Atomic Memory in Dynamic Networks

Readings:
*** Attiya, Welch
*** Rambo paper, long version
Rambo paper, DISC version

Next: Mobile computing, power reduction
*** Li, Halpern et al.
*** Bahramgiri et al.

1 Introduction

The problem of maintaining data objects in a distributed system is one of the most fundamental problems of distributed computing, both theory and practice.
The algorithms that manage such objects are supposed to make the objects look as much as possible like objects in a centralized memory.

However, because of issues of availability (the objects should be accessible even if some of the processes or channels fail), or performance (want copies of the objects close to the users), the objects are typically not kept in a single centralized location.
Replicated in various places in the system.

Then the question is how to manage the copies, in order to keep them "consistent" in some sense.
The most desirable kind of consistency is "atomicity", AKA linearizability.
But that can be expensive to achieve.

Most of you have already seen various strategies for implementing atomic objects in distributed systems.
E.g., in my book, I describe various standard strategies, e.g.:

- (Not fault-tolerant) Using Lamport timestamps. Works for arbitrary updates (which modify the state as a function of the previous state, but do not return any value). Attach timestamp to an update, send to everyone, order updates by timestamp, apply them when you know that you won't get any more with smaller timestamp. Since we are not considering failures here, we can tell when this is by waiting to receive a larger timestamp from everyone.
- (Fault-tolerant) Quorum-based approaches for reads/writes. Attiya, Bar-Noy, Dolev and extensions to allow multiple writers. Reads and writes can be carried out concurrently, using two-phase ($4d$) protocols that access read-quorums in the first phase and write-quorums in the second.
  Review this.
  Even with lots of concurrency, the intersection properties of quorums yield atomicity.
  Tolerates stopping failure of under half of the processes.

Today, I will talk about several results involving implementing atomic memory in a distributed setting. In addition to atomicity, I'll also consider sequential consistency.
Sequential consistency is the same as atomicity except that events at different nodes can be reordered. Formally, it is defined by the existence of a total ordering of all the operations such that, if an operation completes at any location before another operation starts at the same location, then the first operation must precede the second in the total order.
Atomicity adds the requirement that, if an operation finishes anywhere before another one starts anywhere, then the first operation must precede the second in the total order.

What I'll cover:

1. Revisit Lamport and other solutions based on replicated state machines, to see what consistency conditions they yield for the data.
2. Go over the paper by Attiya and Welch, which gives some basic upper and lower bounds for latency of operations on objects (read/write and some other kinds of objects), considering both sequential consistency and atomicity.
3. I'll skip some other material I prepared some time ago, based on a systems paper by Ladin, Liskov, et al. and a more theoretical related paper by Fekete et al. These present algorithms that ensure a somewhat weaker "eventual serializability" correctness condition. This material is in the notes below.
4. Cover the new Rambo algorithm, which gives atomic read/write data, with reasonable efficiency, in rather dynamic settings, tolerating failures.
   Note this is just for read/write objects; I don't have an efficient algorithm to yield atomicity for more general data types, e.g., read-modify-write, in fault-prone settings.
   In fact, note that, in the presence of continuing failures, this is impossible: RMW objects that can tolerate one failure allow solution of consensus, with one failure.
   However, one can seek algorithms that work well in situations in which stabilization occurs.
5. Seth Gilbert's extensions to Rambo, improving performance.

There is still a lot more to be done on this topic (and it's important).

2 The replicated state machine approach

Lamport originated the "replicated state machine approach" to building distributed systems, in his famous "Time, Clocks,..." paper.
The idea is simply that everyone should keep a copy of the object (total replication) and apply the same sequence of operations to that object.
He doesn't distinguish reads, writes, etc.—treats all operations the same.

It is easy to argue that this yields sequential consistency.
But in fact, he gets atomicity:
He uses an ordering based on logical timestamps, as described above.
When an operation returns, it has already acquired a timestamp, and also, everyone is guaranteed to have a local timestamp that is higher.
Therefore, no operation that gets invoked afterwards, anywhere, can acquire a smaller timestamp.

