# A simple totally ordered broadcast protocol

Benjamin Reed
Yahoo! Research
Santa Clara, CA - USA
breed@yahoo-inc.com

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

## ABSTRACT

This is a short overview of a totally ordered broadcast protocol used by ZooKeeper, called Zab. It is conceptually easy to understand, is easy to implement, and gives high performance. In this paper we present the requirements ZooKeeper makes on Zab, we show how the protocol is used, and we give an overview of how the protocol works.

## 1. INTRODUCTION

At Yahoo! we have developed a high-performance highly-available coordination service called *ZooKeeper* [9] that allows large scale applications to perform coordination tasks such as leader election, status propagation, and rendezvous. This service implements a hierarchical space of data nodes, called *znodes*, that clients use to implement their coordination tasks. We have found the service to be flexible with performance that easily meets the production demands of the web-scale, mission critical applications we have at Yahoo!. ZooKeeper foregoes locks and instead implements wait-free shared data objects with strong guarantees on the order of operations over these objects. Client libraries take advantage of these guarantees to implement their coordination tasks. In general, one of the main premises of ZooKeeper is that order of updates is more important to applications than other typical coordination techniques such as blocking.

Embedded into ZooKeeper is a totally ordered broadcast protocol: Zab. Ordered broadcast is crucial when implementing our client guarantees; it is also necessary to maintain replicas of the ZooKeeper state at each ZooKeeper server. These replicas stay consistent using our totally ordered broadcast protocol, such as with replicated state-machines [13]. This paper focuses on the requirements ZooKeeper makes on this broadcast protocol and an overview of its implementation.

A ZooKeeper service usually consists of three to seven machines. Our implementation supports more machines, but three to seven machines provide more than enough performance and resilience. A client connects to any of the machines providing the service and always has a consistent view of the ZooKeeper state. The service tolerates up to $f$ crash failures, and it requires at least $2f + 1$ servers.

Applications use ZooKeeper extensively and have tens to thousands of clients accessing it concurrently, so we require high throughput. We have designed ZooKeeper for workloads with ratios of read to write operations that are higher than 2:1; however, we have found that ZooKeeper's high write throughput allows it to be used for some write dominant workloads as well. ZooKeeper provides high read throughput by servicing the reads from the local replica of the ZooKeeper state at each server. As a consequence, both fault tolerance and read throughput scales by adding servers to the service. Write throughput does not scale by adding servers; instead it is limited by the throughput of the broadcast protocol, thus we need a broadcast protocol with high-throughput.

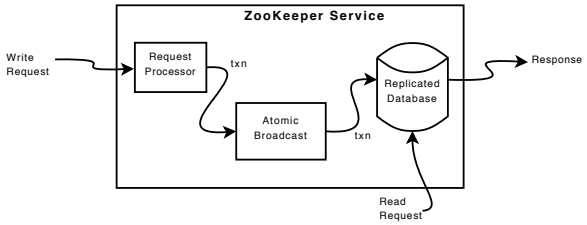


**Figure 1: The logical components of the ZooKeeper service.**

Figure 1 shows the logical makeup of the ZooKeeper service. Read requests are serviced from the local database containing the ZooKeeper state. Write requests are transformed from ZooKeeper requests to idempotent transactions and sent through Zab before a response is generated. Many ZooKeeper write requests are conditional in nature: a znode can only be deleted if it does not have any children; a znode can be created with a name and a sequence number appended to it; a change to data will only be applied if it is at an expected version. Even the non-conditional write requests modify meta data, such as version numbers, in a ways that are not idempotent.

By sending all updates through a single server, referred to as the leader, we transform non-idempotent requests into idempotent transactions. We use the term *transaction* to denote the idempotent version of a request throughout this paper. The leader can perform the transformation because it has a perfect view of the future state of the replicated

database and can calculate the state of the new record. The idempotent transaction is a record of this new state. ZooKeeper takes advantage of the idempotent transactions in many ways that are out of the scope of this paper, but the idempotent nature of our transactions also allows us to relax an ordering requirement of our broadcast protocol during recovery.

