Disk Paxos

Afek, Greenberg, Merritt, Taubenfeld, Computing with Faulty Shared Objects
Jayanti, Chandra, Toueg, Fault-Tolerant Wait-Free Shared Objects
*** Gafni, Lamport. Disk Paxos
*** Chockler, Malkhi. Active Disk Paxos with Infinitely Many Processes

1 Computing with Faulty Objects

1.1 Introduction

Last time, we were working on computing with faulty objects, from the Afek, Greenberg, Merritt, and Taubenfeld paper.
This paper contains results saying that various kinds of wait-free objects can be implemented using faulty low-level objects.
We did a composition result, a result saying that reliable atomic r/w registers can be built from faulty atomic r/w registers, and one saying the same about test-and-set registers.
Actually, the test-and-set result was given for 2 processes only, though the paper extends this to n processes. I'll say a bit more to finish this up, below.

Two results weren't covered: one showing how to implement read-modify-write registers in terms of faulty RMW registers, and one giving consequences for universality constructions. Will describe.

I'll skip the Jayanti results, include the notes here for later reference.
Then go on to the main topic, Disk Paxos.

1.2 AGMT

1.2.1 Test-and-set constructions

Last time, we showed how to construct a reliable "single-use test-and-set" object from unreliable SUT&S objects, where at most 1 may be unreliable. p. 15.
To finish this, we should take a quick look at what you need to do to make this usable by n processes.
In both cases, they tolerate just 1 fault, and then use the boosting idea to increase the fault-tolerance.

SUT&S primitive, sequential specification:
Values $0, 1$ initially 0. One operation, T&S.
Any operation writes 1 into the register, returns the value the register previously had.
Thus, the first process gets 0 and everyone else gets 1.
Atomic version of this—each operation should be serializable at some point in its interval.

Recall 2-process algorithm.

Can extend this to an n-process algorithm still tolerating 1 fault, by adding a doorway that allows at least one and at most two processes though.
Basically, to enter the main algorithm, you have to win some register at each of two preliminary stages.

At most two processes can do this: At least one of the stages has both registers nonfaulty, and each process that wins must grab at least one of these registers.

At least one process will win one at both stages: At most one of the four registers is faulty.
Thus, some process wins one of the registers at stage one and enters stage two.
Then some process wins one of the registers at stage two.

The n-process algorithm can be boosted to an algorithm for T&S from T&S that tolerates any number of failures.

1.2.2 RMR register constructions

Section 6, Theorem 12.
Consensus from RMW, using only $f + 1$ RMW registers to tolerate f faults.
Also uses a quadratic number of atomic binary r/w registers.
The f limitation applies to all the low-level registers—both the RMW and the r/w registers, though that seems too pessimistic.

See code p. 29.

The algorithm operates in $f + 1$ identical stages.
Each stage has 1 associated RMW register and $2f + 1$ v bits.
The actual value is stored in the RMW register, and the r/w registers are used for validation.

For each stage, the process first sets all the validation flags (for that stage) to its current estimate x.

Then it atomically accesses the RMW register for that stage, seeing if it has already been set by anyone.
If not, it sets it to its own value x.
In either case, it remembers the final value of this register in a temporary variable.
Since this is a RMW variable, it can do all this in one step.

Next, it checks whether "enough" $(f + 1)$ of the validation bits for this stage, for this temporary value, have already been set.
If so, it uses this as its new value for the next stage. Otherwise, it doesn't change.

At the end of $f + 1$ stages, just output the final value.

Correctness:
Validity: Claim (stronger) that the value of anyone's x is always a valid value.

Because no one adopts a new value without having read it in $f + 1$ registers for a stage.
If it appears there, at least one register was written by a nonfaulty process.
(Can turn this into an induction.)

Agreement: At least one stage, say g, has no faulty variables (RMW or bits).
At that stage, the first process to attempt a RMW succeeds in writing its value, say x_g.
Has already written this value to the validity bits for this stage.
They persist (all nonfaulty), so all other processes can confirm the validity.
Every other process will also read x_g as the value of this RMW variable, and will therefore adopt the value x_g at stage g.

Claim they all keep this value for the remaining stages.
E.g., consider stage $g+1$. All processes set validity bits for x_g at that stage, and none for the other value. Then no one will see enough evidence to switch its value.

