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

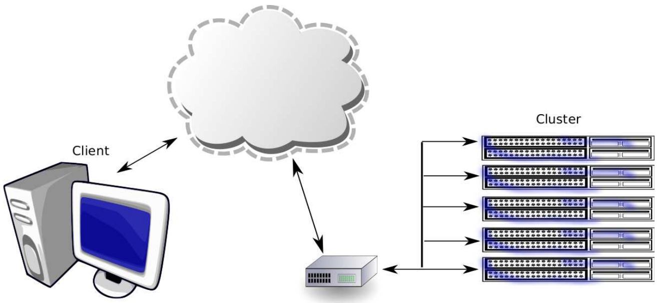


Fig. 1: Testbed setting used for evaluation

A. testbed

Our testbed consists of a cluster of five machines connected by a switch. These nodes will be the servers hosting Zookepper instances. Another machine, we will call it the client machine, will act as the user of Zookeeper. The Round-Trip Time (RTT) between machines in the cluster is around 0.1ms and around 0.2ms for client to cluster communication. Each cluster machine has a 5 quad-core Xeon type and clock with commodity 500GB 7.2k rpm disks. They are interconnected with a single 1Gbps Ethernet switch.

B. consensus protocol

It is important for distributed systems to have a well-defined consensus protocol. High latency and possibility of failures make achieving consensus an intricate task. For this matter consensus protocols such as Two-Phase Commit (2PC), Three-Phase Commit (3PC), and paxos were proposed. In this section we will focus on paxos due its use in distributed consensus servres, such as Chubby and, to some degree, Zookeeper. The use of paxos over other alternatives is its resilience to node and network failures. In addition, paxos does not need a fail recovery model.

At each iteration, a *proposer* initiates the protocol and send a proposal to other nodes. Each proposal is tagged with a unique sequence number. Other nodes are called *acceptors*. When acceptors receive proposals, they would either accept or reject. Accepting a proposal is essentially a promise not to accept other proposals with lower sequence numbers. When a majority (quorum) of servers accept a value, then the proposer can proceed with committing the proposal by sending a commit message to all followers.

C. Zookeeper

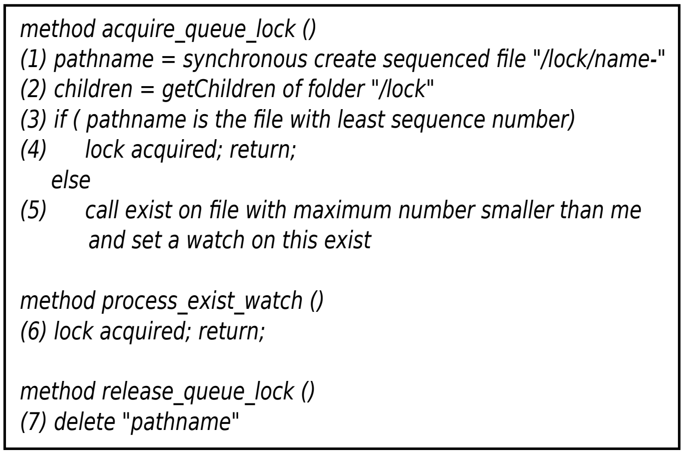
Zookeeper is an open-source coordination service for distributed systems. It runs as a set of replicated distributed servers. A leader election mechanism is used. All requests are forwarded to the leader. A consensus protocol is used to agree on received requests. A combined transaction and write-ahead logging is used in a variant of Paxos as a consensus protocol. Zookeeper levarage an *atomic broadcast* protocol in their consensus protocol implementation. Clients can connect to any of those replicas and issue operations. Zookeeper maintains an order of operations. Each update is stamped. This stamp can then be levaraged in higher level abstractions. Guarantees made by Zookeeper to clients' operations include the following:

- *Sequential consistency*: operations are executed in the order they were received.
- *Atomicity*: operations are either completely executed or aborted.
- *Single system image*: all servers maintain the same view.
- *Reliability*: updates are persistent.
- *Timeliness*: updates will be reflected on all servers within a specified time bound.

