Scalable agreement: Toward ordering as a service

Manos Kapritsos
UT Austin
Austin, TX - USA
manos@cs.utexas.edu

Flavio P. Junqueira
Yahoo! Research
Barcelona, Spain
fpj@yahoo-inc.com

Abstract

Replicated state machines use agreement protocols such as Paxos to order client requests. These protocols are not scalable and can quickly become a performance bottleneck as the degree of fault-tolerance and the demand for throughput performance increase.

We propose a scalable agreement protocol that can utilize additional resources to provide higher throughput, while guaranteeing linearizability for client requests. Our protocol can build on existing optimizations, as it can use protocols like Paxos as a building block. A preliminary performance evaluation shows a throughput increase of $50\%-179\%$ over a baseline strategy, even without adding any hardware; and with additional hardware we are able to achieve even higher performance gains.

1 Introduction

Replicated systems are often used with critical large-scale applications to guarantee availability despite faults, and state machine replication (SMR) is a well-known technique to implement such highly available services. A replicated state-machine comprises a number of replicas that run an agreement protocol to ensure that all replicas execute the same sequence of operations, thus guaranteeing a consistent state across all replicas. State-machine replication has been widely explored in the literature [5, 12, 13, 14, 16, 20] and is used in real systems [4, 11].

As online services increase in importance, scalability becomes a critical feature. As the number of clients of such services increases over time, higher throughput becomes critical, but it is even more important to have ways to increase throughput performance over time without reengineering the system. In fact, given the number of applications relying upon replicated components in Web-scale infrastructures, such as with Google, Amazon, Microsoft, and Yahoo!, it is a viable solution for such infrastructures to run *ordering as a service* if a practical solution is available.

Existing SMR solutions increase the throughput of such systems by fine-tuning implementations of agreement protocols. However, the benefits offered by this approach are rather limited. A truly scalable solution is ideally able to increase the observed throughput by adding more hardware resources to the system.

Traditionally, scalability has been achieved through partitioning of the state space [8]. With partitions, however, the order guarantees are only satisfied for each individual partition, and they are not suitable to the cases we envision. Previous solutions not involving partitioning cannot provide a way to scale the throughput of the system, even if additional hardware is available. In fact, if more machines are added to an ensemble of replicas to increase its degree of fault tolerance, the message complexity of the agreement protocol increases and the throughput decreases. As the number of tolerated faults increases beyond $f = 1$, the agreement protocol rapidly becomes the bottleneck of the system. We therefore believe that scaling the agreement protocol is critical for overall scalability of replicated systems.

Previous work has focused mostly on improving the performance of a single, non-scalable, fault-tolerant cluster (e.g., $2f+1$ replicas running the Paxos protocol). There is the notion of fault scalability [1] for replication protocols, which indicates how much performance degrades as the number of tolerated faults increases. The FSR protocol of Guerraoui *et al.* [10] is an example of a protocol that does not degrade throughput with the number of tolerated faults due to the use of a ring topology. Other protocols [2, 3] have improved the performance of fault-tolerance protocols in wide-area settings through the use of a hierarchical structure. However, none of these protocols can scale their performance when additional machines are available. Ideally, a scalable solution is based on solid principles and it does not need to rely upon fine-tuning of code, even though such improvements can be used regardless.

We present a scalable agreement protocol that increases throughput with the number of available machines. Given the interest in leader-based protocols both in the literature (e.g., Paxos and Zyzzyva) and in practice (e.g., Chubby and ZooKeeper), our proposed approach targets applications that use such protocols. Here we also concentrate on a solution for crash-tolerant systems given our interest in services such as ZooKeeper that tolerate crash faults. Our strategy, however, is generic and not dependent upon a particular choice of a failure model. It is consequently possible to adapt the approach we present here to protocols for weaker failure models such as BFT protocols. We discuss our initial prototype and discuss some preliminary results on an Emulab testbed to show its ability to scale. It is out of the scope of this short paper, however, to discuss how to incorporate our proposed protocol into real applications. This is subject of future work.

Note that the scalability of the agreement protocol does not imply that the execution of requests also scales. This paper proposes an approach to scale the performance of the agreement protocol. As such, it targets scenarios in which the agreement is potentially the bottleneck, which can happen for a few reasons: (a) the execution is not CPU-intensive; (b) the assumption of a weak failure model (e.g., Byzantine), leading to complex message patterns; or (c) the large number of tolerated failures, as discussed above. Finally, there are operational advantages in using the same agreement cluster to support multiple execution clusters, as discussed in Section 5.

