Byzantine Quorum Systems

1 Malkhi-Reiter Byzantine quorum systems paper

1.1 Overview

Goal: highly available and efficient data replication

Assume:

Client (proxies) and server model

Universe of data servers

Quorums: collection of subsets of that universe, each pair of which intersect.

Read / Write from quorums, ensuring fault-tolerance and reduced load (not having to contact all the possible replicas in all operations). Use timestamps for ordering.

Interface of the system. (Not clearly defined in the paper.)

Example (benign failures).

Things get worse with Byzantine failures. Most of us are probably familiar with the term, but to recap Byzantine failures means that faulty components may behave in arbitrary ways. This includes the possibility of a malicious attacker compromising one or more replicas of these data servers. And we wish to continue to provide correct service despite that.

This paper deals with how to extend this simple quorum construct to support arbitrary (Byzantine) failure of a subset of the data servers.

Main results:

- Define how to construct quorum systems that tolerate Byzantine failures of a subset of the servers in the universe.

- Explore the load of their quorums systems (access probability of the busiest server).

- Do this for different variations of the data being stored, etc...

1.2 System model and data semantics

Universe U of servers, $|U| = n$.

Quorum system $Q \subseteq 2^U$. Every pair of subsets must intersect, but there are further restrictions on how they intersect, depending on type of guarantees.

Fail-prone system $\mathcal{B} \subseteq 2^U$. Non-empty set of subsets, none of which is contained in another, such that some $b \in \mathcal{B}$ contains all the faulty servers. Typically expresses that up to a threshold f of servers may fail:

$\mathcal{B} = \{b \subseteq U : |b| = f\}$

Clients all "know" $\mathcal{B}$

semantics provided - "safe variable". Desribed in L. Lamport's paper: "A read that is not concurrent with any writes must return the latests value that was written."

(recall for a "regular variable", a read must return the latest value that was written or the value of a write concurrent with the read. For an "atomic variable", the reads and writes have to be consistent with some global linear order)

note we still have to define start / end of the operations.

This is rather weak semantics, since any arbitrary value can be returned when the read is concurrent with writes. But they actually provide slightly stronger semantics. If the read is concurrent with writes, it either outputs the last write (before the read and the concurrent writes), one of the values being written, or null if no assurances can be made about the value to output.

1.3 Protocol

1.3.1 Basic idea

read and write operations are basically as follows:

Write:

Client c queries servers to obtain a set of timestamps $A = \{< t_u >\}_{u \in Q}$ for some quorum Q; chooses a timestamp greater than every timestamp in A, and greater than any timestamp it has chosen in the past; and sends the update $< v, t >$ to servers until it has received an ack for this update from every server in a quorum Q'. Write ends when all servers in a quorum get the update.

Note timestamps cannot be the same for different clients (by appending id or address to low-order timestamp).

Read:

Query servers in a quorum to obtain a set of value / timestamp pairs. We will see what are the thresholds to accept a value as the output.

How to define start / end of the operations?

read begins when client initiates operation.
read ends when client determines read result.
write begins when client initiates operation.
write ends when all correct servers in a quorum receive the update.

1.3.2 Defining quorums

Definition (masking quorum system):

the basic idea is that clients write to quorums and read from (possibly distinct) quorums.

So suppose we write to quorum Q1, and after the write finishes, read from Q2.

Given that B is that set of faulty servers, we are going to obtain the following replies to the read:

- Correct replies from servers in $(Q_1 \cap Q_2) \setminus B$.

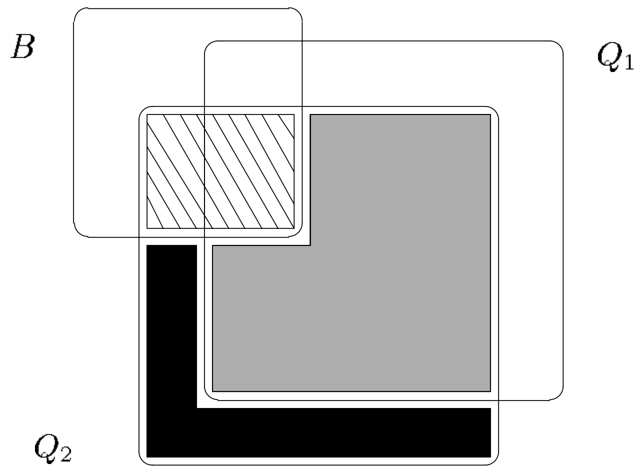

Figure 1: Reading from a masking quorum Q_2.