These guarantees will aid us in reasoning on the way we use Zookeeper operations. The set of offered operations enable more complex services such as coordination, synchronization, configuration management, and naming. These operations operate on a Unix-like filesystem tree architecture. Each file has a path and could be either *permanent* or *ephemeral*. Permenant files persist a client's disconnection whereas ephemeral files are deleted when a client disconnects. Those ”files” are called *znodes*. A useful zookeeper primitive is *watch*. A user can set a watch on a znode. When a change on the state of that znode occur, the watch is triggered and the user is informed on the new state of the znode. We now summarize Zookeeper operations of our interest:

- *create*: create a znode by specifying a path. A *sequenced* flag can be used to tell Zookeeper to append the znode name with a monotonically increasing identifier. Pathname is returned if create is successful. Otherwise, an exception is returned.
- *delete*: deletes a znode.
- *exists*: checks if a znode exists on a supplied location.
- *getChildren*: return a list of children of a znode, hence files in a folder.

Each one of those operations have two implementations, a synchronous and an asynchronous ones. Synchronous operations block until a reply is received from the server. On the other hand, asynchronous operations returns immediately and a callback is set on the client side to receive from the server when operation is complete.



```
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. 2: Pseudo code of acquiring and releasing queue locks

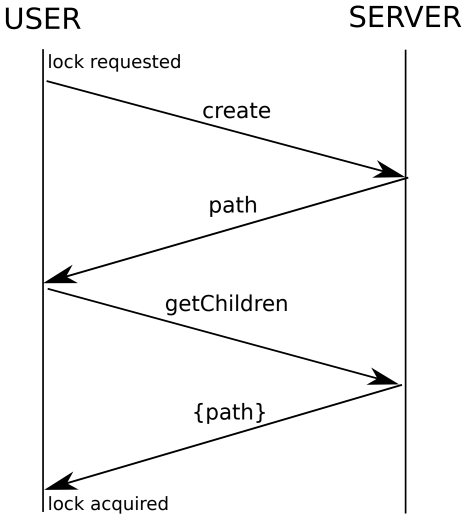


Fig. 3: A time diagram showing time required to acquire a queue lock if no other user is holding it

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

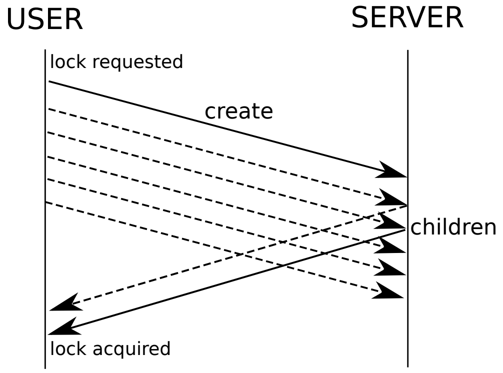


Fig. 4: A time diagram showing time required to acquire a queue lock if no other user is holding it using asynchronous calls. Dashed arrows from user to server are getChildren operations. Dashed arrows from server to user represent getChildren replies before the lock is acquired. Solid arrow from server to user is getChildren reply with user having the smallest sequenced file. The callback for asynchronous create is not shown.

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 2. 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 3. However, as shown in Figure 4 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.

