CONSENSUS IN EVENTUALLY SYNCHRONOUS: PAXOS

DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

RECAP

Synchronous consensus:

- Regular:
  - Hierarchical Consensus: Each round r has a leader the process p_r. p_r decides its proposal v and imposes its proposal on others using a beb. You go in a new round r+1 when either the r crashes or you received from r. The algorithm performs N round (you can only decide when you are round leader).
- Uniform:
  - Uniform Hierarchical Consensus: The round leader p_r, imposes its value with a broadcast-ack mechanism. When it receives ack from all corrects it decides and RegularBrodcast its decision. You go in a new round r+1 only when you detect the crash of r.
- FAULT-TOLERANCE:
  - Both solutions tolerates any number of failures (they use **P**!)

WHAT IF THE SYSTEM IS ASYNCHRONOUS?

Theorem. (FLP) Consider an asynchronous system composed by n processes. If **f>0** then consensus is unsolvable in such system.

The proof shows that for any algorithm there exists a run in which you either:

- Violates a safety
- Violates liveness

EVENTUALLY - SYNCHRONOUS

PAXOS A BIT OF HISTORY

The Paxos family of algorithms was introduced in 1989 with the paper,

The Part-Time Parliament "…A fault-tolerant file system called *Echo* was built at SRC in the late 80s. The builders claimed that it would maintain consistency despite **any number of non-Byzantine faults,** and would make progress if any majority of the processors were working. As with most such systems, it was quite simple when nothing went wrong, but had a complicated algorithm for handling failures based on taking care of all the cases that the implementers could think of. **I decided that what they were trying to do was impossible, and set out to prove it.** Instead, I discovered the **Paxos algorithm,** described in this paper…" (Leslie Lamport).

Later Lamports published a paper "Paxos made simple": "…At the PODC 2001 conference, **I got tired of everyone saying how difficult it was to understand the Paxos algorithm,** published in [122]. Although people got so hung up in the pseudo-Greek names that they found the paper hard to understand, the algorithm itself is very simple. So, I cornered a couple of people at the conference and explained the algorithm to them orally, with no paper. When I got home, I wrote down the explanation as a short note, which I later revised based on comments from Fred Schneider and Butler Lampson. The current version is 13 pages long, and contains no formula more complicated than $n1 > n2$…"

http://lamport.azurewebsites.net/pubs/pubs.html#lamport-paxos

PAXOS MADE SIMPLE

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

1.

2.

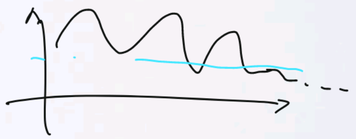
PAXOS

The Paxos provide a viable solution to consensus in eventually synchronous settings with crash failures:

- **Safety** (validity + integrity + agreement) is always guaranteed
- The algorithm is **live** (termination) only when the network behaves in a “good way” for long enough periods of time.

- Assumption: majority of correct processes and **eventually synchronous system**.

eventually synchronous systems =>

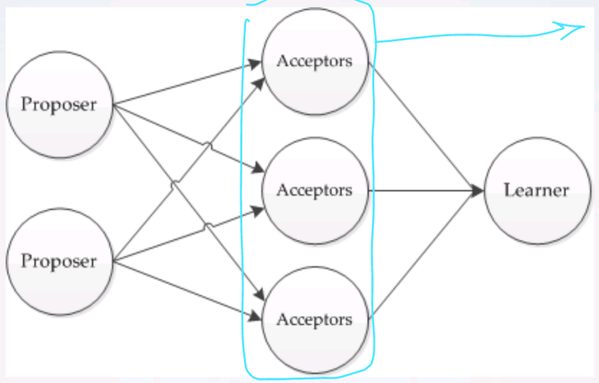


PAXOS

Actors in the basic Paxos protocol

- **Proposers**: propose values.
- **Acceptors**: processes that must commit on a final decided value.
- **Learners**: passively assist to the decision and obtain the final decided value.

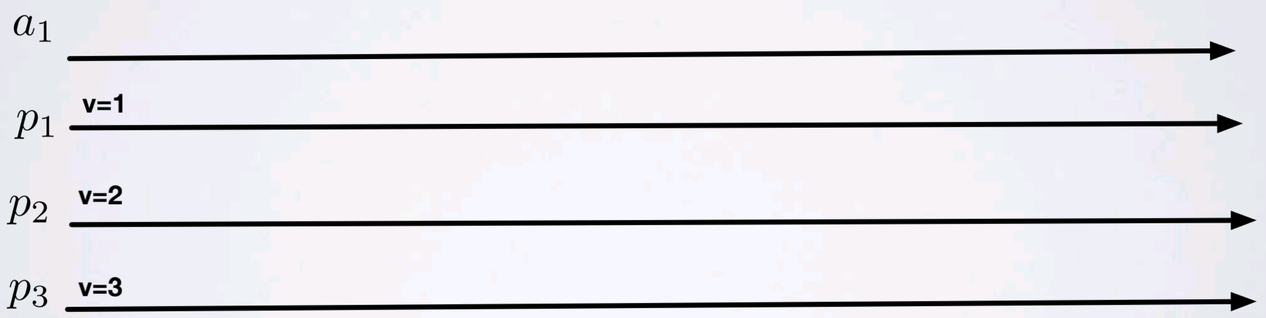
The majority assumption is needed only on the set of acceptors.



PAXOS

Actors in the basic Paxos protocol

- **Proposers**: propose values.
- **Acceptors**: processes that must commit on a final decided value.
- **Learners**: passively assist to the decision and obtain the final decided value.



PAXOS

Let us think about what the protocol should do.

- Only a value that has been proposed may be **chosen** ⇒ validity
- Only a single value is **chosen** ⇒ agreement
- A process never learns that a value has been **chosen** unless it actually has been

Chosen: one on which the consensous will agree.

Time(ehosen) ≠ Time(learned)

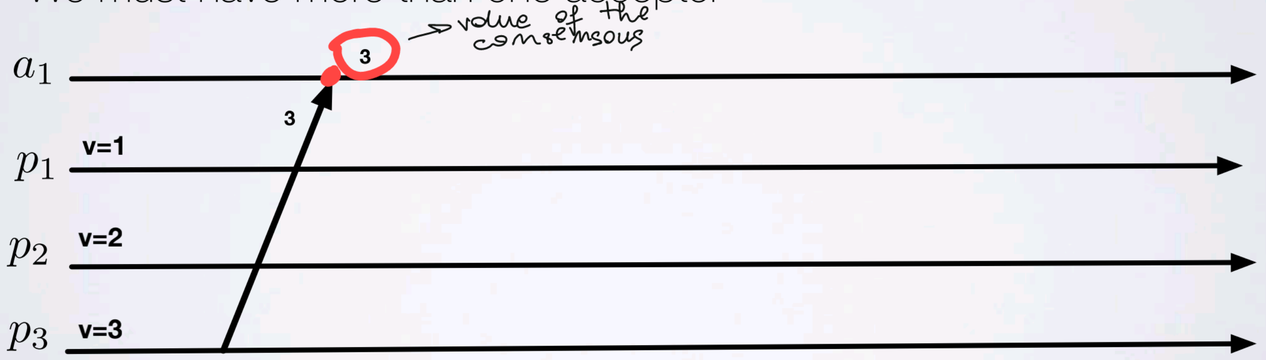
PAXOS

The easiest way to solve the problem is to have a single acceptor

- A proposer sends a proposal to the acceptor
- The acceptor choses the first proposed value it receives

What if the only acceptor fails ?

We must have more than one acceptor



PAXOS

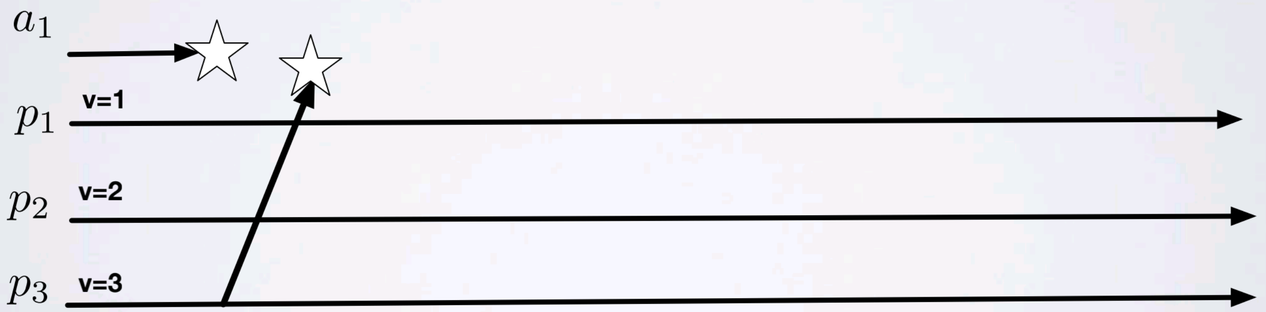
need more acceptors →

The easiest way to solve the problem is to have a single acceptor

- A proposer sends a proposal to the acceptor
- The acceptor choses the first proposed value it receives

What if the only acceptor fails ?

