A High Throughput Atomic Storage Algorithm

Rachid Guerraoui
EPFL, Lausanne
Switzerland

Dejan Kostić
EPFL, Lausanne
Switzerland

Ron R. Levy
EPFL, Lausanne
Switzerland

Vivien Quéma
CNRS, Grenoble
France

Abstract

This paper presents an algorithm to ensure the atomicity of a distributed storage that can be read and written by any number of clients. In failure-free and synchronous situations, and even if there is contention, our algorithm has a high write throughput and a read throughput that grows linearly with the number of available servers. The algorithm is devised with a homogeneous cluster of servers in mind. It organizes servers around a ring and assumes point-to-point communication. It is resilient to the crash failure of any number of readers and writers as well as to the crash failure of all but one server. We evaluated our algorithm on a cluster of 24 nodes with dual fast ethernet network interfaces (100 Mbps). We achieve 81 Mbps of write throughput and 8×90 *Mbps of read throughput (with up to 8 servers) which conveys the linear scalability with the number of servers.*

1 Introduction

Distributed storage systems [1, 7, 20, 26] are gaining in popularity as appealing alternatives to their expensive controller-based counterparts. A distributed storage system relies on a *cluster* of cheap distributed commodity machines. The goal of this distribution is to ensure resilience on the one hand and, on the other hand, to boost performance by adjusting the number of servers to the number of clients to be served concurrently. At the heart of such systems lies a *storage algorithm*. In short, such algorithms allow concurrent clients to share information through shared read/write objects (register abstractions [21]) implemented over a set of failure-prone servers. Distributed storage systems combine multiple of these read/write objects, each storing its share of data, as building blocks for a single large storage systems. Not surprisingly, the performance of such a storage system depends on the performance of the underlying algorithm implementing the read/write objects.

We consider in this paper an *atomic* and *resilient* storage that can be read or written by an unbounded number of clients. An *atomic* storage guarantees that, despite concurrent invocations, every *read* or *write* operation appears to execute instantaneously at some point between its invocation and completion instants [18, 21]. It was recently argued [20, 26] that atomicity is a desired property for distributed storage systems. In our context, *resilience* means that every non-faulty client eventually gets a reply to every (read or write) invocation despite the failures of other clients or (a subset of the) servers. We focus on crash failures and we consider a homogeneous cluster of server machines. Such clusters usually have low inter-server communication latency and fine tuned TCP channels that make failure detection *reliable*. However, one might expect that such clusters also deliver low latency client operations, especially in failure-free and synchronous situations for these are considered the most frequent in practice.

Studying lower bounds on the latency of distributed storage algorithms has been a very active area of research in the last decade [2, 12, 16]. In general, such studies focus on the *isolated* latency of a read or a write operation, assuming in particular that every server is ready to perform this operation. In practice, when a high number of clients are served concurrently, low overall latency can only be provided with high *throughput*. In short, under high load, the latency perceived by clients is the sum of the time spent waiting for the operation to be served plus the actual service time. Clearly, when a lot of clients access the storage concurrently, the higher the throughput, the smaller the waiting time. Ideally, one would aim at *scalability*, meaning that increasing the number of machines should improve the throughput of the storage system.

To motivate the design of our algorithm and illustrate why studying isolated latency might be misleading, we compare in Figure 1 two algorithms. A *quorum-based* traditional one [4, 24] and a less traditional one without inter-server communication. The example involves three servers and clients performing read operations on the storage (Figure 1). Clients always first contact a single server and communication between servers proceeds in a round-based manner. For simplicity, we assume that sending and receiving a message always takes the same time: one round. Therefore in each round, a server can receive a single message and send a single message. Algorithm A is a majority based algorithm: 2 out of 3 servers are needed to complete each operation. Upon receiving a request, server s_1 contacts s_2, and upon receiving a reply from s_2, replies to the client. Likewise, s_2 contacts s_3 and s_3 contacts s_1. The servers need 3 rounds before they can receive a new client request. Under full load, the servers can complete 3 requests every 3 rounds, inducing a throughput of 1 read operation per round. In algorithm B the servers do not communicate in order to complete a read request. The latency is the same as that of algorithm A: 4 rounds. However, after an initial period of 4 rounds, the servers can complete 3 read operations each round, achieving a throughput of 3 read operations per round.