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.

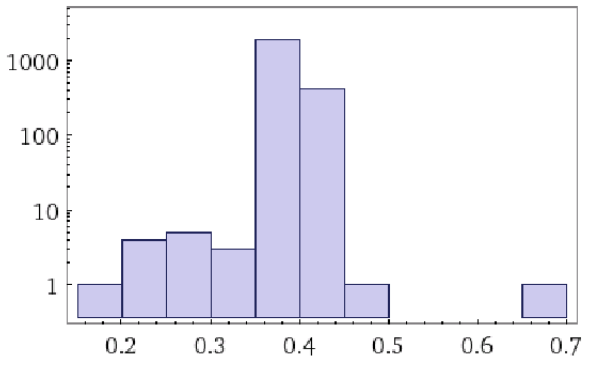


Fig. 5: probability density function of two machines i nthe same cluster. X-axis is RTT and y-axis is number of ocurrences

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.

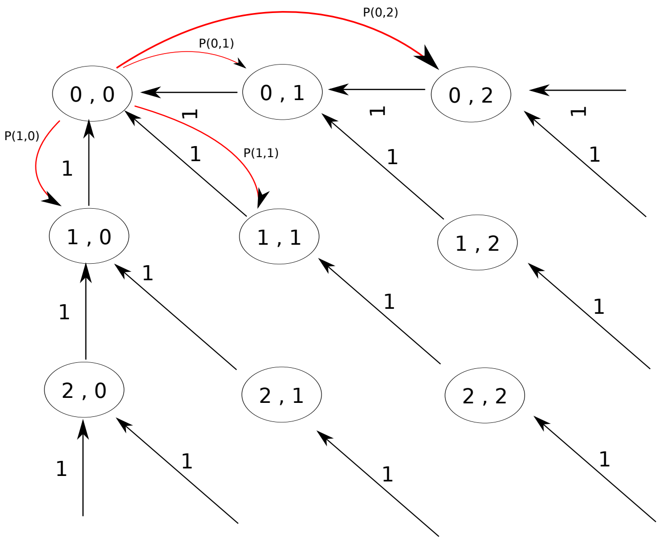


Fig. 6: Markov chain to model latency for one master and two slave servers

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

$$latency_i = max\{RTT_i^1, RTT_i^2, \ldots, RTT_i^n\} \tag{1}$$

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 6 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

This section provides results of our experiments and a discussion of relevant findings. We begin by establishing baseline performance for our Zookeeper testbed which support reasoning for follow-up experiments with synchronization primitves. The synchronization mechanisms benchmarked are Queues and Locks, each implemented with both, synchronous and asynchronous messaging. For this quantitative analysis we focus on average latency and graceful degradation.

A. Experimental setup

The testbed consists of 5 machine containing a quad-core Xeon 2135, 16GB RAM and commodity 500GB 7.2k rpm disks running stock Ubuntu 12.04. They are interconnected with 1Gbps Ethernet on a single switch. A separate machine acts as client without being a Zookeeper server. For our experiments these nodes for quorums out of 1, 3 or 5 nodes as indicated. The server-side uses Zookeeper version 3.3.5 whereas the client-side implementation of benchmarks uses Python bindings for the baseline ”smoke test” and Java bindings for Locks and Queues. In order the measure the impact of client-server and server-server latency we control the transmission latency of packets transmitted between nodes using the Linux ”Traffic Control” utility. We automate the process of enumerating client-server and server-server latency pairs and repeatedly run the experiments and average the results to reduce the impact of jitter.

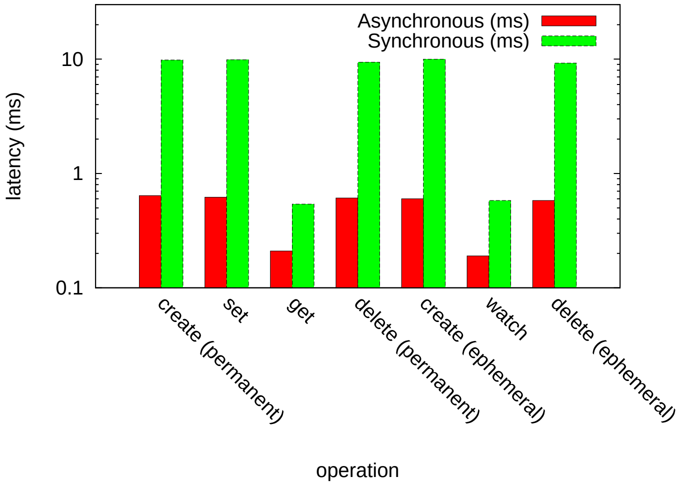


Fig. 7: Zookeeper basic operations latencies for a cluster of five servers

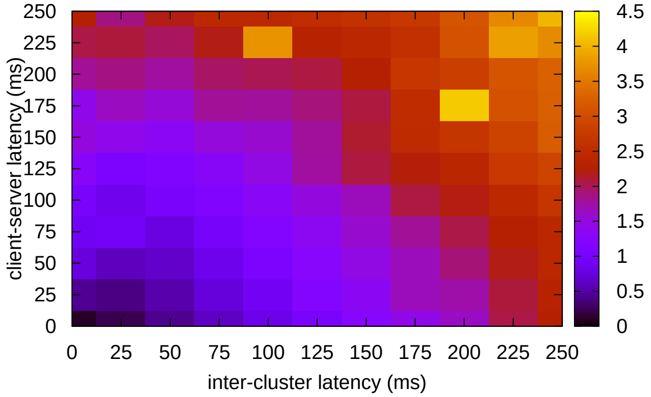


Fig. 8: Zookeeper asynchronous average create operation latency with different inter-cluster and user-server RTTs

B. Baseline performance