We must have more than one acceptor



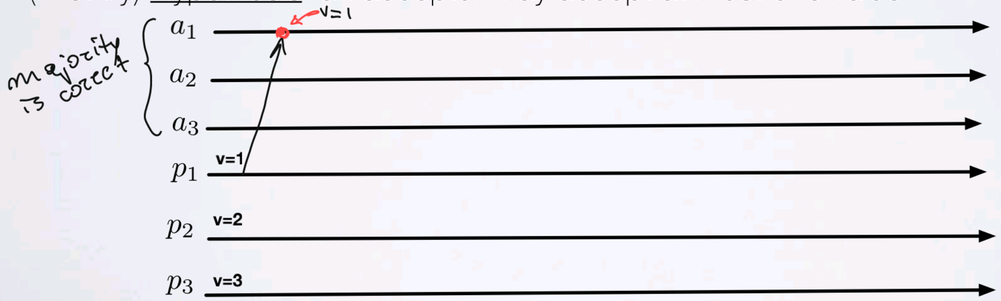
PAXOS

Multiple acceptors

- A proposer sends a proposal to a set of acceptors
- An acceptor MAY accept the proposed value
- A value is chosen when a majority of acceptors accept it

The majority is needed to guarantee that only a value is chosen -> we want to preserve the algorithm invariant that **a value is chosen when accepted by the majority of acceptors.**

(First try) Hypothesis: an acceptor may accept at most one value

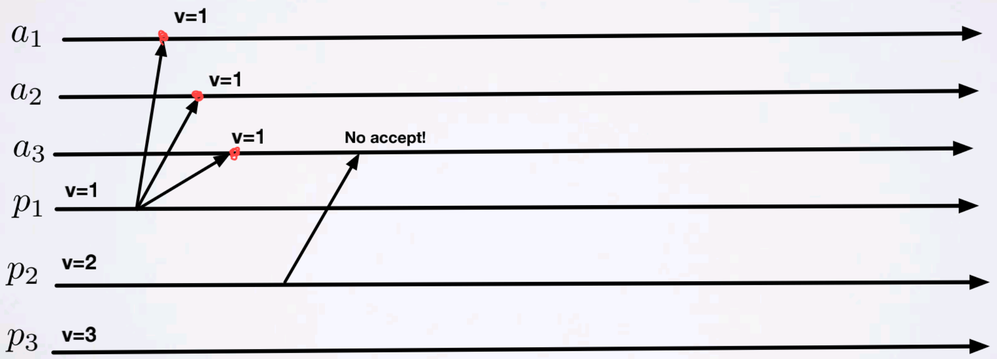


PAXOS

We want a value to be chosen even if only one value is proposed by a single proposer

P1: An acceptor must accept the first proposal it receives

Is it ok if an acceptor refuses other proposal?



PAXOS

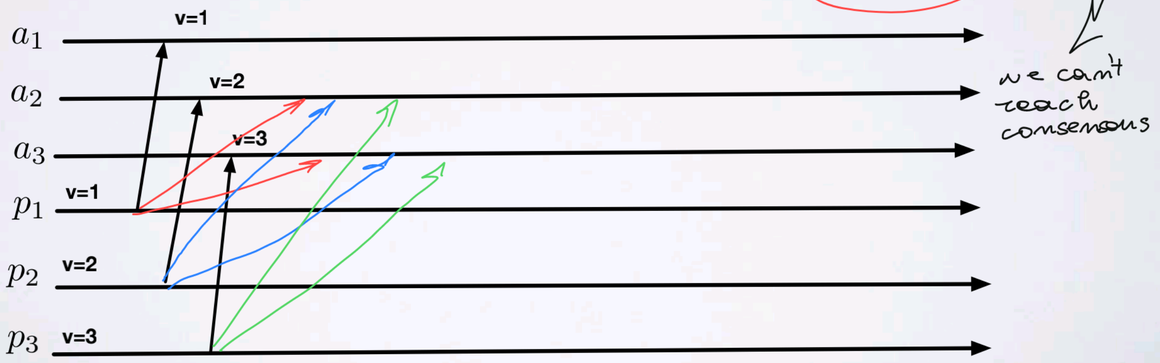
they can start concurrently but

If you assume no failures and no message loss

- We want a value to be chosen even if only one value is proposed by a single proposer

P1: An acceptor must accept the first proposal it receives

Problem: if several values are concurrently proposed by different proposers, none of them could reach the needed majority. We have a deadlock.



we can't reach consensus

PAXOS

If you assume no failures and no message loss

- We want a value to be chosen even if only one value is proposed by a single proposer

P1: An acceptor must accept the first proposal it receives

Problem: if several values are concurrently proposed by different proposers, none of them could reach the needed majority. We have a sort of deadlock.

BUT

This implies that an acceptor should accept multiple proposals

to get majority

PAXOS

We keep track of different proposed values assigning them a proposal number that is **unique** (**total order of proposals**). A proposal will be a pair <round number,value>.

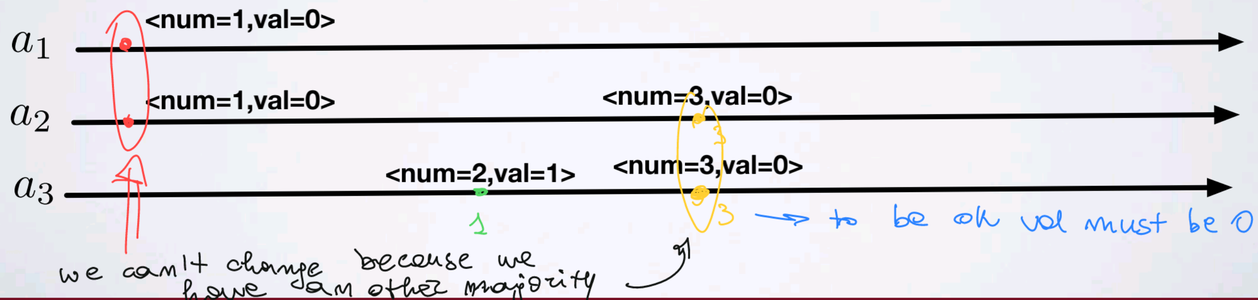
A **value is chosen** when a **single proposal with that value has been accepted** by a **majority of the acceptors** (chosen proposal).

We can allow multiple proposals to be chosen, but **all chosen proposals must have the same value**.

consensus

by majority

P2' if a proposal with value v is chosen, every higher-numbered proposal that is chosen has value v



SAPIENZA UNIVERSITÀ DI ROMA

concept of future is given by rounds → $\langle r=1; v=0\rangle$, $\langle r=3; v=4\rangle$ } example

PAXOS

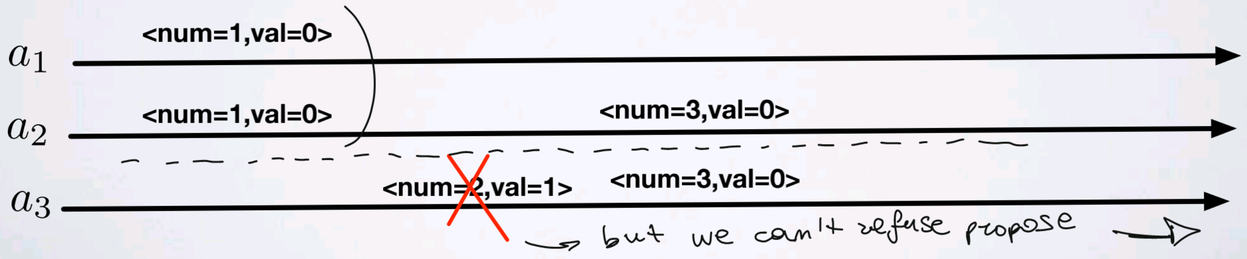
For a value **v** to be chosen, a proposal containing it must have been accepted by at least one acceptor, thus:

*P2a: if a proposal with value **v** is chosen, every higher-numbered proposal that is accepted by any acceptor has value **v***

- P2a ➔ P2

What if a proposal with value **v** is chosen while an acceptor **c** never saw it ?