## 2. REQUIREMENTS

We assume a set of processes that both implement the atomic broadcast protocol and use it. To guarantee the correct transformation into idempotent requests in ZooKeeper, it is necessary that there is a single leader at a time, so we we enforce that there is one such a process through the protocol implementation. We discuss it more when we present more detail of the protocol.

ZooKeeper makes the following requirements on the broadcast protocol:

**Reliable delivery** If a message, $m$, is delivered by one server, then it will be eventually delivered by all correct servers.

**Total order** If message $a$ is delivered before message $b$ by one server, then every server that delivers $a$ and $b$ delivers $a$ before $b$.

**Causal order** If message $a$ causally precedes message $b$ and both messages are delivered, then $a$ must be ordered before $b$.

For correctness, ZooKeeper additionally requires the following prefix property:

**Prefix property:** If $m$ is the last message delivered for a leader $L$, any message proposed before $m$ by $L$ must also be delivered;

Note that a process might be elected multiple times. Each time, however, counts as a different leader for the purposes of this prefix property.

With these three guarantees we can maintain correct replicas of the ZooKeeper database:

1. The reliability and total order guarantees ensure that all of the replicas have a consistent state;
2. The causal order ensures that the replicas have state correct from the perspective of the application using Zab;
3. The leader proposes updates to the database based on requests received.

It is important to observe that there are two types of causal relationships taken into account by Zab:

1. If two messages, $a$ and $b$, are sent by the same server and $a$ is proposed before $b$, we say that $a$ causally precedes $b$;
2. Zab assumes a single leader server at a time that can commit proposals. If a leader changes, any previously proposed messages causally precede messages proposed by the new leader.

**Causal violation example.**
To show the problems caused by violating the second dimension of our causal relationship consider the following scenario:

- A ZooKeeper client $C_1$ requests to set a znode `/a` to 1 which will be transformed into a message, $w_1$, that contains ("/a", "1", 1), where the tuple represents path, value, and resulting version of the znode.
- $C_1$ then requests to set `/a` to 2 which will be transformed into a message, $w_2$, that contains ("/a", "2", 2);
- $L_1$ proposes and delivers $w_1$, but it is only able to issue $w_2$ to itself before failing;
- A new leader $L_2$ takes over.
- A client $C_2$ requests to set `/a` to 3 conditioned on a being at version 1, which will be transformed into a message, $w_3$, that contains ("/a", "3", 2);
- $L_2$ proposes and delivers $w_3$.

In this scenario, the client receives a successful response to $w_1$, but an error for $w_2$ because of the leader failure. If eventually $L_1$ recovers, regains leadership, and attempts to deliver the proposal for $w_2$, the causal ordering of the client requests will be violated and the replicated state will be incorrect.

Our failure model is crash-fail with stateful recovery. We do not assume synchronized clocks, but we do assume that servers perceive time pass at approximately the same rate. (We use timeouts to detect failures.) The processes that make up Zab have persistent state, so processes can restart after a failure and recover using the persistent state. This means a process can have state that is partially valid such as missing more recent transactions or, more problematic, the process may have transactions that were never delivered earlier and now must be skipped.

We provision to handle $f$ failures, but we also need to handle correlated recoverable failures, for example power outages. To recover from these failures we require that messages are recorded on the disk media of a quorum of disks before a message is delivered. (Very non-technical, pragmatic, operations-oriented reasons prevent us from incorporating devices like UPS devices, redundant/dedicated networking devices, and NVRAMs into our design.)

Although we do not assume Byzantine failures, we do detect data corruption using digests. We also add extra metadata to protocol packets and use them for sanity checks. We abort the server process if we detect data corruption or sanity checks fail.

Practical issues of independent implementations of operating environments and of the protocols themselves make the realization of fully Byzantine tolerant systems impractical for our application. It has also been shown that achieving truly reliable independent implementations requires more than just programming resources [11]. To date, most of our production problems have been either the result of implementation bugs that affect all replicas or problems that

are outside the implementation scope of Zab, but affect Zab, such as network misconfiguration.

