the morning paper

a random walk through Computer Science research, by Adrian Colyer

A generalised solution to distributed consensus

MARCH 8, 2019MARCH 2, 2019
A generalised solution to distributed consensus (https://arxiv.org/abs/1902.06776) Howard & Mortier, *arXiv'19*

This is a draft paper that Heidi Howard recently shared with the world via Twitter (https://twitter.com/heidiann360/status/1098167557792718849?s=21), and here's the accompanying blog post (https://hh360.user.srcf.net/blog/2019/02/towards-an-intuitive-high-performance-consensus-algorithm/). It caught my eye for promising a generalised solution to the consensus problem, and also for using reasoning over immutable state to get there. The state maintained at each server is monotonic (https://blog.acolyer.org/2019/03/06/keeping-calm-when-distributed-consistency-is-easy/).

Consensus is a notoriously hard problem (https://blog.acolyer.org/2015/03/01/cant-we-all-just-agree/), and Howard has been deep in the space for several years now. See for example the 2016 paper on Flexible Paxos (https://blog.acolyer.org/2016/09/27/flexible-paxos-quorum-intersection-revisited/). The quest for the holy grail here is to find a unifying and easily understandable protocol that can be instantiated in different configurations allowing different trade-offs to be made according to the situation.

> This paper re-examines the problem of distributed consensus with the aim of improving performance and understanding. We proceed as follows. Once we have defined the problem of consensus, we propose a generalised solution to consensus that uses only immutable state to enable more intuitive reasoning about correctness. We subsequently prove that both Paxos and Fast Paxos are instances of our generalised consensus algorithm and thus show that both algorithms are conservative in their approach.

The distributed consensus problem

We have a set of servers (at least two) that need to agree upon a value. Clients take as input a value to be written, and produce as output the value agreed by the servers.

- The output value must have been the input value of a client (ruling out simple solutions that always output a fixed value for example)
- All clients that produce an output must output the same value
- All clients must eventually produce an output if the system is reliable and synchronous for a sufficient period

Note that we're talking here about the 'inner game' – this is just to agree upon a single value, whereas in a real deployment we'll probably run multiple rounds of the protocol to agree upon a series of values. Moreover, we're assuming here that the set of clients and servers is fixed and known to the clients. The 'clients' are probably not end-user clients, but more likely to be system processes making using of a consensus service. Configuration changes and membership epochs is another layer we'll probably add-on in real deployments, but is also out of scope here.

Building blocks: immutable registers and quorums

We have a fixed set of *n* servers, $S_0, S_1, ..., S_n$. Each server has a tape it can write to, aka, an infinite array of **write-once** persistent registers, $R_0, R_1, ...$.

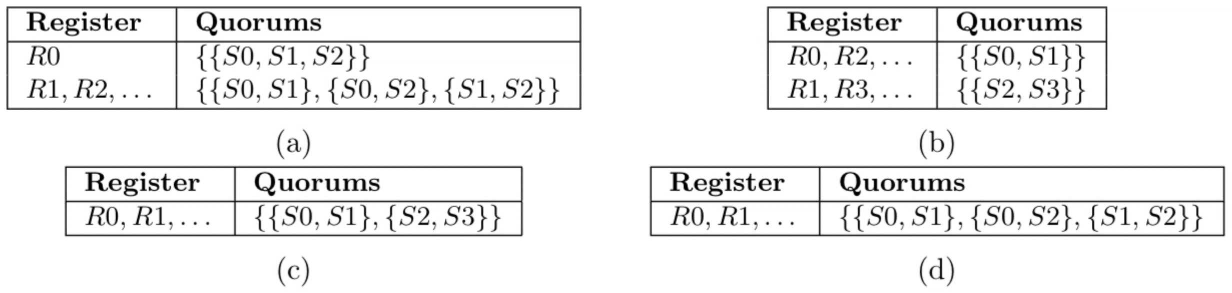
Initially the tape is blank – all registers are *unwritten*. Once we write a value in a register slot it can never be changed. In addition to values provided by clients the system has one special value, denoted *nil* or $\perp$.

We're going to be interested in sets of values from the same register slot across all servers. A *register set*, i is the set of all registers R_i across all servers.

If enough of the registers in a register set have the same (non- $\perp$) value, then we will say that the servers have *decided* upon that value. How many registers in a register set is *enough* though? That's something we get to decide as part of the configuration of the protocol. In fact, we specify more than just the *number* of register values that must be in agreement, we also specify which servers they belong to. A (non-empty) subset of servers that can decide a value between them is called a *quorum*. More precisely, "*a quorum, Q, is a (non-empty) subset of servers, such that if all servers have the same (non-nil) value v in the same register then v is said to be decided.*"

The word 'quorum' is often strongly associated with 'majority,' but note that this doesn't have to be the case. The specification allows us to declare a quorum containing just a single server if we want to. The dictionary definition of quorum is "the minimum number of members of an assembly or society that must be present at any of its meetings to make the proceedings of that meeting valid." Note that there is also no requirement here for overlapping quorum memberships (more on that later).

