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.

Synchonization mechanisms come at a cost however and have been an emphasis of research for a long time. ref PA papers Numerous papers look into reducing latency locally on a single machine through good choice of primitives, reactive algorithms, heuristics such as lock elison or hardware support for transactional memory. A different branch of research looks into distributed synchonization through protocols like two phase commit or Paxos in a search for fault-tolerance and graceful behavior under high contention. While the first approach provides low latency solutions, it does not scale across a local network or geo distributed datacenters. The latter approach provides fault-tolerance and predictable performance but comes with substantial latency overhead in the first place. The current emphasis on strongly consistent geo distributed application aggravates this latency issue. For example, Google's recent F-1 database system shows substantially higher latencies than its predecessor and pushes complexity down to the client API to mask some of this additional cost.

Here is a need to reinvestigate synchronization protocols for large-scale distributed systems. In these systems communication latency depends on network round trip times and 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 traditional synchronization protocols. Existing distributed coordination packages deliver basic coordination schemes to end users and typically provide proven ”recipies” for building synchronization mechanisms. For example, 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 which enables users to quickly create more complex synchronization primitives such as locks and queues. Although this ease of of use is an advantage upfront, the inefficient implementation or inappropriate use of these low level primitives turns into low overall application throughput. This in turn leads to the installation of complex hierarchical layers of servers and quorums in order to cope with latency and throughput bottlenecks. ref?

In this paper we take first steps towards improved synchronization in geo distributed settings and show that the performance of synchronization mechanisms heavily depends on the use-case. For our experiments we focus on providing mutual exclusion for a resource with varying latency between client-server and multiple servers as well as different levels of contention for that resource. We investigate two primitives, queues and locks with two different implementations each. The implementations, synchonous and asynchronous, are inspired by traditional distributed systems ”recipies” as well as approaches taken in parallel hardware architectures. While our results show familiar results for advantages of locks under low contention and queues under high load, a varying range of round trip times shows non-obvious benefits of asynchronous implementations.

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

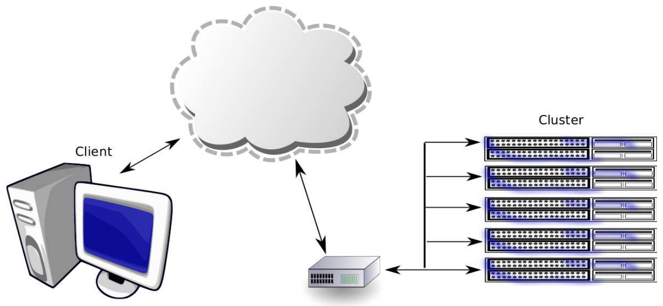


Fig. 1: Testbed setting used for evaluation

A. testbed

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. The Round-Trip Time (RTT) between machines in the cluster is around 0.1ms and around 0.2ms for client to cluster communication. 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 the respective Python bindings for the baseline ”smoke test” and Java bindings for locks and queues implementations. In order to control the impact of client-server and server-server latency we manually set 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.

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 services, such as Chubby and, to a large degree 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. Path-name 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

D. synchronization primitives

One of the main points studied in this paper is the effectiveness of synchronization mechanisms (primitives) and

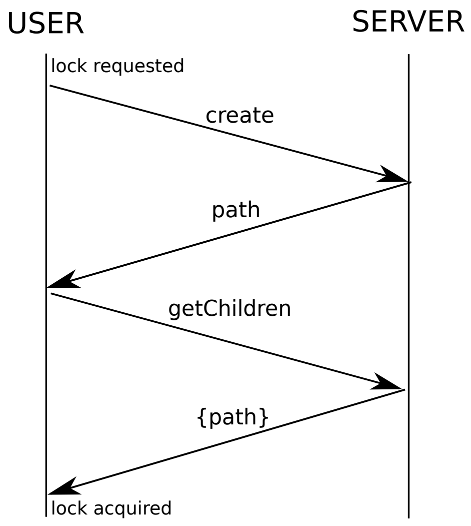


Fig. 3: A time diagram showing time required to acquire a queue lock if no other user is holding it