ZooKeeper uses an in-memory database and stores transaction logs and periodic snapshots on disk. Zab's transaction log doubles as the database write-ahead transaction log so that a transaction will only be written to the disk once. Since the database is in-memory and we use gigabit interface cards, the bottleneck is the disk I/O on the writes. To mitigate the disk I/O bottleneck we write transactions in batches so that we can record multiple transactions in a single write to the disk. This batching happens at the replicas not at the the protocol level, so the implementation is much closer to group commits [4, 6] from databases than message packing [5]. We chose to not use message packing to minimize latency, while still getting most of the benefits of packing through batched I/O to the disk.

Our crash-fail with stateful recovery model means that when a server recovers, it is going to read in its snapshot and replay all delivered transactions after that snapshot. Thus, during recovery the atomic broadcast does not need to guarantee at most once delivery. Our use of idempotent transactions means that multiple delivery of a transaction is fine as long as on restart order is preserved. This is a relaxation of the total order requirement. Specifically, if $a$ is delivered before $b$ and after a failure $a$ is redelivered, $b$ will also be redelivered after $a$.

Other performance requirements we have are:

**Low latency:** ZooKeeper is used extensively by applications and our users expect low response times.

**Bursty high throughput:** Applications using ZooKeeper usually have read-dominant workloads, but occasionally radical reconfigurations occur that cause large spikes in write throughput.

**Smooth failure handling:** If a server fails that is not the leader and there is still a quorum of correct servers, there should be no service interruption.

## 3. WHY ANOTHER PROTOCOL

Reliable broadcast protocols can present different semantics depending on the application requirements. For example, Birman and Joseph propose two primitives, ABCAST and CBCAST, that satisfy total order and causal order, respectively [2]. Zab also provides causal and total ordering guarantees.

A good protocol candidate in our case is *Paxos* [12]. Paxos has a number of important properties, such as guaranteeing safety despite the number of process failures, allowing processes to crash and recover, and enabling operations to commit within three communication steps under realistic assumptions. We observe that there are a couple of realistic assumptions we can make that enable us to simplify Paxos and obtain high throughput. First, Paxos tolerates message losses and reordering of messages. By using TCP to communicate between pairs of servers, we can guarantee that delivered messages are delivered in FIFO order, which enables us to satisfy per proposer causality even when server processes have multiple outstanding proposed messages. Paxos, however, does not directly guarantee causality because it does not require FIFO channels.

Proposers are the agents proposing values for the different instances in Paxos. To guarantee progress, there must be a single proposer proposing, otherwise proposers may contend forever on a given instance. Such an active proposer is a leader. When Paxos recovers from a leader failure, the new leader makes sure all partially delivered messages are fully delivered and then resumes proposals from the instance number the old leader left off. Because multiple leaders can propose a value for a given instance two problems arise. First, proposals can conflict. Paxos uses ballots to detect and resolve conflicting proposals. Second, it is not enough to know that a given instance number has been committed, processes must also be able to figure out which value has been committed. Zab avoids both of these problems by ensuring that there is only one message proposal for a given proposal number. This obviates the need for ballots and simplifies recovery. In Paxos, if a server believes it is a leader, it will use a higher ballot to take over leadership from a previous leader. However, in Zab a new leader cannot take over leadership until a quorum of servers abandon the previous leader.

An alternative way of obtaining high throughput is by limiting the protocol message complexity per broadcast message such as with the Fixed-Sequencer Ring (FSR) protocol [8]. Throughput does not decrease with FSR as the system grows, but latency does increase with the number of processes, which is inconvenient in our environment. Virtual synchrony also enables high throughput when groups are stable for long enough [1]. However, any server failure triggers a reconfiguration of the service, thus causing brief service interruptions during such reconfigurations. Moreover, a failure detector in such systems has to monitor all servers. The reliability of such a failure detector is crucial for the stability and speed of reconfiguration. Leader-based protocols also rely upon failure detectors for liveness, but such a failure detector only monitors one server at a time, which is the leader. As we discuss in the next section, we do not use fixed quorums or groups for writes and maintain the service availability as long as the leader does not fail.