(The proof seems to have some slack—I think it shows that f faulty RMW registers and f faulty r/w registers can be tolerated, not just a total of f of both kinds.)

1.2.3 Universality

AGMT discuss universality results for the new setting, with faulty objects.
Recall Herlihy defined the notion of a "universal object type", which allows construction of a wait-free implementation of any other object type.
Herlihy assumes the basic objects are nonfaulty.
Example universal objects: RMW, consensus.
(The construction for consensus uses some atomic r/w registers also.)

AGMT extend this result to the case where some of the base objects may be faulty, in Theorem 18.

Part 1: RMW registers can be used to construct any object type, even if f may be faulty.

Part 2: Consensus objects plus r/w registers (they use safe, I'll say atomic) are universal, even if some (f out of all the objects, both kinds) may be faulty.

Proof: Part 1: Use Herlihy's non-fault-tolerant universality construction, which implements any atomic object using nonfaulty consensus objects and nonfaulty r/w registers.
Then implement the consensus objects in terms of possibly-faulty RMW objects, using the fault-tolerant implementation of consensus we just described (Theorem 12).
(This uses possibly-faulty RMW registers and r/w registers, but of course we can use RMW registers in place of r/w registers. AGMT use a variant, which uses only RMW registers, $2f + 1$ of them).
Also, implement the r/w registers using possibly-faulty RMW registers (Corollary 8).
Combine everything, using composition theorems to express the fault-tolerance (Theorems 3, 4).

Part 2: Use Part 1 to get a fault-tolerant implementation of the arbitrary object in terms of RMW objects.
Now use Herlihy's non-fault-tolerant universality construction to implement the RMW objects in

terms of consensus objects and atomic r/w registers.
Combine using a composition theorem (Theorem 4).
(If the top level tolerates f failures, then f failures among the lowest-level objects can kill only f of the intermediate objects—even if the lower-level implementation isn't fault-tolerant. The top level can tolerate this.)

1.2.4 Significance

The model seems similar to the Byzantine quorums work, and the basic strategies are similar.
The main idea is that you can tolerate a limited number of faulty objects, in shared-memory algorithms. The basic technique is to use enough redundancy to "swamp out" the bad results.

This only works in a kind of "synchronous" model, in which objects always respond.

1.3 Jayanti, Chandra, and Toueg

1.3.1 Overview

Another paper on fault-tolerant shared objects, with many of the same results as AGMT.
Main differences: They consider other failure modes, like "nonresponsive" failures, and a "graceful degradation" property.
The main results about nonresponsive failures are impossibility results based on FLP and Abu-Amara impossibility results.

They also consider better behaved kinds of failures for objects, e.g., a form of crash failure, where an object just reverts at some point to responding with a known $\perp$ response, which it will use consistently from that point onward.
And omission failures, where the object can mix up $\perp$ and correct responses, but not actually give wrong responses.
I'm not covering the non-responsive and graceful degradation results. I'll focus on Section 5.1, p. 467-475.
Two results:

1. Theorem 5.1.2.1. An implementation of reliable consensus from consensus (only), tolerating f faults, using only $f + 1$ base objects. But, the faults are limited to crash (also omission).
2. Theorem 5.1.3.1. Another implementation of reliable consensus from consensus (only), this time tolerating f Byzantine faulty objects.

1.3.2 Crash-tolerant algorithm

Code, p. 469

Use $f + 1$ consensus objects.
Start with your input as your "estimate".
Go through the sequence of base consensus objects in order.
For each, propose your latest estimate.
If the response you get back isn't the default $\perp$, change your estimate to the new response.

After $f+1$ stages, return your final estimate.

Similar to the AGMT multi-stage algorithm, but now we don't need to use validity bits—we can just accept the value as correct.

Validity is obvious.
Agreement: One of the base consensus objects must be nonfaulty, so it gives the same response to everyone. Then no one ever submits a different value to later objects.

1.3.3 Byzantine-tolerant algorithm

Code p. 471, picture p. 472.

Now consensus from (just) consensus, with Byzantine failures.
Has a lot of similarity to consensus from RMW, in AGMT paper.