A new proposer could wake up and propose to **c** a different value **v'** that **c** must accept due to P1



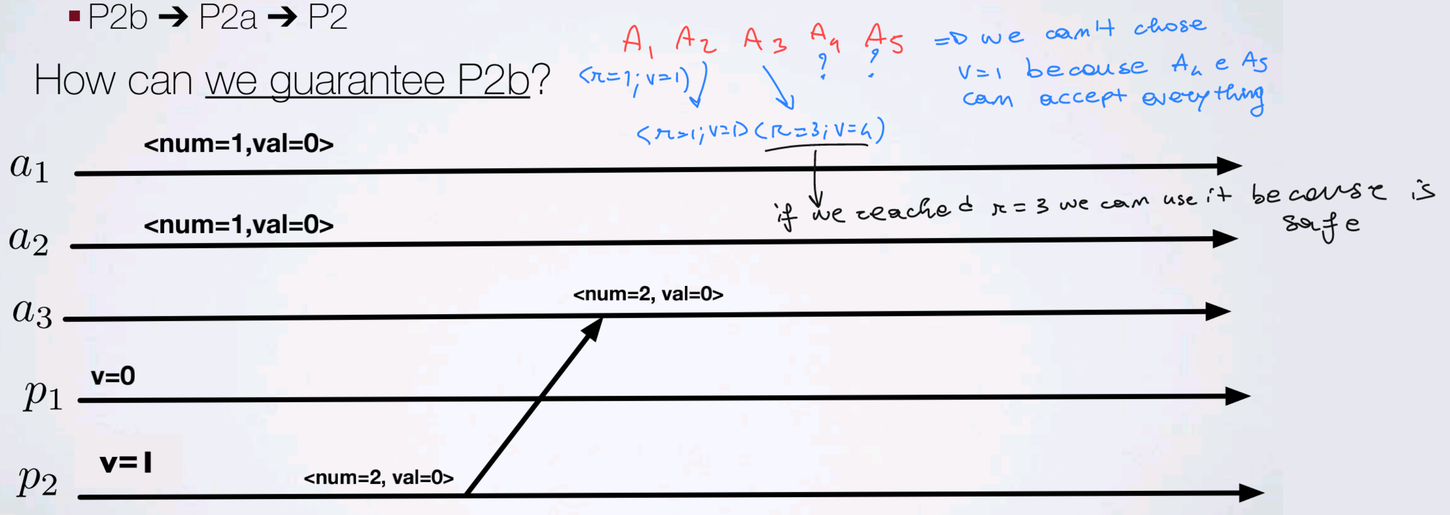
PAXOS

We must slightly modify P2a to take into account this case:

*P2b: if a proposal with value **v** is chosen, every higher-numbered proposal issued by any proposer has value **v***

- P2b ➔ P2a ➔ P2

How can we guarantee P2b?



PAXOS

Assume a proposal of round **m** with value **v** has been chosen.

We want a mechanism that would allow us to prove that any proposal **n**, with round **n>m** contains value **v** by induction on **n** assuming that

- every proposal with round number in **[m, n-1]** contains value **v**

PAXOS

For a proposal in round **m** to be chosen there is a quorum Q of acceptors (a majority) that accepted it.

Therefore, the assumption that some value was chosen in round **m** implies that:

> *Every acceptor in* Q *accepted a proposal with round number* ***m*** *with value* ***v***

PAXOS

Since any majority of acceptors S contains at least one member of Q, we can conclude that a proposal round numbered **n** has value **v** by ensuring that the following invariant is maintained:

*P2c: For any **v** and **n**, if a proposal with value **v** and round number **n** is issued, then there is a set S consisting of a majority of acceptors such that either:*

*(a) no acceptor in S has accepted any proposal round numbered less than **n**.*

*(b) **v** is the value of the highest-numbered proposal among all proposals round numbered less than **n** accepted by the acceptors in S.*

P2c ➔ P2b → P2a → P2 ⇒ consensus

base + induction

PAXOS

P2c ➔ P2b ➔ P2a ➔ P2.

P2c can be maintained by asking to a proposer that wants to propose a value numbered **n** to learn the highest-numbered value (with number less than **n**, if any) that

- has been accepted
- or will by accepted

by each acceptor in a majority.

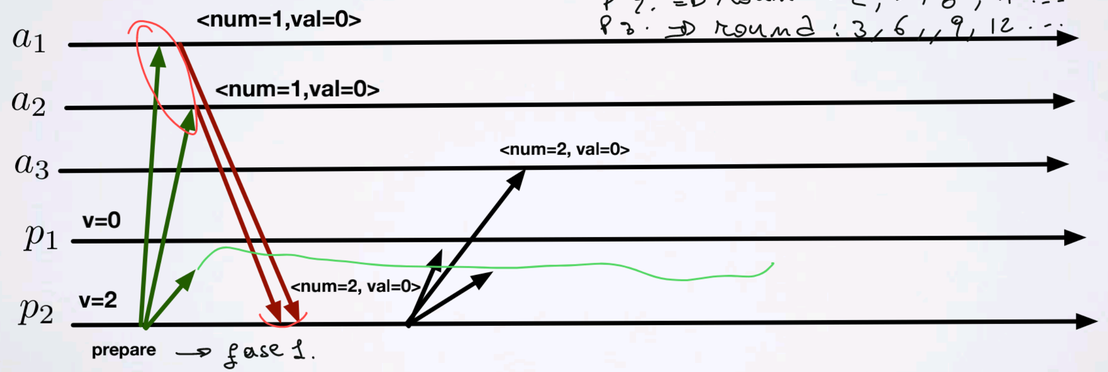
2 proposers can't have same round

⇒ 3 process for example have

p 1. ⇒ round : 1, 4, 7, 10 ...

p 2. ⇒ round : 2, 5, 8, 11 ...

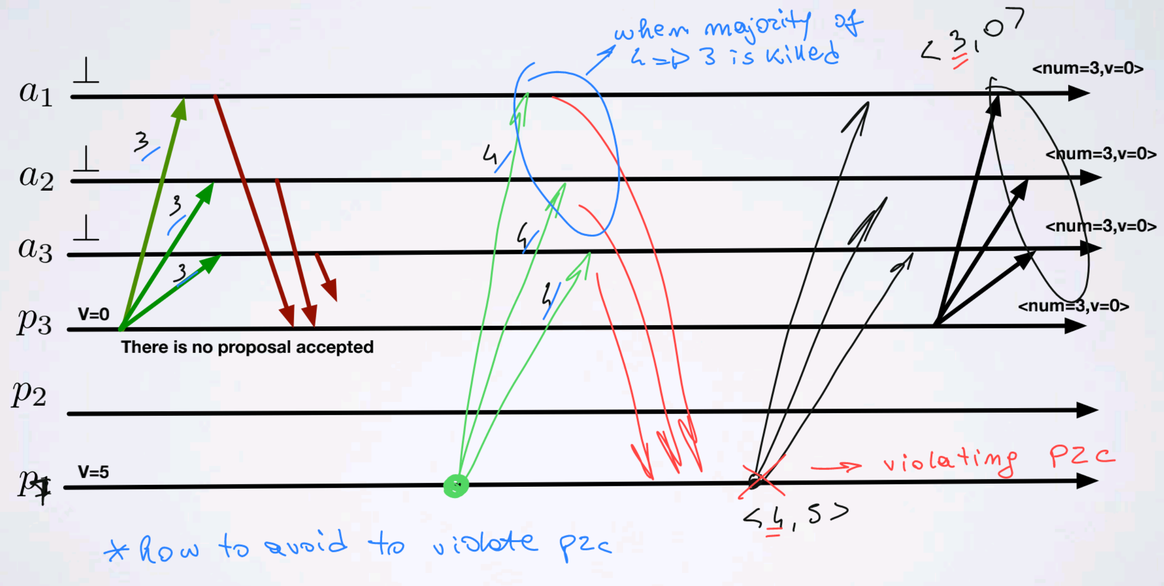
p 3. ⇒ round : 3, 6, 9, 12 ...



PAXOS

P2c ➔ P2b ➔ P2a ➔ P2.

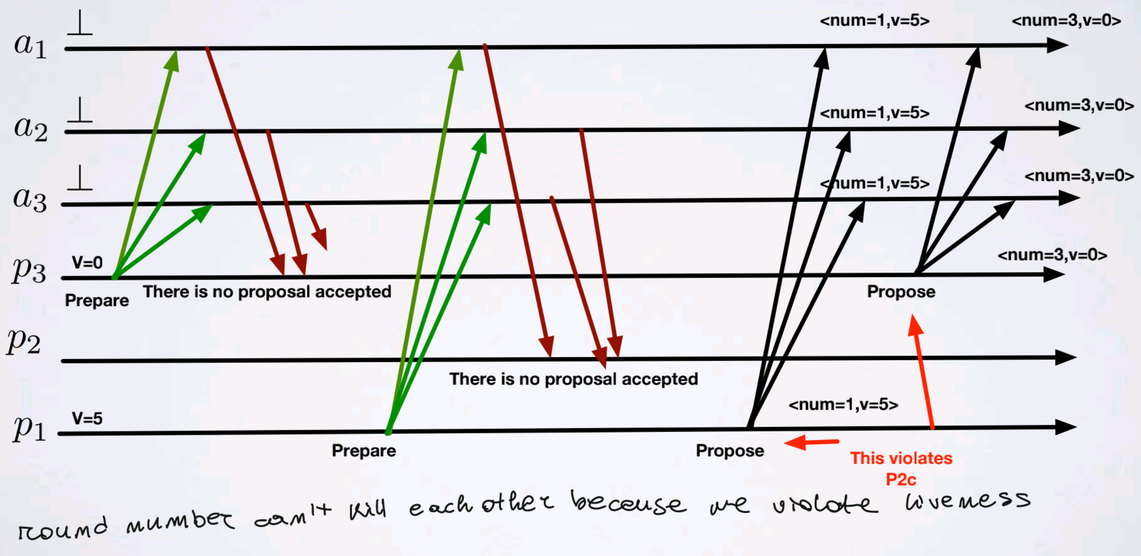
Remember that P2c says `` **v** is the value of the highest-numbered proposal among all proposals numbered less than **n** accepted by the acceptors in S"



PAXOS

P2c ➔ P2b ➔ P2a ➔ P2.

