8th International Erlang/OTP User Conference

Stockholm, November 19, 2002

Proceedings

EUC'2002 http://www.erlang.se/euc/02/
Ericsson AB
P.O. Box 1505
SE-125 25 Älvsjö Stockholm
Sweden

ERICSSON

ERLANG

Erlang/OTP User Conference 2002

Conference Programme

08.30 *Registration*

Applications I

09.00 **Yaws.**
Claes Wikström, Alteon WebSystems.

09.30 **Implementing the Mobile Location Protocol: A Tale from the Trenches.**
Magnus Eklund, Fredrik Linder and Thomas Lindgren, Cellpoint.

10.00 **The BLIS4 Platform and Development Experiences.**
Thomas Verner, BluePosition.

10.30 *Coffee*

Applications II

11.00 **HELGA - A Call Load Generator Written in Erlang/OTP.**
Anand Balagopalakrishnan and Bagirath Krishnamachari, Lucent Technologies.

11.30 **The AXC 105 Fibre Switch.**
Hans Nilsson, Ericsson.

11.50 **Mobile Arts High Performance Telecom Platform.**
Johan Blom and Göran Båge, Mobile Arts.

12.10 **Developing for the Web in Erlang: Why and how ?**
Mickaël Rémond, erlang-fr.org.

12.30 *Lunch*

Technology I

14.00 **On Reducing Interprocess Communication Overhead in Concurrent Programs.**
Erik Stenman and Konstantinos Sagonas, Uppsala university.

14.30 **Distel: Distributed Emacs Lisp (for Erlang).**
Luke Gorrie, Alteon WebSystems.

15.00 **Static Analysis of Communications in Erlang Programs.**
Fabien Dagnat, ENST, Bretagne.

15.30 *Coffee*

Technology II

16.00 **Stand Alone Erlang.**
Joe Armstrong, SICS.

16.30 **Use of Erlang in System Test of AXD 301.**
Karl Olsson, Ericsson.

17.00 **The Erlang/OTP R9 Release.**
Kenneth Lundin, OTP Unit, Ericsson.

17.30 *Close (and pub evening)*

Demonstrations (during intermissions)

BluePosition demonstrates their innovative **B**luetooth **L**ocation **I**nformation **S**ystem - BLIS4.

Yaws - Yet Another Web Server

Claes Wikstrom
klacke@hyber.org

11th November 2002

Contents

Chapter 1

Introduction

YAWS is an ERLANG web server. It's written in ERLANG and it uses ERLANG as its embedded language similar to PHP in Apache or Java in Tomcat.

The advantages of ERLANG as an embedded web page language as opposed to Java or PHP are many.

- Speed - Using ERLANG for both implementing the web server itself as well as embedded script language gives excellent dynamic page generation performance.
- Beauty - Well this is subjective
- Scalability - due to the light weight processes of ERLANG , YAWS is able to handle a very large number of concurrent connections

YAWS has a wide feature set, it supports:

1. HTTP 1.0 and HTTP 1.1
2. Static content page delivery
3. Dynamic content generation using embedded ERLANG code in the HTML pages
4. Common Log Format traffic logs
5. Virtual hosting with several servers on the same IP address
6. Multiple servers on multiple IP addresses.

7. HTTP tracing for debugging
8. An interactive interpreter environment in the Web server while developing and debugging the web site.
9. RAM caching of commonly accessed pages.
10. Full streaming capabilities of both up and down load of dynamically generated pages.
11. SSL
12. Support for WWW-Authenticated pages.
13. Support API for cookie based sessions.
14. Application Modules where virtual directory hierarchies can be made.
15. Embedded mode

1.1 Prerequisites

This document requires that the reader:

- Is well acquainted with the ERLANG programming language
- Understands basic Web technologies.

1.2 A tiny example

We introduce YAWS by help of a tiny example. The web server YAWS serves and delivers static content pages similar to any old web server, except that YAWS does this much faster than most web servers. It's the dynamic pages that makes YAWS interesting. Any page with the suffix ".yaws" is considered a dynamic YAWS page. A YAWS page can contain embedded ERLANG snippets that are executed while the page is being delivered to the WWW browser.

Example 1.1 is the HTML code for a small YAWS page.

It illustrates the basic idea behind YAWS . The HTML code can contain <erl> and </erl> tags and inside these tags an ERLANG function called out/1 gets called and the output of that function is inserted into the HTML document, dynamically.

It is possible to have several chunks of HTML code together with several chunks of ERLANG code in the same YAWS page.

The `Arg` argument supplied to the automatically invoked `out/1` function is an ERLANG record that contains various data which is interesting when generating dynamic pages. For example the HTTP

```
<html>

<p> First paragraph

<erl>
out(Arg) ->
    {html, "<p>This string gets inserted into HTML document dynamically"}.
</erl>

<p> And here is some more HTML code

</html>
```

Figure 1.1: Example 1.1

headers which were sent from the WWW client, the actual TCP/IP socket leading to the WWW client. This will be elaborated on throughly in later chapters.

The `out/1` function returned the tuple `{html, String}` and `String` gets inserted into the HTML output. There are number of different return values that can be returned from the `out/1` function in order to control the behavior and output from the YAWS web server.

Chapter 2

Compile, Install, Config and Run

This chapter is more of a "Getting started" guide than a full description of the YAWS configuration. YAWS is hosted on Sourceforge at *http://sourceforge.net/projects/erlyaws/* . This is where the source code resides in a CVS repository and the latest unreleased version is available through anonymous CVS through the following commands:

```
# export CVS_RSH=ssh
# export CVSROOT=:pserver:anonymous@cvs.erlyaws.sourceforge.net:/cvsroot/erlyaws
# cvs  login
# cvs -z3 co .
```

Released version of YAWS are available either at the Sourceforge site or at *http://yaws.hyber.org/download.*

2.0.1 Compile and Install

To compile and install a YAWS release one of the prerequisites is a properly installed ERLANG system. YAWS runs on ERLANG releases OTP R8 and later. Get ERLANG from *http://www.erlang.org*

Compile and install is straight forward:

```
# cd /usr/local/src
# tar xfz yaws-X.XX.tar.gz
# cd yaws
# make
# make install
```

There is no configure script (yet) since there are no items to configure.

The make command will compile the YAWS web server with the `erlc` compiler found in $PATH.

make install - will install the executable - called **yaws** in /usr/local/bin/ and a working configuration file in */etc/yaws.conf*

make local_install will install the executable in $HOME/bin and a working configuration file in $HOME/yaws.conf

While developing a YAWS site, it's typically most convenient to use the local_install and run YAWS as a non privileged user.

2.0.2 Configure

Let's take a look at the config file that gets written to $HOME after a local_install.

```
# first we have a set of globals

logdir = .
ebin_dir = /home/klacke/yaws/yaws/examples/ebin
include_dir = /home/klacke/yaws/yaws/examples/include

# and then a set of servers

<server localhost>
        port = 8000
        listen = 127.0.0.1
        docroot = /home/klacke/yaws/yaws/scripts/../www
</server>
```

Figure 2.1: Minimal Local Configuration

The configuration consists of an initial set of global variables that are valid for all defined servers.

The only global directive we need to care about for now is the logdir. YAWS produces a number of log files and they will - using the Configuration from Figure 2.1 - end up in the current working directory. We start YAWS interactively as

```
# ~/bin/yaws -i
Erlang (BEAM) emulator version 5.1.2.b2 [source]

Eshell V5.1.2.b2  (abort with ^G)
1>
=INFO REPORT==== 30-Oct-2002::01:38:22 ===
```

```
Using config file /home/klacke/yaws.conf
=INFO REPORT==== 30-Oct-2002::01:38:22 ===
Listening to 127.0.0.1:8000 for servers ["localhost:8000"]

1>
```

By starting YAWS in interactive mode (using the command switch *-i* we get a regular ERLANG prompt. This is most convenient when developing YAWS /http pages. For example we:

- Can dynamically compile and load optional helper modules we need.
- Get all the crash and error reports written directly to the terminal.

The configuration in Example 2.1 defined one HTTP server on address 127.0.0.1:8000 called "localhost". It is important to understand the difference between the name and the address of a server. The name is the expected value in the client Host: header. That is typically the same as the fully qualified DNS name of the server whereas the address is the actual IP address of the server.

Since YAWS support virtual hosting with several servers on the same IP address, this matters.

Nevertheless, our server listens to *127.0.0.1:8000* and has the name "localhost", thus the correct URL for this server is *http://localhost:8000.*

The document root (docroot) for the server is set to the www directory in the YAWS source code distribution. This directory contains a bunch of examples and we should be able to run all those example now on the URL *http://localhost:8000.*

Instead of editing and adding files in the YAWS www directory, we create yet another server on the same IP address but a different port number - and in particular a different document root where we can add our own files.

```
# mkdir ~/test
# mkdir ~/test/logs
```

Now change the config so it looks like this:

```
logdir = /home/klacke/test/logs
ebin_dir = /home/klacke/test
include_dir = /home/klacke/test

<server localhost>
        port = 8000
        listen = 127.0.0.1
        docroot = /home/klacke/yaws/yaws/www
```

```
</server>

<server localhost>
        port = 8001
        listen = 127.0.0.1
        docroot = /home/klacke/test
</server>
```

We define two servers, one being the original default and a new pointing to a document root in our home directory.

We can now start to add static content in the form of HTML pages, dynamic content in the form of .yaws pages or ERLANG .beam code that can be used to generate the dynamic content.

The load path will be set so that beam code in the directory `~/test` will be automatically loaded when referenced.

It is best to run YAWS interactively while developing the site. In order to start the YAWS as a daemon, we give the flags:

```
# yaws -D -heart
```

The *-D* flags instructs YAWS to run as a daemon and the *-heart* flags will start a heartbeat program called heart which restarts the daemon if it should crash or if it stops responding to a regular heartbeat.

Once started in daemon mode, we have very limited ways of interacting with the daemon. It is possible to query the daemon using:

```
# yaws -S
```

This command produces a simple printout of Uptime and number of hits for each configured server.

If we change the configuration, we can HUP the daemon using the command:

```
# yaws -h
```

This will force the daemon to reread the configuration file.

Chapter 3

Static content

YAWS acts very much like any regular web server while delivering static pages. By default YAWS will cache static content in RAM. The caching behavior is controlled by a number of global configuration directives. Since the RAM caching occupies memory, it may be interesting to tweak the default values for the caching directives or even to turn it off completely.

The following configuration directives control the caching behavior

- *max_num_cached_files = Integer* YAWS will cache small files such as commonly accessed GIF images in RAM. This directive sets a maximum number on the number of cached files. The default value is 400.
- *max_num_cached_bytes = Integer* This directive controls the total amount of RAM which can maximally be used for cached RAM files. The default value is 1000000, 1 megabyte.
- *max_size_cached_file = Integer*

  This directive sets a maximum size on the files that are RAM cached by YAWS . The default value i 8000, 8 batters.

It may be considered to be confusing, but the numbers specified in the above mentioned cache directives are local to each server. Thus if we have specified `max_num_cached_bytes = 1000000` and have defined 3 servers, we may actually use $3 * 1000000$ bytes.

Chapter 4

Dynamic content

Dynamic content is what YAWS is about. Most web servers are designed with HTTP and static content in mind whereas YAWS is designed for dynamic pages from the start. Most large sites on the Web today make heavy use of dynamic pages.

4.1 Introduction

When the client GETs a a page that has a .yaws suffix. The YAWS server will read that page from the hard disk and divide it in parts that consist of HTML code and ERLANG code. Each chunk of ERLANG code will be compiled into a module. The chunk of ERLANG code must contain a function `out/1`. If it doesn't the YAWS server will insert a proper error message into the generated HTML output.

When the YAWS server ships a .yaws page it will process it chunk by chunk through the .yaws file. If it is HTML code, the server will ship that as is, whereas if it is ERLANG code, the YAWS server will invoke the `out/1` function in that code and insert the output of that `out/1` function into the stream of HTML that is being shipped to the client.

YAWS will (of course) cache the result of the compilation and the next time a client requests the same .yaws page YAWS will be able to invoke the already compiled modules directly.

4.2 EHTML

There are two ways to make the `out/1` function generate HTML output. The first and most easy to understand is by returning a tuple `{html, String}` where `String` then is regular HTML data (possibly as a deep list of strings and/or binaries) which will simply be inserted into the output stream. An example:

```
<html>
```

```
<h1> Example 1 </h1>

<erl>
out(A) ->
    Headers = A#arg.headers,
    {html, io_lib:format("You say that you're running ~p",
                         [Headers#headers.user_agent])}.

</erl>

</html>
```

The second way to generate output is by returning a tuple `{ehtml, EHTML}`. The term `EHTML` must adhere to the following structure:

$EHTML = [EHTML]|\{TAG, Attrs, Body\}|\{TAG, Attrs\}|\{TAG\}|binary()|character()$

$TAG = atom()$

$Attrs = [\{HtmlAttribute, Value\}]$

$HtmlAttribute = atom()$

$Value = string()|atom()$

$Body = EHTML$

We give an example to show what we mean: The tuple

```
{ehtml, {table, [{bgcolor, grey}],
         [
          {tr, [],
           [
            {td, [], "1"},
            {td, [], "2"},
            {td, [], "3"}
           ],
           {tr, [],
            [{td, [{colspan, "3"}], "444"}]}}]}}.
```

Would be expanded into the following HTML code

```
<table bgcolor="grey">
  <tr>
    <td> 1 </td
    <td> 2 </td>
```

```
    <td> 3 </td>
  </tr>
  <tr>
    <td colspan="3"> 444 </td>
  </tr>
</table>
```

At a first glance it may appears as if the HTML code is more beautiful than the ERLANG tuple. That may very well be the case from a purely aesthetic point of view. However the ERLANG code has the advantage of being perfectly indented by editors that have syntax support for ERLANG (read Emacs). Furthermore, the ERLANG code is easier to manipulate from an ERLANG program.

As an example of some more interesting ehtml we could have an `out/1` function that prints some of the HTTP headers.

In the www directory of the YAWS source code distribution we have a file called `arg.yaws`. The file demonstrates the Arg #arg record parameter which is passed to the `out/1` function.

But before we discuss that code, we describe the Arg record in detail.

Here is the `yaws_api.hrl` file which is in included by default in all YAWS files. The #arg record contains many fields that are useful when processing HTTP request dynamically. We have access to basically all the information which associated to the client request such as:

- The actual socket leading back to the HTTP client
- All the HTTP headers - parsed into a #headers record.
- The HTTP request - parsed into a #http_request record
- clidata - Data which is POSTed by the client
- querydata - This is the remainder of the URL following the first occurrence of a ? character - if any.
- docroot - The absolute path to the docroot of the virtual server that is processing the request.

```
-record(arg, {
          clisock,        %% the socket leading to the peer client
          headers,        %% headers
          req,            %% request
          clidata,        %% The client data (as a binary in POST requests)
          querydata,      %% Was the URL on the form of ...?query (GET reqs)
          appmoddata,     %% the remainder of the path leading up to the query
```

```
          docroot,          %% where's the data
          fullpath,         %% full path to yaws file
          cont,             %% Continuation for chunked multipart uploads
          state,            %% State for use by users of the out/1 callback
          pid,              %% pid of the yaws worker process
          opaque            %% useful to pass static data
         }).


-record(http_request, {method,
                       path,
                       version}).


-record(headers, {
          connection,
          accept,
          host,
          if_modified_since,
          if_match,
          if_none_match,
          if_range,
          if_unmodified_since,
          range,
          referer,
          user_agent,
          accept_ranges,
          cookie = [],
          keep_alive,
          content_length,
          content_type,
          authorization,
          other = []    %% misc other headers
         }).
```

There are a number of *advanced* fields in the #arg record such as `appmod`, `opaque` that will be discussed in later chapters.

Now, we show some code which displays the content of the Arg #arg record. The code is available in yaws/www/arg.yaws and after a a `local_install` a request to *http://localhost:8000/arg.yaws* will run the code.

```
<html>

<h2> The Arg </h2>

<p>This page displays the Arg #argument structure
supplied to the out/1 function.

<erl>


out(A) ->
    Req = A#arg.req,
    H = yaws_api:reformat_header(A#arg.headers),
    {ehtml,
     [{h4,[], "The headers passed to us were:"},
      {hr},
      {ol, [],lists:map(fun(S) -> {li,[], {p,[],S}} end,H)},

      {h4, [], "The request"},
      {ul,[],
       [{li,[], f("method: ~s",  [Req#http_request.method])},
        {li,[], f("path: ~p",    [Req#http_request.path])},
        {li,[], f("version: ~p", [Req#http_request.version])}]},

      {hr},
      {h4, [], "Other items"},
      {ul,[],
       [{li,[], f("clisock from: ~p", [inet:peername(A#arg.clisock)])},
        {li,[], f("docroot: ~s",      [A#arg.docroot])},
        {li,[], f("fullpath: ~s",     [A#arg.fullpath])}]},
      {hr},
      {h4, [], "Parsed query data"},
      {pre,[], f("~p", [yaws_api:parse_query(A)])},
      {hr},
      {h4,[], "Parsed POST data "},
      {pre,[],  f("~p", [yaws_api:parse_post(A)])}]}.

</erl>

</html>
```

The code utilizes 4 functions from the `yaws_api` module. `yaws_api` is a general purpose www api module that contains various functions that are handy while developing YAWS code. We will see many more of those functions during the examples in the following chapters.

The functions used are:

- `yaws_api:f/2` alias for io_lib:format/2. The `f/1` function is automatically `-includeed` in all YAWS code.
- `yaws_api:reformat_header/1` - This function takes the #headers record and unparses it, that is reproduces regular text.
- `yaws_api:parse_query/1` - The topic of next section.
- `yaws_api:parse_post/1` – Ditto.

4.3 POSTs

4.3.1 Queries

The user can supply data to the server in many ways. The most common is to give the data in the actual URL. If we invoke:

```
GET http://localhost:8000/arg.yaws?kalle=duck&goofy=unknown
```

we pass two parameters to the *arg.yaws* page. That data is URL-encoded by the browser and the server can retrieve the data by looking at the remainder of the URL following the ? character. If we invoke the `arg.yaws` page with the above mentioned URL we get as the result of `yaws_parse_query/1`:

$kalle = duck$

$goofy = unknown$

In ERLANG terminology, the call `yaws_api:parse_query(Arg)` returns the list:

```
[{kalle, "duck"}, {goofy, "unknown"}]
```

Note that the first element is transformed into an atom, whereas the value is still a string.

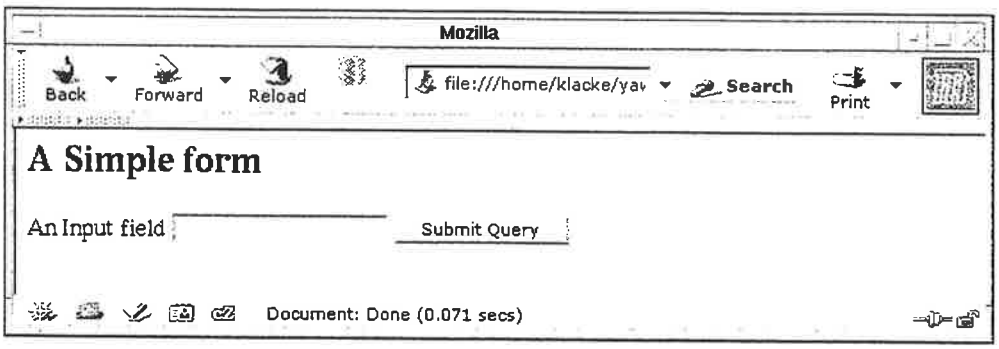
hence, a web page can contain URLs with a query and thus pass data to the web server. This scheme works both with GET and POST requests. It is the easiest way to pass data to the Web server since no FORM is required in the web page.

4.3.2 Forms

In order to POST data a FORM is required, say that we have a page called `form.yaws` that contain the following code:

```
<html>
<form action="/post_form.yaws"
      method="post"

<p> A Input field
<input name="xyz" type="text">
<input type="submit">
</form>
</html>
```

This will produce a page with a simple input field and a Submit button.



If we enter something - say "Hello there " - in the input field and click the Submit button the client will request the page indicated in the "action" attribute, namely `post_form.yaws`.

If that YAWS page has the following code:

```
out(A) ->
   L = yaws_api:parse_post(A),
   {html, f("~p", [L])}
```

The user will see the output

```
[{xyz, "Hello there"}]
```

The differences between using the query part of the URL and a form are the following:

- Using the query arg only works in GET request. We parse the query argument with the function `yaws_api:parse_query(Arg)`
- If we use a form and POST the user data the client will transmit the user data in the body of the request. That is - the client sends a request to get the page using the POST method and it then attaches the user data - encoded - into the body of the request.

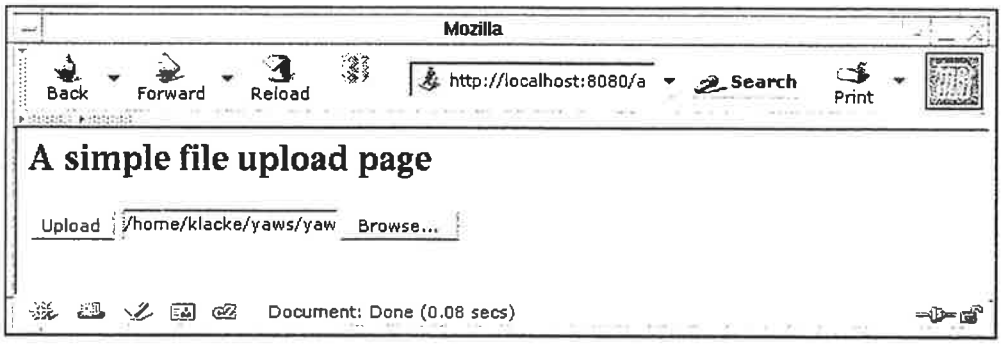
  A POST request can have a query part in its URL as well as user data in the body.

4.4 POSTing files

It is possible to upload files from the client to the server by means of POST. We indicate this in the form by telling the browser that we want a different encoding, here is a form that does this:

```
out(A) ->
    Form =
        {form, [{enctype, "multipart/form-data"},
                {method, post},
                {action, "file_upload_form.yaws"}],
                [{input, [{type, submit}, {value, "Upload"}]},
                 {input, [{type,file}, {width, "50"}, {name, foo}]}]},
    {ehtml, {html,[], [{h2,[], "A simple file upload page"},
                       Form]}}.
```

The page delivers the entire HTML page with enclosing `html` markers. It looks like:



The user get an option to browse the local host for a file or the user can explicitly fill in the file name in the input field. The file browsing part is automatically taken care of by the browser.

The action field in the form states that the client shall POST to a page called `file_upload_form.yaws`. This page will get the contents of the file in the body of the POST message. Here we have one easy

case and one hard case. YAWS will read the data from the client. However if the file is large the entire contents of the file will not be part of the read operation. It is not acceptable to let YAWS continue to read the full POST body and then when that is done, invoke the POST page. YAWS must feed the page with the chunks of the file as they arrive.

First the easy case:

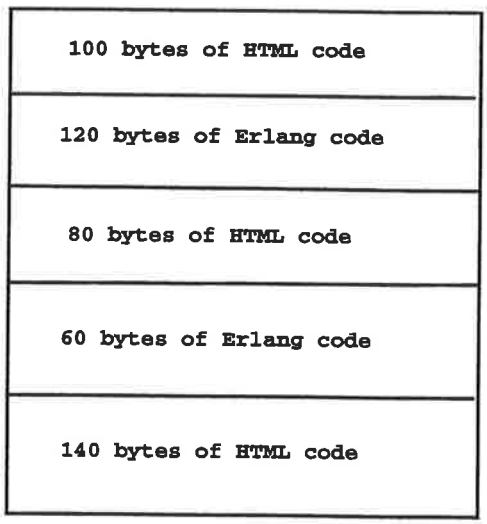
Not YET Written ... . fill this in later .....

Chapter 5

Mode of operation

5.1 On the fly compilation

When the client requests a YAWS page, YAWS will look in its caches (there is one cache per virtual server) to see if it finds the requested page in the cache. If YAWS doesn't find the page in the cache, it will compile the page. This only happens the first time a page is requested. Say that the page is 400 bytes big has the following layout:



The YAWS server will then parse the file and produce a structure which makes it possible to deliver the page in a readily fashion the next time the same page is requested.

When shipping the page it will

1. Ship the first 100 bytes from the file

2. Evaluate the first ERLANG chunk in the file and ship the output from the `out/1` function in that chunk. It will also jump ahead in the file and skip 120 bytes.
3. Ship 80 bytes of HTML code
4. Again evaluate an ERLANG chunk, this time the second and jump ahead 60 bytes in the file.
5. And finally ship 140 bytes of HTML code to the client

YAWS writes the source output of the compilation into a directory /tmp/yaws/$UID. The beam files are never written to a file. Sometimes it can be useful to look at the generated source code files, for example if the YAWS /ERLANG code contains a compilation error which is hard to understand.

5.2 Evaluating the YAWS code

All client requests will execute in their own ERLANG process. For each group of virtual hosts on the same IP:PORT pair one ERLANG process listens for incoming requests.

This process spawns acceptor processes for each incoming request. Each acceptor process reads and parses all the HTTP headers from the client. It then looks at the Host: header to figure out which virtual server to use, i.e. which docroot to use for this particular request. If the Host: header doesn't match any server from *yaws.conf* with that IP:PORT pair, the first one from *yaws.conf* is chosen.

By default YAWS will not ship any data at all to the client while evaluating a YAWS page. The headers as well as the generated content are accumulated and not shipped to the client until the entire page has been processed.

Chapter 6

SSL

SSL - Secure Socket Layer is a protocol used on the Web for delivering encrypted pages to the WWW client. SSL is widely deployed on the Internet and virtually all bank transactions as well as all on-line shopping today is done with SSL encryption. There are many good sources on the net that describes SSL in detail - and I will not try to do that here. There is for example a good document at: `http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/` which describes how to manage certificates and keys.

In order to run an SSL server we must have a certificate. Either we can create a so called self-signed certificate ourselves or buy a certificate from one of the many CA's (Certificate Authority's) on the net. YAWS use the otp interface to openssl.

To setup a YAWS server with SSL we could have a *yaws.conf* file that looks like:

```
 logdir = /var/log/yaws

<server www.funky.org>
              port = 443
              listen = 192.168.128.32
              docroot = /var/yaws/www.funky.org
              <ssl>
                 keyfile = /etc/funky.key
                 certfile = /etc/funky.cert
                 password = gazonk
              </ssl>
      </server>
```

This is the easiest possible SSL configuration. The configuration refers to a certificate file and a key file. The certificate file must contain the name "www.funky.org" as it "Common Name".

The keyfile is the private key file and it is encrypted using the password "gazonk".

Chapter 7

Applications

YAWS is well suited for Web applications. In this chapter we will describe a number of application templates. Code and strategies that can be used to build Web applications.

There are several ways of starting applications from YAWS .

- The first and most easy variant is to specify the `-r Module` flag to the YAWS startup script. This will `apply(Module,start,[])`
- We can also specify runmods in the *yaws.conf* file. It is possible to have several modules specified if want the same YAWS server to run several different applications.

```
runmod = myapp
runmod = app_number2
```

- It is also possible to do it the other way around, let the main application start YAWS . We call this embedded mode and that will be discussed in a later chapter,

7.1 Login scenarios

Many Web applications require the user to login. Once the user has logged in the server sets a Cookie and then the user will be identified by help of the cookie in subsequent requests.

7.1.1 The session server

The cookie is passed in the headers and is available to the YAWS programmer in the Arg #arg record. The YAWS session server can help us to maintain a state for a user while the user is logged in to the application. The session server has the following 5 api functions to aid us:

1. `yaws_api:new_cookie_session(Opaque)` This function initiates a new cookie based session. The Opaque data is typically some application specific structure which makes it possible for the application to read a user state, or it can be the actual user state itself.

2. `yaws_api:cookieval_to_opaque(Cookie)` This function maps a cookie to a session.

3. `yaws_api:replace_cookie_session(Cookie, NewOpaque)` Replace the Opaque user state in the session server.

4. `yaws_api:delete_cookie_session(Cookie)` This function should typically be called when the user logs out or when our web application decides to auto logout the user.

All cookie based applications are different but they have some things in common. In the example that follow we assume the existence of a function `myapp:auth(UserName, Passwd)` and it returns `ok` or `{error, Reason}`

Furthermore - let's have a record:

```
-record(session, {user,
                  passwd,
                  udata = []}).
```

The following function is a good template function to check the cookie.

```
get_cookie_val(CookieName, Arg) ->
    H = Arg#arg.headers,
    yaws_api:find_cookie_val(CookieName, H#headers.cookie).


check_cookie(A, CookieName) ->
    case get_cookie_val(CookieName, A) of
        []  ->
            {error, "not logged in"};
        Cookie ->
            yaws_api:cookieval_to_opaque(Cookie)
    end.
```

So what we need to do is the following: We want to check all requests and make sure the the session_server has our cookie registered as an active session.

If a request comes in without a working cookie we want to present a login page instead of the page the user requested.

Another quirky issue is that the pages necessary for display of the login page must be shipped without checking the cookie.

7.1.2 Arg rewrite

In this section we describe a feature whereby the user is allowed to rewrite the Arg at an early stage in the YAWS server. We do that by specifying an `arg_rewrite_mod` in the *yaws.conf* file.

```
arg_rewrite_mod = myapp
```

Then in the `myapp` module we have:

```
arg_rewrite(Arg) ->
    OurCookieName = "myapp_sid"
    case check_cookie(A, OurCookieName) of
        {error, _} ->
            do_rewrite(Arg);
        {ok, _Session} ->
            %return Arg untouched
            Arg
    end.

%% these pages must be shippable without a good cookie
login_pages() ->
    ["/banner.gif", "/login.yaws", "/post_login.yaws"].

do_rewrite(Arg) ->
    Req = Arg#arg.req,
    {abs_path, Path} = Req#http_request.path,
    case lists:member(Path, login_pages()) of
        true ->
            Arg;
        false ->
            Arg#arg{req = Req#http_request{path = {abs_path, "/login.yaws"}},
                    state =  {abs_path, Path}}
    end.
```

Our arg rewrite function lets all Args go through untouched that either have a good cookie or belong to a set of predefined pages that are acceptable to get without being logged in. If we decode that

the user must log in, we change the path of the request, thereby making the YAWS server ship a login page instead of the page the user requested. We also set the original path in the Arg state argument so that the login page can redirect the user to the original page - once the login procedure is finished.

7.1.3 Authenticating

Now we're approaching the `login.yaws` page, the page that displays the login prompt to the user. The login page consists of two parts, one part that displays the login data as a form and one form processing page that reads the data the user entered in the login fields and performs the actual authentication.

The login page performs a tiny well known Web trick where it passes the original URL request in a hidden field in the login page and thereby passing that information to the form processing page.

The page `login.yaws`:

```
<erl>

out(A) ->
    {ehtml,
     {html,[],
      [{h2, [], "Login page"},
       {hr},
       {form, [{action,"/login_post.yaws"},
               {method,post}],

        [{p,[], "Username"}, {input, [{type,text},{name,uname}]},
         {p,[],"Password"},  {input, [{type,password},{name,passwd}]},
         {input, [{type,submit},{value,"Login"}]},
         {input, [{type,hidden},{name,url},
                  {value, A#arg.state}]}]}]}}.

</erl>
```

The form processing page which gets the POST data from the code above looks like:

```
<erl>

-include("myapp.hrl").
%% we have the session record there
%% we must set the include_path in the yaws.conf file
```

```
%% in order for the compiler to find that file

kv(K,L) ->
    {value, {K, V}} = lists:keysearch(K,1,L),
    V.

out(A) ->
    L = yaws_api:parse_post(A),
    User = kv(user, L),
    Pwd =  kv(passwd, L),
    case myapp:auth(User, Pwd) of
        ok ->
            S = #session{user = User,
                         passwd = Pwd,
                         udata = []},
            %% Now register the session to the session server
            Cookie = yaws_api:new_cookie_session(S),
            [{redirect_local, kv(url, L)},
              yaws_api:setcookie("myapp_sid",Cookie)]
        Err ->
            {ehtml,
             {html, [],
              {p, [], f("Bad login: ~p",[Err])}}}
    end.

</erl>
```

The function returns a list of two new previously not discussed return values: Instead of returning HTML output as in `{html, Str}` or `{ehtml,Term}` we return a list of two new values. There are many different possible return values from the `out/1` function and they will all be described later.

1. The tuple `{redirect_local, Path}`. This particular redirect return value will make the YAWS web server return a 303 redirect to the specified Path.
2. `yaws_api:setcookie("myapp_sid",Cookie)` generates a `Set-Cookie` header

Now if we put all this together we have a full blown cookie based login system. The last thing we did in the form processing code was to register the session with the session server thereby letting any future requests go straight through the `Arg` rewriter.

This way both YAWS pages as well as all or some static content is protected by the cookie login code.

7.1.4 Database driven applications

We can use code similar to the code in the previous section to associate a user session to entries in a database. Mneisa fits perfectly together with YAWS and keeping user persistent state in Mnesia is both easy and convenient.

Once the user has logged in we can typically use the user name as key into the database. We can mix ram_tables and disc_tables to our liking. The Mneisa database must be initialized by means of `create_table/2` before it can be used. This is typically done while installing the web application on a machine.

Another option is to let the application check that Mnesia is initialized whenever the application starts.

If we don't want or need to use Mnesia, it's of course possible to use a simple `dets` file or a text file as well.

7.2 Appmods

Appmods is mechanism to invoke different applications based upon the URL. A URL - as presented to the web server in a request - has a path part and a query part.

It is possible to install several appmods in the *yaws.conf* file as:

```
appmods = foo myapp
```

Now, if the user requests a URL where any component in the directory path is an appmod, the parsing of the URL will terminate there and instead of reading the actual file from the disk, YAWS will invoke the appmod with the remainder of the path inserted into `Arg#arg.appmoddata`.