- out-of-date replies from $Q_2 \setminus (Q_1 \cup B)$.

- wrong replies from $Q_2 \cap B$.

So, in order to obtain a correct reply, the set that is chosen to be identified as the correct set, cannot be formed exclusively by bad guys (otherwise this could be a forged value), i.e.,
$(Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$, for any $B_1, B_2 \in \mathcal{B}$. This is called *M-consistency.*

There is another aspect of this definition: since faulty servers may never reply, and we need to hear from a quorum before the operation is concluded, we need to have at least one quorum that does not contain faulty servers. (Remember communication is asynchronous so we cannot detect failures based on non-responsiveness.) This yields the following definition of quorums:

$\forall Q_1, Q_2 \in Q, B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \backslash B_1 \not\subseteq B_2$

$\forall B \in \mathcal{B}\ \exists Q \in Q : (B \cap Q) = \oslash$

In the specific example of having a threshold of f faulty servers, an "f-masking quorum system" is a quorum system Q s.t.

$\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq 2f + 1$, or
$|Q| \geq (n + 2f + 1)/2$.

[Picture with 4f+1 total servers, quorums of 3f+1.]

1.3.3 Defining operations

So now we've defined quorums, how exactly do we read and write from a variable?

We've defined writes precisely. To read, the intuition is that we discard all values that were returned from sets in B (or subsets thereof) and from the remaining values choose the one with the highest timestamp.

Read:

For a client to read x, it queries servers to obtain a set of value / timestamp pairs $A = \{< v_u, t_u > \}_{u \in Q}$ for some quorum Q. Then it partitions A in sets of replies with the same value/timestamp pair, and discard all sets that could be exclusivly formed by faulty servers — subsets or sets in B. From the remaining (if any) it chooses the set with the highest timestamp.

Parallel of reading and writing to 3f+1 out of 4f+1. When reading, choose the most recent value that was output by more than f servers.

Advert possibility of null result.

1.3.4 Proof - safe variable semantics

So now we want to prove that the constuct above yields safe variable semantics: A read that is not concurrent with any writes must return the latest value that was written in some serializations of all preceding write operations.

Pf outline:
1. W = set of writes preceding read
2. $< v, t >$ write with highest timestamp in W
3. it will be the outcome of the read since...
3.1. will appear in Q_{read} by intersection property of quorums, and
3.2. higher timestamps will only come from f faulty servers (not enough)
4. $< v, t >$ is latest write in some serialization of W.

Pf: Let W be the set of writes preceding the read and $< v, t >$ be the one with highest timestamp. This value will be returned by the read since by the construction of masking quorum systems, this value/timestamp pair will appear in A' and will have the highest timestamp in A' (any pair with a higher timestamp will be returned only by servers in some $B_1 \in B$). So, it suffices to argue that there is a serialization of the writes in W in which this write operation appears last, or in other words, that this write operation precedes no other write operation in W . This is immediate, however, as if it did precede another write operation in W that write operation would have a higher timestamp.

1.4 Example quorum constructs

- "f-masking"

- Construct grid quorums that achieve this lower bound: arrange servers in grid. Quorums consist of 1 column and $2f + 1$ rows. Easy to see that every pair of quorums intersect in at least $2f + 1$ servers. Is this really necessary? Maybe $f + 1$ suffice...

