第9讲 Agreement Protocols

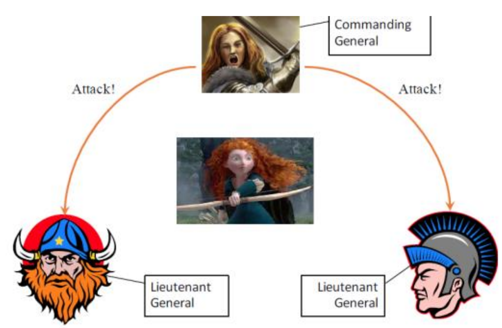
- Paxos：基本的consensus协议
- Raft： 高效的consensus协议
- BFT： 基本的Byzantine协议
- PBFT： 高效的Byzantine协议

共识协议分类

泛洪　　Paxos　Raft　ZAB　BFT　PBFT　PoW

准确检测失效　　能最终检测失效　　拜占庭容错（有恶意行为）

仅“停止失效”容错



拜占庭共识问题（Byzantine agreement)
--L. Lamport, 1982

Paxos

L. Lamport, The Part-time Parliament, ACM Transactions on Computer Systems, 1998.

希腊岛屿Paxon上的执法者在议会大厅中表决通过法律，并通过服务员传递纸条的方式交流信息，每个执法者会将通过的法律记录在自己的账本上。

执法者和服务员都不可靠，他们随时会因为各种事情离开议会大厅，并随时可能有新的执法者进入议会大厅进行法律表决。

问题：使用何种方式能够使得这个表决过程正常进行，且通过的法律不发生矛盾。

Paxos System Model

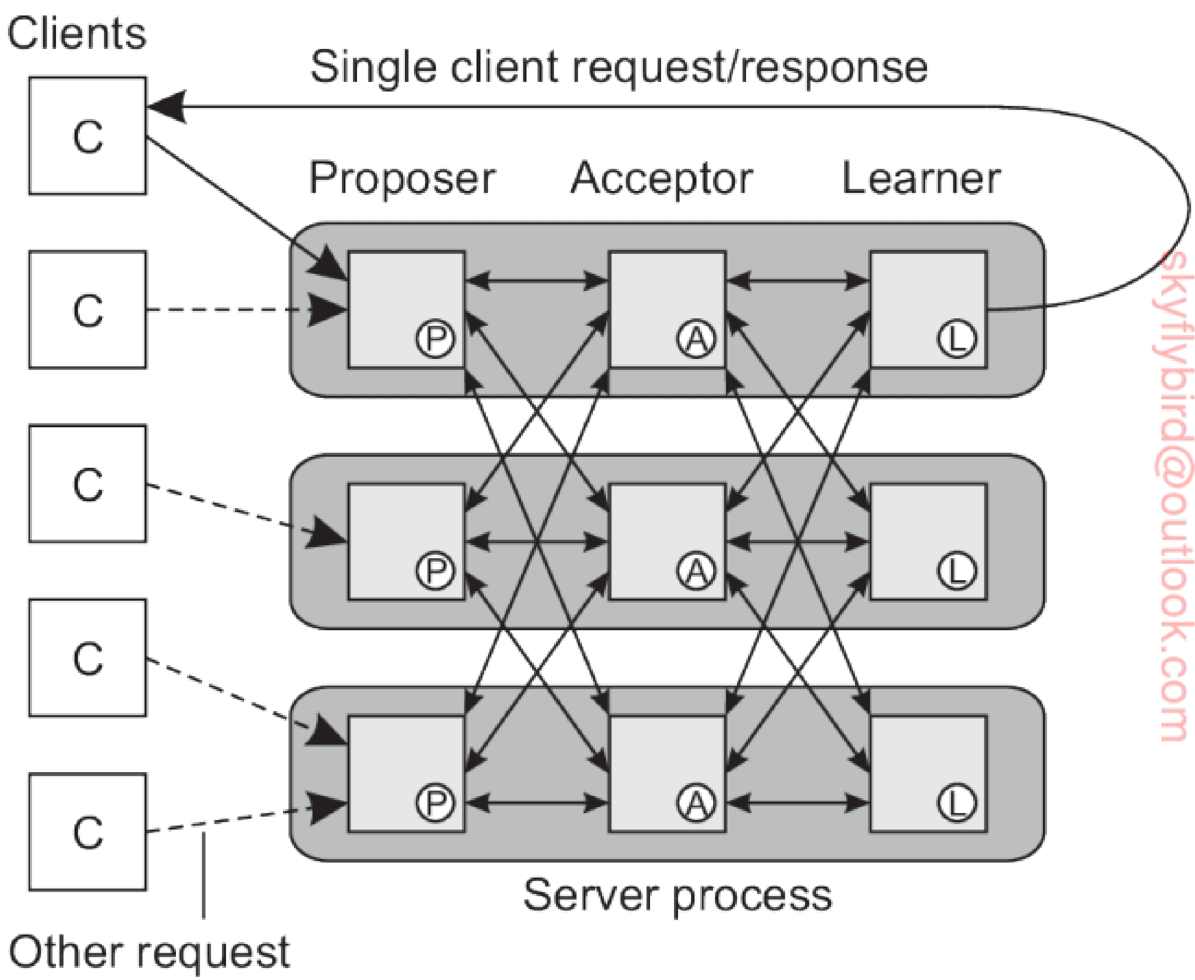
Most of the time it behaves as a synchronous system, yet there is no bound on the time that it behaves in an asynchronous fashion.

- The assumptions under which Paxos operates are rather weak:
  - The system is partially synchronous (in fact, even asynchronous).
  - Communication between processes may be unreliable: messages may be lost, duplicated, or reordered.
  - Messages that are corrupted can be detected as such (and thus subsequently ignored).
  - All operations are deterministic: once an execution is started, it is known exactly what it will do.
  - Processes may exhibit crash failures, but not arbitrary failures, nor do processes collude.

Paxos Players

- Proposer
  – Suggests values for consideration by acceptors.
- Acceptor
  – Considers the values proposed by proposers.
  – Renders an accept/reject decision.
- Learner
  – Learns the chosen value and execute operations accordingly.
- A node can act as more than one roles (usually 3).

Paxos Components



The organization of Paxos into different logical processes.

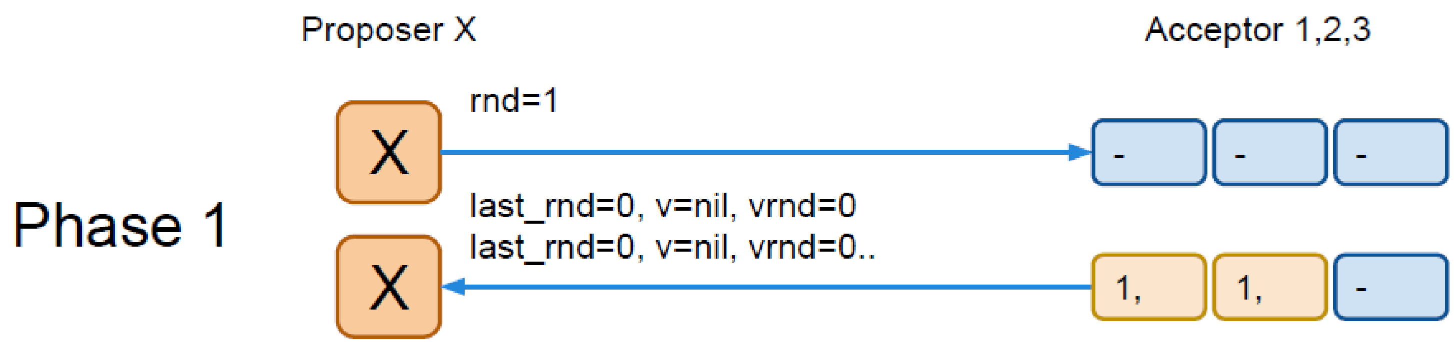
Paxos的主要符号

- Round: 按轮次执行，每一轮包含3个阶段（Phase）。
- 轮编号rnd：单调增; 后写胜出; 全局唯一。
- last_rnd：一个Acceptor看到的最大rnd。

  //Acceptor记住这个值来识别哪个proposer可以写。
- v：一个Acceptor接受的值。
- vrnd：Acceptor接受v的时候的rnd

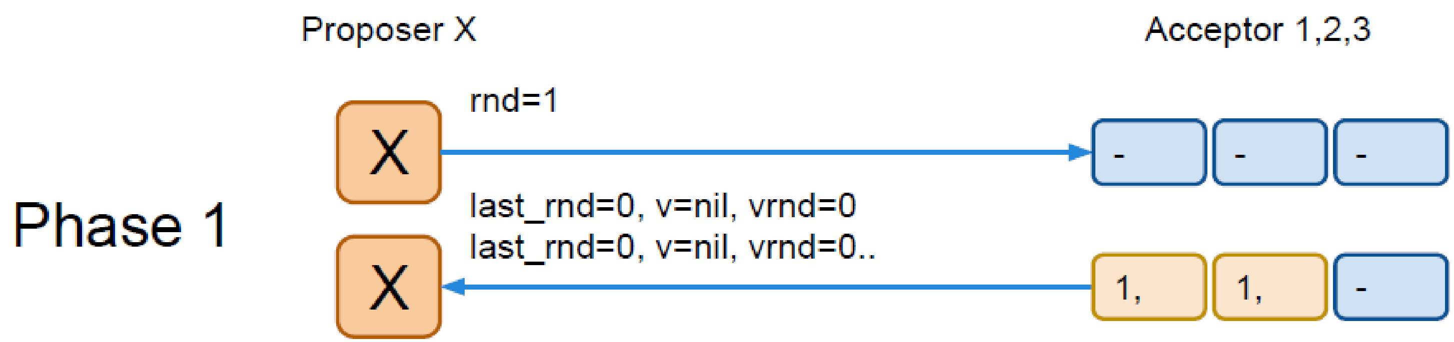
一个值v被确定（达成共识）：被大多数的Acceptor接受。

Paxos阶段1a



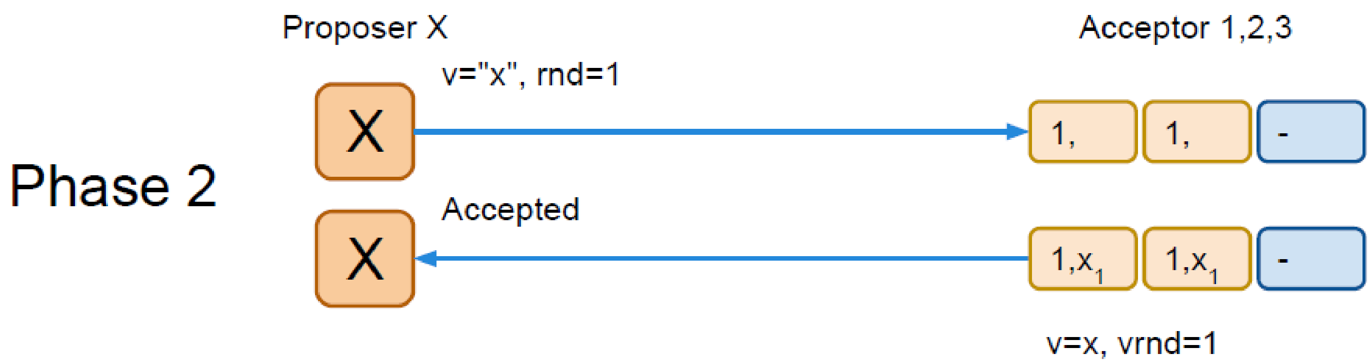
- Proposer:
  - 增加自己的rnd，发送prepare消息，带上自己的rnd；
- Acceptor：收到prepare请求，
  - 如果请求中rnd比Acceptor的last_rnd小，则拒绝请求；
  - 否则，将请求中的rnd保存到本地的last_rnd；
    - 从此这个Acceptor只接受带有这个last_rnd的phase2请求；
  - 返回promise消息，带上自己之前的last_rnd和之前已接受的v。

Paxos阶段1b



- 当Proposer收到Acceptor的应答:
  - 如果应答中的last_rnd大于发出的rnd: 退出。
  - 从所有应答中选择vrnd最大的v：不能改变（可能）已确定的值。
  - 如果所有应答的v都是空，可以选择自己要写入v。
  - 如果应答不够多数派，退出。

