Fig. 2. An ERLANG program with two discrepancies due to a misuse of the bif `size/1`.

The analysis, through such local type propagation aided by liveness analysis and by applying aggressive global sparse conditional constant propagation and dead code elimination [13], tries to reason about the intent of the programmer. If the most likely path out of a node with a type test is removed, or if a guard always fails, this is reported to the user as a discrepancy. Other discrepancies that are identified by local static analysis include function calls that always fail, pattern matching operations that will raise a runtime exception, and dead clauses in switches (perhaps due to an earlier more general clause). For example, on the program of Fig. 2(a), given that the signature of the `erlang:size/1` built-in function is

```
size(tuple | binary) -> integer
```

the analysis first annotates the Icode control-flow graph with type information. This can be seen in Fig. 2(b) which shows the result of propagating types for variable `v0` only. Given such a type-annotated CFG, it is quite easy to discover and report to the user that both the first `case` clause and the catch-all clause are dead, thereby removing these clauses; see Fig. 2(c). In our example, finding code which is redundant, especially the first clause, reveals a subtle programming error as the corresponding 'measuring' function for lists in ERLANG is `length/1`, not `size/1`.

3.3 Making the Analysis Intra- and Inter-Modular

Currently, the only way to provide the compiler with information about the arguments to a function is by using non-variable terms and guards in clause heads. (This information is primarily used by pattern matching to choose between the function clauses.) Since DIALYZER employs a forward analysis, when analyzing only one function, there can be no information at the function's entry point, but at the end of the analysis there is information about the type of the function's return value. By unioning all proper (i.e., non-exception) type values at the exit points of a function, we get additional type information that can then be used at the function's call sites. The information is often non-trivial since most functions are designed to return values of a certain type and do not explicitly fail (i.e., raise an exception). To take advantage of this, the local analysis is extended with a *persistent lookup table*, mapping function names to information about their return values. The table is used both for intra-modular calls and for calls across module boundaries, but since the table only contains information about functions which have already been analyzed, some kind of iterative analysis is needed.

First consider intra-modular calls. One approach is to iteratively analyze all functions in a module until the information about their return values remains unchanged (i.e., until a fixpoint is reached). The possible problem with this approach is that the fixpoint computation can be quite expensive. Another approach, which is currently the default, is to construct a static call graph of the functions in a module, and then perform one iteration of the analysis by considering the strongly connected components of this graph in a bottom-up fashion (i.e., based on a reversed topological sort). If all components consist of only one function, this will find the same information as an iterative analysis. If there are components which consist of mutually recursive functions, we can either employ fixpoint computation or heuristically compute a safe approximation of the return value types in one pass (for example, the type *any*). Note that this heuristic is acceptable in our context; the discrepancy analysis remains sound but is not complete (i.e., it is not guaranteed to find all discrepancies).

Now consider function calls across module boundaries. In principle, the call graph describing the dependencies between modules can be constructed *a priori*, but this imposes an I/O-bound start-up overhead which we would rather avoid. Instead, we construct this graph as the modules are analyzed for the first time, and use this information only if the user requests a complete analysis which requires a fixpoint computation.

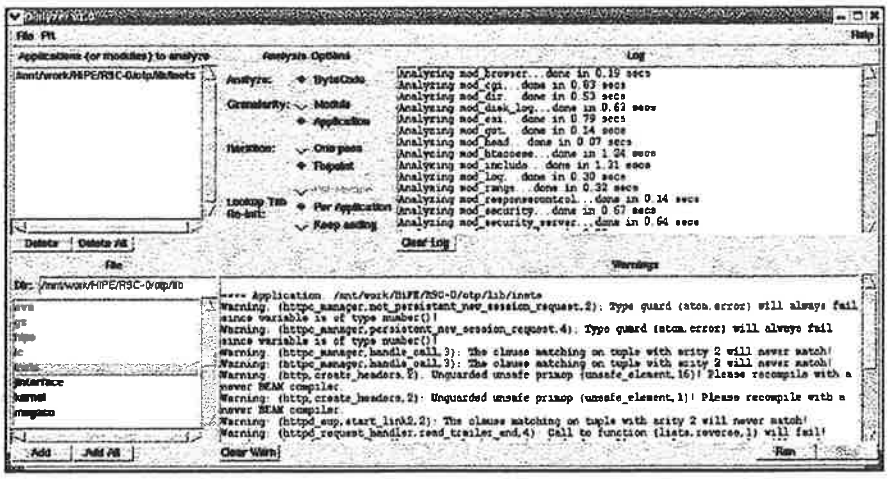
A final note: So far, the analysis has been applied to code of projects which are quite mature. However, as mentioned, our intention is that the tool becomes an integrated component of the program development cycle. In such situations, the code of a module changes often, so the information in the lookup table may become obsolete. When a project is in a phase of rapid prototyping, it might be convenient to get reports of discrepancies discovered based on the code of a single module. The solution to this is to analyze one module till fixpoint, using a lookup table that contains function information from only the start of the analysis of that module. The tool supports such a mode.

4 DIALYZER

4.1 Architecture and Implementation

Figure 3 shows the DIALYZER in action, analyzing the application **inets** from the standard library of the Erlang/OTP R9C-0 distribution.

In DIALYZER v 1.0, the user can choose between different modes of operation. The *granularity* option controls whether the analysis is performed on a single module or on all modules of an application. The *iteration* option selects between performing the analysis till fixpoint or doing a quick-and-dirty, one-pass analysis. The meaning of this



Analyze: [✓ ByteCode]
Granularity: Module [✓ Application]
Iteration: One pass [✓ Fixpoint]
Lookup Tab Re-init: [✓ Per Application] Keep adding

```
Analyzing mod_browser...done in 0.19 secs
Analyzing mod_cgi...done in 0.83 secs
Analyzing mod_dir...done in 0.53 secs
Analyzing mod_disk_log...done in 0.62 secs
Analyzing mod_esi...done in 0.79 secs
Analyzing mod_get...done in 0.14 secs
Analyzing mod_head...done in 0.07 secs
Analyzing mod_htaccess...done in 1.24 secs
Analyzing mod_include...done in 1.31 secs
Analyzing mod_log...done in 0.30 secs
Analyzing mod_range...done in 0.32 secs
Analyzing mod_responsecontrol...done in 0.14 secs
Analyzing mod_security...done in 0.67 secs
Analyzing mod_security_server...done in 0.64 secs
```

```
==== Application /mnt/work/HiPE/R9C-0/otp/lib/inets
Warning: (httpc_manager.not_persistent_new_session_request.2): Type guard (atom.error) will always fail
since variable is of type number()!
Warning: (httpc_manager.persistent_new_session_request.4): Type guard (atom.error) will always fail
since variable is of type number()!
Warning: (httpc_manager.handle_call.3): The clause matching on tuple with arity 2 will never match!
Warning: (httpc_manager.handle_call.3): The clause matching on tuple with arity 2 will never match!
Warning: (http.create_headers.2): Unguarded unsafe primop (unsafe_element.16)! Please recompile with a
newer BEAM compiler.
Warning: (http.create_headers.2): Unguarded unsafe primop (unsafe_element.1)! Please recompile with a
newer BEAM compiler.
Warning: (httpd_esp.start_link2.2): The clause matching on tuple with arity 2 will never match!
Warning: (httpd_request_handler.read_trailer_end.4): Call to function (lists.reverse.1) will fail!
```

Fig. 3. The DIALYZER in action.

option partly depends on the selected granularity. For example, if the granularity is per application and the one-pass analysis is selected, each module is only analyzed once, but fixpoint iteration is still applied inside the module. Finally, the *lookup table re-init* option specifies when the persistent lookup table is to be re-initialized, i.e., if the information is allowed to leak between the analysis elements specified by the granularity. Combinations of options whose semantics is unclear are automatically disabled.

While the analysis is running, a log displays its progress, and the discrepancies which are found are reported by descriptive warnings in a separate window area; see Fig. 3. When the analysis is finished, the log and the warnings can be saved to files. As described in Sect. 3.3, the module-dependency graph is calculated during the first iteration of the analysis of an entire application. If a fixpoint analysis on the application level is requested, DIALYZER uses this information to determine the order in which the modules are analyzed in the next iteration to reduce the number of iterations needed to reach a fixpoint. In fact, even in the one-pass mode the module-dependency graph is constructed, just in case the user decides to request a fixpoint analysis on completion. Requesting this is typically not burdensome as the analysis is quite fast; this can also be seen in the figure. On a 2GHz laptop running Linux, the DIALYZER analyzes roughly 800 lines of ERLANG code per second, including I/O. (For example, the sizes of **mod_cgi**, **mod_disk_log**, and **mod_htaccess** modules are 792, 405, and 1137 lines, respectively. As another example, one run of the analysis for the complete Erlang/OTP standard library, comprising of about 600,000 lines of code, takes around 13 minutes.)

The DIALYZER distribution includes the code for the graphical user interface and the analyzer, both written in ERLANG. As its analyzer depends on having access to a specific version of the HiPE native code compiler, on whose infrastructure (the BEAM bytecode disassembler, the translator from BEAM to Icode, and the Icode supporting code such as SSA conversion and liveness analysis) it relies, the presence of a recent

Erlang/OTP release is also required. Upon first start-up, DIALYZER will automatically trigger the fixpoint-based analysis of the Erlang/OTP standard library, **stdlib**, to construct a persistent lookup table which can be used as a basis for all subsequent analyses.

4.2 The DIALYZER in Anger

In order to show that our analysis is indeed effective in identifying software defects, we present some results obtained from using the DIALYZER to analyze code from large-scale telecom applications written in ERLANG. These applications all have in common that they are heavily used and well-tested commercial products, but as we will see, DIALYZER still exposed problems that had gone unnoticed by testing. Some brief additional information about these applications appears below:

- AXD301 is an asynchronous transfer mode (ATM) switching system from Ericsson [2]. The project has been running for more than eight years now and its team currently involves 200 people (but this number also includes some support staff; not only developers or testers). The ATM switch is designed for non-stop operation, so robustness and high availability are very important and taken seriously during development. As a consequence, a significant effort (and part of the project's budget) has been spent on testing its safety-critical components; see also [17].
- GPRS (General Packet Radio Service) is a telecom system from Ericsson. A large percentage of its code base is written in ERLANG. The project has been running for more than seven years now and its testing includes extensive test suites, automated daily builds, and code coverage analysis. Since this was a pilot-study for the applicability and effectiveness of DIALYZER in identifying discrepancies, only part of GPRS's ERLANG code has so far been analyzed. Although only part of the total code base, the analyzed code is rather big: it consists of 580,000 lines of ERLANG code, excluding comments.
- Melody is a control system for a "Caller Tunes" ringbacktone service developed by T-Mobile. It is an implementation of a customer database with interfaces to media players, short message service centers, payment platforms, and provisioning systems. The core of Melody is significantly smaller than the other telecom products which were analyzed; however, it includes parts of T-Mobile's extensively used and well-tested standard telecom library.

In addition to these commercial applications of ERLANG, we also analyzed the complete set of standard libraries from Erlang/OTP release R9C-0 from Ericsson and code from Jungerl,[7] which is an open-source code repository for ERLANG developers.

In order to have a more refined view of the kinds of discrepancies DIALYZER found, we can manually divide them into the following categories:

Explosives These are places in the code that would raise a run-time exception. Examples of this are calls to ERLANG built-in functions with the wrong type of arguments, operators not defined on certain operands, faulty (byte) code, etc. An explosive can of course be conditional (e.g., firing on some execution paths only, rather than in all paths).

[7] A Junglc of ERLANG codc; scc `sourceforge.net/projects/jungerl/`.

Camouflages These are programming errors that for example make clauses or branches in the control flow graph unreachable — although the programmer did not intend them as such — without causing the program to stop or a supervisor process being notified that something is wrong. The most common error of this kind is a guard that will always silently fail.

Cemeteries These are places of dead code. Such code is of course harmless, but code that can never be executed often reveals subtle programming errors. A common kind of cemeteries are clauses in `case` statements which can never match (because of previous code) and are thus redundant.

For example, in the code of Fig. 2(a) if the analysis encounters, in this or in some other module, a call of the form `test([_|_])`, this is classified as an explosive since it will generate a runtime exception. In the same figure, both the first and the last clause of the `case` statement are cemeteries, as they contain dead code. On the other hand, the code fragment below shows an example of a camouflage: the silent failure of the `size(X)` call in a guard context will prevent this clause from ever returning, although arguably the programmer's intention was to handle big lists.

```
test(X) when is_list(X), size(X) > 10 ->
  {list, big_size};
...  %% Other clauses
```

Table 1 shows the number of discrepancies found in the different projects.[8] The numbers in the column titled "lines of code" show an indication of the size of each project (comments and blank lines have been excluded) and justify our reasoning why requiring type information or any other user annotations *a posteriori* in the development cycle is not an option in our context. Although we would actually strongly prefer to have any sort of information that would make the analysis more effective, we are fully convinced that it would be an enormous task for developers to go through all this code and provide type information — especially since this would entail intimate knowledge about code that might have been written by someone else years ago. Realistically, the probability of this happening simply in order to start using DIALYZER in some commercial project, is most certainly zero.

Despite these constraints, DIALYZER is quite effective in identifying software defects in the analyzed projects; see Table 1. Indeed, we were positively surprised by the amount of discrepancies DIALYZER managed to identify, given the amount of testing effort already spent on the safety-critical components of these projects and the conservatism of the methods which DIALYZER version 1.0 currently employs.

In addition to finding programming errors in ERLANG code, DIALYZER can also expose software errors which were caused by a rather flawed translation of record expressions by the BEAM bytecode compiler. In Table 1, 31 of the reported explosives for Erlang/OTP R9C-0 and 7 for Melody (indicated in parentheses) are caused by the BEAM compiler generating unsafe instructions that fail to be guarded by an appropriate type test. This in turn could result in buffer overruns or segmentation faults if the

[8] Actually, DIALYZER also warns its user about the use of some archaic ERLANG idioms and code relics; these warnings are not considered discrepancies and are not reported in Table 1.

Table 1. Number of discrepancies of different kinds found in the analyzed projects.

Project	Lines of code (total)	Discrepancies (total)	Classification Explosives	Camouflages	Cemeteries
OTP R9C-0	600, 000	57	38 (31)	5	14
AXD301	1, 100, 000	132	26	2	104
GPRS	580, 000	44	10	2	32
Jungerl	80, 000	12	5	2	5
Melody	25, 000	9	8 (7)	1	0

instructions' arguments were not of the (implicitly) expected type. This compiler bug has been corrected in release R9C-1 of Erlang/OTP.