1.5 Variation: self-certifying data

The problem is simplified if the data is self-certifying (e.g., signed) since if a malicious server forges arbitrary answers, they will be detected. Still, the adversary can provide the client stale data, instead of the latest write. Therefore, we need to change the quorum definition (the intersection requirements) so that at least one good replica in the read quorum saw the latest write:

$\forall Q_1, Q_2 \in \mathcal{Q}, B_1 \in \mathcal{B} : (Q_1 \cap Q_2) \not\subseteq B_1$

The availability condition is the same:

$\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : (B \cap Q) = \emptyset$

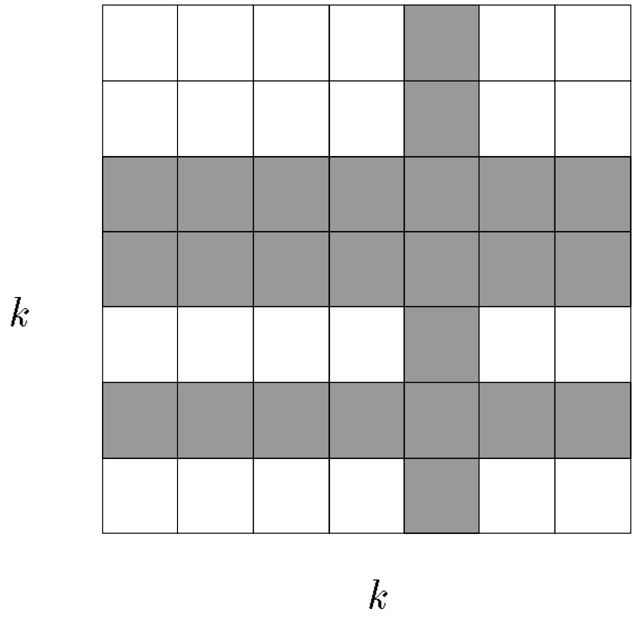

Figure 2: Grid construction, $k \times k = n$, $f = 1$ (one quorum shaded).

The read operation consists now of reading from a quorum, discard the replies that are not correct (i.e., the signature does not match the contents) and pick the one with the highest timestamp.

An interesting result is that if we consider the usual threshold quorum systems, all the results from the masking quorums are still held, but we lose an f. For instance, the minumum number of servers is now $3f$, and with the minimal number of replicas $(3f + 1)$ the quorums are now of $2f + 1$. For a generic n, the quorum size is $|Q| \geq (n + f + 1)/2$.

1.6 Faulty clients

The problem is that a malicious client might send different updates to distinct servers, or fail to contact a quorum.

They change the protocol to provide single-writer multi-reader safe variable semantics (ignoring reads by faulty clients).

What does it mean for the algorithm to be correct against faulty clients? One possibility: $\forall \alpha \exists \alpha' : \alpha'$ has no faulty clients, and $\alpha' \sim \alpha$ to all nonfaulty clients (α, α' are executions of the algorithm).

1.6.1 Protocol

Write protocol is now much more complex:

Maybe draw a picture:

- Client chooses timestamp greater than any it has chosen before, chooses a quorum and sends $< update, Q, v, t >$ msgs to all servers in that quorum. (Repeating if necessary after a timeout.)

- Server receiving "correct" update (increasing timestamp, or same timestamp as last message and same value) sends $< echo, Q, v, t >$ to all members of Q.

- Server receiving identical $< echo, Q, v, t >$ from all members of Q sends $< ready, Q, v, t >$ to all members of Q.

[- Server receiving identical $< ready, Q, v, t >$ from all members of a set $B' \not\subseteq B$ sends $< ready, Q, v, t >$ to all members of Q.] *XXX This I can skip, it is just needed for liveness.*

- Server receiving identical $< ready, Q, v, t >$ from all members of a set Q^- , s.t. $Q^- = Q \setminus B'$ for some $B' \in B$, updates its copy of the variable and replies the client with an $< ack, Q, v, t >$.

After this, client waits for matching acks in a quorum.

1.6.2 Proof of correctness (skipping liveness)

We want to prove that a read that is concurrent with no writes returns the last value written (in some serialization of preceding writes).

First, we need to adapt definition of operation precedence to account for faulty clients. So they define "deliver" as follows:

A server delivers an update $< v, t >$ when it received enough $< ready, Q, v, t >$ to update the variable and reply to the client

Now, they can define:

- write begins when the first correct server receives update.
- write ends when all correct servers in a quorum delivered the update.

For proving safety, they prove the following two lemmas:

Lemma 1 (6.3) A correct server delivers $< v, t >$ only if some correct server previously received $< update, Q, v, t >$.