Paxos阶段2a

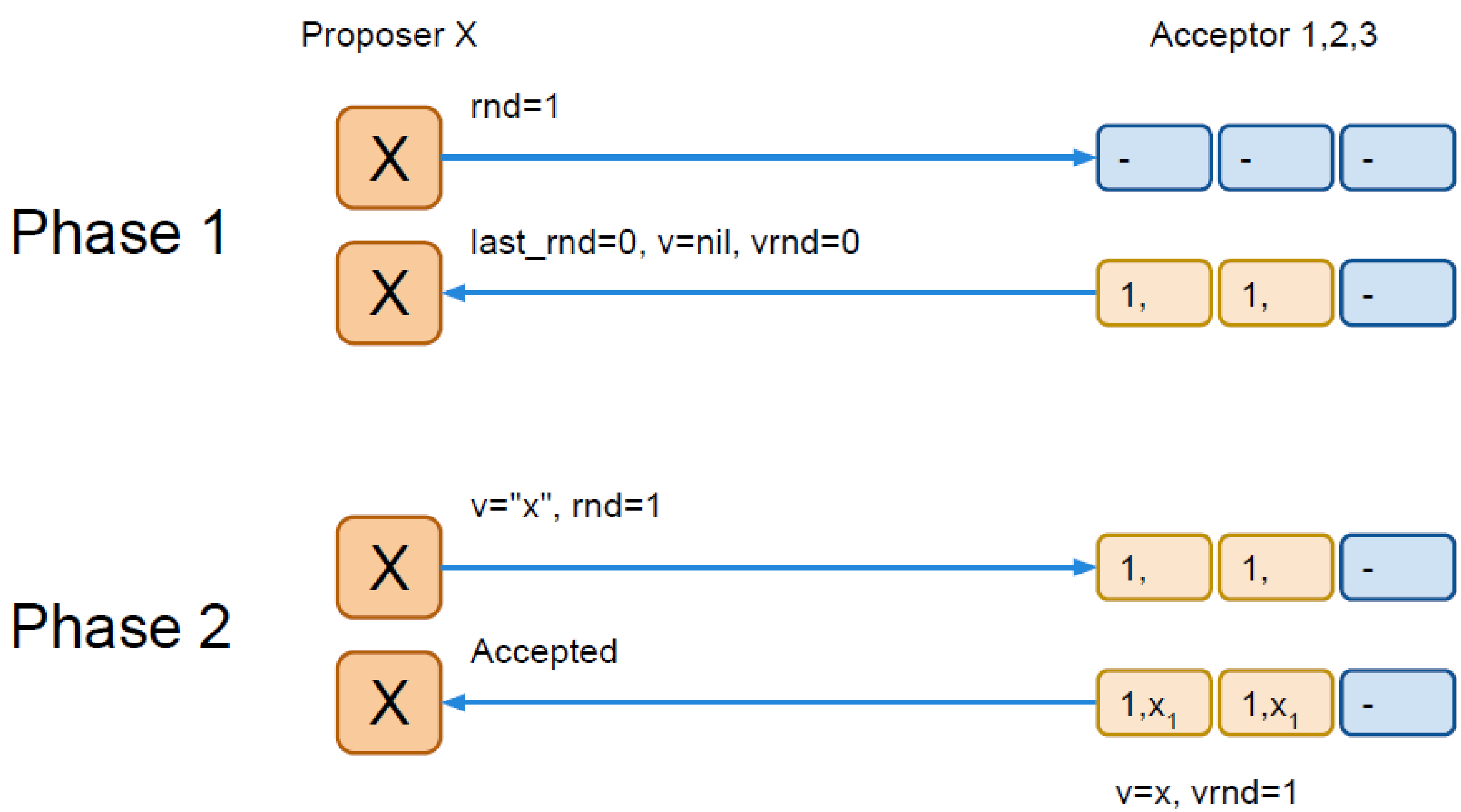


- Proposer：
  - 发送Accept消息，带上 rnd 和上一步选择的 v。
- Acceptor：
  - 拒绝 rnd 不等于自己的 last_rnd的请求（已经promise更大rnd）；
  - 将 Accept中的 v 写入本地，记此 v 为“已接受的值”；
  - last_rnd==rnd 确保没有其他 Proposer 在此过程中写入过其他值。

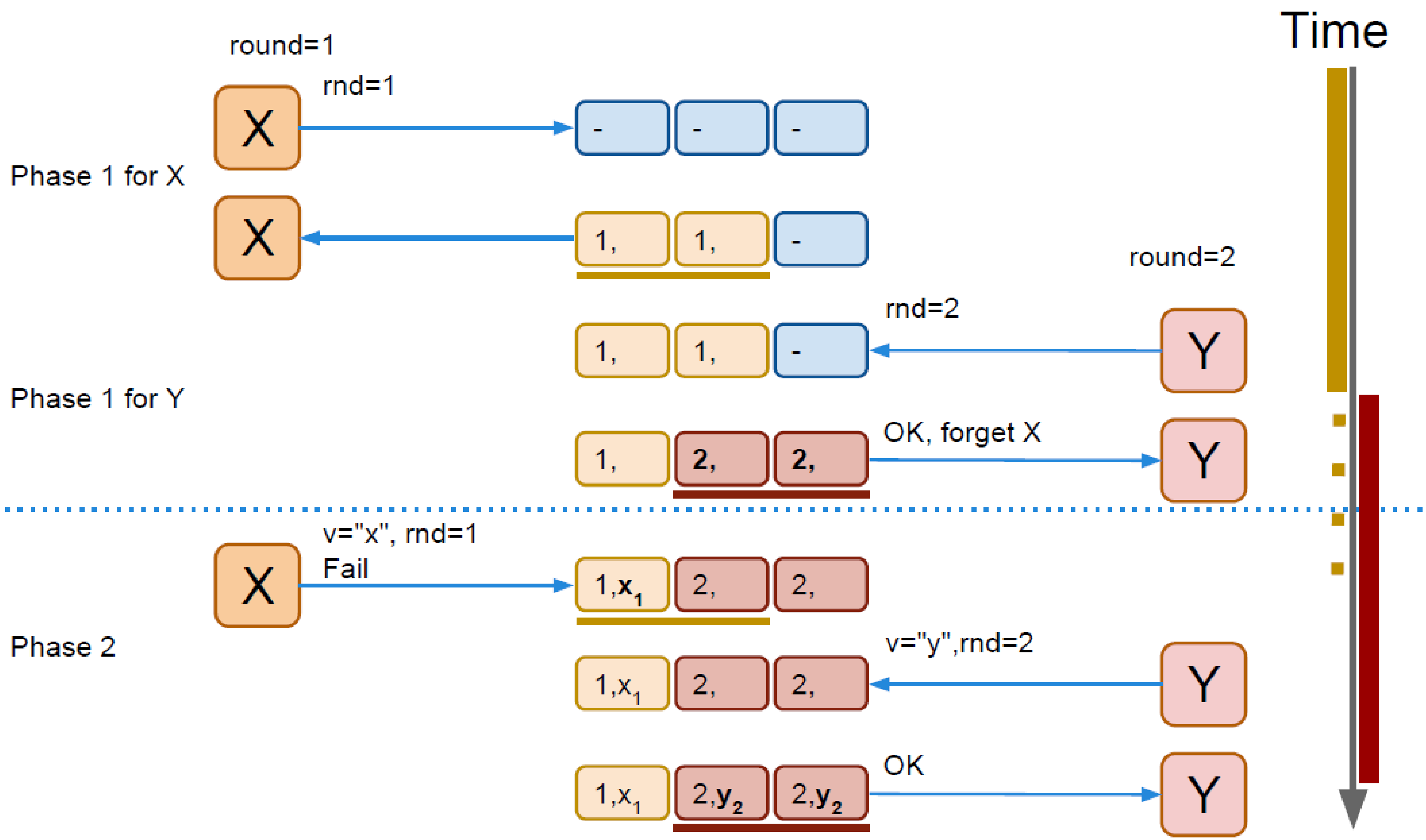
Paxos阶段2b

- 每个Acceptor 发送 Learn消息到所有 Learner;
- 当一个Learner收到大多数Acceptor的Learn消息，知道一个值被确定了。
- 多数场合下 Proposer 就是一个 Learner。

例子1：无冲突

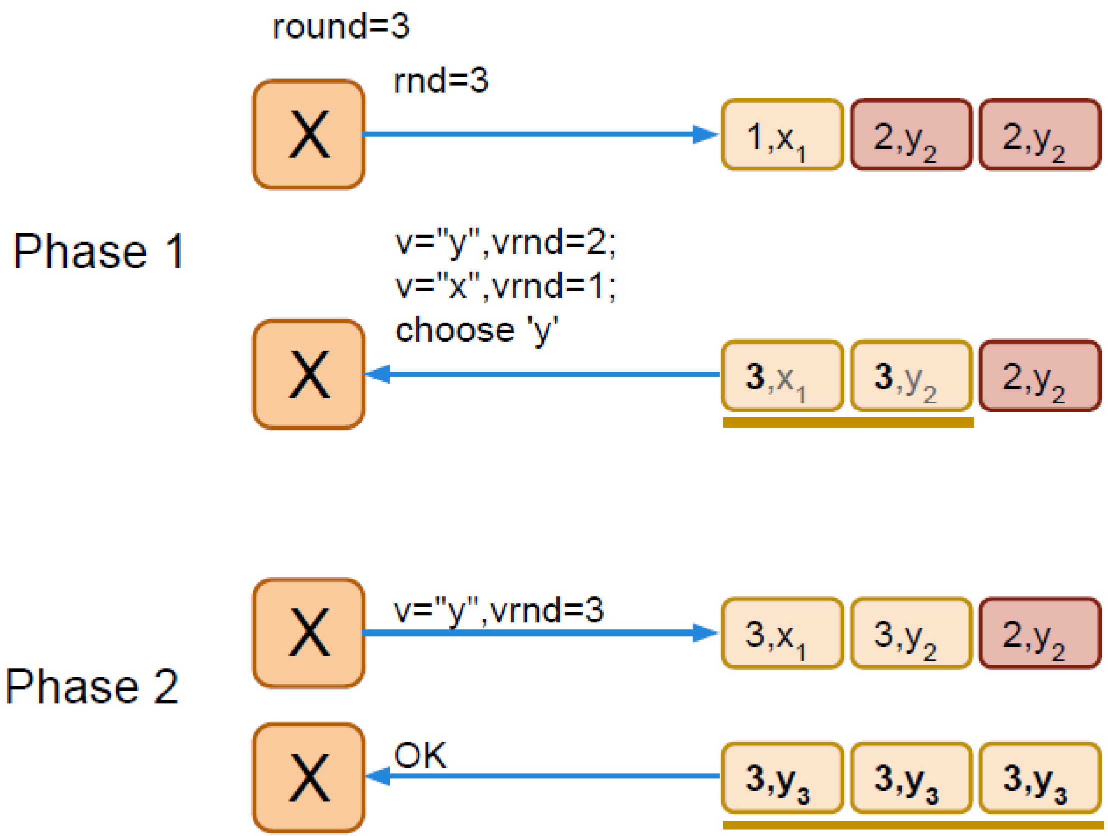

Proposer X
Acceptor 1,2,3
Phase 1
rnd=1
X
-
-
-
last_rnd=0, v=nil, vrnd=0
X
1,
1,
-
Phase 2
v="x", rnd=1
X
1,
1,
-
Accepted
X
1,x1
1,x1
-
v=x, vrnd=1

例子2：解决并发写冲突



round=1
rnd=1
Phase 1 for X
X
- - -
1, 1, -
round=2
rnd=2
Y
Phase 1 for Y
1, 2, 2,
OK, forget X
Time
v="x", rnd=1
Fail
1,x1 2, 2,
Phase 2
v="y",rnd=2
1,x1 2,y2 2,y2
OK

例子3：X 不会修改确定的 V



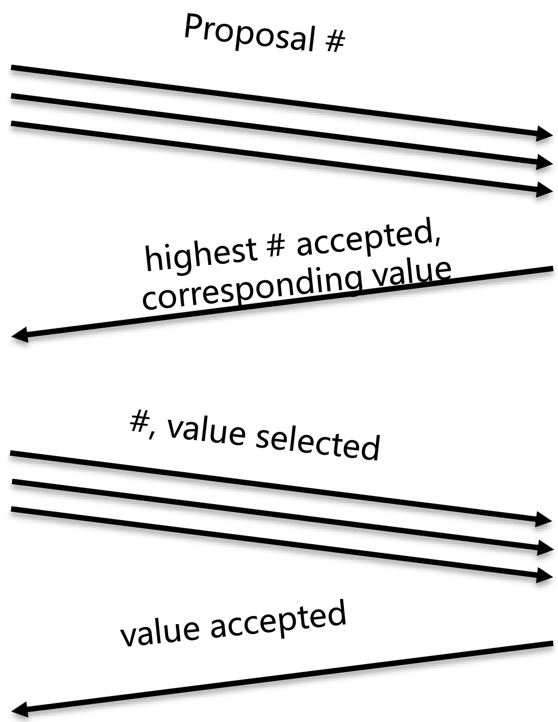
X 只能选择 v= “y”，因为它可能是一个被确定的值

Paxos概括

Proposers

Acceptors

Choose unique
proposal #



proposal # > any previous?

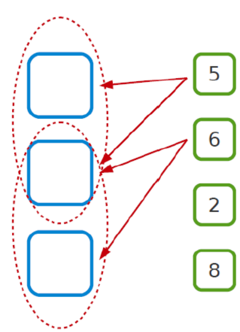
Majority?
Select value for highest
proposal # returned;
If none, choose own value

proposal # >= any previous?

Majority? Value decided

Paxos正确性 – Safety

- Property:
  – If a value v is chosen at proposal number n, any value sent out in phase2 of any later proposal numbers must be also v.
- Decision = Majority
  – Any two majorities share at lease one element
- Safety holds:
  – Therefore after the round in which there is a decision, any subsequent round involves at least one acceptor that has accepted v.



Paxos正确性 – Safety

- Proof (by contradiction):
  - Suppose safety is not true
  - Let *m* be the first proposal number that is later than *n* and in Phase2, the value sent out is $w \neq v$
- This is not possible, because
  - If the proposal *P* was able to start Phase2 for *w*, it means: a majority to accept round for *m* (for $m > n$).
- So, either:
  - *v* would not have been the value decided, or
  - *v* would have been proposed by *P* (i.e., $w == v$).
- Therefore, once a majority accepts *v*, that never changes.

Paxos正确性 – Liveness

If two or more proposers race to propose new values, they might step on each other toes all the time.