Say the user requests the URL *http://www.funky.org/myapp/xx/bar.html* YAWS will not ship the file `bar.html` to the client, instead it will invoke `myapp:out(Arg)` with `Arg#arg.appmoddata` set to the string `xx/bar.html`. Any optional query data - that is data that follows the first "?" character in the URL - is removed from the path and passed as `Arg#arg.querydata`.

Appmods can be used to run applications on a server. All requests to the server that has an appmod in the URL will be handled by that application. If the application decides that it want to ship a page from the disk to the client, it can return the tuple `{page, Path}`. This return value will make YAWS read the page from the disk, possibly add the page to it's cache of commonly accessed pages and ship it back to the client.

The `{page, Path}` return value is equivalent to a redirect, but it removes an extra round trip - and is thus faster.

Appmods can also be used to fake entire directory hierarchies that doesn't exists on the disk.

7.3 The opaque data

Sometimes an application needs application specific data such as the location of its data files or whatever. There exists a mechanism to pass application specific configuration data from the YAWS server to the application.

When configuring a server we have an opaque field in the configuration file that can be used for this purpose. Say that we have the following fields in the config file:

```
<server foo>
    listen = 192.168.128.44
    <opaque>
        foo = bar
        somefile = /var/myapp/db
        myname = hyber
    </opaque>
</server>
```

This will create a normal server that listens to the specified IP address. An application has access to the opaque data that was specified in that particular server through `Arg#arg.opaque`

If we have the opaque data specified above, the Arg opaque field will have the value:

```
[{foo, "bar"},
 {somefile, "/var/myapp/db"},
 {myname, "hyber"}
]
```

7.4 Customizations

When actually deploying an application at a live site, some of the standard YAWS behaviors are not acceptable. Many sites want to customize the web server behavior when a client requests a page that doesn't exists on the web server. The standard YAWS behavior is to reply with status code 404 and a message explaining that the page doesn't exist.

Similarly, when YAWS code crashes, the Reason for the crash is displayed in the Web browser. This is very convenient while developing a sit but not acceptable in production.

7.4.1 404 File not found

We can install a special handler for 404 messages. We do that by specifying a `errormod_404` in the *yaws.conf* file.

If we have:

```
<server foo>
  ..
  ..
  ..
  errormod_404 = myapp

</server>
```

When YAWS gets a request for a file that doesn't exists on the hard disk, it invokes the errormod_404 module to generate both the status code as well as the content of the message.

`Module:out404(Arg, GC, SC)` will be invoked by YAWS . The arguments are

- Arg is a #arg record
- GC is a #gconf record (defined in yaws.hrl)
- SC is a #sconf record (defined in yaws.hrl)

The function can and must do the same things that a normal `out/1` does.

7.4.2 Crash messages

We use a similar technique for generating the crash messages, we install a module in the *yaws.conf* file and let that module generate the crash message. We have:

```
errormod_crash = Module
```

The default is to display the entire formated crash message in the browser. This is good for debugging but not in production.

The function `Module:crashmsg(Arg, SC, Str)` will be called. The Str is the real crash message formated as a string.

7.5 Stream content

If the `out/1` function returns the tuple `{content, MimeType, Content}` YAWS will ship that data to the Client. This way we can deliver dynamically generated content to the client which is of a different mime type than "text/html".

If the generated file is very large and it not possible to generate the entire file, we can return the value: `{streamcontent, MimeType, FirstChunk}` and then from a different ERLANG process deliver the remaining chunks by using the functions:

1. `yaws_api:stream_chunk_deliver(YawsPid, Data)` where the `YawsPid` is the process id of the YAWS worker process. That pid is available in `Arg#arg.pid`
2. `stream_chunk_end(YawsPid)` This function must be called to indicate the end of the stream.

7.6 All out/1 return values

- `{html, DeepList}` This assumes that DeepList is formatted HTML code. The code will be inserted in the page.
- `{ehtml, Term}` This will transform the ERLANG term Term into a stream of HTML content.
- `{content, MimeType, Content}` This function will make the web server generate different content than HTML. This return value is only allowed in a YAWS file which has only one <erl> </erl> part and no html parts at all.
- `{streamcontent, MimeType, FirstChunk}` This return value plays the same role as the con tent return value above. However it makes it possible to stream data to the client if the YAWS code doesn't have access to all the data in one go. (Typically if a file is very large or if data arrives from back end servers on the network.
- `{header, H}` Accumulates a HTTP header. Used by for example the `yaws_api:setcookie/2-6` function.
- `{allheaders, HeaderList}` Will clear all previously accumulated headers and replace them.
- `{status, Code}` Will set another HTTP status code than 200.
- `break` Will stop processing of any consecutive chunks of erl or html code in the YAWS file.
- `ok` Do nothing.
- `{redirect, Url}` Erase all previous headers and accumulate a single Location header. Set the status code.
- `{redirect_local, Path}` Does a redirect to the same Scheme://Host:Port/Path as we currently are executing in.

- `{get_more, Cont, State}` When we are receiving large POSTs we can return this value and be invoked again when more Data arrives.
- `[ListOfValues]` It is possible to return a list of the above defined return values.

Chapter 8

Debugging and Development

YAWS has excellent debugging capabilities. First and foremost we have the ability to run the web server in interactive mode by means of the command line switch `-i`

This gives us a regular ERLANG command line prompt and we can use that prompt to compile helper code or reload helper code. Furthermore all error messages are displayed there. If a .yaws page producees any regular ERLANG io, that output will be displayed at the ERLANG prompt - assuming that we are running in interactive mode.

If we give the command line switch `-d` we get some additional error messages. Also YAWS does some additional checking of user supplied data such as headers.

8.1 Logs

YAWS produces various logs. All log files are written into the YAWS logdir directory. This directory is specified in the config file.

We have the following log files:

- The access log. Access logging is turn on or off per server in the *yaws.conf* file. If access_log is turned on for a server, YAWS will produce a log in Common Access Log Format called *HostName:PortNumber.access*
- *report.log* This file contains all error and crash messages for all virtual servers in the same file.
- *trace.traffic* and *trace.http* The two command line flags `-t` and `-T` tells YAWS to trace all traffic or just all HTTP messages and write them to a file.

Chapter 9

Security

YAWS is of course susceptible to intrusions. YAWS has no (yet) abilities to run under a different user than root - Assuming we need to listen to privileged port numbers. Running as root is generally a bad idea.

Intrusions can happen basically at all places in YAWS code where the YAWS code calls either the BIF `open_port` or when YAWS code does calls to `os:cmd/1`.

Both `open_port` and `os:cmd/1` invoke the `/bin/sh` interpreter to execute its commands. If the commands are nastily crafted bad things can easily happen.

All data that is passed to these two function must be carefully checked.

Since YAWS is written in ERLANG a large class of cracks are eliminated since it is not possible to perform any buffer overrun cracks on a YAWS server. This is very good.

Another possible point of entry to the system is by providing a URL which takes the client out from the docroot. This should not be possible - and the impossibility relies on the correctness of the URL parsing code in YAWS .

9.1 WWW Authenticate

YAWS has support for WWW authenticate protected directories. The access rights to different directories is controlled by directives in the *yaws.conf* file.

We can specify several auth groups in a server configuration. If we have the following in the *yaws.conf* file:

```
<server foo>
..
..
```

```
  <auth>
    realm = secretpage
    dir = /var/yaws/www/protected
    user klacke:gazonk
    user jonny:xyz
    user ronny:12r8uyp09jksfdge4
  </auth>
</server>
```

YAWS will protect all files in the specified directory by means of WWW-Authenticate access. If a user requests a page in the directory, and doesn't have the correct WWW-Authenticate header, YAWS will reply with a proper status code that makes the browser pop up a login window.

Chapter 10

Embedded mode

YAWS is a normal OTP application. It is possible to integrate YAWS into another - larger - application. The YAWS source tree must be integrated into the larger applications build environment. YAWS is then simply started by `application:start()` from the larger applications boot script.

By default YAWS reads its configuration data from a config file, the default is "/etc/yaws.conf". If YAWS is integrated into a larger application that application typically has its configuration data kept at some other centralized place. Sometimes we may not even have a file system to read the configuration from if we run a small embedded system.

YAWS reads its application environment. If the environment key `embedded` is set to `ttrue`, YAWS starts in embedded mode. Once started it must be fed a configuration, and that can be done after YAWS has started by means of the function `yaws_api:setconf/2`.

It is possible to call `setconf/2` several times to force YAWS to reread the configuration.

Chapter 11

The config file - yaws.conf

In this section we provide a complete listing of all possible configuration file options. The configuration contains two distinct parts a global part which affects all the virtual hosts and a server part where options for each virtual host is supplied.

11.1 Global Part

- `dir = Directory` - All YAWS logs will be written to files in this directory. There are several different log files written by YAWS .
  - report.log - this is a text file that contains all error logger printouts from YAWS .
  - Host.access - for each virtual host served by YAWS , a file Host.access will be written which contains an access log in Common Log Format.
  - trace.http - this file contains the HTTP trace if that is enabled
  - trace.traffic - this file contains the traffic trace if that is enabled
- `ebin_dir = Directory` - This directive adds Directory to the ERLANG search path. It is possible to have several of these command in the configuration file.
- `include_dir = Directory` - This directive adds Directory to the path of directories where the ERLANG compiler searches for include files. We need to use this if we want to include .hrl files in our YAWS ERLANG code.
- `max_num_cached_files = Integer` - YAWS will cache small files such as commonly accessed GIF images in RAM. This directive sets a maximum number on the number of cached files. The default value is 400.
- `max_num_cached_bytes = Integer` - This directive controls the total amount of RAM which can maximally be used for cached RAM files. The default value is 1000000, 1 megabyte.

- `max_size_cached_file = Integer` - This directive sets a maximum size on the files that are RAM cached by YAWS . The default value i 8000, 8 kBytes.
- `cache_refresh_secs = Integer` The RAM cache is used to serve pages that sit in the cache. An entry sits in cache at most cache_refresh_secs number of seconds. The default is 30. This means that when the content is updated under the docroot, that change doesn't show until 30 seconds have passed. While developing a YAWS site, it may be convenient to set this value to 0. If the debug flag (-d) is passed to the YAWS start script, this value is automatically set to 0.
- `trace = traffic | http` - This enables traffic or http tracing. Tracing is also possible to enable with a command line flag to YAWS .

11.2 Server Part

YAWS can virthost several web servers on the same IP address as well as several web servers on different IP addresses. The on limitation here is that there can be only one server with ssl enabled per each individual IP address. Each virtual host is defined within a matching pair of `<server ServerName>` and `</server>`. The ServerName will be the name of the web server.

The following directives are allowed inside a server definition.

- `port = Port` - This makes the server listen on Port
- `listen = IpAddress` - This makes the server listen on IpAddress When virthosting several servers on the same IP/port address, if the browser doesn't send a Host: field, YAWS will pick the first server specified in the config file
- `rport = Port` This forces all local redirects issued by the server to go to Port. This is useful when YAWS listens to a port which is different from the port that the user connects to. For example, running YAWS as a non-privileged user makes it impossible to listen to port 80, since that port can only be opened by a privileged user. Instead YAWS listens to a high port number port, 8000, and iptables are used to redirect traffic to port 80 to port 8000 (most NAT:ing firewalls will also do this for you).
- `rscheme = http | https` This forces all local redirects issued by the server to use this method. This is useful when an SSL off-loader, or stunnel, is used in front of YAWS .
- `access_log = true | false` Setting this directive to false turns of traffic logging for this virtual server. The default value is true.
- `docroot = Directory` - This makes the server serve all its content from Directory
- `partial_post_size = Integer` - When a YAWS file receives large POSTs, the amount of data received in each chunk is determined by the this parameter. The default value is 10240.

- `tilde_expand = true|false` - If this value is set to false YAWS will never do tilde expansion. The default is true. tilde_expansion is the mechanism whereby a URL on the form http://www.foo.com/ username is changed into a request where the docroot for that particular request is set to the directory `~username/public_html/` The default value is true.

- `appmods = [ListOfModuleNames]` - If any the names in ListOfModuleNames appear as components in the path for a request, the path request parsing will terminate and that module will be called.

  Assume for example that we have the URL http://www.hyber.org/myapp/foo/bar/baz?user=joe while we have the module foo defined as an appmod, the function foo:out(Arg) will be invoked instead of searching the file systems below the point foo.

  The Arg argument will have the missing path part supplied in its appmoddata field.

- `errormod_404 = Module` - It is possible to set a special module that handles 404 Not Found messages.

  The function Module:out404(Arg, GC, SC) will be invoked. The arguments are

  Arg is a arg record

  GC is a gconf record (defined in yaws.hrl)

  SC is a sconf record (defined in yaws.hrl)

  The function can and must do the same things that a normal `out/1` does.

- `errormod_crash = Module` - It is possible to set a special module that handles the HTML generation of server crash messages. The default is to display the entire formated crash message in the browser. This is good for debugging but not in production.

  The function Module:crashmsg(Arg, SC, Str) will be called. The Str is the real crash message formated as a string.

- `arg_rewrite_mod = Module` - It is possible to install a module that rewrites all the Arg arg records at an early stage in the YAWS server. This can be used to do various things such as checking a cookie, rewriting paths etc.

- `<ssl> .... </ssl>` This begins and ends an SSL configuration for this server.

  - `keyfile = File` - Specifies which file contains the private key for the certificate.
  - `certfile = File` - Specifies which file contains the certificate for the server.
  - `cacertfile = File` File If the server is setup to require client certificates. This file needs to contain all the certificates of the acceptable signers for the client certs.
  - `verify = 1 | 2 | 3` Specifies the level of verification the server does on client certs. 1 means nothing , 2 means the the server will ask the client for a cert but not fail if the client doesn't supply a client cert, 3 means that the server requires the client to supply a client cert.

- `depth = Int` Specifies the depth of certificate chains the server is prepared to follow when verifying client certs.
- `password = String` - String If the private key is encrypted on disk, this password is the 3des key to decrypt it.
- `cciphers = String` This string specifies the ssl cipher string. The syntax of the ssl cipher string is a little horrible sub language of its own. It is documented in the ssl man page for "ciphers".
- `</ssl>` Ends an SSL definition

- `<auth> ... </auth>` Defines an auth structure. The following items are allowed within a matching pair of <auth> and </auth> delimiters.

  - `dir = Dir` Makes Dir to be controlled bu WWW-authenticate headers. In order for a user to have access to WWW-Authenticate controlled directory, the user must supply a password.
  - `realm = Realm` In the directory defined here, the WWW-Authenticate Realm is set to this value.
  - `user = User:Password` Inside this directory, the user User has access if the user supplies the password Password in the pop up dialog presented by the browser. We can obviously have several of these value inside a single <auth> </auth> pair.
  - `</auth>` Ends an auth definition

11.3 Configuration Examples

The following example defines a single server on port 80.

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
</server>
```

And this example shows a similar setup but two web servers on the same IP address

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
```

```
        docroot = /var/yaws/www
</server>

<server www.funky.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www_funky_org
</server>
```

An example with www-authenticate and no access logging at all.

```
logdir = /var/log/yaws
<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
        access_log = false
        <auth>
            dir = /var/yaws/www/secret
            realm = foobar
            user = jonny:verysecretpwd
            user = benny:thequestion
            user = ronny:havinganamethatendswithy
       </auth>

</server>
```

And finally a slightly more complex example with two servers on the same IP, and one ssl server on a different IP.

```
logdir = /var/log/yaws
max_num_cached_files = 8000
max_num_cached_bytes = 6000000

<server www.mydomain.org>
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www
</server>

<server www.funky.org>
```

```
        port = 80
        listen = 192.168.128.31
        docroot = /var/yaws/www_funky_org
</server>

<server www.funky.org>
        port = 443
        listen = 192.168.128.32
        docroot = /var/yaws/www_funky_org
        <ssl>
           keyfile = /etc/funky.key
           certfile = /etc/funky.cert
           password = gazonk
        </ssl>
</server>
```

IMPLEMENTING THE MOBILE LOCATION PROTOCOL: A TALE FROM THE TRENCHES

Thomas Lindgren Fredrik Linder
Magnus Eklund
Cellpoint AB
e-mail: *first.last*@cellpoint.com

Abstract

We describe the first commercial implementation of the Mobile Location Protocol version 3.0.0. The complexities of finishing the development of a feature-rich protocol in a tight time frame led us to an approach of rapid redevelopment, exemplified below.

INTRODUCTION

Location services are and will remain a distinctive feature of wireless data services in 2G and 3G applications. No dominant standard to access location information has emerged until now: the Location Interoperability Forum, LIF, has recently released version 3.0.0 of the Mobile Location Protocol, MLP. Cellpoint is an active LIF participant, and has developed an implementation of MLP 3.0.0 for use in its mobile location platform. This is the first generally available MLP implementation.

We give an overview of Cellpoint's implementation of MLP and how it fits to the mobile location platform, CMLP, developed by Cellpoint. We conclude with a discussion on development issues.

BACKGROUND

Location services are implemented by interrogating the GSM network on the position of subscribers (or rather, their mobile terminals). This is done by a location server, which provides the service to a collection of clients. A client can be, e.g., a portal deployed by an operator, or a third-party application. Cellpoint has two main products: the mobile location server, MLS, and the mobile location broker, MLB. They are also known as the Cellpoint Mobile Location Platform, or CMLP. The task of the MLS is to provide location information. The MLB acts as an aggregator of location requests and location information by acting as a proxy for multiple MLS:s.

Upon receiving a request, the MLS interrogates the GSM network, using one of several available methods (e.g., cell ID, enhanced cell ID, assisted GPS, or others) [3GPP]. The method used depends on the requested quality of service: some application users require a quick answer, others a precise one. If the subscriber is roaming into another mobile network, the MLS contacts the counterparty MLS, a process known as MLS roaming. For the purpose of implementation, the MLS thus has to act as both client and server.

The MLB has a slightly different role. Each MLB knows of a collection of MLS nodes. When receiving a request, the MLB is responsible for dispatching the request

to the right MLS. (This is similar to MLS roaming, but normally occurs inside an operator's network.) The MLB also handles issues such as security, subscriber anonymity, and subscriber privacy, leaving the MLS nodes to act with the MLB as a trusted counterparty. Subscriber anonymity is used to hide the subscriber identity from third parties. Subscriber privacy is used to check whether the subscriber permits the positioning operation.

THE PLATFORM

Each of the CMLP products runs on Solaris and is primarily written in Erlang [2] with supporting code in C and scripting languages. A number of software packages are sourced in to provide the configuration that operators require, e.g., an Oracle database for subscriber information, and Veritas redundant disk software.

When booting, the system first starts an internally developed system for clustering. The clustering system handles failover very quickly (on the order of a few seconds) and provides redundancy for disks [4]. As part of this process, a collection of Erlang nodes are started. When all cluster nodes are operational, the Erlang service software is started as a collection of processes, aka components, that implement the desired functionality. Examples of components are protocol service frontends, subscriber management, database management, the charging subsystem, a WWW-based GUI, the positioning software and the SS7 network interfaces. When all required components are available, the system is ready for service.

THE MLP PROTOCOL

MLP is a complex protocol based on HTTP/1.1 and XML, which has evolved rapidly during the spring of 2002. (As an example, the February 2002 book by Hjelm [6] uses a syntax which is now obsolete.) MLP is being standardized by a committee of operators and equipment providers known as the Location Interoperability Forum, or LIF. At the time of writing, the MLP 3.0.0 standardization process is being finalized [1].

The basic capability is to serve a location request. The client sends a request as follows:

```
<?xml version ="1.0" ?>
<!DOCTYPE svc_init SYSTEM "MLP_SVC_INIT_300.DTD">
<svc_init ver="3.0.0">
  <hdr ver="3.0.0">
    <client>
      <id>application_4</id>
      <pwd>secret</pwd>
    </client>
  </hdr>
  <slir ver="3.0.0">
    <msids>
      <msid type="MSISDN">46702711038</msid>
    </msids>
    <geo_info>
      <CoordinateReferenceSystem>
        <Identifier>
          <code>4326</code>
          <codeSpace>www.epsg.org</codeSpace>
          <edition>6.1</edition>
```

```
        </Identifier>
      </CoordinateReferenceSystem>
    </geo_info>
  </slir>
</svc_init>
```

The client sends its identity, password and service ID for authorization. The client can also use a `<subclients>` tag, much as a HTTP proxy, to tell which location servers have participated in the request. Several applications, portals, MLBs and MLSs may be involved in a request, generally for administrative reasons, and must be tracked for correct billing. The subscribers to be positioned are then identified with `<msid>` tags, in this example a single ID of type MSISDN, indicating an ordinary cell phone number, followed by information on the desired quality-of-position (accuracy, timeliness and so on) and the coordinate system to use. Many parameters are optional; in the example, no explicit quality-of-position is given. The coordinate system tells the server how to format the response. Client and server agree on coordinate system by consulting a database maintained by the European Petroleum Survey Group, EPSG, which is the final authority [5].

The server returns a response, e.g., as below.

```
<?xml version ="1.0" ?>
<!DOCTYPE svc_result SYSTEM "MLP_SVC_RESULT_300.DTD">
<svc_result ver="3.0.0">
  <slia ver="3.0.0">
    <pos>
      <msid>46702711038</msid>
      <pd>
        <time utc_off="+0200">20020623134453</time>
        <shape>
          <CircularArea srsName="www.epsg.org#4326">
            <coord>
              <X>20 30 5.4W</X>
              <Y>0 0 3.5N</Y>
            </coord>
            <radius>570</radius>
          </CircularArea>
        </shape>
      </pd>
    </pos>
  </slia>
</svc_result>
```

The response is a position for the indicated MSID, in the shape of a circle centered at the specified coordinate with the given radius. The position was acquired at the time 13:44.53 on 2002-06-23 in time zone +0200.

If there was an error in the request, or the request could not be fulfilled, an error response is returned instead.

This basic functionality is extended by several extra features.

- There is a class of emergency requests with a slightly different format, which are treated with different priority, privacy settings, and so forth by the location server.
- Client-assisted positioning permits the client to send GSM information to another node. This is used to ask a server to map a cell ID (enhanced with, e.g., timing advance information provided by the client) to an actual position.

- Triggering and zoning permits the client to register an interest in a certain subscriber (pending the approval of the subscriber, known as subscriber privacy). When some event occurs, e.g., the subscriber moves into or out of a certain area, the location server contacts the client with this information, inverting the client-server relationship.

THE SIMPL2 PROTOCOL

Before the current product release, Cellpoint used a small, proprietary protocol named SIMPL for positioning requests. Cellpoint decided to use a subset of MLP, extended with some Cellpoint-specific features, as the next version of SIMPL, named SIMPL2.

Cellpoint's SIMPL2 protocol [8] restricts the MLP protocol in several ways, and does not support some of its features. For example, triggering and zoning are not supported, nor are some lesser features concerning format. SIMPL2 reports all such deviations from MLP as errors or unsupported features.

Cellpoint's extensions to MLP concern DES-encryption of requests and responses, and the capability of sending charging information to the charging subsystem of the location server.

DESIGN AND IMPLEMENTATION

The initial specification and implementation proposal for SIMPL2 was written in August-September 2001, based on a preliminary version of MLP 1.1. Business events put the actual implementation on hold until January 2002, with a new development team (the authors) implementing the SIMPL2 version of MLP 3.0.0. The first commercial release of SIMPL2 was in June 2002. Total development time until final delivery was thus roughly 18 man months.

The final development schedule for SIMPL2 was quite aggressive, with prototyping starting quite some time before the standards were even close to finalization. During six months of development (January to June 2002), there were no less than fourteen releases of the SIMPL2 specification, often driven by changes to the underlying MLP being standardized by LIF. Even as late as May, there were major, incompatible changes and extensions in how to specify such things as the coordinate system being used.

At the same time, there was a considerable learning curve for the developers involved: none of us were experienced in the ways of geographical information systems, and the MLP and SIMPL2 standards mainly specified the syntactic formats of requests and responses, rather than the semantics. Thus, we also had to learn how to interpret the requests and translate the intentions to the underlying positioning system.

The CMLP itself furthermore evolved rapidly during this period of time, adding numerous features that required support or changes in the protocol modules. Parts of CMLP were also redesigned to support SIMPL2.

We began by building a basic prototype implementation, integrating the freely available `xmerl` and `inets` subsystems into CMLP and using them for HTTP and XML processing. We then used existing CMLP code for interfacing to the rest of the positioning system, and wrote the code to convert XML to the internal format.

The responsibility of the validation code is to check that the incoming values are valid according to the protocol, and to turn the values into a suitable data structure for the positioning system. A second task is to manage the MLB and MLS roaming functionality, i.e., to act as a client to some other SIMPL2 or MLP

server, which requires validation and conversion of responses similar to what was done on requests, and to convert internal data into SIMPL2 positioning requests.

The method of development initially was that of exploratory programming. As the implementation matured, we realized that there were a number of as-yet poorly defined features in MLP and SIMPL2, which, along with changes to the specifications, made necessary continuous fixes to the protocol code.

As an example of the latter, the final draft of SIMPL2 was released on May 12, with seven subsequent drafts released between then and June 7. During the entire development period, there were also extensive mail conversations and revisions in between drafts.

This imposed a considerable burden on us, since Cellpoint also works with a system of weekly formal code freezes that are tested by a separate group of testers. The internal goal was to release a working version of SIMPL2 every week, while tracking changes to specifications, external change requests and protocol bug fixes and clarifications. This proved to be difficult just by patching and debugging the existing code, and we found that our calendars were filled with reacting to such events.

We realized that we needed to retool the code to rapidly react to changes. We decided to implement what we called *rapid redevelopment* to enable us to respond very quickly to changes and trouble reports. This was done as follows.

Abstraction and refactoring

Modularity, macros and abstract data types were used extensively in development. As the implementation matured, we tried to refactor the code further.

We found that Erlangs records were a source of errors. Records are useful because they make changes in data representation easy. The record notation is also handy because it can be used in pattern matching as well as expressions. However, there are also complications:

- There are often constraints between the fields, e.g., when user name has been filled in, then user data fields must be consistent with the name. These constraints are frequently undocumented.
- When new fields are added, every existing record creation in the code must be verified and perhaps updated.

These properties slow down code changes. We opted to throw out explicit use of records and deploy abstract data types instead[1].

As a further benefit, abstract data types also made the code clearer, since the operation is named.

Data driven design

A major source of change in MLP was the format of XML data, both in what attributes were available and their format. Initially, we used a straightforward design to validate formats, structured roughly as follows[2]:

```
check([svc_init, hdr, client, id], ID, State) ->
   check_client_id(ID, State);
```

[1]For readers unfamiliar with this concept, an abstract datatype encapsulates the representation of a data structure inside a module; a well-known Erlang example is the `dict` module. Object oriented programming also uses encapsulation, though with objects instead.

[2]This is not the actual code.

```
check([svc_init, hdr, client, pwd], Pwd, State) ->
   check_client_pwd(Pwd, State);
...
```

This turned out to be a source of problems. The reason was that all aspects of the XML code changed: tag and attribute names, contents, locations in the request, as well as data format names and data format definitions, which quickly became confusing. Not all changes in specification were correctly introduced, and it was difficult to keep track of all the versions.

We refactored validation into an interpreter. The XML attributes and formats were specified as a term:

```
-define(tag_specs,
        [{'svc_init', blank,
         [{'ver', {'or', [{member, ["3.0.0"]}, {unsupported, ver}]}}]},
         ...
        ]).
```

In the case above, the tag `svc_init` has an attribute `ver` which must have the value `"3.0.0"` or the request will get an 'unsupported' error. Furthermore, `svc_init` must not have data content itself.

Each SIMPL2 data type was also defined in a rule engine:

```
apply_rule(blank, Value) ->
    [] == strip_whites(Value);
apply_rule({'or', Rules}, Value) ->
    lists:any(fun(Rule) -> apply_rule(Rule, Value) end, Rules);
...
```

When checking data, the refactored program first locates the rule to use from the specification term, then invokes the correct rule in the rule engine.

We call this a *data driven design*. The net effect is to separate the *adminstrative* aspects from the *specificative* aspects of checking.

Ensuring that we followed the current specification was simplified immensely, since the specification part changed most rapidly, and this was now much easier to verify quickly. In the first part of the project, testers found most of the data format bugs. As we neared delivery, the rules engine approach had overcome this problem and we could instead turn to fine-tuning the definitions in the SIMPL2 specification.

Testing framework

As we have described, SIMPL2 is a very flexible protocol with many features to implement and test.

To improve internal quality assurance at releases, we implemented a testing framework, also in Erlang. We decided to implement an external testing framework, but not one for unit testing. The reason was that automated unit testing seemed harder to get right and Erlang's purity and natural interactive bottom-up development at the same time made reasonable manual unit QA more straightforward than one might expect.

Our testing framework generates requests and the expected responses, sequentially tests every case, and signals any deviant responses. At present, there are roughly a hundred test cases. We also implemented a regression testing framework to ensure that old bugs did not reappear: when there is a bug report, we write a test case that triggers the bug and re-run the test at subsequent releases.

The testing framework is data driven: each test request is specified as an Erlang data structure, which is translated into XML text. Specifying a series of test requests looks like:

```
test_series(1, 1) ->
   Clients = [{"service_a", "secret", ?OK}, ...],
   MSID = "...",
   [ {Expect,
      ?svc_init(?hdr_client(Name, Pwd), ?slir(?msids(MSID),
                ?default_geo_info))
     || {Name, Pwd, Expect} <- Clients ];
...
```

This generates one test case per client, with an expected outcome Expect and a request to be sent. The list of Clients consists of cases that should pass or be rejected, as specified by the programmer.

The testing framework sends each request to the server under test, awaits the response and notifies the user when there is an unexpected response (ranging from socket errors to XML validation errors).

Social aspects

Our development methods have also made extensive use of pair programming, brief coordination meetings (sometimes several per day) and the use of instant messaging to keep developers synchronized, even though there were only three main developers (along with half a dozen developers working with related items and three testers). This has improved code quality by keeping all developers "working in the same paradigm". The discussions have also turned out to help with finding and resolving the grey areas of SIMPL2 and MLP; at least one of our findings has led to changes in the MLP standard specification.

DISCUSSION

How did rapid redevelopment work out in practice? Quite well.

Since the XML tags, attributes and validation rules changed very frequently, the rules-based approach to validation was extraordinarily helpful: the rules are easy to read, understand and change compared to code. Verifying that the implementation conforms to the specification is much easier.

A second improvement is that using abstract data types meant we could control how data was manipulated, we could validate inputs and outputs and we could change representations easily. By not exposing record datatypes, we ensure discipline in how crucial data is created, used and accessed[3].

Because of the many features of SIMPL2 and their potential interactions, our testing framework has been invaluable. As a trouble report appears and is fixed,

[3]This requires programmers not to break data abstractions, rather than prohibiting them from doing so as do, say, SML:s abstypes.

we add a regression test case. Prior to each weekly release, we check that the code passes all the tests. We then hand it off to the testers for formal quality assurance.

To improve on this, it would probably be useful to have a dedicated tester to add more test cases. While doing this full time for just SIMPL2 would be overkill, it might be a very useful part-time effort.

One can view rapid redevelopment as a form of extreme programming [3]. Here is a comparison. The item bullets are taken from the reference, page 54.

Small releases. We incrementally released code every week once the system was up and running. However, the QA organization had problems in tracking what could be usefully tested. Serious formal testing began only in the later parts of the project.

Testing. XP mandates a strict testing regime. We did not use XP unit tests, but implemented and used an automated testing framework successfully, as described previously. The effect was gratifying, since it improved code quality while reducing our testing effort.

Refactoring. We refactored code thoroughly, as shown previously, but mainly in reaction to problems rather than proactively. This might have been a mistake.

Pair programming. Considerable portions of the code were developed using pair programming, which turned out to be a good choice.

Continuous integration. Erlang naturally enables continuous integration. The normal development cycle is to edit-debug-recompile, then load the code into a running full CMLP system (configured as an MLS or an MLB) to test it.

However, we also had to integrate the results of other developers. This was done at every release and included changes in the system configuration files (including their format) and CMLP restarts. Continuously integrating such changes would likely have slowed development down.

On-site customer. We did not have an on-site customer. As mentioned above, a dedicated test case writer would be useful. Also, having a customer representative available would have clarified a number of practical issues. The main issue in introducing this is probably to motivate the expense of doing it.

Coding standards. We did not use written coding standards, but some principles were obeyed, such as the use of ADTs instead of records.

FUTURE WORK

We are currently mulling over if the code that converts incoming data to our internal format, and back into outgoing data, can be converted into a rule-based form as well. The main problem is that our current set of sketched conversion rules is too large and unstructured to yield any great advantage on ordinary code.

Given the successful rules-based approach to validation, we are also considering writing a tool to translate the rules into a directly validating and translating XML parser. The advantage of this approach is higher efficiency (as long as code size remains reasonable).

However, it is not at all clear that SIMPL2 is a bottleneck. Positioning requests may naturally have long latency (e.g., several seconds), and the cost for SIMPL2 processing is small in this context. While reducing latency and memory footprint per request is still welcome, because system capacity is improved, other tasks have had priority since SIMPL2 was released to customers.

CONCLUSION

We have described the implementation of the SIMPL2 version of MLP 3.0.0 on Cellpoint's location server series, CMLP.

SIMPL2 evolved quickly and had many features; in this context, rapid redevelopment, an XP-like approach based on abstraction and testing, has been highly successful.

In a development project concerned with a large and changing feature set and an aggressive schedule, the resulting emphasis on flexibility and ease of modification has been invaluable in reducing trouble response times. This in turn leads to a virtuous spiral, where development gets the time to rewrite and extend the code before testing, rather than as a response to testing finding bugs.