4.3 Current Features and Limitations

The tool confuses programming errors with errors in the BEAM bytecode. Typically this is not a problem as DIALYZER has built-in knowledge about common discrepancies caused by flawed BEAM code. When such a discrepancy is encountered, DIALYZER recommends its user to re-generate the bytecode file using a newer BEAM compiler and re-run the analysis. As a matter of fact, we see this ability to identify faulty BEAM code as an advantage rather than as a limitation.

Starting from bytecode unfortunately means that warning messages cannot be descriptive enough: in particular they do not precisely identify the clause/line where the discrepancy occurs; see also Fig. 3. This can often be confusing. Also, since soundness currently is a major concern, the DIALYZER only reports warnings when it is clear that these are discrepancies. For example, if a switch contains a clause with a pattern that cannot possibly match then this is reported since it is a clear discrepancy. On the other hand, if the analysis finds that the patterns in the cases of the switch fail to cover all possible type values of the incoming term, this is not reported since it might be due to over-approximation caused by the path-insensitivity of the analysis. Of course, we could easily relax this and let the programmer decide, but as explained in Sect. 3.1 soundness is a requirement which DIALYZER religiously follows at this point.

4.4 Planned Future Extensions

One of the strengths of DIALYZER version 1.0 is that no alterations to the source code are needed. In fact, as we have pointed out, the tool does not even need access to it. However, if the source code is indeed available, it can provide the analysis with additional information. Work is in progress to generate Icode directly from CORE ERLANG, which is the official core language for ERLANG and the language used internally in the BEAM compiler. Since CORE ERLANG is on a level which is closer to the original source, where it is easier to reason about the programmer's intentions, it can provide DIALYZER with means to produce better warning messages; in particular line number information can be retained at this level. The structure of CORE ERLANG can also help in deriving, in a more precise way, information about the possible values used as arguments to functions that are local to a module.

potential errors, and the NUDE (the NU-Prolog Debugging Environment [14]) and Ciao Prolog [9] systems which also incorporate type-annotation-guided static debuggers.

In the context of ERLANG, two type systems have been developed before: one based on subtyping [11] and a recent one based on soft types [15]. To the best of our knowledge, the latter has not yet been used by anybody other than its author, although time might of course change this. The former ([11]) allows for declaration-free recursive types using subtyping constraints, and algorithms for type inference and checking are also given in the same paper. It is fair to say that the approach has thus far not been very successful in the ERLANG community. Reasons for this include the fact that the type system constraints the language by rejecting code that does not explicitly handle cases for failures, that its inference algorithm fails to infer types of functions depending on certain pattern matching constructs, and that it demands a non-trivial amount of user intervention (in the form of type annotations in the source code). Stated differently, what [11] tries to do is to impose a style of programming in ERLANG which is closer to that followed in statically typed languages, in order to get the benefits of static type-error detection. Clearly this goal is ambitious and perhaps worthwhile to pursue, but then again its impact on projects which already consist of over a million lines of code is uncertain. Our work on the other hand is less ambitious and more pragmatically oriented. We simply aim to locate (some of the) software defects in already developed ERLANG code, *without imposing a new method for writing programs, but by trying to encourage an implicit philosophy for software development* (namely, the frequent use of a static checker tool rather than just relying on testing) which arguably is better than the practice the (vast majority of the) ERLANG community currently follows.

6 Concluding Remarks

DIALYZER version 1.0 represents a first attempt to create a tool that uses lightweight static analysis to detect software defects in large telecom applications and other programs developed using ERLANG. While we believe that our experiment has been largely successful, there are several aspects of the tool that could be improved through either better technology or by relaxing its requirements (e.g., no false warnings), which are currently quite stringent. Given support, we intend to work in these directions.

On a more philosophical level, it is admittedly the case that most of the software defects identified by DIALYZER are not very deep. Moreover, this seems to be an inherent limitation of the method. For example, problems such as deadlock freedom of ERLANG programs cannot be checked by DIALYZER. One cannot help being a bit skeptical about the *real* power of static analysis or type systems in general, and wonder whether a tool that used techniques from software model checking would, at least in principle, be able to check for a richer set of properties and give stronger correctness guarantees. On the other hand, there is enough evidence that neither static analysis nor software model checking are currently at the stage where one dominates the other; see also [5].

More importantly, one should *not* underestimate the power of simplicity and ease of use of a (software) tool. In a relatively short time and with very little effort, DIALYZER managed to identify a large number of software defects that had gone unnoticed after years of testing. Moreover, it managed to identify bugs that are relatively easy to correct

We also plan to extend DIALYZER with the possibility that its user incrementally adds optional type annotations to the source code. The way to do this is not yet decided, but the primary goal of these annotations, besides adding valuable source code documentation, is to aid the analysis in its hunt for discrepancies, not to make ERLANG a statically typed language. If a type signature is provided for a function, and this signature can be verified by DIALYZER as described below, it can be used by the analysis in the same way as calls to bifs and primops are used in the current version. The way to verify a signature is as follows: instead of trying to infer the types at each call site (as would be the case in most type systems), the signature would be trusted until the function is analyzed. At this point the signature would be compared to the result of the analysis and checked for possible violations. Since DIALYZER is not a compiler, no programs would be rejected, but if violations of user-defined signatures are discovered, this would be reported to the user together with a message saying that the results of the discrepancy analysis could not be trusted.

Taking this idea further, we also plan to experiment with relaxing soundness by allowing the user to specify annotations that in general cannot be statically verified (for example, that a certain argument is a non-negative integer). This is similar to the direction that research for identifying defects such as buffer overruns and memory leaks in C (see e.g. [6, 4]) or for detecting violations of specifications in Java programs [8] has recently taken.

5 Related Work

Clearly, we are not the first to notice that compiler and static analysis technology can be employed for identifying defects in large software projects.[9] Especially during the few last years, researchers in the programming language community have shown significant interest in this subject; see e.g. the work mentioned in the last paragraph of the previous section and the references therein. Most of that work has focused on detecting errors such as buffer overruns, memory access errors such as accessing memory which has already been freed or following invalid pointer references in C, race detection in multi-threaded Java programs, etc. These software defects are simply not present in our context, at least not directly so.[10] Similarly to what we do, some of these analyses do not need source code to be present, since they start from the binary code of the executable. On the other hand, we are not aware of any work that tries to detect flaws at the level of virtual machine bytecode caused by its flawed generation.

During the late 80's and the beginning of the 90's, the subject of automatic type inference without type declarations received a lot of attention; see e.g. [12] for an early work on the subject. A number of soft type systems have been developed, most of them for the functional languages Lisp and Scheme, and some for Prolog. The one closest to our work is that of Soft Scheme [18]. Perhaps sadly, only a few of them made it into actual distributions of compilers or integrated development environments for these languages. Some notable exceptions are DrScheme [7], a programming environment for Scheme which uses a form of set-based analysis to perform type inference and to mark

[9] We are also willing to bet our fortunes that we will not be the last ones to do so either!
[10] They can only occur in the VM interpreter which is written in C, not in ERLANG code.

— in fact some of them have been already — which brings software in a state closer to the desired goal of total correctness. One fine day, some projects might actually win their war!

Acknowledgments

This research has been supported in part by VINNOVA through the ASTEC (Advanced Software Technology) competence center as part of a project in cooperation with Ericsson and T-Mobile. The research of the second author was partly supported by a grant by Vetenskapsrådet (the Swedish Research Council). We thank Ulf Wiger and Hans Nilsson from the AXD301 team at Ericsson, Kenneth Lundin from the Erlang/OTP team, and Sean Hinde from T-Mobile for their help in analyzing the code from commercial applications and for their kind permission to report those results in this paper.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.
2. S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.
4. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167. ACM Press, June 2003.
5. D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation. Proceedings of the 5th International Conference,* number 2937 in LNCS, pages 191–210. Springer, Jan. 2004.
6. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.
7. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, Mar. 2002.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, June 2002.
9. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program development using abstract interpretation (and the Ciao system preprocessor). In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, number 2694 in LNCS, pages 127–152, Berlin, Germany, June 2003. Springer.
10. E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43, New York, NY, Sept. 2000. ACM Press.
11. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM Press, June 1997.

12. P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21. ACM Press, 1984.
13. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.
14. L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. In A. Porto, editor, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536. The MIT Press, June 1989.
15. S.-O. Nyström. A soft-typing system for Erlang. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 56–71. ACM Press, Aug. 2003.
16. M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 228–244, Berlin, Germany, Sept. 2002. Springer.
17. U. Wiger, G. Ask, and K. Boortz. World-class product certification using Erlang. *SIGPLAN Notices*, 37(12):25–34, Dec. 2002.
18. A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

In the need of a design...
reverse engineering Erlang software

Thomas Arts[1] and Cecilia Holmqvist[2]

[1] IT university in Göteborg
Box 8718, 40275 Gteborg, Sweden,
email thomas.arts@ituniv.se
[2] Ericsson AB
Lindholmspiren 11, 41756 Gteborg, Sweden

1 Introduction

Software development often faces the problem that over time the design documents and the actually implemented code coincide less and less. After fine-tuning the software, adding features, leaving out other features and correcting design errors in the code, not in the documents, the result is a product that can be sold. However, the design documentation is no longer up-to-date.

Time to market is important and the costs that it takes to keep design documentation updated is sometimes thrown into the shade of being out first. Only if one survives, the up-to-date design documents are of any use. Thus, after obtaining a market share, one is forced to update the design documents in order to effectively propose additional features or major software changes.

In this paper we discuss a techinque to help reverse engineering a part of an Ericsson product. In the economic crisis in which the product was finalized for the market, the resources for updating the design documents were not available. After being successful in the market, the product has stabilized. Now, there is a strong wish for updated design documents. First of all, for understanding the system better (what is actually going on?). Second, for being able to plan a new implementation from scratch for large parts of the system. Third, the difference between actual behaviour and designed behaviour indicates problems in software parts: the larger the difference between design and actual code, the larger the possibility that errors can be found in that part.

The product we looked at was written in Erlang [1] with rather strict design guidelines. For example, the names used for functions and modules were well reflecting the original design; in the software block *Mobilty Management in UMTS (MMU)* all module names started with mmu. These strict guidelines helped us in easily reconstructing a design from traces of the code.

We performed a case study to show the possibility of supporting the reverse engineering attempt with some software tools. These tools strongly depended on the fact that Erlang is the implementation language of the system and that the design guidelines are followed. However, projects with other, but also strict guidelines will be able to use the same tools in order to reverse engineer the code.

Other reverse engineering attempts have been presented by Nyström [6] and by Mohagheghi *et. al.* [7]. Both tools are based on a static analysis of the code, whereas in our approach we assume not to have access to the source code, but only use information available at runtime.

The paper is organized as follows: In Sect. 2 we describe the rough outline of the software we looked at. In Sect. 3 we discuss how we obtained a finite state machine from looking at traces of the running software. The state machine was graphically visualized and manually compared with the original UML design. In Sect. 4 we discuss what differences we were quickly able to find with this technique.

2 Mobility Management in UMTS

In our case-study we concentrate on the Mobility Management in UMTS (MMU), which is a software component, a so called *block*, in one of the systems in a UMTS network.

The component is specified by a set of UML diagrams. A subset of these diagrams consist of state machines that specify the states in which the component can be. This subset resembles much that of a hierarchical state machine, a UML version of statecharts [5]. On the first or top level a state machine with eight states, with names like *ms_attaching*, *ms_connected*, *etc*, is specified (see Fig. 1). On the next level, each of these states is represented as a state machine with several sub-states within this larger state.

Strangely enough, the implementation of the hierarchical state machines is not based upon the *generic finite state machine* behaviour, nor on a specially developed hierarchical state machine behaviour. Without digging into the reason for it, we only state here that the events are implemented as function calls (e.g., `detach_request/6` implements a Detach Request event) and state is implicit in the software. For that reason we cannot use the earlier developed trace visualisation software [2]. Since the application needs to have access to the state now and then, the programmers have added a function call `set_state/2` to store the information of the present state. Whether this is stored in a process dictionary or in a server process, is not important for the rest of the story.

Needless to say that this way of keeping track of the present state is rather error prone. The generic finite state machine behaviour, where state is guaranteed to be updated after every event, is more robust against a programmer that forgets to update a state. In particular, a behaviour can be used to already statically detect such omissions, whereas in the present implementation, one has to find these omissions by running tests.

The component consists of several Erlang modules. The main module is called `mmumoc`, which serves as an interface for all other modules in the component. All communication from other components or subsystems to this component goes via the `mmumoc` module, which means that at least one interface function in this module is called for every external signal. The functions in the `mmumoc` module depend on functions in the other modules (of which the names all start with

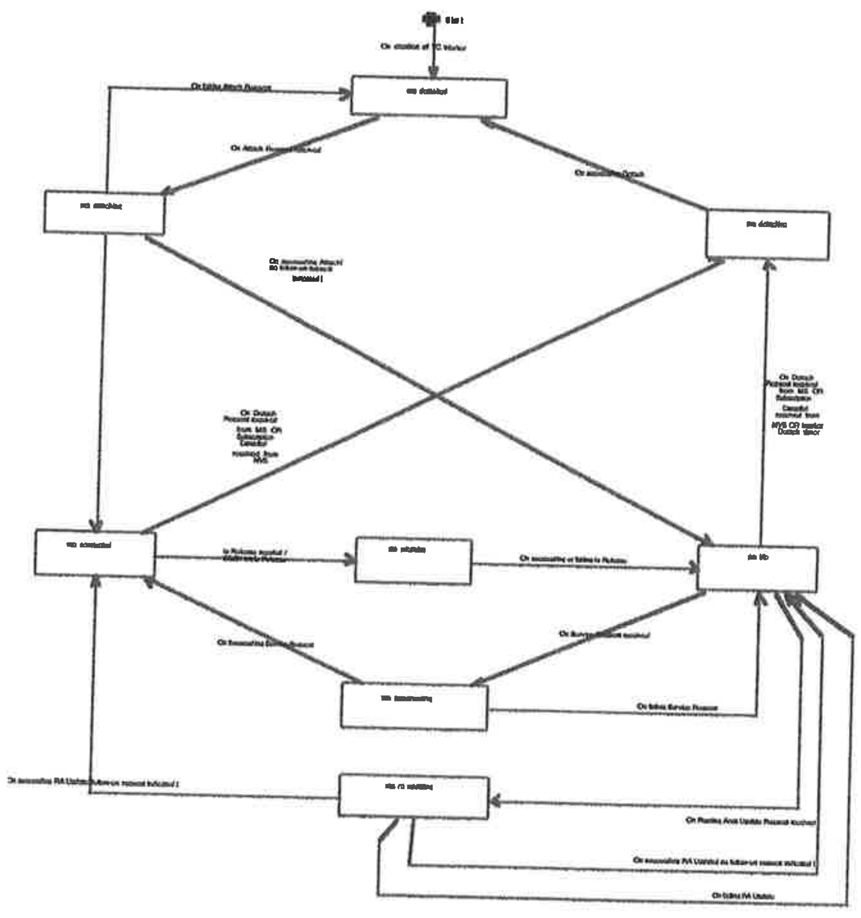

Start
On Attach Request received
On failed Service Request
On Routing Area Update Request received
On failed RA Update

Fig. 1. Top level state machine

mmu). An incoming Detach Request can for example cause over twenty different functions in five 'mmu' modules to be called, many of which are called several times, before the state is updated.