Construction is recursive; what's shown here is the top level.
To tolerate f faulty base objects, use two intermediate consensus objects O_1 and O_2 that tolerate approximately $f/2$ faulty base objects in their implementations.
Also use three arrays of consensus objects, A_1 and A_2 with $3f+1$ objects each, and B with $4f+1$ objects.
None of these objects tolerate any failures.

A process with initial value 0 follows the arrows in the diagram, going through A_0, then O_1, then B, then A_1, then O_2.
One with initial value 1 would follow the arrows "symmetrically", starting with A_1, then O_1, B, A_0, Q_2.

E.g., if initial value 0:

A_0: Go through the entire array, proposing your initial value and counting (in $count[0]$) the number of times the response value agrees with your initial value.

O_1: Propose your initial value to O_1 and record the response, in ans_1.

B: Go through the entire array, this time proposing your ans_1 value, and counting (in $witness-count[0]$ and $witness-count[1]$) the number of responses you get for each of the two values.

A_1: Now go through the entire array, again proposing your initial value. This time, count (in $count[1]$) the number of times the response value *disagrees* with your initial value.

At this point, you might have enough information to return a value.
Choose $belief$ to be the value with the larger $witness-count$.
If the $witness-count[belief]$ is sufficiently large $(3f+1)$, and your $count[belief]$ is also large $(2f+1)$ then simply return $belief$.

Otherwise, rely on the final consensus object, O_2.
Propose to O_2 either your $belief$ (if $witness-count[belief] \geq 2f+1$ and $count[belief] \geq f+1$),

or else your initial value.
Whatever O_2 returns is your final decision.

Proof:

Lemma 1: If O_1 is nonfaulty and returns 0 to any process p, then $count[0]_p \geq 2f+1$.
(And analogously for 1.)

Proof: Since O_1 returns 0 to p, it must be (by validity) that some process must have previously proposed 0 to O_1; let q be the first.
Then before q proposes 0 to O_1:
(1) q must have submitted its 0 to all the consensus objects in array A_0.
(2) no process submitted 1 to any of the consensus objects in A_0 (because they haven't heard from O_1 yet).
(3) all the A_0 objects have returned decisions to q.
Therefore, all the nonfaulty A_0 objects must have returned 0 decisions.
Therefore, any process that reads A_0 sees at least $2f+1$ 0's.

Lemma 2: O_1 and O_2 can't both be faulty.
Because there aren't enough faults in the base objects (of the recursively expanded implementation) to kill both intermediate objects.

Main proof, agreement and validity: Use cases, first assuming that O_1 is nonfaulty and then O_2.

Case 1: O_1 nonfaulty:
Then O_1 gives out the same v to every process.
So every process proposes v to all the B objects, and all the nonfaulty B objects return v.
That gives everyone a high enough $witness - count[v]$.
They also have high enough $count[v]$, by Lemma 1.
So they all decide v.

Case 2: O_2 is nonfaulty:
If someone returns his $belief$ on line 14, then there is a clear enough majority for this value v that everyone will satisfy the condition in line 15 and submit the same value v to the final consensus object O_2.
Since O_2 is nonfaulty, it returns v, so everyone agrees.

On the other hand, if no one returns his $belief$ on line 14, then all return the results of consensus using O_2, which must agree.
Validity also follows, because validity of O_2 implies that the value v was submitted by someone.
It was either that process' initial value, which immediately gives validity, or that process belief, then the condition on line 15 says that the count for belief was sufficiently high that someone must have originally submitted it.

2 Disk Paxos

2.1 Introduction and System Model

Consensus is a central problem in distributed computing.

Used in practical settings like state-machine replication, atomic broadcasts, etc.

Disk Paxos is consensus where processes communicate by reading/writing disks.

Consensus problem: Processes start with an input, and produce an output, subject to:

1. *Validity:* Any output is the input of some process.
2. *Agreement:* All outputs are the same.
3. *Liveness:* Under "some conditions", all processes must produce an output.

Disks in DPaxos are single-writer/multiple-reader. Support only read/write operations.

Processes send operations to disks over a network. The network is *reliable*, *FIFO-ordered*, and *asynchronous*. Operations that succeed get acknowledgement.

DPaxos guarantees validity and agreement always. It guarantees liveness if < majority of disks fail, and if network is "stable".