Remember that P2c says `` **v** is the value of the highest-numbered proposal among all proposals numbered less than **n** accepted by the acceptors in S"

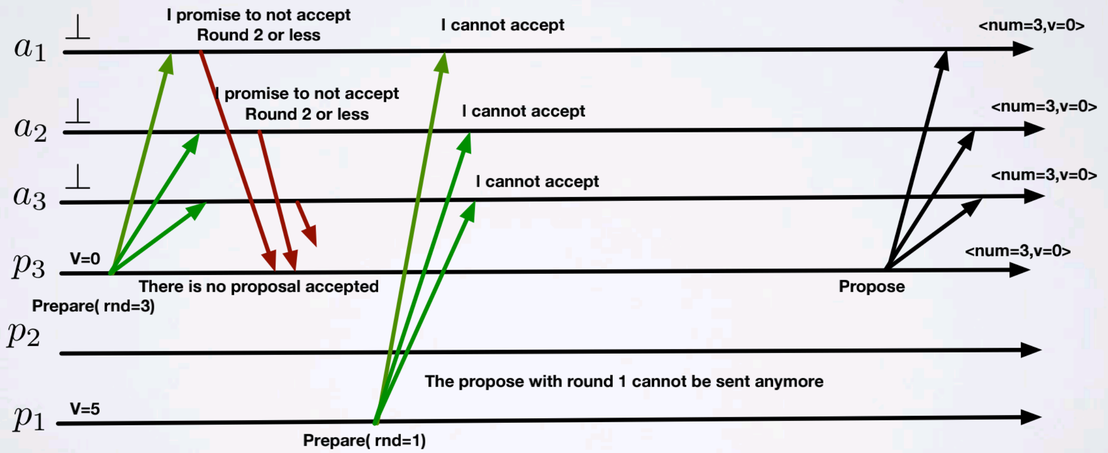


PAXOS

P2c ➔ P2b ➔ P2a ➔ P2.

Remember that P2c says `` **v** is the value of the highest-numbered proposal among all proposals numbered less than **n** accepted by the acceptors in S"

ask acceptors to promise they will not accept proposals numbered less than **n**

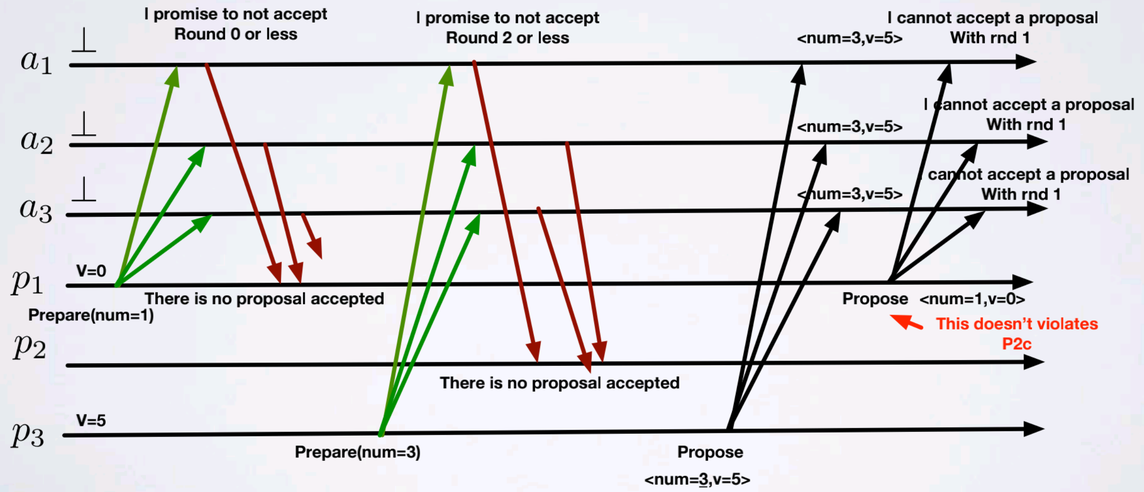


PAXOS

P2c ➔ P2b ➔ P2a ➔ P2.

Remember that P2c says `` **v** is the value of the highest-numbered proposal among all proposals numbered less than **n** accepted by the acceptors in S"

ask acceptors to promise they will not accept proposals numbered less than **n**



PAXOS

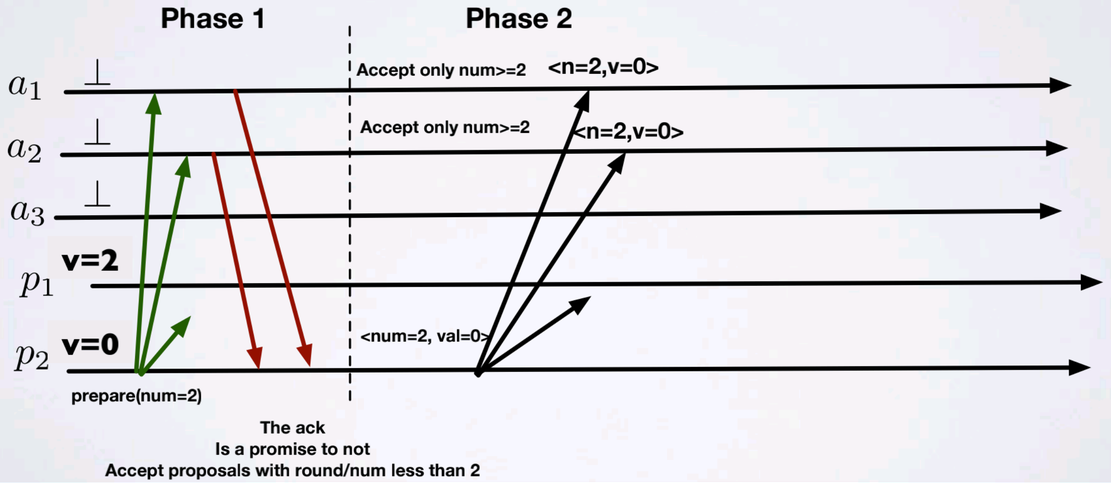
The protocol has two main phases:

- Phase 1: prepare request ↔ response
- Phase 2: accept request ↔ response

PAXOS

Phase 1

- 1) A proposer chooses a new proposal round number n , and sends a prepare request **(PREPARE,n)** to a majority of acceptors:
  - (a) Can I make a proposal with number n ?
  - (b) if yes, do you suggest some value for my proposal?

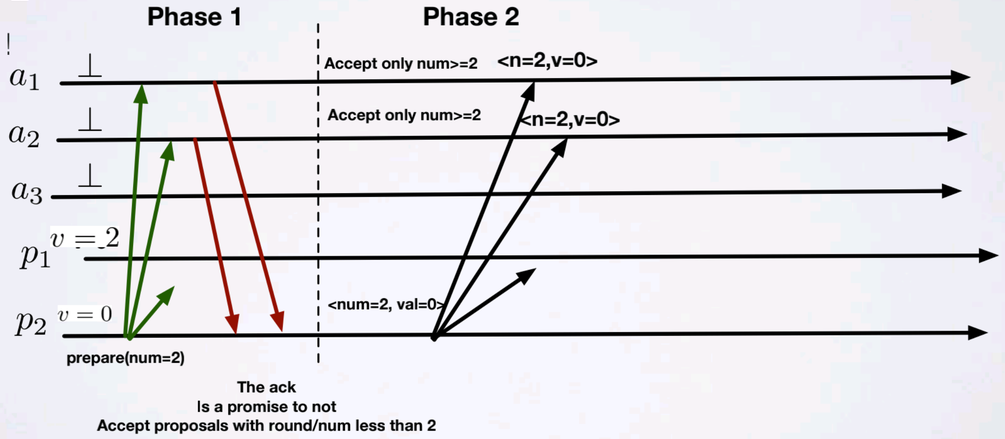


PAXOS

Phase 1

- 2) If an acceptor receives a prepare request **(PREPARE, n)** with n greater than n' from any prepare request it has already responded, sends out **(ACK, n, n', v') or (ACK, n, ⊥ , ⊥)**
  - (a) responds with a promise not to accept any more proposals numbered less than n.
  - (b) suggests the value v' of the highest-number proposal n' that it has accepted if any, else ⊥