We have thus found rapid redevelopment to be a fruitful way to develop feature-rich, committee-specified protocols with high demands on reliability, as is often the case in the telecom world.

A version of CMLP including SIMPL2 is commercially deployed at an operator at the time of writing, making this the world's first implementation of the MLP 3.0.0 standard.

KISTA, JUNE-OCTOBER 2002

ACKNOWLEDGEMENTS

The comments of Lars-Göran Ericson and Bogumil Hausman were helpful in preparing this paper.

References

[1] 3GPP, standard document 03.71. http://www.3gpp.org

[2] Joe Armstrong, Robert Virding, Claes Wikstrm, Mike Williams. Concurrent Programming in Erlang. Prentice Hall.

[3] Kent Beck. Extreme Programming Explained. Addison-Wesley, 2000.

[4] Per Bergqvist. Improving Robustness in Distributed Systems. Proc. Erlang User Conference, 2001.

[5] European Petroleum Survey Group. http://www.epsg.org

[6] Johan Hjelm. Creating Location Services for the Wireless Web. Wiley, February 2002.

[7] Location Interoperability Forum. LIF Mobile Location Protocol. LIF TS 101 v3.0.0, rev 2. June 3, 2002.

[8] Rob Schmersel. SIMPLv2.0 specification. Internal Cellpoint document.

Questions please use : info@blueposition.com

Introducing

BLIS4

Bluetooth Location Information System

Using Erlang

ePosition A/S
e mobility to the next level
omas Verner
O
ww.BluePosition.com

BluePosition A/S

We take mobility to the next level

- Established in Denmark, Spring 2002.
- Spin off from Ericsson Denmark Solution House.
- Member of the Bluetooth SIG
- Ericsson (and others) partner.
- Erlang User 1996-1999

Bluetooth ERICSSON BlueTags

Questions please use : info@blueposition.com

Imagine if...

- Imagine if your phone calls was automatically routed to either your mobile phone or stationary phone depending on your actual location.
- Imagine that you could located a college using a WEB browser.
- Imagine the better customer service you would offer trough this.
- Imagine the improved efficiency.
- Imagine the reduced phone bill...

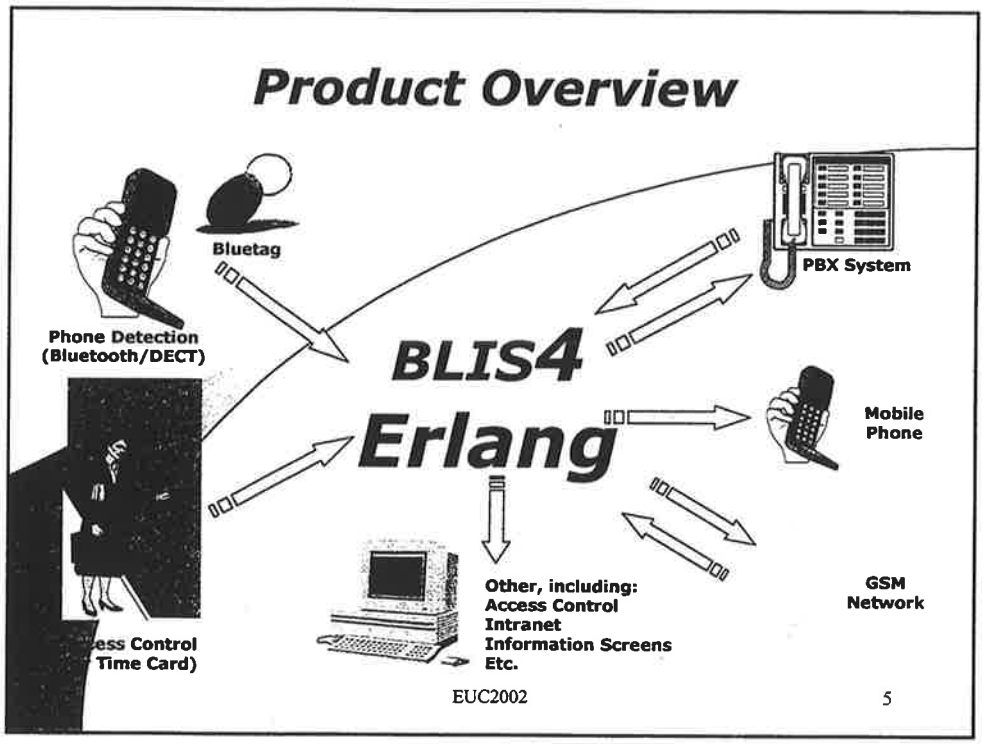
EUC2002 3

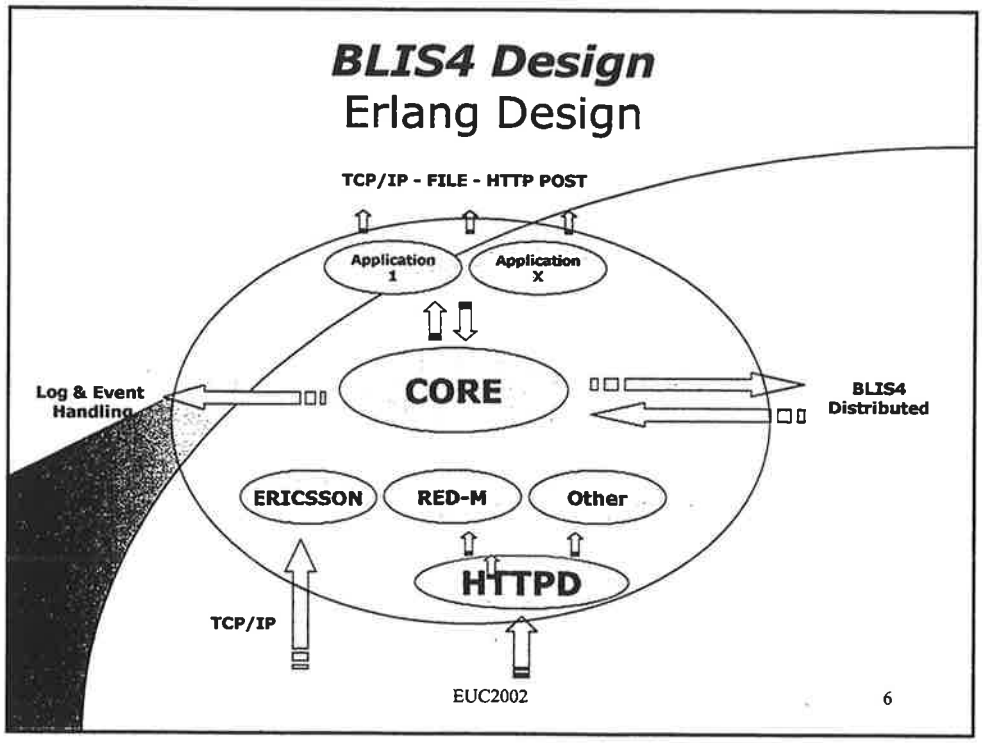
BLIS4
why Erlang

- Solve a problem for "Ericsson"
  - Performance
  - More Logic
  - No time
- Start-up
  - Short Development time
  - A need for low support costs
- Performance
- Fault tolerant / Distribution
- TCP/IP

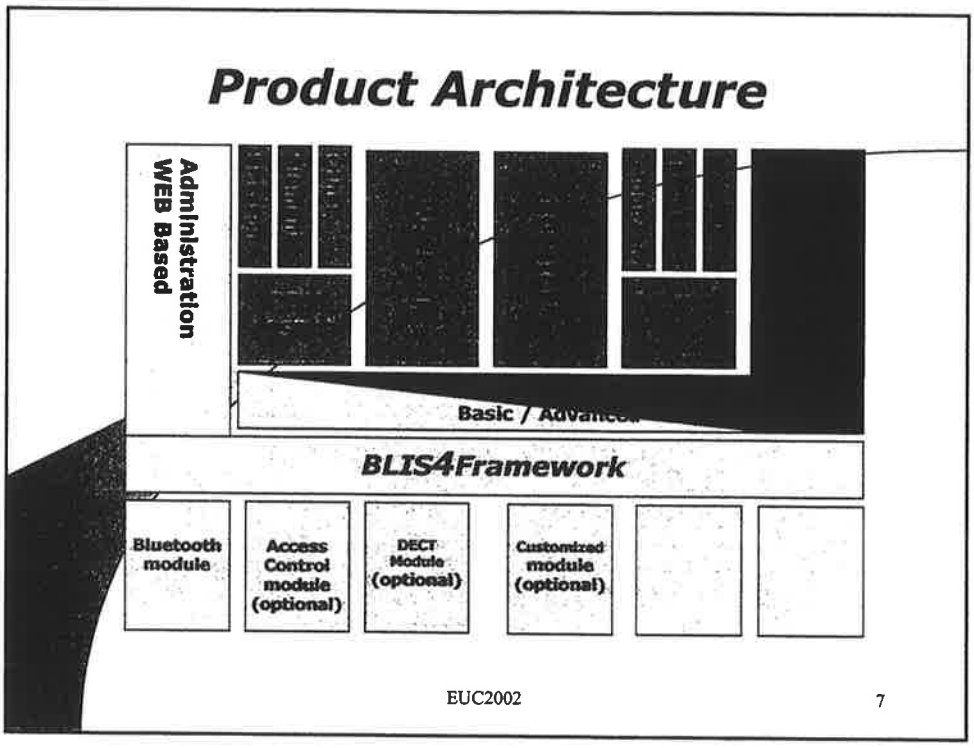
EUC2002 4

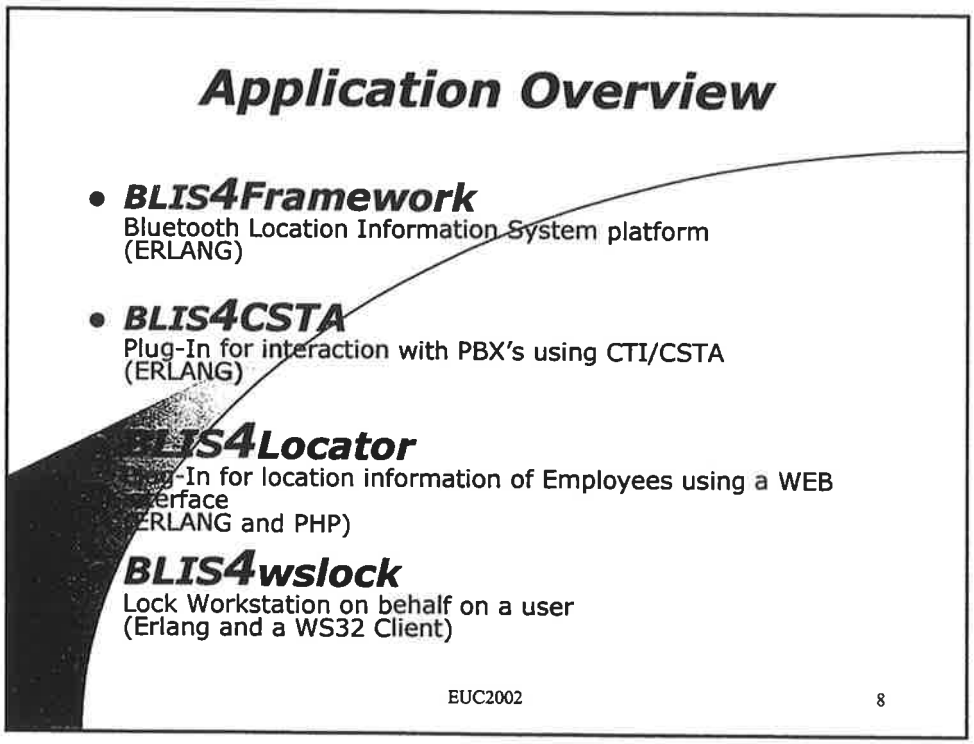
Questions please use : info@blueposition.com

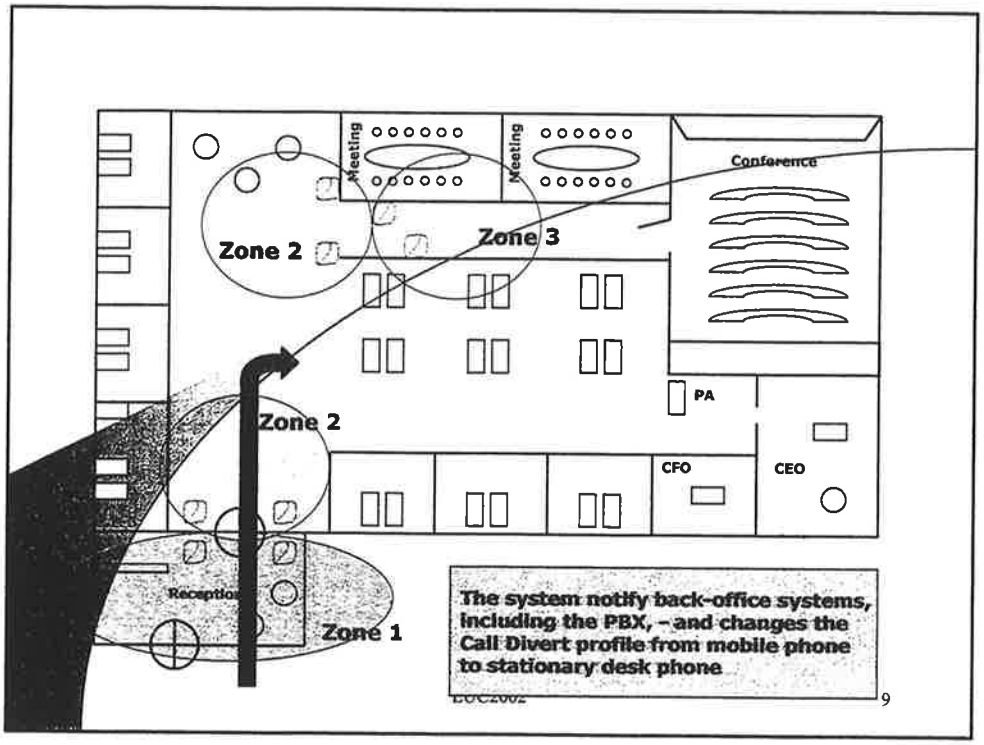




Questions please use : info@blueposition.com







Experiences
using Erlang

- Pretty Code is not an option
- Prototyping still works
- Using only std(lib) functionality
- Building almost all from scratch
  - Easy to introduce new applications
  - Easy to introduce more logic
    - Concepts
    - Constraints

Questions please use : info@blueposition.com

Experiences
interfacing with Erlang

- HTTP / Erlang WEBSERVER
  - POST XML Documents
  - GET for preferences
- Interfacing with third party
  - Specialized TCP/IP Protocols
  - Still a need for C++ & JAVA designers ☹
  - BluePosition Middleware

EUC2002 11

Human Ressource
Experiences

- Sales & Marketing
  - Design keeps their promises (and vice versa)
    - Short time to market, for new applications
    - Trouble shooting
  - Better communication
- Designers
  - Performance is the key
    - They get convinced when the see the performance
  - Stepwise learning
- Customers
  - Seeing is believing ...
    - Both BLIS4 and Erlang

EUC2002 12

Questions please use : info@blueposition.com

Issues

Our "bad" experiences

- Erlang Applications may have overhead
  - Design keeps their promises
    - Short time to market, for new applications
    - Trouble shooting
  - Better communication
- Windows NT (200X(P)) focus
  - ODBC
  - COMET
  - Interface
  - Easy XML
  - Easy HTTP (Client)
  - Often easier to do-it-yourself

Danish Parliament
An Erlang Case

Questions please use : info@blueposition.com

Danish Parliament

- One (1) BLIS4
- 50 + Access Points
- 500 + users
- Handles presence detection to assist call centre
- Eliminates phone calls (ringing) in voting room
- AND YES : IT WORKS !

EUC2002 15

BluePosition A/S

We take mobility to the next level

EUC2002 16

HELGA - A call load generator written in Erlang/OTP

Anand Balagopalakrishnan
Lucent Technologies
Golf View Campus
Wind Tunnel Road
Bangalore, India
anandb@lucent.com

Bagirath Krishnamachari
Lucent Technologies
Golf View Campus
Wind Tunnel Road
Bangalore, India
bagi@lucent.com

ABSTRACT

CDMA2000 1xEV-DO [1] is a 3G standard (TIA/EIA/IS-856, 'CDMA2000 High Rate Packet Data Air Interface Specification') which provides Internet access by providing up to 2.4 Mbps in a 1.25 MHz channel. It is compatible with CDMA networks and is optimized for packet data services.

This paper describes a call load generator written in Erlang [2] which is used to perform load tests on a 1xEV-DO RNC. In this paper we present the details of how such a load generator can be used to perform load tests. We also present some of our experiences in using Erlang/OTP for testing.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Functional languages—*Erlang*
D.1.3 [**Software**]: Concurrent Programming
D.2.5 [**Software Engineering**]: Testing and Debugging—*call generation, load testing*

General Terms

Performance, Verification

Keywords

1xEV-DO, Load generation, Testing, Erlang/OTP

1. INTRODUCTION

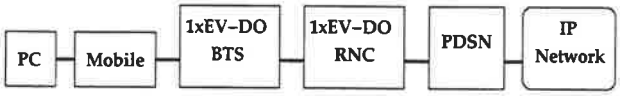


Figure 1: 1xEV-DO Reference Architecture

A high level network diagram of a 1xEV-DO system is given in Figure 1. A PC connected to a 1xEV-DO mobile device is used to connect to the Internet via a Base Station (BTS), a Radio Network Controller (RNC) and a Packet Data Service Node (PDSN). The PDSN terminates the PPP protocol originating from the mobiles and also assigns IP addresses to the mobiles in the network.

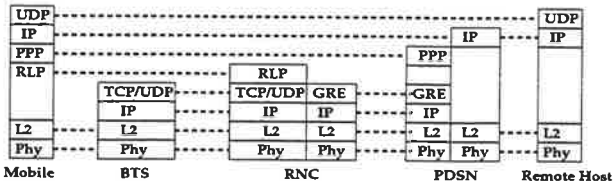


Figure 2: Protocol Stacks for the Reference Architecture

The protocol stacks involved in each network element during call processing are described in Figure 2. The initial part of call setup, which sets up the radio resources, consists of messaging between the mobile, the BTS and the RNC. In the next step, the mobile sets up a PPP tunnel with the PDSN via the resources provided by the RNC. During the PPP tunnel establishment, the PDSN assigns an IP address to the mobile. Once the PPP tunnel is established, the mobile can access the Internet.

HELGA is a software tool written in Erlang which is used to generate call processing load on the RNC. It simulates both a BTS as well as a set of mobiles and sets up data calls via the PDSN. Its primary use is to perform load testing of the traffic processing components of the 1xEV-DO RNC. It is also used to simulate different handoff scenarios. This tool is written entirely in Erlang and is used in conjunction with a packet thrower (written in C) to generate the call load on the system.

2. INTERFACES USED

The RNC call processing components have the following interfaces to the external entities simulated by HELGA

- A TCP interface to the BTS which carries signalling messages
- A UDP interface to the BTS which carries data and signalling messages.

HELGA is the endpoint for these interfaces and passes messages between the "mobiles" and the RNC over these inter-

faces. HELGA implements the IS-856 message set and also simulates the interface between the RNC and the BTS. It also handles the establishment of the PPP tunnel with the PDSN. Each instance of HELGA simulates a single BTS and a (configurable) set of mobiles. Multiple instances of HELGA can be connected to each other by running them over a set of distributed Erlang nodes.

2.1 Functional Decomposition

Since the number of calls under a BTS is configurable and can also change from time to time (depending on the number of handoffs which take place), the implementation of the tool is split into two components - one which simulates the BTS and another which simulates a mobile. This makes the BTS and mobile components loosely coupled and handoffs can be achieved using the distribution mechanism provided by the Erlang emulator. The down link data rate for each call is controlled by an external packet thrower.

2.2 Simulating the BTS interface

For the rest of this paper, the term "BTS" will be used to identify the part of the tool which simulates the BTS interface i.e the set of Erlang processes which simulate the BTS interface. A BTS is simulated by two Erlang processes - one for handling the UDP interface to the RNC and one for handling the TCP interface. These two processes handle the signalling messages between the BTS and the RNC and also route messages between the RNC and individual mobiles. Each mobile is identified by a unique Mobile Identifier and the details of all the mobiles running under a BTS are stored in an ETS lookup table. When run over a distributed set of Erlang nodes, each BTS is capable of communicating with its neighbours and can therefore perform handoffs to other Base Stations.

2.3 Simulating the mobile

For the rest of this paper, the term "mobile" will be used to identify an Erlang process which simulates a mobile. Each mobile is simulated by a unique Erlang process which is identified by its Mobile Identifier. All "mobiles" communicate with the RNC via the processes which constitute the BTS portion of the tool. As soon as a "mobile" is assigned an IP address by the PDSN, it communicates with a packet thrower and requests a data download on the down link at a particular rate. The packet thrower routes the data download to the "mobile" via the PDSN and the RNC call processing software. An inter-BTS handoff is simulated by migrating the corresponding Erlang process from one node i.e. BTS, in the distribution to another.

3. ERLANG/OTP MODULES USED

The main building block used in HELGA is the gen_fsm behaviour which is used to simulate the protocol stack at the mobile. A mobile is simulated by a gen_fsm and uses its Mobile Identifier as its registered name. All mobiles also join a common group, which is created using the pg2 module. All communication within HELGA is done using the Erlang inter-process communication and communication between HELGA and the RNC is done using the gen_tcp and gen_udp modules. All message exchanges between various components of HELGA use binaries. A portion of the state information for each mobile is stored in ETS tables for quick lookup.

At each state of the gen_fsm, a decision is made by the Erlang process simulating a mobile about performing a handoff. This is done by generating a random number which is used to decide whether a handoff needs to be simulated. This number is also used to determine the kind of handoff which will be simulated i.e soft handoff, softer handoff or virtual soft handoff. While simulating a virtual soft handoff, a mobile picks a neighbouring BTS at random and spawns a copy of itself on the node representing the selected BTS. The spawned process is passed a copy of the state information stored in the spawning gen_fsm. Once the new process is spawned, the old process cleans up and exits and the new process continues on the neighbouring BTS. A soft or a softer handoff is simulated by randomly choosing a sector either from the same BTS or from a neighbouring BTS or from the list of sectors in the mobile's active set and then indicating to the RNC that a sector has been added to or dropped from the call.

HELGA can be configured to simulate a stand alone BTS or as part of a distributed set of interconnected base stations. The configuration data for each instance of HELGA includes the details of the BTS, its neighbours, the number of calls to be setup, the location of the packet thrower etc. It is also possible to turn on or turn off handoffs at the BTS level. All the configuration data is read from a file and the parsing of this file is done using the parse_erl_exprs function of the io module.

HELGA also provides periodic snapshots of the throughput received by each mobile under a BTS. This is done by periodically sending a snapshot request message to all the gen_fsms in the system. The gathering, processing and presentation of the periodic snapshot data in a human readable form are made simpler by a judicious combination of higher order functions and pattern matching. The snapshot data are used to study the behaviour and performance of the RNC under different call loads.

3.1 Using HELGA for testing

When started, HELGA reads its configuration file and initializes the BTS component, which in turn establishes TCP connections with the RNC. HELGA then spawns as many gen_fsms as there are calls defined in the configuration file. Each gen_fsm then initializes its protocol stack and proceeds to setup a call with the RNC. When its call setup with the RNC completes, a gen_fsm negotiates with the PDSN to set up a PPP tunnel. After a gen_fsm sets up a tunnel, it is assigned an IP address by the PDSN. Once it receives an IP address, a gen_fsm sends a request for a data download at a particular rate to the packet thrower.

4. EXPERIENCES WITH ERLANG

Before HELGA was developed, Perl[3] was used for developing simulators. The following are some of our observations on using Erlang/OTP

- Learning curve - The learning curve for Erlang is short and steep when compared to languages like C++ and

Perl. Programmers very quickly start getting productive with Erlang.

- Development Time - The time taken to translate a design to its implementation is much shorter with Erlang than it is with Perl. The time taken to implement test tools in Erlang is less than half of the time taken to implement similar tools in Perl. The use of OTP modules like behaviours significantly reduces the development time.

- Speed - Erlang code runs faster than Perl code. Erlang code appears to be approximately five times as fast as Perl code implementing similar functionality.

- Scalability - It is easy to convert a stand alone application into a distributed application because the language (as well as OTP) has constructs which support distributed applications.

- Extensibility - Since the language supports hot code load, it is easy to do incremental development, without even having to restart the tool. Features can also be added or tweaked in real time.

5. CONCLUSIONS

The built in functions of Erlang which support distributed communication and behaviours from OTP helped in radically changing the development cycle. Fewer resources had to be committed to develop a tool in Erlang. Since the tool was developed in an incremental way, with functionality being added iteratively to enhance the tool, bugs in the code were detected very easily. This had a direct impact on the quality of the tool both in terms of stability and in terms of performance. It was easier to run load tests using Erlang code than it was with Perl code.

6. REFERENCES

[1] 3GPP2. cdma2000 high rate packet data air interface specification. Version 2.0, October 2000.

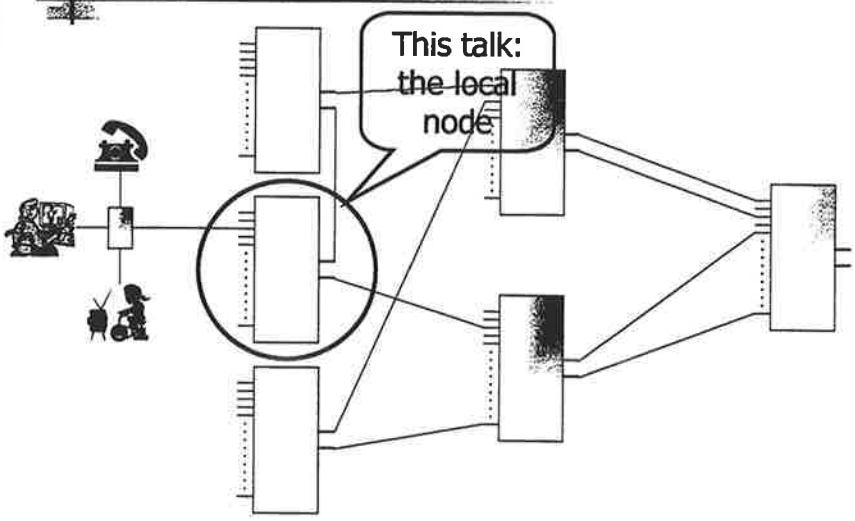
[2] J. Armstrong, R. Virding, C. Wilkstrom, and M. Williams. *Concurrent Programming in ERLANG.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[3] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly, Sebastopol, CA, 1996.

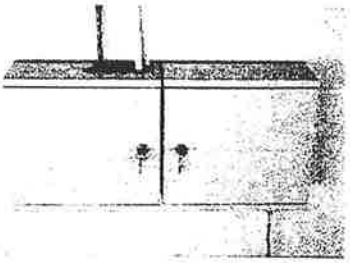
The AXC105 Fibre Switch

Hans Nilsson
hans@erix.ericsson.se

The access system

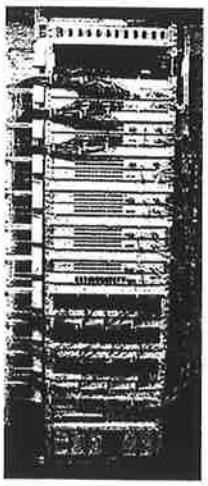


Photos



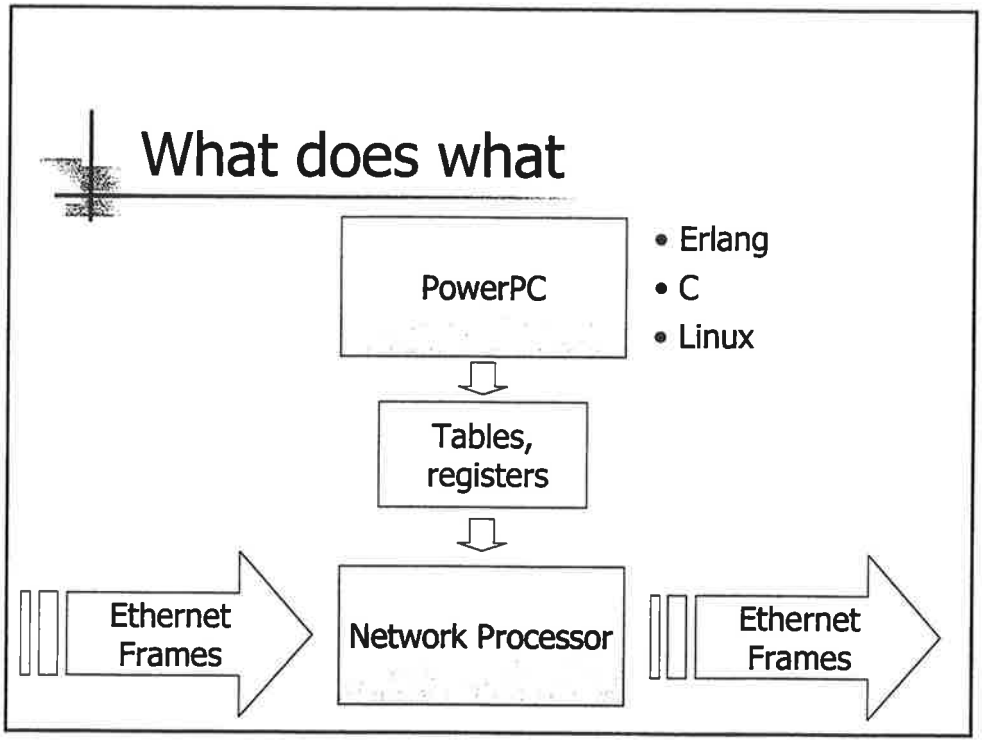
Mounted on a wall with fibres to the users

Rack installation



Inside the local node

- Network processor
  - Assembler
- PowerPC
  - Linux
  - Erlang/OTP *(35 k lines)*
  - C *(10 k lines)*
- Hardware
  - Some strange things...



Experiences

- Good
  - Linux:
    - Much available out there
    - Many experts
  - Erlang:
    - High level
    - Very short development time
  - Network Processor
    - Flexible
- Bad
  - Linux:
    - Not completely perfect
    - Allergic reactions ...
  - Erlang:
    - Nothing bad ☺
    - Allergic reactions ...
  - Network Processor
    - Expensive
    - Allergic reactions ...

Mobile Arts Telecom Platform

19 November 2002

www.mobilearts.se

Content

- Mobile Arts Introduction
- Mobile Arts Telecom Platform
  - ✓ Overview
  - ✓ Technology & Characteristics
- Some great Erlang Open Source applications
  - ✓ jnets
  - ✓ xmerl
  - ✓ ucs

Mobile Arts

Mobile Arts

- Founded in February 2001
- HQ in Stockholm
  - ✓ Branch office in London
- 10 employees
  - ✓ 1 Doctor of Science & 9 Master of Science
  - ✓ Leading edge competence in and experience (>> 125 man-years) from development of GSM/UMTS/Telecom products
    - MSC/VLR/HLR, Mobile SSF, Mobile SCF, UMTS MSC, WAP Gateway, SMSC, MLC, etc.
    - Standardisation(ETSI/3GPP, WAP Forum, LIF, etc.)

Mobile Arts

Strategy

- Concentrate on doing what we know best
  - ✓ Development, GSM/UMTS and other related standards
- Focus on Indirect Marketing & Sales Channels
  - ✓ Mobile Arts will not build-up a large internal marketing and sales organisation
  - ✓ We have established a number of strong partnerships with System Integrators and/or Resellers
- Work closely and actively together with partners and support their marketing & sales activities
- Mobile Arts has been financed entirely through consulting (no loans, no venture capital)

Mobile Arts

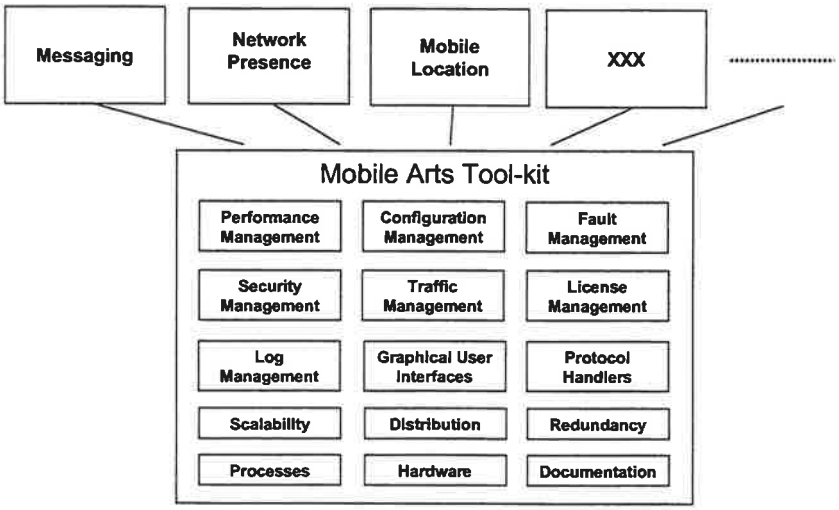
Business Idea

- Mobile Arts provides state of the art Mobile Network gateway products for Messaging, Presence and Location.
- Our products provide Mobile Operators with the key elements required to enhance existing applications as well as launch new applications in various areas, such as SMS, Instant Messaging, Games, Entertainment and Information.
- Mobile Arts products are compatible with GSM/UMTS networks all over the world, regardless of local signalling standards.