Figure 1 also illustrates why quorum-based algorithms do not scale: a majority of servers need to receive all messages and adding more servers does not help. The problem is exacerbated by the fact that quorum-based algorithms typically use one-to-many communication patterns (multicasts) to disseminate the information quickly. The rationale is mainly that the cost of receiving one message is equal to that of receiving multiple messages, especially when compared to the message propagation time. While this might be true in widely distributed environments (e.g., Internet), this assumption does not hold in a cluster environment subject to a heavy load, which we consider in this work. Clearly, techniques that aim at optimizing latency of isolated operations are not necessarily the best when high throughput is desired.[1]

In this work, we exploit the availability of reliable failure detection in a homogeneous cluster environment to alleviate the need for quorum-based strategies. In fact, it might appear trivial to devise a storage algorithm with high throughput if failure detection is reliable. This is not however the case as we discuss below. First, atomicity and resilience induce an inherent trade-off between the throughput of reads and that of writes. Basically, the more servers are updated by a write, the less servers need to be consulted by a read in order to fetch the last written value (and vice-versa). It is typical to favor reads at the expense of writes, following the argument that reads should be expedited for they occur more frequently than writes. Since in our case maximum resilience (tolerating the failure of all but one server) is required, the writer should update all servers. In this case, a simple *read-one write-all-available* algorithm might appear to do the job. To ensure atomicity however, one needs to solve the *read-inversion* problem and prevent any read from returning an old value after an earlier read returned a new value. One way to address this issue is to add a write phase to every read. However, this clearly decreases the throughput. Besides, if write messages are simply broadcast to all servers, the throughput would suffer even more drastically under high load. Modern full-duplex network interfaces can indeed receive and send messages at the same time. However, when receiving several messages at the same time, collisions occur at the network layer [8]. A retransmission is thus necessary, in turn causing even more collisions, ultimately harming the throughput of write operations.

[1]Certain techniques can be used to improve the throughput of quorum-based algorithms. In [26] for instance, reads that are issued during contention free periods are handled more efficiently than reads that are concurrent with write operations. Yet reads still need to contact at least a majority of servers, which means that under high loads, there is no improvement of the throughput.

We present in this paper an atomic storage algorithm that is resilient to the crash failure of any number of readers and writers as well as to the crash failure of all but one server. In failure-free and synchronous periods, our algorithm has a high write throughput and a read throughput that grows linearly with the number of available servers. This is ensured even in the face of contention. Our algorithm is based on two key ideas. First, we ensure atomicity and prevent the read-inversion problem by adding a *pre-write* phase to the write (instead of a *write-phase* to the read). This idea is, we believe, interesting in its own right because reads are local and immediate when there is no contention. Second, we organize the servers following a *ring* to ensure constant write throughput and avoid collisions during concurrent writes. This second idea was also used in implementing a total order broadcast primitive [3, 10, 15], and one might actually wonder here whether it would not have been interesting to consider a modular approach in devising an atomic storage algorithm using such a primitive. Ensuring the atomicity of the storage would however have required to also totally order the reads, hampering its scalability. In [28], servers are organized in a chain to ensure high throughput for replica updates. This replication scheme can then be used to obtain a distributed atomic storage with high write throughput. However, the reads (also called queries) are always directed to the same single server and are therefore not scalable. We ensure liveness by imposing a *fairness* strategy on the servers.

We evaluated the implementation of our algorithm on a cluster of 24 machines (dual Intel 900MHz Itanium-2 processors, 3GB of RAM) with dual fast ethernet network interfaces (100 Mbps). We achieve 81 Mbps of write throughput and 8×90 Mbps of read throughput (with up to 8 servers). To our knowledge, our algorithm is the first atomic storage algorithm to achieve a read throughput that grows linearly with the number of available servers.

The paper is organized as follows. Section 2 describes our system model. Section 3 presents the algorithm. The performance is discussed in Section 4 and 5. A correctness proof of our algorithm can be found in the full version of our paper [14].