3 Runtime software analysis

The MMU component consists of too many lines of code to simply understand what the software is doing by looking at the source code. Even with the design at hand it is far too costly, if not impossible, to statically analyze the code, i.e., determining the behaviour of the code by carefully studying the source code. Therefore, traditionally, a manual runtime analysis is performed.

Erlang has a trace function that allows a person to select whatever function he/she is interested in. Calls to these functions are then monitored in a running system. Runtime analysis uses this trace function. If one wants to analyse the MMU component, one monitors all calls to functions in this component, i.e., where the module name starts with `mmu`. A set of stimuli is provided to the system and the resulting list, a so called *trace*, of function calls (including time stamp, arguments, return value, etc) is stored in a file created by the `disk_log` module.

Before we started our work, these trace files were converted to a textual format and manually analyzed. Traces can differ in length, but we looked at files of about 9MB, containing about 15 thousand entries. Loaded into emacs, it is rather easy to find each occurence of `mmumoc:set_state/2`. However, we immediately wanted to have a tool for quickly extracting these calls from the trace. Note that one could also choose to only monitor calls to this particular function with the trace functions. However, we are not only interested in the possible state transitions. If we find a state transition that differs from the one in the specification, we also want to understand what has happened. The other functions in the trace are necessary to get that understanding. Since we might miss the particular behaviour when we run the software again, it is important to have enough information in the trace to determine what happened in case of an unexpected state transition.

Our traces can be rather large, in principle even several gigabytes. For performance reasons, we do not want to read this binary trace in memory to convert it in a list and then remove most of the unnecessary elements (only very few function calls are calls to the `set_state` function). By using the `disk_log:chunck` function, we would be so much restricted to 64 bytes, that we decided to implement our own log reader. In an eager language we program a lazy file reading, which we make flexible to the kind of filtering by having a filter function as argument.

```
read(FileName,Filter) ->
    {ok,FileDescr} = file:open(FileName,[read, raw, binary]),
    Terms = unpack(FileDescr,Filter,[]),
    file:close(FileDescr),
    Terms.
```

```
unpack(FileDescr,Filter,Terms) ->
    case file:read(FileDescr,5) of
        {ok,<<B1,B2,B3,Size:16>>} ->
          {ok,BTerm} = file:read(FileDescr,Size),
          Term = binary_to_term(BTerm),
          case Filter(Term) of
              true ->
                 unpack(FileDescr,Filter,[Term|Terms]);
              false ->
                 unpack(FileDescr,Filter,Terms)
          end;
        eof ->
          lists:reverse(Terms)
    end.
```

The result of the unpack function is a list of terms. To every term in the log file, we have applied a filter function, returning either *true* or *false*. Only those terms for which *true* was returned, are left in the list.

The filter function is typically something like, *either the term is a* `mmumoc:set_state/2`*, or a an event originating external to the MMU component*. Thus, it is a logical combination of well determined terms. This can be implemented very flexible by defining small filters for the well determined terms and combinators to combine them. For example, the filters to determine state and events from outside, can be defined as:

```
state_mmu() ->
    fun({trace_ts,Pid,call,{mmumoc,set_state,[S,SS]},Caller,TS}) ->
        true;
       (_) ->
        false
    end.

outside_mmu() ->
    fun({trace_ts,Pid,call,{mmumoc,F,A},{CM,CF,CA},TS}) ->
        string:substr(atom_to_list(CM),1,4)==[$m,$m,$u,$_];
       (_) ->
        false
    end.
```

A beautiful way of writing a filter would be `or(state_mmu(),outside_mmu())`, or a more complex case to obtain all calls that are not inside the MMU component `and(call(),not(inside_mmu()))`. Since the logical operators cannot be overloaded in Erlang, we use a different name for them and define them as follows (cf. [3]):

```
filter_or(F1,F2) ->
```

4 Comparing Design and Reality

With the visualization of several traces at hand, it is easy to compare the UML design and the state machines obtained from a trace. It turns out that during the development, the code has really diverted a lot from the actual design. Most generated substate machine contains some states and some transitions that are not present in the design.

In a few hours, many issues are written down. Sometimes the differences between real behaviour and design is so different, that the number of states in common is less than the number in which they differ. The whole design started from use-cases and some of these use-cases have found their way in the code without the state machines in the design being updated.

e even find that some state machines are not modeled at all, i.e., there is no design available.

All together, we can conclude that we quickly (a few hours) detect and administer a number of major differences. After that, other traces visualize the same machines and we only find some minor differences.

If we would have had access to the encoding of the UML pictures, we would have been able to write a small tool to compare state machines, therewith being even faster in our comparison (visually highlighting the difference). However, even with the present way of comparing design and actual code, we are many times faster than by using a manual comparison of design and a text version of the trace information.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[2] T. Arts and L-Å. Fredlund. Trace Analysis of Erlang Programs. In Proc. of *ACM Sigplan Erlang Workshop*, Pittsburgh, USA, 2002.

[3] T. Arts, K. Claessen, H. Svensson. Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang. In Proc. of *FATES 2004*, Linz, Austria, 2004.

[4] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[5] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8:231-274, 1987.

[6] J. Nyström and Bengt Jonsson. A Tool for Extracting the Process Structure of Erlang Applications. Erlang User Conference, Stockholm 2001. Also at the Erlang Workshop, Florens, Italy, 2001.

[7] P. Mohagheghi, J.P. Nytun, Selo, and W. Najib. MDA and Integration of Legacy Systems: An Industrial Case Study. In Proc. of *Workshop on Model Driven Architecture: Foundations and Applications, MDAFA'03.* University of Twente, Enschede, The Netherlands. 26-27 June 2003.

1 (10)

Erlang's Exception Handling Revisited

Richard Carlsson
Department of Information Technology, Uppsala University, Sweden
richardc@csd.uu.se

Björn Gustavsson
Ericsson, Sweden
bjorn@erix.ericsson.se

Patrik Nyblom
Ericsson, Sweden
pan@erix.ericsson.se

ABSTRACT

This paper describes the new exception handling in the ERLANG programming language, to be introduced in the forthcoming Release 10 of the Erlang/OTP system. We give a comprehensive description of the behaviour of exceptions in modern-day ERLANG, present a theoretical model of the semantics of exceptions, and use this to derive the new `try`-construct.

1. INTRODUCTION

The exception handling in ERLANG is an area of the language that is not completely understood by most programmers. There are several details that are often overlooked, sometimes making the program sensitive to changes, or hiding the reasons for errors so that debugging becomes difficult. The existing `catch` mechanism is inadequate in many respects, since it has not evolved along with the actual behaviour of exceptions in ERLANG implementations. The addition of a new exception handling construct to replace the existing `catch` has long been discussed, but has not yet made its way into the language.

The purpose of this paper is twofold: first, to explain the realities of exceptions in ERLANG, and why the creation of a new exception-handling construct has been such a long and complicated process; second, to describe in detail the syntax and semantics of the finally accepted form of the `try`-construct, which is to be introduced in Erlang/OTP R10.

The layout of the rest of the paper is as follows: Section 2 describes in detail how exceptions in ERLANG actually work, and the shortcomings of the current `catch` operator. Section 3 explains how the new `try`-construct was derived, and why `try` in a functional language has different requirements than in an imperative language such as C++ or Java. In Section 4, we refine the exception model and give the full syntax and semantics of `try`-expressions, along with some concrete examples. Section 5 discusses related work, and Section 6 concludes.

2. EXCEPTION HANDLING IN ERLANG

2.1 Exceptions as we know them

The exception handling in ERLANG, as described in [2], was designed to be simple and straightforward. An exception can occur when a built-in operation fails, such as an arithmetic operation, list operation, pattern matching, or case-clause selection, or when the user calls one of the built-in functions `exit(Term)` or `throw(Term)`.

The exception is then described by an ordinary ERLANG term (often called the *reason*), such as an atom `badarg` or `badarith`, or a tuple like `{badmatch, Value}` where `Value` is the data that could not be matched. To prevent exceptions from propagating out of an expression, the expression can be placed within a `catch` operator, as in

```
X = (catch f(Y))
```

(the parentheses are needed in this case because of the low precedence of `catch`), or more often

```
case catch f(Y) of
  ...
end
```

to immediately switch on the result. If the expression within the `catch` completes normally, the resulting value is returned as if the `catch` had not been there.

When an exception occurs, one of two things can happen: either the exception is not caught by the executing process, and in that case the process terminates, possibly propagating a signal to other processes; otherwise, the execution had entered (but not yet exited) one or more catch-expressions before the exception happened, and execution is resumed at the latest entered `catch`, unrolling the stack as necessary. The result of the `catch` expression then depends on how the exception was caused: if it occurred because of a call to `throw(Term)`, the result is exactly `Term`.[1] Otherwise, the result will have the form of a tuple `{'EXIT', Term}`. For example,

```
catch (1 + foo)
```

returns `{'EXIT', badarith}` (since `foo` is an atom), while

```
catch throw(foo)
```

returns the atom `foo`, the intention being that `throw` should be used for "nonlocal returns" (e.g., for escaping out of a deep recursion) and other user-level exceptions. The `exit` function, on the other hand, is intended for terminating the process, and behaves like a run-time failure, so that

[1] Much like `catch/throw` in Common Lisp.

```
    catch exit(user_error)
```

returns {'EXIT', user_error}.

One of the consequences of this (explicitly described in [2] as a design decision) was that it became possible to "fake" a run-time failure or exit, by calling throw({'EXIT', Term}), or by simply returning the value {'EXIT', Term}. For example, in the following code:

```
    R = catch (case X of
                    1 -> 1 + foo;
                    2 -> exit(badarith);
                    3 -> throw({'EXIT', badarith});
                    4 -> {'EXIT', badarith}
                    5 -> throw(ok);
                    6 -> ok
                 end),
    case R of
      {'EXIT', badarith} -> "1-4";
      ok -> "5-6"
    end
```

the semantics of catch makes it impossible to tell whether the value of R (depending on X) is the result of catching a run-time failure or a call to exit or throw, or if the expression completed execution in a normal way. Usually, this is not a problem; for example, most ERLANG programmers would never use a tuple {'EXIT', Term} in normal code.

2.2 Where's the catch?

In some contexts, it becomes more important to know what has actually happened. For example, consider:

```
  lookup(X, F, Default) ->
    case catch F(X) of
      {'EXIT', Reason} -> handle(Reason);
      not_found -> Default;
      Value -> Value
    end.
```

where F is bound to some function, which should either return a value depending on X, or call throw(not_found).

Note that the possible returned values cannot include the atom not_found. To solve this in general, the return values would need a wrapper, such as {ok, Value}, to separate them from any thrown terms (assuming that {ok, ...} is never thrown, much like it is assumed that {'EXIT', ...} is not normally returned by any function). This limits the usefulness of throw somewhat, since it requires that the normal-case return values are marked, rather than the exceptional values, which is counterintuitive and bothersome.

An idiom used by a few knowledgeable ERLANG programmers to create almost-foolproof catches is the following:

```
  lookup(X, F, Default) ->
    case catch {ok, F(X)} of
      {ok, Value} -> Value;
      {'EXIT', Reason} -> exit(Reason);
      not_found -> Default;
      Term -> throw(Term)
    end.
```

Since it is guaranteed that the creation of a tuple such as {ok, *Expr*} will never cause an exception if the subexpression *Expr* completes normally, we have a way of separating exceptions in F(X) from normal return values – as long as we trust that nobody calls throw({ok, ...}) within F(X). Furthermore, any caught exceptions that are not of interest at this point can simply passed to throw or exit again, hoping that some other catch will handle it.

This way of writing safer catches is however rarely seen in practice, since not many programmers know the trick, or bother enough to use it, since their catches mostly work anyway – at least until some other part of the code changes.

2.3 Current practice

The difficulty in properly separating exceptions from return values appears to be the main reason why although ERLANG has a catch/throw mechanism, it is still the case that in existing code, the predominant way of signalling the success or failure of a function is to make it return tagged tuples like {ok, Value} in case of success and {error, Reason} otherwise, forcing the caller to check the result and either extract the value or handle the error. This often leads to a clumsy programming style, in the many cases where errors are actually rare and it is even rarer that the caller wants to handle them. (If a significant part of all calls to a function tend to fail, the above can still be a good way of structuring the function interface, but typically, failure happens in only a very small fraction of all calls.)

In C programs [5], the code is often interspersed with many checks to see if function calls have returned valid results, even though there is seldom much that the programmer can do if this was not the case, except terminate the program. The lack of an exception mechanism makes the code less readable, more time-consuming to write, and more error prone since forgetting to check a value can be fatal. ERLANG programs suffer similar problems: even if the programmer cannot do anything constructive to handle an error, he must still remember whether a called function returns a naked value or {ok, Value}, and in the latter case must also decide what should happen if instead {error, Reason} is returned. The following idiom is often used:

```
  {ok, Value} = f(X)
```

so that if the call succeeds, the relevant part of the result is bound to Value, and if the call instead returns {error, Reason}, it will cause a badmatch exception. The main drawback is that it points out the wrong cause of the problem, which was a failure within f(X), and not in the pattern matching. Also, the wrapping convention remains a cause of irritation because one is forced to write awkward code like

```
  {ok, Y} = f(X),
  {ok, Z} = g(Y),
  {ok, Value} = h(Z)
```

when it would have sufficed with

```
  Value = h(g(f(X)))
```

if the functions had returned naked values and used exceptions to signal errors.