19 November

Mobile Arts

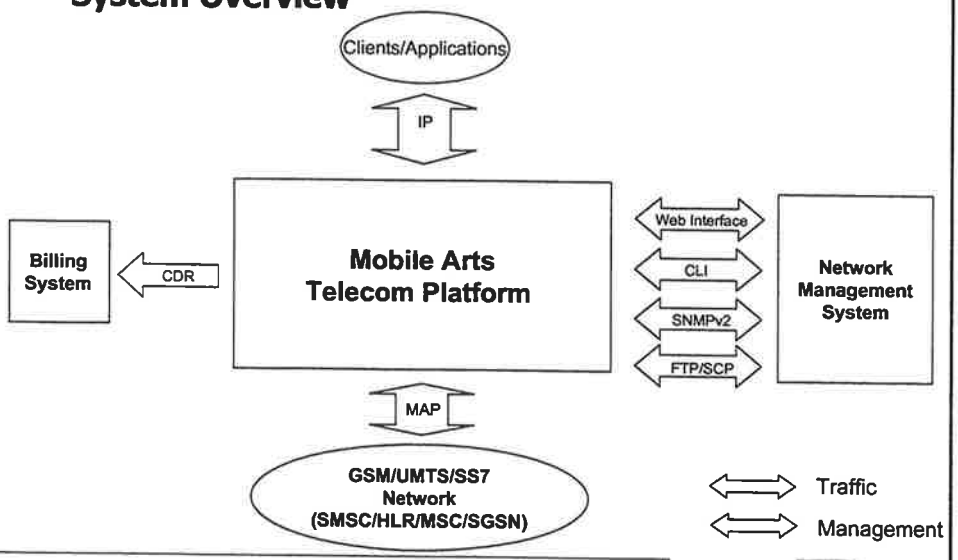
Product Architecture



19 November

Mobile Arts

Mobile Arts Telecom Platform
System Overview



19 November

Mobile Arts

Hardware

- Processor
  - ✓ Currently SUN (e.g., Netra T1 AC 200 or Fire V120)
- SS7 Stack
  - ✓ Ericsson/Tieto-Enator SS7 PCI-boards (one for each host) that each supports two E1 links with up to 16 signalling channels
  - ✓ Full SS7 redundancy (STP/SRP load sharing)
  - ✓ Considering Ericsson "Stack-on-a-Card" SS7 boards
  - ✓ Why Ericsson? Name!

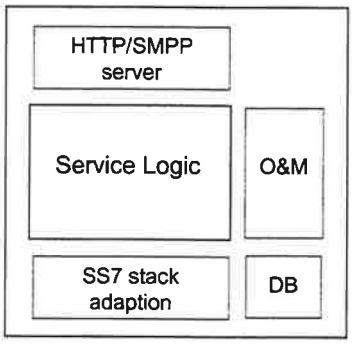
19 November

Mobile Arts

Software

- Operating System
  - ✓ SUN Solaris 8 (2.8)
- Additional software
  - ✓ Ericsson SS7 stack
  - ✓ Erlang OTP (including Mnesia)
  - ✓ OpenSSL
- Application software
  - ✓ Mainly Erlang OTP (drivers to SS7 stack in C)
  - ✓ Full software redundancy with multiple hosts
  - ✓ Current size approximately 5500 lines of C, 130000 lines of Erlang and growing...

Platform Overview



- HTTP server
- XML parser
- SMPP server
- ASN1 encoding/decoding
- SS7 stack adaptation
- Service logic
- O&M
- Database

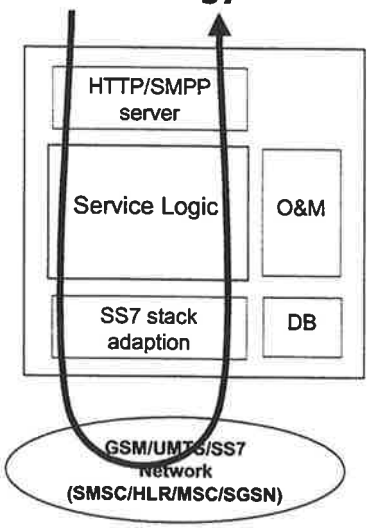
Mobile Arts Telecom Platform
OAM Features

- Configuration management and system administration
  - ✓ Web based GUI
  - ✓ Command Line Interface
  - ✓ FTP/SCP
  - ✓ Local or remote access
- SNMPv2
  - ✓ Fault management
  - ✓ Performance management
- Advanced tailoring of Measurement Reports
- Differentiated Operator access rights
  - ✓ Operator Roles with differentiated Rights

19 November

Mobile Arts

Mobile Arts Telecom System
Technology & Characteristics



- Capacity
  - ✓ ~100 requests/sec (360000 requests per hour) on a single node SUN Netra T1 system (dead slow, but cheap machine)
  - ✓ XML request and XML response
  - ✓ Service Logic: Single FSM MAP operation

19 November

Mobile Arts

Content

- Mobile Arts Introduction
- Mobile Arts Telecom Platform
  - ✓ Overview
  - ✓ Technology & Characteristics
- **Some great Erlang Open Source applications**
  - ✓ jnets
  - ✓ xmerl
  - ✓ ucs

Mobile Arts

jnets – HTTP client and HTTP server

- HTTP client features include
  - ✓ Syncronous/asyncronous request interface
  - ✓ Persistent connections
  - ✓ Pipelines
  - ✓ Proxy support
  - ✓ + more (but lots missing also...)
- Why jnets HTTP server?
  - ✓ Backward compatible with inets 2.6
  - ✓ Standards compliant
  - ✓ Fast core, flexible configuration

Mobile Arts

jnets performance test 1

- Simple GET request against a small static HTML file.
- Setup:
  - ✓ 1 client machine/1 server machine
  - ✓ Client makes a new request immediately after a response was received

Mobile Arts

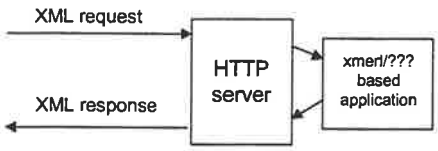
Results 1:

- To be done....

Mobile Arts

jnets performance test 2

- "Web serv:ish" example



Results 2

- To be done

xmerl – The Erlang XML processer

- Written by Ulf Wiger, now maintained at http://sowap.sourceforge.net
  - ✓ Latest release xmerl-0.18
- Late developments:
  - ✓ Improved export functionality.
  - ✓ Support of DOM and SAX style parsing of XML document
  - ✓ Many bugfixes

Mobile Arts

ucs – Erlang Unicode support

- Translates Unicode number to Mnemonic
- Converts from virtually any character sets to Unicode and vice versa, given that there exists a mapping!!
  - ✓ Sometimes very slow
- Converts between IANA defined character set names and corresponding MIB number/character set aliases
- This does NOT give generic Unicode support in Erlang (strings etc)

Mobile Arts

On Reducing Interprocess Communication Overhead in Concurrent Programs

Erik Stenman
Computing Science Dept.
Uppsala University, Sweden
happi@csd.uu.se

Konstantinos Sagonas
Computing Science Dept.
Uppsala University, Sweden
kostis@csd.uu.se

ABSTRACT

We present several different ideas for increasing the performance of highly concurrent programs in general and Erlang programs in particular. These ideas range from simple implementation tricks that reduce communication latency to more thorough code rewrites guided by inlining across process boundaries. We also briefly discuss the impact of different heap architectures on interprocess communication in general and on our proposed optimizations in particular.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, concurrent, distributed, and parallel languages*

General Terms

Languages, Performance

Keywords

Concurrent languages, process scheduling, Erlang

1. INTRODUCTION

Large software systems can conceptually be split into several separate and semi-independent tasks. Concurrency tries to provide a convenient form of abstraction for such situations. Hence it is not surprising that many modern programming languages (such as CML, Caml, Erlang, Oz, Java, and C#) come with some form of built-in support for concurrent processes (or threads). Unfortunately many of these languages only provide very crude low-level support for concurrency; for example interprocess communication is often implemented with shared data structures. Promoters of these designs often motivate the low-levelness with the need for speed, using pretty much the same arguments that adversaries of garbage collection for a long time have been arguing for the need for programmer-controlled memory management.

We believe that a higher level of support is needed, not only for memory management, but also for concurrency. A language should provide high-level concurrency primitives and it should be up to the compiler and runtime system to implement these constructs as efficiently as possible.

With a more natural way to handle interprocess communication, such as through explicit message passing, the programmer can concentrate on what to communicate instead of how. This higher level of abstraction does however come with a price: data sent from one process to another is not automatically *there* in some shared data structure to be used by the other process directly. Instead, the other process must also *receive* the data; this typically requires that a *scheduler* prompts the receiving process to access the message and to possibly take some appropriate action. Scheduling and switching between execution environments of processes do come at a cost and in this paper we will present some ideas for reducing this cost.

Our goal is to eventually have truly lightweight processes where message passing is at least as efficient as method invocation in a modern object oriented language.

We have been experimenting with different extensions to Erlang and different designs and implementations within the HiPE system [3], a native code compiler extension to the Erlang/OTP system provided by Ericsson [6]. The rest of the paper begins with a description of some aspects of this system (Section 2). We then present our three main ideas: a *rescheduling send* (Section 3), a *direct dispatch send* (Section 4), and *interprocess inlining* (Section 5). We end with some concluding remarks.

2. ASPECTS OF CURRENT ERLANG IMPLEMENTATIONS

Although some of the ideas we explore in this paper have also been dealt with by the operating systems (OS) community, it is important to note that our context is different since the concurrency in Erlang is not provided by the underlying operating system. Instead the Erlang runtime system itself is assumed to provide much of the functionality often associated with an OS. For example, the runtime system of Erlang/OTP contains its own scheduler, memory manager, code loader, interface to the file system, and an emulator for BEAM code.

Erlang Workshop '02 Pittsburgh, USA

Clearly, the implementation of the runtime system will have an impact on the performance of concurrent Erlang programs. Let us therefore describe some aspects of the current implementation.

2.1 Erlang processes

Processes in Erlang are extremely light-weight (lighter than OS threads), their number in typical applications is quite large, and their memory requirements vary dynamically. Erlang's concurrency primitives—`spawn`, "`!`" (`send`), and `receive`—allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any data value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs through pattern matching. There is no shared memory between processes and distribution is almost invisible in Erlang. To support robust systems, a process can register to receive a message if another one terminates. Erlang provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

Note that Erlang processes differ from both OS processes and OS threads: An OS-process usually has a separate address space implemented in hardware resulting in the need of TLB flushes and the like, while OS threads usually communicate through shared memory. Finally, OS processes and threads are often implemented in such a way that they can be executed in parallel.

Erlang processes on the other hand are handled by the runtime system scheduler, which selects a process from a *ready queue*. The selected process is assigned a number of *reductions* to execute, called its *time-slice*. Each time the process does a function call, a reduction is consumed. The process is suspended when the time-slice is used up (i.e., the number of remaining reductions reaches zero), or when the process reaches a `receive` and there are no matching messages in its mailbox.

The scheduler is implemented in C as a function that can be called either by the BEAM emulator or directly from native compiled code. The scheduler takes as arguments the process that has been running and the number of executed reduction steps, and returns the next process to execute.

2.2 Heap architectures

Till the fall of 2001, the Ericsson Erlang implementation had exclusively a memory architecture where each process allocates and manages its own memory area. In this architecture, since each process has its own heap, message passing is implemented by copying the message from the heap of the sending process to the heap of the receiving process. After the message is written, a pointer to the message is inserted into the message queue of the receiving process.

We have implemented a *shared heap* memory architecture for Erlang processes [4, 7], which is already included in the current Erlang/OTP release. Concurrently with that work of ours, Feeley [1] argued the case for a unified memory architecture for Erlang, an architecture where all processes get to share the same stack and heap. This is the architecture used in the Etos system [2] that implements concurrency through a *call/cc* (*call-with-current-continuation*) mechanism. The biggest advantage of a shared heap architecture is that sending a message does not require copying. On the other hand, garbage collection stop times might become longer since all processes share the same heap. A shared heap architecture also opens up some other opportunities for optimizing an Erlang system. We exploit some of them in this paper.

2.3 Behaviours

The Erlang/OTP system comes with the powerful concept of *behaviours*. A behaviour can be seen as an implementation of a design pattern. The OTP library supplies a number of predefined behaviours such as *application*, *gen_server*, and *supervisor*.

With these behaviours the programming of concurrent applications can be taken to a higher level since behaviours supply general solutions to common programming tasks. For example, the programmer does not need to get involved in the details of programming a fault-tolerant server that supports code upgrades.

The drawback of using behaviours is a slight loss in efficiency; since the solutions are general, behaviours tend to employ a number of runtime tests to find the specific solution. Unfortunately there is no formal specification of behaviours, and they are implemented entirely in Erlang. This means that with the current implementation the compiler has no real guarantees about the behaviour of a program that uses behaviours. (The only check that is done is by the linter which gives a warning if any callback function needed by a behaviour is missing. There is no guarantee that the callback function does what it should do, and hence currently the compiler can not trust the behaviour declaration for optimization purposes.)

If behaviours became more formally specified, an optimizing compiler could use the behaviour declaration as a hint on where to look for certain types of opportunities for optimization. For example, an application based on the *gen_server* behaviour does indirect (via function calls) message passing and pattern matching on the message. In essence: all messages to the generic server pass through a call function that tags the message with an atom defining the message type and the server then finds the appropriate handler by pattern matching on that tag. In the abstraction of the behaviour the information of the message type is lost, and it can not be found by e.g. conventional partial evaluation since message passing is involved.

This common pattern was actually one of the inspirations to interprocess inlining; we feel that users should be able to use this powerful behaviour without worrying about loss in performance.

3. RESCHEDULING SEND

Interprocess communication in Erlang is asynchronous, and the `send` operation is non-blocking. However, these are actually conceptual aspects on the language level, and there are several ways to implement them in the underlying runtime system.

The current Erlang system is implemented in the natural way, that is, the `send` operation just places the message in the receiving process' mailbox and then the sending process continues executing until it either blocks in a `receive` statement or has exhausted its time-slice.

In most cases, when a process sends a message it is because the application wants the receiver of that message to act upon the sent information. Hence, it would probably be in

the best interest of the sender to yield to the receiver in this case, and let the receiver act on the message. We will refer to this type of `send` as a *rescheduling send* operation.

We therefore propose to implement this by letting the `send` operation, at least in some cases, also suspend the sending process. This would hopefully lead to lower message passing latency since the receiver can start executing directly when a message is sent. We also expect that in many cases the cache behavior would be better since the receiver will get the message while it still is hot in the cache.

In a private heap system, since the message has to be copied, the whole message should be hot in the cache right after the `send`. Hence, it is important to directly switch to the receiving process before the sender starts producing new data. In a shared heap system, the message does not need to be copied but as it is likely to have been created recently, it is also likely to be hot in the cache.

The real benefits of this design will probably depend both on the underlying hardware and on the communication characteristics of the Erlang program. We do not believe that the benefits of this optimization will be very significant in isolation, but the ability to suspend a process directly after a `send` can open up possibilities for further optimizations.

4. DIRECT DISPATCH

The idea to let the `send` operation suspend the process can be taken one step further by completely bypassing the scheduler. Since it is often the case that the sender is suspended waiting for the receiver to react on the sent message, a natural action for the sender to take is to contribute its remaining time-slice to the receiving process hoping that this will lead to a faster response. We therefore propose a *direct dispatch send* operation: After `send` has placed the message in the mailbox of the receiver, any reductions left could be passed to the receiving process, which could be woken up directly (ignoring the ready-queue).

With this approach, some overhead of the scheduler could be eliminated and the latency of message passing would be reduced even further. Since this approach would also guarantee that it really is the receiver of the message that will execute next, the effects of having the message in the cache will hopefully also become more evident.

As with any process, the receiver is allowed to execute until it blocks in a `receive`, or the reduction count reaches zero, or it performs a direct dispatch send of its own. If the receiver was taken from the ready queue and becomes suspended because any of the two latter reasons (i.e. it is still runnable), it is important to reinsert it into the ready queue in the same position as it was taken from, lest it might starve.

If the receiver performs a direct dispatch send back to the original sender then that sender can get back the remaining reductions and can keep on executing as usual. This way the common case, where one process sends a request to another and then receives a reply to the request, can be almost as efficient as a function call.

5. INTERPROCESS INLINING

To take these optimization ideas even further, we would like to not only change the behavior of the send operation in the runtime system, but actually optimize the code executed before a send and after the accompanying receive. The goals of this optimization are to reduce the overhead of message creation (for example, by avoiding enclosing parts of a message in a tuple), reduce context switching overhead, and open up possibilities for further optimizations by considering the code of the receiver in combination with that of the sender.

The optimization is performed on a pair of functions, the function containing the `send` and the function containing the `receive`. We will refer to these functions as f and g respectively, and the pair as a *candidate pair*. The code at the point of the `receive` statement in g is inserted into the code of f at the point of the `send`. The resulting code is then optimized using standard compiler optimization techniques.

To perform this optimization we have to respect the following requirements:

1. Find a program point where a `send` is performed.
2. Find out at which `receive` statement this message is received.
3. Ensure that, at the time of the `send`, the receiving process is suspended at the `receive` statement found in step 2.
4. Ensure that the mailbox of the receiving process is empty.

Since this process communication behavior can be hard to analyze statically — in any concurrent language and in a dynamically typed language such as Erlang in particular — we propose the use of profiling and dynamic optimization to implement this interprocess code merging.

To do this we take advantage of two features of Erlang: *hot code loading* and *concurrency*. The presence of concurrency makes it possible to implement supervision and recompilation in processes in a way which is separated from the application. Support for hot code loading ensures that there are methods for linking and loading re-optimized code into a running system in an orderly way. We also use a special HiPE extension that makes it possible to replace code on a per function rather than on a per module basis.

We first instrument the system in order to profile the aspects that can trigger a recompilation. During normal execution a supervisor process monitors the profile. When the profile indicates that a part of the program should be recompiled, the supervisor starts a separate process for the compilation.

The gathered profile information is then used to choose candidates for inter-process optimization. These candidates consist of pairs of program points; one program point refers to a `send` statement, and the other refers to the corresponding `receive` statement. These pairs are found by profiling each send to collect information during execution. The collected information has two components: information about the destination (*Dest*), and the number of times the instruction is executed (*Times*). The *Dest* field is initialized to *none*, and the *Times* field to zero. When the send is executed, the *Times* field is increased and the receiving process is checked. If the mailbox of the receiver is empty then the program counter (PC) of the receiver is checked; if the PC is equal to *Dest* or if *Dest* is equal to *none* then *Dest* is set to PC. Otherwise *Dest* is set to *unknown*.

In the case where a `send` has only one `receive` destination, the `send/receive` pair is considered as a candidate for the

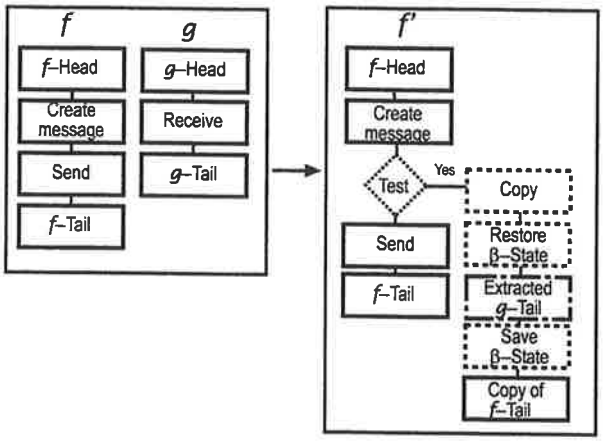


Figure 1: Before the merging, function f is executed by α and function g is executed by β. After the merging, f' is executed by α.

optimization. On the other hand, if a `send` has more than one destination, even if it is just two different destinations, then the profiler will classify the `send` as *unknown* and no optimization will be performed.

When a frequently executed candidate pair is found, the functions containing the `send` and the `receive` are compiled to intermediate code. The intermediate code fragments of the two functions are then merged. In short, the merging is done so that the program point of the `send` statement is connected with the program point of the `receive`. The resulting code is optimized and compiled to native code.

To ensure correct behavior, execution of the optimized version is guarded by a run-time test. This test checks that requirements 3 and 4 in the above list hold; otherwise the original unoptimized version is executed.

5.1 The transformation

We will refer to the sender (the process executing f) as α and the receiver (the process executing g) as β.

For a given `send`, the function f can be divided into the following abstract blocks of code:

1. Head (code preceding the `send`)
2. Message creation
3. `send`
4. Tail (the rest of the code)

The function g is divided into:

1. Head (code preceding the `receive`)
2. `receive`
3. Tail (the rest of the code)

The intention of the transformation is to allow process α to execute code that would otherwise have been executed by process β. Thus, the resulting code for α, function f', will contain fragments of the code from g; see Figure 1.

The merged function f' is a copy of the function f with these six additions:

1. Test — A test is inserted before the `send` in f'. This test checks whether β is suspended at the right program point (at the `receive` in g) with an empty mailbox. If this test succeeds the execution continues with the optimized code (item 2), otherwise the execution continues with the original code of f.
2. (Message copying) — In a system with a private heap architecture the message is copied from the heap of process α to the heap of process β using an explicit copy operation. (In a shared heap system, no copying is needed.)
3. Restore state — All live β–temporaries are read from the stack of β. (This is done by consulting a mapping from intermediate code temporaries to stack positions.)
4. Code from g — The code from g that is suitable for external execution is then executed.
5. Save state — All live β–temporaries are written back to the stack of β.
6. f–tail — A copy of the tail of f is executed.

Since we rely on a subsequent optimization pass to clean things up, performing the merging is straightforward. The subsequent optimization pass, which performs a *generalized constant propagation* and *dead-code elimination* [5], will remove unused paths from g.

In our context, the generalized constant propagation propagates not only true constants, but also Erlang terms such as lists and tuples with dynamic elements. The propagated information is then used to fold tests and element extractions on these structures. When the tests are folded and short-circuited, we perform dead code elimination and removal of unreachable code.

Often in Erlang, parts of the messages are just used for switching on the type of message. Interprocess optimization together with generalized constant propagation helps us avoid the copying of these parts of the message.

The code from g has to be rewritten so that it can be executed "externally", that is, from within process α. This means that the primitives we want to inline have to be rewritten for external execution.

We can extract almost all instructions from g for merging with f, as long as the code fulfills four prerequisites:

1. There has to be some way of ensuring that we do not get code explosion.
2. The code may not suspend.
3. The control flow may not be passed to code that is not adapted to external execution.
4. The extracted code must terminate otherwise process α might hang.

To make sure that these prerequisites are fulfilled some instructions are not extracted:

1. A `call` to another function, a meta call (`apply`), or a `return` can not be extracted since the control could be passed to code that is not adapted for external execution.

2. Instructions that lead to the suspension of the process, such as the explicit suspension instruction or a `receive`.

3. Some built-in functions are large and uncommon and not worth the effort to adapt for external execution.

4. Non-terminating code is unacceptable. If some bug in β makes it loop forever, we do not want this bug to propagate to the process α. To ensure that the extracted code terminates, we do not accept any loops in the control flow graph of the extracted code. Note that this is not such a harsh restriction as it may sound, since the only way to get a loop in the intermediate code is by making a tail-recursive call where the caller and the callee are the same. If there is a loop it will probably contain the `receive` that caused the extraction in the first place. In this case the control-flow graph will be cut at this point and the loop will be broken.

The instructions in the g–tail that do not belong to any of the categories listed above are extracted. A control flow path that contains an instruction that is not extractable is cut just before that instruction.

To propagate changes in the state of β we have to save the new state at the end of the extracted code. To this end, we write all live temporaries back to the stack at the end of each path of the extracted code. At the end of each of these paths, the continuation pointer of β is set to point to a stub containing the instructions from that path that could not be extracted from g.

To simplify optimization we duplicate the tail of f. At the end of each path of the extracted control flow graph we insert a `jump` to this copy. This ensures that when the code in the copy is reached, the execution is guaranteed to have passed through the code extracted from g.

5.2 Further considerations

In a runtime system architecture where each process allocates its private heap, the garbage collector typically relies on the fact that all data structures accessed by a process are allocated on the heap of that process. This invariant is temporarily broken while the process α accesses the state of process β, but since we have control over when α is suspended and when garbage collection is triggered, we can ensure that the invariant is maintained at these points. In a shared heap architecture, this is not a problem.

Our inter-process optimizer will change the scheduling behavior. One might suspect that this could lead to a change in the concurrency semantics of the program. However, note that since in the optimized code we do not allow the code from g to loop and count each reduction that would have been counted before the optimization, the observable behavior will remain unchanged.

The inter-process optimizer will merge code from two functions (f and g). If the module of g is updated with hot-code loading, old code from g will remain inside f (actually in f'). However, this code will never be executed, since the runtime test in f' only succeeds when the receiver is suspended from old code. (If the module containing f is replaced then all optimized code is removed and in this case there is no problem at all.)

5.3 Return messages

The situation where the receiver of a message sends a message back to the original sender is so common that we have decided to handle this situation specially. The technique we have devised requires the following criteria to be fulfilled:

1. There is a `send` in g–tail.

2. The destination of the `send` in g is the process α.

3. All paths through f–tail contain a `receive`.

4. The mailbox of α is empty.

We ensure these criteria by first of all always check that the mailbox of α is empty before we use the optimized code. By doing this check in the beginning, we get a very simplified control flow graph for f'.

In a private heap system we just copy the message from the heap of process β to the heap of process α, if the destination of the `send` is α. In a shared heap system no copying is needed, the pointer to the message can be put directly in the temporary containing the received message. Now, the nice thing is that by using *generalized constant propagation* we can often remove the runtime tests completely. Depending on how the message is used, we might also get rid of the copying between the processes completely even in a private heap architecture.

5.4 Potential gains

With interprocess inlining we can reduce the overhead of process communication in four different ways:

1. **Short-circuit switches on messages**
   We can use the information about the form of the message to short-circuit the pattern matching in the `receive`. Since the switching usually is made up of several tests on heap allocated data, short-circuiting results in a control flow path with fewer `load`, `compare`, and `branch` instructions.

   We also expect that this will also make the hardware prefetching mechanisms work better. If the receiver can receive several different messages that have the same frequency, then the switch will go in different ways each time rendering the prediction useless, which results in pipeline stalls.

2. **Reduce message passing**
   It is quite common in Erlang programs that a process creates a message, sends it to another process, which subsequently performs some matching on the structure of the message, accesses some components of the message and never looks at the whole message again.

   By short-circuiting switching on the message we can avoid the creation of the message (and also save time in the garbage collector).

3. **Reduce context switching**
   We can, in the cases where the receiver immediately answers, remove the context switch completely. This not only means that the receiver does not need to be scheduled, but it also means that the executing process does not need to be suspended. Measurements indicate that in many concurrent Erlang programs the processes do not exhaust their time-slice but they are

instead suspended on `receive`. If the sender can keep on running until the time-slice is used up then the expensive scheduler would be executed less. Letting the same process execute longer also results in better cache behavior.

4. **Enabling of further optimizations**
The most significant gain can come from the ability to do optimizations on the merged code, just as the real gain from procedure inlining comes from the optimizations done after the inlining. We get the possibility to do, for example, constant propagation, common subexpression elimination, and register allocation, on merged code from the sender and the receiver.

6. CONCLUDING REMARKS

We have presented several different methods for cross-process optimization aiming to reduce the overhead for interprocess communication. These methods also enable further optimizations across process boundaries, such as constant propagation and more global register allocation. The context switch can be completely eliminated in some cases, reducing the overhead for concurrency.

These optimizations will speed up existing Erlang programs without requiring any modifications to the source code. Since the use of processes will be less expensive, the usefulness of concurrency is extended, making it possible to use processes in cases where it previously has been considered too expensive.

Acknowledgments

We thank Sven-Olof Nyström for interesting discussions on the implementation of interprocess inlining. This research has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development.

7. REFERENCES

[1] M. Feeley. A case for the unified heap approach to Erlang memory management. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[2] M. Feeley and M. Larose. Compiling Erlang to Scheme. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in LNCS, pages 300–317. Springer-Verlag, Sept. 1998.

[3] 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. ACM Press, Sept. 2000.

[4] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88–99. ACM Press, June 2002.

[5] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.

[6] S. Torstendahl. Open Telecom Platform. *Ericsson Review*, 75(1):14–17, 1997. See also: http://www.erlang.se.

[7] J. Wilhelmsson. Exploring alternative memory architectures for Erlang: Implementation and performance evaluation. Uppsala master thesis in computer science 212, Uppsala University, Apr. 2002. Available at http://www.csd.uu.se/projects/hipe.

Distel: Distributed Emacs Lisp (for Erlang)

Luke Gorrie*

November 10, 2002

Abstract

Distel is an Emacs-based user-interface toolkit for Erlang. We introduce "Emacs nodes" using the Erlang inter-node distribution protocol, and make communication natural by extending Emacs Lisp with Erlang's concurrent programming model. The extensions are intended for creating Emacs front-ends to Erlang programs, in combination with Emacs's traditional user interface facilities.

We present an introduction and tutorial on Distel programming, and show how to write a complete Erlang process manager in Emacs Lisp. We then present a suite of Emacs extensions for Erlang development called the `erlang-extended-mode`, and describe the implementation of the Distel runtime system.

1 Introduction

Distel (rhymes with "crystal") is intended for controlling Erlang [1] [2] programs with Emacs. The idea is to take the most essential features of Erlang and integrate them into Emacs Lisp [3], so that the two can communicate in a natural way. The features we selected are processes, pattern matching, and distribution, and they are reproduced faithfully at a high level, though many details differ. In general, higher priority is given to neat integration with Emacs Lisp than to exact reproduction of Erlang semantics.

The core of Distel is essentially an Erlang distribution library, much like the `erl_interface` for C in OTP [2], extended with ideas from the `Etos` [4] Erlang-to-Scheme compiler. Whereas `Etos` implements a complete Erlang compiler and runtime system in Scheme, Distel is a hybrid system, and implements "just enough" of Erlang to support concurrent programming in Emacs Lisp. In particular, Distel is implemented only with normal Lisp functions and macros, and has no special interpreter loop or compiler.

This paper is organised as follows. Sections 2-4 describe the Emacs Lisp programming extensions, and Section 5 uses them to present a small but complete process-manager application, as a tutorial for Distel application development. Section 6 describes the implementation of Distel itself. Section 7 describes the `erlang-extended-mode` and the development tools it includes. Sections 8-10 discuss the past, present, and future of Distel, Section 11 describes related work, and Section 12 concludes.

2 Processes

Emacs Lisp processes are the fundamental feature of Distel, and are provided with a set of Lisp functions and macros that correspond to Erlang's Built-In Functions (BIFs) and language constructs. In fact, the programming interface for Emacs processes is similar enough to Erlang that the best introduction is to see how a simple Erlang process can be rewritten in Emacs Lisp. A message-counting Erlang process is shown in Figure 1, and an Emacs Lisp version in Figure 2. The similarities of the programs should help to shed light on how the Emacs Lisp process works – we'll fill in the details as we go along.

We can test the Emacs message counter by spawning one and sending it some messages:

```
(erl-spawn
  (spawn-counter)
  (erl-send 'counter 'one)
  (erl-send 'counter 'two)
  (erl-send 'counter 'three))
```

*luke@bluetail.com

```
spawn_counter() ->
    spawn(fun() ->
              register(counter, self()),
              counter_loop(1)
          end).

counter_loop(Count) ->
    receive
        Msg ->
            io:format("Got msg #~p: ~p~n",
                      [Count, Msg])
    end,
    counter_loop(Count + 1).
```

Figure 1: Message counter process in Erlang

```
(defun spawn-counter ()
  (erl-spawn
    (erl-register 'counter)
    (&counter-loop 1)))


(defun &counter-loop (count)
  (erl-receive (count)
      ((msg (message "Got msg #%S: %S"
                     count msg)))
    (&counter-loop (+ count 1))))
```

Figure 2: Message counter process in Emacs Lisp

Which will produce the following reports in the "`*Messages*`" buffer:

```
Got msg #1: one
Got msg #2: two
Got msg #3: three
```

`(erl-spawn ...)` creates a new process. It is a macro, and the enclosed code is executed in the new process. The process has its own buffer, which can be used in any way – contain text, use modes, or visit files. The buffer isn't displayed automatically, but can be made visible with Emacs functions like `display-buffer`. Because each process has its own buffer, buffer-local variables are effectively process-local – they can be used to store process state, much like the Erlang process dictionary.

`(erl-send` *who message*`)` sends a message to a process. The *who* argument accepts the same types as Erlang's `!` operator: a PID (local or remote), a registered name denoted by a symbol, or a remote registered name denoted by a `[name node]` vector. (Here, as elsewhere in Distel, vectors are used where Erlang uses tuples.)

`(erl-register` *name*`)` assigns the current process a registered name, like the `register/2` BIF in Erlang.

`(erl-receive` *saved-vars clauses after...*`)` receives a message by pattern matching (it is more complicated than Erlang's `receive`, due to implementation trade-offs discussed in Section 6.) *Saved-vars* names the local variables that will be used once a message is received (other local variables become unbound.) *Clauses* specifies which messages can be received and how they are handled. The syntax for each clause is `(`*pattern body...*`)`, where *pattern* is an Erlang-style pattern (described in Section 4), and *body* is one or more Lisp expressions to run when the pattern is matched. There are also zero or more *after* expressions, which run after a message is handled, regardless of which clause matches.

Most importantly, `erl-receive` *never returns.* Instead it bundles up the execution state and `throw`'s it directly back up to a scheduler loop, bypassing any code on the stack. This is the biggest difference from Erlang programming style: in Erlang a `receive` means "handle a message and then return," but `erl-receive` means "this process state is complete – here is the next one." This is an important point for programming with Distel, and leads to writing Emacs processes in *continuation-passing style* [5] [6], where "what to do afterwards" is given explicitly to `erl-receive` instead of relying on the stack.

Because `erl-receive` doesn't return, and nor do functions that call it, they should only be *tail-called* – called as the last thing a function does. This rule is made explicit in Distel programs by the convention of naming each function that leads to `erl-receive` with an "`&`" prefix, so

that we know to only call it in tail position. The & naming is applied to all functions that call either `erl-receive` or another &-function, except when the calls are wrapped in an `erl-spawn`, because `erl-spawn` catches the `throw` and returns normally.