Pf. To deliver $< v, t >$, the server must receive a *ready* msg from at least one correct server (by intersection property of quorums, $Q \setminus B_1 \not\subseteq B_2$). That server, in its turn, must have received *echo* msgs from *all* servers in quorum Q (since it is correct). Since a correct members only sends the *echo* msg after receiving the $< update, Q, v, t >$ msg, this proves the lemma.

Lemma 2 (6.4) If a correct server delivers $< v, t >$ and a correct server delivers $< v', t >$ then $v' = v$.

Pf. As argued before, for a correct server S_1 to deliver $< v, t >$, $< echo, Q, v, t >$ msgs must have been seen by that server for all servers in Q. Similarly for S_2 and $< echo, Q', v', t >$. Since all quorums intersect in at least one non-faulty server, and and any correct server sends $< echo, *, v_0, t >$ for at most one value v_0, then v must be equal to v'.

So now we can prove safe variable semantics.

Pf. outline
1. W = set of write operations preceding the read
2. By lemma 2, the same value corresponds to any timestamp at all correct servers that deliver it.
3. By lemma 1, all writes delivered are in W
4. By the construction of masking quorum systems, the read operation will return the value written in the write operation in W with the highest timestamp.
5. This is in fact the latest write in some serialization of W. Because single writer and servers only echo request if it has increasing timestamp. So any later operation has a higher timestamp.

Pf.

Let W denote the set of write operations preceding the read. Note that by Lemma 2 (6.4), any value/timestamp pair in W is well defined, i.e., the same value corresponds to any timestamp at all correct servers that deliver it. By definition, every write in W was delivered to a full quorum, and by assumption and Lemma 1 (6.3), no correct server has delivered any write outside W. Therefore, by the construction of masking quorum systems, the read operation will return the value written in the write operation in W with the highest timestamp. So, it suffices to argue that there is a serialization of the writes in W in which this write operation appears last, or in other words, that this write operation precedes no other write operation in W . This results, however, from the fact that there is a single writer and that servers echo an update request only if its timestamp is higher than the one they have in store, and so any later write operation has a higher timestamp.

2 AMPRW Dynamic Byz quorum systems paper

Safe variable semantics

Assume failure mode is threshold of faulty servers b (previously f). Quorums of size $(n+2b+1)/2$

Now we want to dynamically set this threshold throughout the execution to adjust for changing assumptions about the system.

Note: how dynamic is this really? We are not changing the current configuration — just the threshold. Changing the config seems harder.

Same set of assumptions as previously (with benign clients). Additional assumptions will be stated next.

Basic idea: introduce a new, replicated variable b that contains the current threshold setting. b can be written with integrals in the range $[b_{min}, b_{max}]$.

When a client updates this threshold variable, it must be written to a set of servers whose intersection with all possible quorums (i.e. for any value of b) is large enough to unambiguously determine the correct current threshold. (*announce sets*).

So we need to determine what those intersections look like, and the modified read/write protocols.

Need assumption about number of failures that changes dynamically:

New failure assumption: $b' =$ min (last value written to b, values written to b in any threshold writes that are concurrent with the current operation). Then no quorum access issued within that operation returns more than b' faulty responses.

Accordingly, we have a

New safety condition: A read operation that overlaps no writes to V or threshold writes to b returns the value of the most recent write to the variable that precedes the read, in some serialization of the writes that precede it.

(Maybe a little to weak, what if we are overlapping a threshold or variable write? Maybe they could have gone further...)

2.1 Threshold adjustment

Basic idea is to assume the presence of b_{max} failures when adjusting the threshold. Therefore, we want the announce set for threshold adjustment (set of writers of variable b) to be as large as possible, so we write to a set of size $n - b_{max}$.

Now the issue is how large should the read quorum be? The intersection between any quorum and any announce set is of size at least $2b_{max}+1$.