That paper does not consider failures.
In order to accommodate failures, something must be done to guarantee a consistent sequence everywhere, even in the presence of some faulty processes.

The much later Paxos multidecree algorithm is again designed to yield a consistent sequence of operations everywhere, this time in the face of a wide variety of (non-Byzantine) failures.
Sequential consistency follows easily from the consistent ordering.
He may also get atomicity, though the paper doesn't exactly say.
However, if an operation returns, it has already been assigned an index in the total ordering.
And if another operation is later invoked, the gap-filling policy MIGHT succeed in preventing the later operation from getting assigned to an earlier index.

Q: What if we build a replicated data service on top of an "atomic broadcast service" that is guaranteed to output all the operations everywhere, in a consistent order? Just apply the operations as they arrive and respond to a local operation after you've applied it.
That's surely sequentially consistent—is it also atomic?
A: Seems so—a node must know the position of a local op in the order before it can return a response. Any operation that gets invoked, anywhere, later, must be assigned a later position in the sequence.

3 Attiya, Welch

This paper examines the two notions of atomicity and s.c. more closely, and compares the costs of achieving them. Basic, theoretical upper and lower bound results about the cost (worst-case response time) of achieving sequential consistency vs. atomicity.
No faults considered.
Considers three kinds of objects: read/write objects, FIFO queues, and stacks, but I'll consider just r/w.

Results:

1. If processes have no clocks, or have clocks that are approximately synchronized, then there is a provable difference in the costs of the two kinds of consistency conditions. They get lower bounds for atomicity, and lower (in a sense) upper bounds for sequential consistency.
2. With perfectly synchronized clocks, there's no difference in cost.

3.1 Model

Application processes (external user processes) invoke read, write operations on particular objects. The operations are implemented by a collection of MCS (memory consistency system) processes at the same locations, communicating using a reliable network.

Each MCS process has a real-time local clock, running at the exact rate of real time.
For the first set of results, the local clocks' values are not synchronized with other clocks.
For the second set, they're all perfectly synchronized.
In either case, a process can use its clock to schedule an event after exactly a designated amount

of time has elapsed since another event.

Message delay known to be in interval $[d-u, d]$.
(As in earlier Lundelius-Lynch clock synchronization paper.)

3.2 Results for imperfect clocks

Actually, these are perfect duration timers, but unsynchronized.

3.2.1 Lower bounds for atomicity

Theorem 1: In any implementation of a single atomic read/write object for at least 3 processes:
1. Some read must take time at least $u/4$.
2. Some write must take time at least $u/2$.

Proof of part 1:
WLOG assume the initial value of X is 0.
Assume for contradiction that all reads take time $< u/4$ (writes may take as long as we want).
Construct an execution α in which process 1 reads X, obtaining the value 0.
Then process 3 begins a write of 1 to X.
While the write is going on, processes 1 and 2 alternate reading X, each happening to start right after the other one finishes.
Since the write eventually completes, eventually one of these reads must return 1.
Therefore, there exist a consecutive pair of reads, say R_1 by process 1 and R_2 by process 2, where R_1 returns 0 and R_2 returns 1.

This execution is under-specified. Suppose in addition that the message delays everywhere in α are all exactly the midpoint of the allowed range, $d-u/2$.

Now we can construct a new, bad execution α' by shifting the events at process 2 earlier by $u/2$.
Recall this shifting from the Lundelius, Lynch clock synch lower bound result.
It means to move all events (at the indicated process) earlier in real time, while adjusting the local clock of the shifted process so that everything looks the same to it.
The fact that the message delay is as indicated means that the shifting produces allowable message delays.

But in α', the read of 1 by process 2 completely precedes (we can resolve the race conditions to say this is so) the read of 0 by process 1.
Violates atomicity.

Proof of Part 2:
Similar. This time construct α in which process 1 writes 1 at time 0, then after this finishes, process 2 writes 2 at time $u/2$, then after this finishes, process 3 reads, getting the value 2.
Also, suppose that message delays are d from 1 to 2, $d-u$ from 2 to 1, and $d-u/2$ between every other pair of processes.