Returning to Figure 2, we can see that `&counter-loop` is specially named because it directly calls `erl-receive`, while `spawn-counter` is not because although it calls an &-function, it does so inside an `erl-spawn`.

3 Distribution

Emacs processes can communicate directly with actual Erlang processes in other nodes, via the Erlang distribution protocol [7]. Like in Erlang, most BIFs accept either local or remote PIDs, for example `erl-send`, `erl-link`, `erl-exit`, and so on. The Erlang method of sending messages to remote registered processes also works, so to achieve:

```
{foo, bar@cockatoo} ! Message.
```

We write the equivalent:

```
(erl-send [foo bar@cockatoo] message)
```

This simple mechanism suffices to bootstrap full communication, because normal Erlang nodes automatically run a set of useful registered servers. The RPC server, registered with the name `rex`, is the most handy – it receives requests to apply a function with some arguments, and sends back the results. This server is used throughout Distel programs to make RPCs to Erlang nodes.

Of course, when a message is sent from Emacs to Erlang (or vice-versa), it is necessary to translate the data in the message between languages. In other words, we need a mapping between Erlang types and Emacs Lisp types. For Distel we have chosen a mapping that is convenient to use, though not complete or symmetric.

Some types map perfectly: lists, atoms with symbols, tuples with vectors. Integers are mapped directly, but the mapping is partial because Emacs Lisp integers are only 27 bits (Emacs has no bignums.) PIDs, Ports, and References are mapped onto vector-based structures, and tagged with a special uninterned symbol[1] to distinguish them from the vectors used for tuples.

Mapping strings from Erlang to Emacs Lisp is troublesome. The Erlang binary term encoding includes a string type, but it is used loosely – you never know whether an Erlang string will be encoded as a string or as a list of integers. To sidestep the problem, Erlang binaries are mapped onto Emacs strings, and we always use binaries to reliably send text to Emacs. Emacs Lisp strings are mapped onto Erlang strings.

Other types, such as floats and functions, are not yet mapped, and attempting to send them triggers an error.

4 Pattern Matching

Distel has three pattern matching macros, one being `erl-receive`, which has already been introduced. Each macro uses the same pattern syntax, described below.

(`mlet` *pattern object body...*) matches *object* with *pattern*, and on success executes the *body* forms with all pattern variables bound. If the match fails, an error is signalled. `mlet` is similar to Erlang's = operator.

(`mcase` *object clauses*) matches an object with a series of clauses, where the syntax of each clause is (*pattern body...*). The first clause whose *pattern* successfully matches is selected, and its *body* forms are then executed with all pattern variables bound. If no clause matches, an error is signalled. `mcase` is of course based on Erlang's `case` expressions.

4.1 The Pattern Syntax

The pattern syntax is very similar to Erlang, though it lacks guards in the current implementation. The syntax is specified below, and followed by some examples.

Trivial: `t, nil, [], 42, ...`

Constants, matched literally.

[1] An uninterned symbol in Emacs Lisp is like a `ref` in Erlang, but it looks like a symbol.

Sequence: (pat1 ...), [pat1 ...]

Sequence patterns match the "shape" of the sequence, as well as each individual subpattern. The pattern can be either a list or a vector, and will only match a sequence of the same type.

Pattern variable: `var, my-variable, ...`

Symbols denote variables that the pattern should bind. The first time a particular variable is used it binds to the corresponding value, and then further occurrences must match this bound value.

Following a successful pattern match, a Lisp variable is bound for each pattern variable.

Constant: `'symbol, '(x y z)`

Quoted constants are matched literally by value.

Bound variable: ,var

The pattern `,var` matches the value of the pre-bound Lisp variable `var`. This is like using an already bound variable in a pattern in Erlang.

Wild card: _ (underscore)

Matches anything, with no binding.

For example, the Erlang code:

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

could be written in Lisp as:

```
(mcase result
  (['ok value]      value)
  (['error reason] (erl-exit reason)))
```

and similarly,

```
{ok, Value} = Result,
Value
```

could be written as:

```
(mlet ['ok value] result
  value)
```

5 A Process Manager

This section describes the design and implementation of a small but complete process-manager application. The program does two things: it presents a list of the processes running on an Erlang node, and it provides some commands to operate on them. The process list is shown in an Emacs buffer, with a one-line summary for each process. The summary line shows the PID, registered name (if any), number of reductions, and number of unreceived messages, as shown in Figure 3.

The first step in designing the application is to divide up the work between Emacs and Erlang, and decide how they will interact. The goals are to do the work on the side that makes it the easiest, and to keep the program simple by minimising the interactions.

The task for the Erlang side of the process manager is to create formatted summaries of all the processes in the system, ready for Emacs to display. The Emacs side then must fetch a process list, display it in a buffer, and provide some commands for operating on the processes. The interactions are driven from Emacs, using RPCs to the `rex` server (mentioned in Section 3.)

5.1 The Erlang Side

The Erlang side is implemented by the `procman` module of Figure 4, which exports the function `process_list/0`. This function returns the PID and a one-line summary of each process in the node, plus an extra line containing column headings to match the summary lines. Note that all the text is returned as binaries, to avoid the problem with strings discussed in Section 3.

```
Pid      Name               Reds      Msgs
<0.0.0>  init               3836      0
<0.2.0>  erl_prim_loader    45203     0
<0.4.0>  error_logger       245       0
<0.5.0>  application_contr  2414      0
<0.7.0>  <none>             59        0
```

Figure 3: Process Manager "screenshot"

```
-module(procman).

-export([process_list/0]).

%% Returns: {ok, Header, [ProcessInfo]}
%% ProcessInfo = {Pid, Summary}
%% Header = Summary = binary()
%%
%% Returns a one-line summary of each
%% running process along with its pid,
%% plus a heading that matches the
%% summary format.
process_list() ->
    {ok,
     fmt_row("Pid", "Name", "Reds", "Msgs"),
     [{P, info(P)} || P <- processes()]}.

info(Pid) ->
    PidName = pid_to_list(Pid),
    Reg = item(Pid, registered_name),
    Reds = item(Pid, reductions),
    Msgs = item(Pid, message_queue_len),
    fmt_row(PidName, Reg, Reds, Msgs).

item(Pid, Item) ->
    case process_info(Pid, Item) of
        {Item, Value} -> to_string(Value);
        []            -> "<none>"
    end.

fmt_row(A,B,C,D) ->
    list_to_binary(
      io_lib:format("~-8s ~-17s ~-10s ~s~n",
                    [A,B,C,D])).

to_string(X) ->
    io_lib:format("~p", [X]).
```

Figure 4: Erlang side of process manager

```
(defun pman (node)
  "Show a list of all processes on NODE."
  (interactive (list (erl-read-nodename)))
  (erl-spawn
      (display-buffer (current-buffer))
      (erl-send-rpc
       node 'procman 'process_list '())
      (erl-receive ()
          ((['rex ['ok header plist]]
            (pman-insert header plist)
            (erl-idle))
           (['rex ['badrpc reason]]
            (message "RPC failed: %S"
                     reason))))))

(defun pman-insert (header plist)
  "Insert all process information.
  PLIST is a list of [PID Summary]."
  (insert header)
  (dolist (pinfo plist)
    (mlet [pid text] pinfo
      (insert
       (propertize text
                   'pid pid)))))
```

Figure 5: Emacs pman process

5.2 The Emacs Lisp Side

The job for the Emacs program is to call `process_list/0` on some Erlang node and present the result. It must also record an association between summary text in a buffer and the PID of the process it represents, so that later we can write commands to operate on the process represented by a particular line of text. The code for the Emacs process is given in Figure 5.

The command `pman` creates an Emacs process and uses it to display the process list. The command takes one parameter, the Erlang node to summarise. The `interactive` declaration says that when the command is called interactively (by a key binding or `M-x`), `erl-read-nodename` is called to choose the node. This function is predefined, and will either prompt the user for a node or reuse the most recently chosen one from a cache.

The body of the function is wrapped in an `erl-spawn`, so it runs in a new process. Because an Emacs process has its own buffer, we use `display-buffer` to show it on the screen directly.

Next, the process sends an RPC to the Erlang node to call `procman:process_list()`. The predefined `erl-send-rpc` function is similar to `rpc:call/4` in Erlang, its parameters are *node*, *module*, *function*, and *arguments*. The RPC server sends back the result in a `{rex, Result}` message, so we have an `erl-receive` with two patterns: one to receive the summary information on success, and one to handle any error on the Erlang side (for example, the `procman` module not being available.) If the summary arrives successfully, it is inserted into the buffer, and then the process calls `erl-idle` to enter an idle loop. The idle loop is like a receive with no patterns, meaning "schedule out indefinitely." If we had just returned without entering a receive, the process would terminate with reason `normal` and the user-interface buffer would be killed.

The `pman-insert` function takes the data we got from Erlang and puts it into the buffer for display. The header line is inserted at the top, then each summary is destructured with the `mlet` pattern matching macro and inserted. To preserve the association between the summary text and the process it represents, we use an Emacs feature called "text properties," which allows text in strings and buffers to be tagged with arbitrary key/value properties. The call to `propertize` tags the summary line with a `pid` property, so that later we can use `get-text-property` to look up the PID belonging to a piece of text in the buffer.

The process summary part is now complete, and running "`M-x pman`" will display a summary buffer as we showed in Figure 3.

What remains is to define a way to do things with the processes. Figure 6 shows a command to kill the process on the current line. It finds out which process we want to kill by calling `get-pid-at-point`, which looks up our `pid` property at the current location in the buffer (i.e. where the cursor is). Then it sends the process an exit signal with reason `kill` via the built-in `erl-exit` function, which is equivalent to `erlang:exit/2`.

```
(defun pman-kill ()
  "Kill the process under the cursor."
  (interactive)
  ;; send an EXIT signal to the process
  (erl-exit 'kill (get-pid-at-point)))

(defun get-pid-at-point ()
  "PID of the process at the point."
  (or (get-text-property (point) 'pid)
      (error "No process at point")))
```

Figure 6: Emacs kill process command

A command for displaying a process backtrace is shown in Figure 7. This is more involved than killing a process, because we must send a request for the backtrace and then receive and display the reply asynchronously. We achieve this by spawning a new process to request the backtrace, and then display the result in its own buffer when the reply arrives.

Before spawning the new process, we look up the PID that we want a backtrace for. We do this first because the code inside the `erl-spawn` will run in the new process' buffer, and the lookup has to be done in the buffer that has the process list. Next the new process is spawned, and uses `pop-to-buffer` to make its own buffer visible somewhere on the screen.

The process then makes an RPC to `erlang:process_info(Pid, backtrace)`. The return type is `{backtrace, BacktraceBinary}`, which is very convenient for our purposes, since the binary will be received as a string. When the result arrives, we simply insert the backtrace text into the buffer, and enter an idle loop.

5.3 Summary

This `procman` application, though simple, is complete and useful. The approach to design used here is a good one: minimise the interactions, and do things where they are easiest. It is often best for Erlang to spoon feed Emacs, just as the `procman:process_list/0` function returns a structure that is trivial for Emacs to display.

```
(defun pman-backtrace ()
  "Show backtrace of process at cursor.
  The backtrace pops up in a buffer."
  (interactive)
  (let ((pid (get-pid-at-point)))
    (erl-spawn
      (pop-to-buffer (current-buffer))
      (send-backtrace-rpc pid)
      (erl-receive ()
          ((['rex ['backtrace text]]
            (insert text)
            (erl-idle))
           (['rex ['badrpc reason]]
            (message "RPC failed: %S"
                     reason)))))))

(defun send-backtrace-rpc (pid)
  "Send an RPC for the backtrace of PID."
  (erl-send-rpc (erl-pid-node pid)
                'erlang
                'process_info
                (list pid 'backtrace)))
```

Figure 7: Emacs "backtrace" command

6 Runtime System

The Distel runtime system creates and schedules processes, delivers their messages, cleans up after their errors, and communicates with other nodes on the network. This section sketches the gory details of the implementation, and is not required reading for the rest of the paper.

6.1 Processes and Scheduling

An Emacs Lisp process is represented as an Emacs buffer, with all of its identity and state stored in buffer-local variables. The actual variables we use are `erl-self` (the PID), `erl-mailbox`, `erl-links`, and so on. There are also some cute mappings of process mechanics onto Emacs buffers, for example the `kill-buffer-hook` is used to propagate exit signals, and registered names are implemented with buffer names of "`*reg` *name*`*`". Note that because all process state is stored in buffer-local variables, context-switching just means changing buffers.

While a process is scheduled out, its state also includes a *continuation function* that can be called to resume execution from where it left off. We only ever schedule a process out when it blocks to wait for a message, so the continuations are created by `erl-receive`. The extra arguments that `erl-receive` requires reflect the difficulty of capturing the control state in Emacs Lisp, which lacks lexical closures and first-class continuations.

Each time a new process is spawned, or a message arrives from the network, the scheduler loops by invoking processes one at a time until they have all terminated or blocked in a receive. The scheduler invokes a process by switching to its buffer and then calling the continuation function, which does what it does and then either throws back a new continuation via `erl-receive`, raises an error, or simply returns. If it returns a new continuation then the process is scheduled out until a new message arrives, otherwise it is terminated by setting an `erl-exit-reason` variable and then killing its buffer (which propagates an exit signal via `kill-buffer-hook`.) This simple scheduler is based on a technique called Trampolined Style [8].

While a process is scheduled in and running, it can call BIFs to send messages and to do other process-related things. The semantics of BIFs are based on the Erlang 4.7 specification [9], and their implementation is very simple, averaging about 5 lines of code each. For example, when `(erl-send` P M`)` is called, it either passes the request to the distribution module (if P is remote), or just switches into P's buffer, adds M to the end of `erl-mailbox`, and marks the process as schedulable. Similarly, if process P calls `(erl-link` Q`)`, then Q is added to the `erl-links` list of P, and either the same is done with Q or the request is handed off to distribution, depending on whether Q is local.

6.2 Network Distribution

Distribution over the network is built from three modules: a library for binary encoding, a framework for writing network-attached state ma-

chines, and the state machine for the Erlang distribution protocol [7]. The binary coding library is a straightforward implementation of the Erlang external term format [10] using the mapping from Section 3. The networking framework supports writing simple state machines and attaching them to TCP sockets, with the crucial property of being purely event-driven and using non-blocking I/O. It is necessary that all I/O be done asynchronously, to avoid freezing Emacs while a background task waits on I/O – an often lamented property of many other Emacs networking programs.

The Distributed Erlang state machine first authenticates itself and negotiates features, and then serves requests bidirectionally. The implementation is straightforward because the distribution protocol is very high level – each message maps neatly onto a BIF. The messages implemented in Distel are:

- SEND(*PID*, *MSG*)
- LINK(*FROM*, *TO*)
- UNLINK(*FROM*, *TO*)
- EXIT(*FROM*, *TO*, *REASON*)
- REG_SEND(*FROM*, *NAME*, *MSG*)
  Send a message addressed by registered name. The PID of the sender is included so that an EXIT signal can be sent back if no such name is registered.

When a request arrives from another node, the arguments are decoded and the corresponding BIF is called. Similarly, when an Emacs BIF is called with a remote process, the request is encoded and forwarded to the node where the process is running – perhaps first being queued while a TCP connection is established.

Optional extensions, such as process monitoring, have not yet been implemented.

7 Applications

The Distel software distribution includes a variety of applications and tools for Erlang development. These tools are unified with a minor mode called the `erlang-extended-mode`, which complements the standard `erlang-mode`. The major features are described below, along with their commands and key bindings.

7.1 Dynamic "TAGS"

Distel includes a small source code cross-referencer for Erlang. The basic feature is to jump from a function call in a program to the definition of that function – for instance from the text `lists:sort(L)` to the definition of `sort/1` in `lists.erl`. The feature is similar to `etags` [3], but uses an Erlang node to dynamically find the right source files, instead of a statically generated database. The advantage is that running an Erlang node is a lot easier than maintaining a TAGS file, so the feature can be used all the time.

erl-find-source-under-point (`M-.`)

Jump to a function definition. The definition will be chosen from the text at the point – either a function call, or declaration in an export list.

erl-find-source-unwind (`M-*`)

Jump back from a function definition. This is a multi-level way to backtrack after following a chain of function definitions.

7.2 Debugger

An Erlang debugger interface, called `edb`, is also included with Distel. This uses the same interpreter-based back-end as the OTP `debugger` application, but replaces the `Tk`-based front-end with an Emacs interface. Erlang mode buffers can use `edb` commands to toggle debug-interpretation of a file, toggle a breakpoint on a line, and to pop up a "monitor buffer" to view and control debugged processes.

The monitor buffer shows all processes running debugged code, and lets you "attach" to any process that is stopped in a breakpoint. Attaching to a process pops up a buffer containing the source code of the process's current module, with a visual marker pointing to the current line. From this buffer the process can be single-stepped, its local variables can be inspected, and so on.

edb-toggle-interpret (`C-c C-d i`)

Toggle debug-interpretation of the current file.

edb-toggle-breakpoint (`C-c C-d b`)

Toggle a breakpoint on the current line.

edb-monitor (`C-c C-d m`)

Popup the debugger monitor buffer.

7.3 Process Manager

Distel includes a process manager based on the OTP `pman` application. This program is like the `procman` example of Section 5, but more polished: it uses a major mode for key bindings, and supports tracing process events via the `trace` BIF.

erl-process-list (`C-c C-d l`)

Pop up a process manager buffer.

7.4 Profiler

A front-end to the OTP `fprof` profiler is included. The `fprof` command prompts for an Erlang expression to profile, executes it with profiling on an Erlang node, and presents the results in an Emacs buffer. The result summary shows the time spent in each Erlang function, and can "zoom in" on each function to show its callers and callees.

fprof (`C-c C-d p`)

Profile an Erlang expression from the minibuffer.

7.5 Dilber: The disk_log Viewer

Dilber is a viewer for Erlang `disk_log` files, in the spirit of Unix `tail`. It is also the first "third party" Distel application – written by Vladimir Sekissov, and in on-going use as a system administration tool.

Dilber will be included in a future release of Distel.

7.6 Interactive Sessions

An Interactive Session buffer is to Erlang as the `*scratch*` buffer is to Emacs Lisp – a scratchpad where code snippets can be hacked and executed. The advantages over the Erlang shell are that session buffers are random-access, and that local Erlang functions can be defined individually in the buffer. This is especially useful for playing with code snippets for the `erlang-questions` mailing list – you can try Erlang functions without creating and compiling a real source file.

Interactive session buffers were conceived and implemented by David Wallin, and are included in the Distel distribution.

erl-ie-show-session (`C-c C-d s`)

Pop up a session buffer, creating it if necessary.

erl-ie-copy-buffer-to-session (`C-c C-d c`)

Create a session buffer, and copy the contents of the current buffer into it.

erl-ie-copy-region-to-session (`C-c C-d r`)

Create a session buffer, and copy the contents of the region into it.

7.7 Miscellany

erl-eval-expression (`C-c C-d :`)

Evaluate an Erlang expression from the minibuffer.

erl-reload-module (`C-d C-d L`)

Reload an Erlang module, given by name in the minibuffer.

8 History

Distel represents the evolution of several attempts at using Emacs as a user interface for Erlang. The first was "`erlext.el`", which began as an implementation of the Erlang external term format and was later extended with TCP socket communication. The drawback of this approach is that it needs a special TCP server to run in the Erlang node, which turned out to be too much of an obstacle for spontaneous use.

This was followed by Ermacs,[2] a concurrent Emacs clone written completely in Erlang. Ermacs is fairly complete – it has major modes for Erlang and Scheme programming, a built-in Erlang shell, and support for efficiently editing large files. However, once the core editor was complete, it was obvious that GNU Emacs has an *incredibly large* set of wonderful features, and that extending Ermacs to include "enough" of them was completely out of the question.

The lessons learned from Ermacs lead to Distel, which continues where `erlext` left off. Version 1.0 replaced `erlext`'s custom socket protocol with the Erlang distribution protocol, added very basic Emacs Lisp processes, and included a small process manager application. Version 2.0 greatly improved the programming interface with `erl-receive` and pattern matching, which made it possible for later versions to include the substantial collection of Erlang development tools available today.

9 Implementation Status

Distel is a stable piece of software, compatible with all recent versions of GNU Emacs and XEmacs, and suitable as an Erlang development tool without additional programming. The implementation is free software, with development hosted on SourceForge[3], and source code and documentation available on the Distel homepage:

`http://distel.sourceforge.net/`

At the time of writing, the implementation is 3,714 lines of Emacs Lisp and 994 lines of Erlang. It breaks down as follows:

- 608 lines of Emacs Lisp for the scheduler, BIFs, and process representation.
- 1,231 lines of Emacs Lisp for the distribution protocol (264 for networking, 395 for encoding and decoding, 99 for the port mapper (`epmd`) client, and 473 for the distribution protocol.)
- 1,489 lines of Emacs Lisp for the `erlang-extended-mode` (544 for the debugger, 200 for interactive sessions, and no clear division for the remainder.) All of the Erlang code is used for supporting the `erlang-extended-mode`, Distel's core doesn't require any.

The rest is made up of random examples and test suites.

10 Future Directions

Distel development is focused on the `erlang-extended-mode` and related tools, with language and runtime system extensions being made as they are needed. The plan is to continue adding new applications and extending Distel's capabilities as an integrated Erlang development environment. It would also be desirable to merge the useful features of Distel that don't require the runtime system into the standard (and *wonderful*) `erlang-mode`.

Using Distel for general Emacs-to-Emacs concurrent and distributed programming is another exciting possibility. Today this would require only an implementation of the port mapper (`epmd`) and for Emacs to listen for incoming connections,[4] though it may be preferable to use a completely different communications layer.

11 Related Work

The three main types of related work are Erlang distribution libraries for other languages, the `Etos` compiler, and other Emacs-based integrated development environments (IDEs).

Just like Distel has "Emacs nodes," the OTP applications `erl_interface` and Jive have C and Java nodes respectively. David Schere's "Erlang-Python"[5] implements Python nodes, using a binding to `erl_interface`. Others implementations may well also exist.

`Etos` [4] is an Erlang to Scheme compiler, which is related to Distel in that they both imple-

[2] http://www.bluetail.com/~luke/ermacs/
[3] http://www.sourceforge.net/

[4] At the time of writing, this seems to only be possible with the CVS version of GNU Emacs, or with an external helper program to bind the listen socket.
[5] http://starship.python.net/crew/gandalf/PyErlang/

ment high-level Erlang runtime systems in Lisp dialects. `Etos` was a good source of inspiration, and anyone who studies Distel owes it to them self to see how much more neatly things can be done with first-class continuations.

Two popular and mature Emacs-based IDEs are the Java Development Environment for Emacs (JDEE)[6], and ILISP [11] for Lisp. We hope that Distel will fill a similar niche for Erlang programmers.

Anders Lindgren's "Erl'em" program is said to have been similar in scope and purpose to Distel, but appears to have been swept away in the winds of time.[7] Anders is the main author of the Emacs `erlang-mode`.

12 Conclusion

We have extended Emacs Lisp for concurrent and distributed programming, and applied the extension to developing Erlang development tools. This has been a practical endeavour, and the resulting tools are immediately available to all Erlang programmers who use Emacs, as is a familiar programming interface for writing more tools.

We have also further demonstrated the power and flexibility of Emacs. Several Distel applications are highly concurrent, particularly the `edb` debugger which monitors and controls multiple processes as they run, without interfering with the user's editing. The ease with which these applications are written suggests that Emacs Lisp is very easily extended into a powerful concurrent and distributed programming system – in this case using Erlang's model, but it is easy to envision others.

Is there anything Emacs can't do?

13 Acknowledgements

I would like to thank Vladimir Sekissov, David Wallin, and Mats Cronqvist for their Distel hacking; Darius Bacon and Martin Björklund for their help with Distel's design and invaluable reviews of drafts of this paper (usual disclaimer applies); and all the colleagues and `erlang-questions` readers who have installed Distel and helped to iron out the (many) teething problems.

[6] http://jdee.sunsite.dk

[7] If you have a copy of this that you are allowed to distribute, please get in touch with me.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.

[2] The open source erlang website. `http://www.erlang.org/`.

[3] Bill Lewis, Dan LaLiberte, and Richard Stallman. *The GNU Emacs Lisp Reference Manual.* Free Software Foundation.

[4] Marc Feeley. Etos: an erlang to scheme compiler. August 1997.

[5] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge, MA, 1992.

[6] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, MIT AI Lab, December 1975.

[7] Erlang distribution protocol. Described in a text file included with the Erlang/OTP source distribution, under `lib/kernel/internal_doc/`.

[8] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

[9] Jonas Barklund and Robert Virding. Erlang 4.7.3 reference manual. Draft (0.7), February 1999.

[10] The erlang extended term format. Described in a text file included with the Erlang/OTP source distribution, under `erts/emulator/internal_doc/`.

[11] Todd Kaufmann, Chris McConnell, Ivan Vazquez, Marco Antoniotti, Rick Campbell, and Paolo Amoroso. Ilisp user manual.

Static analysis of communications for Erlang

Fabien Dagnat
Laboratoire Informatique des Télécommunication
ENST de Bretagne, Technopôle Brest Iroise, BP 832
29285 Brest, France
Fabien.Dagnat@enst-bretagne.fr

Marc Pantel
Institut de Recherche en Informatique de Toulouse
LIMA / ENSEEIHT, 2 rue Camichel
31071 Toulouse, France
Marc.Pantel@enseeiht.fr

ABSTRACT

In this paper, we present an insight of the two major contributions of works made to build a static analyzer of ERLANG programs. First, we introduce a general framework based on a process calculus (the *configurations*). This formalism describes concurrent aspects and abstracts functional ones. Obtaining the ERLANG semantics is then just instantiating this framework with an adequate functional setting. The second contribution is a sophisticated type system for ERLANG. This type system infers types and subtyping constraints for a program and ensures that the collected constraints have at least one solution. This system detects usual functional errors but also some of the communication errors. More precisely, for each process, it cumulates all received messages and all handled messages and ensures that the first is included in the second. To do this, it borrows concepts to the object (or record) usual typing in ML.

1. INTRODUCTION

The development of telecommunications industry and the generalization of network use bring concurrent, distributed and mobile computing into the limelight. In that context, programming is a hard task and, generally, the resulting applications contain many more *bugs* than usual sequential centralized software. Indeed, the indeterminism resulting from the unreliability of networks and the size of the code of such applications makes it difficult to validate any distributed functionality using informal approaches. Our work focuses on using static analysis, a kind of formal methods to ease development.

As Erlang software are mainly used in telecommunication equipment that do not tolerate failure, their development must be certified. More precisely every step toward the final application must be *validated* (ideally automatically). Our aim is to participate to this hard task, by building static analysis of communications using type inference techniques.

To give an abstract model to ERLANG programs, we use the actor model developed by Agha in [1]. It is based on a network of autonomous and cooperative agents (called actors and similar to ERLANG processes), which *encapsulate* data and programs. They communicate using an *asynchronous point to point* protocol and store each received message in a mailbox. When idle, an actor handles the first message it can in its mailbox. Besides those conventions (which are also true for concurrent objects), an actor can dynamically (at run-time) change its interface. This property allows to modify the set of messages an actor can handle, yielding a more accurate and widely usable programming model. For example, it can give an abstract model to applets and dynamic code loading.

In a first approach, we defined type systems for the CAP calculus described in [8], a primitive actor calculus derived from asynchronous π-calculus and Cardelli's Calculus of Primitive Objects. Two type systems were developed. The first one [9], based on usual object type abstractions, catches all usual functional and communication errors (erroneous parameters) but only a subset of messages which will never be handled. The second [7], detects all (safety) messages not understood but requires a much more complex type abstraction and a new programming discipline. These systems were proved to be correct. In order to validate their practical use, the need for a programming language implementation arose. In a first approach, we developed a lab language ML-ACT integrating *à la ML* programming with actor primitives and including a sophisticated type system extending the previous work on CAP (see [11]). Then, we studied ERLANG, as it appears that, thought its functional aspects have a strongly different semantics (and typing) than ML-ACT one's, their concurrent semantics and typing were similar. Therefore, we developed a framework abstracting the parts of both languages having semantics (and typing) differences (for example, functional aspects or mailbox semantics). It became possible to build systematically the semantics, the typing and some properties about the typing, once provided the functional setting. Furthermore, this functional setting can use a well known classical one. For example, ML-ACT use the ML functional semantics and typing.

This article gives an introduction to this abstraction and its application to ERLANG. The first section provides a better insight of the form of communication errors we wish to detect and the ones our system captures. Then, we introduce a simplified version of ERLANG and its formal semantics based on configurations, an asynchronous π-calculus like process algebra. Then, we define our type system and illustrate its use on examples. Finally, we discuss scaling this system to the full language and some possible extensions to our work.

2. COMMUNICATION ERRORS

In an usual concurrent setting, a process `P` may receive a message `m` (`P ! m`, in ERLANG). Supposing `P` is idle, there are two possibilities, either `P` can handle `m` or it cannot. Our works focus on the early detection of requests that may not

be handled (the second case). This problem is related to the *method not understood* errors of object oriented programming. In the actor context, a message that may not be understood by its receiver is called an *orphan*.

Typed object oriented languages determine the set of methods an object P understands (**typeof**(P)) and ensures that each method invocation P.m is correct by verifying that m is part of the type of P (m ∈ **typeof**(P)). Furthermore, as the type of an object does not change, the verification can be done when the method is invoked. Adapting this technic to ERLANG (P becoming a process and P.m becoming P!m) raises two problems leading to a much more complex typing: a) the computation of the set of messages a process can handle is dynamic and more complex and b) as the time between sending a message and its reception by its target may be important (the message may travel through large networks), the verification must be done upon reception.

The usual approach for actor languages is to **dynamically** check for *message not understood* errors. A process knows the messages it can (immediately) handle and if a received message does not conform to this interface, it raises a message not understood error (see the initial actor model [1] or the Vasconcelos and Tokoro object calculus [26]). But this approach reduces consequently the set of programs that one may build. In fact, the programmer must adopt a sort of synchronous programming discipline to be sure that messages arrive in *right* states. We think that this strategy is too restrictive. For example, consider a printer device that has two states: `working` (it accepts printing requests) and `stopped` (it waits for initialization). A client must wait that an initialization message has been sent to the printer before printing. It would be much more flexible to enqueue all requests received when the printer is `stopped` and to process all pending requests when it is initialized (possibly independently by another process) which is the usual behavior of unices print spoolers.

The second and opposite approach never rejects a message. When a process receives a message that it cannot handle, it silently enqueues it. Notice that, in this context, a message may stay indefinitely in a mailbox (their size is unbound). This semantics has been chosen by the blue calculus [4], the join calculus [14] and ERLANG.

We believe that a combination of both approaches may be much more appropriate. Such a system would reject programs that contains *message **never** understood* and would accept all other messages warning the programmer that they may never be handled. To achieve this goal, we use a powerful behavioral[1] type system to enforce the rejection of such messages. Our type system detects all messages that are not in the set of messages the receiver may handle during its execution. This means that **typeof**(P) cumulates all the `receive` that P could execute. To do this the system must follow the flow of functions called by P. It is clear that, in general, our analysis will answer ⊤ (top) to express the fact that a process may assume an externally defined receive and therefore understands virtually everything. But, we think that the results are generally already helpful and we are working on extending our techniques to those open programs as will be discussed later.

[1]By opposition with a more usual class name type system as in C++ or Java.

For example, a process P executing the first function of the program below (`ping`) has a type containing `ping`, `change` and all messages accepted by all possible behaviors F. This means that sending a message `{change, pong}` to P adds `pong` to the type of P.

```
ping() -> receive ping -> ping();
                  {change, F} -> apply(F,[])
          end.
pong() -> receive pong -> pong() end.
```

3. A SIMPLIFIED VERSION OF ERLANG

Following a common use in the definition of static and dynamic semantics, we simplify the ERLANG language by suppressing syntactic sugar and ignoring constructions that are typed orthogonally to our work (for example, exceptions, lists or records). Furthermore, we do not address the semantics of the real time part of the language which is complex but do not add any specific problem to the type system. An effort has been made to define precisely a small (but still too big) language named CORE ERLANG ([5] or [6]). Therefore, we use a smaller version of the language named μErlang:

$$
\begin{array}{lll}
prg & ::= & c;\ldots;c. \;\mid\; c;\ldots;c.\ prg \\
c & ::= & s(p,\ldots,p)\ \texttt{->}\ e \\
p & ::= & \_ \mid V \mid s \mid i \mid \{p,\ldots,p\} \\
e & ::= & V \mid s \mid i \mid \{e,\ldots,e\} \mid (e) \mid e,e \mid e!e \\
 & & \mid e(e,\ldots,e) \mid \texttt{case}\ e\ \texttt{of}\ f\ \texttt{end} \mid \texttt{receive}\ f\ \texttt{end} \\
f & ::= & p\ \texttt{->}\ e \;\mid\; p\ \texttt{->}\ e;f
\end{array}
$$

A μErlang program is a set of function definitions including a function named `main`. This main function is launched to start the execution of the program. The rest of the language is very close to ERLANG. Each function is composed of clauses separated by semi-colons and terminated by a dot. All clauses ($s(p,\ldots,p)$ -> e) must refer to the same function name s and have the same arity. Notice that this language does include guards to simplify the semantics and the type system for this paper. A pattern may be a joker (always succeeding), a variable V (always succeeding and binding the variable[2]), an atom s, an integer i or a tuple. An expression may be any of those values and add parentheses, sequencing (,), message sending (!), function call, choice (`case`) and message handling operation (`receive`). The choice (resp. the receive operation) matches an expression (resp. the mailbox of the current process) using a set of filters composed of a pattern and an expression (f is named **interface**). Finally, some atoms represents built-in functions, as for example, `spawn` and `self`.