In the context of Replicated State Machine systems we use agreement protocols to agree on a specific ordering of requests. In that sense we will use the terms "agreement" and "ordering" interchangeably in the remainder of this paper.

Rationale There are three main parts that influence the performance of an agreement protocol: CPU, network, and disk. In such systems, CPU is used to process messages and in some cases verify the integrity of messages using CRC checks, MAC authenticators or digital signatures, depending on the failure model assumed. In leader-based protocols, such as Paxos, the volume of messages the leader has to process is higher compared to followers, which may cause the CPU of the leader to be the main system bottleneck. In fact, the volume of messages increases with the value of tolerated replica crashes f, thus aggravating the problem as we increase the degree of replication of a protocol. In such cases, splitting the functionality of the leader across multiple or even all machines leads to higher performance.

The network usage reflects the complexity of the protocol: a higher volume of messages implies a higher network utilization. Typically, leader replicas have to manipulate a larger number of messages compared to followers. Spreading the load of the leader more evenly across a number of servers enables a higher utilization of switch bandwidth, which is important for data-center applications. Other protocols in the literature have made a similar point [18]. Mencius [16] uses the network resources available in the system more uniformly and provides higher throughput compared to Paxos. Our proposal is more aligned with the techniques Mencius proposes. Mencius, however, targets WAN settings and it does not thoroughly explore the scalability aspect of its approach. Chain [9] uses a pipeline message pattern instead to deal with high-contention scenarios and leads to a good utilization of the switch bandwidth. This is a different design approach, which gives different performance trade-offs compared to typical leader-based protocols.

Finally, replicas use disks to log messages, guaranteeing that critical protocol information persists across crashes and recoveries. For correctness, replicas have to force writes to disk, which tends to slow down write operations. There are techniques, however, to mitigate this problem, such as using group commits [7] and write-ahead logging [17]. Even with such techniques, we can make a better use of multiple disks with implementations that are disk-bound. Having multiple disks is quite common in modern server configurations. By having sub-groups of replicas as opposed to one single group, we can split the work of a replica across multiple sub-groups and assign each sub-group to a disk, thus leveraging the I/O bandwidth of multiple disks to process the traffic of distinct groups in parallel. The use of multiple disks is not unique to our approach, and our point is simply that the presence of multiple sub-groups facilitates the design and implementation of such techniques.

Model We consider asynchronous distributed systems, where the speed of processes can vary arbitrarily and message delays are unbounded. We do not assume a specific failure model (i.e. crash-fail, Byzantine), as the approach we describe is not bound to a specific one. However, the details of the final solution are different depending on the failure model considered. In this paper, we use a crash-failure model to illustrate our approach.

In the following sections, we outline a solution to the problem of scalable agreement. We first present an overview of the system architecture and then present the protocol that achieves total ordering of requests, while being able to scale its throughput as more resources are added to the system. Finally, we present a preliminary performance evaluation of our prototype.

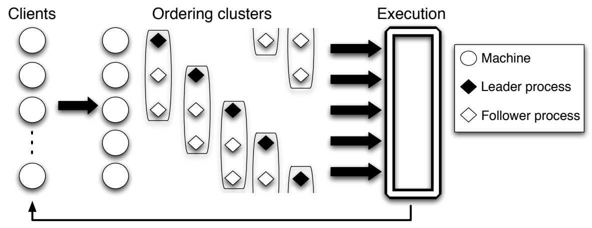

Figure 1: Architecture outline: 5 clusters of 3 processes each are being deployed on 5 machines. For each cluster, we use a rectangle to identify all machines participating in the cluster. Each cluster sends their ordered requests to the execution cluster.

2 Overall architecture

To improve specific aspects of an RSM system, it is important to follow a modular approach in the design of the architecture. To this end, we separate ordering from the execution as with previous work [20]. In our architecture, clients send requests to an ordering module, implemented with multiple replicas, that establishes a total order for the proposed requests. Once ordered, requests are sent to the execution module to be executed, and responses are sent directly back to the clients.

Traditional RSM systems use a single cluster of ordering replicas that cooperate to provide a total order of requests. For crash-tolerant systems, it is sufficient to have $2f + 1$ replicas, when up to f replicas can fail [13]. Our architecture leverages multiple such clusters, each of them providing a total order for only a part of the requests. We use a virtual slot scheme at the execution replicas to combine all these partial orders into a single total order for all requests. We describe this scheme in more detail in the following section.