Sometimes, programmers attempt to handle the error case as follows:

```
  case f(X) of
    {error, Reason} -> exit(Reason);
    {ok, Value} -> ...
  end
```

but often, the error term **Reason** returned by the function is very particular to that function, and is not suitable for passing to **exit**, so that anyone who catches the resulting exception will only be confused since there is no longer any context available for interpreting the term. So even though the programmer simply wishes to pass on the problem to be handled by someone else, it really requires interpreting the error and creating a more comprehensible report. In fact, the **badmatch** solution above is to be preferred, because it will show precisely where the program gave up, rather than pass on a cryptic term with **exit**.

2.4 Processes and signals

Since ERLANG is a concurrent language, every program is executed by a *process* (similar to a thread), and many processes can be running concurrently in an ERLANG runtime system. A signalling system is used for informing processes about when other processes terminate. As for exceptions, an *exit signal* is described by a term, which if the process terminated normally (by returning from its initial function call) is the atom **normal**. If the process terminated because it called **exit(Term)** (and did not catch the exception), the exit term is exactly the value of **Term**; thus, a process can also terminate "normally" by calling **exit(normal)**, e.g. in order to easily exit from within a deep call chain. Similarly, if the process terminated because of a run-time failure that was not caught, the exit term is the same term that would be reported as **{'EXIT', Term}** in a **catch**-expression, as for instance **badarg** or **{badmatch, Value}**.

A special case is when a process terminates because it called **throw(Term)** and did not catch the exception. In this case, the exit term will be changed to **{nocatch, Term}**, to distinguish this case from other kinds of exits.

2.5 The Truth...

To simplify the above discussion (as many readers will doubtless have noticed), we have left out a few details about exceptions as they appear in modern ERLANG implementations. The presentation in the preceding sections follows the description of exceptions in "The ERLANG Book" [2] (Concurrent Programming in ERLANG, Second Ed., 1996).

The most apparent change since then is that when a run-time failure occurs (and is then either caught in a **catch** or causes the process to terminate), the term that describes the error will also include a symbolic representation of the topmost part of the call stack at the point where the error occurred. (This does not happen for calls to **exit** or **throw**.) The general format is **{Reason, Stack}**, where **Reason** is the normal error term as described in the previous sections. For example, calling **f(foo)** where:

```
f(X) -> "1" ++ g(X).
g(X) -> "2" ++ h(X).
h(X) -> X ++ ".".
```

will generate an exception with a descriptor term such as the following:

```
{badarg, [{erlang,'++',[foo,"."]},
          {foo,h,1},
          {foo,g,1},
          {foo,f,1}]}
```

Details in the stack representation may vary depending on implementation, cause of error, and call history.[2] (Also note that because of tail call optimization, many intermediate function calls cannot be reported, since there is by definition no trace left of them.)

Thus, for example the call **f(0)** where

```
f(X) -> catch 1/X.
```

will actually return

```
{'EXIT', {badarith, [{foo,f,1}, ...]}}
```

in a modern system, rather than **{'EXIT', badarith}**. However, the following code:

```
catch exit(Term)
```

will still yield **{'EXIT', Term}**, without any symbolic stack trace, and similarly

```
catch throw(Term)
```

yields just **Term**, as before.

2.6 ...The Whole Truth...

Now, the observant reader may have noticed that although it would appear that an exception is fully determined by the "reason" term only, in fact at least one other component is necessary to completely describe an exception, namely, a flag that signals whether or not it was caused by **throw(Term)**. (This follows from the semantics of process termination and signals; cf. Section 2.4.)

Internally, an exception is then a pair ⟨*term*, *thrown*⟩, where *thrown* is either **true** or **false**, and *term* is the "reason" term. The semantics of **catch** ***Expr*** can now be described as shown in Figure 1. Note that it is the **catch** operator that decides (depending on the *thrown* flag) whether or not to wrap the reason term in **{'EXIT', ...}**.

```
If evaluation of Expr completed normally with result R
then
    the result of catch Expr is R,
else
    the evaluation threw an exception ⟨term, thrown⟩;
    if thrown is true
    then
        the result of catch Expr is term,
    else
        the result of catch Expr is {'EXIT', term}
```

Figure 1: Semantics of catch *Expr*

2.7 ...And Nothing But The Truth

Something, however, is still missing. When **throw(Term)** is not caught, and causes the process to terminate (as described in Section 2.4), the exit term is no longer simply **{nocatch, Term}**, but rather **{{nocatch, Term}, [...]}**, with a symbolic stack trace just as for a run-time failure. This means that the stack trace cannot be added onto the "reason" term *until it is known what will happen to the exception*,

[2]The actual arguments to the last called function are not always included; only the arity of the function. The next-to-last call is often missing because its return address was never stored on the stack and could not be recovered.

```
If evaluation of Expr completed normally with result $R$
then
    the result of catch Expr is $R$,
else
    the evaluation threw exception $\langle term, thrown, trace \rangle$;
    if thrown is true
    then
        the result of catch Expr is term,
    else
        if trace is null
        then
            the result is {'EXIT', term}
        else
            the result is {'EXIT', {term, trace}}
```

Figure 2: Modified semantics of catch *Expr*

```
If evaluation of the initial call completed normally
then
    the exit term is the atom normal
else
    the evaluation threw exception $\langle term, thrown, trace \rangle$;
    if thrown is true
    then
        the exit term is {{nocatch, term}, trace}
    else
        if trace is null
        then
            the exit term is term
        else
            the exit term is {term, trace}
```

Figure 3: Semantics of process termination

since if it is caught in a `catch`, it *must not* include any stack trace.

As a consequence, we have to extend the full description of an exception to a triple $\langle term, thrown, trace \rangle$, where *trace* is either a symbolic stack trace or a special value *null*, so that *trace* is *null* if and only if the exception was caused by a call to `exit`.

The semantics of `catch` must also be modified as shown in Figure 2, so that in the case where the expression has thrown an exception, and *thrown* is `false`, we have a choice depending on the value of *trace*. The exit term for a terminating process is determined in a similar way, shown in Figure 3.

One last, not very well documented, detail is that when a process terminates due to an exception, and the exception was not caused by a call to `exit(Term)`, this event will be reported by the ERLANG runtime system to the *error logger* service. (In other words, as long as a process terminates normally, or through a call to `exit`, it is considered a normal event from the runtime system's point of view.) This shows once again that it is necessary to preserve the information about whether or not the exception was caused by `exit`, until it is known how the exception will be handled.

2.8 Love's Labour's Lost in Space

For the programmer, currently the only means of intercepting and inspecting an exception is the `catch` operator, but as we have seen, this will lose information which cannot be re-created. For example, as described in Section 2.2, the following code attempts to separate exceptions from normal execution, and transparently pass on all exceptions that do not concern it:

```
case catch {ok, ...} of
  {ok, Value} -> ...;
  {'EXIT', Reason} -> exit(Reason);
  not_found -> ...;
  Term -> throw(Term)
end
```

However, when `throw(Term)` is executed in the last clause, it will create a *new* exception $\langle term, thrown, trace \rangle$ having the same values for *term* and *thrown* as the caught exception, but with a different *trace*. This is observable if the exception causes the process to terminate, and since the original stack trace was lost, it will hide the real reason for the exception.

Furthermore, in the `exit(Reason)` case, the *trace* component of the new exception will be set to *null* by `exit`. Now note that if the caught exception had a non-*null* trace component, the `catch` will already have added that trace onto `Reason`, so in a sense, the term has been "finalized": if the new exception is caught in another `catch`, or causes the process to terminate, the term will look exactly the same as if it had never been intercepted by the above code. But there is one problem: since we used `exit` to pass on the exception, it will *not* be reported to the error logger if it causes the process to terminate – even if the original exception was caused by a run-time failure, which ought to be reported.

One built-in function that we have not mentioned so far, because it is not well known, and has mostly been used in some of the standard libraries, is `erlang:fault(Term)`, which is similar to `exit` but instead causes a run-time failure exception, i.e., such that $trace \neq null$ and $thrown =$ `false`. We could then try to improve on the above code by splitting the `{'EXIT', Reason}` case in two:

```
{'EXIT', {Reason, Stack}} when list(Stack) ->
    erlang:fault(Reason);
{'EXIT', Reason} ->
    exit(Reason);
```

which will preserve the error logging functionality, as long as we can trust that the first clause matches all run-time errors, and nothing else. But like in the `throw` case, we now lose the original stack trace when we call `erlang:fault(Reason)`. What if we tried `erlang:fault({Reason, Stack})` instead? Well, if the exception is caught again, it will then get the form `{{Reason, Stack1}, Stack2}`, and so on if the process is repeated. This preserves all the information, but could cause problems for code that expects to match on the `Reason` term and might not recognize the case when it is nested within more than one `{..., Stack}` wrapper.

Thus, with a fair amount of careful coding, we can now catch exceptions, examine them, and pass them on if they are not of interest, but still not without affecting their semantics in most cases – for `throw`, we lose the stack trace, and for run-time failures we modify the "reason" term. The method is also not foolproof – calls to `throw({ok, Value})`, `throw({'EXIT', Reason})`, or `exit({Term, [...]})` will all cause the wrong clause to be selected. Not to mention that the code is difficult to understand for anyone who is not very familiar with the intricacies of exceptions.

There really should be a better way.

```
try
  Expressions
catch
  Exception₁ -> Body₁;
  ...
  Exceptionₙ -> Bodyₙ
end
```

Figure 4: Basic form of try-expression

3. TRY AND TRY AGAIN

At least a few of the problems with **catch** have been widely known by ERLANG programmers, and for several years, there has been an ongoing discussion among both language developers and users about the addition of a new, more flexible and powerful construct for exception handling to the language. The following attempts to be a complete list of requirements for such a construct:

1. It should be possible to strictly separate normal completion of execution from the handling of exceptions.
2. It should be possible to safely distinguish exceptions caused by **throw** from other exceptions.
3. It should be possible to safely distinguish exceptions caused by **exit** from run-time failures.
4. The behaviour of the existing **catch** must not change; nor the behaviour when an exception causes process termination. Existing programs should work exactly as before.
5. It should be possible to use ordinary pattern matching to select which exceptions are to be handled. Exceptions which do not match any of the specified cases should automatically be re-thrown without changing their semantics.
6. It should be straightforward to rewrite all or most uses of **catch** to the new construct, so that there is no incentive for using **catch** in new code.
7. It should be simple to write code that guarantees the execution of "cleanup code" regardless of how the protected section exits.

3.1 A first try

A form of **try...catch...end** construct for ERLANG was first described in the tentative Standard Erlang Language Specification [3] (only available in a draft version) by Barklund and Virding; however, this work was never completed. Their suggested construct (mainly inspired by C++ [10], Java [4], and Standard ML [7]) had the general form shown in Figure 4, where if evaluation of *Expressions* succeeded with result *R*, the result of the **try** would also be *R*; otherwise, the clauses would be matched in top-down order against either **{'THROW', Value}**, for an exception caused by a **throw**, or **{'EXIT', Reason}** for other exceptions. In the terminology of Section 2.6, it would make the *thrown* flag of the exception explicit (which the **catch** operator does not).

This would fulfill requirements 1 and 2 above, and partially requirements 5 and 4; in particular, the distinction between **exit** and run-time failures was not noted in [3]. In fact, it was during an early attempt by the first author of the present paper to implement the **try** construct, that many of the complications described in Section 2 were first uncovered. As it turned out, the *de facto* behaviour of exceptions in the Erlang/OTP implementation was no longer consistent with any existing description of the language. The only result at that time was that the inner workings of the exception handling were partially rewritten in preparation for future extension, but it was apparent that the **try** construct had to be redesigned before it could be added to the language.

3.2 Making a better try

Since then, several variations of the **try** construct have been considered, but all have been found lacking in various respects. The main problem has turned out to be the balance between simplicity and power of expression. For example, most of the time, the programmer who wishes to catch a **throw** will not be interested in viewing the stack trace, and should preferably not be forced to write a complicated pattern like **{'THROW', Term, Stack}** when the only important part is **Term**. (Also, it would be a waste of time to construct a symbolic trace which is immediately discarded by the catcher.) However, the stack trace should be available when needed.

Also, point 6 above was more of a problem than expected. As seen in some of our previous examples, the following is a common way of using **catch** in existing programs:

```
case catch f(X) of
  {'EXIT', Reason} -> handle(Reason);
  Pattern₁ -> Body₁;
  ...
  Patternₙ -> Bodyₙ
end
```

i.e., which uses a single list of patterns for matching both in the case of success and in the case of catching an exception. As we have described, this makes it possible to mistake a returned result for an exception and vice versa. However, it is an extremely convenient idiom, because it is very often the case that regardless if the evaluation succeeds or throws an exception, a switch will be performed on the result in order to decide exactly how to proceed.

With the **try...catch...end** as suggested in [3], the same effect could only be achieved as follows:

```
R = try {ok, f(X)}
  catch
    Exception -> Exception
  end,
case R of
  {ok, Pattern₁}} -> Body₁;
  ...
  {ok, Patternₙ}} -> Bodyₙ;
  {'THROW', Term} -> ...;
  {'EXIT', Reason} -> ...
end
```

using the trick from Section 2.2 to make sure that the result of normal evaluation is always tagged with **ok**; the main difference being that the **try** version cannot be fooled by e.g. calling **throw({ok, Value})**. So, although the above code is safe, it is quite inconvenient to have to write such a complicated expression for what could be so easily expressed using **catch**.

Analyzing the **try** construct in terms of continuations helps us understand what is going on here. (A continuation is simply "that code which will take the result R of the current expression and continue", and can thus be described as a function $c(R)$; for example, when a function call finishes, it conceptually passes the return value to its continuation, i.e., the return address.[3]) First of all, we have a main continuation c_0 which will receive the final result of evaluating the whole `try...catch...end` expression. Now consider the expression between `try...catch`: if its evaluation succeeds with result R, it will use the *same* continuation $c_0(R)$, i.e., R becomes the result of the whole `try`. On the other hand, if the expression throws an exception E, it will use *another* continuation $c_f(E)$, which we call the "fail-continuation".