- *P1* : prepare(*n1*)
- *P2* : prepare(*n2*)
- *P1* : accept(*n1, v1*)
- *P2* : accept(*n2, v2*)
- *P1* : prepare(*n3*)
- *P2* : prepare(*n4*)
- ...
- $n1 < n2 < n3 < n4$

Livelock：

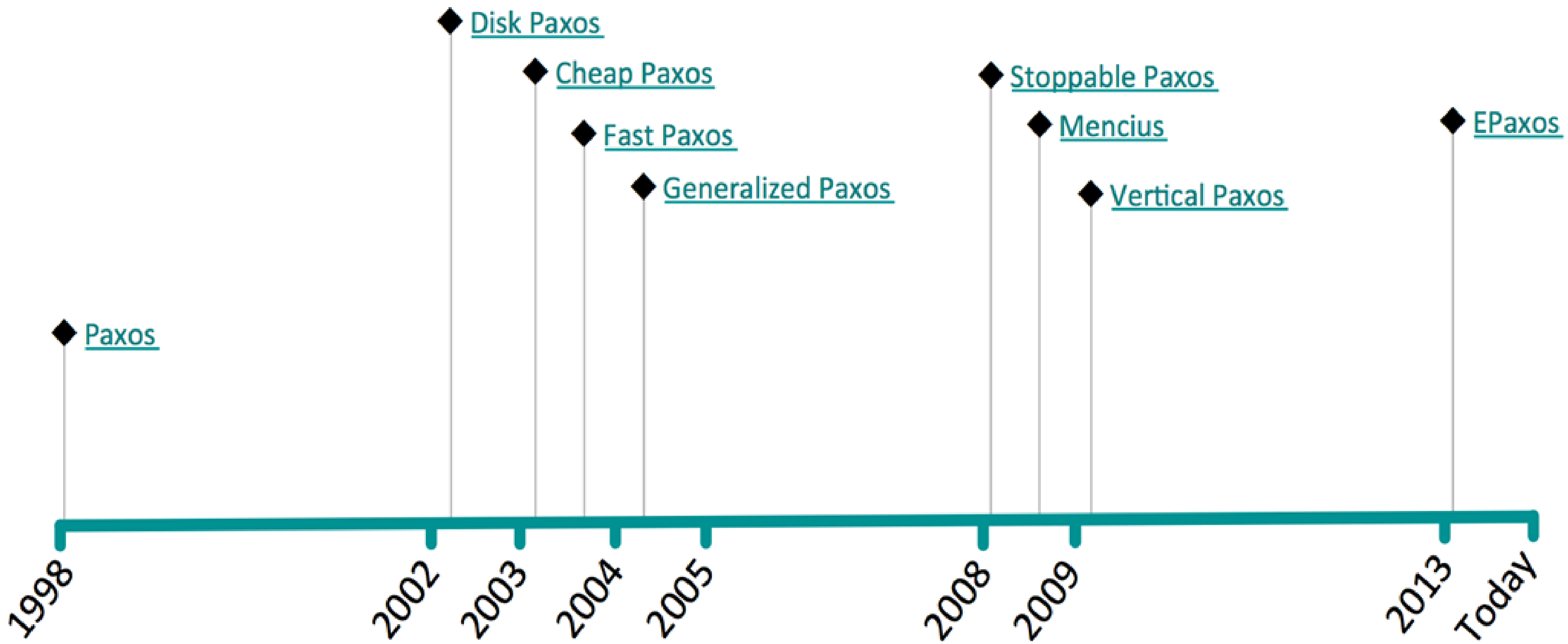
多个 Proposer 并发对 1 个值运行 Paxos 的时候，可能会互相覆盖对方的 rnd，然后提升自己的 rnd 再次尝试，然后再次产生冲突，一直无法完成。

With randomness, this occurs exceedingly rarely.

Paxos – leader based version

- A single proposer can be elected as the leader:
  - receives requests from clients (or forwarded by other proposers).
  - increments and associates a unique round number with every request.
  - sends its proposal to all acceptors, telling each to accept
    the requested operation.
- Benefit:
  - Largely reduce concurrent proposal and livelock.
- Problem:
  - Additional election mechanism;
  - Due to asynchrony, multiple leaders may co-exist, still need to handle concurrent proposals.

Paxos变种



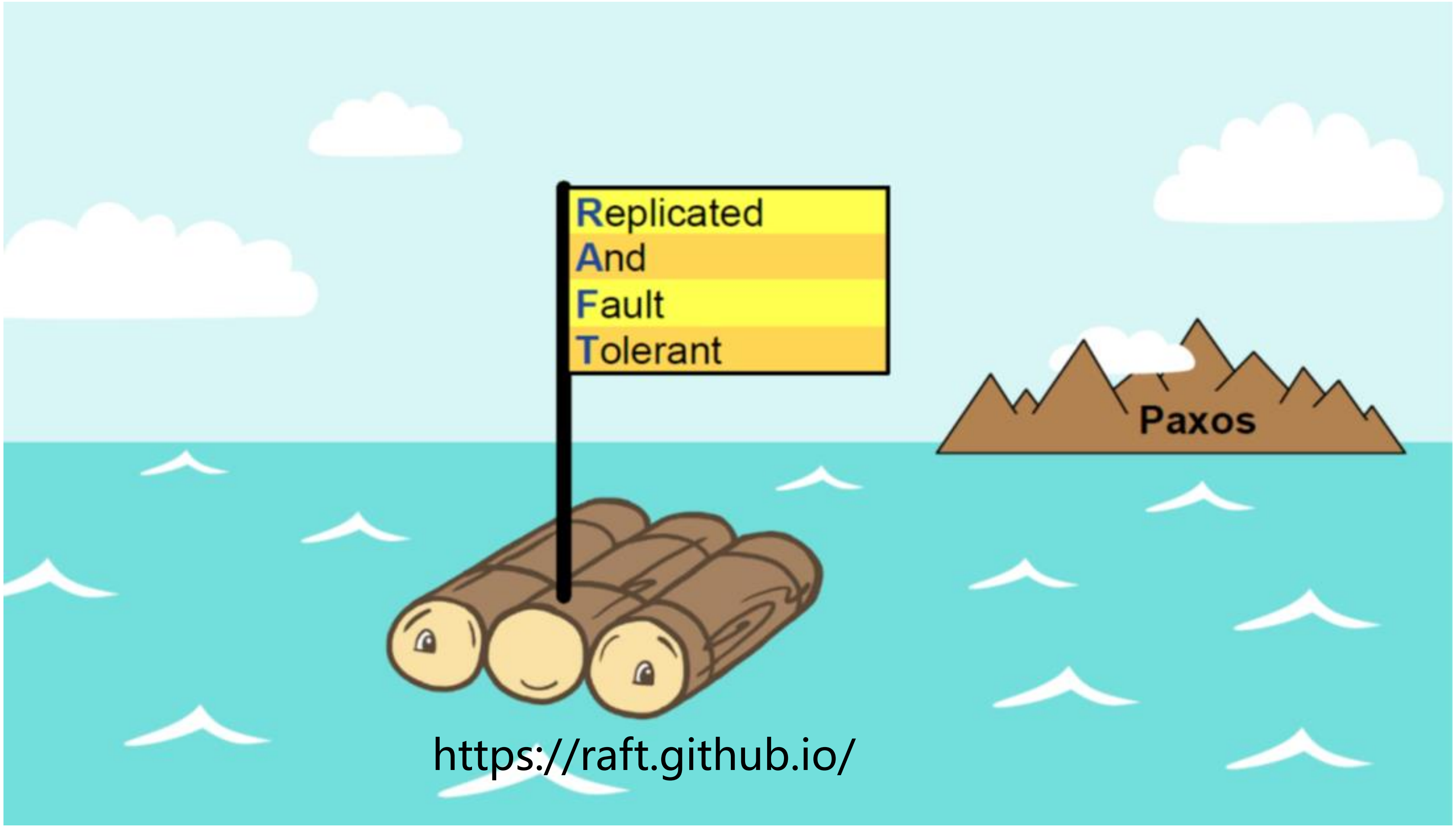
http://paxos.systems/variants/

Problems of Paxos

- Impenetrable: hard to develop intuitions
  - Why does it work?
  - What is the purpose of each phase?
- Incomplete
  - Only agrees on single value
  - Doesn’t address liveness
  - Choosing proposal values?
  - Clustering membership management?
- Inefficient
  - Two rounds of messages to choose one value
- No agreement on the details
- Not a good foundation for practical implementations

> “The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos :-)”
> -- NSDI reviewer

RAFT (Replicated And Fault Tolerant)



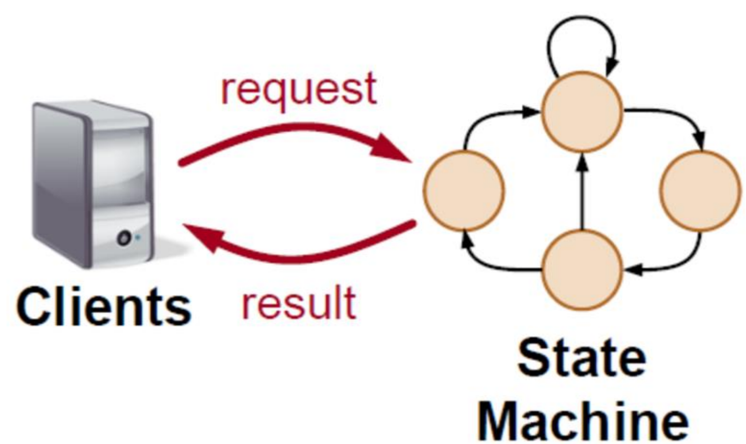
https://raft.github.io/

RAFT

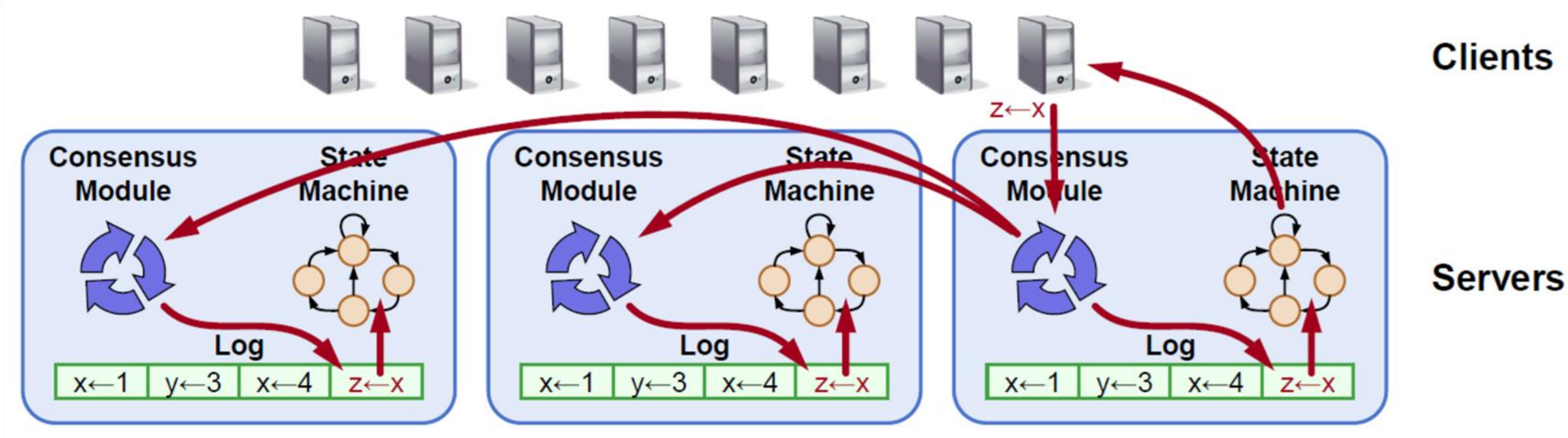
- Paxos is:
  - Hard to understand
  - Not complete enough for real implementations
- New consensus algorithm: Raft
  - Primary design goal: understandability (ease of explanation)
  - Complete foundation for implementation
  - Different problem decomposition
- Results:
  - User study show Raft more understandable than Paxos
  - Widespread adoption

State Machine

- Responds to external stimuli
- Manages internal state
- Examples: many storage systems, services
  - Memcached
  - RAMCloud
  - HDFS name node
  - …



Replicated State Machine

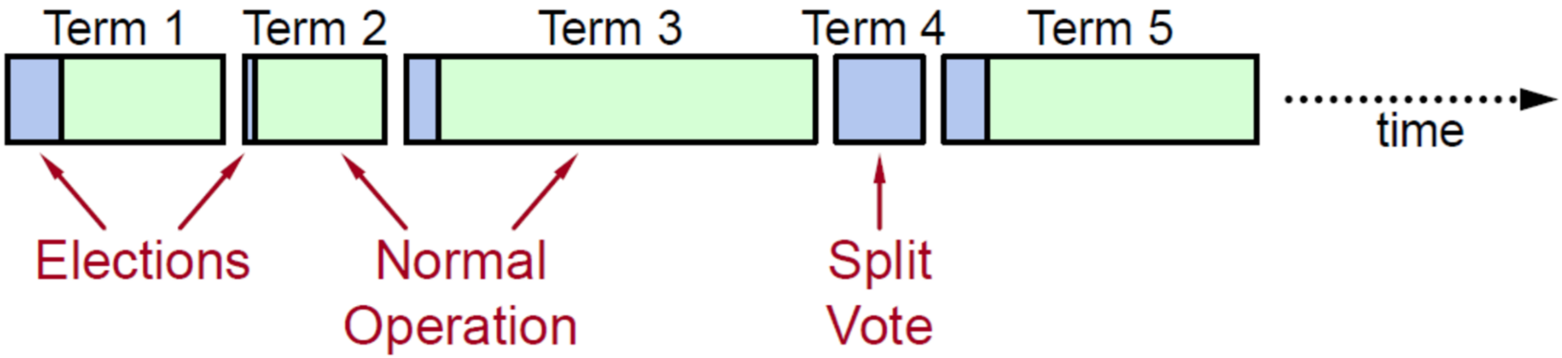


- Replicated log ensures state machines execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: delayed/lost messages, fail-stop (not Byzantine)