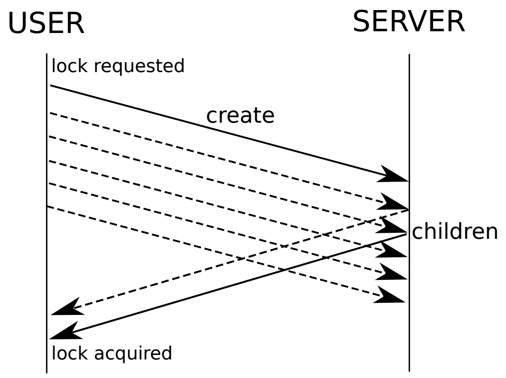


Fig. 4: A time diagram showing time required to acquire a queue lock if no other user is holding it using asynchronous calls. Dashed arrows from user to server are getChildren operations. Dashed arrows from server to user represent getChildren replies before the lock is acquired. Solid arrow from server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

their behavior under different network conditions and contention levels. Here, we give a summary of the primitives and implemetations we consider. Primitives are queue locks and test-and-set (TAS) and implementations are divided into synchronous and asynchronous depending on the pattern of client-server interactions. We translate the primitives into an actual implementation using Zookeeper.

E. Synchronous test-and-set lock

To acquire a lock, the client repeatedly executes atomic test-and-set (TAS) until it succeeds. TAS typically tests a boolean flag until it flips it from false to true and returns sucessfully. Initially, TAS surfaced in bus-based shared memory 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. The client tries to create a node with a known name (all competing clients try to create the same node). If the file already exists the create operation fails, which means that another client is holding the lock. On a successful response, the client obtains the lock. To release the lock, the node is deleted. 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, whichever request arrives first after the lock is cleared is served. There is no order as in entering a queue which impacts fairness. The execution path of TAS when no client is holding the lock is only one synchronous *create* call. Otherwise, it is the time until the client happens to be the fastest to re-create the file after deletion.

F. Synchronous queue lock

This lock uses a queue structure maintaing the process of acquiring a lock. If a client tried to acquire a lock while another was holding it, then the request is queued. Thus, clients acquire the lock based on the order in which they entered the queue, helping achieve fairness. This is implemented in Zookeeper following the ”queue recipe” by synchronously creating an ephemeral, sequenced znode. The create operation returns the file name, indicating the sequence number. The client the gets all sibling nodes and determines its predecessor. The client then installs a watch for deletion of this node, i.e. the client waits until it posesses the smallest sequence number. When the watch response returns the lock is considered acquired. Later, the lock is released by merely deleting the previously created node. Pseudo code is displayed in Figure 2. There are two possible execution paths. First, a client 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 client 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.

G. Synchronous test-and-set lock

The asynchonous lock proactively issues asynchronous 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 without dependence on the rount-trip time. In contrast to the synchronous implementation a negative response does not need to transmitted back before a new request can be issued, which allows the next request to arrive at the server within a predetermined period of time. As a trade-off this puts higher load on the network and the quorum, but reduces the time taken to aquire a lock by avoiding an additional notify-request roundtrip.

H. Asynchronous queue lock

Above, we showed the serial implementation of our queue primitive using synchronous versions of Zookeeper operations. Thus, we block until we receive a reply from the server. We find, however, that we can parallelize create requests and obtaining the Id of the predecessor to reduce latency. We issue the create request ansynchonously and immediately start polling for its siblings by issuing asynchonous getChildren requests on a timer basis. As soon as the create response returns the Id, the most recent list of siblings containing this Id (”consistent list”) can be analyzed to either find the lock acquired or determine the predecessor. 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 multiple read requests on the server, but in the optimal case cuts the time taken to aquire a lock by avoiding the delay for receiving the response to the create request and only then issuing the getChildren request. Further analysis shows that the overhead of polling for longer wait times can be cut off after receiving the consistent list of sibling by installing a watch on the predecessor and as for the synchronous queue.

For illustration the 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*

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 5. 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

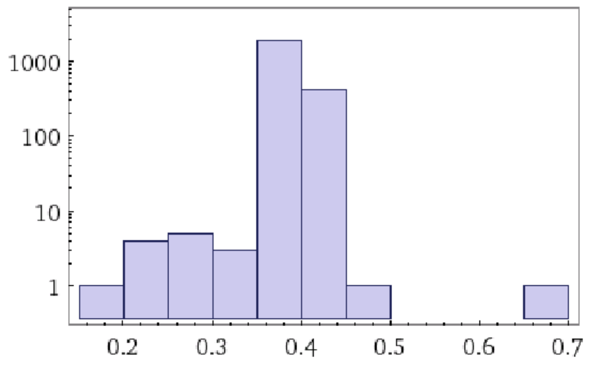


Fig. 5: probability density function of two machines i nthe same cluster. X-axis is RTT in milliseconds and y-axis is number of ocurrences

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.