Now construct α' by shifting process 1's events later by time $u/2$ and process 2's events earlier by $u/2$.
This keeps the message delays in the allowed bounds.
But now, the write of 2 precedes the write of 1, which means that process 3 is supposed to get the value 2, not 1.

3.2.2 Upper bounds for sequential consistency

They describe two algorithms, one with fast reads and one with fast writes. In each case the fast operation is immediate and the slower operation is bounded by an upper bound h on the message delivery latency in a separate atomic broadcast service.

Atomic broadcast: Their version of this has simple requirements, since they do not deal with failures: Has ABsend inputs, ABreceive outputs. Guarantees that every message sent is received at every process, all messages are delivered in the same order at all processes, and two messages sent by the same process are delivered in the same order they were sent.

They describe (in the appendix), a fast implementation of atomic bcast, in which the latency bound h is only $2d$.
It's simple, looks like Lamport's timestamp-based totally-ordered bcast.
Keep a vector of the latest local timestamps (counters) known for everyone.
Tag messages with local timestamp (counter), send to all, accumulate timestamped messages in a buffer.
When you learn that everyone has reached a certain timestamp, you can deliver all the smaller-timestamped messages, in order.

Algorithm for fast reads:
Everyone keeps a local copy of every object.
A read returns the value of the local copy immediately.
When a write comes in to i, i broadcasts it using ABsend, but doesn't yet generate an ack to its client.
When a node receives a write operation using ABreceive, it writes to its local copy.
In addition, if the write was originally generated by this process, it then acks its client.

The algorithm appears to work for read/update as well.
I believe it's the same as the algorithm used in the Orca system.
Seems reasonably practical.

Yields sequential consistency: the total order constructed is consistent with the order of operations at any individual location.
The possibility of out-of-order reads happening at different locations shows that it doesn't yield atomicity, however.

Time 0 for read, $2d$ for writes.

Algorithm for fast writes:
A write results in an ABsend as before, but instead of waiting to receive its own write back from the ABCast service, the process acks to the client immediately.

It keeps track, however, of the number of writes it has acked but not gotten back from the ABCast, using a counter *num*.

A read results in waiting for the *num* counter to be 0, then reading the local copy and returning the value to the client.
Thus, the process waits for all the writes that it originated locally to pass through the ABCast service, before performing the read.

A "pipelined writes" strategy, in the terminology of multiprocessor memories.
Again, yields sequential consistency.
Now time 0 for writes, $2d$ for reads.

3.3 Perfect clocks

Here, it turns out that there isn't a distinction. So their results take the form of upper and lower bounds results that apply to both conditions.

3.3.1 Lower bound

For sequential consistency, and hence for linearizability.

Theorem 2: (Lipton, Sandberg) In any implementation of a sequentially consistent read/write memory, the sum of the worst-case latencies of read and write operations is at least d.

Proof: By contradiction, as before. Consider two objects, X and Y.
Construct execution α_1: process 1 writes 1 to X then, immediately after return, reads $Y = 0$.
All messages take time exactly d.
Thus, by assumption that the sum of the latencies is $< d$, all this is finished before any messages are delivered.

Symmetrically, construct α_2: process 2 writes 1 to Y then reads $X = 0$.
Again, all this is finished before any messages are delivered.

Then we can "paste" these two executions into one.
The resulting execution isn't sequentially consistent.
To see this, suppose we have some sequence of the four operations that's consistent with what is observed at each node.
In this sequence, one of the two writes must be ordered before the other, say $W(X,1)_1$ comes before $W(Y,1)_2$.
But then $R(X,0)_2$ must come after the $W(Y,1)_2$, since the order is consistent with the local order of operations at process 2.
But this means that $R(X,0)_2$ must come after the $W(X,1)_1$, which means that the response isn't correct.