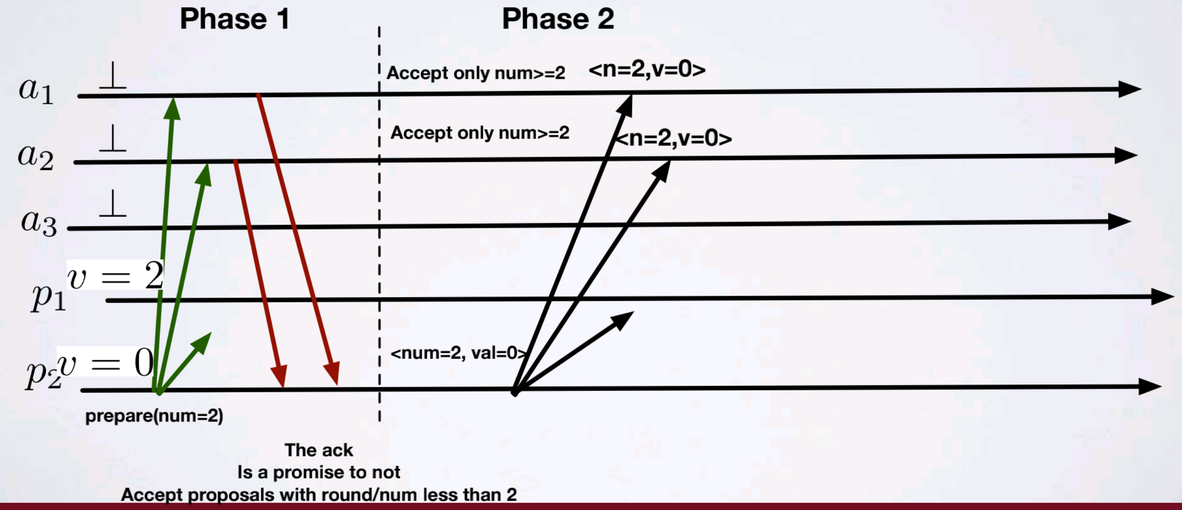
Note: n'<n !



PAXOS

Phase 1

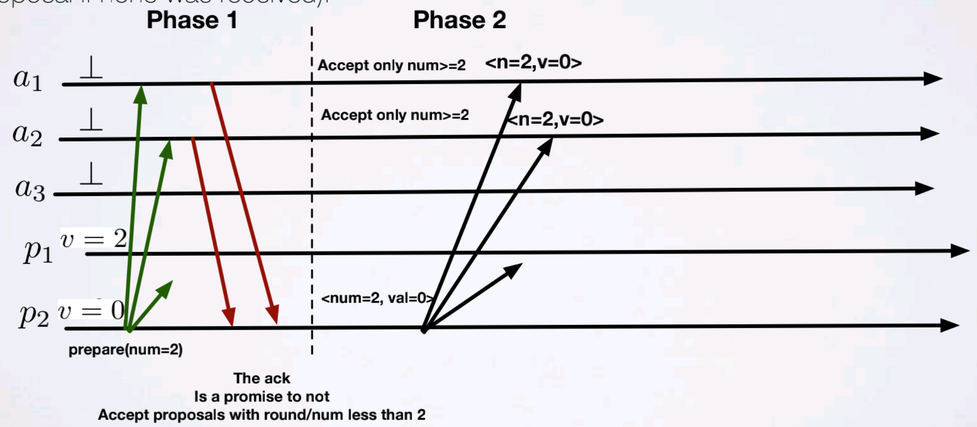
- 2) If an acceptor receives a prepare request **(PREPARE, n)** with n lower than n' from any prepare request it has already responded, sends out **(NACK, n')**
  - (a) responds with a denial to proceed with the agreement as the proposal is too old.



PAXOS

Phase 2

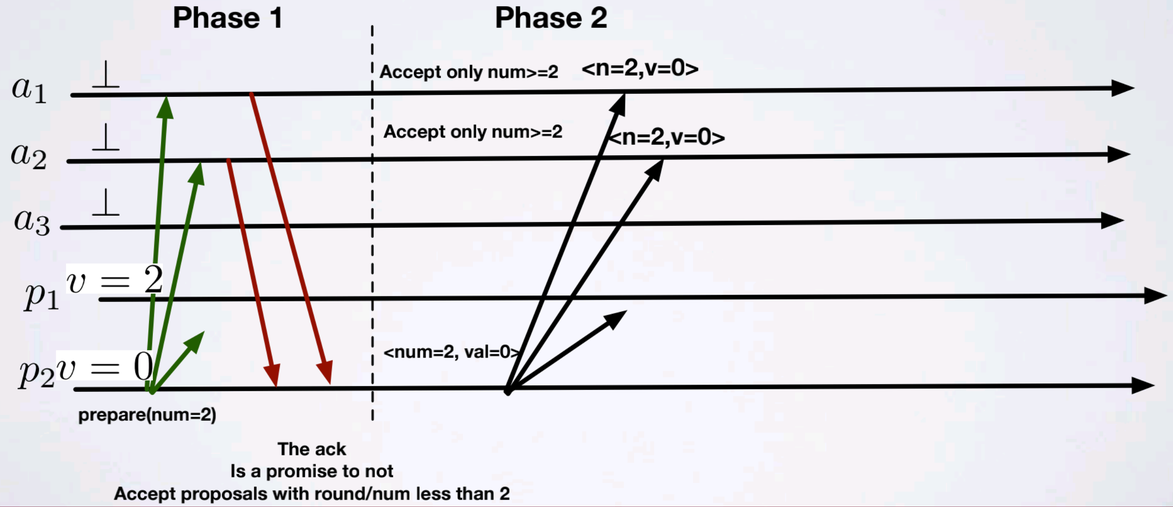
- 3) If the proposer receives responses from a majority of the acceptors, then it can issue an accept request **(ACCEPT, n , v)** with number n and value v:
  - (a) n is the number that appears in the prepare request.
  - (b) v is the value of the highest-numbered proposal among the responses (or the proposer's own proposal if none was received).



PAXOS

Phase 2

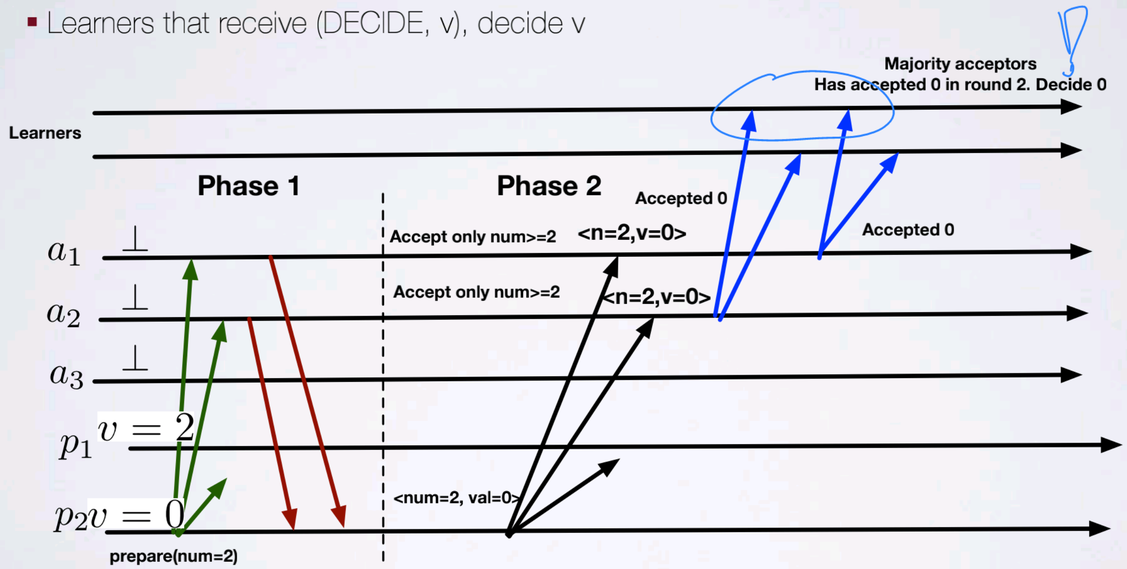
- 4) If the acceptor receives an accept request **(ACCEPT, n , v)** , **it accepts the proposal unless it has already responded to a prepare request having a number greater than n.**