The code in c_f is that which does the pattern matching on the exception E over the clauses between `catch...end`. If none of the patterns match, the exception will be re-thrown, and we are done. Otherwise, the first matching clause is selected, and its body is evaluated. (If this should throw a new exception, it will not be caught here since it is not within the `try...catch` section.) The resulting value is finally passed to c_0, and becomes the result of the `try`.

Now let's look at what the continuation $c_0(R)$ gets: its input R is either the result of evaluating the `try...catch` section (if that succeeded), or otherwise it is a value returned by one of the `catch...end` clauses. This is useful in typical situations where exceptions are handled by substituting a default value, as in:

```
Value = try lookup(Key, Table)
        catch
          not_found -> 0
        end
```

However, if we want to perform a *different* action in case the `try...catch` part succeeds, than if an exception occurs, we have no choice but to pass to c_0 not only the result, but also an indication of which path was actually taken. (This is what we did in the previous example, using `{ok, ...}` to tag the result upon success.) It is this limitation of the `try...catch...end` that forces us to go from control flow to a data representation and back to control flow again. It should be noted here that the exception handling in Standard ML [7] (and possibly other functional languages with exceptions) suffers from the same limitation.

A much more elegant solution would be if the programmer could specify code to be executed only in the success case, before control is transferred to the main continuation c_0. In addition to c_f, we therefore introduce a success-continuation c_s, so that if evaluation of the `try...catch` section succeeds with result R, the continuation used would be $c_s(R)$, rather than $c_0(R)$.

A practical syntax for expressing both the success case and the exception case in a single `try` is shown in Figure 5,[4] where the code in $c_s(R)$ does pattern matching on R over the clauses between `of...catch`. If a clause matches, its body is evaluated, and the result is passed to c_0, just like for an exception-handling clause. (If no clause should match, it is a run-time error, and will cause a `try_clause` exception.) Each clause may also have a guard, apart from the pattern, just like any `case`-clause; we have left this out for the sake of readability.

```
try Expressions of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

Figure 5: General form of try-expression

Note that the old syntax from Figure 4, which leaves out the `of...` part, can still be allowed, by simply defining it as equivalent to

```
try Expressions of
  X -> X
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

(where `X` is a fresh variable), in effect making $c_s(R) = c_0(R)$, as before.

At this point, the reader might be wondering why the problem could not have been solved as follows, using the original form of `try`:

```
try
  case Expressions of
    Pattern_1 -> Body_1;
    ...
    Pattern_n -> Body_n
  end
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

The difference is that in this case, all the success actions $Body_i$, as well as the pattern matching, *are now within the protected part of the code.* Thus, the `catch...end` section will handle *all* exceptions that may occur – not only those within *Expressions*. If this was what we wanted, we might as well have moved the whole `case...end` to a new function `f` and simply written

```
try f(...)
catch
  ...
end
```

however, often we *do not* want the same exception handling for both the protected part and the success actions.

It is interesting to note that in the imperative languages which pioneered and made popular the `try...catch` paradigm, i.e., mainly Ada [6], C++ [10] and Java [4], there has never been any need for an explicit success-continuation syntax. The reason is simple: in these languages, the programmer can change the flow of control by use of "escapes"

[3]If it helps, just think of continuations as goto-labels.

[4]First suggested by Fredrik Linder, who also pointed out the weakness in the original `try`, in a discussion at the ERLANG Workshop in Firenze, Italy, 2001.

such as **return**, **goto**, **break**, and **continue**; for example, forcing an early return from within an exception-handling branch. In a functional language such as ERLANG, this is not an option.

4. PUTTING IT ALL TOGETHER

Now we are ready to specify the full syntax and semantics of the new `try` construct as it will appear in Release 10 of Erlang/OTP. We begin by revising the exception model from Section 2.

In Section 2.7, we came to the conclusion that exceptions had to be described by a triple ⟨*term, thrown, trace*⟩, and gave the semantics of `catch` and process termination in terms of this. In retrospect, we can refactor the representation to make it more intuitive and easier to work with.

4.1 Semantics of exceptions in Erlang

We define an ERLANG exception to be a triple

⟨*class, term, trace*⟩

such that:

- *class* specifies the class of the exception: this must be either `error`, `exit`, or `throw`,
- *term* is any ERLANG term, used to describe the exception (also referred to as the "reason"),
- *trace* is a partial representation of the stack trace at the point when the exception occurred. The trace may also include the actual parameters passed in function calls; the details are implementation-dependent. There is no special *null* value for the *trace* component.

The different classes of exceptions can be raised as follows:

- When a run-time failure occurs (with reason *term*), or the program calls `erlang:fault(`*term*`)`, the raised exception will have the form ⟨`error`, *term, trace*⟩.
- When the program calls `exit(`*term*`)`, the exception will have the form ⟨`exit`, *term, trace*⟩.
- When the program calls `throw(`*term*`)`, the exception will have the form ⟨`throw`, *term, trace*⟩.

Let T be a function that creates a symbolic representation of a trace as an ERLANG term. The modified semantics of the `catch` operator is shown in Figure 6, and the semantics of process termination in Figure 7. Note that the behaviour remains functionally equivalent to what we described earlier in Figures 2 and 3.

4.2 Syntax and semantics of try-expressions

First, we note that the convention of using tagged tuples such as `{'EXIT', ...}` was introduced only as a means of distinguishing errors and exits from normal return values and thrown terms in the `catch` operator. In our new `try` construct, it is not necessary to stick to this convention. Instead, we will use a syntax which is both easier to read and requires less typing.

A `try`-expression has the general form from Figure 5, where ***Expressions*** and (for $i \in [1, n]$, and $j \in [1, m]$) all the ***Body***$_i$, and ***Handler***$_j$ are comma-separated sequences of one or more expressions, and the ***Pattern***$_i$ are arbitrary

```
If evaluation of Expr completed normally with result R
then
    the result of catch Expr is R,
else
    the evaluation threw exception ⟨class, term, trace⟩;
    if class is throw
    then
        the result of catch Expr is term,
    else if class is exit
    then
        the result is {'EXIT', term}
    else
        the result is {'EXIT', {term, T(trace)}}
```

Figure 6: Final semantics of catch *Expr*

```
If evaluation of the initial call completed normally
then
    the exit term is the atom normal
else
    the evaluation threw exception ⟨class, term, trace⟩;
    if class is exit
    then
        the exit term is term
    else
        the event will be reported to the error logger;
        if class is throw
        then
            the exit term is {{nocatch, term}, T(trace)}
        else
            the exit term is {term, T(trace)}
```

Figure 7: Final semantics of process termination

patterns. As noted in Section 3.2, the `of...` part may be left out. Like in a `case`-expression, variable bindings made in ***Expressions*** are available within the `of...catch` section (but not between `catch...end`), and variables are exported from the clauses if and only if they are bound in all cases.

The exception patterns ***Exception***$_j$ have the following general form:

Class*:*Reason

where ***Class*** is either a variable or a constant. If it is a constant, or a variable which is already bound, the value should be one of the atoms `error`, `exit`, or `throw`.

The ***Class***: part of each exception pattern may be left out, and the pattern is then equivalent to

`throw:`***Reason***

The reason for this shorthand is that typically, only `throw` exceptions should be intercepted. `error` exceptions should in general be allowed to propagate upwards, usually terminating the process, so that unexpected run-time failures are not masked. Furthermore, an `exit` exception means that a decision was made to terminate the process, and a programmer should only override that decision if he knows what he is doing. Therefore, the default class is `throw`.

The semantics of `try` is shown in Figure 8. (As before, clause guards have been left out for the sake of readability, and for the same reason, we do not show the variable

```
If evaluation of Expressions completed normally
with result R,
then
    the result of try...end is equivalent to that of
        case R of
            Pattern_1 -> Body_1;
            ...
            Pattern_n -> Body_n
        end
else
    the evaluation threw exception ⟨class, term, trace⟩;
    for each Exception_j = Class_j:Reason_j
        let Triple_j = {Class_j, Reason_j, _}
    and for each Exception_j = Reason_j
        let Triple_j = {throw, Reason_j, _};
    the result of try...end is then equivalent to that of
        case {class, term, trace} of
            Triple_1 -> Handler_1;
            ...
            Triple_m -> Handler_m;
            X -> rethrow(X)
        end
    where X is a fresh variable.
```

Figure 8: Semantics of try-expressions

scoping rules.) The *rethrow* operator is a built-in primitive which cannot be accessed directly by the programmer; its function is simply to raise the caught exception again, without losing any information. Note that the *trace* component of an exception is assumed to have a cheap, implementation-dependent representation which we do not want to expose to users.

To inspect the stack trace of an exception, a new built-in function `erlang:get_stacktrace()` is added, which returns $T(trace)$ where *trace* is the stack trace component of the last occurred exception of the current process, and T is the function used in the semantics of `catch` in Figure 6. If no exception has occurred so far in the process, an empty list is returned.

Finally, we could add a generic exception-raising function `erlang:raise(Class, Reason)`, where `Class` must be one of the atoms `error`, `exit`, and `throw`, and `Reason` may be any term. We will see one possible use of such a function in the following section.

4.3 Cleanup code

A very common programming pattern is to first prepare for a task (e.g. by allocating one or more resources), then execute the task, and finally do the necessary cleanup. It is then often important that the cleanup stage is executed independently of how the main task is completed, i.e., whether it completes normally or throws an exception. Figure 9 shows an example of how this could be done using try-expressions. However, the code has several weak points:

- The cleanup code, in this example the calls to `close`, must be repeated in each case. (We only show two cases, but in general there could be any number.)
- In the success case, we must bind the result to a new variable just to hold it temporarily while we perform the cleanup.

```
read_file(Filename) ->
  FileHandle = open(Filename, [read]),
  try
    read_opened_file(FileHandle)
  of
    Data ->
      close(FileHandle),
      Data
  catch
    Class:Reason ->
      close(FileHandle),
      erlang:raise(Class, Reason)
  end.
```

Figure 9: Allocate/use/cleanup example

```
try
  Expressions
after
  Cleanup
end
```

Figure 10: `try...after...end` expressions

- The generic failure-case, i.e., the code that catches any exception, performs the cleanup, and then re-throws the exception, is unnecessarily verbose.
- The explicit re-throw using `erlang:raise()` will not preserve the stack trace of the original exception.

All of the above problems can be solved by adding one more feature to our try-expressions. In Common Lisp [9], the special form `unwind-protect` is used to guarantee execution of cleanup-code; in Java [4], this is done by including a `finally` section in try-expressions. The same idea can be used in ERLANG. We start out by defining a new form of try-expression, shown in Figure 10,[5] with a well-defined meaning.

The semantics of `try...after...end` is shown in Figure 11. We can now easily allow the use of `after` directly together with `try...catch...end`, by defining the fully general syntax shown in Figure 12 as equivalent to

```
try
  try Expressions of
    Pattern_1 -> Body_1;
    ...
    Pattern_n -> Body_n
  catch
    Exception_1 -> Handler_1;
    ...
    Exception_m -> Handler_m
  end
after
  Cleanup
end
```

(where as before, the `of...` part may be left out), which guarantees that in all cases, the *Cleanup* code will be exe-

[5]The use of the `after` keyword for this purpose is not yet decided; possibly, a new keyword such as `finally` will be added instead.

```
If evaluation of Expressions completed normally
with result R,
then
    the result of try...after...end is equivalent
    to that of
        begin
            X = R,
            Cleanup,
            X
        end
    where X is a fresh variable,
else
    the evaluation threw exception ⟨class, term, trace⟩;
    the result is then equivalent to that of
        begin
            Cleanup,
            rethrow({class, term, trace})
        end
```

Figure 11: Semantics of try...after...end

```
try Expressions of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
after
  Cleanup
end
```

Figure 12: Fully general try-expression

cuted, after evaluation of one of the *Body*$_i$ or *Handler*$_j$ has completed. This is expected to be the desired behaviour in most cases, since it gives the exception handling clauses a chance to act before any resources are deallocated by the cleanup code. It is also easy to manually nest try-expressions to get another evaluation order, e.g.:

```
try
  try
    Expressions
  after
    Cleanup
  end
of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

4.4 Examples

To demonstrate some uses of try-expressions, we begin by showing in Figure 13 how the **catch** operator may be implemented in a transparent way using **try**. The reader is invited to compare this to the semantics in Figure 6 and verify that they are equivalent.

```
try Expr
catch
  throw:Term -> Term;
  exit:Term -> {'EXIT', Term};
  error:Term ->
    Trace = erlang:get_stacktrace(),
    {'EXIT', {Term, Trace}}
end
```

Figure 13: catch *Expr* implemented with try

```
open(Filename, ModeList) ->
  case file:open(Filename, ModeList) of
    {ok, FileHandle} ->
      FileHandle;
    {error, Reason} ->
      throw({file_error, Reason})
  end.
```

Figure 14: Wrapper for file:open/2.

Figure 14 shows a wrapper function for the standard library `file:open/2` function. The functions in the `file` module return `{ok, Value}` or `{error, Reason}`, while the wrapper always returns a naked value upon success, and otherwise throws `{file_error, Reason}`. The latter makes it easy to identify a file-handling exception even if it is not caught close to where it occurred. Note that if an exception is generated within `file:open/2`, we do not catch it, but allow it to be propagated exactly as it is.

Figure 15 demonstrates the use of **after** in a typical situation where the code allocates a resource, uses it, and afterwards does the necessary cleanup (cf. Figure 9). Note the two-stage application of **try**: First it is used to handle errors in **open** (see Figure 14). Since `FileHandle` is only defined if the call to **open** succeeds, only then need we (or can we) close the file. The next **try** makes sure that the cleanup is done regardless of how the code is exited. An important detail is that by keeping the allocation and the release of the resource close together in the code, no other part of the program needs to know that there is any cleanup to be done.

In an imperative language like Java, it is common to initially assign a null value to variables, and then let the protected section attempt to update them when it is allocating resources. The cleanup section can then test for each resource whether it needs to be released or not. In a functional language where variables cannot be updated, it is cleaner to handle each resource individually, and separate the allocation from the use-and-cleanup, as we did above.

5. RELATED WORK

The concept of user-defined exception handling seems to have originated in PL/I [1], and from there made its way (see e.g. [8]) in different forms into both Lisp [9] and Ada [6], as well as other languages. Ada in its turn was a direct influence on C++ [10], and thus indirectly on Java [4].

