UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 739
Distributed Systems

Michael Swift
Notes (c) Andrea C. Arpaci-Dusseau

Byzantine Generals

One paper:

- “The Byzantine Generals Problem”, by Lamport, Shostak, Pease, In *ACM Transactions on Programing Languages and Systems*, July 1982

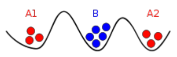
Notes from Reviews

- Is BFT/PBFT too complex for the protocols & better off using firewalls?
- Is the model of only *f* nodes can fail reasonable? Why not MTTF?

Background on Failure

- Two generals problem:
  - Two armies, each led by a general, are preparing to attack a fortified city. The armies are encamped near the city, each on its own hill. A valley separates the two hills, and the only way for the two generals to communicate is by sending messengers through the valley.
  - Unfortunately, the valley is occupied by the city's defenders and there's a chance that any given messenger sent through the valley will be captured. Note that while the two generals have agreed that they will attack, they haven't agreed upon a time for attack before taking up their positions on their respective hills.
  - The two generals must have their armies attack the city at the same time in order to succeed. They must thus communicate with each other to decide on a time to attack and to agree to attack at that time, and each general must know that the other general knows that they have agreed to the attack plan

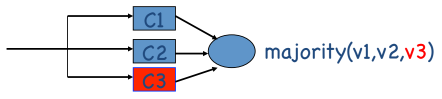
Two Generals Problem



- Challenge: how do you agree on a time?
  - Send a message “attack at noon”
  - Send a response “o.k.”
  - QUESTION: can they attack?
  - Try with two more messages
    - Did you get my message
    - Yes
    - QUESTION: Can they attack? No
  - Can never agree:
    - Assume shortest protocol takes N messages
    - From perspective of sender, doesn’t know outcome of last message
    - Will take same action independent of that
      - Recipient must take same action, so must do same thing whether or not last message is received
    - So could use N-1 messages…
  - Bigger point: dealing with failure is complicated

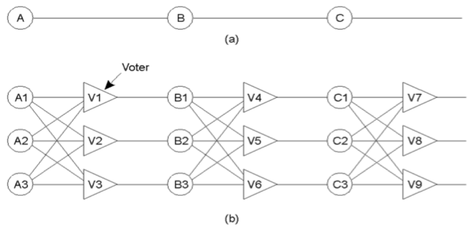
Motivation

- Build reliable systems in the presence of faulty components
  - Extension of lamport clock replicated state machines
- Common approach:
  - Have multiple (potentially faulty) components compute same function
  - Perform majority vote on outputs to get “right” result



f faulty, f+1 good components ==> 2f+1 total

Example: Triple Modular Redundancy



- What if A produces different results to each voter?

What is a Byzantine Failure?

- Three primary differences from Fail-Stop Failure
  1) Component can produce arbitrary output
     - Fail-stop: produces correct output or none
     - Different receivers see different outputs
  2) Cannot always detect output is faulty
     - Fail-stop: can always detect that component has stopped
  3) Components may work together maliciously
     - No collusion across components

Process Resilience

- Handling faulty processes: organize several processes into a group
  - All processes perform same computation
  - All messages are sent to all members of the group
  - Majority need to agree on results of a computation
  - Ideally want multiple, independent implementations of the application (to prevent identical bugs)
- Replicated state machines

Assumption

- Good (nonfaulty) components must use same input
  - Otherwise, can't trust their output result either

- For majority voting to work:
1) All nonfaulty processors must use same input
2) If input is nonfaulty, then all nonfaulty processes use the value it provides

Byzantine Generals

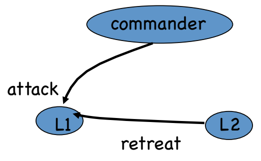
- Algorithm to achieve agreement among "loyal generals" (i.e., working components) given m "traitors" (i.e., faulty components)
- Agreement such that:
  A) All loyal generals decide on same plan
  B) Small number of traitors cannot cause loyal generals to adopt "bad plan"
- Terminology
  - Let v(i) be information communicated by ith general
  - Combine values v(1)...v(n) to form plan
- Rephrase agreement conditions:
  A) All generals use same method for combining information
  B) Decision is majority function of values v(1)...v(n)