Our protocol has a fixed sequencer, according to the classification of Defago *et al.* [3], that we call the leader. Such a leader is elected through a leader election algorithm and synchronized with a quorum of other servers, called followers. Since the leader has to manage messages to all the followers this decision of a fixed sequencer distributes load unevenly across the servers that compose the system with respect to the broadcast protocol. We have taken this approach, though, for the following reasons:

- Clients can connect to any server, and servers have to serve read operations locally and maintain information about the session of a client. This extra load of a follower process (a process that is not a leader) makes the load more evenly distributed;
- The number of servers involved is small. This means that the network communication overhead does not become the bottleneck that can affect fixed sequencer protocols;
- Implementing more complex approaches was not necessary as this simple one provides enough performance.

Having a moving sequencer, for example, increases the complexity of the implementation as we have to handle problems such as losing the token. Also, we opted to move away from models based on communication history, such as sender-based ones, to avoid the quadratic message complexity of such protocols. Destination agreement protocols suffer from a similar problem [8].

Using a leader requires that we recover from leader failures

to guarantee progress. We use techniques related to view changes such as in the protocol of Keidar and Dolev [10]. Different from their protocol, we do not operate using group communication. If a new server joins or leaves (perhaps by crashing), then we do no cause a view change, unless such an event corresponds to the leader crashing.

# 4. PROTOCOL

Zab's protocol consists of two modes: recovery and broadcast. When the service starts or after a leader failure, Zab transitions to recovery mode. Recovery mode ends when a leader emerges and a quorum of servers have synchronized their state with the leader. Synchronizing their state consists of guaranteeing that the leader and new server have the same state.

Once a leader has a quorum of synchronized followers, it can begin to broadcast messages. As we mentioned in the introduction, the ZooKeeper service itself uses a leader to process requests. The leader is the server that executes a broadcast by initiating the broadcast protocol, and any server other than the leader that needs to broadcast a message first forwards it to the leader. By using the leader that emerges from the recovery mode as both the leader to process write requests and to coordinate the broadcast protocol, we eliminate the network latency of forwarding messages to broadcast from the write request leader to the broadcast protocol leader.

Once a leader has synchronized with a quorum of followers, it begins to broadcast messages. If a Zab server comes online while a leader is actively broadcasting messages, the server will start in recovery mode, discover and synchronize with the leader, and start participating in the message broadcasts. The service remains in broadcast mode until the leader fails or it no longer has a quorum of followers. Any quorum of followers are sufficient for a leader and the service to stay active. For example, a Zab service made up of three servers where one is a leader and the two other servers are followers will move to broadcast mode. If one of the followers die, there will be no interruption in service since the leader will still have a quorum. If the follower recovers and the other dies, there will still be no service interruption.

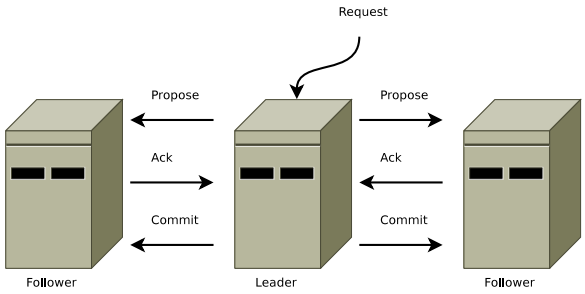


**Figure 2: The protocol for message delivery. Multiple instances of this protocol can be running concurrently. The message is delivered once the leader issues the COMMIT.**

## 4.1 Broadcast

The protocol we use while the atomic broadcast service is operational, called broadcast mode, resembles a simple a two-phase commit [7]: a leader proposes a request, collects votes, and finally commits. Figure 2 illustrates the flow of messages with our protocol. We are able to simplify the two-phase commit protocol because we do not have aborts; followers either acknowledge the leader's proposal or they abandon the leader. The lack of aborts also mean that we can commit once a quorum of servers ack the proposal rather than waiting for all servers to respond. This simplified two-phase commit by itself cannot handle leader failures, so we will add recovery mode to handle leader failures.

The broadcast protocol uses FIFO (TCP) channels for all communications. By using FIFO channels, preserving the ordering guarantees becomes very easy. Messages are delivered in order through FIFO channels; as long as messages are processed as they are received, order is preserved.

