The Cost of Consensus: Synchronization in Distributed Systems

***Abstract*—**

***Index Terms*—**

I. Introduction

Coordination is essential for large-scale distributed applications. Handling the simplest operations in a distributed manner gave rise to unprecedented challenges. Different forms of coordination are used to handle a variety of tasks. Leader election and group membership is one way. Worrying about aspects of synchronization, concurrency, and distributed management is a huge burden on application developers. This is why many distributed coordinators were designed to be leveraged by those developers, such as ZooKeeper and Chubby. Other packages focus on one primitive, or aspect, of distributed coordination such as Amazon Simple Queue Service that focuses on queueing.

Locking is a powerful coordination primitive. It guarantees mutual exclusion when accessing critical sources. However, it is also widely used to provide a mean of synchronization between distributed applications. The choice of synchronization primitive is not an easy decision. Different applications have different characteristics. The amount of contention for example is cruicial on the choice of synchronization primitive and is highly dependent on the application type. The computing environment is of importance too. The latency of coordination and consensus have an effect on the performance of different primitives. The topic of synchronization protocols' pros and cons and comparison of both are widely studied in the literature of multiprocessors.

here is a need to reinvestigate synchronization protocols for large-scale distributed systems. In these systems communication latency can reach hundreds of milliseconds. This dramatic difference to the conventional multiprocessor environment might carry with it new revelations on the community's prejudice on synchronization protocols. General distributed coordination packages delivers basic coordination primitives to end users. ZooKeeper provides a simple API to manipulate hierarchically organized wait-free data objects, resembling a file system. These manipulations are guaranteed to be FIFO ordered and writes are linearizable. Using these primitives allow users creating more complex coordination primitives (*e.g.*, synchronization primitives). Chubby, on the other hand, provides locking with strong guarantees.

In this paper, we carry the first steps into realizing the question of synchronization protocols in distributed systems. We leverage ZooKeeper to coordinate between different machines. Synchronization protocols are then implemented using ZooKeeper's primitives. In our study we will display protocols shortcomings in different operation conditions. The protocols we will be focusing on are test-and-set and queues. After we map each one of these two to a favorable operation condition, we lay the ground for a reactive mechanism that, according to current operation, choose the better protocol to manage synchronization.

The rest of the paper is organized as follows. Section II describe the framework of our experiments and overview basic concepts and technologies used. A mathematical analysis is presented in Section III where we provide a model of observed latency. Experimental results are then displayed in Section IV. Finally, the paper concludes with a summary and future directions in Section V.

II. Framework and overview

A. testbed

describe machines and topology

B. consensus protocol

talk about consensus

C. Zookeeper

talk about basic operations here because they are mentioned in next subsection, talk also about modes of znode creation.

```
method acquire_queue_lock ()
(1) pathname = synchronous create sequenced file "/lock/name-"
(2) children = getChildren of folder "/lock"
(3) if ( pathname is the file with least sequence number)
(4)      lock acquired; return;
    else
(5)      call exist on file with maximum number smaller than me
         and set a watch on this exist

method process_exist_watch ()
(6) lock acquired; return;

method release_queue_lock ()
(7) delete "pathname"
```

Fig. 1: Pseudo code of acquiring and releasing queue locks

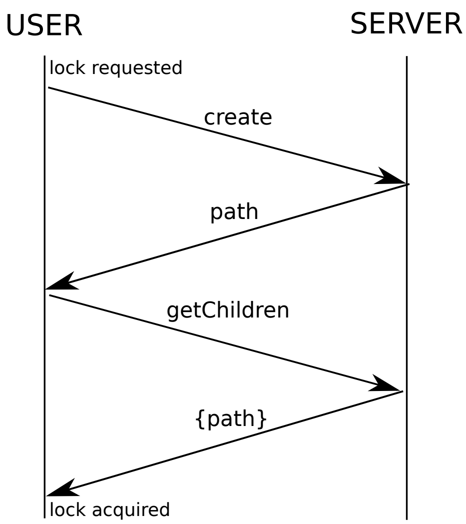

Fig. 2: A time diagram showing time required to acquire a queue lock if no other user is holding it