PAXOS

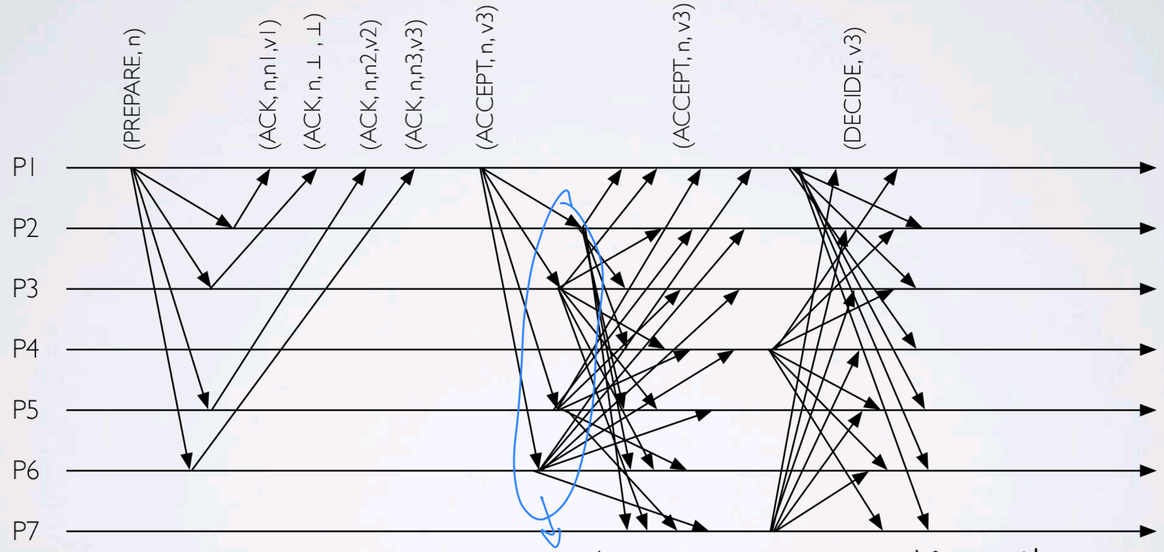
Learning the decided value

- Whenever an acceptor accepts a proposal, it sends to all learners (ACCEPT, n, v).
- A learner that receives (ACCEPT, n, v) from a majority of acceptors, decides v, and sends (DECIDE, v) to all other learners.
- Learners that receive (DECIDE, v), decide v



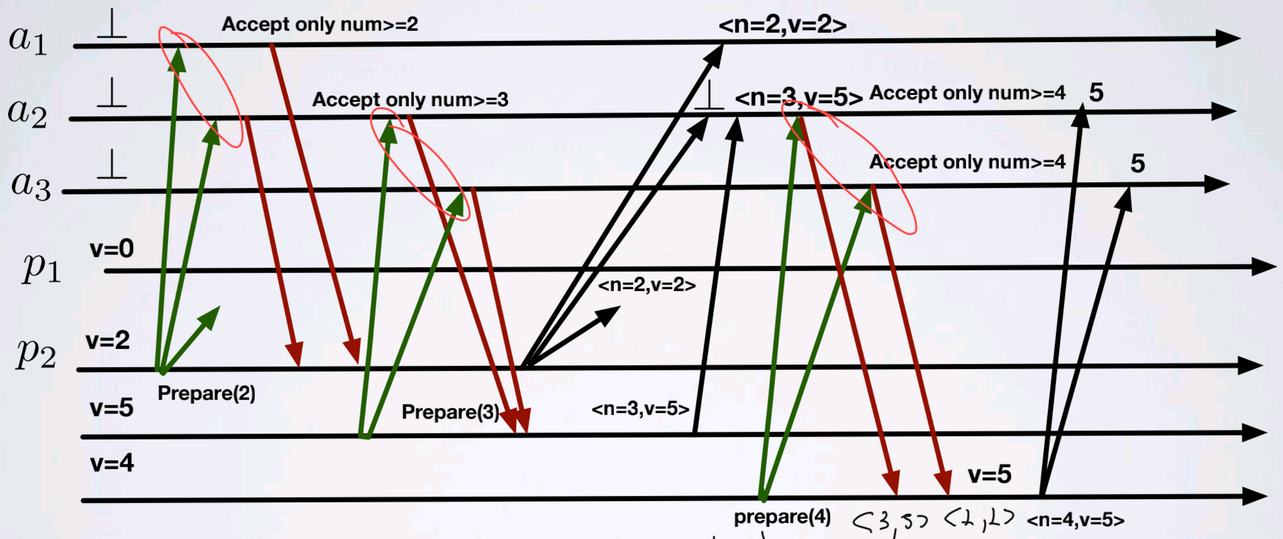
PAXOS

Agents can run on the same physical process



ANOTHER - EXAMPLE

#A #A #L·#A



https://angus.nyc/2012/paxos-by-example/

Deeloy: unbounded - worst case
4n - best case

HOW TO PICK ROUND NUMBERS?

If there are two proposers P1, P2 one takes even numbers the other odd numbers (impossible for them to generate two proposals with the same number).

If there are three Proposers P1,P2,P3.

P1:{1,4,7,10,…}

P2:{2,5,8,11,…}

P3:{3,6,9,12,…}

If there are four Proposers…

Algorithm 1 Paxos — Proposer p

ALGORITHM assumes that acceptors have a timeout => not needs Acks

1: **Constants:**
2: A, n, and f. {A is the set of acceptors. $n = |A|$ and $f = \lfloor (n-1)/2 \rfloor$.}
3: **Init:**
4: $crnd \leftarrow -1$ {Current round number}
5: **on** $\langle$PROPOSE, $val\rangle$
6: $crnd \leftarrow$ pickNextRound($crnd$) ~> slide precedente (assegna i round nel modo delle
7: $cval \leftarrow val$
8: $P \leftarrow \emptyset$ ~> set of promisses by the acceptor
9: **send** $\langle$PREPARE, $crnd\rangle$ **to** A
10: **on** $\langle$PROMISE, $rnd, vrnd, vval\rangle$ with $rnd = crnd$ **from** acceptor a
11: $P \leftarrow P \cup (vrnd, vval)$ -> if first message => (-1, -1)
12: **on event** $|P| \geq n - f$
13: $j = \max\{vrnd : (vrnd, vval) \in P\}$
14: **if** $j \geq 0$ **then**
15: $V = \{vval : (j, vval) \in P\}$
16: $cval \leftarrow$ pick(V) {Pick proposed value $vval$ with largest $vrnd$}
17: **send** $\langle$ACCEPT, $crnd, cval\rangle$ **to** A

1 2 = 5

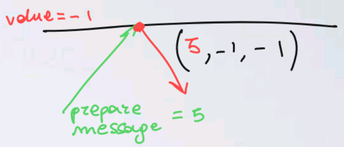
Code from the paper: **A Simpler Proof for Paxos and Fast Paxos**

Algorithm 2 Paxos — Acceptor a

1: **Constants:**
2: L {Set of learners}

3: **Init:**
4: $rnd \leftarrow -1$
5: $vrnd \leftarrow -1$
6: $vval \leftarrow -1$

sentinel value (we know that it's lower than any real round number)



7: **on** $\langle$PREPARE, $prnd\rangle$ with $prnd > rnd$ **from** proposer p
8: $rnd \leftarrow prnd$
9: **send** $\langle$PROMISE, $rnd, vrnd, vval\rangle$ **to** proposer p

10: **on** $\langle$ACCEPT, $i, v\rangle$ with $i \geq rnd$ **from** proposer p
11: $rnd \leftarrow i$
12: $vrnd \leftarrow i$
13: $vval \leftarrow v$
14: **send** $\langle$LEARN, $i, v\rangle$ **to** L

PAXOS CODE

Algorithm 3 Paxos — Learner l

1: **Init:**
2: $V \leftarrow \emptyset$

3: **on** $\langle \text{LEARN}, (i, v) \rangle$ **from** acceptor a
4: $\quad V \leftarrow V \uplus (i, v)$

5: **on event** $\exists i, v : |\{(i, v) : (i, v) \in V\}| \geq n - f$
6: $\quad v$ **is chosen**

Multiset (every set have a cardinality) ex: $\{1, 1, 2, 3, 3, 3, 3\}$

$\downarrow$

$\uplus$

Code from the paper: **A Simpler Proof for Paxos and Fast Paxos**

PAXOS PROOF

If an acceptor a_i accepts to a proposal <num=r,val> we say that a_i voted for val at round r.

We have to prove three properties:

1 Only a value proposed can be chosen

2 A single value is chosen

proof

3 Only a chosen value is learned by learners

PAXOS PROOF

We will show the following:

If acceptor **a** has voted for value v at round i, then no value v' different from v can be chosen in a previous round j<i.

This implies (2) ``A single value v is chosen" (In each round a single value is proposed -**why?**).

PAXOS PROOF

Claim: If acceptor **a** has voted for value v at round i, then no value v' different from v can be chosen in a previous round j<i.

Proof.

Observation: If **a** voted for v at round r=i then, there is a proposer p that collected a set of acks S from a quorum Q of acceptors for its prepare(r=i).

The proof is by induction on rounds number:

- r=0. The claim is correct since round r=-1 does not exist.
- Inductive hypothesis, that the claim is correct for rounds r=1,2,3,4,…,i-1.
- Inductive step. Let us investigate round r=i. Let <r=j,v> the vote for the highest round seen in the set S (recall that i>j).

PAXOS PROOF

Claim: If acceptor **a** has voted for value v at round i, then no value v' different from v can be chosen in a previous round j<i.

