Introductory Guide to Message Passing
In Distributed Systems

Florian Willich
Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences Berlin
Course: Distributed Systems
Lecturer: Prof. Dr. Christin Schmidt

June 13, 2015

Abstract

Message passing in distributed systems is a model to exchange messages within a process pair by making use of several standards and implementation details. Those have been developed to offer the right message passing models for the different areas of applications. The programming language Erlang natively supports an asynchronous message passing model which makes the implementation of concurrent applications transparent to the software developer.

Contents

1 Message Oriented Communication

1.1 Introduction

Communication in distributed systems defines a distributed system itself: N systems execute one or more tasks by computing and communicating with each other. Every time systems shall communicate with each other to provide services, a proficient communication model has to be implemented. There are several models of communication in distributed systems such as Remote Procedure Calls (RPC) or object method invocation (Tanenbaum & Steen, 2007, chap. 4.3 / p. 203). With this paper I'll introduce you to the message passing model in distributed systems.

As human beings we already have a deep understanding of different models of message passing. While you read these words, I am passing a message to you, which seems to be a trivial thing to people who can read. In more philosophical terms, now that you are reading those words, a message is being passed from one entity (me, the writer) to another entity (you, the reader) which is obviously non trivial considering all the assumptions we would have to make to actually realize this message passing model between human beings.

Message passing in distributed systems is based on messages, composed of bit strings, exchanged within a process pair which would be the equivalent to the entity pair. It is important to understand that message passing is a model designed for Inter-Process Communication (IPC). Whether those processes are located on one or on two systems, is irrelevant to the provided functionality.

1.2 Requirements in Theory

The following items are the theoretical basic requirements for a system to provide the functionality of message passing:

- **Connectivity**: A connection to communicate has to be established between the process pair
- **Ability**: Each process has to be able to receive or send messages
- **Integrity**: Sent messages have to be delivered as is
- **Intelligibility**: The receiving process has to be able to interpret the message as intended
- **Executability**: The delivered message has to lead to an execution of the desired instructions

1.3 Requirements Provider in Practice

To provide the above mentioned requirements there several message passing models resulting in well defined standards and concrete implementations have been developed in the last decades. To facilitate understanding the relations between the theoretical requirements and a concrete implementation, I will choose the Transmission Control Protocol / Internet Protocol (TCP/IP) which uses several socket primitives as example (Tanenbaum & Steen, 2007, chap 4.3.1 on p. 141 - 142):

- **Connectivity**: TCP/IP provides the *Socket* primitive to create that end point and also the *Bind* primitive to bind a local address to that socket. The *Connect* primitive then provides the functionality to establish a connection.
- **Ability**: TCP/IP provides the two primitives *Send* and *Receive* to simply send and receive data via the connection.
- **Integrity**: TCP/IP provides several mechanisms to ensure that there has been no data loss when sending or receiving messages. On the other hand this makes the protocol slower than other protocols such as the Real-Time Transport Protocol (RTP) or the User Datagram Protocol (UDP).

The requirements of **Intelligibility** and **Executability** do not come under the area of responsibility of TCP/IP and thus other standards have to take place, such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON) to structure the messages in a standardized manner, and the message parsing implementation to call the desired instructions.

There are other socket primitives than the above mentioned to provide the functionality of TCP/IP (Tanenbaum & Steen, 2007, ch. 4.3.1 on p. 141). The referenced book (Tanenbaum & Steen, 2007) provides useful and valuable information on this topic.

It is also important to understand that this was just a round-up of how one can make basic use of the socket primitives to implement message passing in an application. There is plenty to discuss about how message passing is realized, such as the Open Systems Interconnection Reference Model (OSI) which was developed by the International Organization for Standardization (ISO) to model the different layers of network oriented communication (Tanenbaum & Steen, 2007, ch. 4.1.1 on p. 116). Furthermore, the Operating System (OS) has to reserve local memory to provide a buffer for the in- and outgoing messages.