Each register set is associated with a set of quorums. That is, on a register-set by register-set basis, we can specify exactly which subsets of servers are allowed to decide a value. For example:



Register	Quorums
$R0$	$\{\{S0, S1, S2\}\}$
$R1, R2, \ldots$	$\{\{S0, S1\}, \{S0, S2\}, \{S1, S2\}\}$

(a)

Register	Quorums
$R0, R2, \ldots$	$\{\{S0, S1\}\}$
$R1, R3, \ldots$	$\{\{S2, S3\}\}$

(b)

Register	Quorums
$R0, R1, \ldots$	$\{\{S0, S1\}, \{S2, S3\}\}$

(c)

Register	Quorums
$R0, R1, \ldots$	$\{\{S0, S1\}, \{S0, S2\}, \{S1, S2\}\}$

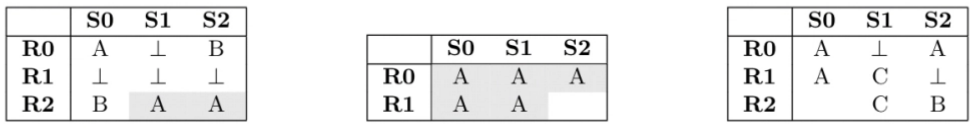
(d)

Figure 1: Sample configurations for systems of three or four servers.

Why on earth we would want to have different quorum sets for different register sets— since we're only agreeing on a single value at the end of the day!— will become more apparent when we look at how to layer a consensus protocol on top of the registers.

Given quorum configurations and register values for register sets, we can see if any quorum is satisfied and a decision reached.

> The state of all registers can be represented in a table, known as a *state table*, where each column represents the state of one server and each row represents a register set. By combining a configuration with a state table, we can determine whether any decision(s) have been reached.
>
> Figure 1: Sample configurations for systems of three or four servers.



	S0	S1	S2
R0	A	⊥	B
R1	⊥	⊥	⊥
R2	B	A	A

(a) A decided by R2

	S0	S1	S2
R0	A	A	A
R1	A	A	

(b) A decided by R0 & R1

	S0	S1	S2
R0	A	⊥	A
R1	A	C	⊥
R2		C	B

(c) No decisions yet

Figure 2: Sample state tables for a system using the configuration in Figure 1a.

Now, we have to be a little careful here and remember that the state table is a logical construct. Each server knows the values of its own registers, and clients can maintain their own view of the global state table as they receive information from servers. Given the immutable constructs though, we know that once a client has assembled enough information to call a value decided, it is finally decided.

Four rules

There are some rules that need to be followed for this to work as a consensus system.

1. **Quorum agreement**. A client may only output a (non-nil) value v if it has read v from a quorum of servers in the same register set.
2. **New value**. A client may only write a (non-nil) value v provided that v is the client's input value or that the client has read v from a register.
3. **Current decision**. A client may only write a (non-nil) value v to register r on server s provided that if v is decided in register set r by a quorum $Q \in Q_r$ where $s \in Q$ then no value v' where $v \neq v'$ can also be decided in register set r.
4. **Previous decisions**. A client may only write a (non-nil) value to register r provided no value v' where $v \neq v'$ can be decided by the quorums in registers sets 0 to $r-1$.

The first two rules are pretty straightforward. Rule 3 ensures that all decisions made by a register set will be for the same value, while rule 4 ensures that all decisions made by different register sets are for the same value.

Obeying these rules places some further constraints on the system configuration. Say we have two quorums for register set *i*, $\{S_0\}$ and $\{S_1\}$. A client writing to the register on S_0 has no way of knowing whether or not some other client might be writing a different value to S_1, hence we couldn't uphold rule 3. There are three ways we can handle this situation:

- The trivial solution is to allow only one quorum per register set
- With multiple quorums, we can require that all quorums for a register set intersect (i.e., any two quorums will have at least one overlapping member)
- We can use *client restricted configurations*. Here ownership of a given register set is assigned to a given client, such that only that client can write values in that register set. Assuming the client is consistent with the values it

> writes, all will be well. One such strategy would be a round-robin ownership of register sets:
>
> | Register | Client |
> |---|---|
> | $R0, R3, \ldots$ | C0 |
> | $R1, R4, \ldots$ | C1 |
> | $R2, R5, \ldots$ | C2 |
>
> Figure 4: Sample round robin allocation of register sets to clients.