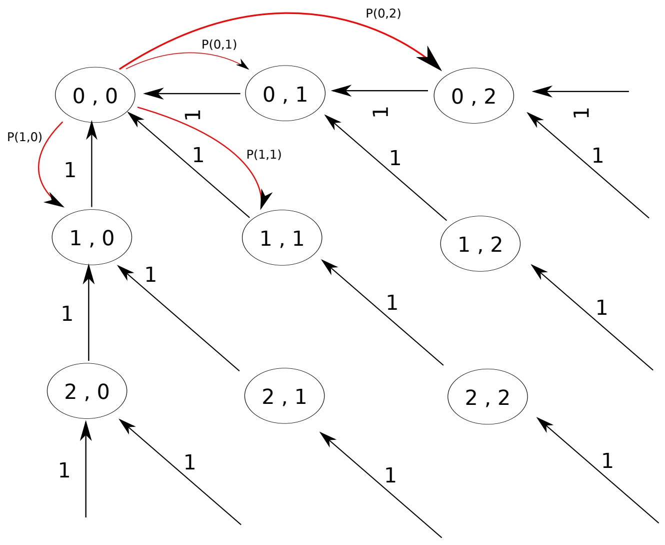


Fig. 6: Markov chain to model latency for one master and two slave servers

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

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

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 $\bar{RTT}$. 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.

This model can be used to reach the real latency of communication on such broadcast systems. To get a sense of the effect of Equation 1 lets take the cumulative distribution function (cdf) of latency process:

$$latency_{cdf}(x) = \prod_{i \in N} RTT_i(X \leq x) \tag{2}$$

where $latency_{cdf}(x)$ is latency's cdf for a value x, N is the set of servers to communicate with, and $RTT_i(X \leq x)$ is the cdf for the process RTT_i. Since the value of any cdf is less than or equal to one, multiplying two cdf functions will result in a value smaller than both of them. Therefore, the cumulative distribution function of the latency of waiting for replies of a broadcast message is at least as bad as the slowest RTT process.

B. Consensus latency

Zookeeper uses paxos algorithm for consensus management. In paxos, when a proposal is broadcasted, the proposal does not wait for all followers to reply. Rather, a majority is enough to proceed with committing the proposal. This is important to notice when applying our findings from the previous subsection. Thus, Equation 1 for describing the latency process becomes:

$$latency_i^p = max\{RTT_{fastestmajority}\} \tag{3}$$

where $latency_i^p$ is latency of each broadcast of paxos, and $RTT_{fastestmajority}$ is the set of $\lceil \frac{n+1}{2} \rceil$ fastest nodes RTTs for iteration i.

C. Synchronization primitives latency

We will consider the latency of queue locks and TAS. For queue locks, a simple M/M/1 queue model can be used to characterize the system. Having λ as the interarrival rate of requests to enter the critical section and μ as the service rate, hence inverse of the time spent inside the critical section. We assume that both interarrival and service times are exponentially distributed. In that case the average number of concurrent lock requests, $\bar{N}$ is:

$$\bar{N} = \frac{\rho}{1-\rho} \tag{4}$$

where ρ is the ratio of interarrival to service time. Using Little's theorem we can find an expression of average time spent in the system (time from acquiring until releasing the lock)

$$W = \frac{\bar{N}}{\lambda} = \frac{1}{\mu - \lambda} \tag{5}$$

where W is the total wait time in the system.

TAS can be modeled as a M/M/1/∞/∞/SIRO queue. This is identical to M/M/1 queue with one difference; queueing discipline is SIRO (Service In Random Order) rather FIFO (First In First Out). The calculations that lead to Equation 5 is not affected by the queueing discipline. Thus, $W_{FIFO} = W_{SIRO}$.

