

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	3
1.1	INTRODUCTION	3
1.2	BASIC REQUIREMENTS IN THEORY	3
1.3	REQUIREMENTS PROVIDER IN PRACTICE	4
1.4	ASYNCHRONOUS VS. SYNCHRONOUS MESSAGE PASSING	4
2	MESSAGE-PASSING INTERFACE STANDARD	5
3	MESSAGE PASSING IN ERLANG	6
3.1	Introduction	6
3.2	Concurrency in Erlang	6
3.3	Implementation of a Simple Diffie-Hellman Key Exchange in Erlang	7
3.4	Running the Simple Diffie-Hellman Key Exchange in Erlang	11
4	DEFINITION OF MESSAGE PASSING IN DISTRIBUTED SYSTEMS	13
A	my_math Erlang Module	14

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 (2, 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 (2, 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 (2, 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 an 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.learnyouesomeerlang.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. Please be aware that this Erlang application is not in any way acceptable as a practical tool to exchange keys with the Diffie-Hellman algorithm! For me, this is only a nice example to show how the Erlang message passing model works in practice. 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!

```

1 %%% @author      Florian Willich
2 %%% @copyright   The MIT License (MIT) Copyright (c) 2014
3 %%%              University of Applied Sciences, Berlin, Germany
4 %%%              Florian Willich
5 %%%              For more detailed information, please read the
6 %%%              licence.txt in the erlang root directory.
7 %%% @doc         This module represents a simple implementation of
8 %%%              the Diffie-Hellman key exchange algorithm and is
9 %%%              part of my technical report 'Introductory Guide
10 %%%             to Message Passing in Distributed Systems'.
11 %%% @end
12 %%% Created 2015-06-06
13
14 -module(diffie_hellman).
15 -author("Florian Willich").
16 -export([ computeMyPublicKey/3,
17            computeSharedPrivateKey/3,
18            startKeyExchange/5,
19            listenKeyExchange/2,
20            startExample/0,
21            startRemoteExample/1,
22            startRemoteExample/5]).
23

```

```

24 %%% @doc Public data represents the data which is publicly shared
25 %%% within two communication partners when exchanging keys
26 %%% with the Diffie-Hellman key exchange algorithm.
27 %%% p: The public prime number
28 %%% g: The public prime number (1 ... p - 1)
29 %%% componentKey: The computed component key
30 %%% pid: The pid of the one who instantiated this record
31 %%% name: The name of whom creates this public data.
32 -record(publicData, {p, g, componentKey, pid, name}).
33
34 %%% @doc Returns the value G to the power of MySecretKey Modulo P
35 %%% which is the public component key for the Diffie-Hellman
36 %%% key exchange algorithm.
37 %%% For more information see http://goo.gl/pzdiH
38 %%% @end
39 -spec computeMyPublicComponentKey(pos_integer(), pos_integer(),
    pos_integer()) -> pos_integer().
40 computeMyPublicComponentKey(P, G, MySecretKey) ->
41 my_math:pow(G, MySecretKey) rem P.
42
43 %%% @doc Returns the value ComponentKey to the power of
44 %%% MySecretKey Modulo P which is the private
45 %%% shared key for the Diffie-Hellman key exchange
46 %%% algorithm.
47 %%% For more information see http://goo.gl/pzdiH.
48 %%% @end
49 -spec computeSharedPrivateKey(pos_integer(), pos_integer(),
    pos_integer()) -> pos_integer().
50 computeSharedPrivateKey(P, ComponentKey, MySecretKey) ->
51 my_math:pow(ComponentKey, MySecretKey) rem P.
52
53 %%% @doc Starts the Diffie-Hellman key exchange algorithm by
54 %%% taking P (a prime number), G (1 ... P - 1), MySecretKey
55 %%% is the secret integer of the one who executes this
56 %%% function and the PartnerPID which is the PID of the
57 %%% communication partner with whom a key exchange shall be
58 %%% initiated. MyName shall be the name of the executing
59 %%% partner. This function sends the term
60 %%% {startKeyExchange, PublicData} to the PartnerPID where
61 %%% PublicData is of type publicData. Afterwards, the
62 %%% function starts a receive construct which is receiving
63 %%% the following:
64 %%% {componentKey, PublicData}:
65 %%% The message including all information needed for
66 %%% computing the private shared key and then prints it
67 %%% out.
68 %%% UnexpectedMessage:
69 %%% Prints out any unexpected incoming message and
70 %%% calls a recursion.
71 %%% After 3000 milliseconds:
72 %%% The function will return timeout.
73 %%% @end
74 -spec startKeyExchange(pos_integer(), pos_integer(), pos_integer(),
    term(), string()) -> term() | {error, atom()}.
75 startKeyExchange(P, G, MySecretKey, PartnerPID, MyName) ->
76 MyComponentKey = computeMyPublicComponentKey(P, G, MySecretKey),