In the Object-Oriented languages C++ and Java, an exception is completely described by the thrown object. In

```
read_file(Filename) ->
  try open(Filename, [read]) of
    FileHandle ->
      try
        read_opened_file(FileHandle)
      after
        close(FileHandle)
      end
  catch
    {file_error, Reason} ->
      print_file_error(Reason),
      throw(io_error)
  end.
```

Figure 15: Allocate/use/cleanup with `try...after`

C++, any object can be thrown; in Java, only subclasses of `Throwable` may be thrown. For Java, this means that implementation-specific information like a stack trace may be easily stored in the object without exposing its internal representation. Since this cannot be done in ERLANG without adding a new primitive data type to the language, we have instead chosen to make the stack trace and any other debugging information part of the process state.

In Common Lisp [9], `catch` and `throw` are used for non-local return, and only indirectly for handling actual errors. Also, setting up a `catch` requires specifying a tag (any object, e.g. a symbol) for identifying the catch point, to be used later in the `throw`. To guarantee execution of cleanup-code regardless of how an expression is exited, the special form `unwind-protect` is used. The same effect is achieved in Java by including a `finally` section in `try`-expressions.

In Standard ML [7], the exception handling works much like in Ada, using the operators `raise` and `handle`. As in the originally suggested `try...catch...end` for ERLANG, there is no way of explicitly specifying a success-continuation.

6. CONCLUDING REMARKS

Exceptions in ERLANG has been a not very well understood area of the language, and the behaviour of the existing `catch` operator has for a long time been insufficient. We have given a detailed explanation of how exceptions in modern-day ERLANG actually work, and presented a new theoretical model for exceptions in the ERLANG programming language. Using this model, we have derived a general `try`-construct to allow easy and efficient exception handling, which will be introduced in the forthcoming Release 10 of Erlang/OTP.

7. ACKNOWLEDGMENTS

The authors would like to thank Robert Virding, Fredrik Linder, and Luke Gorrie for their comments and ideas.

8. REFERENCES

[1] American National Standards Institute, New York. *American National Standard: programming language PL/I*, 1979 (Rev 1998). ANSI Standard X3.53-1976.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[3] J. Barklund and R. Virding. Specification of the Standard Erlang Programming Language. Draft version 0.7, June 1999.

[4] J. Gosling, B. Joy, and G. Steele. *The Java™ Programming Language*. The Java Series. Addison-Wesley, 3rd edition, 2000.

[5] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1989.

[6] Military Standard. *Reference Manual for the Ada Programming Language*. United States Government Printing Office, 1983. ANSI/MIL-STD-1815A-1983.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[8] K. Pitman. Condition handling in the Lisp language family. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, number 2022 in LNCS. Springer-Verlag, 2001.

[9] G. L. Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[10] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

Johannes Gutenberg
Born 1394-99 died 1467-1468

Erlguten

Joe Armstrong
joe@sics.se

1(13)

Erlguten

Erlguten is a system for high-quality typesetting

Goal: Better than TeX

Erlguten is work in progress

Erlguten was written by

Joe Armstrong
Michael Karlsson
Sean Hinde

[illegible]

Motivation

TeX does not like absolute positioning

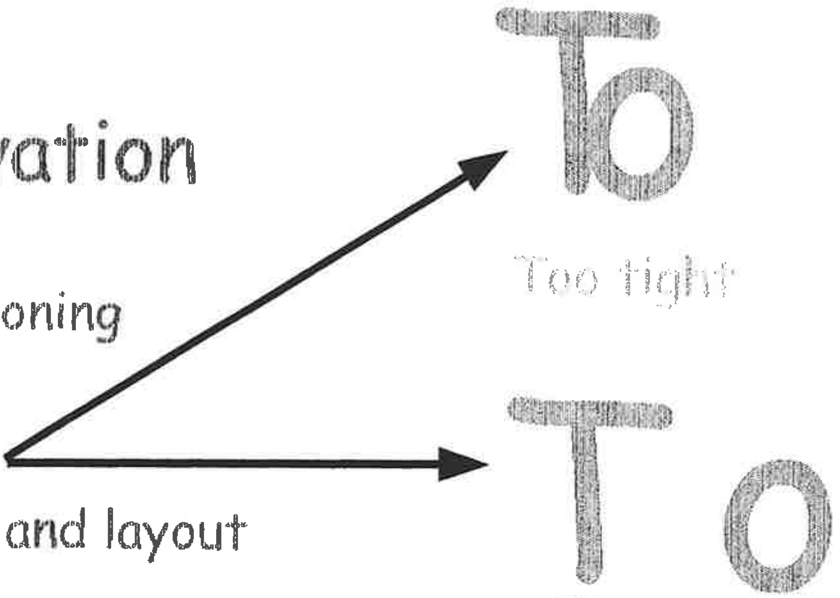
WYSIWYG isn't

Most systems get kerning wrong

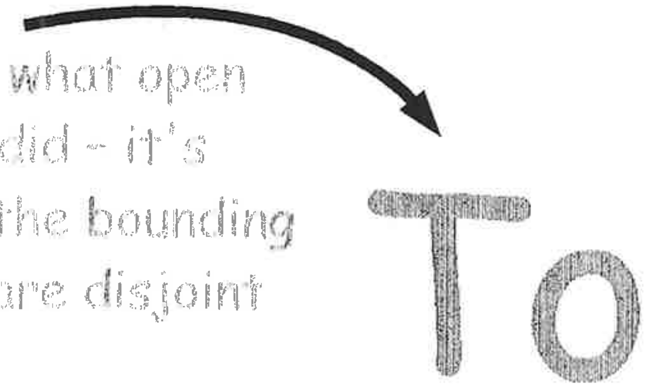
No systems do advanced kerning and layout

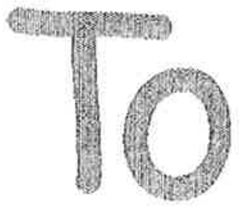
Gutenburg was right

Digital typography has made things worse not better



This is what open office did - it's **wrong** the bounding boxes are disjoint





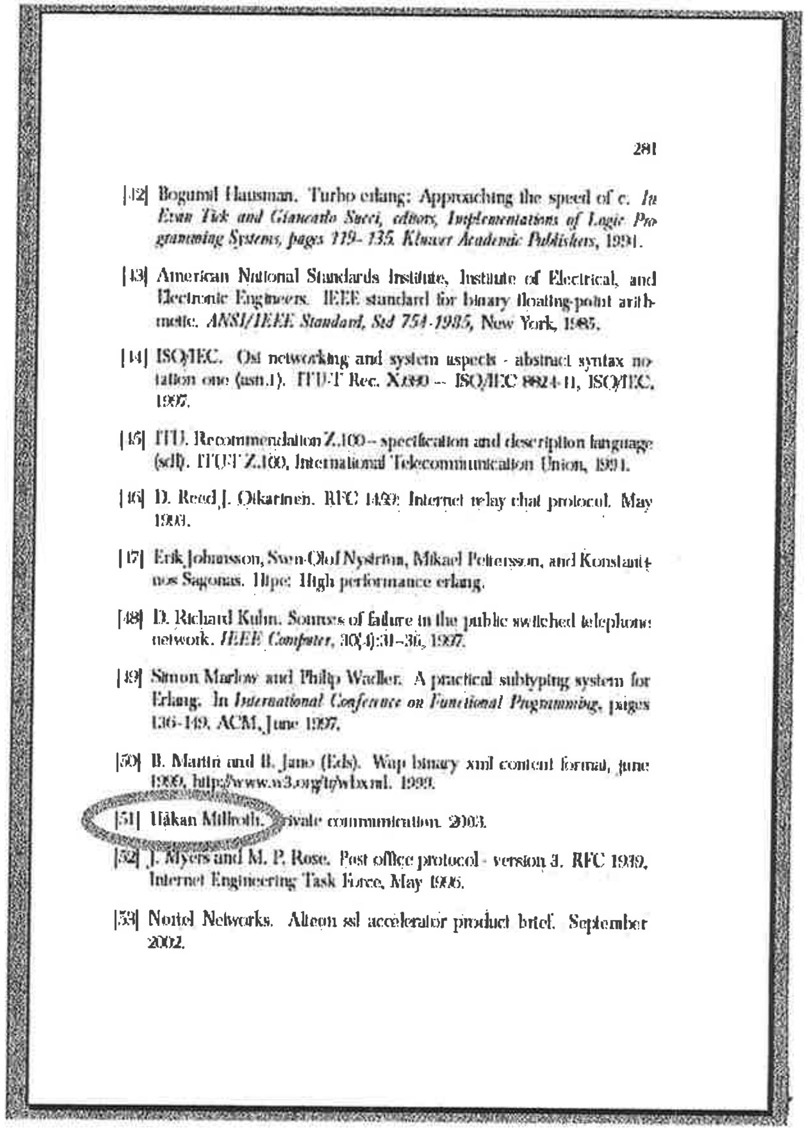
About right

Motivation

The thesis bug

[51] Håkan

These are different a's
Times Roman and Baskerville



281

[42] Bogumil Hausman. Turbo erlang: Approaching the speed of c. *In Evan Tick and Giancarlo Succi, editors, Implementations of Logic Programming Systems, pages 119–135. Kluwer Academic Publishers*, 1994.

[43] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, New York, 1985.

[44] ISO/IEC. Osi networking and system aspects - abstract syntax notation one (asn.1). ITU-T Rec. X.680 -- ISO/IEC 8824-1, ISO/IEC, 1997.

[45] ITU. Recommendation Z.100 -- specification and description language (sdl). ITU-T Z.100, International Telecommunication Union, 1993.

[46] D. Reed J. Oikarinen. RFC 1459: Internet relay chat protocol. May 1993.

[47] Erik Johansson, Sven-Olof Nyström, Mikael Pettersson, and Konstantinos Sagonas. Hipe: High performance erlang.

[48] D. Richard Kuhn. Sources of failure in the public switched telephone network. *IEEE Computer*, 30(4):31–36, 1997.

[49] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *International Conference on Functional Programming*, pages 136-149. ACM, June 1997.

[50] B. Martin and B. Jano (Eds). Wap binary xml content format, june 1999, http://www.w3.org/tr/wbxml. 1999.

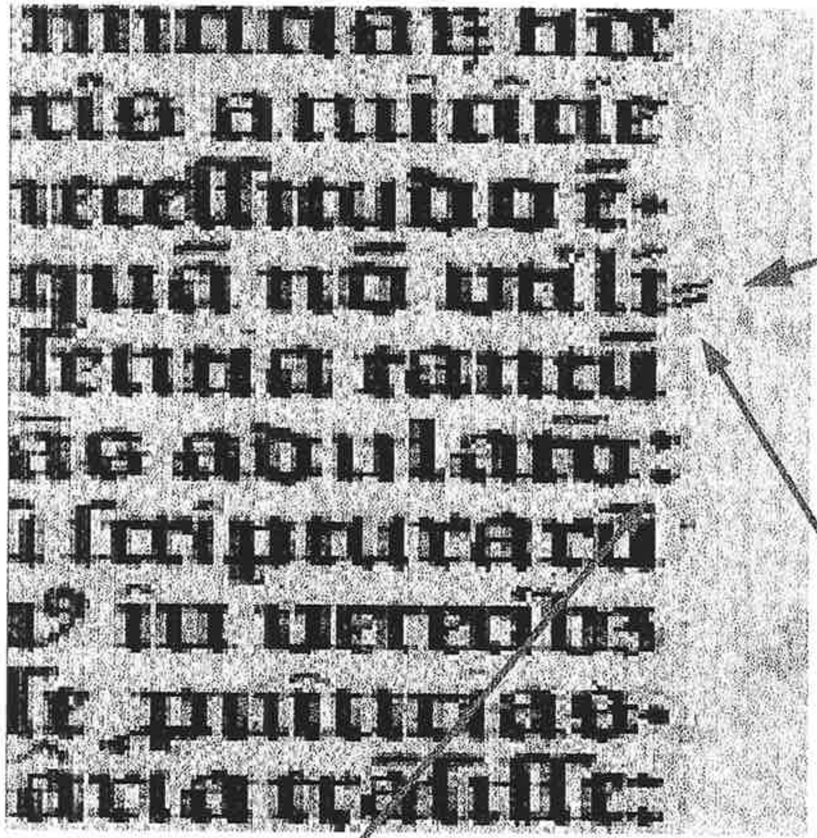
[51] Håkan Millroth. Private communication. 2003.

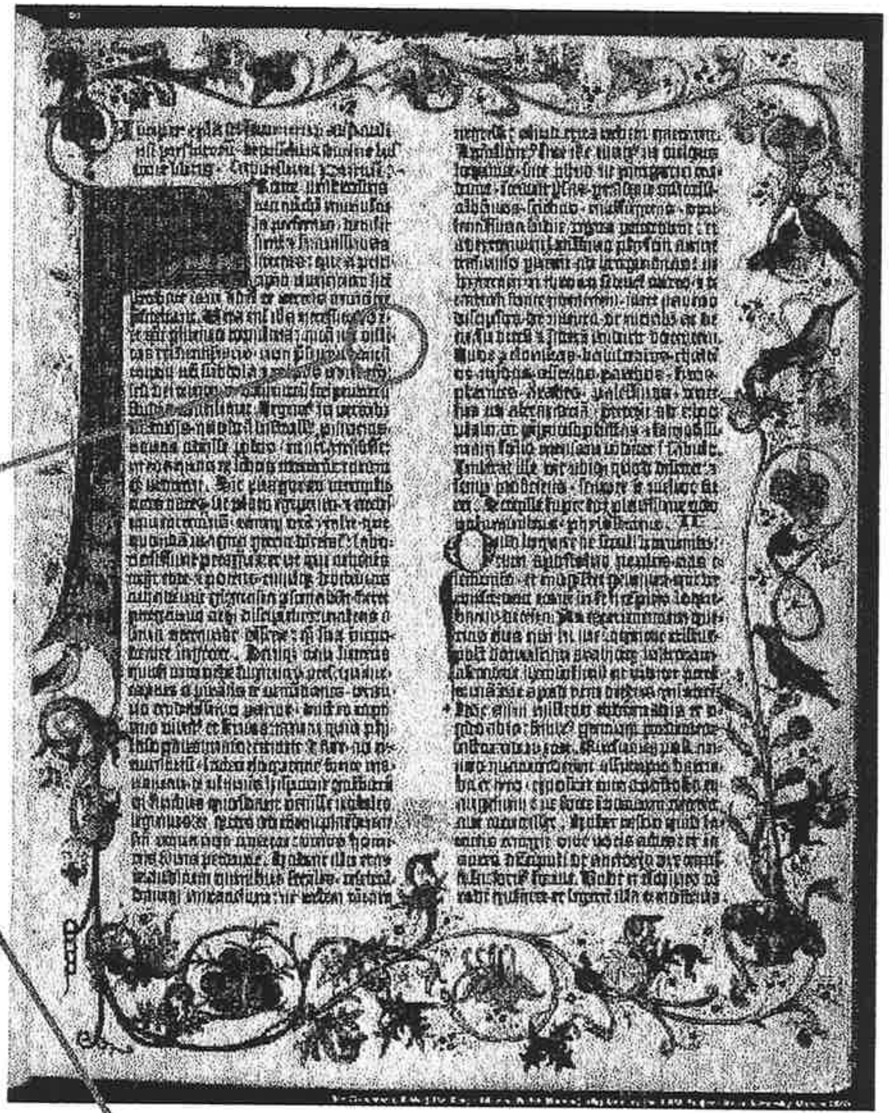
[52] J. Myers and M. P. Rose. Post office protocol - version 3. RFC 1939, Internet Engineering Task Force, May 1996.