Raft Methodology

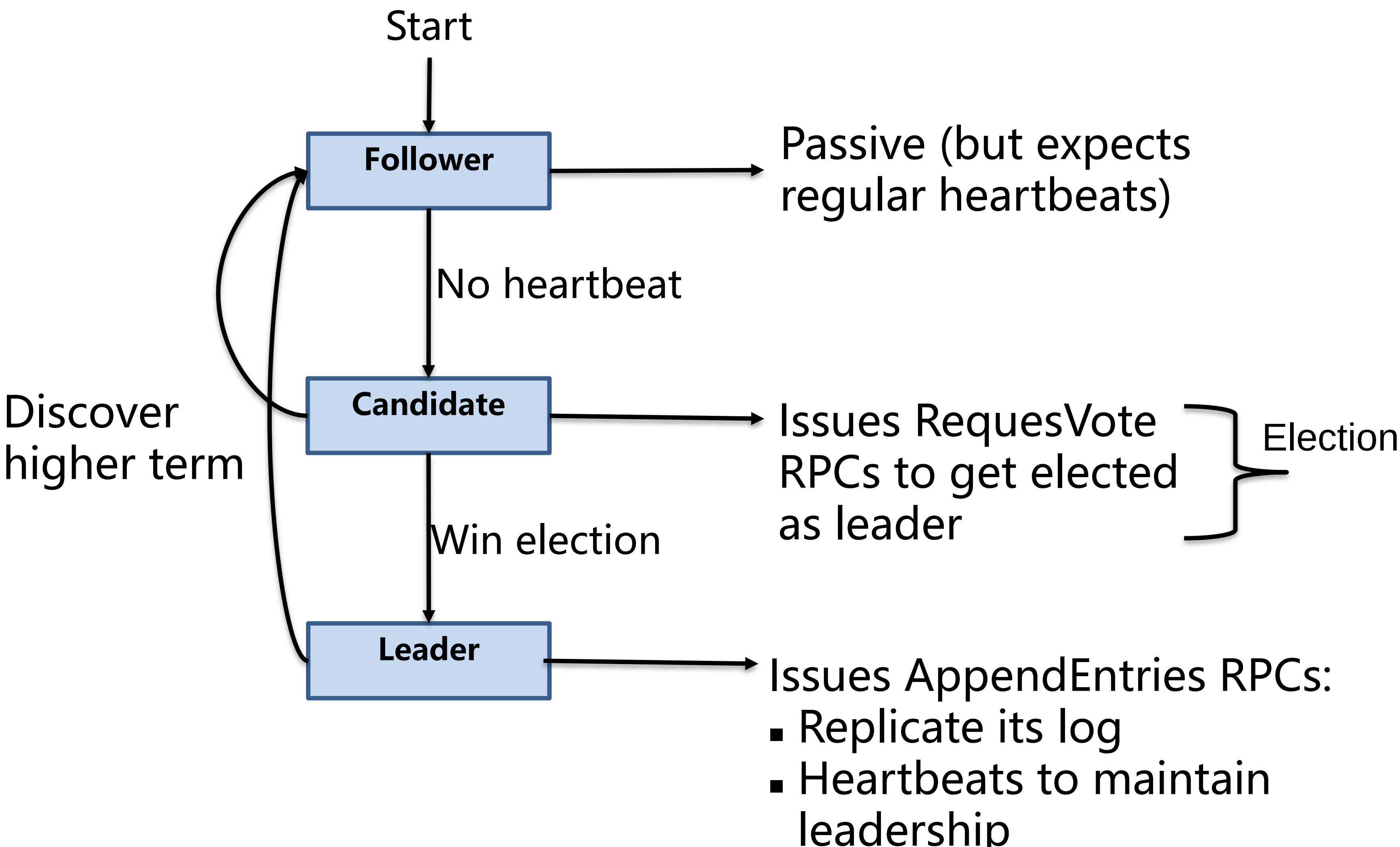
- Leader election
  - Select one server to act as leader
  - Detect crashes, choose new leader
- Log replication (normal operation)
  - Leader accepts commands from clients, appends to its log
  - Leader replicates its log to other servers (overwrites inconsistencies)
- Safety
  - Keep logs consistent
  - Only servers with up-to-date logs can become leader

Terms

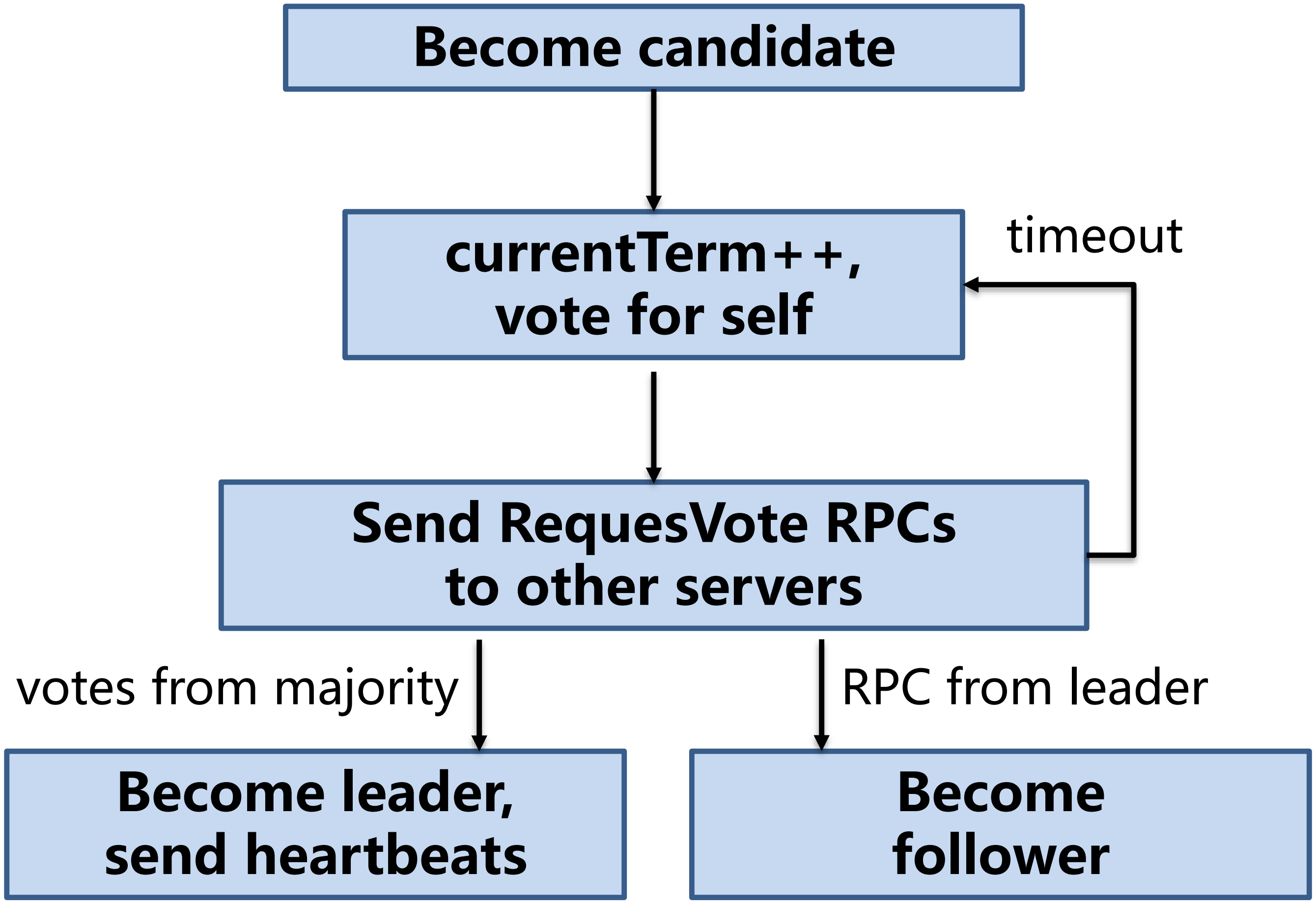


- At most 1 leader per term
  - Some terms have no leader (failed election)
- Each server maintains current term value (no global view)
  - Exchanged in every RPC
  - Peer has later term? Update term, revert to follower
  - Incoming RPC has obsolete term? Reply with error
- Terms identify obsolete information

Server States and RPCs

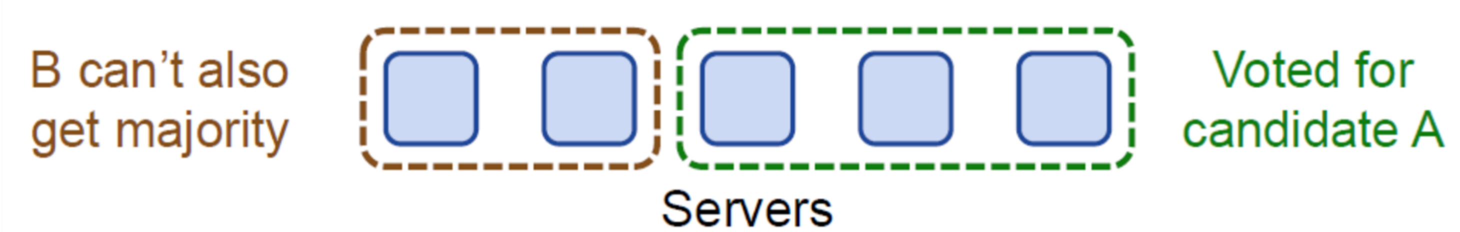


Leader Election



Election Correctness

- Safety: allow at most one winner per term
  - Each server gives only one vote per term (persist on disk)
  - Majority required to win election



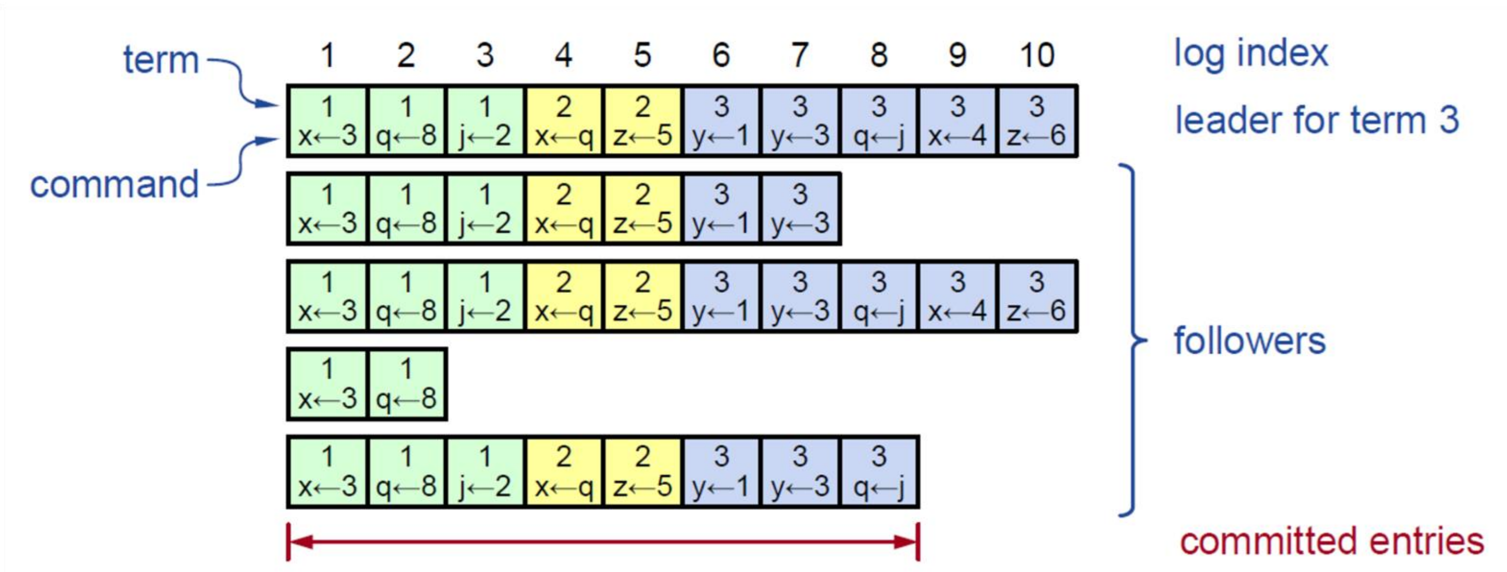
- Liveness: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T] (e.g. 150-300 ms)
  - One server usually times out and wins election before others timeout
  - Works well if T ≫ broadcast time
- Randomized approach simpler than ranking

Normal Operation – Log Replicating

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to all followers
- Once new entry committed:

  (replicated on a majority of servers)
  - Leader executes command, returns result to client
  - Leader includes the highest committed index in all later AppendEntries
  - Followers execute committed commands
- Crashed/slow followers?
  - Leader retries AppendENtries RPCs until they succeed
- Performance improvement in common case:
  - One successful RPC to any majority of servers