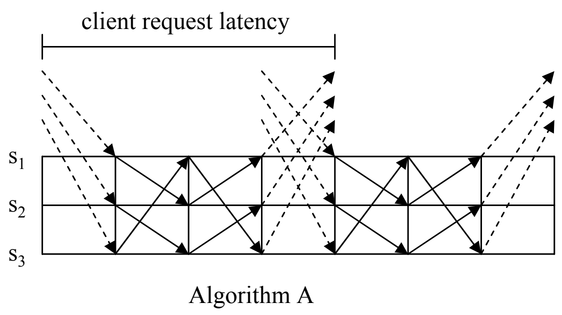


Figure 1. Throughput comparison between two algorithms: A and B. Both have the same latency but the throughput of B outperforms that of A. Client server communication messages are represented by the dotted lines.

2 Model

We consider a cluster environment where n homogeneous servers are connected via a local area network. We do not bound the number of client processes nor the concurrency among these. Any client can read or write in the storage. Every pair of processes communicate by message-passing using a bi-directional reliable communication channel (we do not assume FIFO channels here).[2]

We focus on process crash failures: when a process crashes, it stops performing any further computation steps. A process that crashes is said to be *faulty*. A process that is not faulty is said to be *correct*. We make no assumption on the number of possible crashes except that at least one server should be correct in every computation.

We use a ring communication pattern, meaning that servers are organized in a ring and communicate only with their neighbors. Each server creates a TCP connection to its successor in the ring and maintains this connection during the entire execution of the algorithm (unless the successor fails). Because of the simple communication pattern, the homogeneous environment and low local area network latency, it is reasonable to assume that when a TCP connection fails, the server on the other side of the connection failed [11]. Using this mechanism we implement a *Perfect* failure detector (P) [6]. Although our algorithm tolerates asynchrony, its performance is optimized for synchronous periods during which message transmission delays are bounded by some known value. The throughput is measured during such periods.

Evaluating the performance of message-passing protocols requires an adequate performance model. Some models only address point-to-point networks, where no native broadcast primitive is available [5, 9]. Our algorithm does not use any broadcast primitive, but we do not wish to exclude it from our performance model for the sake of comparison with other algorithms. A recently proposed model [27] is useful for reasoning about throughput, although it assumes that processes do not simultaneously send and receive messages. We would like to better capture the behavior of modern network cards which provide full duplex connectivity. The round-based model [13, 19, 23] is in that sense more convenient as it assumes that a process can send a message to one or more processes at the start of each round, and can receive the messages sent by other processes at the end of the round. It is however not realistic in our cluster context to consider that several messages can be simultaneously received by the same process. Indeed, receiving two messages at the same time might result in a collision at the network level, requiring a retransmission. Whereas, this model is well-suited for proving lower bounds on the latency of protocols, it is illsuited for predicting the throughput of these protocols.

We propose to evaluate message-passing protocols considering a synchronous round-based model but assuming the following: in each round k, every process p_i can execute the following steps: (1) p_i computes the message for round k, $m(i,k)$, (2) p_i broadcasts $m(i,k)$ to all or a subset of processes and (3) p_i receives at most one message sent at round k. The synchrony assumption implies that, at any time, all processes are in the same round. The broadcast primitive we assume corresponds to the multicast primitive provided at the ethernet level. There are no reliability guarantees, except in the absence of failures or collisions. The analytical analysis of the performance of our algorithm will be based on this model. Interestingly, the experimental evaluation confirms these numbers, conveying in some sense the validity of this model in our context of a homogeneous cluster.

[2]Even if we did assume FIFO channels, our fairness mechanism would make this assumption useless.

3 The Storage Algorithm

Our storage algorithm was designed with the three following properties in mind: resilience, atomicity and high throughput. Our algorithm satisfies these properties using two key mechanisms: a *read-one pre-write/write-all-available* strategy and a *fairness* rule orchestrating the servers in a *ring*. In this section, we explain these key mechanisms and how they ensure the desired properties.