Notice that as CORE ERLANG, we adopt lexical scoping of variables to ease the presentation. Our prototype uses ERLANG strategy mixing dynamic and lexical scoping. Therefore, the real system uses systematically an input and an output environment for each expression. Again for sake of simplicity, μErlang does not include lists that are replaced in application and spawning by tuples.

[2]This is not true for ERLANG, but our system can easily adopt ERLANG policy.

4. FORMAL SEMANTICS OF ERLANG

Our work focuses on static analysis and more precisely on typing. In order to prove the correctness of our type system, we need a formal semantics of ERLANG. To our knowledge, few works have addressed such a hard task. Indeed, as ERLANG is a full fledge functional, concurrent, distributed and mobile language, its semantics is complex. Some efforts have been made to give an informal, but clear and systematic description of its semantics ([3] and [6]). But, this is not sufficient to build and prove some static verification system. It seems that only two papers ([12] and [15]) try to build such a formal semantics. These two papers define two *Labeled Transition System* that does not suit our need (proving the correctness of a type system). Inspired by those approaches and our previous works on semantics for actors, we built our own formal semantics by instanciating a general framework called *configurations* previously build on a lab language extending ML to actors (ML-ACT). This framework defines a general syntax for concurrent actions and abstracts (in the sense of taking as parameter) the functional part of the studied language. With this approach, we can reuse existing semantics and typing from the functional world. The μErlang semantics is obtained by instantiating this framework with an adequate functional semantics.

We are not going to give all the formal definitions and justifications of this model that may be found in [10]. We are only going to give insights on configurations to deduce the μErlang semantics. Most rules are given in appendix for the interested reader.

Configuration

A configuration is a term that represents a concurrent system at a given time. Its definition is parameterized by three sets : the **name** set $a \in \mathbb{A}$, the message set $m \in \mathcal{M}ess$ and the expression set $e \in \mathcal{E}xp$ with $\mathbb{A} \subset \mathcal{E}xp$ and $\mathcal{M}ess \subset \mathcal{E}xp$. The set of configurations noted W is built from the following grammar:

$$\begin{array}{lcl} w & ::= & \epsilon \mid \mathbf{Err} \mid \nu a.w \mid w \parallel w \mid a \lhd m \mid \alpha \rhd e \\ \alpha & ::= & \star \mid \langle a \,|\, \widetilde{m} \rangle \end{array}$$

A configuration looks like a π-calculus term with a send operation, noted $a \lhd m$ (a is the receiver and m the message), and a process, noted $\alpha \rhd e$ (α is the identity and e is the executed expression). The identity of a process is either unspecified $\star$ to model toplevel computations[3] or, $\langle a \,|\, \widetilde{m} \rangle$ a pair composed of a name (*pid* in ERLANG tradition) and a mailbox (the tilde notation denotes sequence). As it is usual in process calculi, we use a name binder ν to simulate the name creation and suppose that the corresponding notion of free names and substitution are defined.

In the context of μErlang, $\mathcal{E}xp$ represents the syntax introduced in the previous section, addresses are built automatically when the built-in function `spawn` is called and a message can be any value (atom, integer or tuple).

A congruence is defined to state which configurations are equivalents:

- $(W, \parallel, \epsilon)$ is a commutative monoid, the order of subconfigurations is not important and we can suppress all occurrence of ϵ.
- $w \parallel \mathbf{Err} \equiv \mathbf{Err}$ and $\alpha \rhd \mathbf{Err} \equiv \mathbf{Err}$, errors are propagated until the program evaluation stops.
- $\nu a.w \equiv w$ if a is not free in w, $\nu a.w \equiv \nu b.[b/a]w$ if b is not free in w and $\nu a_1.\nu a_2.w \equiv \nu a_2.\nu a_1.w$; those three usual properties allow to forget the bindings of unused names, to rename a bounded name and to modify the order of restrictions.
- the restriction rule, $\nu a.w_1 \parallel w_2 \equiv \nu a.(w_1 \parallel w_2)$ if a is not free in w_2, allows to enlarge the scoping of a name. Combined with the previous rule, it enables (up to a renaming of a in w_1) to extend the scoping and to simulate name propagation in the medium.
- $\star \rhd v \equiv \epsilon$ and $\nu a.(\langle a \,|\, \varnothing \rangle \rhd v) \equiv \epsilon$ if v is a value (it cannot be reduced) ; therefore, a global computation (or a process) which reduce to a value can be destroyed by a garbage collector. Notice that the process must have an empty mailbox and be inaccessible to the outside world.

Notice that it is possible to add a rule to express the fact that a stopped process waiting for a message, that do not understand any of its mailbox messages and is no more accessible from outside is an error. But, as our type system cannot capture all such messages (for example in a deadlock case), we cannot prove its correctness with this rule.

The appendix contains all the configuration reduction rules. Let us discuss only original rules.

As introduced in the second section of this paper, we try to detect communication errors. To define those errors more precisely, they are introduced in the semantics of configurations. Therefore, when a process receives a message, it can accept it (and put it in its mailbox) or reject it by raising an error:

$$\langle a \,|\, \widetilde{m} \rangle \rhd e \parallel a \lhd m \longrightarrow \begin{cases} \langle a \,|\, m\, \widetilde{m} \rangle \rhd e & \text{if } \mathcal{P}(m,e) \\ \mathbf{Err} & \text{else} \end{cases}$$

To abstract the choice of reaction, a (communication) *potential* $\mathcal{P}(m,e)$ is defined. This predicate approximates e to determine whether m may be understood or not. This allows the semantics of our framework to behave differently toward such messages. It is possible, for example, to code usual ERLANG semantics with a predicate always true. In the next section on typing, we will discuss more deeply this subject.

Our general semantics includes a rule to specify the interaction between functional and concurrent reduction:

$$\frac{a \notin \mathcal{FN}(\alpha \rhd e) \qquad a \vdash \alpha,\ e \xrightarrow{w}_e \alpha',\ e'}{\alpha \rhd e \longrightarrow \nu a.(\alpha' \rhd e' \parallel w)}$$

Where, we suppose that the functional reduction have the given shape with a being a fresh name ($a \notin \mathcal{FN}(\alpha \rhd e)$) that may be used during the expression evaluation and w being a configuration describing the concurrent effect of the functional reduction step. In the rest of the paper, if the label of such a reduction is ϵ, it is omitted. Notice that if a is unused, the third congruence rule enable to forget its binding.

[3]Those expressions cannot access `receive` or `self`.

Functional reduction

A μErlang program is a set of function definitions and its execution corresponds to the reduction of the body of the `main` function in a context where all the other functions are defined. By consequence, the first step of the functional semantics builds the function environment (noted $\mathcal{F}$). This process will not be described here, its result is an environment associating an atom and an arity to the body (all the pattern matching converted to a tuple matching) of the corresponding function. For example:

$$\begin{cases} \texttt{f}(p_1,p_2) \texttt{ -> } e_1\texttt{;} \\ \texttt{f}(p_3,p_4) \texttt{ -> } e_2\texttt{.} \end{cases} \quad \text{produces} \quad (\texttt{f},2) \mapsto \begin{cases} \{p_1,p_2\} \texttt{ -> } e_1\texttt{;} \\ \{p_3,p_4\} \texttt{ -> } e_2\texttt{.} \end{cases}$$

To simplify our presentation this set is abstracted and supposed to be accessible in all rules. This could be done by tagging each expression with this environment: $e_{\mathcal{F}}$ and by propagating it during reduction.

Functional reduction uses the classic notion of evaluation context. A context noted C[] is an expression with a hole marking the sub-expression subject of the current reduction step. The reduction $\mathrm{C}[e_1] \longrightarrow_e \mathrm{C}[e_2]$ reduce the expression e_1 and replace it by the result e_2. The evaluation context grammar is also given in the appendix, it expresses the fact that the order of evaluation is undefined when evaluating a tuple, a message sending or an application. On the contrary, evaluation of a sequence (resp. a choice) starts with the first expression (resp. the tested value). In addition we suppose that an error cause the end of the evaluation process: $\mathrm{C}[\mathbf{Err}] \triangleq \mathbf{Err}$.

Variables once defined have their values propagated by a substitution noted σ that we will not describe here. The matching operator / uses a function **match** to compare a pattern and a value and build the substitution of the variables in the pattern by their corresponding values. This function either returns a substitution or fails. It tries to match the first filter $p \rightarrow e$. If $\mathbf{match}(p,v)$ returns σ, / returns $\sigma(e)$. Else, if it did not matched, the process continue with the remaining filters. At the end, if none of the filter have matched, we get an error.

Purely functional evaluation is classic. The most original rules concerns application:

$$a \vdash \alpha,\ \mathrm{C}[v(v_1,...,v_n)] \longrightarrow_e \alpha,\ \begin{cases} \mathbf{Err} & \text{if } (v,n) \notin dom(\mathcal{F}) \\ \mathrm{C}[\{v_1,...,v_n\}/\mathcal{F}(v,n)] \end{cases}$$

The called function must be in the current function environment ($\mathcal{F}$). The result corresponds to the matching of its body with the tuple of actual arguments. This rule suppose that the expression describing the function must reduce to a valid atom and therefore, it extends slightly ERLANG semantics.

The functional actions that are connected with concurrent behavior have an original form and must be explained:

- Sending a message impose that the first argument is a name, returns the sent value and is labeled by the configuration sending term:

$$a \vdash \alpha,\ \mathrm{C}[v_1 \,!\, v_2] \xrightarrow{v_1 \triangleleft v_2}_e \alpha,\ \begin{cases} \mathbf{Err} & \text{if } v_1 \notin \mathbb{A} \\ \mathrm{C}[v_2] \end{cases}$$

- Spawning impose that its second argument is a tuple, returns the name (guaranteed to be fresh by concurrent reduction) of the future process and is labeled by the configuration describing the newly created process.This is only rules where the fresh name is used.

$$a \vdash \alpha,\ \mathrm{C}[\mathbf{spawn}(v,v_1,...,v_n)] \xrightarrow{\langle a\,|\varnothing\rangle \triangleright v(v_1,...,v_n)}_e \alpha,\ \mathrm{C}[a]$$

- A call to the built-in function `self` must be done in a process and is replaced by the name of the current process:

$$a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[a']$$

- Accessing the mailbox is similar to the choice except that the order of matching is different. The process try first to match each message with the first pattern and try next patterns only if none of the mailbox messages successfully matched the first pattern. For this we use a function **matchmailbox** that returns the resulting mailbox and the reaction. Notice that if the mailbox is empty no reduction can take place and by consequence the process is stopped (until a message reaches its mailbox).

$$a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{receive } f \texttt{ end}] \longrightarrow_e \langle a' \,|\, \widetilde{m}'\rangle,\ \mathrm{C}[e]$$
$$\text{where } \mathbf{matchmailbox}(f,\widetilde{m}) = \widetilde{m}',\ e$$

5. TYPING μErlang

When building a type system to statically detect errors in programs. The first thing to do is to define precisely what kind of errors, we want to avoid. In a concurrent setting, two families of errors arise: functional errors and concurrent errors. The former family is usual in the sequential world and correspond to the erroneous use of a value (for example, using an undefined variable or using 1 as a function). The latter is rather unusual and has been described in details in the section 2.

A type system can provide several level of precision. Two prototypes have already been built for ERLANG (see [17] and [16]) that concentrates on typing purely functional computation by simplifying the language semantics. Our ambition is to build a more useful system for ERLANG programs that also analyzes concurrent parts. As we use similar technics for collecting and solving constraints, our work may be considered as an extension of those systems.

Type inference and Constraints

Our system allows the synthesis of the types of every program entity without requiring any type annotation from the programmer. To do this, a fresh type variable is associated with each node of the syntactic tree of the program and constraints between those variables are collected. At the end of this collect phase, a resolution tool determines whether the constraint set has solutions. If this is the case, the program is declared well-typed. The schema of figure 1 describes this process.

To type functions and give them widely usable types, ML uses parametric polymorphism. For example, `map` has the type $\forall\alpha,\beta\ \ (\alpha \rightarrow \beta) \rightarrow \alpha\ \mathit{list} \rightarrow \beta\ \mathit{list}$ meaning that it can be used with any type α and β. We advocate that in

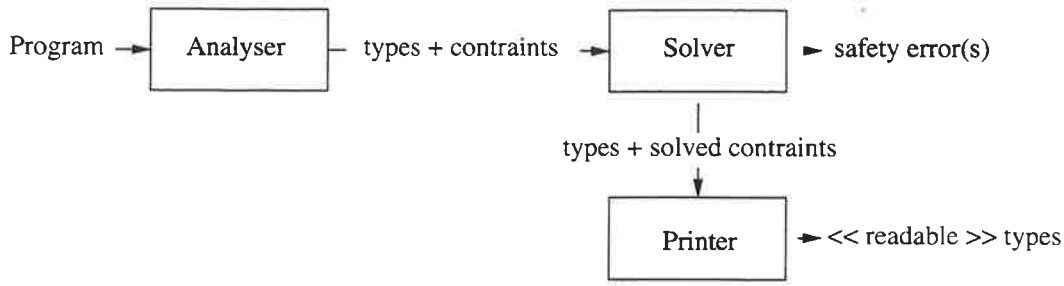


Figure 1: The analyzer schema

the concurrent context, this form of polymorphism becomes too restricting. Our system adopts inclusion polymorphism that intuitively means that the system ensures the correctness only for all values used in the program as real arguments (that is finite intersections rather than infinite ones). Therefore, in our context, we use the *subtyping* relation. A type t_1 being a subtype of a type t_2 ($t_1 \sqsubseteq t_2$) if a value of type t_1 may be used (safely) where a value of type t_2 is required. For ERLANG, the main use of subtyping is on process type: a process that understands more messages and sends itself less messages than another process, can replace this one. Typing an expression e under assumptions A will produce a type t and a subtyping constraint set C: $A \vdash e : t,\ C$, this deduction being valid only if C has at least one solution.

Notice that usual ML type system such as SML or Ocaml can be viewed as following the same process collecting equality constraints. But, when subtyping is needed (as for ERLANG), the constraints become complex and their resolution must use sophisticated and powerful graph algorithm. We refer the interested reader to the works of Pottier [19] or Fänhdrich [13]. Indeed, a constraint set is viewed as a graph where type variables are nodes (with their upper and lower bounds) and subtyping relation defines the edges.

The type of `map` becomes $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$ and each application with an argument of type t_1 and another of type t_2 produces the constraint set $\{t_1 \sqsubseteq \alpha \to \beta,\ t_2 \sqsubseteq \alpha\ list,\ \beta\ list \sqsubseteq t_r\}$ where t_r is the resulting type. This strategy collects all possible argument types and ensures that they can all be used safely:

$$\{\bigsqcup_i t_1^i \sqsubseteq \alpha \to \beta,\ \bigsqcup_i t_2^i \sqsubseteq \alpha\ list,\ \beta\ list \sqsubseteq \bigsqcap_i t_r^i\}$$

Potential and Errors

Before going on, let us look at the example below to precise some vocabulary:

```
state1(V) ->
  receive
    {add,V1} -> state1(V1 + V);
    {change,V1} -> state2(V,V1)
  end.
state2(V1,V2) ->
  receive
    {add,V3,V4} -> state2(V1 + V3, V2 + V4);
    {mute,F} -> F()
  end.
state3() ->
  receive
    kill -> true
  end.

main() ->
  case (spawn(state1,1)) of
    P -> P ! {add,1,3}, P ! kill,
         P ! {change,11}, P ! {mute,state3}
  end.
```

A function may contain two forms of interfaces (the filters f of a `receive` f `end`). One called *immediate* that is present in the body of the function or in the body of another called function ignoring received datas (in messages). And the second category corresponds to interfaces received via messages. This notion is extended to processes, the set of immediate interfaces of a process being the set of immediate interfaces of its initializing function. In the example, `state1` calls `state2` and itself and `state2` only calls itself. By consequences, the immediate interfaces set of `P` is:

$$\{\{\texttt{add,V1}\}\ \{\texttt{change,V1}\}\ \{\texttt{add,V3,V4}\}\ \{\texttt{mute,F}\}\}$$

The immediate interfaces may be viewed as the static automaton describing our process and the others as some dynamic part (in the exemple, `kill`).

Our type system captures all orphans that leads to error (in the semantics) using the potential introduced in the previous section. It is possible to give a predicate that collects all immediate interfaces (we refer the interested reader to [10]). Such a potential would approximates the previous set (keeping only labels) and would be defined by:

$$\mathcal{P}(m,e) \triangleq (\mathbf{label}(m) \in \{\texttt{add change mute}\}) \qquad (*)$$

Furthermore, as we do not want to raise an error and forbid the sending of the message `kill`, the potential of a processe calling a received function accepts anything. The real potential of `P` is then an *open potential*: $\mathcal{P}(m,e) \triangleq true$. In fact, the potential defined in $(*)$ would correspond to the same process if we change `state2`'s second filter body (the `mute` reaction) to any code not calling `F`.

Building the rules for such a system is already complex and does not capture all errors that our type system detects. Indeed, if in the example, we send a message `sub` to `P`, it is not rejected because the potential of `P` is opened. Building a more precise predicate (with respect to the captured errors) is hard and in fact corresponds to a slight simplification of the type inference. By consequence, we will not give precise

definition of the potential predicate and one can view it as a simplification of the type. Each atom sent in `mute` message is collected and its potential is added to the potential of P which becomes:

$$\mathcal{P}(m,e) \triangleq (\mathbf{label}(m) \in \{\texttt{add change mute kill}\})$$

The message `kill` is not declared orphan but the message `sub` causes a type error (it raises a dynamic error if not rejected).

We are currently devising a new definition of errors based on a dedicated arborescent temporal logic (see [25]). However, this approach currently only handle immediate interfaces.

Message and Process Types

An automatic analysis of the ERLANG compiler code, its standard libraries and programs freely available on internet[4] revealed that sent messages and receive interfaces are mainly tuples where one element is an atom. This atom plays the role of a label for messages. Furthermore rule 5.7 from [27] states that all messages should be tagged. Following the pioneer work of [17], we impose to all programs this precept. Notice that the only (less rare) exceptions are the use of jokers or variables to delegate the treatment of the message to a choice instruction or to another process. These two uses do not go against our precept since they just serve as forwarder. Finally, a program not following this principle may easily be adapted manually.

Those labels play a role similar to those of record label in ML or of method names in objects (for example). We borrow the *row* technology, used to type records, to approximate interfaces. Rows are now frequently used for static analysis in ML world (see for example, exception analysis [18] or object typing in Ocaml [20]). In our context, a process type is a row, which is a partial function from labels to pair of types describing arguments the message contains. The first one describes received messages content and the second handled messages content. Indeed, the originality of our types is the fact that they contain both received and handled messages in the type of a process. A process receiving messages labeled m_1 containing datas of type T_1 and handling it with values of type T_2 will have the following type: $@\{m_1 : (T_1, T_2),\ i\}$. The (row) variable i expresses the fact that the type of the process is only partially known. The conversion from a tuple type T to a message type $\widehat{T}$ (if it is sent) or $\overline{T}$ (if it is handled) is done in a lazy way and is defined in the appendix. Either the system knows the form of the type and converts it, or its structure is unknown and the system waits. A message reduced to an atom s has the type s and correspond to the message type $\{s : (unit, \top)\}$ or to $\{s : (\bot, unit)\}$. Meaning respectively that it is a sent message (the handling part is meaningless[5]) or a handled message (the received part is meaningless). The conversion of tuple message is similar. In the paper [17], the conversion was done for all tuples but we think that this is not really necessary. Back to our example, the process P has the following type if α and i are variables:

$$T_{\mathtt{P}} \triangleq @\{\texttt{add} : (1\times 3,\ int \sqcup (int \times int)),\ \texttt{change} : (11,\ int),\ \texttt{mute} : (\texttt{state3},\ T),\ \texttt{kill} : (unit,\ \alpha),\ i\}$$

Where T is the type of the function F taken as parameter. Notice that the unknown part i is related to the type T.

The correctness of the system is ensured by generating for each spawn process a fresh interface type i verifying $\diamond i$. This predicate is true if each received message is understood and is mathematically defined by:

$$\diamond\{m_i : (T_i, T_i')\}_{i\in I} \triangleq \forall i \in I\ \ T_i \sqsubseteq T_i'$$

Applied on previous type $T_{\mathtt{P}}$, we get:

$$\{1\times 3 \sqsubseteq int \sqcup (int \times int),\ 11 \sqsubseteq int,\ \texttt{state3} \sqsubseteq T,\ unit \sqsubseteq \alpha\}$$

We have not yet defined subtyping but intuitively, one can see that the two first constraints are trivial. The complete is discussed resolution after the presentation of types and subtyping.

Types and Subtyping

In ERLANG, one of the difficulties, is that being untyped, an expression may evaluate to values of really different structures (for example, a boolean and a function). Therefore, the type language must include a notion of union $t_1 \sqcup t_2$ meaning that a value of this type may be of type t_1 or t_2. Moreover to get sufficient precision, each constant has its own type (for example, 1 is of type 1 subtype of the integer *int*).

In ERLANG, any expression can execute a `receive` (*i.e*, access the mailbox of the current process). Therefore, the system use an indirect effect calculus inspired by [24] to collect, in the type of `self`, all interfaces matched against the mailbox. This effect is then included in the type of a function. When a process is spawned the effect of its initial function is added to the process type. In our example, `state3` has the following function type where the effect is the superscript of the arrow:

$$unit \xrightarrow{\{\texttt{kill}\,:\,(\bot,\,unit)\}} \texttt{true}$$

The language of types needed for μErlang is built by the following grammar:

$$\begin{array}{rcll}
T & ::= & \bot \mid \top \mid t \mid T \sqcup T \mid T \sqcap T & \\
 & \mid & i \mid int & \text{integers} \\
 & \mid & s \mid atom & \text{atoms} \\
 & \mid & unit \mid T\times\ldots\times T \mid tuple & \text{tuples} \\
 & \mid & T \xrightarrow{I} T & \text{functions} \\
 & \mid & @I & \text{processes} \\
I & ::= & \{\} \mid \top_I \mid i \mid \{m : (T,\ T),\ I\} & \text{interfaces type}
\end{array}$$

Subtyping is defined in the formula appendix, only three rules are unusual:

- Process types are contravariant because a process may replace another one only if its interface is larger, $@I \sqsubseteq @I'$ is equivalent to $I' \sqsubseteq I$.
- Function types are contravariant on arguments as usual and covariant on effect and on result. Indeed, if a function must replace another one, it must have a smaller

[4] This represent 200 000 code lines.

[5] The sens of the $\top$ or $\bot$ will become clear when subtyping will be defined. The intuition is that it is *nothing*.

concurrent effect: $T_1 \xrightarrow{I} T_2 \sqsubseteq T_1' \xrightarrow{I'} T_2' \iff T_1' \sqsubseteq T_1 \wedge I \sqsubseteq I' \wedge T_2 \sqsubseteq T_2'$

- Interface subtyping is covariant on received type, contravariant on handled type and compose covariantly.

$$\{m : (T_1, T_2),\ I\} \sqsubseteq \{m : (T_1', T_2'),\ I'\} \iff T_1 \sqsubseteq T_1' \wedge T_2' \sqsubseteq T_2 \wedge I \sqsubseteq I'$$

The intuition behind this rule is that the system must keep the largest type T_r of received messages and the lowest type T_u of handled messages. The correctness predicate $\diamond$ leads to $T_r \sqsubseteq T_u$ and any received content of type T is guaranteed to be understood by any receiver state T' because $T \sqsubseteq T_r \sqsubseteq T_u \sqsubseteq T'$.

Attentive readers may have remarked that the subtyping on interfaces is defined only for rows beginning by the same message label. A complete algebraic theory exists and proves that it is the only needed rule. If one label of the left side row is absent from right side row, the subtyping is clearly false and once all left side labels are treated, the system reduces to $\{\} \sqsubseteq I$ which is an axiom.

Another example

Before going into further discussion on this type system, consider a function that realizes a timer waiting for a message `cancel` or the end of a time specified at its creation to throw an alarm:

```
timer({Pid, Time, Alarm}) ->
    receive {cancel,Pid} -> true
    after Time -> Pid ! Alarm
    end.
```

A `timeout` function spawns such a timer process using the *pid* of the current process and returns the pid of the timer. The same process may cancel this timer using the returned *pid*:

```
timeout({Time, Alarm}) ->
    spawn(timer, {self(),Time,Alarm}).
cancel(Timer) ->
    Timer ! {cancel,self()}.
```

Supposing arguments of `after` (`Time`) are integers, our system infers:

$$\texttt{timer}: \quad \widehat{\alpha} \times int \times \alpha \xrightarrow{\{\texttt{cancel}:(\bot,\ @\widehat{\alpha})\}} \texttt{true} \sqcup \alpha$$

$$\texttt{timeout}: \quad int \times \alpha \xrightarrow{\widehat{\alpha}} @\{\texttt{cancel} : (\bot,\ @\widehat{\alpha})\}$$

$$\texttt{cancel}: \quad @\{\texttt{cancel} : (@\phi,\ \top)\} \xrightarrow{\phi} \texttt{cancel} \times @\phi$$

meaning that:

- The `timer` function takes three arguments: an address (receiving the third argument), an integer and a value (a message). The result is either `true` or this value and the current process receives a `cancel` message containing (an address of) a process that receives the third argument.
- `Alarm` (of type α) must be a legal message (tuple beginning by an atom).
- The process calling `timeout` receives the alarm (it appears in `timeout` effect).
- The result of this function is the name of a process understanding `cancel` messages containing an address that receives the alarm message.
- A call to `cancel` must includes an argument that receives a cancellation message containing the address of the current process and returns this cancellation message.

Those types are complex but very informative about the behavior of these functions. For example, the system can ensure that the *pid* returned by a call to `timeout` does not receive messages other than cancellation. It can also ensure that the process calling this function is able to receive the alarm message.

Functional Typing

Pattern matching cannot be treated in the usual ML way: $(\alpha_1 \to \beta_1) \sqcup (\alpha_2 \to \beta_2)$ cannot be equal to $(\alpha_1 \sqcap \alpha_2) \to (\beta_1 \sqcup \beta_2)$. In fact, the type system must include pattern matching, to do this [2] introduced the notion of conditional type $t_1?t_2$. This type means t_1 (if t_2 is different from $\bot$) or $\bot$. For example, if $e : t_e$, `case` e `of true -> 1; false -> foo` is of type $(int?(t_e \sqcap \texttt{true})) \sqcup (\texttt{foo}?(t_e \sqcap \texttt{false}))$. Our system does not use this conditional type which enjoys good algebraic properties but is not really readable and leads to the loss of the pattern matching structure. Instead, we use a conditional constraint $c_1 \Rightarrow c_2$ meaning that if c_1 is verified then the system must also ensure c_2. This constraint, generated to approximate pattern matching, allows to keep a high level of precision on the link between matched values and results. Typing previous choice lead to the following set of constraints: $C = \{t_e \sqsubseteq \texttt{true} \Rightarrow int \sqsubseteq t_r,\ t_e \sqsubseteq \texttt{false} \Rightarrow \texttt{foo} \sqsubseteq t_r,\ t_e \sqsubseteq \texttt{true} \sqcup \texttt{false}\}$ where t_r is the result type. Either the system knows the structure of t_e and C can be simplified, or it is decomposed in two sub-systems (because the matching is composed of two branches):

- One, in which, t_e is subtype of `true` and therefore $C = \{t_e \sqsubseteq \texttt{true},\ int \sqsubseteq t_r\}$
- Otherwise (due to third constraint), t_e is a subtype of `false` and $C = \{t_e \sqsubseteq \texttt{false},\ \texttt{foo} \sqsubseteq t_r\}$

As, in general, we do not know precisely the matched value, all those decomposed sub-systems must have a solution. This means that a n branch pattern matching fires the resolution of n sub-systems. However, the practice have shown that this is not a real problem. Indeed, when applying a pattern matching to a value, we often know more or less its structure and many of the sub-systems are trivial.

The typing judgments have the following shape:

$$\text{Environment} \vdash \text{Expression} : \text{Type}\ ,\ \text{ConstraintSet}$$

As, many typing rules are classic, we limit our explainations to sends, choices, receives and calls:

- Typing $e_1!e_2$ returns the second sub-expression type and the constraint set containing all constraints produced by the typing of e_1 and e_2, plus a constraint specifying that e_1 must evaluate to a process that receives the value of e_2:

$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \qquad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1!e_2 : t_2,\ C_1 \cup C_2 \cup \{t_1 \sqsubseteq @\widehat{t_2}\}}$$

- Typing a choice consists in typing the tested value and all patterns and associated expressions of the filter. A reaction expression must be typed after adding to the current environment the environment resulting from typing of the corresponding pattern:

$$\frac{\mathcal{E} \vdash e : t_e,\, C_e \qquad \mathcal{E} \vdash p_i : t_i^p,\, \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\, C_i}{\mathcal{E} \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \rightarrow e_1;\ \ldots : t,\ C_e \cup \bigcup_i C_i \cup C}$$

  where the resulting constraints cumulate all already calculated constraints and those due to the choice (C). C specifies that the tested value must be taken into account by one of the patterns and add all already explained conditional constraints (one for each branch):

$$C = \{t_e \sqsubseteq \bigsqcup_i t_i^p\} \cup \bigcup_i (\{t_e \sqsubseteq t_i^p \Rightarrow t_i \sqsubseteq t\})$$

  This means that the result type t will be the *union* of the type of each pattern that may match the tested value.

- Typing the message handling may result in any possible branch type (hence the union) and adds all pattern types to the current self type:

$$\frac{\begin{array}{c}\mathcal{E} \vdash p_i : t_i^p,\, \mathcal{E}_i \\ \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\, C_i \qquad C_i' = \{\mathcal{E}(\texttt{self}) \sqsubseteq @\overline{t_i^p}\}\end{array}}{\mathcal{E} \vdash \texttt{receive}\ p_1 \rightarrow e_1;\ \ldots : \bigsqcup_i t_i,\ \bigcup_i (C_i \cup C_i')}$$

- Typing an application is much more complex. First, one must type the function expression and each argument expression.

$$\frac{\mathcal{E} \vdash e : t_e,\, C_e \qquad \mathcal{E} \vdash e_i : t_i,\, C_i}{\mathcal{E} \vdash e(e_1, \ldots, e_n) : t,\ C_e \cup \bigcup_i C_i \cup C}$$

  where C is composed of $t_e \sqsubseteq dom(T_{\mathcal{F}})$, $\mathcal{E}(\texttt{self}) \sqsubseteq @I$, $\mathbf{Fun}(T_{\mathcal{F}}, t_e, n) \sqsubseteq (t_1 \times \ldots \times t_n) \xrightarrow{I} t$ meaning that:
  - The function must be defined.
  - Its effect I is added to the current process effect.
  - All possible functions are subtype of a function type accepting the n actual arguments t_i, having an effect I and resulting in t (it is the result of the application). To get the set of possible functions, we use a function **Fun** which applied to $(T_{\mathcal{F}}, t_e, n)$ returns the union of all function types associated to an atom (and the arity n) of t_e in $T_{\mathcal{F}}$. Like the transformation from tuple type to message type, this function is lazy and waits to know the value of t_e to perform its action.

For each possible functions of type $\alpha \xrightarrow{I'} \beta$, the last constraint ensures that all applications are legals because by substyping it leads to $\{t_1 \times \ldots \times t_n \sqsubseteq \alpha,\ I' \sqsubseteq I,\ \beta \sqsubseteq t\}$. Furthermore, all effects (resp. results) are cumulated in the global effect I (resp. result t).

The function typing environment $T_{\mathcal{F}}$ results from the typing of all functions in $\mathcal{F}$. A mapping $(s, n) \mapsto f$ in $\mathcal{F}$ adds a mapping $(s, n) \mapsto t_f$ if the typing of f by the rule below results in t_f. And, We suppose that all constraints it may produce are added to the global constraint set before resolution.

$$\frac{\mathcal{E} \vdash p_i : t_i,\, \mathcal{E}_i \qquad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i',\, C_i}{\mathcal{E} \vdash p_1 \rightarrow e_1;\ \ldots : \bigsqcup_i (t_i \rightarrow t_i'),\ \bigcup_i C_i}$$

Going back to our example, the application of `F` leads to:

$$\left\{\begin{array}{l}\texttt{state3} \sqsubseteq T,\ \mathit{unit} \sqsubseteq \alpha,\ \ T \sqsubseteq \{\texttt{state1}, \texttt{state2}, \texttt{state3}\}, \\ T_P \sqsubseteq @I,\ \ \mathbf{Fun}(T_{\mathcal{F}}, T, 0) \sqsubseteq \mathit{unit} \xrightarrow{I} t\end{array}\right\}$$

The first constraint combined with the fifth leads to:

$$\mathit{unit} \xrightarrow{\{\texttt{kill} : (\bot,\, \mathit{unit})\}} \texttt{true} \sqsubseteq \mathit{unit} \xrightarrow{I} t$$

This imply that $T_P \sqsubseteq @I \sqsubseteq @\{\texttt{kill} : (\bot, \mathit{unit})\}$ and $\texttt{true} \sqsubseteq t$. The first constraint simulates (in the type system) the reception of *unit* message: $(\bot, \mathit{unit}) \sqsubseteq (\mathit{unit}, \alpha)$ equivalent to $\{\bot \sqsubseteq \mathit{unit},\ \alpha \sqsubseteq \mathit{unit}\}$. Adding this to the initial constraint set leads to a solvable constraint set (where $\alpha = \mathit{unit}$). This allows the system to guarantee the correctness.