```



```

77  MyPublicData = #publicData{p = P, g = G, componentKey =
      MyComponentKey, pid = self(), name = MyName},
78  PartnerPID ! {startKeyExchange, MyPublicData},
79
80  receive
81
82    {componentKey, #publicData{p = P, g = G, componentKey =
      PartnerComponentKey, pid = PartnerPID, name = PartnerName}}
      ->
83    PrivateSharedKey = computeSharedPrivateKey(P,
      PartnerComponentKey, MySecretKey),
84    printSharedPrivateKey(self(), MyName, PartnerName,
      PrivateSharedKey);
85
86  UnexpectedMessage ->
87    printUnexpectedMessage(UnexpectedMessage),
88    startKeyExchange(P, G, MySecretKey, PartnerPID, MyName)
89
90  after 3000 ->
91    {error, timeout_after_3000_ms}
92
93  end.
94
95  %%% @doc Listens on Messages to start the Diffie-Hellman key
96  %%% with the transferred MySecretKey and MyName
97  %%% 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 incoming 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", [
141         UnexpectedMessage]).
142
143 %%% @doc Prints out the shared private key as follows:
144 %%% 'MyName' ('PID'): The shared private Key,
145 %%% exchanged with 'PartnerName' is: 'SharedKey'
146 %%% @end
147 -spec printSharedPrivateKey(term(), string(), string(), string())
148     -> term().
149 printSharedPrivateKey(PID, MyName, PartnerName, SharedKey) ->
150     io:format("~p (~p): The shared private Key, exchanged with ~p is:
151         ~p\n", [MyName, PID, PartnerName, SharedKey]).
152
153 %%% @doc Starts a key exchange example by spawning the Alice
154 %%% process, which executes the listenKeyExchange function
155 %%% with MySecretKey = 15, and the Bob process, which
156 %%% executes the startKeyExchange function with P = 23,
157 %%% G = 5, MySecretKey = 6 and PartnerPID = Alice. Returns
158 %%% {Alice, Bob} (pids).
159 %%%
160 -spec startExample() -> term().
161 startExample() ->
162     Alice = spawn(diffie_hellman, listenKeyExchange, [15, "Alice"]),
163     Bob = spawn(diffie_hellman, startKeyExchange, [23, 5, 6,
164         Alice, "Bob"]),
165     {Alice, Bob}.
166
167 %%% @doc Starts a key exchange remote example by spawning the
168 %%% Alice process, which executes the listenKeyExchange
169 %%% function with MySecretKey = 15, and the Bob process,
170 %%% located on the RemoteNode, which executes the
171 %%% startKeyExchange function with P = 23, G = 5,
172 %%% MySecretKey = 6 and Alice PID.
173 %%% Returns the pids of Alice and Bob.
174 %%% @end
175 -spec startRemoteExample(atom()) -> term().
176 startRemoteExample(RemoteNode) ->
177     Alice = spawn(RemoteNode, diffie_hellman, listenKeyExchange, [
178         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 executes 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}.