This is the pseudo-code of our algorithm:

```
At the client $c$:
procedure write ($v$)
    send $\langle write$,v$\rangle$ to any $p_i \in S$
    wait until receive $\langle write\_ack\rangle$ from $p_i$
    return ok
end

procedure read ()
    send $\langle read\rangle$ to any $p_i \in S$
    wait until receive $\langle read\_ack$,v$\rangle$ from $p_i$
    return $v$
end

At the server process $p_i$:
procedure initialization:
    $v \leftarrow \perp, ts \leftarrow 0, id \leftarrow \perp$
    $pending\_write\_set \leftarrow \emptyset$
    $forward\_queue \leftarrow \emptyset$
    $write\_queue \leftarrow \emptyset$
    $nb\_msg[p_j] \leftarrow 0 \;\forall p_j \in S$
end

upon receive $\langle write, v'\rangle$ from $c$ do
    $write\_queue.last \leftarrow [v', c]$
end upon

procedure write($v'$, $c$)
    $highest = \max_{lex}(pending\_write\_set)$
    $tag \leftarrow [\max(highest.ts, ts) + 1, i]$
    $pending\_write\_set \leftarrow pending\_write\_set \cup tag$
    send $\langle pre\_write, c, v', tag\rangle$ to $p_{next}$
    $nb\_msg[p_i] \leftarrow nb\_msg[p_i] + 1$
    $write\_queue \leftarrow write\_queue - [v', c]$
end

upon receive $\langle pre\_write, v', [ts', id']\rangle$ do
    if $id \neq i$ then
        $forward\_queue.last \leftarrow \langle pre\_write, v', [ts', id']\rangle$
    else
        if $[ts', id'] >_{lex} [ts, id]$ then
            $[ts, id] \leftarrow [ts', id']$
            $v \leftarrow v'$
        end if
        $pending\_write\_set \leftarrow pending\_write\_set - [ts', id']$
        send $\langle write, v, [ts', id']\rangle$ to $p_{next}$
    end if
end upon

upon receive $\langle write, v', [ts', id']\rangle$ do
    if $id' \neq i$ then
        if $[ts', id'] >_{lex} [ts, id]$ then
            $[ts, id] \leftarrow [ts', id']$
            $v \leftarrow v'$
        end if
        $pending\_write\_set \leftarrow pending\_write\_set - [ts', id']$
        $forward\_queue.last \leftarrow \langle write, v', [ts', id']\rangle$
    else
        send $\langle write\_ack\rangle$ to $c$
    end if
end upon

task queue handler
    if $forward\_queue = \emptyset$ then
        $nb\_msg[p_j] \leftarrow 0 \;\forall p_j \in S$
        if $write\_queue \neq \emptyset$ then
            write($write\_queue.first$)
        end if
    else
        if $write\_queue \neq \emptyset$ then
            select $p_j$ s.t. $nb\_msg[p_j]$ is minimal
        else
            select $p_j \neq p_i$ s.t. $nb\_msg[p_j]$ is minimal
        end if
        if $p_j = p_i$ then
            write($write\_queue.first$)
        else
            $msg \leftarrow$ select first in $forward\_queue$ sent by $p_j$
            send $msg$ to $p_{next}$
            $forward\_queue \leftarrow forward\_queue - msg$
            $pending\_write\_set \leftarrow pending\_write\_set \cup msg.tag$
            $nb\_msg[p_j] \leftarrow nb\_msg[p_j] + 1$
        end if
    end if
end

upon receive $\langle read\rangle$ from $c$ do
    if $pending\_write\_set = \emptyset$ then
        send $\langle read\_ack, v\rangle$ to $c$
    else
        $highest = \max_{lex}(pending\_write\_set)$
        wait until receive $\langle write, v', [ts', id']\rangle \wedge ([ts', id'] \geq_{lex} highest)$

        send $\langle read, v'\rangle$ to $c$
    end if
end upon

upon $p_j$ crashed do
    if $p_j = p_{next}$ then
        $p_{next} = p_{j+1}$
        send $\langle write, v, [ts, id]\rangle$ to $p_{next}$
        for each $v', [ts', id'] \in pending\_write\_set$ do
            send $\langle pre\_write, v', [ts', id']\rangle$ to $p_{next}$
        end for
    end if
end upon
```

Clients send Read and Write requests to any server in S. If the server contacted by the client crashes, the client reissues the request to another server. Clients do not directly detect the failure of a server, but when their request times-out, they simply re-send it to another server.