6. SCALING TO ERLANG TYPING

The simplified system presented here does not correspond to the real prototype implementation. To scale to this system, we have to:

- extend the types by lists, characters, floating point numbers and all other basic types (corresponding to ERLANG basic values). This extension and the definition of built-in function is straightforward but need to add a lot of rules.
- change scoping rule policy. Our system needs to have an input and an output environment for each expression. This is also boring routine.
- add guards to the pattern matching (again routine extension). Notice that in the prototype, it is one of the constructions that contains a lot of type informations.
- take care of dynamic patterns. Indeed, in ERLANG, a variable in a pattern is a definition only if the variable is not already defined. This small modification of the semantics and more precisely of the semantics of patterns needs important changes in the type system summarized just below.

One of the biggest problem that we faced when typing ERLANG is dynamic pattern matching. Indeed, in the patterns, a variable is not always a binding occurrence, that is, if the variable is already bound, its value replaces the variable before pattern matching is realized. For example, consider:

```
g(X) -> case 1 of X -> ok; _ -> no end.
```

The term `{g(1),g(2)}` reduces to:

```
{case 1 of 1 -> ... , case 1 of 2 -> ...}
```

and then to `{ok,no}`. Usual typing of this function gives $\alpha \rightarrow t$ with the constraints:

$$\{1 \sqsubseteq \alpha \Rightarrow \texttt{ok} \sqsubseteq t;\ 1 \sqsubseteq (\top \backslash \alpha) \Rightarrow \texttt{no} \sqsubseteq t\}$$

Therefore, the application has type $(\texttt{ok} \sqcup \texttt{no}) \times (\texttt{ok} \sqcup \texttt{no})$ because the two applications gives $1 \sqcup 2 \sqsubseteq \alpha$ meaning that both branches may be used. The problem comes from the fact, that the usual function typing impose to all possible real argument types to be **simultaneously** compatibles with all their potential use in the body of the function. For this, when typing the body of the function, the system collects constraints of the form $\alpha \sqsubseteq t$ where α is the type of an argument. And each call to the function produces constraints of the form $t' \sqsubseteq \alpha$ which enable by transitivity to ensure that $t' \sqsubseteq t$. But, in the body of a function, if a pattern includes an argument, the system generates a constraint $t \sqsubseteq \alpha$ incomparable with $t' \sqsubseteq \alpha$. This means that we cannot guarantee that the argument respect one of the constraints required by the function.

The type obtained for `{g(1),g(2)}` is not very precise (using usual strategy) but above all, if the joker branch is not in the choice, the program cause an error that cannot be detected by the type system. To solve this problem, the system is going to type each application of a function using a fresh instance of its type. With this strategy no harmful flow (of information) may happen between two application sites as before. Indeed, the intuition behind this problem is that when a function use one of its arguments in a pattern, each application produces a new (and different) version of the body (of the function). Therefore, the constraints it imposes are not the same and the return type are different too.

The typing of a function leads to a type $\alpha \to \beta$ and a constraint set C. Its calling on an argument of type t will use type $t \to \beta'$ (where β' is fresh) and add $[t/\alpha,\ \beta'/\beta]C$ to the global constraint set. Therefore, typing:

```
g(X) -> case 1 of X -> ok end.
```

gives $\alpha \to t$ with $\{1 \sqsubseteq \alpha,\ \texttt{ok} \sqsubseteq t\}$. Therefore, the type of `{g(1),g(2)}` is $t_1 \times t_2$ with $\{1 \sqsubseteq 1,\ \texttt{ok} \sqsubseteq t_1,\ \boxed{1 \sqsubseteq 2},\ \texttt{ok} \sqsubseteq t_2\}$ where the boxed constraint is false. The error is now detected!

The drawback of this strategy is that the number of type variables and constraints grow more rapidly. To solve this problem, in practice, the system apply this strategy only to a subset of functions. More precisely, this strategy is applied to the arguments of functions using one of their arguments in a pattern. As this situation is not the most usual, the cost to pay (for this strategy) is not too expensive (in general).

7. DISCUSSION

In this paper, we have proposed a formalization of the ERLANG semantics using a two level reduction system. A first level concentrates on concurrent aspects of the language using a formalism inspired by the π-calculus, the configurations. And a second expressing the functional semantics (and its potential concurrent effects) using a more classic setting. Finally, we have introduced a type system for ERLANG insisting in the original parts of our works: message typing and the fact that the system try to stay close to the language. The versions presented in this article represent only insight of the complex system developed and the prototype of static analyzer realized.

Formal semantics of Erlang

This work though not complete can be a good beginning to reach a good formalization of the semantics of ERLANG. A complete formalization of the whole language would require a lot of work because one would have to:

- **add the node (site) notion**. For this, configurations must be extended by a set of node names and by a construction $\langle n \mid w \rangle_n$ meaning that w is executed on node n. A configuration describing a two nodes could then be $\nu n_1, n_2.(\langle n_1 \mid w_1 \rangle \parallel \langle n_2 \mid w_2 \rangle)$.
- **implement dynamic code replacement**. Each site must include the environment of defined functions and the values of those functions could change: $\langle n \mid \mathcal{E} \mid w \rangle$.
- **allow sending message between sites**. The target of the message may be local keeping the same syntax or remote on node n and the transit message could be $a@n \triangleleft m$.
- **integrate the time notion**. In ERLANG, the message handling operation has a clause `after` that allows to stop the execution of this instruction after a specified delay. One solution could be to add a notion of counter to each node.
- **add a notion of symbolic names and a dictionary**. A service can be abstracted by associating it with a name. This declared name represent a process (that can change). Each node needs to maintain dictionary: $\langle n \mid \mathcal{E}_f \mid \mathcal{E}_n \mid w \rangle$.
- **add signals**. ERLANG use signals to propagate exceptions among processes. For example, we could add a flag to the message making it possible for the receiver to distinguish a signal from a message.

Some recent work on distributed process calculi like Dπ (see [21]) or the join calculus (see [14]) can also help in such a project of formalization of the semantics of ERLANG. Notice that those points are not all the problems that needed to be solved, we refer the interested reader to the chapter 10, 11 and 12 of [3]. Those three chapters does not include a formal semantics but their informal systematic description of ERLANG semantics enable to view all possibilities.

Complete Erlang Typing

To become a complete and widely usable tool our system needs some extensions.

First, the ERLANG messages does not contain label so the type of process must be retailored. The works on XMλ (a typed functional language used to manipulate XML documents) of [23] can be a good basis. Indeed, to type correctly the choices of XML, they build a typed λ-calculus including a notion of record without label. For example, $(1) + (\texttt{"test"}) + (\lambda x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 0)$ is typed by $\{int;\ string;\ bool \to int\}$. This adaptation does not seem to be straightforward because the type system of XMλ use equality constraints and is based upon a notion of constraint implication. Therefore, its integration with the subtyping needed for ERLANG needs studies about subtyping constraint implication and to our knowledge, none of the work made in this area have really achieved that goal yet.

In the context of telecommunication systems, exceptions are very important to reach a certain level of quality for programs. Indeed, the reliability of such applications needs a precise treatment of every possible exceptions. A type system helping the programmer in this task would be a real aid. It could estimate the set of potential exception caused by every expressions of the program and ensure that they are treated. An extension of [18] may be a good start point toward such a static analyzer.

Finally, the most difficult point with ERLANG is that the approximation made by this ideal type system should have to be compatible with *hot code swapping*. Indeed, in ERLANG, a module is used by hundreds or thousands of nodes that cannot be stopped or restarted. An evolution of such a module use dynamic code replacement and therefore, the old version and the new one have to be executed simultaneously and must cooperate safely (at least for a temporary period). Such a task is totally out of reach at the moment, but a first step to its resolution could start from [22].

8. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. The MIT Press, Cambridge, MA, USA, 1986.

[2] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proc. of POPL*, pages 163–173, Portland, USA, Jan. 1994. ACM Press.

[3] J. Barklund and R. Virding. ERLANG *4.7.3 Reference Manual*, February 1999. downloadable from `www.erlang.org`.

[4] G. Boudol. The π-calculus in direct style. In *Proc. of POPL*, pages 228–241. ACM, Jan. 1997.

[5] R. Carlsson. An introduction to core erlang. Erlang Workshop. Principles, Logics, and Implementations of High-level Programming Languages. Florence, 2001.

[6] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. *Core Erlang 1.0.2, language specification*, Oct. 2001.

[7] J.-L. Colaço, M. Pantel, F. Dagnat, and P. Sallé. Static safety analysis for non-uniform service availability in actors. In *Proc. of FMOODS*, pages 371–386, Florence, Italy, Feb. 1999. Kluwer.

[8] J.-L. Colaço, M. Pantel, and P. Sallé. CAP: An actor dedicated process calculus. In *Proc. of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[9] J.-L. Colaço, M. Pantel, and P. Sallé. A set-constraint based analysis of actors. In *Proc. of FMOODS*, Canterbury, UK, July 1997. Chapman & Hall.

[10] F. Dagnat. A framework for typing actors and concurrent objects. Ongoing report, available from `perso-info.enst-bretagne.fr/~fdagnat`, 2002.

[11] F. Dagnat, M. Pantel, M. Colin, and P. Sallé. Typing concurrent objects and actors. *L'Objet - Méthodes formelles pour les objets*, Volume 6(1/2000):pages 83–106, May 2000.

[12] M. Dam and L. Fredlund. On the verification of open distributed systems. In *Proc. of the ACM Symposium on Applied Computing*, volume 28, pages 532–540. ACM, June 1998.

[13] M. Fahndrich. *BANE: A library for Scalable Constraint-Based Program Analysis.* PhD thesis, University of California at Berkley, 1999.

[14] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proc. of CONCUR, Pisa, Italy*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[15] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *Proceedings of ICFP '99*, 34(9):261–272, Sept. 1999.

[16] A. Lindgren. A prototype of a soft type system for erlang. Master's thesis, Computing Science Departement, Uppsala University, 1996.

[17] S. Marlow and P. Wadler. A practical subtyping system for ERLANG. In *Proc. of International Conference on Functionnal Programming*, June 1997.

[18] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[19] F. Pottier. Simplifying subtyping constraints: a theory. *Information & Computation*, 170(2):153–183, Nov. 2001.

[20] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.

[21] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. of ICALP '98. LNCS 1443*, pages 695–706. Spinger-Verlag, July 1998.

[22] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. of POPL*, pages 236–247, London, UK, Jan. 2001.

[23] M. Shields and E. Meijer. Type-indexed rows. In *Proc. of POPL*, pages 261 - 275, London, UK, Jan. 2001.

[24] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[25] X. Thirioux, M. Pantel, and M. Colin. Multi-set abstraction of non-uniform behavior concurrent objects. Work in progress, Nov. 2002.

[26] V. T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. of OTAS, Kanazawa, Japan*, volume 742 of *LNCS*, pages 460–474, New York, USA, 1993. Springer-Verlag.

[27] M. Williams and J. Armstrong. *Program Development Using Erlang - Programming Rules and Conventions.* ERICSSON, mar 1996. Doc. EPK/NP 95:035.

APPENDIX

Configurations reduction rules:

CONGRUENCE :
$$\frac{w_1 \equiv w_1' \qquad w_1' \longrightarrow w_2' \qquad w_2' \equiv w_2}{w_1 \longrightarrow w_2}$$

PARALLEL :
$$\frac{w_1 \longrightarrow w_2}{w \,||\, w_1 \longrightarrow w \,||\, w_2}$$

RESTRICTION :
$$\frac{w_1 \longrightarrow w_2}{\nu a.w_1 \longrightarrow \nu a.w_2}$$

ACCEPT :
$$\frac{\mathcal{P}(m,e)}{\langle a \,|\, \widetilde{m}\rangle \triangleright e \,||\, a \triangleleft m \longrightarrow \langle a \,|\, m\ \widetilde{m}\rangle \triangleright e}$$

REJECT :
$$\frac{not(\mathcal{P}(m,e))}{\langle a \,|\, \widetilde{m}\rangle \triangleright e \,||\, a \triangleleft m \longrightarrow \mathbf{Err}}$$

EXPRESSION :
$$\frac{a \notin \mathcal{FN}(\alpha \triangleright e) \qquad a \vdash \alpha,\ e \xrightarrow{w}_e \alpha',\ e'}{\alpha \triangleright e \longrightarrow \nu a.(\alpha' \triangleright e' \,||\, w)}$$

Evaluation context grammar:

$$\begin{array}{l} \mathrm{C} ::= [] \mid (\mathrm{C}) \mid \{\mathrm{A}\} \mid \mathrm{C},e \mid \mathrm{C}\,!\,e \mid e\,!\,\mathrm{C} \mid \mathrm{C}(e,\ldots,e) \mid e(\mathrm{A}) \mid \texttt{case}\ \mathrm{C}\ \texttt{of}\ f\ \texttt{end} \\ \mathrm{A} ::= [] \mid e,\mathrm{A} \mid \mathrm{A},e \end{array}$$

Matching semantics:

$$\begin{cases} v/[] \triangleq \mathbf{Err} \\ v/(p\ \texttt{when}\ g \to e) :: f \triangleq \begin{cases} v/f & \text{if } \mathbf{match}(p,v) = \mathbf{fail} \\ \sigma(e) & \text{if } \mathbf{match}(p,v) = \sigma \end{cases} \end{cases}$$

Functional reduction rules:

VARIABLE ERROR :
$$a \vdash \alpha,\ \mathrm{C}[x] \longrightarrow_e \alpha,\ \mathbf{Err}$$

SEQUENCE :
$$a \vdash \alpha,\ \mathrm{C}[v,\ e] \longrightarrow_e \alpha,\ \mathrm{C}[e]$$

APPLICATION ERROR :
$$\frac{(v,n) \notin dom(\mathcal{F})}{a \vdash \alpha,\ \mathrm{C}[v(v_1,\ldots,v_n)] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

APPLICATION :
$$a \vdash \alpha,\ \mathrm{C}[v(v_1,\ldots,v_n)] \longrightarrow_e \alpha,\ \mathrm{C}[\{v_1,\ldots,v_n\}/\mathcal{F}(v,n)]$$

CASE :
$$a \vdash \alpha,\ \mathrm{C}[\texttt{case}\ v\ \texttt{of}\ f\ \texttt{end}] \longrightarrow_e \alpha,\ \mathrm{C}[v/f]$$

SEND ERROR :
$$\frac{v_1 \notin \mathbb{A}}{a \vdash \alpha,\ \mathrm{C}[v_1\,!\,v_2] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

SEND :
$$\frac{v_1 \in \mathbb{A}}{a \vdash \alpha,\ \mathrm{C}[v_1\,!\,v_2] \xrightarrow{v_1 \triangleleft v_2}_e \alpha,\ \mathrm{C}[v_2]}$$

SPAWN ERROR :
$$\frac{v' \text{ is not a tuple}}{a \vdash \alpha,\ \mathrm{C}[\texttt{spawn}(v,v')] \longrightarrow_e \alpha,\ \mathbf{Err}}$$

SPAWN :
$$a \vdash \alpha,\ \mathrm{C}[\texttt{spawn}(v,v_1,\ldots,v_n)] \xrightarrow{\langle a \,|\, \varnothing\rangle \triangleright v(v_1,\ldots,v_n)}_e \alpha,\ \mathrm{C}[a]$$

SELF ERROR :
$$a \vdash \star,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \star,\ \mathbf{Err}$$

SELF :
$$a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{self}()] \longrightarrow_e \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[a']$$

RECEIVE ERROR :
$$a \vdash \star,\ \mathrm{C}[\texttt{receive}\ f\ \texttt{end}] \longrightarrow_e \star,\ \mathbf{Err}$$

RECEIVE :
$$\frac{\mathbf{matchmailbox}(f,\widetilde{m}) = \widetilde{m}',\ e}{a \vdash \langle a' \,|\, \widetilde{m}\rangle,\ \mathrm{C}[\texttt{receive}\ f\ \texttt{end}] \longrightarrow_e \langle a' \,|\, \widetilde{m}'\rangle,\ \mathrm{C}[e]}$$

Mailbox semantics:

$$\frac{\exists j\ \ (\forall i < j\ \ m_i/f_1 = \mathbf{Err}) \qquad m_j/f_1 = e}{\mathbf{matchmailbox}(f_1 ::\_, (m_i)_{i\in J}) = (m_i)_{i\in J\setminus\{j\}},\ e} \qquad \frac{(\forall i \in J\ \ m_i/f_1 = \mathbf{Err})}{\mathbf{matchmailbox}(f_1 :: fl, (m_i)_{i\in J}) = \mathbf{matchmailbox}(fl, (m_i)_{i\in J})}$$

Type Conversion:

$$\begin{cases} \widehat{s} \triangleq \{s : (unit,\ \top)\} \\ \widehat{s \times T_1 \times \ldots \times T_n} \triangleq \{s : (T_1 \times \ldots \times T_n,\ \top)\} \\ \widehat{\top} \triangleq \top_I \\ \widehat{\bigsqcup_i T_i} \triangleq \bigsqcup_i \widehat{T_i} \\ \widehat{\bigsqcap_i T_i} \triangleq \bigsqcap_i \widehat{T_i} \\ \widehat{\alpha} \triangleq \widehat{\alpha} \quad \text{if } \alpha \text{ is a type variable} \\ \widehat{T} \triangleq \mathbf{Err} \quad \text{otherwise} \end{cases} \qquad \begin{cases} \overline{s} \triangleq \{s : (\bot,\ unit)\} \\ \overline{s \times T_1 \times \ldots \times T_n} \triangleq \{s : (\bot,\ T_1 \times \ldots \times T_n)\} \\ \overline{\top} \triangleq \top_I \\ \overline{\bigsqcup_i T_i} \triangleq \bigsqcup_i \overline{T_i} \\ \overline{\bigsqcap_i T_i} \triangleq \bigsqcap_i \overline{T_i} \\ \overline{\alpha} \triangleq \overline{\alpha} \quad \text{if } \alpha \text{ is a type variable} \\ \overline{T} \triangleq \mathbf{Err} \quad \text{otherwise} \end{cases}$$

Subtyping Deduction System:

$$\bot \sqsubseteq T \qquad T \sqsubseteq \top \qquad \{\} \sqsubseteq I \qquad I \sqsubseteq \top_I \qquad \frac{T \sqsubseteq T_1 \quad T \sqsubseteq T_2}{T \sqsubseteq T_1 \sqcap T_2} \qquad \frac{T \sqsubseteq T_1}{T \sqsubseteq T_1 \sqcup T_2} \qquad \frac{T \sqsubseteq T_2}{T \sqsubseteq T_1 \sqcup T_2} \qquad \frac{i \in \mathbb{N}}{i \sqsubseteq int} \qquad \frac{s \in \mathbb{A}t}{s \sqsubseteq atom}$$

$$T_1 \times \ldots \times T_n \sqsubseteq tuple \qquad \frac{\forall i \;\; T_i \sqsubseteq T_i'}{T_1 \times \ldots \times T_n \sqsubseteq T_1' \times \ldots \times T_n'} \qquad \frac{I' \sqsubseteq I}{@I \sqsubseteq @I'} \qquad \frac{T_1' \sqsubseteq T_1 \quad I \sqsubseteq I' \quad T_2 \sqsubseteq T_2'}{T_1 \xrightarrow{I} T_2 \sqsubseteq T_1' \xrightarrow{I'} T_2'}$$

$$\frac{T_1 \sqsubseteq T_1' \quad T_2' \sqsubseteq T_2 \quad I \sqsubseteq I'}{\{m : (T_1, T_2),\ I\} \sqsubseteq \{m : (T_1', T_2'),\ I'\}}$$

Typing Deduction System:

Var
$$\frac{V \in dom(\mathcal{E})}{\mathcal{E} \vdash V : \mathcal{E}(V),\ \{\}}$$

Constant
$$\mathcal{E} \vdash c : c,\ \{\}$$

Tuple
$$\frac{\mathcal{E} \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \{e_1, \ldots, e_n\} : t_1 \times \ldots \times t_n,\ \bigcup_i C_i}$$

Paren
$$\frac{\mathcal{E} \vdash e : t,\ C}{\mathcal{E} \vdash (e) : t,\ C}$$

Sequence
$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \quad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1, e_2 : t_2,\ C_1 \cup C_2}$$

Send
$$\frac{\mathcal{E} \vdash e_1 : t_1,\ C_1 \quad \mathcal{E} \vdash e_2 : t_2,\ C_2}{\mathcal{E} \vdash e_1 ! e_2 : t_2,\ C_1 \cup C_2 \cup \{t_1 \sqsubseteq @\widehat{t_2}\}}$$

Case
$$\frac{\mathcal{E} \vdash e : t_e,\ C_e \quad \mathcal{E} \vdash p_i : t_i^p,\ \mathcal{E}_i \quad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \textbf{case}\ e\ \textbf{of}\ p_1 \rightarrow e_1;\ \ldots : t,\ C_e \cup \bigcup_i C_i \cup \{t_e \sqsubseteq \bigsqcup_i t_i^p\} \cup \bigcup_i (\{t_e \sqsubseteq t_i^p \Rightarrow t_i \sqsubseteq t\})}$$

Application
$$\frac{\mathcal{E} \vdash e : t_e,\ C_e \quad \mathcal{E} \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash e(e_1, \ldots, e_n) : t,\ C_e \cup \bigcup_i C_i \cup \{t_e \sqsubseteq dom(T_{\mathcal{F}}),\ \mathcal{E}(\textbf{self}) \sqsubseteq @I,\ \textbf{Fun}(T_{\mathcal{F}}, t_e, n) \sqsubseteq (t_1 \times \ldots \times t_n) \xrightarrow{I} t\}}$$

Receive
$$\frac{\mathcal{E} \vdash p_i : t_i^p,\ \mathcal{E}_i \quad \mathcal{E} \cup \mathcal{E}_i \vdash e_i : t_i,\ C_i}{\mathcal{E} \vdash \textbf{receive}\ p_1 \rightarrow e_1;\ \ldots : \bigsqcup_i t_i,\ \bigcup_i (C_i \cup \{\mathcal{E}(\textbf{self}) \sqsubseteq @\overline{t_i^p}\})}$$

Stand-alone Erlang

Stand-alone Erlang is a minimal Erlang distribution. It features:

- ecc - the Erlang compiler
- elink - the Erlang linker
- ear - the Erlang archiver
- escript - the Erlang scripting interface
- esh - the Erlang shell
- You can generate windows `.exe` files in Linux, and vice. versa.
- You can builds Erlang applications in very few files.
- Applications start very quickly.
- Applications have a small memory footprint.

Download.

Installation (linux)

Here is a typescript of a session where we fetch, install and test SAE:

Fetching and installing SAE

```
$ wget http://www.sics.se/~joe/sae-r9b0-1.tgz
$ gunzip sae-r9b0-1.tgz
$ tar -xf sae-r9b0-1.tar
$ cd sae-r9b0-1/dist
./Install
installing executables in /home/joe/bin
installing code in /home/joe/lib/sae
patching executables
rebase: /home/joe/bin/ecc as /home/joe/lib/sae
rebase: /home/joe/bin/elink as /home/joe/lib/sae
rebase: /home/joe/bin/esh as /home/joe/lib/sae
rebase: /home/joe/bin/escript as /home/joe/lib/sae
rebase: /home/joe/bin/ear as /home/joe/lib/sae
ecc is ok
elink is ok
beam_evm is ok
Compiling and building a test program
Running the test
Running test
test worked
```

Test that we can find ecc and elink.

```
$ which ecc
/home/joe/bin/ecc
$ which elink
/home/joe/bin/elink
```

Make the test programs:

```
$ cd ../examples
$ make
ecc test_hello.erl
ecc test_url.erl
```

```
ecc test_autoload.erl
ecc test_autoload1.erl
ecc test_dets.erl
ecc test_bug.erl
ecc test_include.erl
elink test_hello.beam
elink test_url.beam
elink test_autoload.beam
elink test_dets.beam
elink test_bug.beam
elink test_include.beam
./test_hello
Hello world
Args=["./test_hello"]
test_hello worked
./test_url
test_url worked
./test_include
test_include worked
./test_dets
test_dets worked
./test_autoload
Auto load test incomplete $ROOTDIR not defined
./test_bug
aaaa
test_bug worked
```

Test that scripting works. Note: *fib1* is an interpreted script. `fib2` is a compiled script.

```
$ cd ../dist
./factorial 20
factorial 25 = 15511210043330985984000000

$ cd ../examples
$ time ./fib1 24
fib 24 = 46368

real    0m2.603s
user    0m2.490s
sys     0m0.060s

$ time ./fib2 24
fib 24 = 46368

real    0m0.451s
user    0m0.400s
sys     0m0.020s
```

Very quick start

Store the following in a file called `hello.erl`:

```
-module(hello).

-export([start/1]).

start(Args) ->
    io:format("Hello world~nArgs=~p~n", [Args]),
    erlang:halt().
```

Compile and run as follows:

```
$ ecc hello.erl
$ elink hello.beam
$ ./hello 1 2 3
Hello world
Args=["./hello","1","2","3"]
```

The file hello is a 2 KByte executable which can be distributed together with the original system files.

Quick start (Linux)

To compile:

```
> ecc *.erl
```

This will compile all the `.erl` files in the current directory producing `.beam` files if the compilation was successful.

To create a Unix application called `myUnixProg` give the command:

```
> elink -o myUnixProg *.beam
```

When the program is run in Unix with the command:

```
> myUnixProg Arg1 Arg2 ...
```

The function:

```
myUnixProg:start([Arg1, Arg2, ...])
```

Will be called. All the arguments are Erlang strings.

Making windows executables

This does not work yet

Note: Windows executables can be built in Unix or vice. versa

To create a stand-alone windows application called `myWindowsProg.exe` give the following command:

```
> elink -t windows -o myWindowsProg.exe *.beam
```

elink reference

```
> elink
elink [-shell] [-windows] [-o OUT[.exe]]
    [-s M] -m M1.beam [M2.beam M3.beam ...]

      Make an executable OUT
      Excuting OUT will load the code in M1 M2 M3
      The command >OUT[.exe] Arg1, Arg2
      Will cause M:start([Arg1, Arg2, ...]) to be evaluated

   > elink [-m] M1.beam M2.beam M3.beam
```

```
      implies s = M1 o=M1
> elink -o Mi[.exe] -m M1.beam M2.beam
      Mi must be in M1 M2 ...
      implies s = Mi
> elink -s Mj -o Mi.[exe] -m M1.beam M2.beam
    Mj must be in M1 M2 ...
> elink -r Dir Executable
    Rehomes Executable to ERL_EARS=Dir
```

elink examples

```
> elink a1.beam a2.beam ...
```

Makes an executable called `a1`. The start function is `a1:start(...)`

```
> elink a1.beam a2.beam ...
```

Makes an executable called `a1`. The start function is `a1:start(...)`

```
> elink -o a3 a1.beam a2.beam ...
```

Makes an executable called `a3`. The start function is `a3:start(...)` - one of the beam files must be `a3.beam`

```
> elink -o a3 -s a4 a1.beam a2.beam ...
```

Makes an executable called `a3`. The start function is `a4:start(...)` - `a3.beam` and `a4.beam` must be included on the command line

```
> elink -shell ...
```

Makes an executable that runs in a shell capable of understanding line editing commands. This is useful if you want to write an interactive application that behaves like the Erlang shell.

```
> elink -r /home/joe/foo/bar /home/joe/bing/bongo
```

Changes the environment `${ERLANG_EARS}` which is hidden inside the executable `/home/joe/bing/bongo` to `/home/joe/foo/bar`.

The directory `/home/joe/foo/bar` *must* contain the files `erlang.ear` and `ErlBoot_new.img` - this command is so obscure that you will probably never ever need to use it. If you got this far then no manual will help.

Erlang scripts

`escript` is the scripting interface to Erlang.

Here is an example of a simple Erlang script:

```
#!/usr/bin/env escript
-export([main/1]).

main([X]) ->
    J = list_to_integer(X),
    N = fac(J),
    io:format("factorial ~w = ~w~n",[J, N]).
```

```
fac(0) -> 1;
fac(N) ->
    N * fac(N-1).
```

Make the file executable and run:

```
> chmod u+x ./factorial
> ./factorial 123
factorial 123 = 121463043670253296757662432418812958554542170884833823153289181618292358923621676688311569606126402021707358352212940477825910915704116514721860295199062616467307339074198149529600000000000000000000000000000
```

The script must export `main/1`. The directive `-mode(compile)` can be included to improve efficiency.

esh

`esh` is an interactive query shell. To start it:

```
> esh
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1>
```

ecc

ecc is the Erlang compiler.

```
> ecc *.erl
```

Compiles files. `ecc` accepts the following flags:

- `-W` Warn
- `+compressed` Make a compressed binary file.
- `+strip` Strip the binary file

elink

`elink` is the Erlang linker.

```
elink [-shell] [-windows] [-o OUT[.exe]]
      [-s Mod] [-m] M1.beam ...
```

Arguments can be pretty much in any order.

ear

`ear` is a command line utility for maintaining Erlang archives.

Erlang stand-alone applications use demand-code loading or static code loading.

The demand code loader loads code from Erlang archives (`.ear` files) - Erlang archives are created

with the `ear` command.

In a statically loaded application all necessary code is loaded contained in the application.

The preferred method of building an application is to use demand code loading. The base distribution includes a single archive `erlang.ear` which contains all the modules and include files in `stdlib` `kernel` and `compiler` - this is sufficient for making a large class of applications.