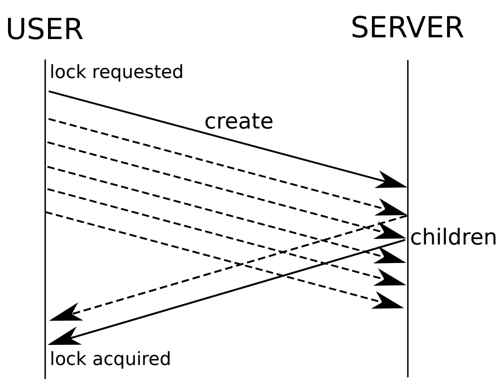

Fig. 3: A time diagram showing time required to acquire a queue lock if no other user is holding it using asynchronous calls. Dashed arrows from user to server are getChildren operations. Dashed arrows from server to user represent getChildren replies before the lock is acquired. Solid arrow from server to user is getChildren reply with user having the smallest sequenced file. The callback for asynchronous create is not shown.

D. synchronization primitives

One of the main points studied in this paper is the effectiveness of synchronization primitives (primitives) and their comparative behavior giving different network conditions. Here we give a summary of the primitives we consider. Primitives are queue locks and test-and-set (TAS). We divide primitives to synchronous and asynchronous depending on the client-server interactions. In the following we will summarize those primitives and how they translate in distributed systems using Zookeeper operations:

- Synchronous queue lock: a queue structure maintaing the process of acquiring a lock. If a user tried to acquire a lock while another was holding it, then it is queued. Thus, users acquire the lock based on the order in which they entered the queue, helping achieve fairness. This is implemented in Zookeeper by synchronously creating an ephemeral, sequenced znode. Each create operation will return the file name, indicating the sequence number. The client waits until it has the smallest sequence number. Then, it acquire the lock. Releasing the lock is done by merely deleting the corresponding file. Pseudo code is displayed in Figure 1. There are two possible execution paths. First, a user tries to acquire a lock while no one is holding it. In this case algorithm exist in line (4) after calling only two Zookeeper operations, namely synchronous *create* and *getChildren*. Second, another user is holding the lock when we try to acquire it. In this case the total latency is of operations *creat*, *getChildren*, and *exists*, in addition to waiting time until the *watch* returns to the client.
- Synchronous TAS: to acquire a lock, the user repeatedly execute TAS until it succeeds. TAS tests a boolean flag until it flips it from false to true. Traditionally, TAS surfaced in memory-sharing systems due to the advent of atomic TAS operations. In Zookeeper we are able to create a primitive that is similar in spirit to hardware TAS. A user tries to create a file with a known name (all users try to create the same file). If the file already existed that means that another user is holding the lock and create will return an exception. Otherwise, the user will hold the lock. To release the lock, the user deletes the file. We repeatedly try to acquire a lock by calling *create* in a busy loop. Alternatively, a watch on the file can be used to avoid busy waiting. Apparently, there is no order in entering the queue, hurting fairness. Execution path of TAS when no user is holding the lock is only one synchronous *create* call. Otherwise, it is the time until the user is the fastest to create the file. A more detailed analysis of wait times are found in the analysis section.
- Asynchronous primitives: We showed synchronous implementations of our primitives. In them, we use synchronous versions of Zookeeper operations. Thus, we block until we receive a reply from the server. It is found, however, that we can cut some latency by using asynchronous operations and continuously poll servers for desired state. For example, time required to acquire a lock when no user is holding it for a synchronous queue lock is two RTTs as shown in Figure 2. However, as shown in Figure 3 we can proceed with calling parallel *getChildren* operations after the asynchronous create. Latency in this case is one RTT plus the time required to create the file and half the latency between *getChildren* requests (remember we are assuming no user held the lock at that time). Likewise, different primitives and asynchronous TAS cut latency in the same way. describe all primitives and put pseudo codes of them if enough time.