Proof. (Inductive step)

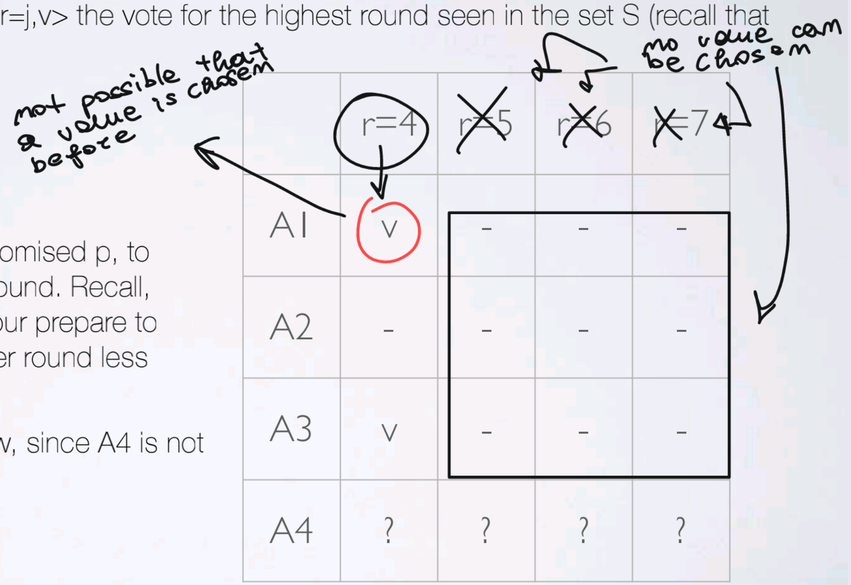
Let us investigate round r=i. Let <r=j,v> the vote for the highest round seen in the set S (recall that i>j).

Example with j=4, and i=8

Q:{A1,A2,A3}

The - means that the Acceptor promised p, to not vote for any proposal in that round. Recall, that if **a** sends us a response to our prepare to round i, it will not vote for any other round less than i.

The ? Means that we do not know, since A4 is not in Q.



PAXOS PROOF Generalization

Claim: If acceptor **a** has voted for value v at round i, then no value v' different from v can be chosen in a previous round j<i.

Proof. (Inductive step)

Let us investigate round r=i. Let <r=j,v> the vote for the highest round seen in the set S (recall that i>j).

For rounds that are between j and r=i-1 we have that acceptors in Q have promised to not vote. Thus, it is not possible for another proposer to get a quorum. Recall that a quorum is n/2+1 acceptors.

So no value can be chosen between r=j and r=i-1.

It remains to show that a different value cannot be chosen between r=0 and r=j.

	r=4	r=5	r=6	r=7
A1	v	-	-	-
A2	-	-	-	-
A3	v	-	-	-
A4	?	?	?	?

if highest round = -1
=> not possible that a value is accepted => every value can be accepted

PAXOS PROOF

Claim: If acceptor **a** has voted for value v at round i, then no value v' different from v can be chosen in a previous round j<i.

Proof. (Inductive step)

It remains to show that a different value cannot be chosen between r=0 and r=j.

Note that if j=0 (that is no acceptor in our quorum voted for v in previous rounds, then the previous case prove the bound).

Therefore j>0, also if in r=j some acceptor voted for v then we can apply the inductive hypothesis and this shows that in all round between 0 and j no value different from v can be chosen.

	r=4	r=5	r=6	r=7
A1	v	-	-	-
A2	-	-	-	-
A3	v	-	-	-
A4	?	?	?	?

PAXOS PROOF

Pay attention to the properties that we have proved

1 Only a value proposed can be chosen

2 A single value is chosen

3 Only a chosen value is learned by learners

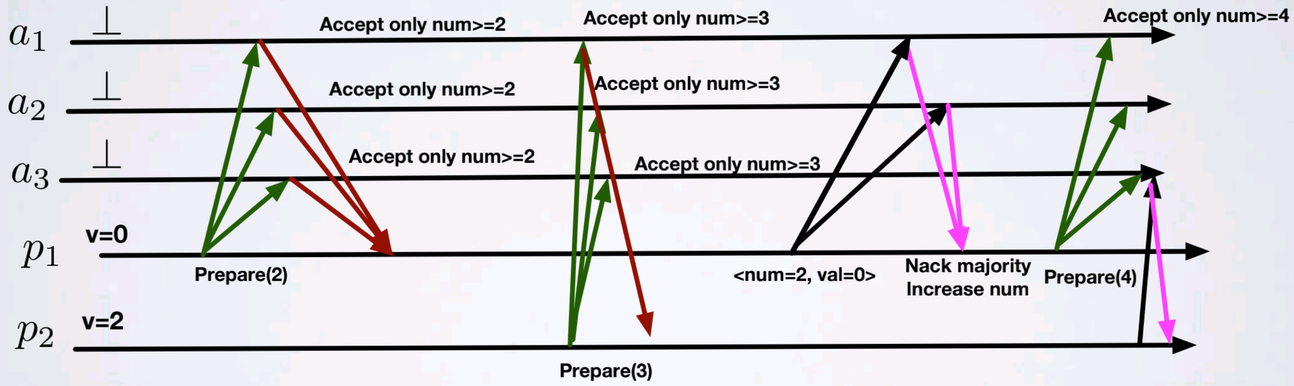
Is it there something missing? A liveness property

PAXOS

Liveness is not guaranteed (due to FLP)

- Imagine a scenario with two competing proposers

Lost messages delay the protocol but do not block it (if you use timeouts on proposers)



NACKS are not specified in the algorithm

PAXOS - QUESTION

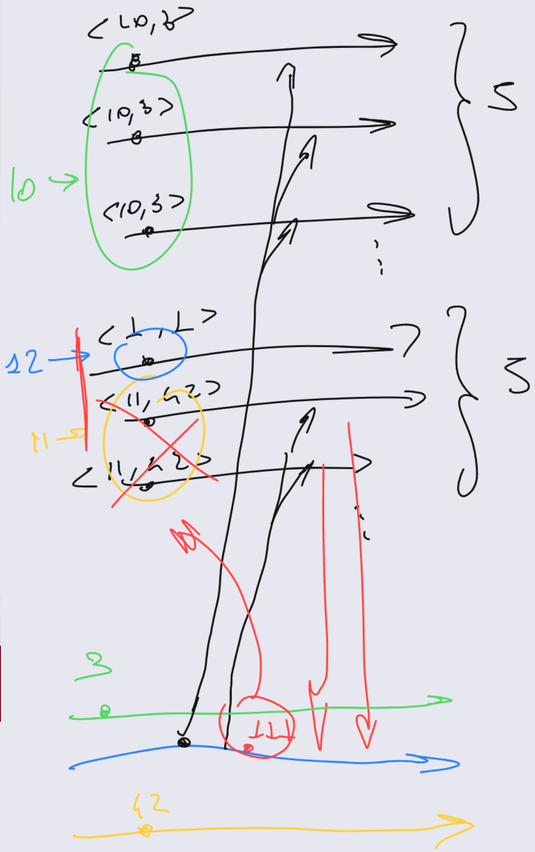
Let us suppose that we have a run of paxos where proposer P1 is able to make n/2-1 acceptors vote its proposal at a round j (a_1,a_2,.. a_{n/2-1}), and let us say that its proposal has value 3,

vote = accept

And where process P2 makes the other n/2-1 acceptors vote its proposal at round j+1 (a_{n/2}+1,… ,a_{n}) and its proposal has value 42.

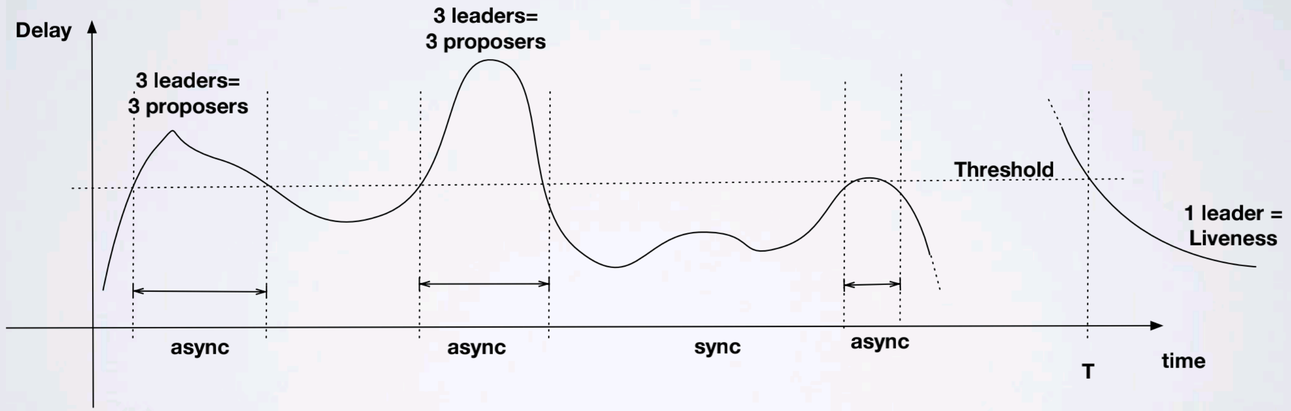
One acceptor a_{n/2} does not receive any proposal.

