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 8, 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 (**?**, 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. When you read this 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 read those words, a message is passed from one entity (me, the writer) to another entity (you, the reader) which is obviously non trivial considering all the assumptions we would've to make to actually perform this message passing model of human beings.

Message passing in distributed systems e.g. computer science 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). Where those processes are located, on one or on different systems, is subsequent to the provided functionality.

1.2 Basic Requirements in Theory

The following items are the basic requirements (in theory) 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**: Send 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 have been developed several message passing models resulting in well defined standards and concrete implementations in the last decades. To make it easier to understand the relations between the theoretical requirements and a concrete implementation, I will take the Transmission Control Protocol / Internet Protocol (TCP/IP) which uses several socket primitives as example (**?**, chap 4.3.1 on p. 141 - 142):

- **Connectivity**: A communication end point has to be provided so the application can write data to and read data from: 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 ?over? the connection. (A little but important detail: The Operating System (OS) has to reserve local memory to provide a buffer for the in- and outgoing messages)
- **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 more slow 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 depend on TCP/IP and thus other standards have to take place to provide those requirements such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON) to structure the messages in a standardized manner and last but not least the message parsing implementation to call the desired instructions.

There are other primitives than the above mentioned to provide the functionality of TCP/IP. If you're interested in reading more, I recommend you the referenced book (**?**, ch. 4.3.1 on p. 141).

It is also important to understand, that this was just a round-up of how you could basically make use of the socket primitives to implement message passing in your 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 (**?**, ch. 4.1.1 on p. 116).

1.4 Asynchronous vs. Synchronous Message Passing

Now that we discussed a protocol to handle the socket primitives one major difference in message passing models is the whether we 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 our message asynchronously another application layer or middleware has to take place. The call of a send routine can either return when the middleware (**?**, ch. 4.1 on p. 125):

1. took over the transmission of the request.
2. successfully send the request to the receiver.
3. successfully send the request to the receiver, assuring this by a corresponding message.

The middleware can also perform some preparatory work e.g. separating stand alone request strings and parsing them into data structure and returning this more easy to parse data 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 and outgoing message queue. This makes all the send and receive mechanisms fully transparent to the application (**?**, ch. 4.3.2 on p. 145 - 147)

2 Message-Passing Interface Standard

The Message-Passing Interface (MPI) is a message-passing library interface specification, made for high performance and scale able 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 (**?**, ch. Abstract/ii & Acknowledgements/xx & 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) (**?**, ch. History/iii). Furthermore, the MPI simplifies the communication primitives and brings them to an abstraction level to perfectly fit the programmers needs of writing efficient and clean code for such distributed systems (**?**).

TCP 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 (**?**, ch. 4.3 on p. 143).

3 Message Passing in Erlang

3.1 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 (**?**, 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 (**?**, chap. 8 on p. 39).

Since there is a lot confusion with the naming of Erlang: Joe Armstrong, who started to design Erlang by adding functionality to Prolog, named it after the Danish mathematician Agner Krarup Erlang (creator of the Erlang loss formula) following the tradition of naming programming languages after dead mathematicians (**?**, chap. 4.1 on p. 13).

Erlang uses a native, asynchronous message passing model to communicate between light weight processes, also called actors (**?**, chap. 1 on p. 1). Erlang does not make heavy use of your 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 (**?**, chap. 1 & 3 on p. 1 - 3).

3.2 Concurrency in Erlang

Erlang provides semantics and built-in functions to parallelize your applications by message passing (**?**, 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 returns the Process Identifier (pid), which uniquely identifies an actor, of it.

- The receive construct that allows the function that is executed by an actor to receive messages by using a message queue.
- The ! operator which sends the right handed term to the left handed pid. The right handed term is the message we send.
- self()
  Function that returns the pid of the actor who executes the function.

Although the above described semantics and functions require a little knowledge on the Erlang programming language it was important for me to show you that Erlang is not only offering you an easy to understand interface for concurrency, it rather is a concurrent functional programming language. I recommend you to have a look into the referenced official Erlang documentation or to *learn you some Erlang* on `www.learnyousomeerlang.com`.

3.3 Implementation of a Simple Diffie-Hellman Key Exchange in Erlang

The following implementation of a Diffie-Hellman key exchange algorithm in Erlang is simple and short. I'm explicitly committing that its not complete nor error free. You can find the code on Github:

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

and is 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").
-compile(export_all).