```

3.4 Running the Simple Diffie-Hellman Key Exchange in Erlang

To run the listed implementation of a simple Diffie-Hellman key exchange in Erlang (3.3) in a Linux shell we shall do the following (assuming that Erlang is already installed on our machine, help can be found here http://www.erlang.org/doc/installation_guide/INSTALL.html):

We open a Linux shell and type:

```
$ erl -sname bob
```

Which will output (of course the output depends on the executing system e.g. how many physical cores we have etc.):

```
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 we compile the two modules and afterwards we run the *startExample* function:

```

(bob@localhost)1> c(diffie_hellman).
{ok,diffie_hellman}
(bob@localhost)2> c(my_math).
{ok,my_math}
(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 those two values 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. This means that the messages between the actors have been successfully transmitted and also the key exchange worked (the shared private key computed by the two communication partners is 2).

Now we run the key exchange algorithm on two nodes. To do so, we need to open another shell and type:

```
$ erl -sname alice
```

Which will output:

```
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)
(alice@localhost)1>
```

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

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

This will output the following:

```
{<9879.54.0>,<0.58.0>}
"Alice" (<9879.54.0>): The shared private Key, exchanged with "Bob"
is: 2
"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 Erlang recognizes where the ... ????

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

```
1 %%% @author      Florian Willich
2 %%% @copyright   The MIT License (MIT) Copyright (c) 2014
3 %%%              University of Applied Sciences, Berlin, Germany
4 %%%              Florian Willich
5 %%%              For more detailed information, please read the
6 %%%              licence.txt in the erlang root directory.
7 %%% @doc         This is my math module for mathematical
8 %%%              functions not provided by the erlang standard
9 %%%              library.
10 %%% @end
11 %%% Created 2015-06-06
12
13 -module(my_math).
14 -author("Florian Willich").
15 -export([pow/2]).
16
17 %%% @doc Returns the value of Base to the power of Exponent.
18 %%%      If Base and Exponent is 0 the function returns
19 %%%      {error, undefined_arithmetic_expression},
20 %%%      The motivation to implement this function was that
21 %%%      there is no erlang standard library pow function
22 %%%      returning an integer.
23 %%% @end
24 -spec pow(integer(), integer()) -> number() | {error, atom()}.
25 pow(0, 0) ->
26     {error, undefined_arithmetic_expression};
27
28 pow(Base, 0) ->
29     case Base < 0 of
30         true -> -1;
31         false -> 1
32     end;
33
34 pow(Base, Exponent) ->
35     case Exponent < 0 of
36         true -> 1 / pow(Base, -Exponent, 0);
37         false -> pow(Base, Exponent, 0)
38     end.
39
40 %%% @doc Returns the value of Base to the power of Exponent.
41 %%%      Acc should be 0 for initiating computation.
42 %%%      The motivation to implement this function was that
43 %%%      there is no erlang standard library pow function
44 %%%      returning an integer.
45 %%% @end
46 -spec pow(pos_integer(), non_neg_integer(), non_neg_integer()) ->
47     integer().
48 pow(_, 0, Acc) -> Acc;
49
50 pow(Base, Exponent, 0) ->
51     pow(Base, Exponent - 1, Base);
52
53 pow(Base, Exponent, Acc) ->
54     pow(Base, Exponent - 1, Acc * Base).
```

B Acronyms

Acronyms

FIFO First In - First Out. 5

IPC Inter-Process Communication. 3

ISO International Organization for Standardization. 4

JSON JavaScript Object Notation. 4

MPI Message-Passing Interface. 5

MPiF Message-Passing Interface Forum. 5

OS Operating System. 4, 6

OSI Open Systems Interconnection Reference Model. 4

pid Process Identifier. 6, 7, 12

RPC Remote Procedure Calls. 3

RTP Real-Time Transport Protocol. 4

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

UDP User Datagram Protocol. 4

XML Extensible Markup Language. 4

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