3.3.2 Upper bounds

They give two algorithms that give atomicity. Both have d as the sum of the worst-case latencies for reads and writes.

One puts the burden entirely on the reads, the other on the writes.

Algorithm for fast reads:
Similar to the approximate-clocks case, but don't use ABCast.
Each process keeps a copy of all objects.
A read returns the value of the local copy immediately.
When a write request arrives at i, process i sends it to everyone else, waits time d, then performs the write and acks.
When anyone receives a write message, just perform the write (break ties with process identifiers).

Gives sequential consistency based on times of the invocations, breaking ties with process ids.
Reads take time 0, writes time d.

Algorithm for fast writes:
Each process keeps a copy of all objects.
A write request triggers messages to all processes (including oneself) and an immediate ack.
When a write message arrives, change the value in local memory. When a read request occurs, wait for time d and read the value then in local memory.

All these algorithms seem to work the same for read/update, a richer object type than read/write.

Later results by others (Mavronicolas, Roth) have shown that these algorithms can be modified to yield any combination of upper bounds r and w for reads and writes, such that $r + w = d$.
They also produce algorithms to show that the lower bounds (based on u) given for the inexact clocks are tight (are these exact bounds, or just $\Omega(u)$ bounds)?

3.4 Discussion

The lower bounds are cute, don't seem surprising.
The algorithms have some interesting ideas:
Using ABCast for writes only to get sequential consistency rather than atomicity. (Lamport produced total order for all operations.)
Pipelining writes to get fast response to writes at a cost of delaying reads.

The subtle distinctions between atomicity and sequential consistency are good to know about—though the bounds here aren't high enough to demonstate that this distinction matters in terms of practical performance. (The lower bounds for atomicity aren't very high.)

I'm not convinced that atomicity is impractical.
Or that, if it has to be weakened for practical considerations, that sequential consistency is exactly the right notion.

4 Eventual Consistency

4.1 Ladin, Liskov, Shrira, Ghemawat

This paper defines a new, weaker consistency condition, and an algorithm that achieves it.
But (warning) this is not an algorithms paper, but a systems paper.
It does not try to present the specifications and algorithms precisely, nor to generalize and abstract, and decompose as much as possible.
Rather, it describes a particular implementation, with many kinds of theoretical and engineering issues mixed together.
We can try to pick them apart. The paper by Fekete et al. is a decent attempt to do this, though that doesn't seem so easy to read.

The new specification for memory consistency is given on p. 364, where the API is given, and p. 365, Figure 1, where some of the correctness conditions to be satisfied by the inputs and outputs are presented.

The inputs are invocations and the outputs are responses, as usual.
The kinds of operations considered are "update" and "query".
An update, as before, can modify the state as a function of the previous state, but does not return a response.
A query does not modify the state, but returns a function of the state (essentially a read).

Notice the extra stuff in the API: prev, uid, newl.
Every update is supposed to get assigned a uid by the system, which is returned to the user in the response to the update.
When a user then submits any operation (update or query), he is free to include a "label", which represents a set of uids.
This "prev set" represents the set of updates that are required to "precede" the operation, that is, the new operation is supposed to be ordered after all these updates in any ordering created by the system.

The query returns another label—the set of uids that the query actually used in determining its response. The query invocatio specified a prev set of updates that the user said it had to depend on. This response set must include those, but may include more. This response is simply informing the user about what was used.

The service guarantees, in Figure 1, say:
1. The service can use only updates that were actually requested by the users.
2. The set of updates returned by a query (newl) includes the set of updates requested by the user (prev).
3. The set of updates returned by a query (newl) must be closed under the "dependency" relation: update u depends on update v proviced that v is in u's prev set.
4. The value returned by a query is consistent with the newl set of updates, applied in some order that is consistent with the dependency order.
(It seems that this order need not be unique. In fact, from what is said here, the resulting value need not even be unique.)

They make some assumptions about the client's use of the labels:
They assume that each client maintains a local label (set of updates), which is what it uses for its prev set for each of its operations.
Then when a response comes back, it takes the union of the response set with its current label set, to get the new label.