D. Overall system

We laid the building blocks of modeling the whole synchronization system above. Now, we will put them together to see the big picture and model the latency of the system. Each client go through the following stages for each lock request:

- *Connection establishment*: this is described by the RTT process between the user and cluster, *i.e.*, $RTT_{user-to-server}$.
- *Acquiring the lock*: this includes the time for executing necessary operations to try acquiring the lock (before actually acquiring it) and to establish the acquirement of the lock (before entering the critical section). We will denote those two as $T_{pre-acquire}$, and $T_{post_acquire}$. Note that both are functions of $RTT_{user-to-server}$ and $latency^p$.
- *Waiting for the lock*: The time spent waiting for the lock to be available, including the current user holding the lock and other users that will hold the lock while you are waiting. This period is usually between trying acquiring the lock and establishing the acquirement of the lock. The wait time is described in Equation 5 as W.
- *Time in critical section*: This is the time spent holding the lock. The average of this time is $\frac{1}{\mu}$, where μ is the rate of processing requests in mutual exclusion block.
- *Releasing the lock*: The time required to release the lock. Our implementations release the lock by asynchronously deleting a file representing holding the lock, denoted as $T(delete_{async})$.
- *Closing the connection*: time required to close the connection, denoted as $T(close)$.

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. 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

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

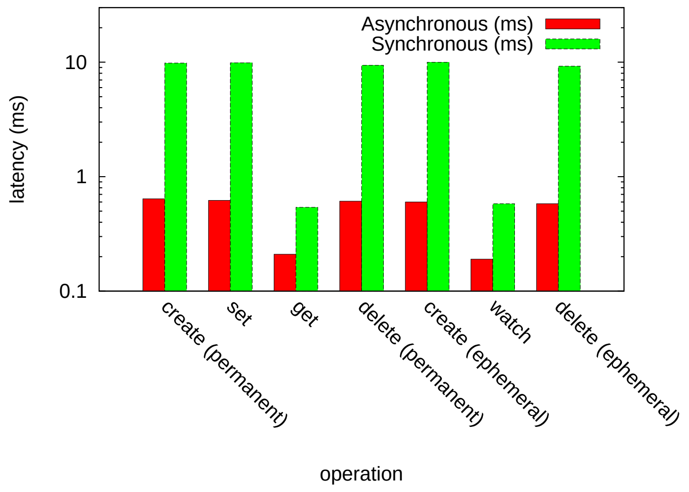


Fig. 7: Zookeeper basic operations latencies for a cluster of five servers

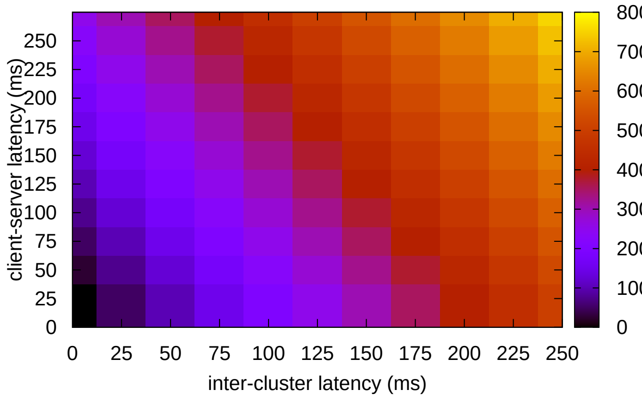


Fig. 9: Zookeeper synchronous average create operation latency with different inter-cluster and user-server RTTs

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.

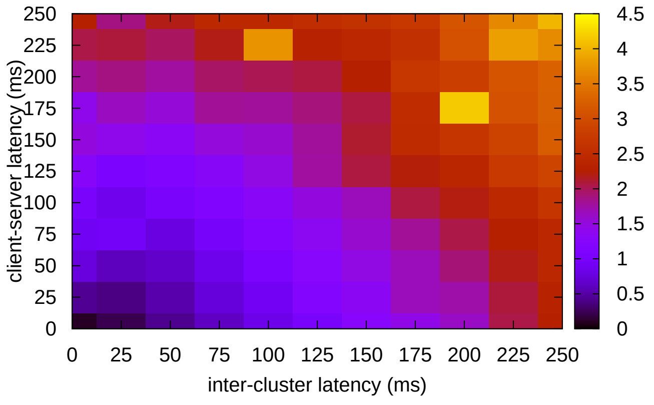


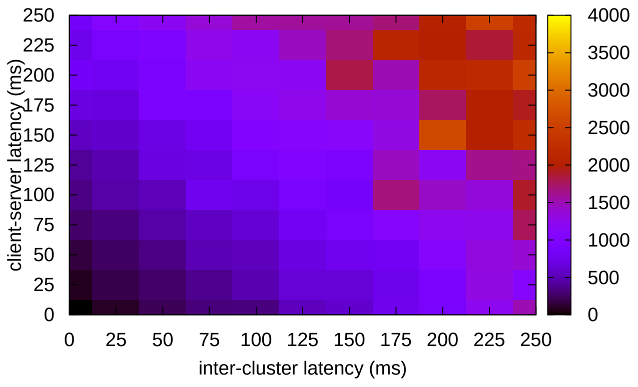
Fig. 8: Zookeeper asynchronous average create operation latency with different inter-cluster and user-server RTTs

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.