[53] Nortel Networks. Alteon ssl accelerator product brief. September 2002.

7

Gutenburg 42 line Bible
c. 1450





Optical alignment

Non-overloaded hypen. This practise was discontinued by Claude Garamond in c. 1545.

Goals

Easy to use - simple textual input

Multi-mode inputs (suited for technical reports, magazine layout and presentations)

Very high quality layout engine (better than TeX, in-design, quark etc.)

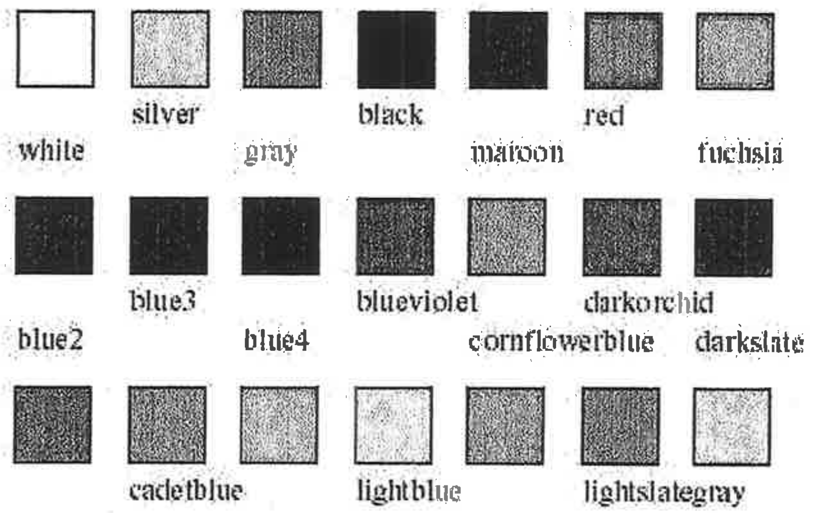
Technical

Pure Erlang
Uses only the in-built PDF fonts
Implement "micotypographic techniques in the HZ program"
(margin kerning, otical alignment of margins, kerning and
tracking adjustments to text)

[illegible]

pdf.erl (mikael)

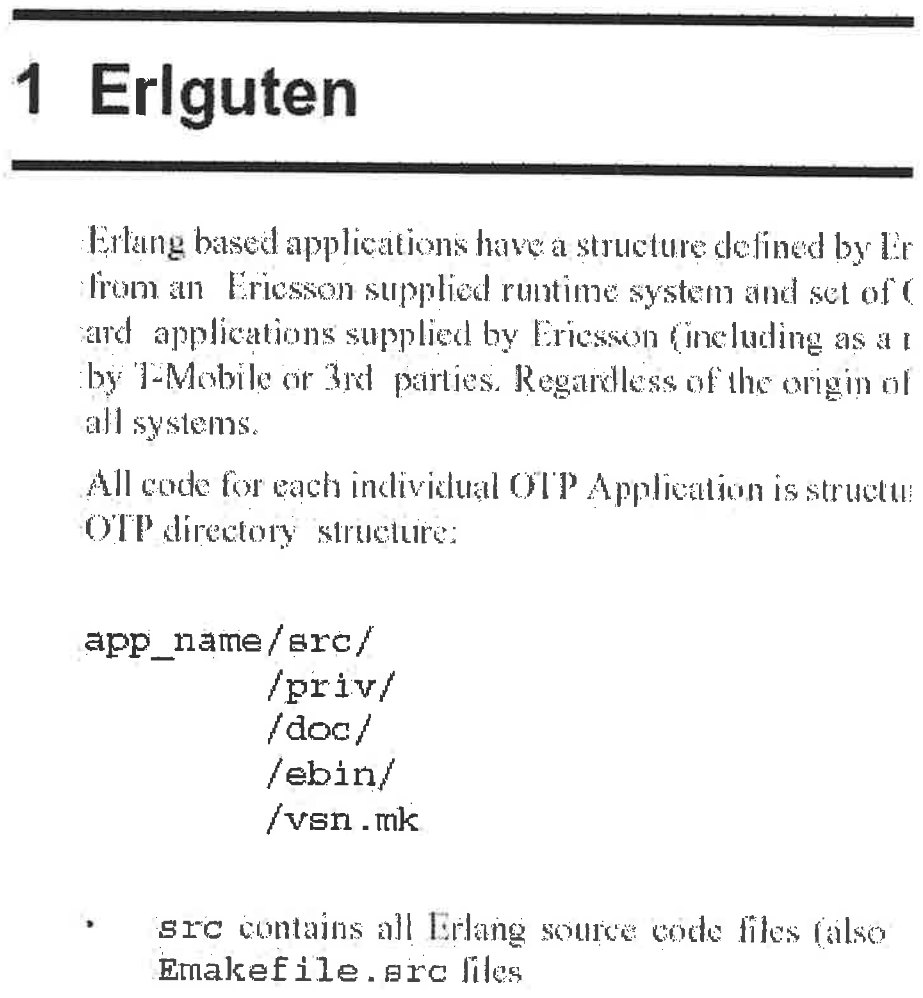
```
colortest1(PDF,N,[])->
     [];
colortest1(PDF,N,[H|T])->
    pdf:set_fill_color(PDF,H),
    pdf:rectangle(PDF,{0,20},{20,20}),
    pdf:path(PDF,fill_stroke),
    pdf:set_fill_color(PDF,black),
    pdf:begin_text(PDF),
    pdf:set_font(PDF,"Times-Roman", 8),
    pdf:set_text_pos(PDF,0,(N rem 2)*10),
    pdf:text(PDF,atom_to_list(H)),
    pdf:end_text(PDF),
    pdf:translate(PDF,30,0),
    colortest1(PDF,N+1,T).
```



DOCBOOK Light (Sean)

```
<section1>
  <title>Erlguten</title>
  <para>Erlang based applications
  ...
  <code>
app_name/src/
        /priv

  <list>
    <item>
      <code>src</code> contains
```



1 Erlguten

Erlang based applications have a structure defined by Er from an Ericsson supplied runtime system and set of (ard applications supplied by Ericsson (including as a r by T-Mobile or 3rd parties. Regardless of the origin of all systems.

All code for each individual OTP Application is structu OTP directory structure:

```
app_name/src/
        /priv/
        /doc/
        /ebin/
        /vsn.mk
```

- `src` contains all Erlang source code files (also `Emakefile.src` files

ML9 (Joe)

Simple data structures

```
<tag1 name="joe" date="2004-10-11">
 <list>
  <item>
     <code>src</code> contains
     ...
  </item>
  ...
 </list>
</tag1>
```

```
@tag1 {name={"joe","armstrong"},
       date={2004,10,11}}

* ''src'' contains
```

Complex Data structures

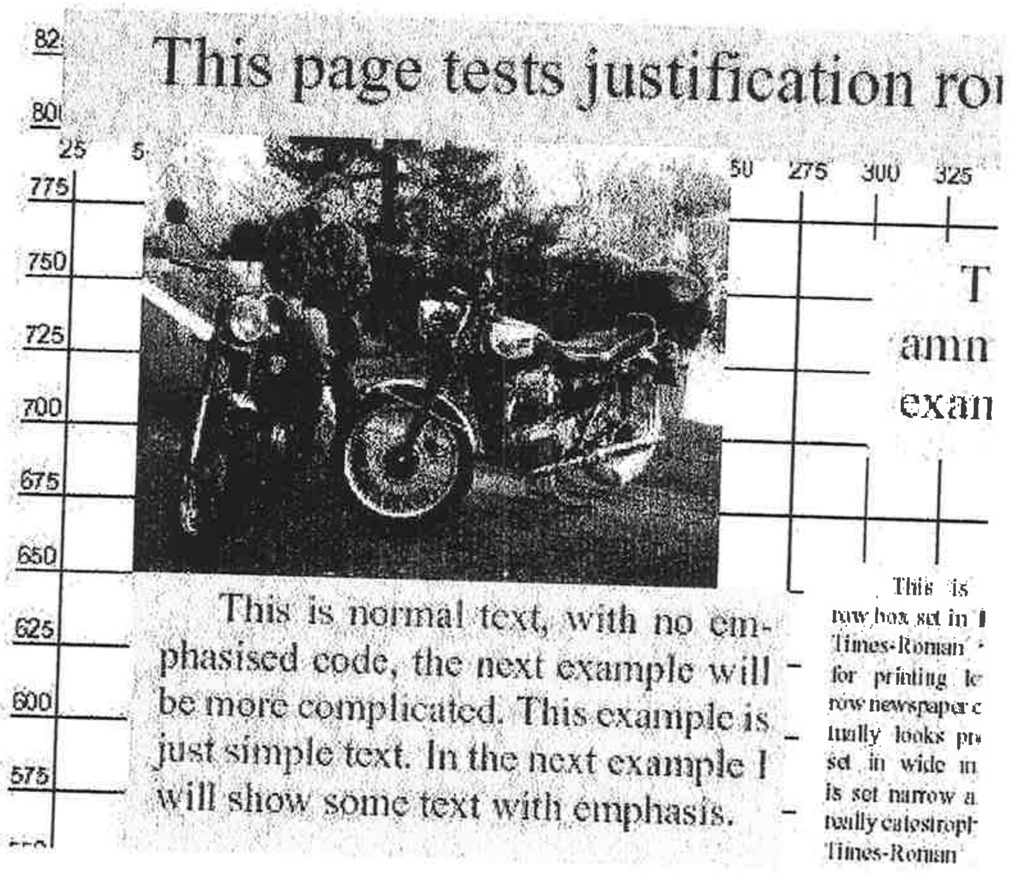
ML9

```
@autoexec {mod="handler"}                ←――――――  Run handler on this file
@erlang
-module(foo).                            ←――――――――――――――――  Erlang text
...
@doc {keywords=[code,doc]}
This is ''very nice'' because:           ←――――――――――――――――  Wiki text
* it's ''easy'' to learn
* easy to type
@java
import java.applet.*;
public class Foo exetends Applet {          ←――――――――――――  Java
  private boo=0;
@makefile name="Makfile"
.SUFFIXES .erl.beam
...
```

Magazine production (Joe)

```
@autoexec {mod="pdf"}
@include "magazine"
@heading This page tests
justification routines
@leftBox
This is normal text with no
emphasised code.
...
@image "bikes.jpg"
```

```
@grid {page=a4,dx=20,dy=20}
@box {name="heading",x=1,y=1,
color="yellow", face="TimeRoman",
pointSize=24,justification=left}
@image {x=2,y=3}
```



This page tests justification ro

T
amn
exan

This is normal text, with no emphasised code, the next example will be more complicated. This example is just simple text. In the next example I will show some text with emphasis.

This is
row box set in
Times-Roman
for printing
row newspaper c
tually looks pr
set in wide m
is set narrow a
tually catastroph
Times-Roman

Status

It works but ...

Unstable (I keep re-writing it)

Taking a lot longer than I expected

(Knuth "paragraph justification is really difficult") (most programs get it wrong!)

Fun

Needs integrating with ex11

Proposal for an Erlang foundation

Erlang User Conference '04

21st. October 2004

Mickaël Rémond <mickael.remond@erlang-fr.org>

Thierry Mallard <thierry.mallard@erlang-fr.org>

ERLANG projects

www.erlang-projects.org

1(13)

Erlang: What did we achieve ?

- Erlang/OTP rely on a strong and powerful technical asset.
  It stays very high in network & server development
- Some potential killer-apps (Yaws, Wings3D, Tsunami, Ejabberd, J-EAI, ...)
- Maturity: 10th Erlang User Conference, 18th year for Erlang
- Nice progression in statistics on Erlang.org
- More companies producing Erlang products and software
- Open Source since 1998

ERLANG

20

Erlang: What's next ?

We still have significant weaknesses :

Mindshare

- Marketing: despite being a top technical environment Erlang is not largely covered in computing media.
- Userbase: We need to accelerate the growth of the Erlang users base.

In a competive world, if we do not grow very fast, we are decreasing relatively to the other alternatives. This means that:

- We are losing opportunities to develop Erlang in new interesting environments and to replace legacy systems
- We might lose the lead we current have over the competition.

ERLANG

3

A common effort is needed to promote and develop Erlang mindshare

- To show the industry that Erlang is used by many companies and is here to stay (potential customers are worried)
- To help increase userbase, and increase Erlang business
- To make it easier to hire Erlang developers
- To make it easier to find Erlang services and consultancy companies
- To allow coordination of developments needed common software, bindings, extensions or demonstrators
- To help develop Erlang as a strong marketing point for Erlang products and software (*Branding*)

Toward an Erlang foundation (1/2)

- We propose to turn Erlang-projects into an Erlang foundation to gather companies having interests in Erlang development.
- A place to discuss what would help companies doing better business with Erlang.
- discuss and coordinate developments that will help improving the Erlang environment:
  - Commonly needed libraries
  - Marketing tools
  - Enhancing OTP

ERLANG

Toward an Erlang foundation (2/2)

- To provide a hardware and software infrastructure for members to participate in the foundation activity:
  - Metafrog to coordinate developments,
  - Erlang application hosting,
- Participation in events (conferences, booth on expos, etc.)
- Creation of local country based representatives of Erlang-projects: Local country based representings for the Erlang-Project foundation.