1.4 Asynchronous vs. Synchronous Message Passing

After discussing a protocol to handle the socket primitives, it is necessary to consider one major difference in message passing models, which is whether

to use asynchronous or synchronous message passing. Passing a message synchronously means that the called message send routine returns after the request has been successfully transmitted (or an error occurred). Receiving a message synchronously means that the called message receiving routine reads a specific amount of bytes from the socket and returns that message.

To pass or receive a message asynchronously, an additional application layer i.e. middleware is introduced. The call of a send routine can either return when the middleware took over the transmission of the request, or when it successfully sent the request to the receiver, or when it successfully sent the request to the receiver, assuring this by a corresponding message (Tanenbaum & Steen, 2007, ch. 4.1 on p. 125).

The middleware can also perform some preparatory work e.g. separating standalone request bit strings and parsing them into a data structure and returning it when calling the receive routine of the message passing middleware.

One key model to provide such middleware functionalities are message queues. A message queue is a queue to store messages by the First In - First Out (FIFO) principle. This enables the application to asynchronously send and receive messages by offering an incoming message queue and an outgoing message queue. This makes all the send and receive mechanisms fully transparent to the application (Tanenbaum & Steen, 2007, ch. 4.3.2 on p. 145 - 147).

2 Message-Passing Interface Standard

To pass messages between processes is adding large overhead to the communication model. Nevertheless, providing a well defined interface to control exactly how messages are passed regains the ability to write High-Performance Computing (HPC) applications.

The Message-Passing Interface (MPI) is a message-passing library interface specification, created for high performance and scalable distributed systems where high-efficiency is needed. The MPI standard was designed by the Message-Passing Interface Forum (MPIF) which is an open group with representatives from many organizations. The current version of the MPI standard is MPI-3.0 (Message Passing Interface Forum, 2012, ch. Abstract/ii & ch. Acknowledgements/xx & ch. 1.1 on p. 1).

The MPIF aims to offer a standard which establishes a practical, portable, efficient and flexible way of implementing message-passing in various high-level programming languages (e.g. C, C++, Fortran) (Message Passing Interface Forum, 2012, ch. History/iii). Furthermore, the MPI simplifies the communication primitives and brings them to an abstraction level to perfectly fit the programmer's needs of writing efficient and clean code for such HPC

distributed systems (Tanenbaum & Steen, 2007, ch. 4.3.1 on p 143).

TCP/IP does not fit those requirements. While the socket primitives *read* and *write* are sufficient for several general-purpose protocols (managing the communication across networks), they are insufficient for high-speed interconnection networks such as super computers or server clusters. The MPI standard offers a set of functions and datatypes with which the software developer is able to explicitly execute synchronous and asynchronous message passing routines (Tanenbaum & Steen, 2007, ch. 4.3 on p. 143).

3 Definition

The given task of Prof. Dr. Christin Schmidt was to define the term *Message Passing*. After discussing the theoretical basis and practical implementations, message passing is a term in distributed systems to characterize a communication model that deals with messages:

> *Message passing in distributed systems is a model to exchange messages within a process pair by making use of several standards and implementation details. The specifically used message passing model can diverge extremely in its provided functionality and defines how to provide the connection and send or receive messages.*

However, no message passing model defines the medium that transports the messages nor the outcome of a message and thus the following delimitation has to be made:

> *The physical conditions and the implementation of executing the desired instructions is not defined by the used message passing model.*

4 Message Passing in Erlang

4.1 Introduction

Erlang is a functional, declarative programming language that was written for the need of real-time, non-stop, concurrent, very large and distributed system applications (Armstrong, 1996, chap. 1 / p. 1). While the language was originally designed by Joe Armstrong, Robert Virding and Mike Williams for Ericsson (a Swedish telecommunications provider) in 1986, it finally became open source in 1998, thanks to the open source initiatives lead by Linux (Däcker, 2000, chap. 8 on p. 39).