As we pointed out, our algorithm is resilient in the sense that it tolerates the failure of $n-1$ out of n server processes and the failure of any number of clients. Atomicity [18, 22] dictates that every read or write operation appears to execute at an individual moment of time, between its invocation and responses. In particular, a read always returns the last written value, even if there is only one server that did not crash. We ensure this using a *write-all-available* scheme. Newly written values are sent to all processes before the write operation returns and each process keeps a local copy of the latest value.

Values are ordered using a timestamp which is stored together with the value. Processes only replace their locally stored values with values that have a higher timestamp. Since a write contacts all processes, a process wishing to perform a write does not need to contact any other process to get the highest timestamp. Before each write, the locally stored timestamp can simply be incremented, thus ensuring that timestamps increase monotonically. (Ties are broken using process ids).

Read operations do not involve any communication between server processes. Clients directly access the value stored locally at a server process. The difficulty here lies in ensuring atomicity using these *local* reads and in particular in preventing the *read inversion* problem. Consider an execution where a value is written and stored at all processes. Due to asynchrony, not all processes might learn about the new value at the same time. Before the write completes, a reader contacts a process that has the new value and thus returns the new value. Afterwards a second reader contacts a process which does not know of the new value (since the write is not yet completed) and thus returns the old value, violating atomicity.

Our algorithm handles this issue using a pre-write mechanism. The write involves two consecutive phases: a pre_write phase and a $write$ phase. In the pre_write phase, all processes are informed of the new value that is going to be written. Only when all processes acknowledge the pre_write, does the $write$ phase actually start.

During a read, if a process knows of a pre_write value that has not yet been written, it waits until the value has been written before returning it. This ensures that when the new value is returned, all processes know of the new value through the pre_write phase and any subsequent read will also return the new value. Consequently, when there are no unwritten pre_write values, processes can immediately return the latest written value. In the case of concurrent writes, processes might see several unwritten pre_write values, in which case they wait for the value with the highest timestamp to be written. As will be explained in Section 4, waiting increases the latency for a single request, but does not influence the throughput of a loaded system. An illustration of an algorithm execution is provided in Figure 2.

So far, no mention was made about the communication pattern that is used for contacting all processes during the actual writes. The choice of the communication pattern has no influence on the correctness of the algorithm, but it does influence the throughput. Our algorithm organizes all servers in a ring and messages are forwarded from each server to its neighbor. This simple communication pattern avoids any unpredictability causing message collisions, especially under high loads. Also, there is no need for explicit acknowledgment messages, since knowing that a message has been forwarded around the ring once implies that all processes have seen the message.

In the case of a crash of a server process p_j, the crash will eventually be detected by the crashed process' predecessor in the ring p_{j-1} using the perfect failure detector. The crashed process p_j will be removed from the ring. Any pending messages that were not forwarded due to the crash are forwarded to the new successor by p_j, ensuring that all *pre-write* and *write* messages are eventually forwarded around the ring. This however is not enough to ensure resilience. Under high loads, processes must decide to either forward messages received from their neighbor or initiate a new write upon receiving a request from a client. If each process would prioritize requests received from clients, no message would ever be forwarded on the ring.

Our algorithm handles this issue using a *fairness* mechanism which ensures that each process can complete its fair share of writes and that all write operations eventually complete. Each process keeps two queues: a $write\_queue$ which contains *write* requests received from clients and a $forward\_queue$ which contains messages received from the predecessor which are to be forwarded to the successor. A table nb_msg keeps track of how many messages have been forwarded for each process: there is an entry for each process p_j, counting the number of messages originating at p_j that were forwarded. Messages in the $forward\_queue$ are not forwarded in FIFO order, but the first message from the processes that has the smallest number of forwarded messages will be sent to the successor.