We calculate the size of any read quorum Q_r: $|Q_r \cap$ announce set $| \geq 2b_{max}+1$, so $|Q_r|+n-b_{max} \geq n+2b_{max}+1$, so $|Q_r| \geq 3b_{max}+1$. And that's it. In the paper they make the following (wrong, in my opinion) reasoning:

The announce set must intersect every possible quorum (of size $(n+2b+1)/2$, for $b_{min} \leq b \leq b_{max}$ in at least $2b_{max}+1$ servers. This way, the response to any query will include at least $b_{max}+1$ good guys who now about the latest value of b. That is, we need

$2b_{max}+1 \leq ((n+2b_{min}+1)/2+(n-b_{max})-n.$

or $n \geq 6b_{max}-2b_{min}+1$.

(I don't understand where the $(n+2b+1)/2$ read quorum came from. Moving along...)

Protocol:

Write: same as writing to a variable, but use announce set instead of quorum.

Read:

query quorum Q of size $(n+2b_{min}+1)/2$ to obtain value/timestamp pairs. Of the pairs returned by at least $b_{max}+1$ servers select the highest timestamp, provided it was not *countermanded.* If it is countermanded set b to $\perp$.

Definition: a value/timestamp pair is countermanded in a given query if at least $b_{max}+1$ servers return threshold timestamps (not necessarily identical) greater than t. A threshold value b is countermanded if all the pairs it appears in are countermanded.

Thrm 1: If b is older than the most recently completed threshold write at the time of a threshold read, then it will be countermanded in that read.

Proof: If no threshold write operations are taking place concurrently with the threshold read, then the result follows immediately from Assumption 1, the intersection property between announce sets and quorums, and the fact that $b_{max} \geq b$. Furthermore, for any b, the number of correct servers whose threshold timestamp exceeds that of b is monotonically nondecreasing over the course of a threshold write. Therefore the result holds during threshold writes as well.

2.2 Read / Write Protocol

Basic idea: read threshold first, then use that threshold (or b_{max} if read fails) as threshold in a normal read or write.

Problem: When reading, we do not know what was the threshold when the write was done. Thus $(n+2b+1)/2$ might not suffice.

In the worst case, we wrote to $(n+2b_{min}+1)/2$. So we need,

$|Q_{read}|+(n+2b_{min}+1)/2 \geq n+2b+1$, or

$|Q_{read}| \geq (n+2b+1)/2+b-b_{min}$.

A similar situation happens when we are determining the timestamp value to the write operation. We need to read the variable in order to determine a timestamp greater than all preceding writes.

Except that in this case we can conservatively set the timestamp to a value higher than any read, so we only need to intersect in $b+1$ servers, yielding at least one correct response, i.e.

$|Q_{ts-read}| + (n + 2b_{min} + 1)/2 \geq n + b + 1$, or

$|Q_{ts-read}| \geq (n + 2b + 1)/2 - b_{min}$.

So we have the following read and write protocol:

Full version (as in paper - optimized)

Read: 1. Read threshold b. if $b = \perp$ return $\perp$.
2. Query quorum of $(n + 2b + 1)/2$ for value, timestamp pairs.
3. Of the $< v_u, t_u >$ pairs returned by at least $b_{min} + 1$ servers consider the one with the highest timestamp (if none exists, return $\perp$). If it appears in $b+1$ identical responses return v as the result of the read.
4. otherwise query additional $b - b_{min}$ servers, in order to obtain $(n + 2b + 1)/2 + b - b_{min}$ values.
5. Choose the highest timestamp that occurs in at least $b+1$ responses, or return $\perp$ if none exists.

Short version (more traffic, but simpler)

Simplified Read: 1. Read threshold b. if $b = \perp$ return $\perp$.
2. Query quorum of $(n + 2b + 1)/2 + b - b_{min}$ for value, timestamp pairs.
3. Of the $< v_u, t_u >$ pairs returned by at least $b + 1$ servers return the one with the highest timestamp (if none exists, return $\perp$).

Write: 1. Read threshold b. if $b = \perp$ set b to b_{max}.
2. Query servers in Q of size $(n + 2b + 1)/2 - b_{min}$ to obtain timestamps, and choose one higher than any returned (conservative).
3. Write value with the timestamp chosen in (2) to a quorum of size $(n + 2b + 1)/2$.

Correctness arguments are similar to the ones in the previous paper.