Our first experiments 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 7. 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 (operations that update znodes), and around double the latency for *get* operations (operations that does not update znodes). Another interesting observation is that for both synchronous and asynchronous operations set operations (and get operations) have a close latency. This will enable us to pick only a representative of those operations to collect more thorough results. In the next experiment we use create as a representative for set operations. We chose create as it is involved in all our synchronization primitives.

We would like to test the effect of latency on Zookeeper operations. Therefore, we introduce latency in two ways. First, we vary the RTT between the client machine and the cluster and call it *user-to-server* RTT. Second, we vary RTT between Zookeeper servers and we will call it *inter-cluster* RTT. Reported numbers are not the final RTT, but the addition to the original RTT. Since the original RTT is very low compared to our introduced values, reported values act as a good approximation. The first set of results are of asynchronous create operation while varying user-to-server and inter-cluster RTTs and is shown in Figure 8. Asynchronous creates average values do not exceed 4.5ms in our experiment, although RTTs go up to 250. This is because asynchronous operations are allowed to start immediately after invokation of the previous operation. Thus, these results act as an indicator of the level of parallelism that can be supported by Zookeeper. Assuming worst case, we can calculate a lower bound on the average number of concurrent asynchronous creates by dividing RTT by maximum reported latency, hence $\frac{250}{4.5} = 55.56$ operations. An interesting finding is that though asynchronous requests are used, both user-to-server and inter-cluster RTTs have an effect on operation latency.

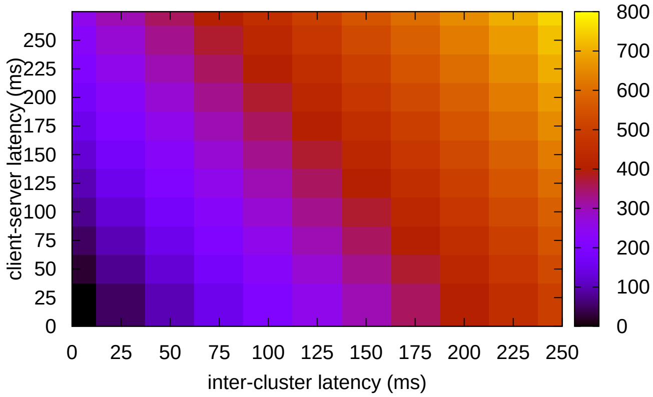


Fig. 9: Zookeeper synchronous average create operation latency with different inter-cluster and user-server RTTs

Now we test synchronous create while varying user-to-server and inter-cluster latencies. Results are shown in Figure 9. Since synchronous creates are used, reported results are the sum of user-to-server RTT and time it takes for Zookeeper servers to process the operation. From the linear relation observed with user-to-server latency, it is apparent that two RTTs are involved in the exchange. Those RTTs corresponds to starting the connection and issuing the create operation. Another observation is that at least one (inter-cluster) RTT is involved in the cluster-side processing of the operation.