Normal Operation – Log Replicating

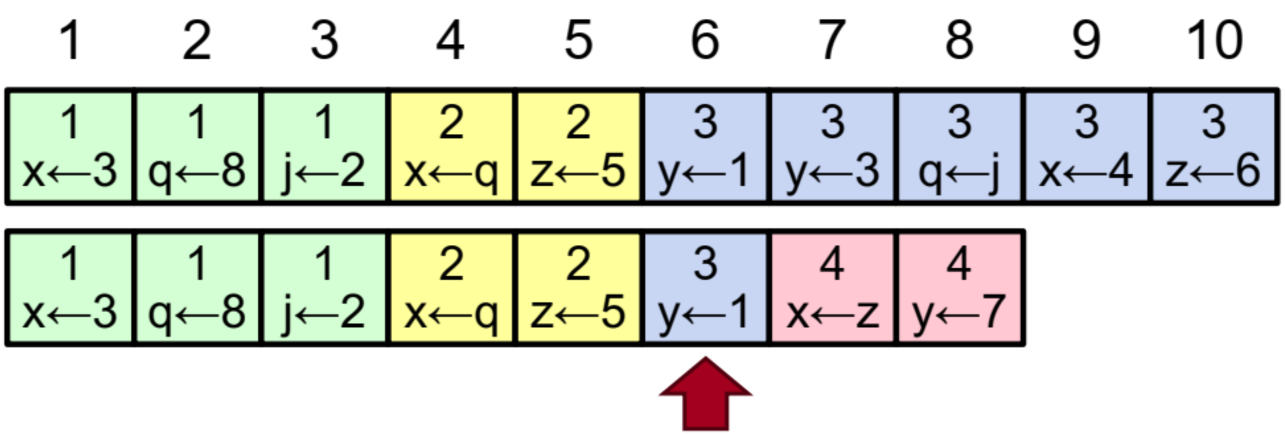


- Must survive crashes (store on disk)
- Entry committed if safe to execute in state machines
  - Replicated on majority of servers by leader of its term

Log Matching Property

Goal: high level of consistency between logs

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries



- If a given entry is committed, all preceding entries are also committed

Log Matching Property

- Ensuring property S1
  (same <index,term> -> same command)
  – Leader creates at most one entry at a given index in a term
  – This is sent to all the followers
- Property S2:
  (same <index,term> ->All previous match)
  – In <AppendEntries>, leader sends <index,term> of the previous entry in its log.
  – If the follower finds the previous entry doesn't matching, it refuses to accept the message
  – Ensures the Log Matching property by induction

Log Inconsistencies

Crashes can result in log inconsistencies, then:

Raft forces followers to replicate the leader’ s logs

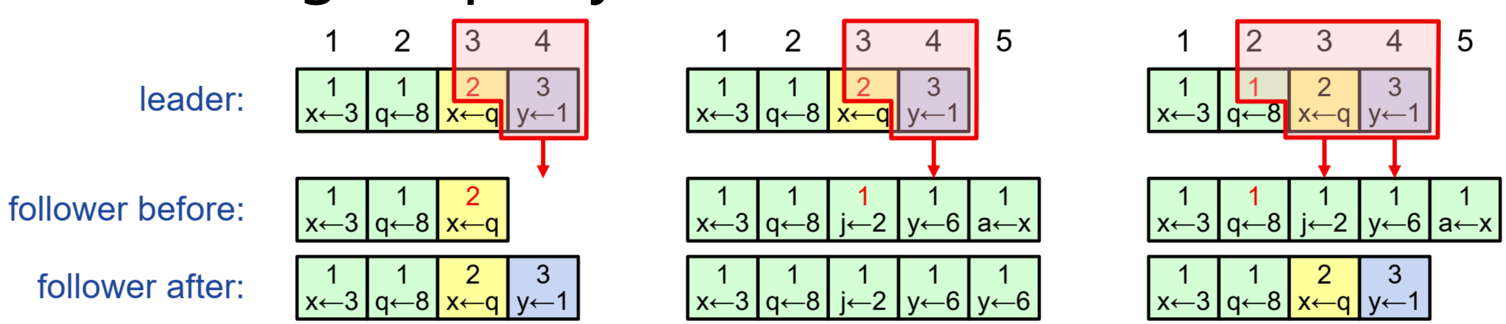
(Leader assumes its log is correct,

never overwrites or deletes entries in its own log.)

	1	2	3	4	5	6	7	8	9	10	log index
s_1	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3	3 q←j			leader for term 4
s_2	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3				followers
s_3	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3	3 q←j	3 x←4		followers
s_4	1 x←3	1 q←8									followers
s_5	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	2 y←3	2 q←j	2 x←8	2 x←4		followers

AppendEntries Consistency Check

- AppendEntries include <index, term>of entry preceding new one
- Follower must contain matching entry;
- Otherwise it rejects request:
  - Leader retries with lower log index;
    - Ultimately the logs match.
    - Follower appends all remaining entries from leader’s log.
- Implements an induction step, ensures Log Matching Property



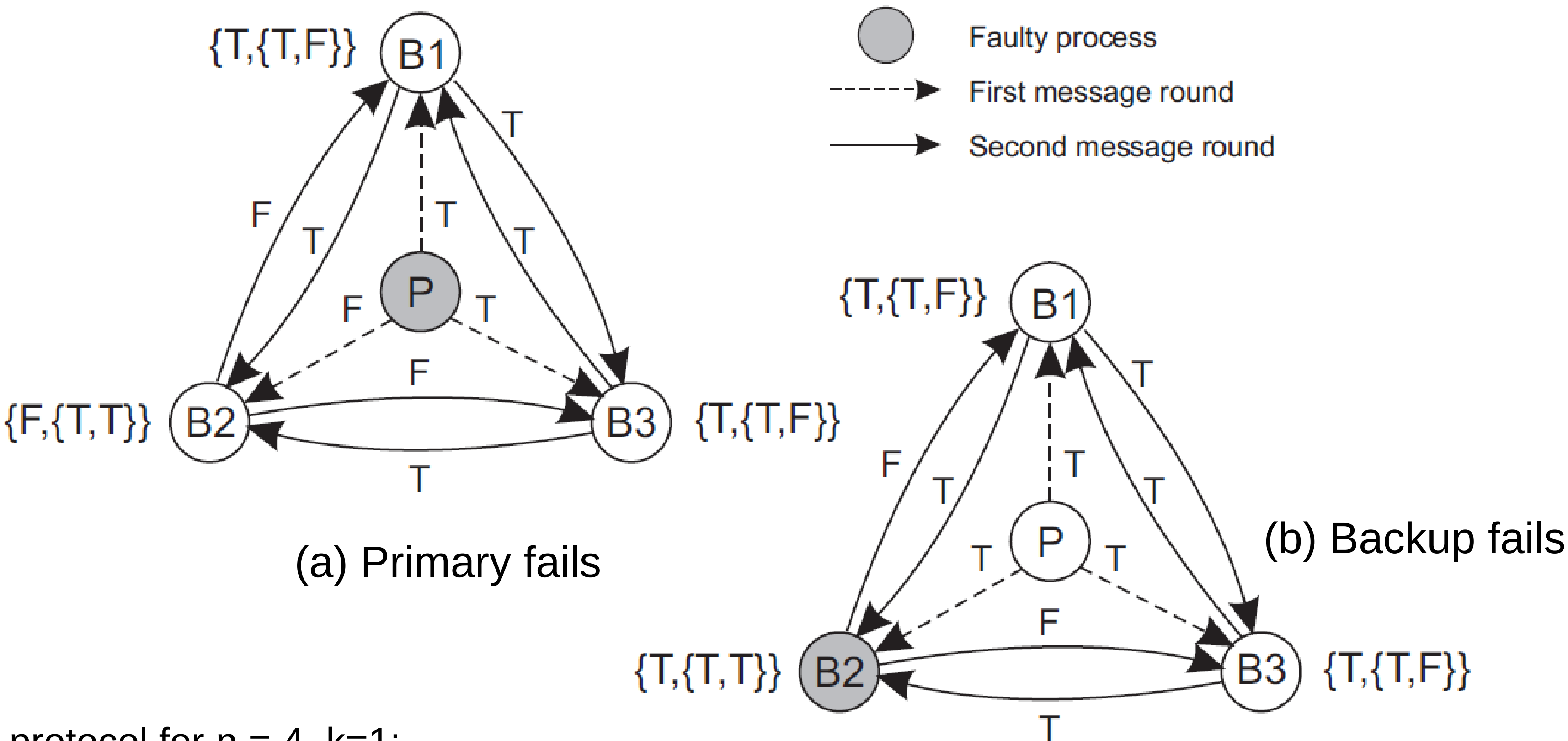
Example #1: success Example #2: mismatch Example #3: success

BFT

- 基本的拜占庭容错协议

L. Lamport, R. Shostak, and M. Pease, The Byzantine Generals Problem, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 382-401.

BFT协议



The protocol for n = 4, k=1:

1. P broadcasts command to backups.
2. Each backup rebroadcasts command from P to one another.
3. When all three messages arrive, each subordinate takes the majority decision to be the final decision.

BFT协议

System model

- We consider a primary P and $n-1$ backups $B_1, \ldots, B_{n-1}$.
- A client sends $v \in \{T, F\}$ to P
- Messages may be lost, but this can be detected.
- Messages cannot be corrupted beyond detection.
- A receiver of a message can reliably detect its sender.

同步系统！

Byzantine agreement: requirements

BA1: Every nonfaulty backup process stores the same value.

BA2: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.

Observation

- Primary faulty $\Rightarrow$ BA1 says that backups may store the same, but different (and thus wrong) value than originally sent by the client.
- Primary not faulty $\Rightarrow$ satisfying BA2 implies that BA1 is satisfied.

递归形式的BFT协议

f要已知？

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
Oral_Msg$(v, Dests, List, faulty)$, where
v is a boolean,
Dests is a set of destination process ids to which the message is sent,
List is a list of process ids traversed by this message, ordered from most recent to earliest,
faulty is an integer indicating the number of malicious processes to be tolerated.

Oral_Msg(f), where $f > 0$:

1. The algorithm is initiated by the Commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.
2. **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j, the process i uses the value v_j it receives from the source, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)
   To act as a new source, the process i initiates *Oral_Msg*$(f'-1)$, wherein it sends
   $OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f'-1))$
   to destinations not in $concat(\langle i \rangle, L)$
   in the next round.
3. **[Recursion folding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in Step 2, each process i has computed the agreement value v_k, for each k not in *List* and $k \neq i$,corresponding to the value received from P_k after traversing the nodes in *List*, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process i then uses the value $majority_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

Oral_Msg(0):

1. **[Recursion unfolding:]** Process acts as a source and sends its value to each other process.
2. **[Recursion folding:]** Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

A. Kshemkalyani and M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge press, 2008

迭代形式的BFT协议

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
tree of boolean:

- level 0 root is v^L_{init}, where $L = \langle\rangle$;
- level $h(f \geq h > 0)$ nodes: for each v^L_j at level $h - 1 = sizeof(L)$, its $n - 2 - sizeof(L)$ descendants at level h are $v_k^{concat(\langle j\rangle, L)}$, $\forall k$ such that $k \neq j$, i and k is not a member of list L.

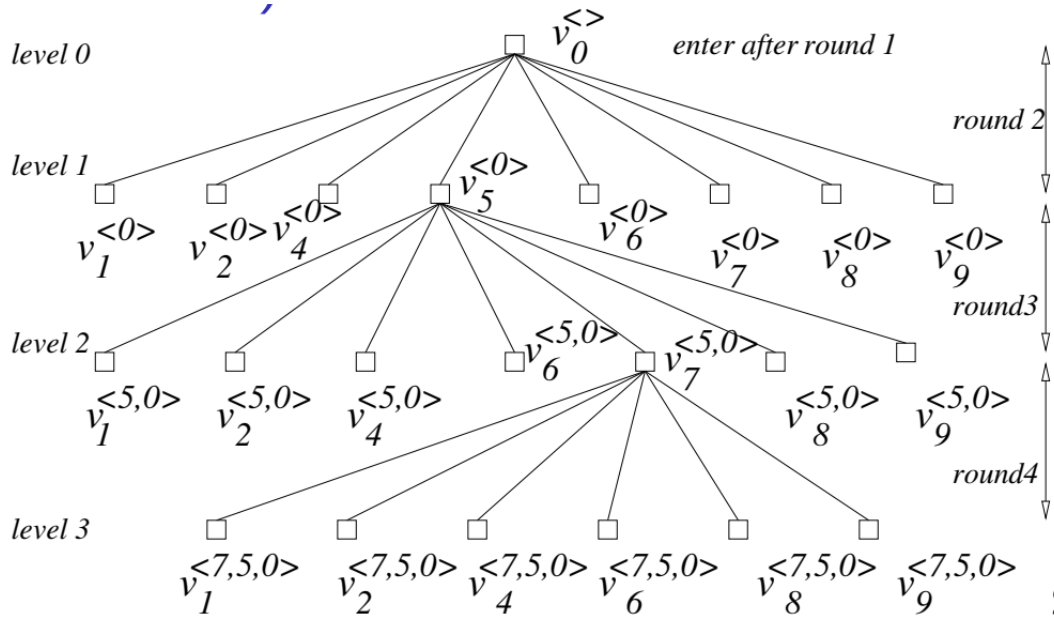
(message type)
OM(v, $Dests$, $List$, $faulty$), where the parameters are as in the recursive formulation.