Figure 1 demonstrates the design of our system. We assume $N \geq 2f+1$ machines, each running one or more processes, and establish N clusters of $2f + 1$ processes each. These clusters overlap to fully utilize the available machines. Moreover, the leader of each of these clusters runs on a different node. This is important as the leader of an ordering cluster is much more loaded than the replicas and spreading the leaders on all available machines distributes the load more evenly.

Each ordering cluster receives a subset of the client requests and runs an agreement protocol to establish a total order on those requests. It then forwards that order to all execution replicas, according to the protocol that we describe in the next section. The execution replicas receive the requests from the ordering clusters and use the virtual slot scheme to establish the total order on those requests. They execute the requests in that order and send the response back to the clients. Note that from now on we will use the term *replica* to refer to a process running on a machine, rather than the machine itself.

3 Protocol briefing

3.1 Overview

The goal of this protocol is to coordinate multiple ordering clusters to provide a total order on all requests. As mentioned before, each of the clusters runs a traditional agreement protocol to establish a total order on a fraction of the requests. However, naively combining the distinct sequences from each cluster yields a partial rather than a total order on all requests. We use the following scheme to provide the total order.

We establish a virtual *slot sequence* that corresponds to the total order in which requests should be executed. The i^{th} cluster can only propose requests in slots of the form $i + kN$, where k is an integer and N is the number of clusters. Note that a cluster can propose a batch of requests for the same slot, since those requests are ordered amongst themselves.

This scheme takes advantage of the static configuration of the clusters to transform the partial ordering provided by the clusters into a total ordering. However, there are several aspects of this new approach that make it different from a traditional RSM system. We outline the most important ones below.

3.2 Backlog of requests

An execution replica must execute requests in the order specified by the virtual slot sequence. That means that the execution replicas cannot execute the requests as they arrive. Upon receipt of an ordered request, an execution replica checks its sequence number n and compare it with the last executed sequence number s. If $n < s$, then the request is not executed and a response is sent back to the client according to a reply cache similar to [6]. If $n > s$, then the request is put in a backlog of requests, to be executed when all previous slots have been executed. Only if $n = s$ is the request executed immediately. After executing a request with sequence number n, the replica executes all consecutive backlogged requests, starting from $n + 1$, until no such request can be found.

3.3 Skipping

Since the execution replicas can only execute requests in the specified total order, it is important that a single slow cluster does not delay the other clusters. To avoid these delays, when a cluster sends a batch of requests to the execution cluster, it also sends a $\langle \text{FLUSH}, n \rangle$ to all other clusters, where n is the sequence number of the slot being filled. Upon receipt of such a message, the leader of each cluster "flushes" any sequence number that is smaller than n by proposing $\texttt{no-ops}$ for those slots. Specifically, it sends a $\langle \text{SKIP}, n[\,] \rangle$ to all execution replicas, where $n[\,]$ is an array of sequence numbers that it skips (i.e. proposes a $\texttt{no-op}$ for). This mechanism is similar in spirit to skipping in Mencius [16].

Note that it is not necessary to send the FLUSH message to all replicas of a cluster. Instead, we can send it only to the leader. Even if the leader is crashed, the sending cluster eventually learns who the new leader is (Section 3.4) and immediately sends a FLUSH message to that replica, thus flushing all previous slots.

3.4 Failure handling

A replica failure has a dual nature in this context. First of all, the replica's cluster will take appropriate actions after detecting this failure. If the replica was the leader, a new leader must be elected. We allow the internal cluster mechanisms to work as designed. However, we add some extra messages to enable the inter-cluster communication and to make sure the load is well balanced.

When the leader of a cluster fails, the new leader will send a message to all clusters notifying them of its leadership, so that FLUSH messages are sent to it. Also, when any replica fails, the cluster can broadcast this information to all clusters and they can choose to rearrange the assignment of replica processes to machines. It is important to keep in mind that in this context a replica is not a machine, but rather a process running on one of the machines. We try to balance the load on all machines by having one cluster leader on each machine and roughly the same number of replica processes. However, failures can change this balance and a reassignment might be in order. For example, if 2 replicas that execute on machine M fail, then some replica from machine K must be moved onto M to balance the load. There are several design choices in this area that we would like to explore. We assume, however, that replica failures are not frequent enough to cause state transfers to become a bottleneck.