Joe Armstrong, who started to design Erlang by adding functionality to Prolog, named the new programming language after the Danish mathematician Agner Krarup Erlang (creator of the Erlang loss formula) following the tradition of

naming programming languages after dead mathematicians (Däcker, 2000, chap. 4.1 on p. 13).

Erlang uses a native, asynchronous message passing model to communicate between light weight processes, also called actors (Armstrong, 1996, chap. 1 on p. 1). Erlang does not make heavy use of the executing OS to provide the described concurrency model, which implicitly means that Erlang decouples the underlying OS and thus providing a cross-platform and transparent message passing model (Armstrong, 1996, chap. 1 & 3 on p. 1 - 3).

4.2 Concurrency in Erlang

Erlang provides semantics and built-in functions to parallelize applications by message passing (Ericsson AB, 2015, ch. 4.3 on p. 95 - 104):

- spawn(*Module*, *Exported Function*, *List of Arguments*)
  Function which creates a new actor by running the *Exported Function* with the *List of Arguments* located in the *Module* (a set of functions located in one file) and which returns the Process Identifier (pid), which uniquely identifies the created actor.

- The receive construct allows the function executed by an actor to receive messages by using a message queue.
- The ! operator sends the right-handed term to the left handed pid. The right-handed term is the sent message.
- self()
  Function returning the pid of the actor executing the function.

Although the semantics and functions described above may require some foreknowledge on the Erlang programming language, it was important for me to show that Erlang is not only offering an easily intelligible interface for concurrency, it rather is a concurrent functional programming language. Nevertheless, there is plenty to discuss about the concurrency model of Erlang e.g. how to manage processes or handle errors. I recommend to take a look at the referenced official Erlang documentation or to *learn you some Erlang* on `www.learnyousomeerlang.com`.

4.3 Implementation of a Simple Diffie-Hellman Key Exchange Algorithm

The following implementation of a Diffie-Hellman key exchange algorithm in Erlang is simplified. Please note that this Erlang application is not in any way acceptable as an adequate implementation of the Diffie-Hellman key exchange

algorithm for use in the field! The code demonstrates the simplicity of implementing concurrent applications in Erlang by using message passing. Due to the need for a pow function returning the datatype *Integer* I implemented my own pow function see Appendix A. The sourcecode is hosted by Github:

`https://github.com/c-bebop/message_passing`

Licensed under the MIT License. You're welcome to contribute!

```
%%% @author      Florian Willich
%%% @copyright  The MIT License (MIT) Copyright (c) 2014
%%%              University of Applied Sciences, Berlin, Germany
%%%              Florian Willich
%%%              For more detailed information, please read the
%%%              licence.txt in the erlang root directory.
%%% @doc         This module represents a simple implementation of
%%%              the Diffie-Hellman key exchange algorithm and is
%%%              part of my technical report 'Introductory Guide
%%%              to Message Passing in Distributed Systems'.
%%% @end
%%% Created 2015-06-06

-module(diffie_hellman).
-author("Florian Willich").
-export([ computeMyPublicComponentKey/3,
          computeSharedPrivateKey/3,
          startKeyExchange/5,
          listenKeyExchange/2,
          startExample/0,
          startRemoteExample/1,
          startRemoteExample/5]).

%%% @doc  Public data represents the data which is publicly shared
%%%       within two communication partners when exchanging keys
%%%       with the Diffie-Hellman key exchange algorithm.
%%%       p: The public prime number
%%%       g: The public prime number (1 ... p - 1)
%%%       componentKey: The computed component key
%%%       pid: The pid of the one who instantiated this record
%%%       name: The name of whom creates this public data.
-record(publicData, {p, g, componentKey, pid, name}).

%%% @doc  Returns the value G to the power of MySecretKey Modulo P
%%%       which is the public component key for the Diffie-Hellman
%%%       key exchange algorithm.
%%%       For more information see http://goo.gl/pzdiH
%%% @end
-spec computeMyPublicComponentKey(pos_integer(), pos_integer(),
    pos_integer()) -> pos_integer().
computeMyPublicComponentKey(P, G, MySecretKey) ->
  my_math:pow(G, MySecretKey) rem P.

%%% @doc  Returns the value ComponentKey to the power of
%%%       MySecretKey Modulo P which is the private
%%%       shared key for the Diffie-Hellman key exchange
%%%       algorithm.
%%%       For more information see http://goo.gl/pzdiH.
```

```
%%% @end
-spec computeSharedPrivateKey(pos_integer(), pos_integer(),
    pos_integer()) -> pos_integer().
computeSharedPrivateKey(P, ComponentKey, MySecretKey) ->
  my_math:pow(ComponentKey, MySecretKey) rem P.

%%% @doc  Starts the Diffie-Hellman key exchange algorithm by
%%%       taking P (a prime number), G (1 ... P - 1), MySecretKey
%%%       is the secret integer of the one who executes this
%%%       function and the PartnerPID which is the PID of the
%%%       communication partner with whom a key exchange shall be
%%%       initiated.  MyName shall be the name of the executing
%%%       partner. This function sends the term
%%%       {startKeyExchange, PublicData} to the PartnerPID where
%%%       PublicData is of type publicData. Afterwards, the
%%%       function starts a receive construct which is receiving
%%%       the following:
%%%       {componentKey, PublicData}:
%%%           The message including all information needed for
%%%           computing the private shared key and then prints it
%%%           out.
%%%       UnexpectedMessage:
%%%           Prints out any unexpected incoming message and
%%%           calls a recursion.
%%%       After 3000 milliseconds:
%%%           The function will return timeout.
%%% @end
-spec startKeyExchange(pos_integer(), pos_integer(), pos_integer(),
     term(), string()) -> term() | {error, atom()}.
startKeyExchange(P, G, MySecretKey, PartnerPID, MyName) ->
  MyComponentKey = computeMyPublicComponentKey(P, G, MySecretKey),
  MyPublicData = #publicData{p = P, g = G, componentKey =
      MyComponentKey, pid = self(), name = MyName},
  PartnerPID ! {startKeyExchange, MyPublicData},

  receive

    {componentKey, #publicData{p = P, g = G, componentKey =
        PartnerComponentKey, pid = PartnerPID, name = PartnerName}}
         ->
      PrivateSharedKey = computeSharedPrivateKey(P,
          PartnerComponentKey, MySecretKey),
      printSharedPrivateKey(self(), MyName, PartnerName,
          PrivateSharedKey);

    UnexpectedMessage ->
      printUnexpectedMessage(UnexpectedMessage),
      startKeyExchange(P, G, MySecretKey, PartnerPID, MyName)

  after 3000 ->
    {error, timeout_after_3000_ms}

  end.

%%% @doc  Listens on Messages to start the Diffie-Hellman key
%%%       with the transferred MySecretKey and MyName
%%%       which shall be the name of the executing partner.
```

```
98 %%%       exchange by starting the following receive construct:
99 %%%       {startKeyExchange, PublicData}:
100 %%%           The message including all information needed to