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:
(1a) **send** $OM(v, N - \{i\}, \langle P_i\rangle, f)$ to $N - \{i\}$;
(1b) **return**(v).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message OM:
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}}\rangle, faulty)$ from P_{k_1};
// $faulty + round = f$; $|Dests| + sizeof(L) = n$
(2d) $v_{head(L)}^{tail(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}}\rangle, faulty - 1)$ to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n-2) \cdot \ldots \cdot (n - (level + 1))$ nodes v^L_x in level $level$, **do**
(2h) $v^L_x(x \neq i, x \notin L) = majority_{y \notin concat(\langle x\rangle, L); y \neq i}(v^L_x, v_y^{concat(\langle x\rangle, L)})$;

消息交换过程

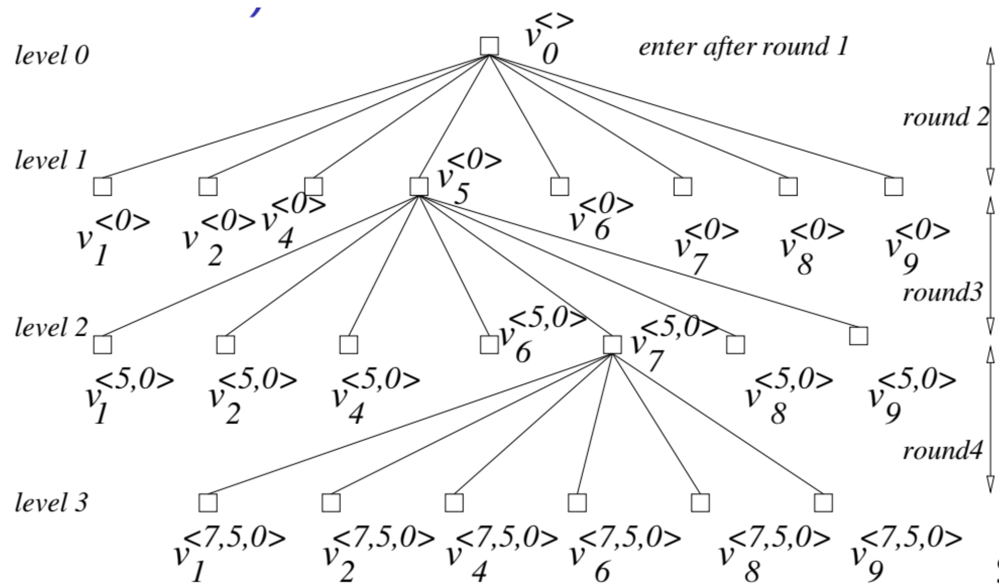
- n=10, f=3，发起节点P0，当前节点P3



(round 1) P_0 sends its value to all other processes using *Oral Msg*(3), including to P_3.
(round 2) P_3 sends 8 messages to others (excl. P_0 and P_3) using *Oral Msg*(2).
P_3 also receives 8 messages.
(round 3) P_3 sends 8 × 7 = 56 messages to all others using *Oral Msg*(1);
P_3 also receives 56 messages.
(round 4) P_3 sends 56 × 6 = 336 messages to all others using *Oral Msg*(0);
P_3 also receives 336 messages.
The received values are used as estimates of the majority function at this level of recursion.

共识值计算

- 当前节点P3
- 基于Majority
- 层层计算
- 最终确定v值



$$v_7^{\langle 5,0\rangle} \Leftarrow majority\left(v_7^{\langle 5,0\rangle}, v_1^{\langle 7,5,0\rangle}, v_2^{\langle 7,5,0\rangle}, v_4^{\langle 7,5,0\rangle}, v_6^{\langle 7,5,0\rangle}, v_8^{\langle 7,5,0\rangle}, v_9^{\langle 7,5,0\rangle}\right)$$

$$v_5^{\langle 0\rangle} \Leftarrow majority\left(v_5^{\langle 0\rangle}, v_1^{\langle 5,0\rangle}, v_2^{\langle 5,0\rangle}, v_4^{\langle 5,0\rangle}, v_6^{\langle 5,0\rangle}, v_7^{\langle 5,0\rangle}, v_8^{\langle 5,0\rangle}, v_9^{\langle 5,0\rangle}\right)$$

$$v_0^{\langle\rangle} \Leftarrow majority\left(v_0^{\langle\rangle}, v_1^{\langle 0\rangle}, v_2^{\langle 0\rangle}, v_4^{\langle 0\rangle}, v_5^{\langle 0\rangle}, v_6^{\langle 0\rangle}, v_7^{\langle 0\rangle}, v_8^{\langle 0\rangle}, v_9^{\langle 0\rangle}\right)$$

消息开销

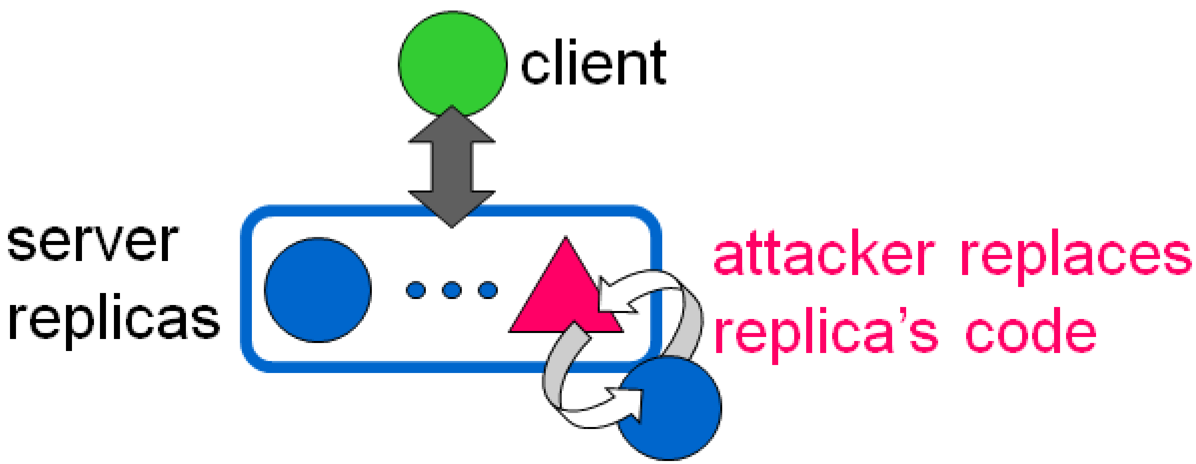
- Number of Messages Per Round

round number	a message has already visited	aims to tolerate these many failures	and each message gets sent to	total number of messages in round
1	1	f	$n-1$	$n-1$
2	2	$f-1$	$n-2$	$(n-1)\cdot(n-2)$
...	...	...	...	...
x	x	$(f+1)-x$	$n-x$	$(n-1)(n-2)\ldots(n-x)$
$x+1$	$x+1$	$(f+1)-x-1$	$n-x-1$	$(n-1)(n-2)\ldots(n-x-1)$
$f+1$	$f+1$	0	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

Complexity: $f + 1$ rounds, exponential amount of space, and
$(n - 1) + (n - 1)(n - 2) + : : : + (n - 1)(n - 2)::(n - f - 1)$messages

$O(n^f)$

PBFT



Miguel Castro, Barbara Liskov: Practical Byzantine Fault Tolerance. OSDI 1999.

*Partially based on slides by Georgios Piliouras @ Cornell

PBFT vs. Previous

Previous Work is not really practical:

- Strong assumption: (synchrony system)
  - Bounds on message delay and processing speed
- Poor performance: too many messages.

System Model

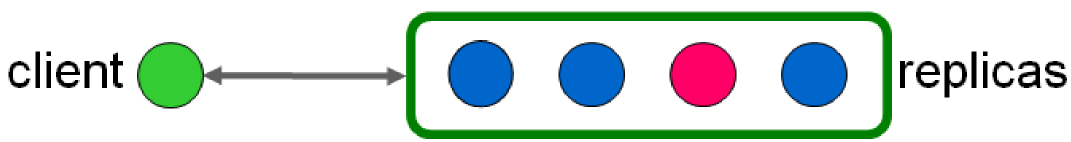
- Asynchronous system
  - No bounds on msg delay, or processing speed
    (Eventual time bounds for liveness)
- Byzantine nodes
  - Arbitrary behaviors: delay msg, inconsistent info, et al.
  - n : number of processes, f : number of faults
  - n > 3 * f + 1
- Networks are unreliable
  - Can delay, reorder, drop, retransmit
- Nodes can verify the authenticity of messages
  - Adversary can’t break cryptographic protocols

SMR in PBFT

State Machine Replication

Paxos→Raft

- Node maintains a state
  - Log, view number, state
- Can perform a set of operations
  - Need not be simple read/write
  - Must be deterministic
- Well behaved nodes must
  - Start at the same state
  - Execute requests in the same order
  - Produce identical replies upon same request



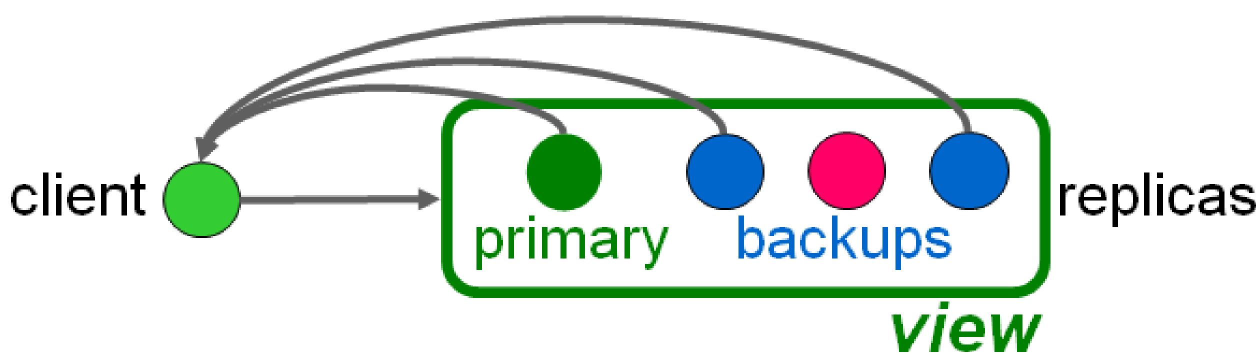
Views in PBFT

- Views are similar as "rounds"
- Operations occur within views (i.e., rounds)
- For a given view:
  - one node in is designated the primary
  - e.g., primary = v mod n

  (n is number of nodes, v is the view number)

Request Ordering

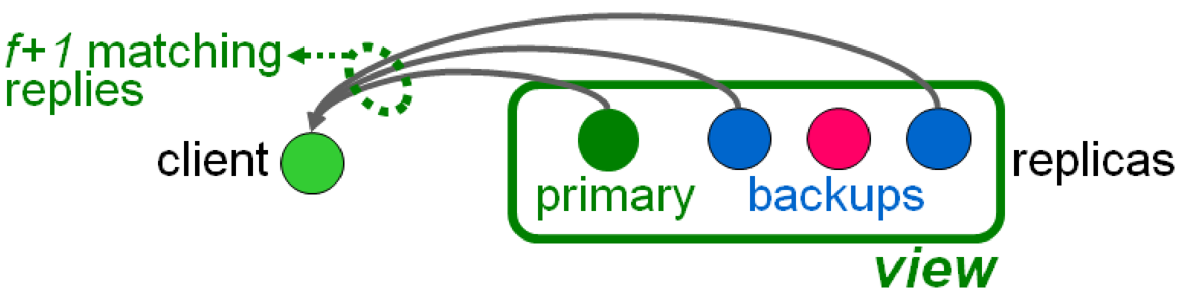
- Primary picks ordering
- Backups ensure primary behaves correctly
  - certify correct ordering
  - trigger view changes to replace faulty primary



Overall Procedure

- A client sends a request to the primary;
- The primary multicasts the request to backups;
- Each replica executes the request and send a reply to the client;
- The client waits for *f+1* replies from different replicas with the same result;

  This is the result of the operation.



Overall Procedure

- If the client does not receive replies soon enough
  - it broadcasts the request to all replicas
  - if the request already processed, simply resend the reply
- If the replica is not the primary
  - it relays the request to the primary
- If the primary does not multicast the request to others
  - it will eventually be suspected to be faulty by enough replicas to cause a view change

Protocol Components

- Normal case operation
- View change
- Garbage collection
- Recovery

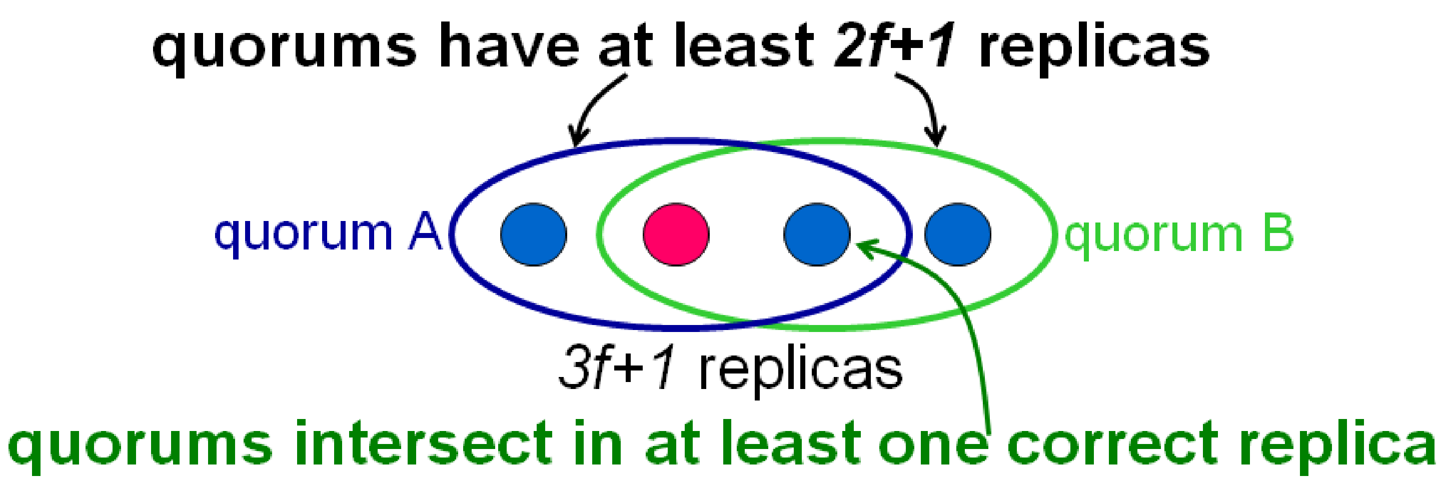
All have to be designed to work together.