4 Analytical Evaluation

We consider two performance metrics: *Latency*, defined here as the number of rounds required for a client to complete a read or write invocation and *Throughput*, defined here as the number of invocations (read or write) that can be completed per round. Note that our throughput definition is similar to the one proposed in [17].

4.1 Latency

The read latency of our algorithm is equal to 2 rounds, since a read operation only requires 1 round-trip from the client to the server. The latency of a write operation is equal to $2N + 2$ rounds. A write operation first requires the client to send a $write$ message to the server (1 round). Then, the server sends a pre_write message along the ring (N rounds). Once it receives its own pre_write message, the server sends a $write$ message along the ring (N rounds). Finally, upon the reception of its own $write$ message, the server replies to the client with a $write\_ack$ message (1 round). The write latency is thus linear with respect to the number of servers.

4.2 Throughput

For simplicity of presentation, this analysis only considers messages exchanged between servers. Note that in our experimental setup, client messages do indeed transit on their own dedicated network. Our evaluation also shows however, that when clients and servers use the same network, they both share the available bandwith evenly. Assuming that there exists at least 1 server that receives

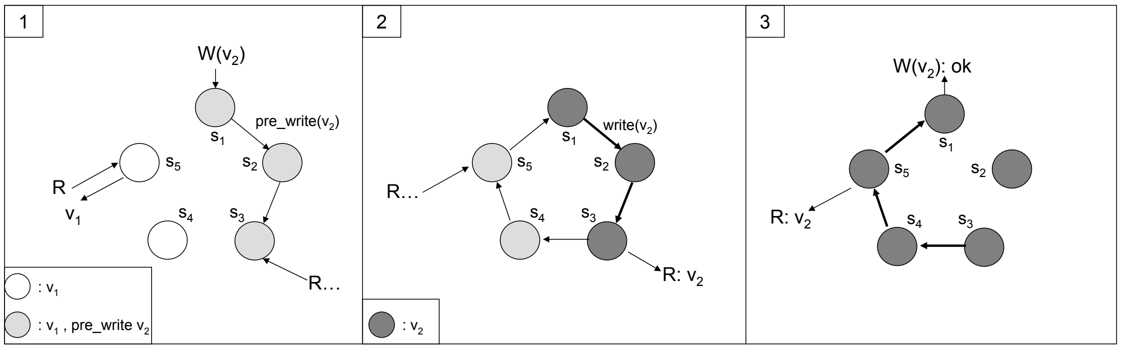


Figure 2. Illustration run of the storage algorithm. (1) After receiving the write request $W(v_2)$, s_1 sends a $pre\_write(v_2)$ message to its successor. A read request is received by s_3, which must wait before replying because of the pre-write, whereas s_5 can reply directly to the client's read request. (2) Upon receiving its own $pre\_write(v_2)$ message, s_1 sends a $write(v_2)$ message. Upon receiving this message s_3 can reply to its client's read request. Now s_5 must wait until it receives the $write(v_2)$ message before replying to a new read request. (3) Upon receiving its own $write(v_2)$ message, s_1 replies to the client and s_5 can also reply to its client.

1 new write request per round, our storage algorithm allows completing 1 write operation per round on average. This is due to the fact that (1) messages are disseminated along a ring once, (2) $write$ messages are piggybacked on $pending\_write$ messages without the need for explicit acknowledgements, and (3) the fairness mechanism guarantees that write requests eventually complete. This ensures that each server can forward a new write message at the end of each round, and thus the algorithm allows 1 write request per round to complete on average.

Assuming that there are only read requests, the read throughput is equal to n. This is due to the fact that each of the n servers can reply to a different read invocation at each round. Thus, increasing the number of machines does not impact the write throughput, and favorably impacts the read throughput.

We now analyze the impact that concurrent writes have on reads. The fairness mechanism that is integrated into our algorithm guarantees that 1 write can be completed per round on average, and that the maximum latency of a write request is bounded (let l_{max} be this maximum latency). Consider a server s_i which receives an infinite number of read requests. Before replying to the client, s_i must wait for the latest pre-write to complete, i.e. l_{max} rounds in the worst case. After an initial period of l_{max} rounds, s_i can fulfill 1 read request each round. The read throughput for s_i is therefore 1. Since reads do not involve additional communication between servers, each server can serve clients independently at a throughput of 1, bringing the total read throughput during R_{hl} to n.