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

However, as the inter-cluster RTT increase the latency increase faster than the rate of RTT change.

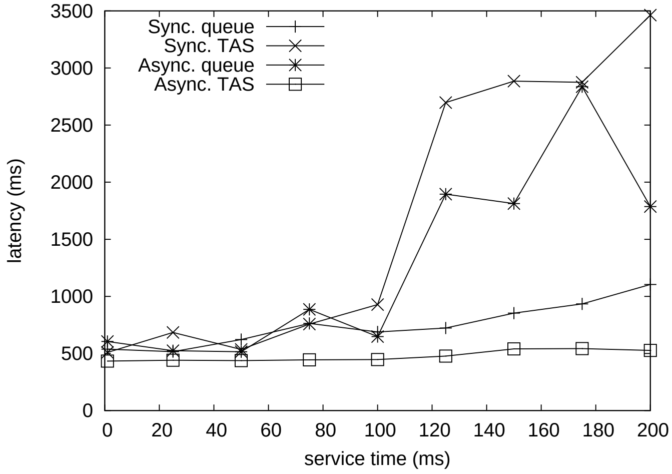


Fig. 11: Synchronization primitives latency while varying contention and fixing RTT to 100ms

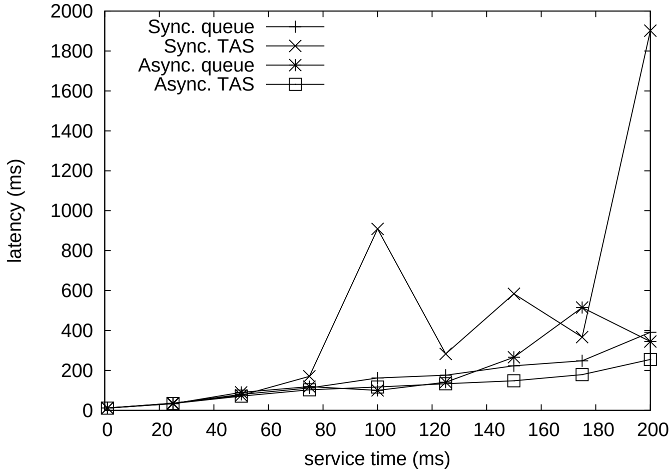


Fig. 12: Synchronization primitives latency while varying contention and no additional delay to RTT

C. *synchronization primitives*

The experiments in this section focus on two fundamental synchonization primites, locks and queues, which are commonly found as builind blocks of larger coordination schemes. The implementation of synchronous primitives follows the Zookeeper recipies with minor improvements, while the asynchronous implementation is described in detail in the following. Locks follow a test-and-set approach by repeatedly issuing create requests and upon successful response from the server assuming mutual exclusion. The lock is then released by deleting the node. The synchonous lock issues a single create request for a fixed node path and blocks until a response arrives. On a positive response mutual exclusion is guaranteed. A negative respose is directly followed up by another create request. This lock has at most a single request or response in transit at a given time. The asynchonous lock issues create requests on a timer basis until a positive response is recorded. This implementation has multiple requests and responses in transit at a time and effectively ”polls” the availability of the lock. This puts higher load on the network and server, but may reduce the time taken to aquire a lock by avoiding an additional notify-request roundtrip. Queues are built using server-side sequentially increasing ids for creating the node. After the node is created the nodes predecessors are obtained and upon their deletion mutual exclusion is guaranteed. As for locks release is performed by deleting the node. The synchonous implementation creates a sub-node, and upon sucess notification requests the list of all siblings from the server using ”getChildren”. If the created node holds the lowest Id mutual exclusion is guaranteed, otherwise a watch on deletion of its predecessor is installed. The asynchonous queue issues the create request in parallel with polling children at a fixed interval. When the create request returns the obtained Id and the most recent list of children contains the Id in first place mutual exclusion is guaranteed. This approach follows the same trade-offs as asynchonous locks, but can be extended to install a watch on the predecessor and stop polling after a round trip interval.