Normal Case Operation

- Three-phase algorithm:
  - pre-prepare picks order of requests
  - prepare ensures order within view
  - commit ensures order across views
- Replicas remember messages in log
- Messages are authenticated

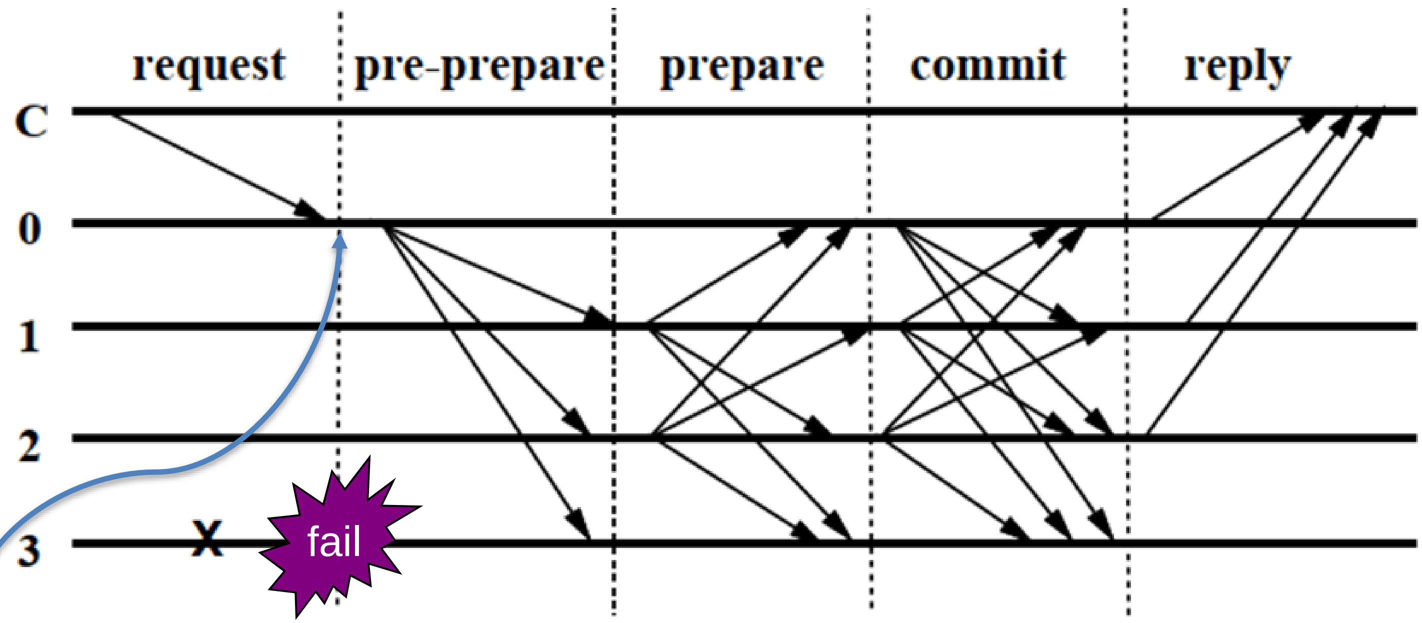
Normal Case Operation

- Certificate:
  - Set with messages from a quorum
- Algorithm steps are justified by certificates



Normal Case Operation

Request to Primary{REQUEST, operation, ts, client}



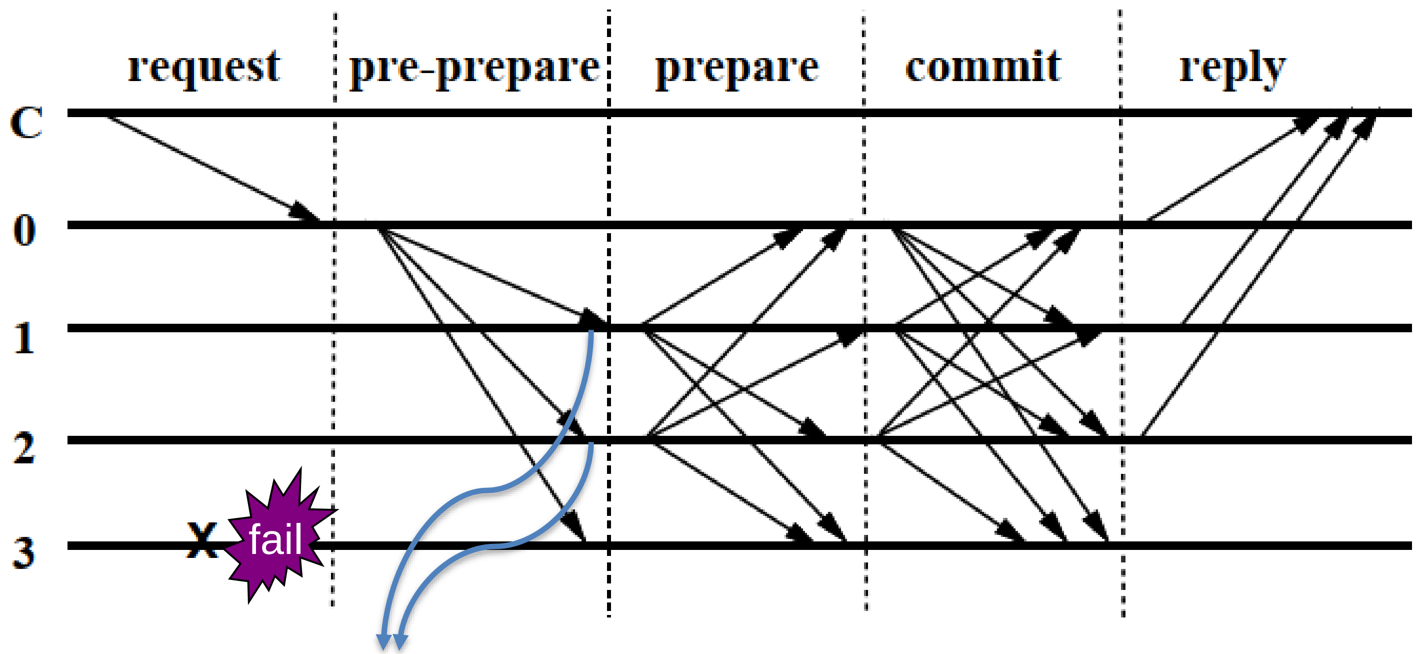
Primary assigns seq# to the request.

Normal Case Operation

Pre-prepare from primary to backup

Pre-prepare message

<{PRE_PRAPARE,v,seq#,msg_digest}, msg>

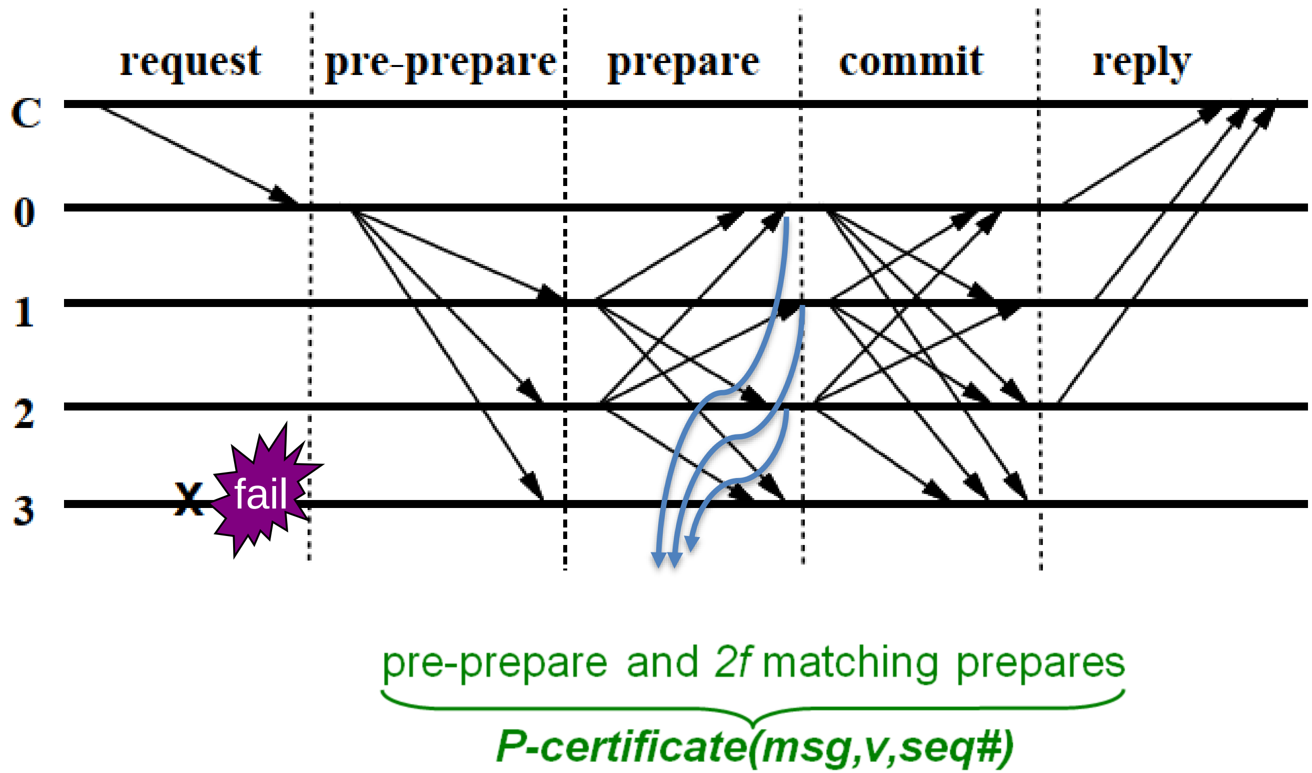


– Backups accept pre-prepare if in view v:
  - never accepted pre-prepare for v,seq# with different request

Normal Case Operation

Prepare from backups to all replicas

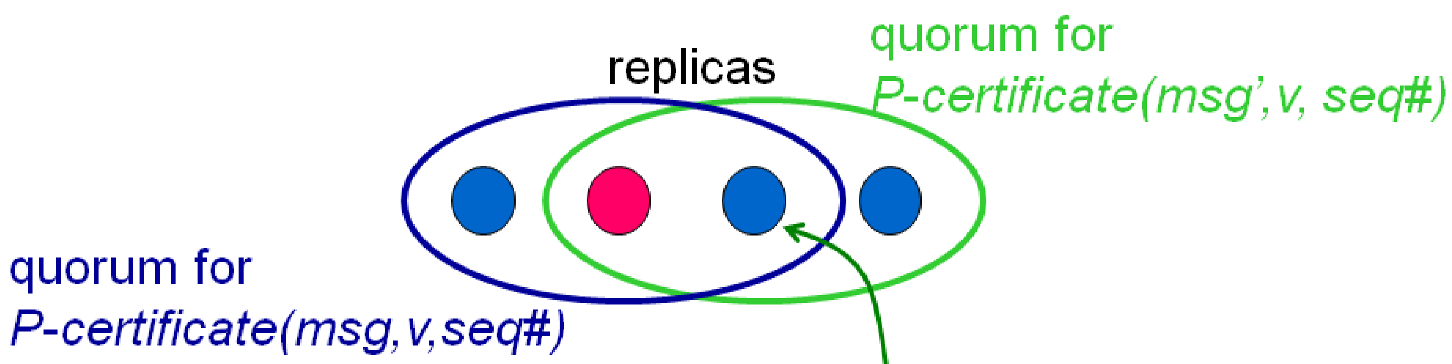
- Prepare message
- {PRAPARE,view,seq#,msg_digest,i}



Normal Case Operation

Order Within View

– No P-certificates: with the same view same sequence number but different requests

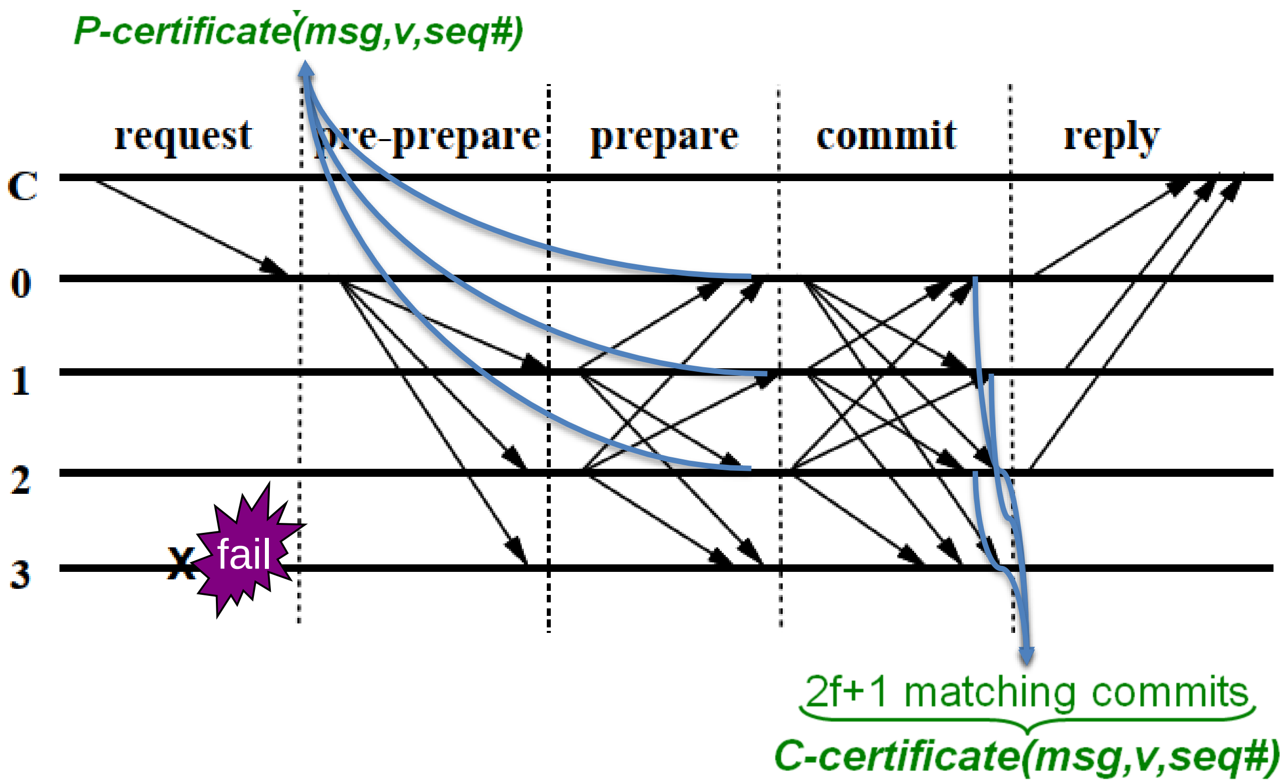


one correct replica in common ⇒ msg = msg'