Algorithms based on underlying total order broadcast primitives have the same throughput as the underlying atomic broadcast algorithm for both read and write operations. The highest throughput we know of for such algorithms is 1 [15]. Determining the throughput of a quorum-based algorithm is complicated in the general case and depends on the exact quorum configuration [25]. However, in [25] it is mathematically proven that the throughput of a quorum system cannot scale linearly as is the case in our algorithm.

5 Experimental Evaluation

Our algorithm was implemented in C (approximately 1500 lines). The implementation consists of separate code for a client (reader or writer) and a server. In order to stress the servers without needing an enormous number of client machines, the client application can emulate multiple clients, i.e. it can send multiple read and write requests in parallel. Thus, a single writing node can saturate the storage implementation (the servers and the network links) and so the maximum throughput, under high load, can be captured.

We performed the experiments using up to 24 homogeneous nodes (Linux 2.6, dual Intel 900MHz Itanium-2 processors, 3GB of RAM, 100Mbit/s switched ethernet). Similarly to the assumption made in Section 4.2, servers and clients are interconnected by two separate networks: server nodes are connected to each other on one network and communicate with clients on the other. The load is generated by two dedicated client machines for each server, either performing reads or writes depending on the experiment. Every measurement has been performed at least 3 times and the average over all measurements has been recorded.

Figure 3 depicts the results of the experiments. In the first chart, each server is connected to two client machines which generate read requests. There are no concurrent writes. The read throughput is measured at each client and the total is reported on the chart. It can easily be seen that the total read throughput increases linearly and is equal to 90 MBit/s per server. In the second chart, the clients generate only write requests. The write throughput when the number of servers is between 2 and 8 remains almost constant and is about 80 Mbit/s. It is also interesting to note that during the experiment, each client machine roughly observed the same write throughput, i.e. 80 Mbit/s divided by the number of servers.

The following experiment examines the total throughput of the system with write contention. The load on each server is generated by a dedicated reader and a dedicated writer. This represents a more realistic case in which read and write requests are issued concurrently by many clients. The results are shown on the third chart. The implementation behaves as predicted by our analytical analysis: the write throughput remains constant at around 80 Mbit/s and the read throughput scales linearly and is almost as high as in the contention free case (a performance penalty of about 15% is incurred). The decrease in performance is, we believe, due to the additional overhead of queuing client read requests while at the same time running the fairness mechanism for write requests.

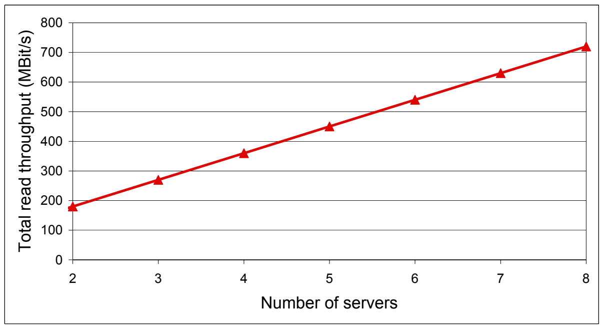
The next experiment examines the total throughput of the system during contention when clients and servers share a single network connection. The results are shown in bottom most chart. Obviously, read and write throughput suffer, but the write throughput remains constant at around 45 Mbit/s whereas the read throughput scales linearly at about 31 Mbit/s per additional server. This means that each server uses about 76 Mbit/s of its incoming and outgoing network bandwith despite concurrency.

The latency measurements are presented in Figure 4. Because of the ring topology, the write latency grows linearly with the number of servers. The read latency stays constant since it involves only a single round-trip between the client and a server.

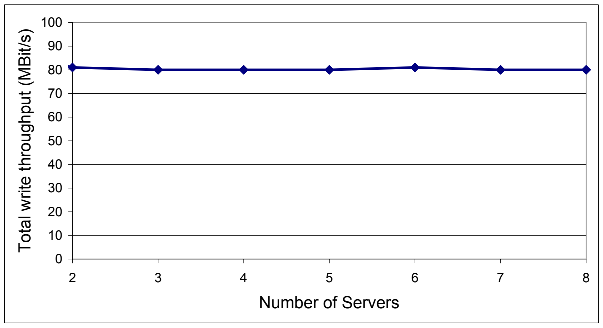
References

[1] M. Abd-El-Malek, W. V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*. USENIX, 2005.