This specification seems to maintain a kind of causality.

It's not clear to me exactly what the consistency guarantees mean—how can they be used, how strong are they, what properties do they have w.r.t. composition, etc.
The spec is a little messy-looking (esp. w.r.t. its division of responsibility between system and user), which makes it harder to tell. Is it expressed in the simplest way?

Anyway, they include two special additions to the basic service: two special kinds of updates, which they call "forced" and "immediate".
These have stronger consistency requirements:
Forced updates are said to be "totally ordered with respect to each other".
Immediate updates are ordered with respect to all other updates.
In terms of the spec, who is responsible for ensuring this?
Basically, they seem to be strengthening their dependency order among updates to include the extra requirements, then using the stronger dependency relation in their closure condition (3).

Skip this part for now, just focus on the "causal" spec and its algorithm.

The main algorithm is described on p. 365-371.
But it is described in terms of vector timestamps—a particular way that the sets of updates are represented.
That is, each update is assigned a vector timestamp as a uid.
Each label that is attached to an invocation or response is again a vector timestamp; the set of updates that it represents is (as we have seen before) the set of updates whose vector timestamps are $\leq$ the given vector.
The resulting algorithm description is quite complicated, and mixes the timestamp handling with more fundamental issues involving update dependencies.

However, I think it's useful to describe the algorithm first in terms of sets of updates, then think later about how the actual vector timestamps are used to represent those sets (a lower level of refinement, an optimization).

For this, I switch to the other paper, by Fekete et al.

4.2 Fekete et al.

4.2.1 Overview

This is an attempt to represent the essence of the algorithm in the other paper in a more abstract, more general style.
Also, a student, Oleg Cheiner, implemented this version of the algorithm, and implemented some small demo applications (a name service) on top of it.

This paper first points out a subtle issue with the LLSG paper:
They seem to rely on the clients to guarantee: for any pair of non-commutative updates, one must be specified as depending on the other.
Without this, their algorithm may assign incomparable timestamps to them, which may mean that, forever after the point where both of these updates are known to everyone, no one will know what order to apply them in.
So this order will be forever undetermined, which means that forever, clients may decide arbitrarily what order to use.
This can lead to infinite inconsistencies.

Discussion with the authors suggested that they didn't mean for this to happen—they really did intend that eventually, a total ordering of updates should be produced.

That led us to produce a new version of the spec, that says that eventually such a total ordering should be produced.
After presenting our abstract algorithm, we prove its correctness and analyze its performance.
It's all very carefully done, but I think the ideas are sometimes hard to extract from the formalism.

Most of our paper deals with general operations, not even divided into reads and updates. The optimization to handle reads specially is described later.

4.2.2 Problem specification: ESDS

We consider operations that are "ordinary" and others that are "strict". The "strict" ones have stronger consistency requirements.

Our clients (p. 8) are allowed to request any operation at any time, as long as the prev set describes some previously-requested operations.
We didn't include the assumption that the client must submit, in its prev set, everything it has ever heard about.
That's not needed to prove that the algorithm satisfies the spec.

Our main spec appears on p. 10.

Conceptually, the service is building up a global partial order of operations, *po*, which gravitates eventually to a total order.

The spec is expressed in terms of some abstract internal "milestones".
An operation that is invoked first gets put into a buffer *wait*, and its response comes from another buffer *rept*.
In between, the milestones are: the operation gets "entered" into *po*; at some point it may get "stabilized", when its prefix in the eventual total order is determined; a response value gets "calculated".
For an ordinary operation, the response can be calculated anytime after the operation is entered into the partial order.
For a strict operation, this can occur only after the operation has been stabilized.

4.2.3 Our algorithm

p. 15-19.
It's based on LLSG, but it's simpler because it's more abstract.
It's written in terms of a partial order, of sets of updates, etc.

Our version, like LLSG, is written assuming separate "front end" automata and "replica" automata. The front end automata simply forward requests to replicas and collect return values, however. The interesting work is done by the replicas.

