Replication in Rack-scale Systems

Roni Häcki (PhD student since December 2015)
System Group, Department of Computer Science, ETH Zürich
Advisor Timothy Roscoe

Abstract

Tightly coupled computer systems (rack-scale), which contain heterogeneous low latency connections within and between machines, are being built today. In such rack-scale systems, state has to be replicated to ensure correct operation. Even within machines there is a strong case for replication. In the future, the component machines of rack-scale systems will undergo partial failures as well as hot replacement of parts making replication more dynamic since reconfiguration is more frequent.

In a rack-scale environment existing replication algorithms do not work well because they are based on the assumption of a asynchronous network where message can be changed, reordered, arbitrarily delayed, or even dropped. Furthermore, with low latency connections between and within machines the propagation time no longer dominates changing another key assumptions.

Distributed replication algorithms need to be adapted to incorporating knowledge about the system like link properties and performance characteristics. This improves performance of replication within a rack-scale system and helps find a solution so that programmers do not have to deal with the additional complexity to enable fault tolerance.

1 Introduction

Replication between machines is a well known technique to implement high performance and fault tolerant services like Chubby [3] or the Hadoop Distributed File System [11]. In the future hardware failures have to be considered as part of the programming model of a machine [6], making it necessary to apply replication techniques between machines and within machines.

A rack-scale system consists of tens of machines that are tightly coupled with a low latency interconnect like InfiniBand, which provides strict ordering of messages and prevents message loss. Even within a controlled environment like a datacenter using 10/40G Ethernet connections, packets are mostly ordered and loss is unlikely [10]. Applying replication algorithms that make worst case assumption and do not consider these guarantees will not achieve in optimal performance. Furthermore, Remote Direct Memory Access (RDMA) primitives are more and more used [5, 9] to communicate between machines. RDMA changes the properties of the communication link by offloading functionality to the network card but also increases complexity for the programmer.

In a rack-scale system there are both cross machine links, and communication links within a machine. Core-to-core propagation time with cache coherent shared memory message passing such as FastForward [7] is in the order of a few hundred cycles, whereas over a network it can take up to a few micro seconds. Additionally, message passing channels guarantee that messages are received eventually in the order they were sent. In recent years, a machine itself is seen as a distributed system [2] and also operating systems [1] are designed this way. These operating system extensively use message passing to avoid scalability issues with complex cache coherency protocols, but also to prepare for the future where non-coherent memory might be managed by the operating system [6].

The different link characteristics and properties (some are shown in Table 1) make it hard to find a replication algorithm that can perform well in all cases. I will adapt suitable replication algorithms to incorporate the additional assumptions for certain link types and combine different adapted algorithms to improve replication performance in a rack-scale system.

Link type	**Latency**	**Guarantees**
10G Ethernet	$\sim$50 μs	-
InfiniBand	$\sim$1 μs	FIFO, no loss
Shared memory message passing	$\sim$500 ns	FIFO, no loss

Table 1: Comparison of communication link types

2 Approach

First, I will focus on consensus-based replication within a machine and the fail-stop and fail-recovery models. In future work I will consider Byzantine failures to handle

software failures or software bugs but this requires a different set of algorithms.

Algorithms that might work very well in an asynchronous network, might not be optimal in a multicore environment. Previous research [4] in the direction of consensus in a multicore machine indicates that performance depends not on the number of communication rounds, but the number of messages sent. One of the most popular consensus algorithms Multipaxos [8] does not try to reduce the number of messages. An alternative is 1Paxos [4] that is designed with a multicore machine in mind.

My preliminary results show that send/receive costs are the dominating factor for communication protocols within a multicore. 1Paxos still features a single node that sequentially broadcasts messages. This node often becomes the bottleneck.

By building broadcast trees, which use system knowledge and adapt to failures, I want to avoid the leader as a bottleneck and help scale replication within a machine to a larger number of cores. Moreover, I will consider using shared resources (L3 cache) for broadcast trees by using shared memory to send multiple messages at once. Shared memory can help improve the performance of subtrees that contain nodes within the same NUMA node.

To make the broadcast trees failure resilient, I need to be able to change the structure of the tree at runtime without introducing inconsistencies. Failure detection is often part of a consensus algorithm. Using the consensus algorithm to detect failures and adapt the broadcast tree by excluding failed nodes, is the approach I will take.

Failure recovery i.e. rejoining of nodes to the replication process is not only an important matter to handle failures, but also in the context of shutting down parts of a rack-scale system to save power or updating hardware while the system is running.

Overall the consensus protocol and the broadcast tree need to be highly machine aware to deliver high performance and dynamically reconfigurable to handle failures. With a solution to fault tolerant replication within a machine as a building block, I will combine multiple of them and extend them to a rack-scale system.

References

[1] A. Baumann et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[2] A. Baumann et al. Your computer is already a distributed system. Why isn't your OS? In *HotOS*, 2009.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[4] T. David, R. Guerraoui, and M. Yabandeh. Consensus Inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, 2014.

[5] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Apr. 2014.

[6] P. Faraboschi et al. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, May 2015.

[7] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, 2008.

[8] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998.

[9] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[10] D. R. K. Ports et al. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[11] K. Shvachko et al. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, 2010.