Building and maintaining Erlang archives is done with the program `ear`

Shipping several applications with your own personal library

If you have several applications you might like to proceed as follows:

Create a personal library

```
> ear -a joe.ear PathToCompiledModules1/*.beam
> ear -a joe.ear PathToCompiledModules2/*.beam
> ear -a joe.ear PathToCompiledModules3/*.beam
```

Now distribute `joe.ear` the installer must place this in the same directory as `erlang.ear`, module names in `joe.ear` must not collide with the names in `erlang.ear`

Use of Erlang in system test of AXD301

Karl Olsson

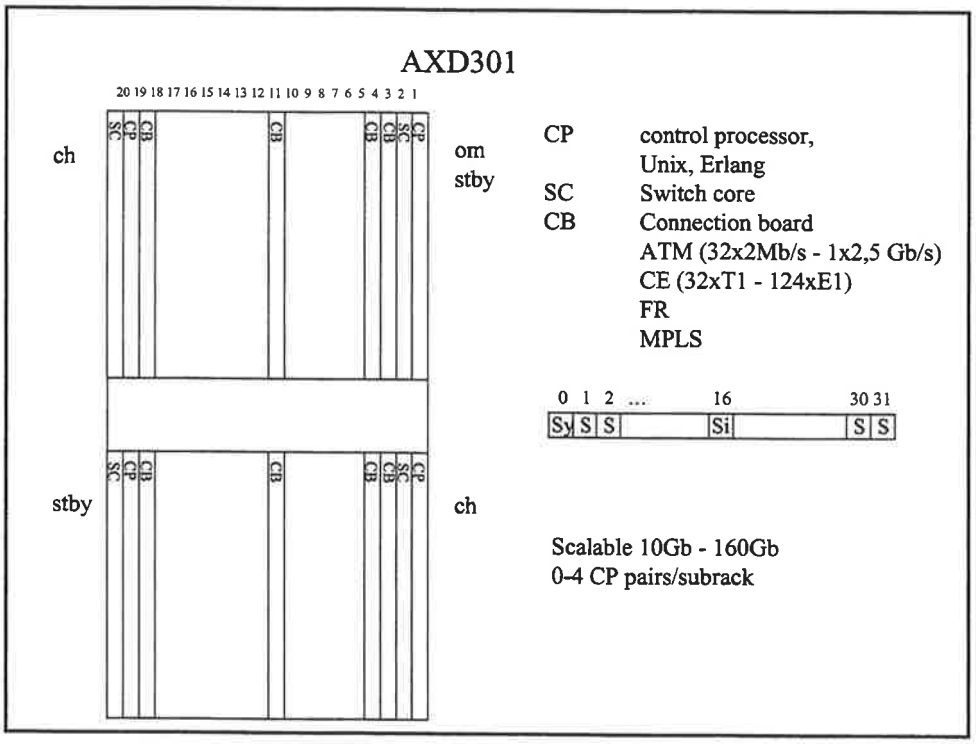
Erlang distribution

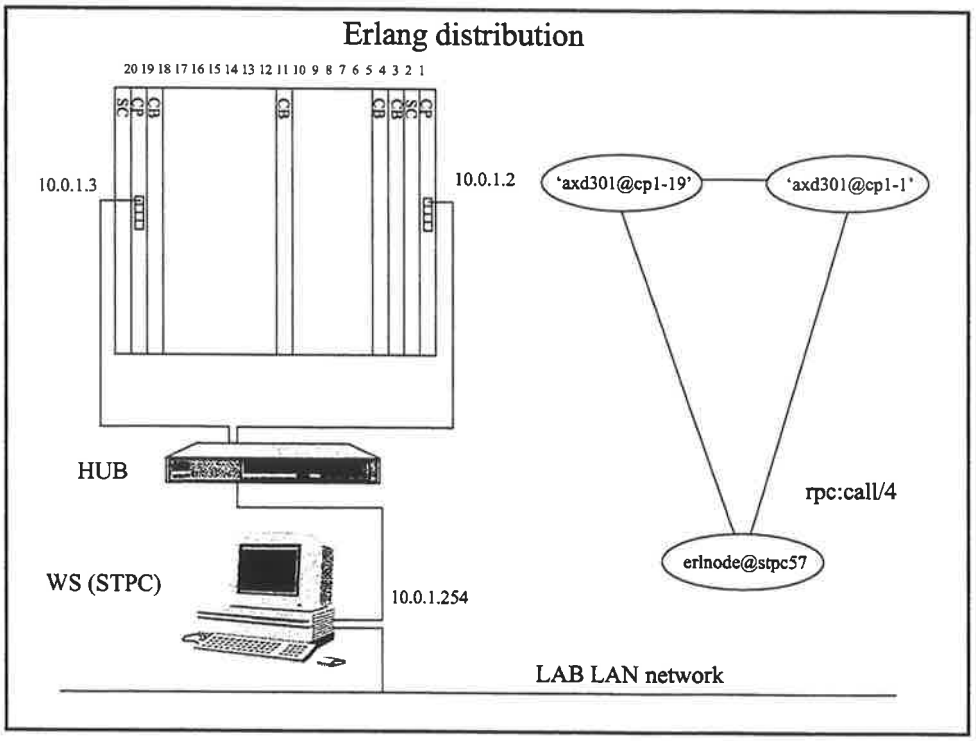
Tools

conf	Configuration tool
sssg	UNI traffic generator
xAXE	AXE 10 traffic simulator/generator
xMG	Media gateway simulator
OTP test server	Automatic test tool

dbg

loading of modules


AXD301
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
ch
om
stby
stby
ch
CP control processor, Unix, Erlang
SC Switch core
CB Connection board
ATM (32x2Mb/s - 1x2,5 Gb/s)
CE (32xT1 - 124xE1)
FR
MPLS
0 1 2 ... 16 30 31
Sy S S Si S S
Scalable 10Gb - 160Gb
0-4 CP pairs/subrack


Erlang distribution
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
10.0.1.3
10.0.1.2
'axd301@cp1-19'
'axd301@cp1-1'
HUB
rpc:call/4
erlnode@stpc57
WS (STPC)
10.0.1.254
LAB LAN network

Tools - conf

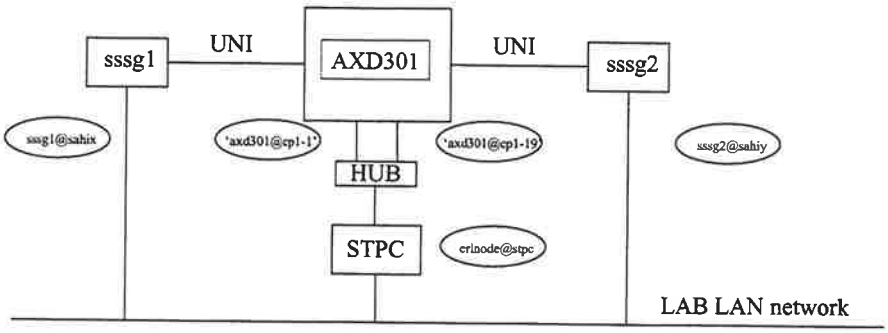
AXD301 user interfaces

- AMS (http)
- snmp
- NO command line interface

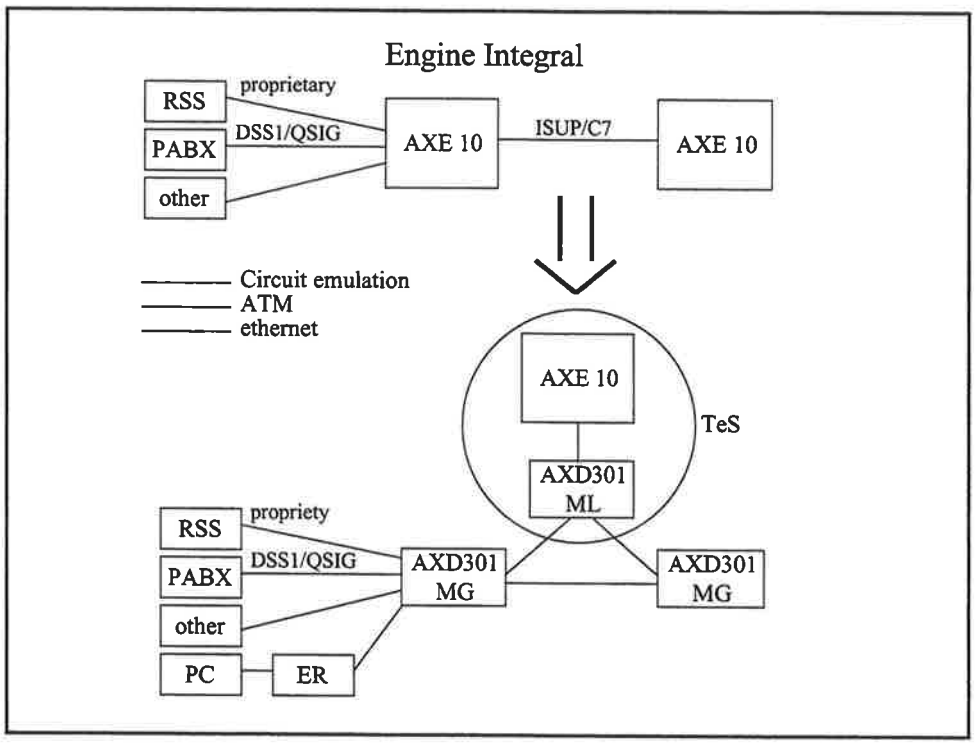
```
%%% ----------------------------------------------------------------------
%%% block_board(Subrack,SlotNo,Type) ->
%%% ok | Other
%%% Input: Subrack integer
%%% SlotNo integer
%%% Type cp1,sc2,clk2,cp19,sc20,clk20,atb,cb
%%% Output: ok | Other
%%% Exceptions:
%%% Description: blocks a board
%%% Example: conf:block_board(1,11,atb).
%%% Example: conf:block_board(1,19,cp19).
%%% ----------------------------------------------------------------------
block_board(Subrack,SlotNo,Type) ->
    {EmSlot,PiuSlot,Side} = case Type of
        cp1 -> {1,1,1}; sc2 -> {2,2,1}; clk2 -> {2,2,2};
        cp19 -> {19,19,1}; sc20 -> {2,20,1}; clk20 -> {2,20,2};
        atb -> {SlotNo,SlotNo,1}; cb -> {SlotNo,SlotNo,2}
    end,
    SubmitFunctions = {eqmLmsPIU,submit_open_piu},
    SubmitRequests =
        [{action,eqmMi,piuTable,[Subrack,EmSlot,PiuSlot,Side],null,
        {{"Block",7,2,null}},
        [{errors,rollback}]}],
    SubmitProperties = [{{pageSource,{eqmLmsPIU,open_piu}},{postAction,0}}].

rpc_call(SubmitFunctions,SubmitRequests,SubmitProperties,
    "block_board -p -p -p Successfull",[Subrack,SlotNo,Type],
    "block_board -p -p -p UNSUCCESSFULL",[Subrack,SlotNo,Type],
    "block_board").
```

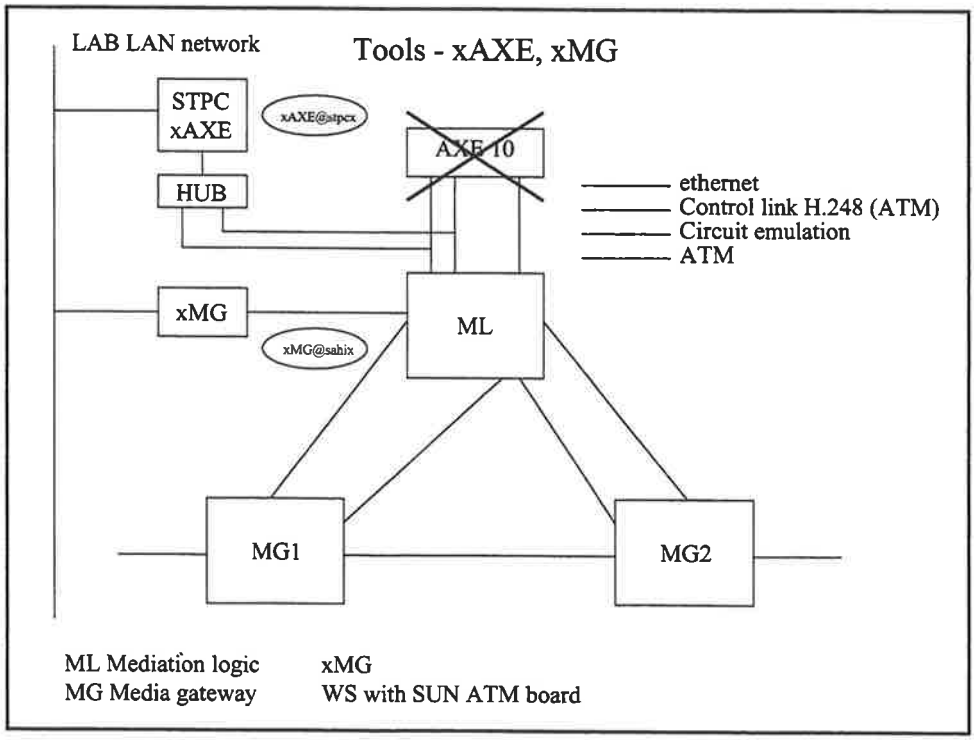
Tools - sssg

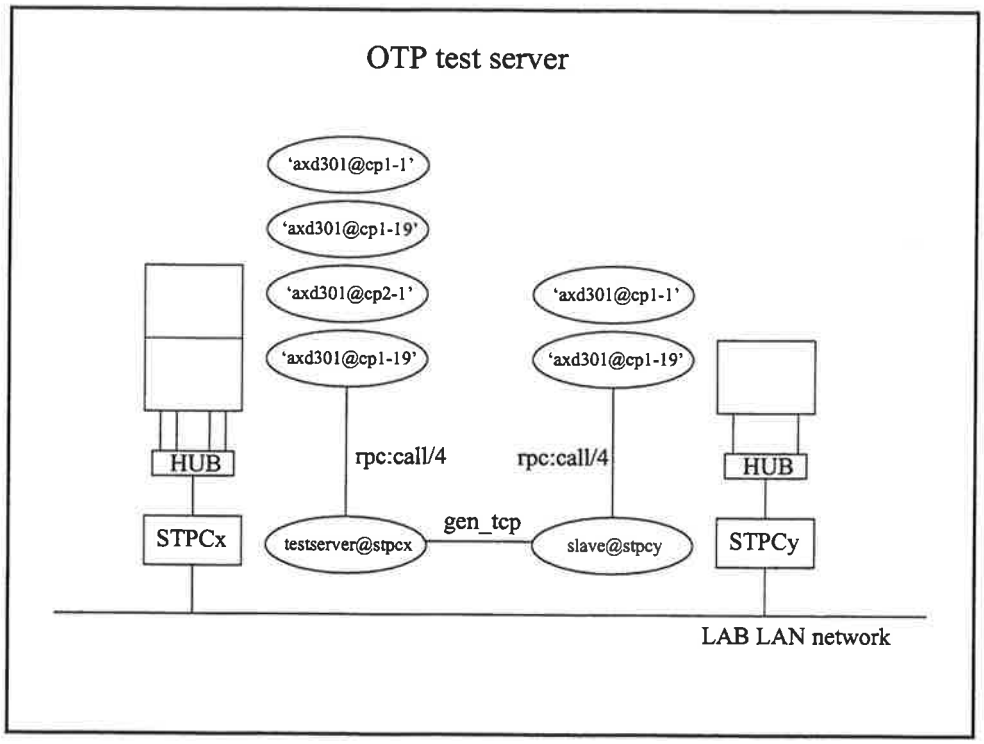
sssg
WS with SUN ATM board

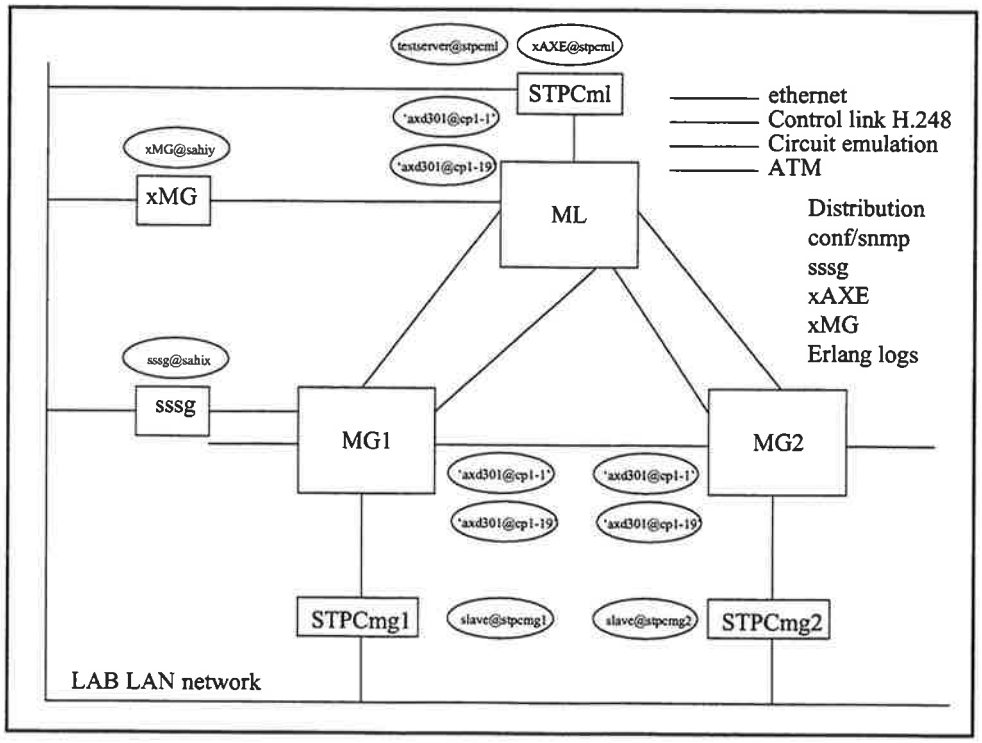


ATM
ethernet


Engine Integral
RSS
PABX
other
proprietary
DSS1/QSIG
AXE 10
ISUP/C7
AXE 10
Circuit emulation
ATM
ethernet
AXE 10
TeS
AXD301
ML
RSS
PABX
other
PC
propriety
DSS1/QSIG
ER
AXD301
MG
AXD301
MG


LAB LAN network
Tools - xAXE, xMG
STPC
xAXE
xAXE@stpcx
AXE 10
HUB
ethernet
Control link H.248 (ATM)
Circuit emulation
ATM
xMG
xMG@sahix
ML
MG1
MG2
ML Mediation logic
MG Media gateway
xMG
WS with SUN ATM board


OTP test server
'axd301@cp1-1'
'axd301@cp1-19'
'axd301@cp2-1'
'axd301@cp1-19'
'axd301@cp1-1'
'axd301@cp1-19'
HUB
rpc:call/4
rpc:call/4
HUB
STPCx
testserver@stpcx
gen_tcp
slave@stpcy
STPCy
LAB LAN network


testserver@stpcml
xAXE@stpcml
STPCml
ethernet
Control link H.248
Circuit emulation
ATM
'axd301@cp1-1'
'axd301@cp1-19'
xMG@sahiy
xMG
ML
Distribution
conf/snmp
sssg
xAXE
xMG
Erlang logs
sssg@sahix
sssg
MG1
MG2
'axd301@cp1-1'
'axd301@cp1-1'
'axd301@cp1-19'
'axd301@cp1-19'
STPCmg1
slave@stpcmg1
slave@stpcmg2
STPCmg2
LAB LAN network

Highlights in Erlang 5.2/OTP R9B

This document describes the major new features and changes in version R9B of Erlang/OTP. The changes are described as a comparision with the original R8B release and some of them have already been delivered as patches to R8B and/or R7B. For more detailed information, please refer to the release notes for the individual applications. Interesting news in this release is that we have integrated results from the ASTEC-HiPE research project as part of the product. See Hipe and Packages below for more info on that.

Significantly updated or new applications in OTP R9B

Asn1	
Comet	**REMOVED**
Compiler	
CosEventDomain	**NEW**
Erlang Runtime System (ERTS)	
Et (Event Tracer)	**NEW**
GS (Graphical System)	
HiPE	**NEW**
IC (IDL compiler)	
Inets	
Kernel	
Megaco (H.248)	
Mnesia	
Observer	**NEW**
ODBC	
Orber	
OS_mon	
Packages	**NEW**
Runtime_tools	
SNMP	
Stdlib	
Tools	

Applications with minor changes and bugfixes

The following applications have only minor changes in R9B:
CosNotification, CosTransactions, Debugger, Erl_interface, EVA (Event & Alarm), Mnesia_session, Pman, SASL, SSL, Toolbar, Tv (Table Visualizer), WebTool.

Asn1, compiler and runtime functions for ASN.1

- New option optimize in combination with per_bin and ber_bin which makes the encode/decode functions much faster than before. In rough terms the encode/decode is 2 times faster when optimize is used.
- It is now possible to add options to the Erlang compiler to be used when compiling the generated .erl file. Any option that is not recognized as a specific ASN.1 option will be passed to the final step like: erlc +debug_info Mymodule.asn or asn1ct:compile('Mymodule',[debug_info]).
- The feature "multi file compilation" which compiles several ASN.1 modules together and produces one .erl file is improved.

Comet, COM client for Erlang (REMOVED)

The Comet application is removed from the product because we currently have no resources to maintain it. We plan to make it available on the Open Source site. It still works on Windows NT 4, but there are problems on Windows XP.

Compiler

- The documentation for the 'compile' module now lists several options that were previously undocumented or only documented in the 'erl_lint' documentation. One very useful option is warn_unused_vars, which also is improved in the compiler. Use of this option can reveal bugs and dead code, it is highly recommended.
- The endianess specification 'native' has been added to the bit syntax. It will resolve to either big or little endian at load time. It is specially useful for communcating with linked-in drivers.

cosEventDomain, OMG Event Domain Admin Service NEW

A new Corba service "cosEventDomain" is added as a separate application. cosEventDomain is compliant with the OMG service CosEventDomainAdmin.

ERTS, Erlang emulator

- The previous hard system limit of 255 known remote nodes has been removed. With the exception of node-name atoms, all data regarding remote nodes is now garbage collected.
- Major improvements regarding memory handling, introduction of sl_alloc version 1 default and sl_alloc version 2 which makes it possible to tune the memory allocation behaviour to best suite a certain system. A number of other memory handling improvements are also added.
- The endianess specification 'native' has been added to the bit syntax. It will resolve to either big or little endian at load time. It is specially useful for communcating with linked-in drivers.
- The maximum number of Erlang processes within one Erlang node is increased to 2^18 –1 =262143 , previously the limit was 32768. To enable the higher limit the +P flag must be used when Erlang is started.

Et, an Event Tracer with graphical viewing of trace data NEW

The two major components of the Event Tracer (ET) tool is a graphical sequence chart viewer (et_viewer) and its backing storage (et_collector).
One collector may be used as backing storage for several simultaneous viewers where each one may display a different view of the same trace data.

GS, a Graphics System

GS is updated to use Tcl/Tk 8.3.4. This is a major update since GS previously used an ancient version of Tcl/Tk.

HiPE, High Performance Erlang NEW

A number of useful and promising features from the HiPE project at Uppsala University is integrated into this version of Erlang/OTP. The major features are:

- Native code generation for Sparc (Solaris) and x86 (Linux). The native option to the compiler is used to select this codegeneration. It is then possible to run the generated modules together with the ordinary interpreted modules on a standard OTP R9B system.
  The native codegeneration can give significant performance improvements especially on sequential code. See http://www.csd.uu.se/projects/hipe/hipe.html for more info. This feature is intended for evaluation and may be supported in future versions. Feedback is velcome.
- "Shared heap" a new approach to memory handling within the Erlang emulator where all Erlang processes share a common heap. This way of handling memory is very interesting and have a potential to reduce memory consumption an improve performance. A separate emulator is built to support "shared_heap", it is started with 'erl -shared'. This feature is intended for evaluation and may be supported in future versions. Feedback is velcome.

IC, an IDL compiler

A number of minor improvements and corrections.

Inets, HTTP server and FTP client.

- A HTTP client is added to the application. Author: Johan Blom of Mobile Arts AB. It is provided as is with very limited documentation in this version but we plan to support it fully in coming versions of Inets.
- Updated to handle HTTP/1.1.

Kernel

- The set_net_ticktime/[1,2] and get_net_ticktime/0 functions has been added to the net_kernel module (see net_kernel(3)) which makes it possible to change the net_tick time during operation.
- There are new functions bchunk/2,3 in the disk_log module that are to be used like chunk/2,3 but return objects as binaries.
- The loading of BEAM code at start-up of embedded systems has been optimized: if the thread pool is non-empty (see the system flag +A in erl(3)) and files are read from a file system (the default, see the value efile of the -loader flag in erl(3)) disk seek times have been reduced.

Megaco, a Megaco/H.248 protocol stack

- The binary codecs ber_bin and per_bin is now both compiled with the +optimize asn1-compiler flag for better runtime performance.
- The previously included tool, et, has been moved out of the Megaco application. It is now provided as a separate application called Et.

Mnesia, a heavy duty real-time distributed database

- The table fragmentation functionality in Mnesia has been improved.

  - Select and match_object is done in parallel which should improve performance.
  - A new concept of hash modules has been introduced. This means that a user now can define its own mapping between record keys and the actual table fragment hosting the record.

- Improved table loading performance during startup. Mnesia should be able to utilize the network bandwidth better, and Mnesia also uses new dets functionality to improve the loading of disc_only_copies tables, if possible.

Observer NEW

Observer is a new application with various facilities for "observing" a live system with minimal disturbance. The application is fully functional and supported, but the functionality and API's are still in beta-status i.e they can be changed in the next versions. We are very interested in feedback from users regarding the functionality in Observer.
Observer currently contains two different parts:

- Trace Tool Builder , a base for building trace tools for single node or distributed Erlang systems.
- Erlang Top, a tool for monitoring of Erlang processes similar to the UNIX top utility.

ODBC

- The Erlang ODBC application consists of both Erlang and C code. The C code is now delivered as a precompiled executable for Windows and Solaris.
- Various optimizations.

- New API that has an Erlang/OTP touch and feel instead of being a C-interface with Erlang syntax. The old interface is deprecated and will be removed in Erlang/OTP R10.

Orber, a CORBA Object Request Broker

- Support for fragmented IIOP-1.2 messages.
- Possible to add and use the IOR component TAG_ALTERNATE_IIOP_ADDRESS.
- Unique VMCID:s assigned to Orber by the OMG.
- Supports the Fixed datatype.
- Possible to add new initial references.
- The NameService can be configured to be stored on disk.
- It is now possible to set Orber's configuration parameters in, for example, an Erlang shell.
- Possible to list which port numbers Orber may use locally when connecting to another ORB.
- Improved documentation.
- Several new debugging facilities:
    - Two IIOP trace interceptors included (different verbosity).
    - Type checking within an Erlang node.
    - OrberWeb, which is an extension of the WebTool application.
    - IOR dump.

OS_Mon, monitoring of disk usage and OS resources

- cpu_sup:util/0 and cpu_sup:util/1 which returns information about cpu utilization have been added. For further information see cpu_sup(3).
- Nodename is now used as key in loadtable (os_mon mib).
- The loadCpuLoad5, loadCpuLoad15 values has been added to the os_mon mib.

Packages NEW

This is an extension to Erlang with structured program module packages, in a simple, straightforward and useful way. The implementation is done by Richard Carlsson from the HiPE team at Uppsala University and is intended for evaluation. This or a slightly modified solution may be supported in future versions of Erlang/OTP. The debugger might have some problems with the naming of modules when packages are used. See http://www.erlang.se/publications/packages.html for more info. There is also a paper about Packages at http://www.it.uu.se/research/reports/2000-001.

Runtime_Tools

- Trace ports can now be opened on remote nodes
- It is possible to use the local node as a "trace control node", i.e. trace only remote nodes.
- The function dbg:i/0 now prints information about all traced nodes
- Added a number of functions for controlling tracing on remote nodes.

SNMP

Minor additions and bugfixes.

STDLIB, Erlang standard libraries

- A number of improvements in dets.
- The function ets:select_count/2 is added to the stdlib application.
- New functions sofs:extension/3 and sofs:partition/3.
- A new module ms_transform which implements a parse transform that translates 'fun' syntax into "match specifications". This simplifies writing of "match specifications" used in ets:select and in dbg.
- The undocumented and deprecated modules bplus_tree and unix has been removed.

Tools

There is a new tool `cprof`, a call count profiler. It is something inbetween `cover` and `fprof`, and can be used to get a picture of which functions are most frequently called. See Tools User's Guide and Reference Manual.

Erlang/OTP User Conference 2002 - Participants

Chairmen and speakers			
Joe Armstrong	SICS	Stockholm, Sweden	joe@sics.se
Johan Blom	Mobile Arts	Stockholm, Sweden	johan.blom@mobilearts.se
Göran Båge	Mobile Arts	Stockholm, Sweden	goran.bage@mobilearts.se
Fabien Dagnat	ENST, Bretagne	Brest, France	Fabien.Dagnat@enst-bretagne.fr
Bjarne Däcker		Huddinge, Sweden	bjarne@cs-lab.org
Magnus Eklund	Cellpoint	Stockholm, Sweden	magnus.eklund@cellpoint.com
Luke Gorrie	Alteon WebSystems	Stockholm, Sweden	luke@bluetail.com
Seif Haridi	SICS	Stockholm, Sweden	seif@sics.se
Bagirath Krishnamachari	Lucent Technologies	Bangalore, India	bagi@lucent.com
Fredrik Linder	Cellpoint	Stockholm, Sweden	fredrik.linder@cellpoint.com
Thomas Lindgren	Cellpoint	Stockholm, Sweden	thomasl_4711@yahoo.com
Kenneth Lundin	OTP Unit, Ericsson	Älvsjö, Sweden	kenneth.lundin@uab.ericsson.se
Hans Nilsson	Ericsson	Kista, Sweden	hans@erix.ericsson.se
Mickaël Rémond	erlang-fr.org	Paris, France	mickael.remond@erlang-fr.org
Konstantinos Sagonas	Uppsala university	Uppsala, Sweden	kostis@csd.uu.se
Erik Stenman	Uppsala university	Uppsala, Sweden	happi@home.se
Thomas Verner	BluePosition	Copenhagen, Denmark	tv@blueposition.com
Claes Wikström	Alteon WebSystems	Stockholm, Sweden	klacke@bluetail.com
Rasmus Wätjen	BluePosition	Copenhagen, Denmark	
Participants			
Mats Andersson	Ericsson	Älvsjö, Sweden	etxmaga@cbe.ericsson.se

Peter Andersson	OTP Unit, Ericsson	Älvsjö, Sweden	peppe@erix.ericsson.se
Ingela Anderton	OTP Unit, Ericsson	Älvsjö, Sweden	ingela@erix.ericsson.se
Marcus Arendt	Marcus Arendt AB	Sollentuna, Sweden	marcus@arendt.se
Thomas Arts	IT-university of Gothenburg	Göteborg, Sweden	thomas.arts@ituniv.se
Gösta Ask	Ericsson	Älvsjö, Sweden	Gosta.Ask@etx.ericsson.se
Mia Berg	Sjöland & Thyselius Telecom	Stockholm, Sweden	mia.berg@st.se
Johan Bevemyr	Alteon WebSystems	Stockholm, Sweden	jb@bluetail.com
Martin Björklund	Alteon WebSystems	Stockholm, Sweden	
Hans Bolinder	OTP Unit, Ericsson	Älvsjö, Sweden	hasse@erix.ericsson.se
Kent Boortz	OTP Unit, Ericsson	Älvsjö, Sweden	kent@erix.ericsson.se
Pascal Brisset	Cellicium	Bagneux, France	pascal.brisset@cellicium.com
Mikael Bylund	Telia Promotor	Uppsala, Sweden	mikael.m.bylund@telia.se
Richard Carlsson	Uppsala university	Uppsala, Sweden	richardc@csd.uu.se
Francesco Cesarini	Cesarini Consulting Ltd	London, UK	francesco@erlang-consulting.com
Mats Cronqvist	Ericsson	Älvsjö, Sweden	mats.cronqvist@etx.ericsson.se
Niclas Eklund	OTP Unit, Ericsson	Älvsjö, Sweden	Niclas.Eklund@uab.ericsson.se
Magnus Fröberg	Alteon WebSystems	Stockholm, Sweden	magnus@bluetail.com
Catrin Granbom	Ericsson	Älvsjö, Sweden	catrin@erix.ericsson.se
Pär Grandin	Ericsson	Älvsjö, Sweden	Par.Grandin@uab.ericsson.se
Joakim Grebenö	Alteon WebSystems	Stockholm, Sweden	
Dan Gudmundsson	OTP Unit, Ericsson	Älvsjö, Sweden	dgud@erix.ericsson.se
Martin Gustafsson		Stockholm, Sweden	martin-g@home.se
Per Gustafsson	Uppsala university	Uppsala, Sweden	pegu2945@csd.uu.se
Björn Gustavsson	OTP Unit, Ericsson	Älvsjö, Sweden	bjorn@erix.ericsson.se
Siri Hansen	OTP Unit, Ericsson	Älvsjö, Sweden	siri@erix.ericsson.se

Per Hedeland	Alteon WebSystems	Stockholm, Sweden	per@bluetail.com
Pekka Hedqvist	PH IT Konsult	Stockholm, Sweden	pekka@home.se
Sean Hinde	T-Mobile	Borehamwood, Herts, UK	Sean.Hinde@t-mobile.co.uk
Henrik Jonasson	Ericsson	Stockholm, Sweden	Henrik.Jonasson@etx.ericsson.se
Micael Karlberg	OTP Unit, Ericsson	Älvsjö, Sweden	micael.karlberg@ericsson.com
Bertil Karlsson	OTP Unit, Ericsson	Älvsjö, Sweden	bertil.karlsson@uab.ericsson.se
Håkan Karlsson	Ericsson	Kista, Sweden	hakan.karlsson@ericsson.com
Mikael Karlsson	Creado Systems	Stockholm, Sweden	mikael.karlsson@creado.com
Bengt Kleberg	Ericsson	Stockholm, Sweden	eleberg@cbe.ericsson.se
Håkan Larsson	Ericsson	Kista, Sweden	hakan.larsson@ericsson.com
Tord Larsson	Alteon WebSystems	Stockholm, Sweden	tord@bluetail.com
Jani Launonen	University of Oulu	Oulu, Finland	jabba@ees2.oulu.fi
Tobias Lindahl	Uppsala university	Uppsala, Sweden	toli6207@csd.uu.se
Erik Lindblom	Ericsson	Stockholm, Sweden	Erik.Lindblom@etx.ericsson.se
Björn Lisper	Mälardalens Högskola	Västerås, Sweden	lisper@it.kth.se
Matthias Läng	Corelatus	Stockholm, Sweden	matthias@corelatus.se
Ann-Marie Löf	Sjöland & Thyselius Telecom	Stockholm, Sweden	ann-marie.lof@st.se
Anna Löfgren	Sjöland & Thyselius Telecom	Stockholm, Sweden	anna.lofgren@st.se
Luca Manai	Ericsson	Älvsjö, Sweden	luca.manai@ericsson.com
Håkan Mattsson	Ericsson	Stockholm, Sweden	hakan@erix.ericsson.se
Håkan Millroth	Alteon WebSystems	Uppsala, Sweden	hakanm@nortelnetworks.com
Chandrashekhar Mullaparthi	T-Mobile	Borehamwood, Herts, UK	Chandrashekhar.Mullaparthi@t-mobile.co.uk
Hans Nahringbauer	Telia Promotor	Uppsala, Sweden	hans.h.nahringbauer@telia.se
Raimo Niskanen	OTP Unit, Ericsson	Älvsjö, Sweden	raimo@erix.ericsson.se
Annika Nordqvist	Ericsson	Älvsjö, Sweden	etxnora@cbe.ericsson.se

Arto Nummelin	Ericsson	Älvsjö, Sweden	etxarnu@cbe.ericsson.se
Patrik Nyblom	OTP Unit, Ericsson	Älvsjö, Sweden	pan@erix.ericsson.se
Jan Nyström	Uppsala university	Uppsala, Sweden	jann@it.uu.se
Sven-Olof Nyström	Uppsala university	Uppsala, Sweden	svenolof@csd.uu.se
Leif Näs	Ericsson	Älvsjö, Sweden	etxlnas@cbe.ericsson.se
Anders Ramsell	Telia Promotor	Uppsala, Sweden	anders.a.ramsell@telia.se
Erik Reitsma	Ericsson	Rijen, The Netherlands	erik.reitsma@eln.ericsson.se
Tony Rogvall	Alteon WebSystems	Stockholm, Sweden	tony@bluetail.com
Per Romin	Ericsson	Nacka, Sweden	Per.Romin@ebc.ericsson.se
Dan Sahlin	Sahlin Innovation	Stockholm, Sweden	dan@fatburen.org
Christian Schulte	IT University	Kista, Sweden	schulte@imit.kth.se
Christer Skeppstedt	Ericsson	Älvsjö, Sweden	Christer.Skeppstedt@etx.ericsson.se
Håkan Stenholm		Stockholm, Sweden	hakan.stenholm@mbox304.swipnet.se
Per Sternås	Ericsson	Nacka, Sweden	Per.Sternas@ebc.ericsson.se
Robert Tjarnström	Ericsson	Älvsjö, Sweden	erandig@cbe.ericsson.se
Lars Thorsén	Ericsson	Älvsjö, Sweden	lars@erix.ericsson.se
Torbjörn Törnkvist	Alteon WebSystems	Stockholm, Sweden	tobbe@bluetail.com
Mats Westerling	Ericsson	Älvsjö, Sweden	etxmweg@cbe.ericsson.se
Ulf Wiger	Ericsson	Älvsjö, Sweden	ulf.wiger@etx.ericsson.se
Jesper Wilhelmsson	Uppsala university	Uppsala, Sweden	jesperw@csd.uu.se
Christopher Williams	Ericsson	Älvsjö, Sweden	chris.williams@ericsson.com
Michael C Williams	Ericsson	Älvsjö, Sweden	mike@erix.ericsson.se
Erik Åckander	Ericsson	Älvsjö, Sweden	etxerac@cbe.ericsson.se
Lennart Öhman	Sjöland & Thyselius Telecom	Stockholm, Sweden	lennart.ohman@st.se
			Updated 2002-11-12