2.2 Disk Paxos Overview

Basic idea of DPaxos (and all Paxos variants) is for *leader* process to get other processes to agree to a value.

Example: If all processes knew the leader, and leader doesn't fail, consensus is easy. Just wait for the leader to write a value and accept that value.

If there are multiple leaders or leaders fail, then conflicts can occur.

DPaxos assumes an outside "leader election service". DPaxos guarantees liveness if service produces unique leader forever, it doesn't fail, and majority of disks don't fail.

2.2.1 Leader Algorithm

Leader runs a 2 phase algorithm.

1. *Phase 1:* Pick a value.
2. *Phase 2:* Make other processes accept the value.
3. *Optional phase 3:* Tell other processes the value.

Leader must pick value which doesn't differ from previously committed value.

Leaders distinguished by *ballot numbers.* Higher ballots cause lower ballots to abort.

2.3 Disk Paxos in Detail

2.3.1 Data Structures

Process p has $dblock[p]$ with 3 components:

1. $mbal$ A current ballot number. (initially any ballot)
2. bal The largest ballot for which p entered phase 2. (initially 0)
3. inp The value p tried to commit in ballot bal. (initially $NotAnInput$)

Disk d has a block $disk[d][p]$ for each process p. Each $disk[d][p]$ has the same components as $dblock[*]$. It's writable only by d, and readable by all processes.

2.3.2 Algorithm for Process p

Phase 1.1: For a *majority* of the disks d, write $dblock[p]$ to $disk[d][p]$, then read $disk[d][q]$ for all processes q.

Phase 1.2: If any block read has $mbal$ higher than $dblock[p].mbal$, abort and restart phase 1 with higher $mbal$.

Phase 1.3: Set $dblock[p].bal$ to $dblock[p].mbal$.

Phase 1.4: let $nonInitBlks$ be the blocks read in phase 1.1 with $inp \neq NotAnInput$.

1. If $nonInitBlks = \emptyset$, set $dblock[p].inp = input[p]$.
2. Otherwise, pick a block in $nonInitBlks$ with the maximal value of $bk.bal$, and set $dblock[p].inp = bk.inp$ for that block bk.

Phase 2.1: Same as phase 1.1.

Phase 2.2: Same as phase 1.2, except if no block read has higher $mbal$, commit to $dblock[p].inp$.

Recovery Procedure: Read $disk[d][p]$ for a majority of disks d, and set $dblock[p]$ to be a block with the maximal value of $mbal$.

2.4 Correctness

We prove the easy properties first:

Lemma 2.1 (Validity) Any value that aporcess commits to is the input of some process.

Proof. From the Gafni/Lamport paper. The val field of any disk block is always set either to the val field of some other block or to $input[p]$ for some processor p. ■

Lemma 2.2 (Liveness) If after some time, one process is the unique leader forever, and if less than a majority of the disks fail, then eventually every process commits to a value.

Proof. Consider the time after which there's a unique leader, say p. Then since p always retries a higher ballot, eventually it will use a ballot high enough to not abort in either phase 1 or 2, and will commit a value. ∎

Agreement is harder to show. Gafni/Lamport proof is long and hard to follow. Following proof is (hopefully) simpler. But first some lemmas. (For clarity, we refer to ballots run by processes as *mballots*, *i.e.* the value of *mbal* during that ballot.)

Lemma 2.3 Let d be a disk, p be a process and $b > 0$ be a ballot number. If $disk[d][p].bal = b$ in some mballot m of p, then p changed the value of $dblock[p].inp$ in mballot b.

Proof. $dblock[p].bal$ must have been set to b at the end of phase 1 of mballot b. After this, $disk[d][p].bal$ could only have been set to b during phase 2 of mballot b, or during a later mballot. Before either of these events can happen, p must have first changed the value of $dblock[p].inp$. ∎

Lemma 2.4 Let p be a process which committed in some mballot m, and let q be a process which finished phase 1 in some mballot $m' > m$. Let d be a disk accessed both by p in phase 2 of m, and by q in phase 1 of m'. Then the ordering of p and q's operations at d is as shown in Figure 1.

p writes $disk[d][p]$ in phase 2 of m
p reads $disk[d][q]$ in phase 2 of m
q writes $disk[d][q]$ in phase 1 of m'
q reads $disk[d][p]$ in phase 1 of m'

Figure 1: Ordering of p and q's operations.