Key Step: Agree on inputs

- Generals communicate v(i) values to one another:
  1) Every loyal general must obtain same v(1)..v(n)
  1') Any two loyal generals use same value of v(i)
     - Traitor i will try to get loyal generals into using different v(i)'s
  2) If ith general is loyal, then the value he sends must be used by every other general as v(i)
- Problem: How can each general send his value to n-1 others?
- A commanding general must send an order to his n-1 lieutenants such that:
  IC1) All loyal lieutenants obey same order
  IC2) If commanding general is loyal, every loyal lieutenant obeys the order he sends
- These are *Interactive Consistency* conditions
  - Everybody agrees on a vector, and agrees on the [ith] element if node i is correct

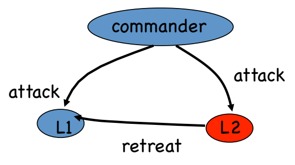
Impossibility Result

- With only 3 generals, no solution can work with even 1 traitor (given oral messages)



What should L1 do? Is commander or L2 the traitor???

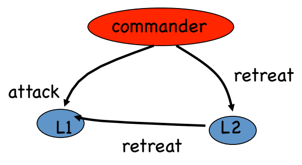
Option 1: Loyal Commander



What must L1 do?

By IC2: L1 must obey commander and attack

Option 2: Loyal L2



What must L1 do?

By IC1: L1 and L2 must obey same order --> L1 must retreat
Problem: L1 can't distinguish between 2 scenarios

Problem: L1 and L2 do not agree on inputs, cannot vote

General Impossibility Result

- No solution with fewer than 3m+1 generals can cope with m traitors
- *Why does not work with just 3:*
  - *A commander, B and C lieutenants, B traitor*
    - *B can lie so that it produces exactly the same results as if A had been the liar.*
- Why not less than 3m+1
  - Can group all m failures into one group, all successful ones in other groups, and solve the 3-node problem

How many failures occur?

- Big question: how do you know what f is, make sure you never exceed it?
  - Answer: look at RAID – has same problem
- What if you have a correlated failure?
  - power failure
  - simultaneous attack
  - bug in coordination software

Simple solution for 1 failure

- Leader sends message to all lieutenants
- Lieutenants exchange their messages
- (SHOW)

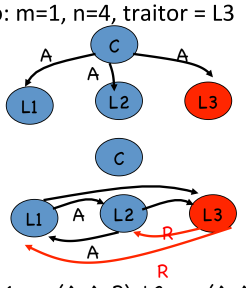
- How do we extend this to more failures?

Example: Bad Lieutenant

- Scenario: m=1, n=4, traitor = L3

OM(1):

OM(0):???

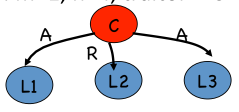


Decision?? L1 = m (A, A, R); L2 = m (A, A, R); Both attack!

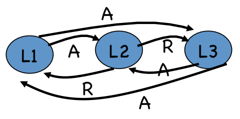
Example: Bad Commander

- Scenario: m=1, n=4, traitor = C

OM(1):



OM(0):???



Decision?? L1=m(A, R, A); L2=m(A, R, A); L3=m(A,R,A); Attack!