101 %%%           start the key exchange by computing the own public
102 %%%           data which will then be send to the PartnerPID as
103 %%%           follows: {componentKey, MyPublicData}.
104 %%%           Afterwards, the private shared key will be printed
105 %%%           out and the function calls a recursion.
106 %%%       terminante:
107 %%%           Prints out that this function terminates with the
108 %%%           executing PID and returns ok.
109 %%%       UnexpectedMessage:
110 %%%           Prints out any unexpected incomping message and
111 %%%           calls a recursion.
112 %%% @end
113 -spec listenKeyExchange(pos_integer(), string()) -> term().
114 listenKeyExchange(MySecretKey, MyName) ->
115   receive
116
117     {startKeyExchange, #publicData{p = P, g = G, componentKey =
          PartnerComponentKey, pid = PartnerPID, name = PartnerName}}
           ->
118       MyComponentKey    = computeMyPublicComponentKey(P, G,
            MySecretKey),
119       MyPublicData      = #publicData{p = P, g = G, componentKey =
            MyComponentKey, pid = self(), name = MyName},
120       PartnerPID ! {componentKey, MyPublicData},
121       PrivateSharedKey  = computeSharedPrivateKey(P,
            PartnerComponentKey, MySecretKey),
122       printSharedPrivateKey(self(), MyName, PartnerName,
            PrivateSharedKey),
123       listenKeyExchange(MySecretKey, MyName);
124
125     terminate ->
126       io:format("~p terminates!~n", [self()]),
127       ok;
128
129     UnexpectedMessage ->
130       printUnexpectedMessage(UnexpectedMessage),
131       listenKeyExchange(MySecretKey, MyName)
132
133   end.
134
135 %%% @doc  Prints out the UnexpectedMessage as follows:
136 %%%       Received an unexpected message: 'Unexpected Message'
137 %%% @end
138 -spec printUnexpectedMessage(string()) -> term().
139 printUnexpectedMessage(UnexpectedMessage) ->
140   io:format("Received an unexpected message: ~p~n", [
        UnexpectedMessage]).
141
142 %%% @doc  Prints out the shared private key as follows:
143 %%%       'MyName' ('PID'): The shared private Key,
144 %%%       exchanged with 'PartnerName' is: 'SharedKey'
145 %%% @end
146 -spec printSharedPrivateKey(term(), string(), string(), string())
      -> term().
```

```
147 printSharedPrivateKey(PID, MyName, ParnterName, SharedKey) ->
148   io:format("~p (~p): The shared private Key, exchanged with ~p is:
          ~p~n", [MyName, PID, ParnterName, SharedKey]).
149
150 %%% @doc  Starts a key exchange example by spawning the Alice
151 %%%       process, which executes the listenKeyExchange function
152 %%%       with MySecretKey = 15, and the Bob process, which
153 %%%       exectues the startKeyExchange function with P = 23,
154 %%%       G = 5, MySecretKey = 6 and PartnerPID = Alice. Returns
155 %%%       {Alice, Bob} (pids).
156 %%%
157 -spec startExample() -> term().
158 startExample() ->
159   Alice   = spawn(diffie_hellman, listenKeyExchange, [15, "Alice"])
          ,
160   Bob     = spawn(diffie_hellman, startKeyExchange, [23, 5, 6,
          Alice, "Bob"]),
161   {Alice, Bob}.
162
163 %%% @doc  Starts a key exchange remote example by spawning the
164 %%%       Alice process, which executes the listenKeyExchange
165 %%%       function with MySecretKey = 15, and the Bob process,
166 %%%       located on the RemoteNode, which exectues the
167 %%%       startKeyExchange function with P = 23, G = 5,
168 %%%       MySecretKey = 6 and Alice PID.
169 %%%       Returns the pids of Alice and Bob.
170 %%% @end
171 -spec startRemoteExample(atom()) -> term().
172 startRemoteExample(RemoteNode) ->
173   Alice   = spawn(RemoteNode, diffie_hellman, listenKeyExchange, [
          15, "Alice"]),
174   Bob     = spawn(diffie_hellman, startKeyExchange, [23, 5, 6,
          Alice, "Bob"]),
175   {Alice, Bob}.
176
177 %%% @doc  Starts a key exchange remote example by spawning the
178 %%%       Alice process, which executes the listenKeyExchange
179 %%%       function with AliceSecretKey, and the Bob process,
180 %%%       located on the RemoteNode, which exectues the
181 %%%       startKeyExchange function with P, G,
182 %%%       BobSecretKey and Alice PID.
183 %%%       Returns the pids of Alice and Bob.
184 %%% @end
185 -spec startRemoteExample(atom(), pos_integer(), pos_integer(),
        pos_integer(), pos_integer()) -> term().
186 startRemoteExample(RemoteNode, P, G, BobSecretKey, AliceSecretKey)
        ->
187   Alice   = spawn(RemoteNode, diffie_hellman, listenKeyExchange, [
          AliceSecretKey, "Alice"]),
188   Bob     = spawn(diffie_hellman, startKeyExchange, [P, G,
          BobSecretKey, Alice, "Bob"]),
189   {Alice, Bob}.
```