Proof. d exists because both p and q accessed a majority of the disks. We argue that p must have read $disk[d][q]$ in phase 2 of m before q wrote to $disk[d][q]$ in phase 1 of m'. If not, then since $m' > m$, p would have read $disk[d][q].mbal = m' > dblock[p].mbal = m$, and aborted mballot m, a contradiction. Rest of ordering follows since p writes $disk[d][p]$ before it reads $disk[d][q]$, and since q reads $disk[d][p]$ after it writes $disk[d][q]$. ∎

Lemma 2.5 (Agreement) All processes which produce an output produce the same output.

Proof. Consider execution α, and let m_0 be the smallest mballot that commits in α. Let the value committed be v, and the process running m_0 be p.

Let $M = \{m \mid m > m_0\}$ be the set of mballots in α greater than m_0. Need to show that any mballot in M which committed committed to v.

WLOG, assume that every mballot in M set $dblock[*].inp$ at end of phase 1.

We order M as $m_1, m_2, \ldots$, based on the *time* in α when an mballot sets $dblock[*].inp$ at the end of its phase 1. *I.e.*, m_1 is the earliest mballot in M to set $dblock[*].inp$ during its phase 1, m_2 is the second earliest, etc. For each $m_i \in M$, let q_i be the process which ran m_i. We use induction on the ordering of M.

Consider base case m_1. Let d_1 be a disk which both p accessed in phase 2 of m_0, and q_1 accessed in phase 1 of m_1.

Consider the point in m_1 when q_1 finished reading and writing to a majority of the disks, and before q_1 set $dblock[q_1].inp$.

By Lemma 2.4, q_1 read $disk[d_1][p]$ after p wrote to $disk[d_1][p]$, so q_1 found $disk[d_1][p].inp \neq NotAnInput$, and $p \in q_1.nonInitBlks$, and q_1 doesn't try to commit to its own input.

Claim 2.6 Let b be the maximal value of $bk.bal$ for all $bk \in q_1.nonInitBlks$. Then $\forall bk \in q_1.nonInitBlks : (bk.bal = b) \Rightarrow (bk.val = v)$.

Proof. Pick a bk with $bk.val = b$, say $bk = disk[d][q]$. Then $disk[d][q_1].bal \leq m_0$, since otherwise, by Lemma 2.3, q_1 must have changed $dblock[q_1].inp$ in mballot $b > m_0$, contradicting our choice of q_1 as the first to change $dblock[*].inp$. Also, $b \geq m_0$, since q_1 reads $disk[d_1][p].bal \geq m_0$ by Lemma 2.4. Remains to show if $\forall bk \in q_1.nonInitBlks : (bk.bal = m_0) \Rightarrow (bk.val = v)$. This is easy if p never fails. Also easy to show if p did fail and recover (induct on number of recoveries). ■

So q_1 sets $dblock[q_1].inp = v$, and can't commit any value other than v.

Inductive step for process q_i is similar. Let b be max $bk.bal$ q_i read in phase 1, such that $bk.inp \neq NotAnInput$. Then $bk.bal \geq m_0$ by Lemma 2.4. Also, if $b = disk[d][q]$ for some d, q, then $q \in \{p, q_1, \ldots, q_{i-1}\}$, because by Lemma 2.3, q must have set $dblock[q].inp$ before q read $disk[d][q]$. By induction, $b.inp = v$, and so q_i sets $dblock[q_i].inp = v$, and we're done. ■

3 Active Disk Paxos

3.1 Introduction

Disk Paxos for an unbounded number of processes.

Motivated by development of storage area networks (SANs) using active disks, along with the two impossibility results shown below. Active disks are disks that support RMW operations.

Original disk paxos relied on specific locations on disk for each process, so adding/removing members was expensive.

3.1.1 Impossibility results

- Can't have: a wait-free implementation of a reliable RMW object from a collection of faulty RMW objects. This is from Jayanti, etc. al. of last week, a reduction to FLP.
- Can't have: Ranked register (see below) implemented by a collection of finite RW memory. They prove this one by tieing into mutual exclusion.