3.5 Client connections

Another aspect that needs to addressed is client-related state. As opposed to the majority of the related work, we allow clients to have multiple outstanding connections. This is an important feature required of real applications to improve per-client throughput. To accommodate this feature, there are two design options, each with its own advantages and disadvantages. The first one is to let clients send requests to all clusters. In this case, the requests should carry a client sequence number and the execution replicas should make sure to execute the requests in the order specified by the client. Client ordering can be achieved by using a backlog of requests for each client, similar to the one described in Section 3.2. Also, a session must be maintained from the client to the execution replicas, to make sure that if the client crashes, all its backlogged requests are garbage collected. This approach is convoluted and induces some overhead to the execution replicas.

The second approach is to have each client send requests to a specific cluster. This way, the cluster can make sure that requests are ordered with respect to the FIFO constraints at the client. There is no need for a client backlog or a client session at the execution replicas. This approach, however, might lead to load imbalance, especially if there are very few clients. However, even with as few as N clients (where N is the number of clusters), we can achieve good load balancing, as long as the clients issue roughly the same load to the system.

The second approach is in general more desirable due to its simplicity. Our prototype implements both approaches, but the experiments presented in the evaluation only use the second approach.

Independent of the approach we use, clients do not need to know the internal details of the agreement protocol. They send the requests to the appropriate cluster (either random or fixed, depending on the approach) and the cluster guarantees that the request is totally ordered. To satisfy request dependencies, we can use similar techniques proposed in the literature. Same-client dependencies (FIFO) can be identified using sequence numbers, while dependencies across clients (causal) can be identified using vector clocks. The system must check both dependencies at execution time and delay a request if it arrives out of order, as described above.

4 Evaluation

We have implemented the system described in Sections 2 and 3, and deployed it on an Emulab testbed [1] with 28 quad-core 2.4GHz Xeon machines. We report on the performance benefits we have observed using our system. We have not included logging to stable storage, in order to focus on the benefits of additional CPU resources and clarify the presentation of our results.

We start with the traditional approach, which we use as our baseline; a single fault-tolerant cluster of $2f + 1$ replica processes on $2f + 1$ machines. We then deploy our system by adding more clusters to the existing hardware, without adding any more machines. The resulting configuration has $2f + 1$ clusters of $2f + 1$ replicas, run-

[1] http://boss.cias.utexas.edu

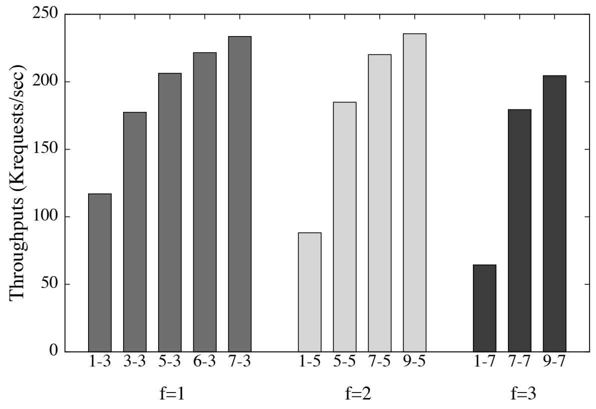

Figure 2: Throughput performance of various configurations. The 1-N bars are the baseline systems with a single cluster on N machines. The M-N counterparts denote a configuration of M clusters of N replicas each, deployed on M machines.

ning on $2f + 1$ machines total. This configuration has a significant advantage over the single cluster, since it spreads the role of the leader on all machines, thereby yielding much higher throughput.

We proceed by adding more machines (and corresponding clusters) to the system. Our generic configuration has N clusters of $2f+1$ replicas, running on N machines. We report on the observed throughput as we vary N and f. Figure 2 summarizes our performance results. We demonstrate the throughput of traditional clusters for $f = 1, 2$ and 3 (denoted as 1-3, 1-5 and 1-7). Next to each of these bars, we demonstrate the performance of our prototype system when multiple clusters are used instead. For example, the 5-3 bar shows the performance of a system with 5 ordering clusters of 3 replicas each, deployed on 5 machines total (the same as our architecture example in Figure 1).