III. Analysis of consensus cost

A. Round-trip time latency

display data we got from experiments (using ping for example) to get the distribution of RTTs. Maybe we should check with inter- and intra-datacenter communications. we discuss insights from the behavior of RTTs regarding their cost on consensus then we develop a model to describe RTTs, and develop a model to expect latency of getting all responses. In distributed systems communication overhead is much larger than multiprocessing systems. RTTs can reach up to a millisecond in systems in a single data center and can reach hundreds of milliseconds in geographically separated data centers. It is important to understand the behavior of RTTs and the factors that affect its value. Also, it is important to observe how different communication patterns affect observed RTTs. In this section we will show some results on simple experiments to observe the distribution of RTT values. Afterwards, we will analyze the effect of RTT distribution, control patterns, and number of systems on observed latency.

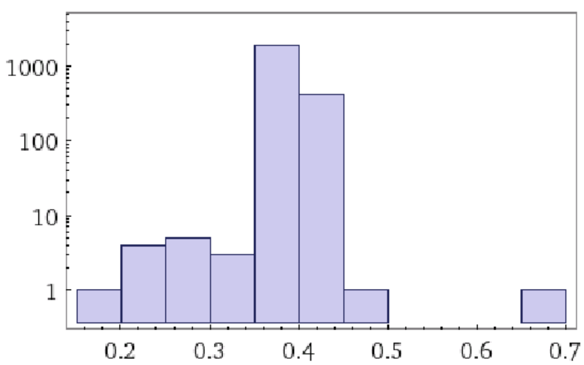

Fig. 4: probability density function of two machines i nthe same cluster. X-axis is RTT and y-axis is number of ocurrences

First, we show a probability density function of RTTs between two machines in the same vicinity in Figure **??**. It is apparent from the figure that most RTT values are clustered around the average, but also experience variation in values. What is interesting is that obtained results do not resemble traditionally used distributions to approximate them, namely exponential and Gaussian distributions. They are better approximated by a uniform distribution that captures the two largest bars in the displayed histogram.

Simple analysis of a distributed protocol's latency might be deceptive. Lets take 2-Phase commit (2PC) for example. In this protocol, two rounds of message exchange are required. An observer might naively expect the latency of each operation to be 2 RTTs. However, as we will show in our analysis, this is not the case. The importance of this observation is driven from the fact that coordination systems employ, in one way or another, a consensus or atomic broadcast protocols. These protocol exhibit the same behavior that we will demonstrate analytically in the rest of this section.

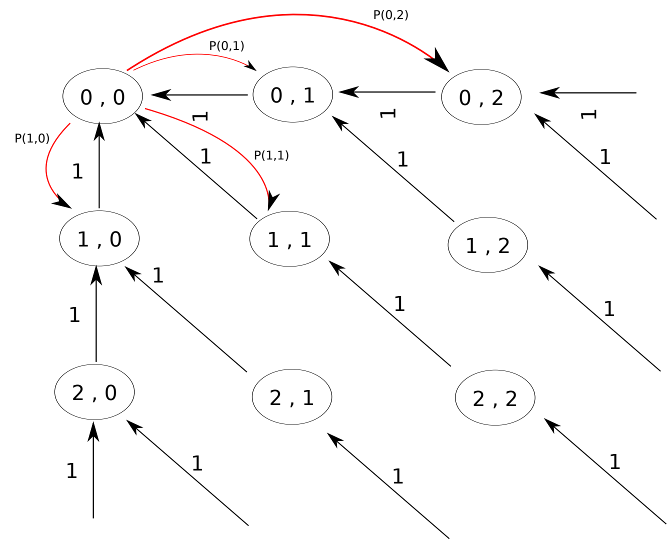

Fig. 5: Markov chain to model latency for one master and two slave servers

Consensus protocols requires the master coordinator to send control messages to associated slave coordinator. Although the average of RTTs when observed with a pair of servers, when we have the master coordinator waiting for more than one slave server the waiting time becomes

$$latency_i = max\{RTT_i^1, RTT_i^2, \ldots, RTT_i^n\} \qquad (1)$$

where $latency_i$ is the latency experienced to receive all replies from slaves for request i, and RTT_i^j is the RTT for request i for the communication between master and slave j. It is clear that the process $latency_i$ has an average larger than $R\bar{T}T$. We model the system as a Markov chain as the one represented in Figure 5 for the case of one master and two slaves. Each state represent the time until receiving the reply of the control message. State (i, j) for example denote that i and j time units are remaining until receiving a reply from slave 1 and 2 respectively. The transition probabilities are described as the following:

- A transition from state (i, j) to state (N($i-1$), N($j-1$)) for all i and j satisfying $i + j > 0$. N(i) returns i if it is positive or zero otherwise.
- A transition from state (0, 0) to state (i, j) with probability $\psi_i \psi_j$ where ψ_k is the probability distribution of the RTT process.

solve to find average using the model

B. Consensus latency

developa model to describe the latency of requests (such as the ones used for our baseline case, like zookeeper's paper). this will develop over previous section in addition to accounting the effect of service times in clients.