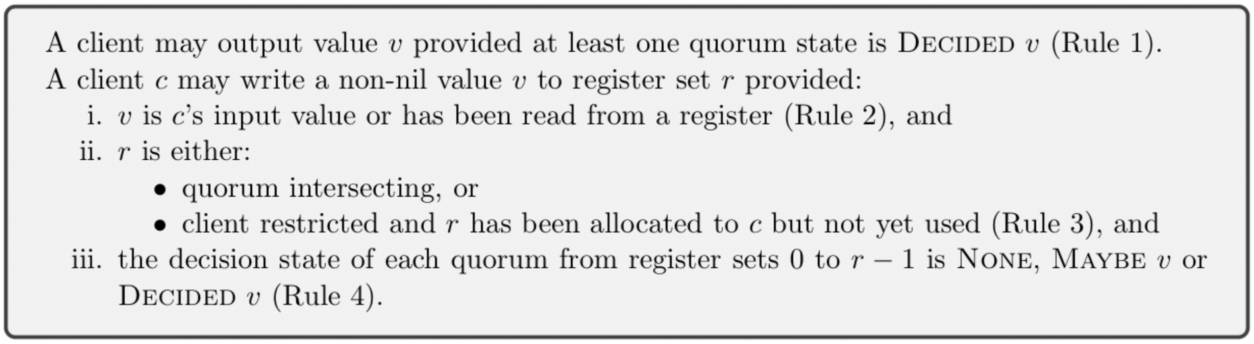
To comply with rule 4, clients need to consult their emerging view of the global state table, aka their *local* state table, and figure out whether any decision has been or could be reached by earlier register sets. The client maintains a decision table with one entry for each quorum of each register set. This table tracks what the client knows about the decisions made by the quorum. Possible values for a quorum are:

- ANY – any value could yet be decided. (All quorums for all register sets start out in this state)
- MAYBE v – if this quorum reaches a decision, then the value will be v
- DECIDED v – the value v has been decided by the quorum (a final state)
- NONE – the quorum will not decide a value; a final state

As the client receives information from servers, it updates the decision table according to the following rules:

- If the client reads the value *nil* from a quorum member, then the state of that quorum is set to NONE.
- If the client reads a non-nil value v for register set r and all quorum members have the same value then the state of the quorum is set to DECIDED. Otherwise the state is set to MAYBE v. In this latter case, for the same quorum in earlier register sets a value of ANY will also be set to MAYBE v, but a value of MAYBE v' where $v \neq v'$ will be updated to NONE.

Clients use the decision table to implement the four rules for correctness as follows:



A client may output value v provided at least one quorum state is Decided v (Rule 1).
A client c may write a non-nil value v to register set r provided:
i. v is c's input value or has been read from a register (Rule 2), and
ii. r is either:
- quorum intersecting, or
- client restricted and r has been allocated to c but not yet used (Rule 3), and

iii. the decision state of each quorum from register sets 0 to $r-1$ is None, Maybe v or Decided v (Rule 4).

Figure 5: Client decision table rules

> Our aim is to make reasoning about correctness sufficiently intuitive that proofs are not necessary to make a convincing case for the safety…

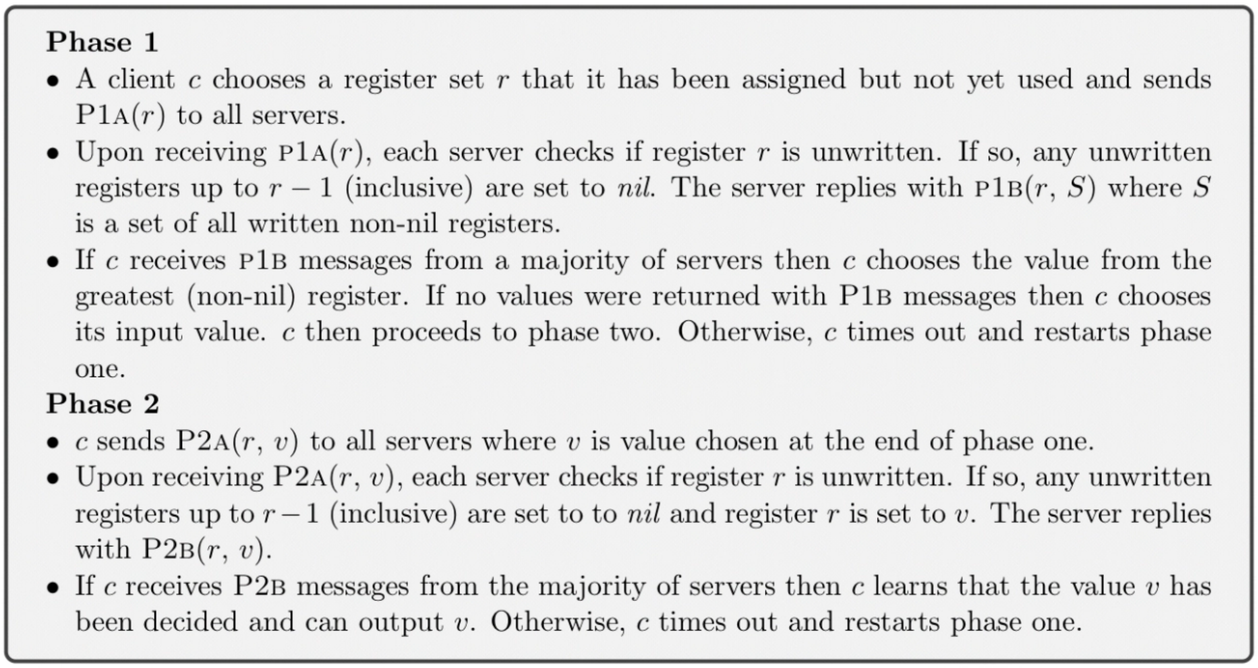
You'll be pleased to know however that proofs *are* included, and you'll find them in Appendix A.

