Failure Detection over RDMA

Pavel Georgiev

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

The collection and exploitation of data are becoming of utmost importance for an increasing number of companies. Their data-driven services are the keystone in the way they interact with their customers and the value they can provide. To optimize their performance and throughput in the transfer of the data the use of the state-of-the-art networking solution Remote Direct Memory Access (RDMA) is getting more prevalent. Furthermore, the number of components in data centres keeps growing, which makes failures more of the expectation rather than the exception. To achieve fault tolerance in those circumstances a failure detection implementation over RDMA is needed.

In this paper, we explore the implementation of such a failure detector. The paper studies the methods chosen for detecting the crashes and reaching consensus over a suspected component failure within an asynchronous and distributed setting. It also outlines the details around the integration of RDMA methods for communication between nodes. Finally, the paper analyses the performance difference between the implementation using RDMA for communication versus the one using TCP/IP.

Acknowledgements

Acknowledgements go here.

Contents

Chapter 1

Introduction

Big Data market revenues are continuously growing with a projected increase of $42B in 2018 to $103B in 2027 [17] (see Appendix A, Figure A-1). Because of the amount of data companies collect and exploit, they strive for the most optimal data transfer and computation speeds they can achieve. The amount of data they work with not only directly ties with an increase in their revenue but also helps them scale their solutions to a global level.

It has been confirmed that Moore's law is coming to its natural end [19,20] so we can no longer rely on big improvements in performance by vertical scaling a machine by upgrading its hardware. However, the shift towards distributed computing allowed for the increase in computational power to continue. Gone are the days of a single server within a data centre. By following the trends on Top500.org [8] we can see that it has been the case for a long time that the work of a single logical service is done over many machines grouped in physical clusters. From data storage [11] to batch or stream data processing [12], the number of problems that could be solved in a distributed manner is countless. In order to reach peak throughput and minimal latency, companies are placing their focus on optimizing the intra-cluster and inter-cluster data transfer process. Regular TCP/IP connections, without any optimizations or specialized hardware, can become a bottleneck for the efficiency of a distributed system. With Remote Direct Memory Access (RDMA) implementations having the potential for better throughput, lower latency and less CPU interrupts [1] it is becoming clear why the use of RDMA methods is getting more and more prevalent. Wider adoption is enabled by the development of the technology and vendors releasing RDMA-compliant interconnecting hardware [6]. From network architecture standards like InfiniBand [3] to using RDMA over Converged Ethernet (RoCE) [4] to Internet-wide RDMA protocols like iWARP [5], there are an increasing number of ways of integrating RDMA into data centres with different fabrics and protocols.

Because of the shift toward distributed computing, the number of components within a system has substantially grown, which further increases the rate of expected failure. After analysis on data of 22 high-performance computing (HPC) systems, Gibson and Schroeder [7] suggest that the failure rate grows in a rate proportional to the number of processor chips in the system. With some systems in their data showing a rate of

1100 failures per year, it is almost expected that a node will fail during execution of a process. In order to maintain the expected performance of the system, there should be measures in place to make sure the system is fault tolerant and that a failure of one node won't bring the whole service down.

Similarly, the same principles hold on comparable systems that employ RDMA methods for their data transfer. Failure detection mechanisms and properties have been studied before [13, 14, 15, 16] but little work has been done on implementing one over RDMA with most of the work being done focusing on Replicated State Machines (RSM) [10].

1.1 Goals

The main goal is to implement a failure detector over RDMA.

In order to achieve this and evaluate the results the following contributions have been made:

- Implementation of a micro testing framework to emulate a multi-node setup in a distributed system environment
- Implementation of failure detection over TCP/IP
- Evaluation of adaptive timeout algorithm in the failure detector implementation
- Integration of RDMA methods of communication in the failure detector implementation
- Analysis of the performance difference between RDMA and TCP/IP based failure detector implementation

1.2 Report outline

- **Chapter 2**: Gives the necessary background needed to understand the nature of the work presented in the paper. Specifically, it introduces important concepts about failure detection (2.1) and RDMA (2.2).
- **Chapter 3**: Describes the implementation of a failure detector. It goes into details about the class of failure detector needed, implementation of dynamic timeout and choice of consensus protocol.
- **Chapter 4**: Describes the way the implementation is altered to work with RDMA.
- **Chapter 5**: Analyses the performance of the difference in performance of the failure detector when communication is done over RDMA versus when it is done over TCP/IP.
- **Chapter 6**: Summarizes the work that is done and observations have been made in the process of researching and implementing the desired outcomes. It also outlines possible ways the implementation could be improved or expanded upon.

1.3 Tools/

The source code for the failure detector has been written in C[1] language. In order to streamline the development workflow, the open-source build process management software CMake[2] language has been used. The library ZeroMQ[3] has been utilised for the initial development of the failure detector over TCP/IP. This is a high-performance messaging library that is aimed at distributed applications. Another tool that helped in the development was Docker[4]. It allowed for an easy way to simulate a distributed network.

