xxIntroductory Guide to Message Passing
In Distributed Systems

Introductory Guide to Message Passing
In Distributed Systems

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

June 19, 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 Passing

1.1 Introduction

To provide services and execute tasks, a distributed system has to have a proficient communication model implementation. There are several models of communication in distributed systems. With this technical report I am giving an introduction to different *message-oriented message passing communication models* (Tanenbaum & Steen, 2007, chap. 4.3 on p. 140 - 141).

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 (Tanenbaum & Steen, 2007, ch. 4 - 4.1.1 on p. 115 - 117).

1.2 Requirements in Theory

The following items are the theoretical basic requirements for a message passing model:

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

I intentionally left out the requirement of executability without which there is no effect when the message is passed. It is an implementation detail of the executed program code to call the desired instructions in the receiving process, and thus not a requirement of the message passing model in itself.

1.3 Requirements Provider in Practice

To provide the requirements mentioned above 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 chose 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 creating a socket 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** do not fall within the 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.

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 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, an important detail is that the Operating System (OS) has to reserve local memory to provide a buffer for the incoming 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. 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 stand-alone 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 facilities are message queues. A message queue is a data structure to store messages in 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 of 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

Passing messages between processes adds significant 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 & ch. 1.2 on p. 2).

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).

The MPIF offers a very detailed technical report (Message Passing Interface Forum, 2012) of the MPI standard which is not only defining the standard but offers detailed information on the organization and motivation.

3 Definition

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

Message passing in distributed systems is a model to exchange messages within a process pair. It defines how to establish the connection and how to send, receive and interpret messages. This is realized by making use of several standards and implementation details. The specifically used message passing model can diverge extremely in its provided functionality compared to others.

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 any message passing model.

4 Message Passing in Erlang

The preceding discourse repeatedly demonstrated that message passing does not only mean adding computational overhead but also a number of things the programmer has to keep track of. The message passing model implemented in Erlang simplifies and abstracts the use of message passing which enables the programmer to keep the focus on application logic. Erlang's transparent message passing model is therefore well suited as an illustration.

4.1 Introduction

Erlang is a functional, declarative programming language 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 (Dxcker, 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 (Dxcker, 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 provides 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*)
> A 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) returning the Process Identifier (PID), which uniquely identifies the created actor for addressing.
>
> 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()
> A 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 is important to point out 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. It is recommended to take a look at the referenced official Erlang documentation (Ericsson AB, 2015) or to *learn you some Erlang* on `www.learnyousomeerlang.com`.