Paxos over immutable registers

As a paper summary, I guess this write-up so far has been a spectacular fail! I'm already over my target length, and have probably taken more words to explain the core ideas than the original paper does!! The only defence I can offer is that there are no short-cuts when it comes to thinking about consensus ;)

An important part of the paper is to show that the register-based scheme just outlined can be instantiated to realise a number of consensus schemes from the literature, including Paxos and Fast Paxos. I'm out of space to do that proper justice, but here's a sneak-peek at the core Paxos algorithm:

> We observe that Paxos is a conservative instance of our generalised solution to consensus. The configuration used by Paxos is majorities for all register sets.. Paxos also uses client restricted for all register sets. The purpose of phase one is to implement rule 4, and the purpose of phase two is to implement rule 1.



Phase 1

- A client c chooses a register set r that it has been assigned but not yet used and sends P1A(r) to all servers.
- Upon receiving P1A(r), each server checks if register r is unwritten. If so, any unwritten registers up to $r-1$ (inclusive) are set to *nil*. The server replies with P1B(r, S) where S is a set of all written non-nil registers.
- If c receives P1B messages from a majority of servers then c chooses the value from the greatest (non-nil) register. If no values were returned with P1B messages then c chooses its input value. c then proceeds to phase two. Otherwise, c times out and restarts phase one.

Phase 2

- c sends P2A(r, v) to all servers where v is value chosen at the end of phase one.
- Upon receiving P2A(r, v), each server checks if register r is unwritten. If so, any unwritten registers up to $r-1$ (inclusive) are set to to *nil* and register r is set to v. The server replies with P2B(r, v).
- If c receives P2B messages from the majority of servers then c learns that the value v has been decided and can output v. Otherwise, c times out and restarts phase one.

Figure 8: The Paxos algorithm using write-once registers.

If you differentiate between the quorums used for each register set and which phase of Paxos the quorum is used for, you can arrive at Flexible Paxos (https://blog.acolyer.org/2016/09/27/flexible-paxos-quorum-intersection-revisited/).

Mixing it up

Section 5 of the paper shows how to derive Fast Paxos, and in section six you'll find a brief sketch of three other points in the configuration space named co-located consensus, fixed-majority consensus, and reconfigurable consensus.

> In this paper, we have reframed the problem of distributed consensus in terms of write-once registers and thus proposed a generalised solution to distributed consensus. We have demonstrated that this solution not only unifies existing algorithms including Paxos and Fast Paxos but also demonstrates that such algorithms are conservative as their quorum intersection requirements and quorum agreement rules can be substantially weakened. We have illustrated the power of our generalised consensus algorithm by proposing three novel algorithms for consensus, demonstrating a few interesting points on the diverse array of algorithms made possible by our abstract.

DISTRIBUTED SYSTEMS

11 thoughts on “A generalised solution to distributed consensus”

1. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – Latest news
2. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – Hckr News
3. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – World Best News
4. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – Golden News
5. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – Outside The Know
6. Pingback: A generalised solution to distributed consensus | My Tech Blog
7. Pingback: New top story on Hacker News: A generalised solution to distributed consensus – News about world
8. Pingback: A generalised solution to distributed consensus – Hacker News Robot
9. Pingback: A generalised solution to distributed consensus – TARIAN LIMITED
10. Pingback: === popurls.com === popular today
11. Pingback: A generalised solution to distributed consensus