It is clear that traditional systems can benefit substantially from this approach. For all values of f, the system throughput increases significantly, even when no extra machines are added, just by deploying multiple overlapping clusters (configurations 3-3, 5-5 and 7-7). By spreading the role of the leader across multiple machines, we are able to balance the load of the system more evenly and consequently obtain a higher performance compared to the approach that uses a single cluster. Naturally, this gain is more pronounced as the value of f increases. Note that this throughput gain comes at no additional hardware cost. For example, both the 1-7 and the 7-7 configurations use 7 machines each, but the 7-7 has a 180% throughput gain over the 1-7.

By adding more machines to the system, we can stretch our performance gains even further, while keeping the benefit of load balancing. Our approach allows us to split the load more evenly among all the available machines and also to make use of the extra resources, which is not achievable with the state-of-the-art protocols. We note here that, as we add machines to a "N-N" (e.g. 3-3) configuration, the gain is not linear as one would expect, but rather sub-linear. We suspect that it is possible to engineer the system to eliminate bottlenecks and overcome this negative effect, but further investigation is necessary to determine whether our suspicion is correct.

5 Discussion

Ordering service. We believe such an approach will enable the deployment of an *ordering service*, which will be available to multiple applications. Large infrastructures for Web-scale applications often comprise a number of replicated systems that could use an ordering service as part of an application that implements strong consistency guarantees. Managing a single cluster is operationally cheaper compared to having each application implement its own strategy or even having an ordering library and having an instance running for each application.

A key point that makes it feasible is that this service is able to scale its throughput on demand, without having to partition or re-engineer the whole system. Naturally, a key aspect to design and implement is reconfiguration. This feature is necessary to enable dynamic addition of machines, and there are techniques in the literature we can leverage to enable configuration changes [15]. We think that an ordering service is more desirable than application-specific ordering clusters, which are much harder to maintain and cannot scale throughput as the throughput requirements increase beyond their capacity.

Applications. Our technique is based on the active replication approach [19]. Consequently, it is more directly applicable to protocols and systems that implement this approach, such as Paxos, PBFT and Zyzzyva. The principle of separating ordering from execution applies naturally to those systems and can therefore benefit from our approach with only minimal modifications. For systems that have the leader propagating state updates, like Chubby and ZooKeeper, separating ordering from execution is more complex. As a result, the existing implementations of both Chubby and ZooKeeper require modifications to be able to accommodate our approach to scalability.

Bottlenecks. In the settings we have been working with and in our experience so far with our prototype, the leader CPU seems to be the main bottleneck for leader-based protocols, especially when messages have a small number of bytes. Spreading the role of the leader across multiple nodes alleviates this problem and distributes more uniformly the CPU of the system. Our approach also enables a better utilization of disks for logging, since the sub-group strategy is able to split the I/O traffic across multiple disks, thus increasing the performance of I/O operations. Finally, if the network is the main bottleneck, then spreading the role of the leader might be of little help. Although our technique spreads the load across multiple replicas, the overall number of messages is still the same. One advantage of our strategy is the concrete scenario of a data-center application, where all replicas are connected to the same switch. In this case, spreading the network traffic more evenly across replicas enables a more efficient utilization of the switch backplane and leads to higher throughput performance.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 59–74, New York, NY, USA, 2005. ACM.

[2] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Customizable fault tolerance for wide-area replication. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 65–82, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-rotaru, J. Olsen, and D. Zage. Steward: Scaling Byzantine fault-tolerant systems to wide area networks. In *In Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI '99: Proceedings of the third Symposium on Operating Systems Design and Implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[6] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290, New York, NY, USA, 2009. ACM.

[7] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. In *IEEE Database Eng. Bull. 8(2)*, pages 3–10, 1985.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, New York, NY, USA, 1996. ACM.

[9] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems*, pages 363–376, New York, NY, USA, 2010. ACM.

[10] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quema. High throughput total order broadcast for cluster environments. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 549–557, Washington, DC, USA, 2006. IEEE Computer Society.

[11] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10: Proceedings of the USENIX Annual Technical Conference)*, Berkeley, CA, USA, 2010. USENIX Association.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[14] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[15] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

[16] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI '08: Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[17] R. J. Peterson and J. P. Strickland. Log write-ahead protocols and ims/vs logging. In *PODS '83: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 216–243, New York, NY, USA, 1983. ACM.

[18] G. Santos, M. Correia, A. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *SRDS '09: Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems*, Niagara Falls, USA, 2009.

[19] F. Schneider. Replication management using the state-machine approach. In *Distributed Systems*, pages 169–198. Addison Wesley, 1993.

[20] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, 2003.