4.4 Running the Application

To run the applicatoin (4.3) in a Linux shell one shall do the following (assuming that Erlang is already installed, help can be found at `http://www.erlang.org/doc/installation_guide/INSTALL.html`).

4.4.1 Preparatory Work

Open a Linux shell, go to the *source* directory of the Erlang code and type:

```
$ erl -sname bob
```

Which will output (meta data depends of the executing system):

```
Erlang/OTP 17 [erts-6.3] [source] [64-bit] [smp:4:4] [async-threads
    :10] [hipe] [kernel-poll:false]

Eshell V6.3  (abort with ^G)
(bob@localhost)1>
```

The option *-sname bob* tells the Erlang shell to run on a node called *bob*. First one shall compile the two modules and afterwards:

```
(bob@localhost)1> c(diffie_hellman).
{ok,diffie_hellman}
(bob@localhost)2> c(my_math).
{ok,my_math}
```

4.4.2 Running the Application on One Node

Now execute the *startExample* function as follows:

```
(bob@localhost)3> diffie_hellman:startExample().
"Alice" (<0.50.0>): The shared private Key, exchanged with "Bob" is
    : 2
"Bob" (<0.51.0>): The shared private Key, exchanged with "Alice" is
    : 2
{<0.50.0>,<0.51.0>}
```

Executing the function *startExample* spawns a process that executes the *listenKeyExchange* function with *15* as the *private key* and *"Alice"* as the name and binds the returning pid to the value called Alice. Afterwards, *startExample* spawns another process that executes the *startKeyExchange* function with *23* for *P*, *5* for *G*, *6* for the *private key*, the pid of Alice and *"Bob"* as the name which binds the returning pid to the value called Bob. At the end the function returns the two pids of *Alice* and *Bob*. As we can see from the output, Alice has the pid <0.50.0> and the calculated private key, exchanged with Bob is 2. Bob has the pid <0.51.0> and has computed the same private key.

4.4.3 Running the Application on Two Nodes

Now run the key exchange algorithm on two nodes. Open another shell, go to the *source* directory of the Erlang code and type:

```
$ erl -sname alice
```

Which will output (meta data depends of the executing system):

```
1 Erlang/OTP 17 [erts-6.3] [source] [64-bit] [smp:4:4] [async-threads
      :10] [hipe] [kernel-poll:false]
2
3 Eshell V6.3  (abort with ^G)
4 (alice@localhost)1>
```

This means we're now running another Erlang shell on the node alice (compile both files in the new Erlang shell, too). Now in the Erlang shell running the node *bob* execute the function *startRemoteExample* with the atom *alice@localhost* as the transferred parameter to specify on which node the actor shall be spawned that executes the *listenKeyExchange* function:

```
1 (bob@localhost)4> diffie_hellman:startRemoteExample(alice@localhost
      ).
```

This will output the following:

```
1 {<9879.54.0>,<0.58.0>}
2 "Alice" (<9879.54.0>): The shared private Key, exchanged with "Bob"
       is: 2
3 "Bob" (<0.58.0>): The shared private Key, exchanged with "Alice" is
      : 2
```

All transferred parameters are the same with the difference that the *Alice* actor is now located on the node *alice*. The output is still printed out in the *bob* node, since the Erlang io system recognizes where the process is spawned from and sends all the output to it (Ericsson AB, 2015, ch. 4.3.4 on p. 104).