[1]https://en.wikipedia.org/wiki/C_(programming_language)

[2]https://cmake.org/

[3]http://zeromq.org/

[4]https://www.docker.com

Chapter 2

Background

For the rest of the paper, if not mentioned otherwise, every system discussed would be a distributed system that consists of a finite set of nodes n. For simplicity would assume that every node runs a single process. Therefore, we have a set of processes $\pi = \{p_1, p_2, p_3 \dots p_n\}$. Also, we would assume that every node in the system is interconnected with the rest of the nodes by a reliable channel and can communicate with them by sending messages.

We would assume an asynchronous model of distributed computing for the systems we are discussing, which means that no timing assumptions are made. This allows for a great general model because we need not concern ourselves with delays caused by computations, network congestion, scheduled delays etc. Although we don't measure physical time, for convenience we would use a monotonically increasing virtual clock τ that increases at every event occurrence.

2.1 Failure Detection

In this section we are not concerned with the software implementation of a failure detector or the hardware that is running underneath; for implementation details refer to Chapter 3. Instead, we focus on the properties a failure detector has and discuss ways to detect failures in an asynchronous system.

2.1.1 Consensus Problem

In distributed computing, it is important to be able to achieve system reliability even in a presence of a failed component. This requires processes to agree on a certain aspect – computed value, elected leader, clock synchronization etc. Different approaches decide the consensus value differently, but all consensus protocols end with a unanimous decision at the end of their run. The consensus problem is bounded by the following properties [13]:

- **Termination**: every correct process eventually decides upon a value
- **Uniform integrity**: every process decides at most once

- **Agreement**: no two correct processes decide differently
- **Uniform validity**: if a process decides on a value v, then v was proposed by some process

There is also an extension of the general consensus problem called the *Uniform Consensus* problem, which enforces that no two processes decide differently.

2.1.2 Impossibility Result

An important thing to note is the proof by Fischer et al. [16], which states that in an asynchronous distributed system, even a single failed process can render the reach of consensus *impossible*. Simply put, this stems from the fact that in an asynchronous system we have no way of knowing whether a process has crashed or it is taking a long time to communicate with the rest of the nodes.

2.1.3 Unreliable Failure Detectors

There are several ways to remedy the impossibility of reaching consensus, explained in Subsection 2.1.2, by making a minimal set of assumptions on top of the asynchronous model. We would focus on the unreliable failure detector abstraction described by Chandra and Toueg [13], which allows us to build reliable distributed systems. To combat the difficulty of determining if a process has crashed or just behaving abnormally slow, the authors propose the introduction of failure detectors that can make mistakes.

In this abstraction, we have a set of distributed failure detectors. That set consists of local failure detector modules attached to every process in our system, which monitors the status of the rest of the processes in the system. Every failure detector module maintains a list of suspected crashes and can append or delete suspected processes from that list. For example, node p_i can suspect that node p_j has crashed and put it in its list of suspects. If later p_i decides that a mistake has been made it can remove p_j from the list. Also, two processes can have different lists of suspected failures.

Failure detectors can make mistakes so we should expect live processes to be suspected as crashed and the other way around. To be useful to us, failure detectors should also provide correct information about the status of the system. To judge the degree of trustworthiness of the information that failure detectors provide we classify them by their *completeness* and *accuracy*.

2.1.4 Failure Detector's Properties

Completeness

Failure detectors are classified into two groups based on their completeness [13]:

- **Strong completeness**: eventually *every* crashed process is permanently suspected by *every* correct process

- **Weak completeness**: eventually *every* crashed process is permanently suspected by *some* correct process

Completeness by itself is not useful because we can trivially satisfy strong completeness, by suspecting every process, if there are no limitations on the number of mistakes we can make.

Accuracy

By introducing the accuracy property, we have a way of classifying the number of mistakes expected from a failure detector. A failure detector can be classified as having:

- **Strong accuracy**: no process is suspected before it crashes
- **Weak completeness**: : at least one correct process is never suspected

Strong accuracy and even weak accuracy properties are hard to satisfy since there must be at least one alive process that is never suspected. Because of this Chandra and Toueg [13] introduce the concept of *eventual* satisfiability. The resulting *eventual strong accuracy* and *eventual weak accuracy* properties are much like their non-eventual, or often called perpetual, counterparts but only require the accuracy constraint to be satisfied after a certain point in time.

Combination of the completeness and accuracy properties give us 8 types of failure detectors.

For the rest of the paper, the classes of failure detectors would be referred by the notation given in Figure 1.

2.2 RDMA

Remote Direct Memory Access (RDMA) allows one computer to directly access the memory of another remote computer without involving the operating systems of either of the machines.

Bypassing the operating system (OS) gives the application direct access to the network adapter and allows the CPU to communicate directly with it. Because the need for context switches between user mode and kernel mode is eliminated, transfers can happen in parallel with other operations running on the same machine. RDMA also allows for zero-copy transfers. This means that no copies of data are needed to or from the memory buffer. This enables the data to be read directly from another node's memory. The result is a lower CPU load, higher throughput and lower latency.

Chapter 3

Failure detector implementation

3.1 Testing framework

3.2 Heartbeats

3.3 Adaptive timeout

3.4 PAXOS

Chapter 4

Failure detection over RDMA

Chapter 5

Analysis

Chapter 6

Conclusion

6.1 Overview

6.2 Future work

Of course you may want to use several chapters and much more text than here.

Bibliography