The leader broadcasts a proposal for a message to be delivered. Before proposing a message the leader assigns a monotonically increasing unique id, called the zxid. Because Zab preserves causal ordering, the delivered messages will also be ordered by their zxids. Broadcasting consists of putting the proposal with the message attached into the outgoing queue for each follower to be sent through the FIFO channel to the follower. When a follower receives a proposal, it writes it to disk, batching when it can, and sends an acknowledgement to the leader as soon as the proposal is on the disk media. When a leader receives ACKs from a quorum, the leader will broadcast a COMMIT and deliver the message locally. Followers deliver the message when they receive the COMMIT from the leader.

Note that the leader does not have to send the COMMIT if the followers broadcast the ACKs to each other. Not only does this modification increases the amount of network traffic, but it also requires a complete communication graph rather than the simple star topology, which is easier to manage from the point of view of starting TCP connections. Maintaining this graph and tracking the ACKs at the clients was also deemed an unacceptable complication in our implementation.

## 4.2 Recovery

This simple broadcast protocol works great until the leader fails or loses a quorum of followers. To guarantee progress, a recovery procedure is necessary to elect a new leader and bring all servers to a correct state. For leader election we need an algorithm that succeeds with high probability to guarantee liveness. The leader election protocol enables not only the leader to learn that it is the leader, but also a quorum to agree upon this decision. If the election phase completes erroneously, servers will not make progress, they eventually timeout, and restart the leader election. In our implementation we have two different implementations of leader election. The fastest completes leader election in just a few hundreds of milliseconds if there is a quorum of correct servers.

Part of the complication of the recovery procedure is the sheer number of proposals that can be in flight at a given time. The maximum number of in-flight proposals is a configurable option, but the default is one thousand. To enable such a protocol to work despite failures of the leader there are two specific guarantees we need to make: we must never forget delivered messages and we need to let go of messages that are skipped.

A message that gets delivered on one machine must be

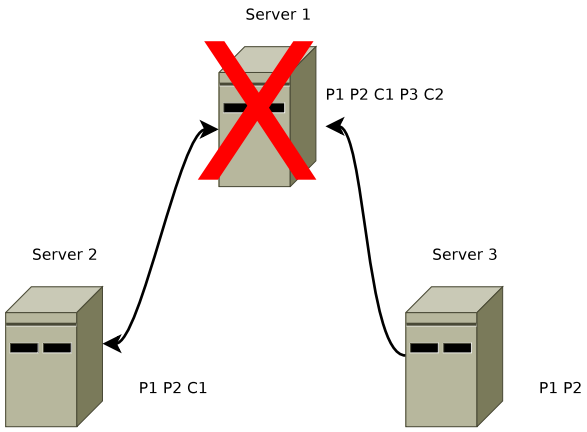


**Figure 3: An example of a message that cannot be forgotten. Server 1 is the leader. It fails, and it was the only only server to see the COMMIT of message 2. The broadcast protocol must ensure that message 2 gets committed on all correct servers.**

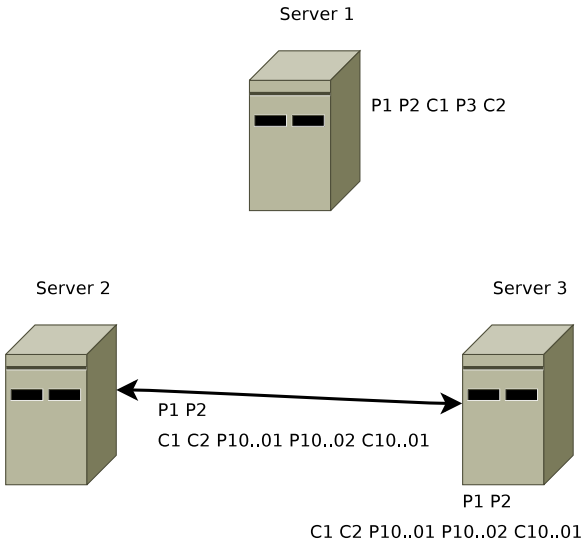


**Figure 4: An example of a message that must be skipped. Server 1 has a proposal for message 3 that was not committed. Since later proposals have been committed message 3 must be forgotten.**

delivered on all even if that machine fails. This situation can easily occur if the leader commits a message and then fails before the COMMIT reaches any other server as shown in Figure 3. Because the leader committed the message, a client could have seen the result of the transaction in the message, so the transaction must be delivered to all other servers eventually so that the client sees a consistent view of the service.