Thus, they use special RMW-like objects called ranked registers to do Paxos for an unbounded number of processes. Claim: active disks can implement ranked registers.

3.1.2 Model

- Stopping failures for processes.
- Non responsive stopping failures for objects.
- No definition of infinite processes. They mean unbounded?

3.1.3 Results summary

Ranked registers explained in the next subsection.

- One ranked register is sufficient for non-fault tolerant consensus with unlimited participants. Not wait free.

  Make wait-free by using an oracle leader election service.
- Constructing fault tolerant register that's wait free, from faulty ranked registers. This is possible because consensus is not wait free from ranked registers.
- Consensus with multiple ranked regsiters that are unreliable. So this is always fault tolerant, but not wait free (use leader election).

So ranked registers are of *intermediate* strength.

3.1.4 Ranked registers

Like ballot+value in Disk Paxos, where a rank is a ballot id. Supports a read and a write:

`Read(r:rank) -> (r':rank, v':value) |` $\perp$: Reads can reply with value or nil.

`Write(r:rank, v:value) -> commit | abort`: Writes reply with commit or abort.

Properties:

Safety A read operation returns a value written by some write operation, and sees all write operations before it in rank. This means a write operation aborts if it is ranked before an invoked read operation.

Non triviality If a write operation aborts, then a read operation of higher rank commits beforehand or concurrently.

Liveness eventual return of invokation.

Note that lower ranked writes are allowed with respect to other writes. Assumption: Ranks for ranked registers unique (like unique ballots).

3.2 Ranked registers for consensus: ranked Paxos

See algorithm figure 1.

Proof that the algorithm implements consensus.

Let W_0 be the first commit in time with value $\langle w_0, v_0\rangle$. Make a chain of $W_1, W_2, W_3 ... W_i$ that are successful writes after W_0. We argue by induction on the chain by saying that each $W_i = \langle r_i, v_i\rangle$ has to equal W0 in value. Each write also does a read R_i, according to the algorithm.

```
[Time is downwards here]

  W0 = r0, v0

  R1  = r1, v1 (reply)
```

```
W1  = r1, v1 (invocation)

Ri  = ri, vi (reply)
Wi  = ri, vi (invocation)
```

By register safety property, $r_i > r_0$. Otherwise W_i wouldn't happen.

Now prove by induction: v returns v_0 or some value after.

`Basis step`: v_1 of has to equal v_0 since v_0 is the value associated with r_0 the highest rank in the register when R_1 happens[1].

`Inductive step`: v_i is either v_0 (same argument as above) or v_j where $j < i$. But by inductive assumption, all v_j have the same value.

Thus, ranked register Paxos implements safety properties of consensus. Validity: registers assigned values only from *inp*. Liveness: when there is one leader, leader chooses to write to a high rank and finishes in 2 rounds.

3.3 Multiple faulty ranked registers implement reliable ranked register

See algorithm in figure 4. Terminology: Write/Read/Abort are high level, and write/read/abort are low level equivalents. Proof that n registers implement a ranked register if $n > 2f + 1$.

Liveness Just need to read majority, which will be unfaulty. Returns because of lower liveness.

Non triviality If there is an Abort, then there must be an abort. At that register, there must be or have been a read with higher rank. This read could only have come from a Read. Thus the Abort is concurrent or after a Read of higher rank.

Safety A Read sees every previous Write of lower rank. Each Read and Write must read and write at a common register (because majorities intersect). Assume that there's a Read of higher rank than a Write. If the Read finished before the Write then the read finished before the write, and the Write will return abort because the write will return abort. If the Write finished before the Read then the write finished before the read, and the read will reflect the write, and so will the Read (or a later Write, so we're okay). If the Read and Write were concurrent, then we simply choose to serialize the Read and Write by the order of the read and write.

3.4 Multiple faulty ranked registers implement consensus

Use faulty registers to create wait-free reliable ranked register. Then use reliable ranked register in register Paxos to have non wait-free consensus.

3.5 Other notes

Universality: ranked registers are universal except that using them won't be wait free, since the algorithm for implementing an arbitrary atomic object involves leader election.

[1] Why is r_0 the highest rank? Because otherwise W_0 wouldn't have happened