Brief Announcement: A Shared Disk on Distributed Storage

Stefan Vijzelaar
sjj.vijzelaar@few.vu.nl

Herbert Bos
herbertb@few.vu.nl

Wan Fokkink
wanf@few.vu.nl

Department of Computer Science, Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

ABSTRACT

A shared disk implementation on distributed storage requires consistent behavior of disk operations. Deterministic consensus on such behavior is impossible when even a single storage node can fail. Atomic registers show how consistency can be achieved without reaching consensus, but suffer from a crash consistency problem. The presented shared disk algorithm, based on atomic registers and probabilistic consensus, can survive multiple storage node failures, as long as a majority of nodes respond.

Categories and Subject Descriptors: D.4.5 [Operating Systems]: Reliability – fault-tolerance, verification; H.3.4 [Information Storage and Retrieval]: Systems and Software – distributed systems.

General Terms: Algorithms, Reliability, Verification.

1. INTRODUCTION

Sharing storage is an important application of network technology. Storage can be shared at the block level, in storage area networks (SAN); or at the file level, as network attached storage (NAS). Nodes accessing a SAN or NAS may use shared disk or distributed filesystems. Reliability in such a shared setting is a major concern, since the potential impact of system failure is high. A failure-resistant shared disk implementation could be used as a reliable basis for both shared disk and distributed filesystems.

A shared disk implementation based on distributed storage should behave similarly to a single shared disk. In other words, the observed behavior of the disk should always be consistent with one or more possible orderings of disk operations. These so-called shared disk semantics can be difficult to achieve in an asynchronous distributed system, especially when confronted with failing nodes.

Deterministic consensus on the ordering of disk operations is impossible [6]; however, atomic registers [10, 2, 9] show that consistent behavior can be achieved without reaching consensus. Atomic registers are able to implement a consistency model called linearizability [7]. Unfortunately, this model suffers from a crash consistency problem: if a writer crashes, its write operation can resurface at an arbitrary time in the future.

Shared disk semantics will require a stronger consistency model: strict linearizability [1]. This model ensures incomplete writes can only take effect before the next read. A subsequent read will either return the new value, if the write was successful; or an older value, if the write failed. However, since there is no way to distinguish slow from crashed processes, some operations may need to be aborted to prevent obstruction [4, 3].

The shared disk algorithm as presented below, combines atomic registers with probabilistic consensus to prevent unnecessary aborts. It benefits from the fault-tolerance of atomic registers in general, and uses probabilistic consensus to resolve crash consistency in particular.

PODC'10, July 25–28, 2010, Zurich, Switzerland.
ACM 978-1-60558-888-9/10/07.

2. THE SHARED DISK ALGORITHM

For the shared disk algorithm an asynchronous network connecting client nodes with multiple storage server nodes is assumed. Clients need to communicate with a majority of the storage servers, when reading from or writing to the shared disk. This majority or quorum ensures subsequent disk operations have at least one storage server in common.

A typical algorithm for atomic registers [2] works as follows. Each storage server records the most recent value and timestamp pair it encounters. Write operations create timestamps for new values, and send the pair to a majority of the storage servers. The timestamp is used to identify the most recent pair. Read operations request value and timestamp pairs from a majority of the storage servers. The most recent pair is selected and sent to a majority of the storage servers, ensuring subsequent reads will detect either this pair, or a more recent pair.

When a client crashes during a write operation, its pair might not have reached a majority of servers. It is therefore possible that subsequent reads will not detect this write operation for some time. We have a crash consistency problem. This situation is however not unique to incomplete writes: concurrent writes might not be detected either. It raises the question of how to distinguish between crashed and concurrent write operations. Aborting undetected write operations, could prevent concurrent operations from completing.

Our shared disk algorithm trades timestamps for round numbers. This allows it to run a probabilistic consensus algorithm with voting rounds, to solve the crash consistency problem. Each storage server stores a single tuple containing a value, a highest round and a concurrent round. It records the highest round received from any client, together with the first value voted for within that round. The concurrent round records the highest round with possible votes for more than one value.