It is possible that a different proposers, on a round j+k, makes acceptors decides on value 7? NO



PAXOS - HOW FIX LIVENESS?

In real life what you do is to use an eventual leader elector on proposers (recall \omega), and when the system becomes synchronous you have one leader and therefore progress.



PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

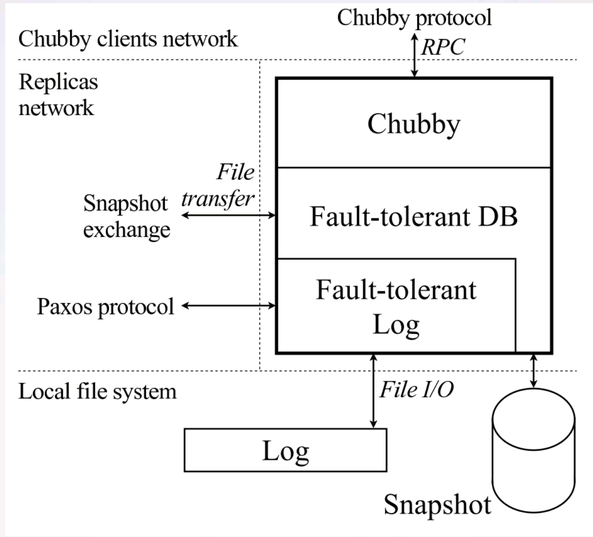


Figure 1: A single Chubby replica.

PAXOS - IN REAL LIFE read *

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.
- The fault-tolerant algorithms community is accustomed to proving short algorithms (one page of pseudo code) correct. This approach does not scale to a system with thousands of lines of code. To gain confidence in the "correctness" of a real system, different methods had to be used.
- Fault-tolerant algorithms tolerate a limited set of carefully selected faults. However, the real world exposes software to a wide variety of failure modes, including errors in the algorithm, bugs in its implementation, and operator error. We had to engineer the software and design operational procedures to robustly handle this wider set of failure modes.
- A real system is rarely specified precisely. Even worse, the specification may change during the implementation phase. Consequently, an implementation should be malleable. Finally, a system might "fail" due to a misunderstanding that occurred during its specification phase.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

4.2 Multi-Paxos

Practical systems use Paxos as a building block to achieve consensus on a *sequence of values*, such as in a replicated log. The simple way to implement this is to repeatedly execute the Paxos algorithm. We term each execution an *instance* of Paxos. We refer to *submitting* a value to Paxos (or equivalently, to the log) to mean executing an instance of Paxos while submitting that value.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

5.2 Master leases

When the basic Paxos algorithm is used to implement a replicated data structure, reads of the data structure require executing an instance of Paxos. This serializes the read with respect to updates and ensures that the *current* state is read. In particular, read operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master. In this case, the read operation at the master runs the risk of returning stale data. Since read operations usually comprise a large fraction of all operations, serializing reads through Paxos is expensive.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

5.2 Master leases

When the basic Paxos algorithm is used to implement a replicated data structure, reads of the data structure require executing an instance of Paxos. This serializes the read with respect to updates and ensures that the *current* state is read. In particular, read operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master. In this case, the read operation at the master runs the risk of returning stale data. Since read operations usually comprise a large fraction of all operations, serializing reads through Paxos is expensive.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

5.4 Group membership

Practical systems must be able to handle changes in the set of replicas. This is referred to as the group membership problem in the literature [3]. Some Paxos papers point out that the Paxos algorithm itself can be used to implement group membership [8]. While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos, disk corruptions, etc. Unfortunately the literature does not spell this out, nor does it contain a proof of correctness for algorithms related to group membership changes using Paxos. We had to fill in these gaps to make group membership work in our system. The details – though relatively minor – are subtle and beyond the scope of this paper.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

6.1 Expressing the algorithm effectively

Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. It becomes harder to see the core algorithm, to reason about it, or to debug it when a bug is present. It also makes it difficult to change the core algorithm in response to a requirement change.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

6.2 Runtime consistency checking

The chance for inconsistencies increases with the size of the code base, the duration of a project, and the number of people working simultaneously on the same code. We used various active self-checking mechanisms such as the liberal use of `assert` statements, and explicit verification code that tests data structures for consistency.

- The first incident was due to an operator error.
- We have not found an explanation for the second incident. On replaying the faulty replica's log we found that it was consistent with the other replicas. Thus it is possible that this problem was caused by a random hardware memory corruption.
- We suspect the third was due to an illegal memory access from errant code in the included codebase (which is of considerable size). To protect against this possibility in the future, we maintain a second database of checksums and double-check every database access against the database of checksums.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

6.3 Testing

Given the current state of the art, it is unrealistic to prove a real system such as ours correct. To achieve robustness, the best practical solution in addition to meticulous software engineering is to test a system thoroughly. Our system was designed to be testable from the onset and now contains an extensive suite of tests. In this section we describe two tests that take the system through a long sequence of random failures and verify that it behaves as expected. Both tests can run in one of two modes:

1. **Safety mode**. In this mode, the test verifies that the system is consistent. However, the system is not required to make any progress. For example, it is acceptable for an operation to fail to complete or to report that the system is unavailable.

2. **Liveness mode**. In this mode, the test verifies that the system is consistent and is making progress. All operations are expected to complete and the system is required to be consistent.

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

7 Unexpected failures

So far, our system has logged well over 100 machine years of execution in production. In this period we have witnessed the following unexpected failure scenarios:

- When we tried to upgrade this Chubby cell again a few months later, our upgrade script failed because we had omitted to delete files generated by the failed upgrade from the past. The cell ended up running with a months-old snapshot for a few minutes before we discovered the problem. This caused us to lose about 30 minutes of data. Fortunately all of Chubby's clients recovered from this outage.
- A few months after our initial release, we realized that the semantics provided by our database were different from what Chubby expected. If Chubby submitted an operation to the database, and the database lost its master status, Chubby expected the operation to fail. With our system, a replica

PAXOS - IN REAL LIFE

Paxos Made Live - An Engineering Perspective 2007

Tushar Chandra, Robert Griesemer, Joshua Redstone.

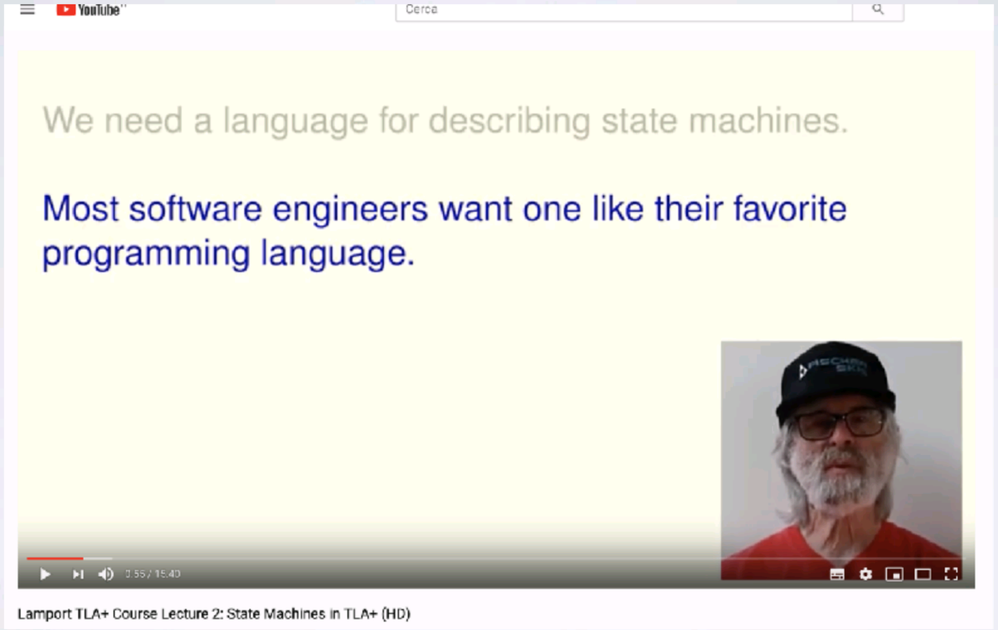
Conclusions

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.
- The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms.
- The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems.

HOW TO PROVE NON-TRIVIAL CORRECTNESS?

HOW TO PROVE NON-TRIVIAL CORRECTNESS?

HOW TO PROVE NON-TRIVIAL CORRECTNESS?



HOW TO PROVE NON-TRIVIAL CORRECTNESS?

Dharma Shukla, Microsoft Technical Fellow, writes this about TLA+ use in Azure Cosmos DB, Microsoft's globally distributed database service:

> The Cosmos DB engineering team have been using TLA+ for specifying and validating the correctness of the core algorithms as well as the high-level specifications of the five consistency models that the system offers to our customers. We have found TLA+ extremely useful in two fundamental ways:

https://lamport.azurewebsites.net/tla/industrial-use.html