A my_math Erlang Module

```
%%% @author     Florian Willich
%%% @copyright  The MIT License (MIT) Copyright (c) 2014
%%%              University of Applied Sciences, Berlin, Germany
%%%              Florian Willich
%%%              For more detailed information, please read the
%%%              licence.txt in the erlang root directory.
%%% @doc        This is my math module for mathematical
%%%              functions not provided by the erlang standard
%%%              library.
%%% @end
%%% Created 2015-06-06

-module(my_math).
-author("Florian Willich").
-export([pow/2]).

%%% @doc  Returns the value of Base to the power of Exponent.
%%%       If Base and Exponent is 0 the function returns
%%%       {error, undefined_arithmetic_expression},
%%%       The motivation to implement this function was that
%%%       there is no erlang standard library pow function
%%%       returning an integer.
%%% @end
-spec pow(integer(), integer()) -> number() | {error, atom()}.
pow(0, 0) ->
  {error, undefined_arithmetic_expression};

pow(Base, 0) ->
  case Base < 0 of
    true -> -1;
    false -> 1
  end;

pow(Base, Exponent) ->
  case Exponent < 0 of
    true -> 1 / pow(Base, -Exponent, 0);
    false -> pow(Base, Exponent, 0)
  end.

%%% @doc  Returns the value of Base to the power of Exponent.
%%%       Acc should be 0 for initiating computation.
%%%       The motivation to implement this function was that
%%%       there is no erlang standard library pow function
%%%       returning an integer.
%%% @end
-spec pow(pos_integer(), non_neg_integer(), non_neg_integer()) ->
    integer().
pow(_, 0, Acc) -> Acc;

pow(Base, Exponent, 0) ->
  pow(Base, Exponent - 1, Base);

pow(Base, Exponent, Acc) ->
  pow(Base, Exponent - 1, Acc * Base).
```

Acronyms

FIFO First In - First Out. 5

HPC High-Performance Computing. 5

IPC Inter-Process Communication. 3

ISO International Organization for Standardization. 4

JSON JavaScript Object Notation. 4

MPI Message-Passing Interface. 5, 6

MPIF Message-Passing Interface Forum. 5

OS Operating System. 4, 7

OSI Open Systems Interconnection Reference Model. 4

pid Process Identifier. 7, 12

RPC Remote Procedure Calls. 3

RTP Real-Time Transport Protocol. 4

TCP/IP Transmission Control Protocol / Internet Protocol. 3, 4, 6

UDP User Datagram Protocol. 4

XML Extensible Markup Language. 4

References

Armstrong, Joe. 1996. *Erlang - A survey of the language and its industrial applications. In: Proceedings of the symposium on industrial applications of Prolog (INAP96)*. `http://www.erlang.se/publications/inap96.ps`. The 9'th Exhibitions and Symposium on Industrial Applications of Prolog. 16-18, October 1996. Hino, Tokyo Japan. [Online. Accessed 5th May 2015].

Däcker, Bjarne. 2000. *Concurrent functional programming for telecommunications: A case study of technology introduction*. `http://www.erlang.se/publications/bjarnelic.ps`. Licenciate Thesis, Department of Teleinformatics. TRITA-IT AVH 00:08, ISSN 1403-5286. Royal Institute of Technology, Stockholm, Sweden. [Online. Accessed 20th May 2015].

Ericsson AB. 2015. *Erlang/OTP System Documentation 6.4*. `http://www.erlang.org/doc/pdf/otp-system-documentation.pdf`. [Online. Accessed 31th May 2015].

Hebert, Fred. 2013. *Learn You Some Erlang for Great Good! : A Beginner's Guide*. `http://learnyousomeerlang.com/content`. No Starch Press. [Online. Accessed 20th May 2015].

Message Passing Interface Forum. 2012. *MPI: A Message-Passing Interface Standard Version 3.0*. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. [Online. Accessed 20th May 2015].

Tanenbaum, Andrew S., & Steen, Maarten Van. 2007. *Distributed Systems: Principles and Paradigms (Second Edition)*. Pearson Prentice Hall. ISBN 0-13-239227-5.

This document was written with LaTeX
Typeface: Open Sans by Steve Matteson.