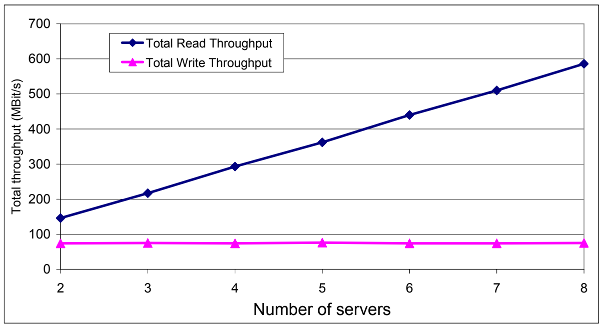
Read throughput without contention:



Write throughput without contention:



Read & write throughput contention on separate networks:



Read & write throughput contention on shared network:

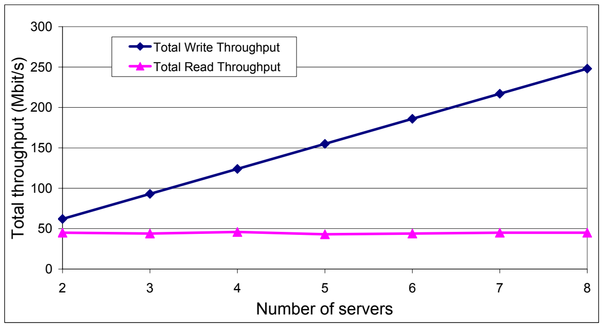


Figure 3. Read and write throughput.

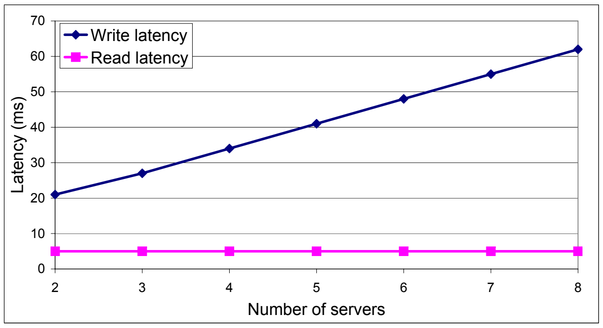


Figure 4. Read and write latency.

[2] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2005.

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[7] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993.

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

[11] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman. Fuse: Lightweight guaranteed distributed failure notification. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[12] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 236 – 245, 2004.

[13] E. Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing (PODC'98)*, pages 143–152, 1998.

[14] R. Guerraoui, R. R. Levy, D. Kostić, and V. Quéma. A high throughput atomic storage algorithm. In *http://lpd.epfl.ch/site/Publications*, 2007.

[15] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. High Throughput Total Order Broadcast for Cluster Environments. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, 2006.

[16] R. Guerraoui and M. Vukolic. How Fast Can a Very Robust Read Be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, 2006.

[17] D. Hendler and S. Kutten. Constructing shared objects that are both robust and high-throughput. In *Proceedings of 20th international symposium on distributed computing (DISC '06)*, pages 428–442, 2006.

[18] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[19] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC'06)*, pages 169–178, 2006.

[20] D. R. Kenchammana-Hosekote, R. A. Golding, C. Fleiner, and O. A. Zaki. The design and evaluation of network raid protocols. Research report RJ 10316, IBM Almaden Research Center, 2004.

[21] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[22] L. Lamport. The part-time parliament. *DEC SRC 1989. (Also in ACM Transactions on Computer Systems)*, 1998.

[23] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[24] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems (FTCS'97)*, 1997.

[25] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[26] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Operating Systems Review*, 38(5):48–58, 2004.

[27] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, pages 582–589, 2000.

[28] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of the 6th Symposium on Operationg Systems Design and Implementation*, December 2004.