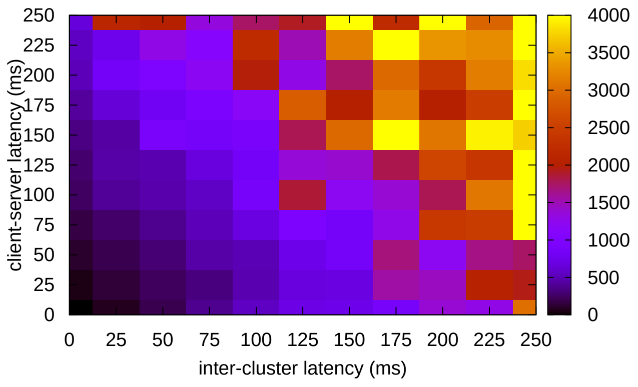
Now we test synchronous create while varying user-to-server and inter-cluster latencies. Results are shown in Figure 9. Since synchronous creates are used, reported results are the sum of user-to-server RTT and time it takes for Zookeeper servers to process the operation. From the linear relation observed 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. However, as the inter-cluster RTT increase the latency increase faster than the rate of RTT change.

B. *synchronization primitives*

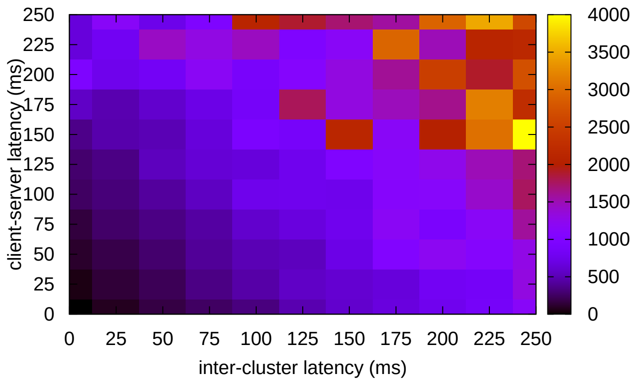
In this section we will study our primitives while we vary to important metrics. The first metric is traditionally important for synchronization primitives, namely contention. We vary contention by fixing interarrival time of requests to enter the critical section while varying the service time (*e.g.*, time



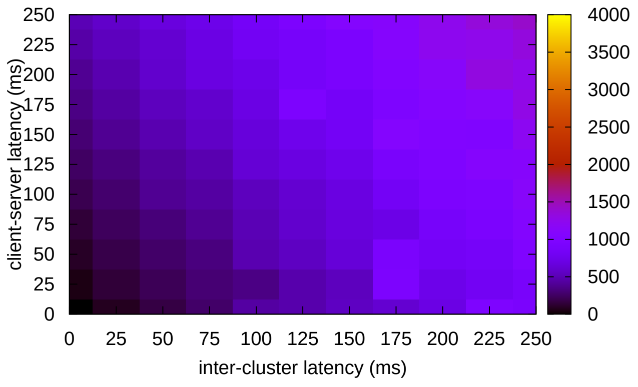
(a) Latency of synchronous queue locks



(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.

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 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.

In the following we study the two-dimensional spectrum of RTT and contention effect on synchronization primitives' latency. We only vary user-to-server RTT. Inter-cluster latency affect the throughput of all synchronization primitives with the same magnitude. This is because synchronization primitives on top of Zookeeper control the exchange of messages between the user and the server and does not affect the exchange of inter-cluster controls. This is apparent in Figure 11, where we vary inter-cluster latency while fixing user-to-server latency and contention. In the figure it is observed that increasing inter-cluster latency have a fixed additive effect on all primitives. We performed four experiments to examine the possible spectrum of values while varying RTT and contention. The span of each experiment is represented as an oval in Figure 12. The following are description of our results:

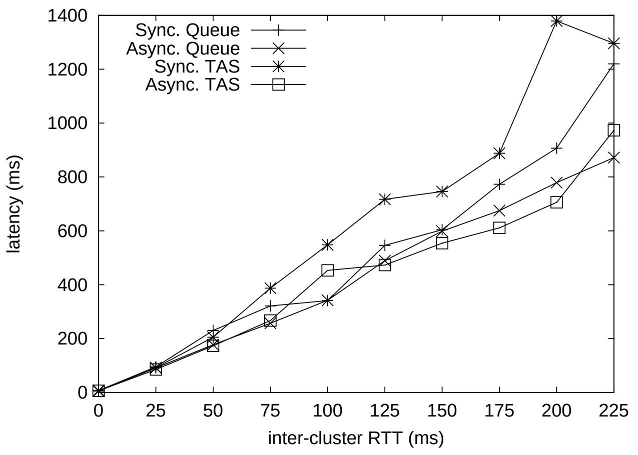


Fig. 11: Synchronization primitives latency while varying inter-cluster latency and not changing user-to-server latency

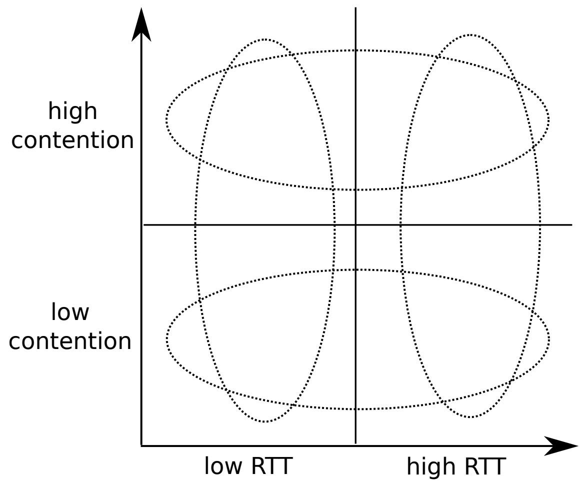


Fig. 12: The spectrum of RTT and contention used for our experiments. Ovals represent a span of a performed experiment

- *High RTT and varied contention*: Figure 13 shows the result of the effect of contention with high RTT. We fix user-to-server RTT to 100ms. Interarrival times between lock requests are fixed to 200ms. Until a critical point (100ms service time in this case) all primitives perform similarly. When service time is above 100ms Asyncronous queue and synchronous TAS performs poorly. Thus, for high RTT and high contention synchronous queue is better than synchronous TAS.
- *Low RTT and varied contention*: Figure 14 display results for the effect of contention with low RTT. We keep the original user-to-cluster latency as it is (*i.e.*, 0.2ms). Interarrival times between lock requests are fixed to 200ms. Here we notice that the effect is not as visible as the high RTT case. However, all primitives perform closely except for asynchronous queues.
- *Low contention and varied RTT*: this experiments show the effect of varying RTT while having low contention (*i.e.*, service time is 1ms and interarrival of requests is 500ms). Figure 15 shows obtained results. From these results it is shown that for low contention, TAS protocols perform better than queue ones. As RTT increases, the different in latency becomes more apparent. Asynchronous TAS is better than synchronous TAS which perform closely to asynchronous queues for low RTT. Thus, we conclude that for low contention, TAS protocols perform better, specially for large RTTs.
- *High contention and varied RTT*: this experiments show the effect of varying RTT while having high contention (*i.e.*, service time is 150ms and interarrival of requests is 200ms). Figure 16 shows obtained results. Confirming our results when we tested varying contention, asynchronous TAS and synchronous queues perform best, specially for high contention.

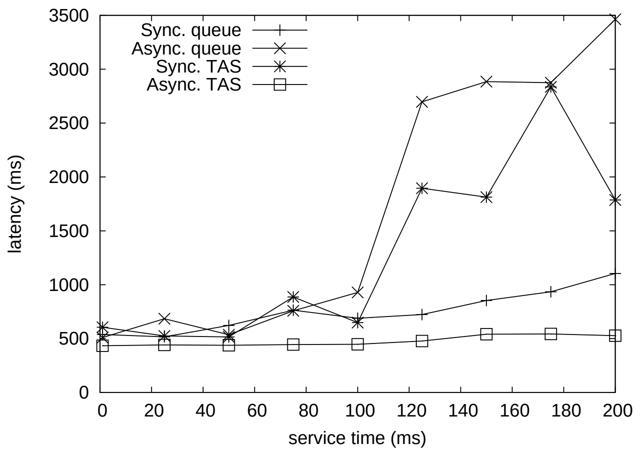


Fig. 13: Synchronization primitives latency while varying contention and fixing RTT to 100ms (varying contention with high RTT)