The replica code is on p. 18.
Information is gossiped in the background, vis send and receive actions.

A replica keeps track ($rcvd$) of the set of operations it has learned about.
At some point, it "does" each such operation, which means that it assigns it a locally-generated "label" from a totally-ordered set.
Furthermore, it keeps two vectors, *done* and *stable*.
The *done* vector records its knowledge of the set of operations that it knows have been done at each node.
The *stable* vector contains second-order information: An operation is said to be "stable" if it has been done everywhere.
Process i puts an operation into its own stable set (in $stable[i]_i$) when it is in the intersection of all its $done[j]_i$ sets.
Process i puts an operation into $stable[j]_i$ when it learns, via gossip, that the operation is in $stable[j]_j$.

The local ordering on the operations is captured by the derived variable lc.
This is based completely on the labels for the operations.
An interesting complication: the same operation can be assigned a label at different locations, independently.
The node keeps updating its label to be the minimum one that it has heard for the update.
In this way, the ordering can change over time.

When can the replica send a response?
If the operation is ordinary, this can be anytime after the operation is *done locally.*
The value returned is a value that could be produced by ordering the things in $done[i]_i$ according to the ordering of labels expressed by lc. (This looks unique to me, though it can change over time.)

If the operation is strict, the process waits for the operation to become *stable everywhere*—this implies that the ordering of the prefix, and the value, will never change in the future.

4.2.4 Discussion

Is this really doing the same thing as the LLSG algorithm?

Is the specification, or some variant of it that expresses causal consistency, a useful weakening of atomicity?

Evidence that something like this is useful is the collection of applications/implementations that have been built on top, but I don't think this is well understood from a theoretical viewpoint.

5 Rambo

5.1 Overview

Implements atomic read/write memory in dynamic network setting. Uses replication.
To ensure atomicity in the presence of transient failures, uses read and write quorums as in ABD.
To accommodate long-term changes, allows reconfiguration.
Configuration = set of members, sets of read-quorums and write-quorums.

Configurations can change during computation.
Does not cause violations of atomicity. Any quorum installed at any time (compare with Yeger-Lotem, De Prisco,...)

Tolerates process stopping failures, message loss, timing anomalies.

Performs three activities, all concurrently: Reading and writing objects; choosing new configs and notifing members; identifying and removing old configurations.

Algorithm splits into main algorithm and reconfiguration algorithm.

Atomicity always, conditional performance properties based on timing and failure assumptions.
Main performance result: $8d$ bound for reads and writes in "normal case", even while limited failures continue to happen and reconfiguration is going on.

5.2 From DISC talk

Intro, as is.
Specs: Refer to details on p. 6-8 if necessary.

Algorithm:
Skip joiner part. Describe joining protocol briefly: Just tells one process about itself and acquires that process' knowledge. Might be better to get better integrated into the system, e.g., learning about a quorum or becoming known to a quorum. Have not tried to analyze the performance/fault-tolerance impact of different kinds of joining protocols.

Static protocol review.
Data structures: Configuration maps.
Reader-writer state: Go over p. 11.

Go over operation of algorithm, from slides and from code p. 14 and 16.

Proof: First from slides, and look at details from paper:
History variables p. 18
Phase guarantees, p. 22-24

Behavior of gc:
GC sequential: Lemma 6.6, p. 24
Tags propagate: Lemma 6.7, p. 25

Read/write vs. gc:
No prior gc implies include some config that's leq: Lemma 6.8, p. 25
Prior gc implies include some config that's geq: Lemma 6.9, p. 26
Tags propagate from gc to a following read/write: Lemma 6.11, p. 26
Sequential r/w: Lemmas 6.12, 6.13, 6.14, p. 27-29

Implementing Recon

Latency: From slides. Describe why the analysis is hard. Emphasize independence of the consensus bound.

Normal case; beginning of work on normal from some point onward.

Conclusions: From slides.

6 Improvements to Rambo: concurrent handling of obsolete configurations

Seth Gilbert