C. Synchronization primitives latency

barriers, test-and-set, queues

IV. Experimental evaluation

In this section, we will display results of our experiments. First, we will establish baseline performance results for Zookeeper. These results will aid us in reasoning for following experiments. Next, we implemented previously discussed synchronization primitives, namely Test-and-set, queues, and their asynchronous counterparts. We perform several experiments with them to test their performance on various conditions. if we did application subsection then add here.

A. Test bed

multiple physical machines The testbed consists of 5 quad-core Xeon type and clock machines with commodity 500GB 7.2k rpm disks. They are interconnected with 1Gbps Ethernet on a single switch. should we move this and expand it to section 2? I think a figure showing connections and showing that hamilton is connected to them would be a nice display. Average RTT for pings is (ping lat) Average network throughput is (throughput=). this for inter-cluster and cluster to hamilton.

B. Baseline performance

smoketest, zk-latencies and baseline In this section we will perform experiments to establish a baseline for later experiments. We would like to establish limits on the system. These limits represent workloads and environment conditions that will saturate the system. Workload is represented by the number of clients, number of requests per second, and the type of requests. Environment conditions are the number of zookeeper servers and condition of links connecting them.

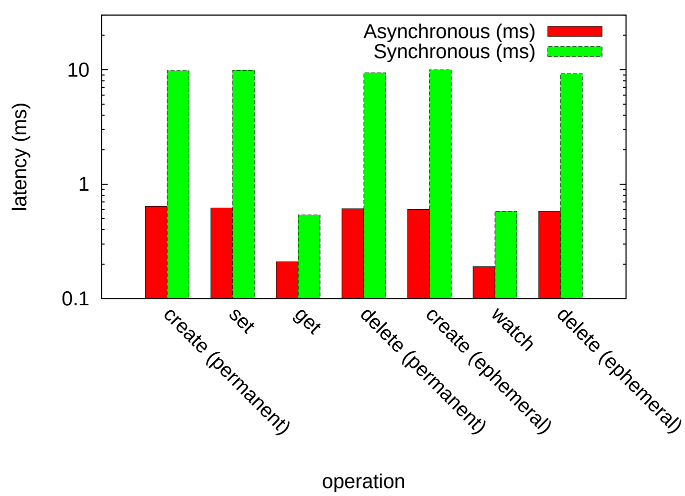

Fig. 6: Zookeeper basic operations latencies for a cluster of five servers

Our first experiment test basic Zookeeper operations' latencies. We test both synchronous and asynchronous versions of these operations. For testing we use zk-smoketest[1]. Each operation is run for a thousand time and we report average latency. Results are shown in Figure 6. Synchronous operations block until the operation is performed, thus giving an indication on the total time taken to perform the actual operation. Asynchronous operations on the other hand do not block. The figure shown that synchronous operations takes more than ten times the latency of asynchronous operations for *put* operations, and around double the latency for *get* operations. define put and get operation classes either here or before (section II).

C. synchronization primitives

test test-and-set and queues (as in paper: reactive synchronization

D. application performance

map reduce. effect of adding machines, effect of adding zookeeper servers.

V. Conclusions and Future Work

dummy citations [1]–[11].

References

[1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[2] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, , and B. Falsa. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2012.

[3] A.B. Hastings. Distributed lock management in a transaction processing environment. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 22 –31, oct 1990.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[5] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. pages 11–11, 2010.

[6] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. *Dependable Systems and Networks, International Conference on*, 0:245–256, 2011.

[7] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[8] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. *SIGOPS Oper. Syst. Rev.*, 28(5):25–35, November 1994.

[9] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[10] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

[11] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[1] https://github.com/phunt/zk-smoketest