Conversely, a skipped message must remain skipped. Again, this situation can easily occur if a proposal gets generated by a leader and the leader fails before anyone else sees the proposal. For example, in Figure 3 no other server saw proposal number 3, so in Figure 4 when server 1 comes back up it needs to be sure to throw away proposal number 3 when it reintegrates with the system. If server 1 became a new leader and committed 3 after messages 100000001 and 100000002 have been delivered, we would violate our ordering guarantees.

The problem of remembering delivered messages is handled with a simple tweak of the leader election protocol. If the leader election protocol guarantees that the new leader has the highest proposal number in a quorum of servers, a newly elected leader will also have all committed messages. Before proposing any new messages a newly elected leader first makes sure that all messages that are in its transaction log have been proposed to and committed by a quorum of followers. Note that having the new leader being a server process with highest zxid is an optimization as a newly elected leader in this case does not need to find out from a group of followers which one contains the highest zxid and fetch missing transactions.

All servers that are operating correctly will either be a leader or be following a leader. The leader ensures that its followers have seen all the proposals and have delivered all that have been committed. It accomplishes this task by queuing to a newly connected follower any PROPOSAL it has that the follower has not seen, and then queuing a COMMIT for all such proposals up to the last message committed. After all such messages have been queued, the leader adds the follower to the broadcast list for future PROPOSALs and ACKs.

Skipping messages that have been proposed but never delivered is also simple to handle. In our implementation the zxid is a 64-bit number with the lower 32-bits used as a simple counter. Each proposal increments the counter. The high order 32-bit is the epoch. Each time a new leader takes over it will get the epoch of the highest zxid it has in its log, increment the epoch, and using a zxid with the epoch bits set to this new epoch and the counter set to zero. Using epochs to mark leadership changes and requiring that a quorum of servers recognize a server as the leader for that epoch, we avoid the possibility of multiple leaders issuing different proposals with the same zxid. One advantage of this scheme is that we can skip instances upon a leader failure, thus speeding up and simplifying the recovery process. If a server that has been down is restarted with a proposal that was never delivered from a previous epoch, it is not able to be a new leader since every possible quorum of servers has a server with a proposal from a newer epoch and therefore a higher zxid. When this server connects as a follower, the leader checks the last committed proposal for the epoch of the follower's latest proposal and tell the follower to truncate its transaction log (*i.e.*, forget) to the last committed

proposal for that epoch. In Figure 4 when server 1 connects to the leader, the leader tells it to purge proposal 3 from its transaction log.

# 5. FINAL REMARKS

We were able to implement this protocol quickly and it has proven to be robust in production. Most important, we met our goals of high throughput and low latency. On a non-saturated system, latencies are measured in milliseconds since the typical latencies will correspond to 4 times the packet transmission latency irrespective of the number of servers. Bursty loads are also handled gracefully since messages are committed when a quorum ack proposals. A slow server will not affect bursty throughput since fast quorums of servers that do not contain the slow server can ack the messages. Finally, since the leader will commit messages as soon as any quorum acks a proposal followers that fail do not affect the availability or even the throughput of the service as long as there is a quorum of correct servers.

As result of using efficient implementations of these protocols, we have an implementation of the system that reaches tens to hundreds of thousands of operations per second for workload mixes of 2:1 read-to-write ratios and higher. This system is currently in production use, and it is used by large applications such as the Yahoo! crawler and the Yahoo! advertisement system.

## Acknowledgements

We would to thank both of our reviewers for useful comments, and Robbert van Renesse for a very nice offline discussion on Zab which helped us to clarify several points.

# 6. REFERENCES

[1] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[5] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, pages 233–242, 1997.

[6] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[7] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[8] 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.

[9] http://hadoop.apache.org/zookeeper. Zookeeper project page, 2008.

[10] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions. In D. R. Avresky, editor, *Dependable Network Computing*, chapter 3, pages 51–75. Kluwer Academic, 2000.

[11] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986.

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

[13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.