The information contained in a set of tuples can be combined to form a new tuple. This can be done by both clients and storage servers. Take for example the tuple $\langle a, 2, 1\rangle$, indicating the highest voting round is 2 which was first seen together with a vote for value a; and the tuple $\langle b, 3, 1\rangle$, indicating the highest voting round is 3 which was first seen together with a vote for value b. These tuples can be combined to $\langle b, 3, 2\rangle$: value b in the higher round 3 takes precedence over value a from the lower round 2. The concurrent value is updated to 2, because in that round both values a and b could have votes.

The goal of the algorithm is to advance a value two rounds ahead of the concurrent round: the last two rounds should only contain votes for a single unique value. Other, possibly concurrent, clients are then guaranteed to detect this value. In the worst case they would write values to the second highest round, and detect the unique value afterward. This form of probabilistic consensus is used to determine the result of a disk operation. If consensus is reached within a predetermined number of rounds, the disk operation succeeds, else it aborts.

Disk operations start by the client requesting tuples from a majority of the storage servers. The information in these tuples is combined to form a new tuple. This tuple is then sent to a majority of the storage servers to ensure all subsequent operations detect at least this round. Subsequent rounds can now be started by advancing a value to the next round: either an existing value in case of a read operation, or a new value in case of a write operation. Tuples are repeatedly bounced and combined between clients and a majority of storage servers until we either reach consensus or abort. To make a correct decision, it is essential to let values advance only one round at a time.

Notice that probabilistic consensus is always successful when only a single new value is proposed. In other words: aborts can be completely ruled out when the shared disk encounters no concurrent disk operations while performing a write. This is not an unreasonable assumption for a shared disk storing a filesystem. Read operations concurrent with other reads will always succeed.

The distinguishing feature of the presented algorithm is the combination of both atomic registers [2] and probabilistic consensus [5]. Thanks to probabilistic consensus, we can allow for limited amounts of concurrency and can still guarantee consistency when operations have to abort. The FAB distributed disk array [11], for example, aborts when any concurrency is detected; and it requires synchronized clocks to minimize this risk.

3. BOUNDED TIMESTAMPS

The benefit of using bounded timestamps [8], or in our case round numbers, is that they come from a finite domain. This is useful for implementations where timestamps are stored as finite data types, or in model checking where unbounded timestamps would result in an infinite statespace. Bounded timestamps are therefore often presented as a convenient replacement for unbounded timestamps; for example, when implementing atomic registers [2].

Our shared disk algorithm has a preliminary implementation for MINIX which has been model checked using Spin. It turned out that using bounded timestamps, causes a problem similar to the crash consistency we are trying to solve. Resurfacing values of incomplete write operations not only prevent shared disk semantics, but can also block a bounded timestamp algorithm. To side-step this problem, the model for Spin had to be limited to a bounded range of timestamps.

Bounded timestamps are calculated using a list of all potentially active timestamps in the system. It is however not sufficient to ensure that the input for the timestamp algorithm contains at least all currently active timestamps in the system. Take for example a simple system using stamps from the set $\{0, 1, 2\}$, with the circular relation $0 < 1 < 2 < 0$. This relative order states that 2 is the larger timestamp when compared to 1, but the smaller timestamp when compared to 0. An input set containing two timestamps will block the timestamp system, since it cannot select a timestamp larger than those contained in the set. A similar problem can occur when values, partially recorded by a failing node, reappear at an inopportune moment.

Unless a bounded timestamp algorithm can be found that works for an arbitrary input set of timestamps, it is impossible to apply bounded timestamps to the problem at hand.

4. CONCLUSION

The ability of atomic registers to achieve consistency without reaching consensus can not be directly translated to a shared disk algorithm. Shared disk semantics require some form of consensus to decide on the state of disk operations. Since it is impossible to distinguish between crashed and slow nodes, there is a chance of aborting legitimate concurrent disk operations. Our algorithm shows the risk can be mitigated by either limiting concurrency or increasing the number of rounds for reaching consensus. Consistency is always guaranteed.

5. REFERENCES

[1] Marcos K. Aguilera and Svend Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Laboratories Palo Alto, 2003.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4):1–33, 2009.

[4] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

[5] Ling Cheung. Randomized wait-free consensus using an atomicity assumption. In *Proceedings OPODIS 2005*, pages 36–45, 2005.

[6] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[7] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[8] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[9] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 71–82. ACM, 1992.

[10] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[11] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM, 2004.