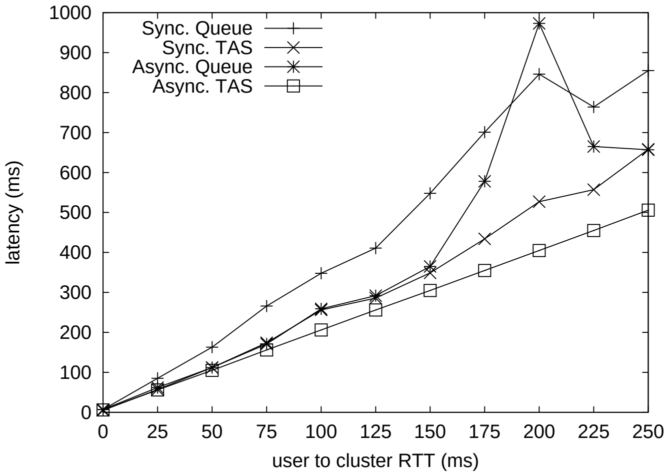
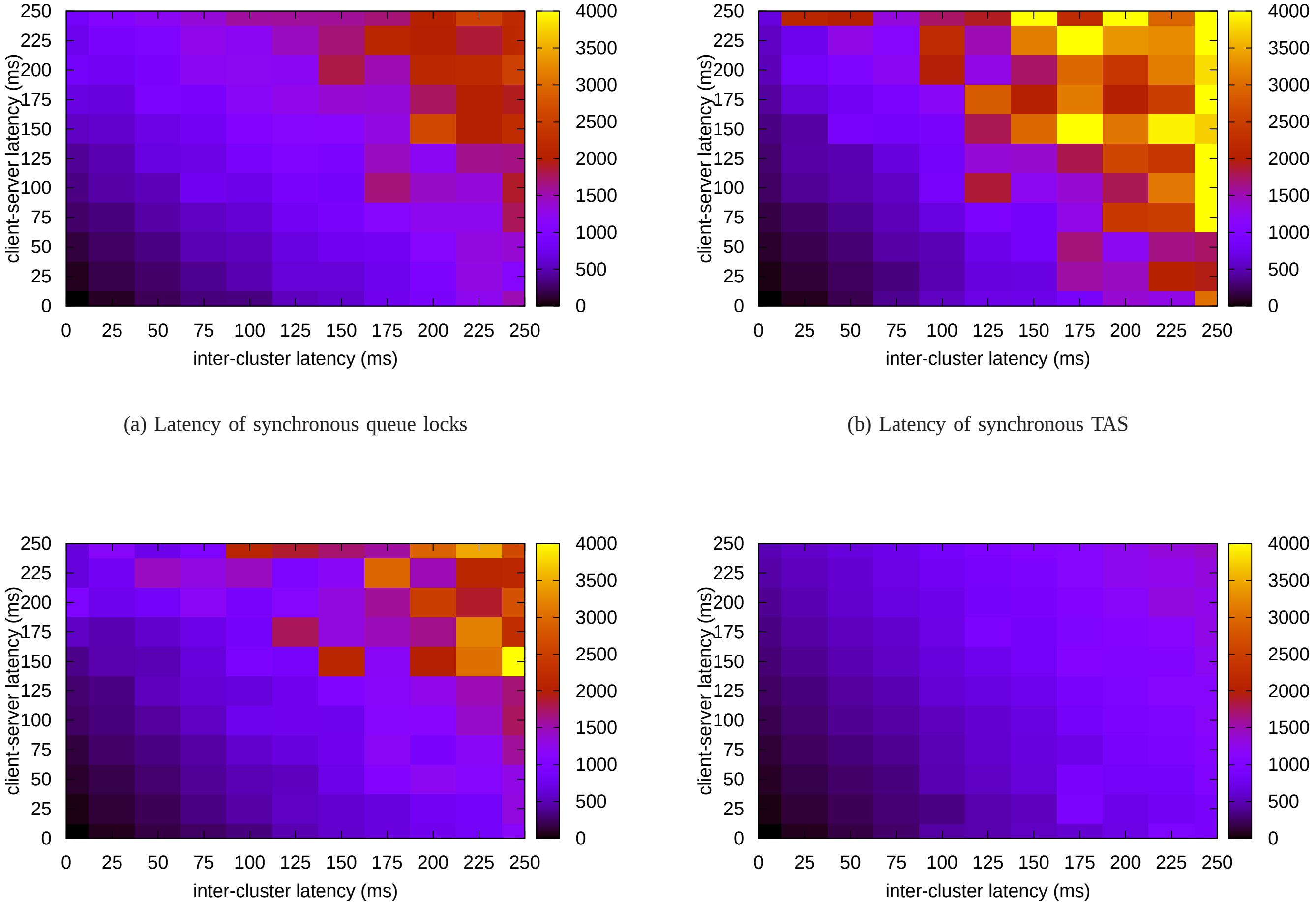


Fig. 13: Synchronization primitives latency while varying user-cluster RTT, with no additional inter-cluster delay, and fixing service time to 1ms and interarrival time to 500ms

In this section we will study our primitives while we vary to important metrics. The first metric is traditionally important for synchronization primitives, namely contention. We vary contention by fixing interarrival time of requests to enter the critical section while varying the service time (*e.g.*, time spent inside the critical section). The second metric is latency. Specifically we are interested in large delays that characterize multi-datacenter communication. The goal of this study is to come out with conclusions on which primitives to use giving different contention and network conditions.

We test all our synchronization primitives while we vary user-to-server and inter-cluster RTTs. These experiments will



(a) Latency of synchronous queue locks

(b) Latency of synchronous TAS

(c) Latency of asynchronous queue locks

(d) Latency of asynchronous TAS

Fig. 10: Heatmaps of different synchronization primitives' latencies while varying inter-cluster RTT and user-to-server RTT.

give us hints on how these primitives act with different RTTs, what is the difference between inter-cluster and user-to-server effects, and what primitives are better when large RTTs are experienced. We show these results in Figure 10. In them, interarrival time between lock requests is exponentially distributed with a mean of 500ms and time spent in the critical section is exponentially distributed with a mean of 1ms. Asynchronous TAS are the most efficient and stable amongst studied primitives. On the other hand, synchronous TAS perform poorly when RTT is high. Queue implementations are better than synchronous TAS for large delays, though are comparable and slightly worse with low RTTs. We will examine relashionship between primitives in specific cases later in this section.

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.