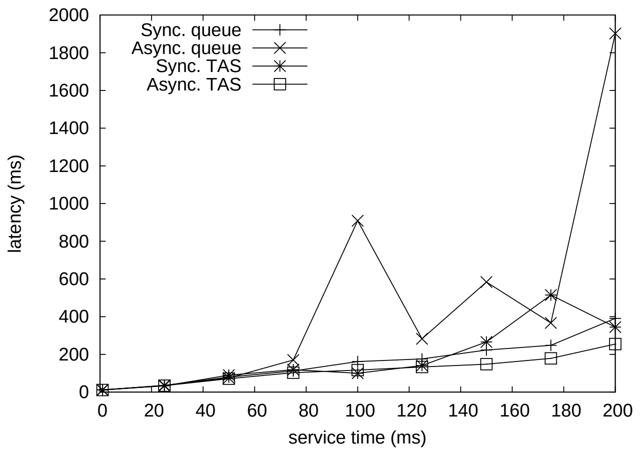


Fig. 14: Synchronization primitives latency while varying contention and no additional delay to RTT (varying contention with low RTT)

Giving the results of our previous experiments, we are now able to infer the tradeoffs of using synchronization primitives with varying RTT and contention conditions. The following summarizes our findings relative to each part of the RTT and

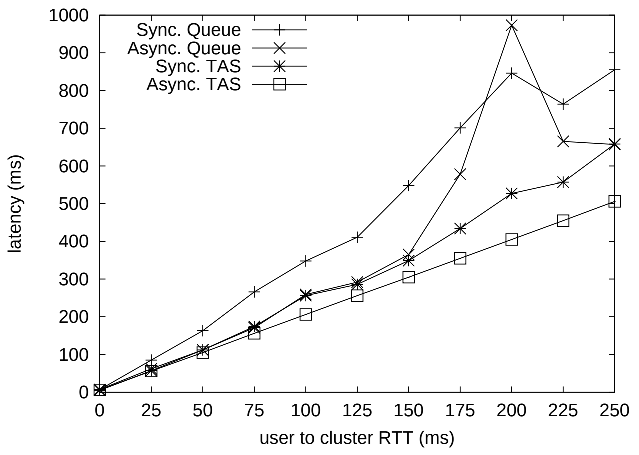


Fig. 15: Synchronization primitives latency while varying user-cluster RTT, with no additional inter-cluster delay, and fixing service time to 1ms and interarrival time to 500ms (low contention with varying latency)

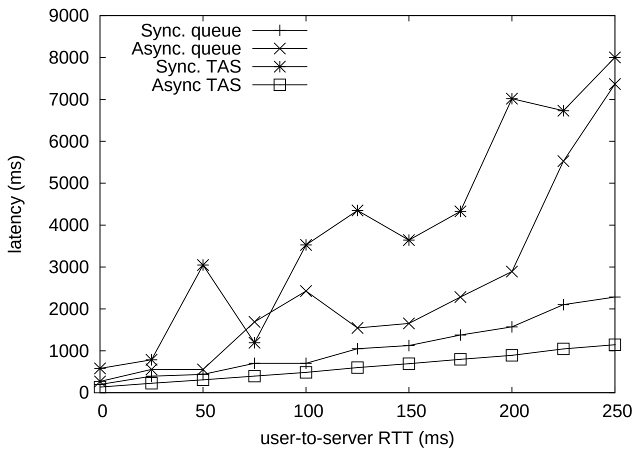


Fig. 16: Synchronization primitives latency while varying user-cluster RTT, with no additional inter-cluster delay, and fixing service time to 150ms and interarrival time to 200ms (high contention with varying latency)

contention spectrum:

- *Low RTT/low contention*: Different primitives do not have much difference in latency. Thus other factors can come in to decide which one to be used. Synchronous queues are better here for their fairness and resources (*i.e.*, computation and network) conservation characteristics.
- *Low RTT/high contention*: Asynchronous TAS performs better. However, if a synchronous primitive is preferred, then queue locks are better than TAS.
- *high RTT/low contention*: TAS protocols perform better than queue locks.
- *high RTT/high contention*: Asynchronous TAS performs better. However, if a synchronous primitive is preferred, then queue locks are better than TAS.

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.