%%% @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.
%%%       exchange by starting the following receive construct:
%%%       {startKeyExchange, PublicData}:
%%%           The message including all information needed to
%%%           start the key exchange by computing the own public
%%%           data which will then be send to the PartnerPID as
%%%           follows: {componentKey, MyPublicData}.
%%%           Afterwards, the private shared key will be printed
%%%           out and the function calls a recursion.
%%%       terminante:
%%%           Prints out that this function terminates with the
%%%           executing PID and returns ok.
%%%       UnexpectedMessage:
%%%           Prints out any unexpected incomping message and
%%%           calls a recursion.
%%% @end
-spec listenKeyExchange(pos_integer(), string()) -> term().
listenKeyExchange(MySecretKey, MyName) ->
  receive

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

    terminate ->
      io:format("~p terminates!~n", [self()]),
      ok;

    UnexpectedMessage ->
      printUnexpectedMessage(UnexpectedMessage),
      listenKeyExchange(MySecretKey, MyName)

  end.

%%% @doc  Prints out the UnexpectedMessage as follows:
%%%       Received an unexpected message: 'Unexpected Message'
%%% @end
-spec printUnexpectedMessage(string()) -> term().
printUnexpectedMessage(UnexpectedMessage) ->
  io:format("Received an unexpected message: ~p~n", [
      UnexpectedMessage]).

%%% @doc  Prints out the shared private key as follows:
%%%       'MyName' ('PID'): The shared private Key,
%%%       exchanged with 'PartnerName' is: 'SharedKey'
%%% @end
-spec printSharedPrivateKey(term(), string(), string(), string())
    -> term().
printSharedPrivateKey(PID, MyName, ParnterName, SharedKey) ->
  io:format("~p (~p): The shared private Key, exchanged with ~p is:
       ~p~n", [MyName, PID, ParnterName, SharedKey]).

%%% @doc  Starts a key exchange example by spawning the Alice
%%%       process, which executes the listenKeyExchange function
%%%       with MySecretKey = 15, and the Bob process, which
%%%       exectues the startKeyExchange function with P = 23,
%%%       G = 5, MySecretKey = 6 and PartnerPID = Alice. Returns
%%%       {Alice, Bob} (pids).
%%%
-spec startExample() -> term().
startExample() ->
  Alice   = spawn(diffie_hellman, listenKeyExchange, [15, "Alice"])
      ,
  Bob     = spawn(diffie_hellman, startKeyExchange, [23, 5, 6,
      Alice, "Bob"]),
  {Alice, Bob}.

%%% @doc  Starts a key exchange remote example by spawning the
%%%       Alice process, which executes the listenKeyExchange
%%%       function with MySecretKey = 15, and the Bob process,
%%%       located on the RemoteNode, which exectues the
%%%       startKeyExchange function with P = 23, G = 5,
%%%       MySecretKey = 6 and PartnerPID = Alice.
%%%       Returns Alice and Bob.
%%% @end
-spec startRemoteExample(atom()) -> term().
startRemoteExample(RemoteNode) ->
  Alice   = spawn(RemoteNode, diffie_hellman, listenKeyExchange, [
      15, "Alice"]),
  Bob     = spawn(diffie_hellman, startKeyExchange, [23, 5, 6,
      Alice, "Bob"]),
  {Alice, Bob}.
```

3.4 Running the Simple Diffie-Hellman Key Exchange in Erlang

4 Definition of Message Passing in Distributed Systems

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 whereas the physical conditions and the implementation of executing the desired instructions is undefined.

A my_math Erlang Module

```erlang
%%% @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).
```

B Acronyms

Acronyms

FIFO First In - First Out. 4

IPC Inter-Process Communication. 2

ISO International Organization for Standardization. 3

JSON JavaScript Object Notation. 3

MPI Message-Passing Interface. 4

MPIF Message-Passing Interface Forum. 4

OS Operating System. 3, 5

OSI Open Systems Interconnection Reference Model. 3

pid Process Identifier. 5, 6

RPC Remote Procedure Calls. 2

RTP Real-Time Transport Protocol. 3

TCP/IP Transmission Control Protocol / Internet Protocol. 2, 3

UDP User Datagram Protocol. 3

XML Extensible Markup Language. 3

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