ERLANG

σ

Examples from current Erlang-projects activities

Metafrog

- Coordination **platform** for Erlang development efforts
- **Showcase** for big Erlang-based application. Demonstrate Erlang/OTP and Yaws high-performance and reliability
- Simple and **efficient** collaborative platform.

ERLANG

x1

Examples from current Erlang-projects activities

Solutions Linux 2004 booth

- Presence of Erlang projects on the expo: Booth in the non-profit area.
- Conference on Erlang and clustering.
- Lots of interesting contacts.
- Presence with other consortium such as Objectweb on the expo.

It was important to be there for Erlang promotion and helped dissemination of the technology.

Conference and booth for Solutions Linux 2005 already planned

ERLANG

Examples from current Erlang-projects activities

Erlang REPOS

- preconfigured Erlang environment and applications (on CDROM, dumpable on hard drive).
- Big marketing impact (hundreds of downloads already). Magazines are interested for cover CD diffusion.
- Has proven very useful to distribute software (J-EAI)
- Has a strong impact on multiplatform compliance improvement (Example: Patches for Yaws, ejabberd, etc.).

ERLANG

9

Examples from current Erlang-projects activities

Other ideas:

- Pluggable distribution layer in Erlang ?
- Improving Erlang interoperability is needed. We need to provide robust and easy tools to quickly develop multi-platform binding to C libraries.
- More interoperability libraries are needed (SOAP, WSDL, ...).
- Develop example web applications that will prove Erlang high performance (Example RUBIS implementation, comparable with Java and PHP).

ERLANG

Erlang-projects foundation

Built on the same model than the Apache Foundation.

See http://www.apache.org/foundation/how-it-works.html

Erlang-projects foundation mission could be stated as:

Develop and promote valuable projects that could turn Erlang into a long-term solution for middleware development, server applications and highly concurrent graphical user applications.

Please do not hesitate to promote your own mission

ERLANG

Erlang-projects foundation

Discuss organisation:

- Fees (each euro in communication through Erlang project will have more impact than isolated communication).
- Define the long-term mission
- Roadmap
- Bootstrap period
- Executive board for the foundation. This is the basis for a lightweight organisation of the foundation.

ERLANG

Erlang-projects foundation

Open discussion

ERLANG

1(2)

Highlights
Erlang 5.4/OTP R10B

This document describes the major new features of and changes to Erlang 5.4/OTP R10B, compared to Erlang 5.3/OTP R9C with focus on the major completely new things which has not already been delivered as patches to R9C. Some of the new features and changes have already been delivered as patches to R9C and/or R9B. For Open Source users this document can also serve as a good approximation to the differences compared to R9C-2 (corresponding to OTP patchlevel 663).

For more detailed information, please refer to the release notes for the individual applications.

Documentation

- A new tutorial **"Getting Started With Erlang"**.
  This is a "kick start" tutorial to get you started with Erlang. Everything in it is true, but only part of the truth. For example, It will only tell you the simplest form of the syntax, not all esoteric forms. Together with the Erlang Reference Manual it should be a very good start for learning Erlang.

ERTS, Erlang emulator

- Process and port identifiers have been made more unique. Now we use 28 bits for the internal representation compared to 18 before. This also increases the maximum number of Erlang processes (the new maximum is 268435456 previously it was 262144). The new representation of a Pid (process identifier) has impact on the Erlang distribution protocol. The use of new or old Pid representation can be controlled with a flag when the Erlang node is started.
- Significantly improved performance regarding handling of links and monitors in the emulator. This will have positive impact on performance for a system with many dynamical processes who are using links. Measurement in a real system is necessary in order to know exactly how much this improves performance.
- Support for the new language construct **try?catch**.

Standard libraries (kernel, stdlib)

- Query List Comprehensions QLC which is a very convienient way to perform queries on ets and mnesia tables. QLC is intended to replace mnemosyne in forthcoming releases. The use of mnemosyne is already strongly discouraged especially in time critical applications since it is very resource consuming and cpu demanding. This is the first version and there will be more optimizations added in forthcoming releases. There is also additional functionality in mnesia to support QLC. Note that mnemosyne is still included and works exactly as before.
- Support for records in the Erlang Shell. There are commands for reading record definitions from files and for manipulating record definitions. The record syntax can be used in the shell and return values are printed as records when possible using the record definitions known to the shell.

- New function insert_new/2 in ets and dets. Will only succed if this is the first insert for a key.

Compiler, Erlang compiler

- The semantics for boolean operators in guards have been changed to be more consistent.
- The compiler now warns for more types of suspect code, e.g as expressions that will fail in runtime (such as atom-42), guards that are always false and patterns that cannot match.
- Improved compilation speed with ERLC (starts as little of the Erlang system as possible, for example avoiding ?start_sasl?).
- The new language construct **try?catch** is supported.

New Applications

- New application **xmerl** which is a validating XML-parser with many useful features for handling and transforming XML-data in Erlang. This application was originally created by Ulf Wiger and has been developed at Sourceforge.net for several years.. It has now been included in the Erlang/OTP distribution and is supported by the OTP team.
- New applications **edoc** and **syntax_tools** which can be used to produce documentation of Erlang modules by means of special comments in the Erlang source code. These applications are originally created by Richard Carlsson at Uppsala University and are now included in the Erlang/OTP distribution. The reason for this is to offer one "standardized" way to document Erlang, where the necessary tools are always available.

Other Applications

- **Erl_interface** and **J_interface**: Support for new format of Pid and a compatibility mode so that old format can be used.
- **Mnesia**: support for QLC (the mnesia:table function) otherwise the same as in R9C.
- **Observer/Crashdump_viewer** can now handle really large crashdumps without getting problems with that the Internet browser times out.

Erlang/OTP User Conference 2004

Speakers and chairman

Joe	Armstrong	SICS	Kista	Sweden	joe@sics.se
Thomas	Arts	IT university	Göteborg	Sweden	thomas.arts@ituniv.se
Per	Bergqvist	Synapse	Stockholm	Sweden	per@synap.se
Johan	Blom	Mobile Arts	Stockholm	Sweden	Johan.Blom@mobilearts.se
Richard	Carlsson	Uppsala univ	Uppsala	Sweden	richardc@csd.uu.se
Bjarne	Däcker	cs-lab.org	Segeltorp	Sweden	bjarne@cs-lab.org
Tobias	Lindahl	Uppsala univ	Uppsala	Sweden	Tobias.Lindahl@it.uu.se
Kenneth	Lundin	Ericsson Erlang/OTP unit	Stockholm	Sweden	kenneth.lundin@ericsson.com
Göran	Oettinger	Mobile Arts	Stockholm	Sweden	goran.oettinger@mobilearts.se
Mickaël	Rémond	Erlang projects	Paris	France	mickael.remond@erlang-fr.org
Jouni	Rynö	Finnish Meteorological Institute	Helsinki	Finland	Jouni.Ryno@fmi.fi
Juan José	Sánchez Penas	Univ of A Coruña	Coruña	Spain	juanjo@dc.fi.udc.es
Fredrik	Thulin	Stockholm univ	Stockholm	Sweden	ft@it.su.se

Participants

Kristoffer	Andersson	Synapse	Stockholm	Sweden	
Peter	Andersson	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Ingela	Andin	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Gösta	Ask	SalveLinus	Stockholm	Sweden	g.ask@telia.com
Simon	Aurel	Erlang Training & Consulting	London	England	
Knut	Bakke	Teleca Wireless Solutions AS	Grimstad	Norway	Knut.Bakke@teleca.no
Sreedhar	Bandaru	Ericsson Enterprise	Stockholm	Sweden	sreedhar.bandaru@wipro.com
John-Olof	Bauner	Ericsson	Kista	Sweden	john-olof.bauner@bredband.net
Johan	Berg	Ericsson	Stockholm	Sweden	johan.berg@ericsson.com
Johan	Bevemyr	Nortel Networks	Stockholm	Sweden	jb@bluetail.com
Katrin	Bevemyr	Nortel Networks	Stockholm	Sweden	
Martin	Björklund	Nortel Networks	Stockholm	Sweden	mbj@bluetail.com
Hans	Bolinder	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Pascal	Brisset	Cellicium	Bagneux	France	pascal.brisset@cellicium.com

1(4)

Participants cont.

Göran	Båge	Mobile Arts	Stockholm	Sweden	goran.bage@mobilearts.se
Martin	Carlson	Erlang Training & Consulting	London	England	
Francesco	Cesarini	Erlang Training & Consulting	London	England	francesco@erlang-consulting.com
Mats	Cronqvist	Ericsson	Budapest	Hungary	mats.cronqvist@ericsson.com
Niclas	Eklund	Ericsson Erlang/OTP unit	Stockholm	Sweden	nick@erix.ericsson.se
Morgan	Eriksson	Nortel Networks	Stockholm	Sweden	
Lars-Åke	Fredlund	SICS	Kista	Sweden	fred@sics.se
Magnus	Fröberg	Nortel Networks	Stockholm	Sweden	magnus@bluetail.com
Luke	Gorrie	Synapse	Stockholm	Sweden	
Pär	Grandin	Ericsson	Stockholm	Sweden	par.grandin@ericsson.com
Joakim	Grebenö	Nortel Networks	Stockholm	Sweden	jocke@bluetail.com
Rickard	Green	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Dan	Gudmundsson	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Björn	Gustavsson	Ericsson Erlang/OTP unit	Stockholm	Sweden	bjorn@erix.ericsson.se
Per	Hallin	Synapse	Stockholm	Sweden	
Niklas	Hanberger	Nortel Networks	Stockholm	Sweden	
Siri	Hansen	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Per	Hedeland	Nortel Networks	Stockholm	Sweden	per@bluetail.com
Sean	Hinde	T-mobile	London	England	sean.hinde@mac.com
John	Hughes	Chalmers	Göteborg	Sweden	rjmh@cs.chalmers.se
Klas	Johansson	Ericsson	Linköping	Sweden	klas.johansson@ericsson.com
Leif	Johansson	Ericsson	Göteborg	Sweden	leif.d.johansson@ericsson.com
Torbjörn	Johnson		Stockholm	Sweden	torbjorn.k.johnson@swipnet.se
Bertil	Karlsson	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Mikael	Karlsson	Creado Systems	Stockholm	Sweden	mikael.karlsson@creado.com
Bengt	Kleberg	Ericsson	Stockholm	Sweden	bengt.kleberg@ericsson.com
Tord	Larsson	Nortel Networks	Stockholm	Sweden	tlarsson@nortelnetworks.com
Lars	Lindgren	Synapse	Stockholm	Sweden	
Peter	Lund	Lundata AB	Stockholm	Sweden	Peter.Lund@lundata.se

Participants cont.

Peter	Lundell	Ericsson	Stockholm	Sweden	peter.lundell@ericsson.com
Matthias	Läng	Corelatus AB	Stockholm	Sweden	matthias@corelatus.se
Ann-Marie	Löf	Sjöland & Thyselius Telecom AB	Stockholm	Sweden	ann-marie.lof@st.se
Arild	Løvendahl	Teleca Wireless Solutions AS	Grimstad	Norway	Arild.Lovendahl@teleca.no
Luca	Manai	Ericsson	Stockholm	Sweden	luca.manai@ericsson.com
Peter-Henry	Mander		Thame	England	erlang@manderp.freeserve.co.uk
Håkan	Mattsson	Ericsson Erlang/OTP unit	Stockholm	Sweden	hakan@erix.ericsson.se
Peter	Nagy	Ericsson	Budapest	Hungary	peter.nagy@ericsson.com
Vinay	Navsingoju	Ericsson Enterprise	Stockholm	Sweden	vinay.navsingoju@wipro.com
Hans	Nilsson	Ericsson	Stockholm	Sweden	hans.r.nilsson@ericsson.com
Raimo	Niskanen	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Patrik	Nyblom	Ericsson Erlang/OTP unit	Stockholm	Sweden	pan@erix.ericsson.se
Laurent	Picouleau	Erlang Training & Consulting	London	England	
Robert	Raschke		Edinburgh	Scotland	rrerlang@tombob.com
Tony	Rogvall	Synapse	Stockholm	Sweden	tony@rogvall.com
Per	Romin	Ericsson Enterprise	Stockholm	Sweden	per.romin@ericsson.com
Håkan	Stenholm	Stockholm univ	Stockholm	Sweden	hakan.stenholm@mbox304.swipnet.se
Erik	Stenman	Virtutech AB	Stockholm	Sweden	stenman@virtutech.com
Per	Sternås	Ericsson Enterprise	Stockholm	Sweden	per.sternas@ericsson.com
Sebastian	Strollo	Nortel Networks	Stockholm	Sweden	seb@bluetail.com
Anton	Strydom	Synapse	Stockholm	Sweden	
Per Einar	Strömme	K-tech	Stockholm	Sweden	stromme@telia.com
Göran	Stupalo	Ericsson Erlang/OTP unit	Stockholm	Sweden	
Ulf	Svarte Bagge	Corelatus AB	Stockholm	Sweden	ulf@corelatus.se
Torbjörn	Törnkvist	Nortel Networks	Stockholm	Sweden	tobbe@nortelnetworks.com
Jane	Walerud		Stockholm	Sweden	jane@walerud.com
Paul	van Teeffelen	Ericsson	Stockholm	Sweden	paul.van.teeffelen@ericsson.com
Esko	Vierumäki	Ericsson	Stockholm	Sweden	esko.vierumaki@ericsson.com
Ulf	Wiger	Ericsson	Stockholm	Sweden	ulf.wiger@ericsson.com

Participants cont.					
Daniel	Wiik	Ericsson	Linköping	Sweden	daniel.wiik@ericsson.com
Claes	Wikström	Nortel Networks	Stockholm	Sweden	klacke@nortelnetworks.com
Jerker	Wilander		Lucenay	France	edberg.wilander@stockholm.mail.telia.com
Chris	Williams	Ericsson	Stockholm	Sweden	chris.williams@ericsson.com
Mike	Williams	Ericsson	Stockholm	Sweden	michael.williams@ericsson.com
Johan	Wärlander	St Jude Medical AB	Järfälla	Sweden	jwarlander@sjm.com
Lennart	Öhman	Sjöland & Thyselius Telecom AB	Stockholm	Sweden	lennart.ohman@st.se
Göran	Östlund		Stockholm	Sweden	goran.may@chello.se

Updated 2004-10-14

4/4