Normal Case Operation

Commit among all replicas

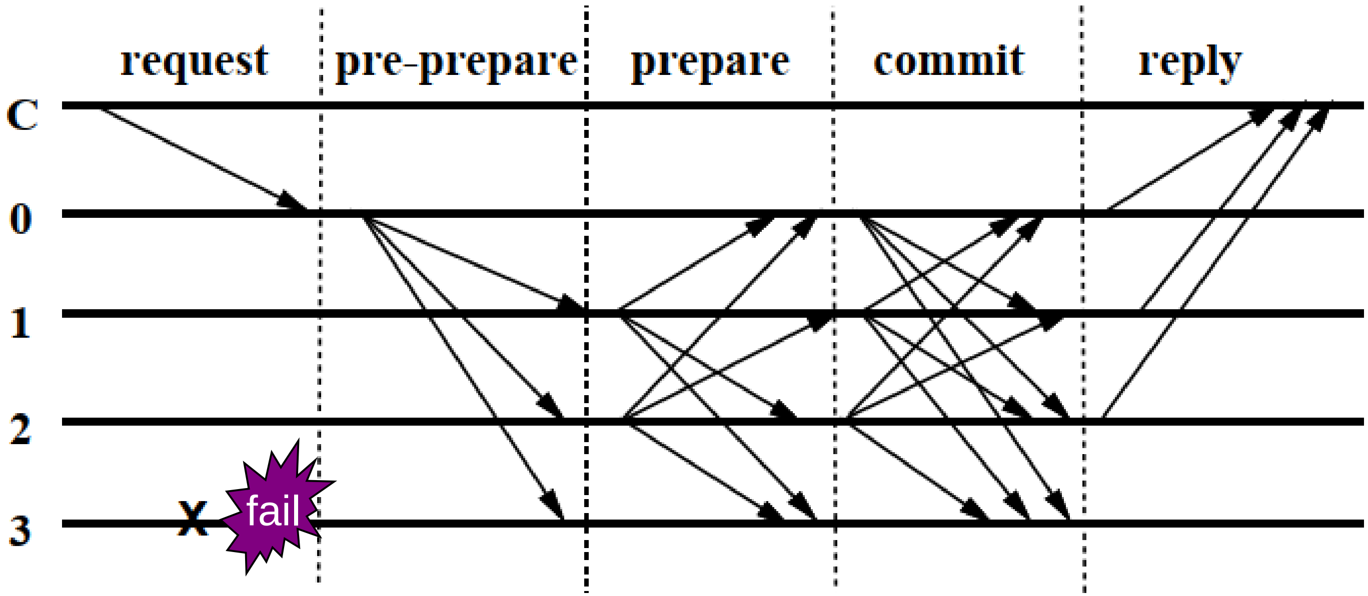
- Commit message
- {COMMIT,view,seq#,msg_digest,i}



Normal Case Operation

Reply

Reply{REPLY,view,ts,client,i,response}



Request m executed after:

- having C-certificate(msg,v,seq#)
- executing requests with number less than seq#

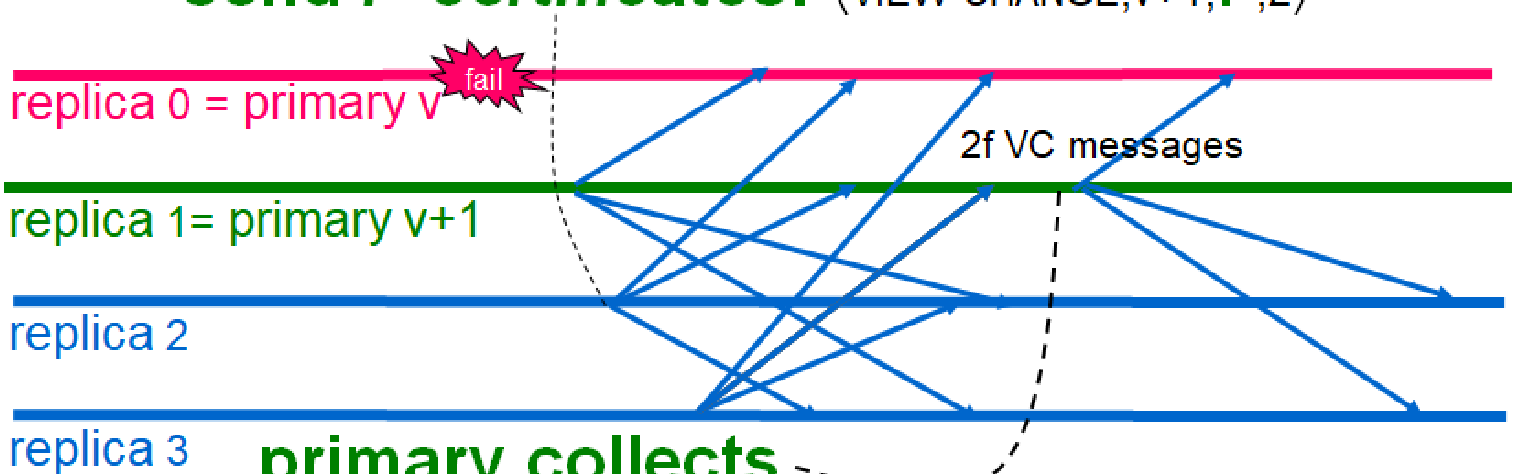
View Change

- Provide liveness when primary fails:
  - timeouts trigger view changes
  - select new primary
- But also need to:
  - preserve safety
  - ensure replicas in the same view long enough
  - prevent denial-of-service attack

View Change

P：当前节点未完成的请求的PRE-PREPARE和PREPARE消息集合

send *P-certificates*: ⟨VIEW-CHANGE,v+1,P,2⟩



primary collects
***VC-messages in X*:** ⟨NEW-VIEW,v+1,X,O⟩

pre-prepares messages
for v+1 view in O with the same sequence number

backups multicast prepare messages for pre-prepares in O

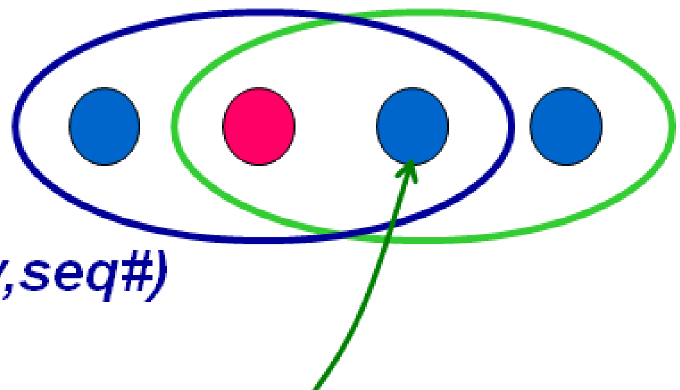
(O：Primary重新发起的未完成PRE-PREPARE消息集合)

View Change

View Change Safety

Goal: No *C-certificates* with the same sequence number and different requests

- Intuition: if replica has ***C-certificate(msg,v,seq#)*** then



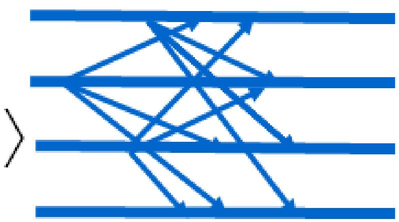
correct replica in Q has *P-certificate(msg,v,seq#)*

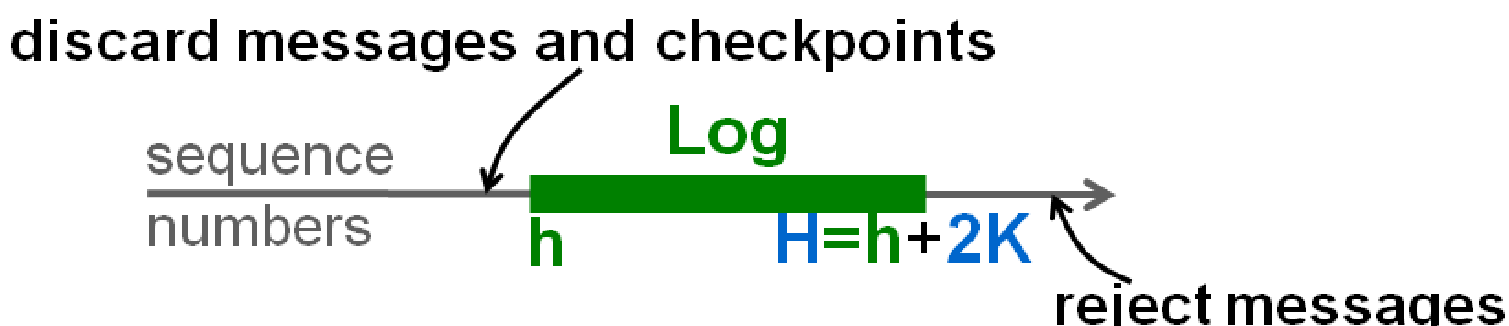
Garbage Collection

Truncate log with **certificate**:

- periodically checkpoint state (**K**)
- multicast ⟨CHECKPOINT,seq#,D(checkpoint),i⟩
- all collect *2f+1 checkpoint messages*

S-certificate(h,checkpoint)





send checkpoint in view-changes

PBFT Correctness

Formal Correctness Proofs

- Complete safety proof with I/O automata:
  - invariants
  - simulation relations
- Partial liveness proof with timed I/O automata:
  - invariants

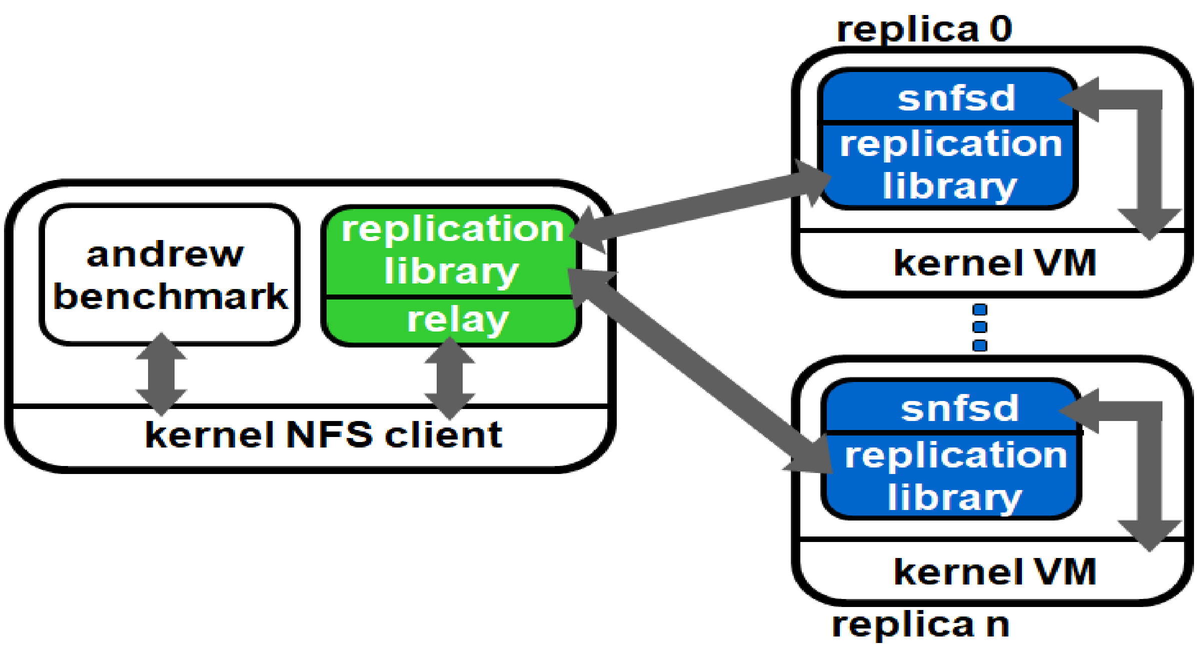
PBFT Optimizations

- Digest replies: send only one reply with full result
- Optimistic execution: execute prepared requests
- Read-only operations: executed in current state

PBFT Implementation

Implementation Example

BFS: A Byzantine-Fault-Tolerant NFS



No synchronous writes – stability through replication

A Summary

- Different consensus/agreement protocols
- System models: syn vs. asyn
- Fault types: crash, Bynzantine
- Paxos, Raft
- BFT, PBFT

谢谢！

wuweig@mail.sysu.edu.cn