Oral Messages

- Assumptions
  - A1) Every message is delivered correctly
  - A2) Receiver knows who sent message
  - A3) Absence of message can be detected
- Question: are these realistic?
  - use tcp/ip for correctness, IP address for source, time for absence of messages
  - Assume not outside malicious actors, or spoofing

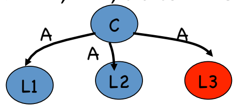
Oral Message Algorithm

- `OM(0)`
  - Commander sends his value to every lieutenant
- `OM(m), m>0`
  - Commander sends his value to every lieutenant
  - For each `i`, let v_i be value Lieutenant i receives from commander; act as commander for `OM(m-1)` and send `vi` to `n-2` other lieutenants
  - For each `i` and each `j` **`not`** `i`, let v_j be value Lieut `i` received from Lieut `j`. Lieut i computes `majority(`$v_1, \ldots, v_{n-1}$`)`

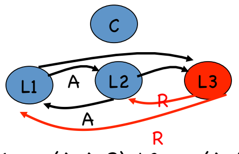
Example: Bad Lieutenant

- Scenario: m=1, n=4, traitor = L3

OM(1):



OM(0):???

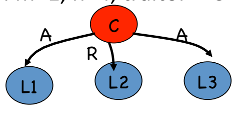


Decision?? L1 = m (A, A, R); L2 = m (A, A, R); Both attack!

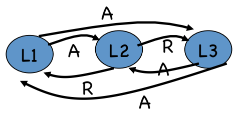
Example: Bad Commander

- Scenario: m=1, n=4, traitor = C

OM(1):



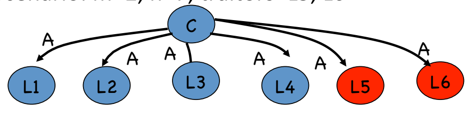
OM(0):???



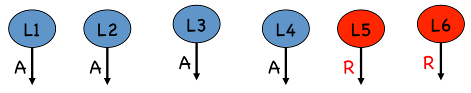
Decision?? L1=m(A, R, A); L2=m(A, R, A); L3=m(A,R,A); Attack!

Bigger Example: Bad Lieutenants

- Scenario: m=2, n=7, traitors=L5, L6



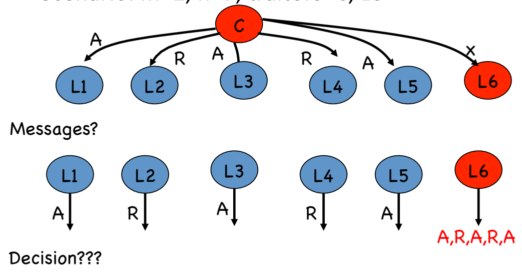
Messages?



Decision??? m(A,A,A,A,R,R) ==> All loyal lieutenants attack!

Bigger Example: Bad Commander+

- Scenario: m=2, n=7, traitors=C, L6



Decision with Bad Commander+

- L1: m(A,R,A,R,A,A) ==> Attack
- L2: m(R,R,A,R,A,R) ==> Retreat
- L3: m(A,R,A,R,A,A) ==> Attack
- L4: m(R,R,A,R,A,R) ==> Retreat
- L5: m(A,R,A,R,A,A) ==> Attack
- Problem: All loyal lieutenants do NOT choose same action
  – SO: need more communication – commander + cheater can tip majority

Next Step of Algorithm

- **Key: Verify that lieutenants tell each other the same thing**
  – Requires # rounds = m+1 (0 + m)
  – OM(0): Msg from Lieut i of form: “L0 said v0, L1 said v1, etc...”
    - At each level, get agreement on what the sender sends out, so cannot have some A some R
- What messages does L1 receive in this example?

```
– OM(2): A (commander sends)
– OM(1):      2R, 3A, 4R, 5A, 6A (recv what cmdr sent from others)
– OM(0):    2{    3A, 4R, 5A, 6R} (2 sends all msgs heard in round 1)
–           3{2R,     4R, 5A, 6A}
–           4{2R, 3A,     5A, 6R}
–           5{2R, 3A, 4R,     6A}
–           6{ total confusion }
```

- All see same messages in OM(0) from L1,2,3,4, and 5
  – Uses majority for each other node – look down columns of matric in OM(0)
- m(1A,2R,3A,4R,5A,-) ==> All attack
- NOTE: # of messages: (n-1)(n-2)(n-3)…(n-m-1) messages for m traitors

Signed Messages

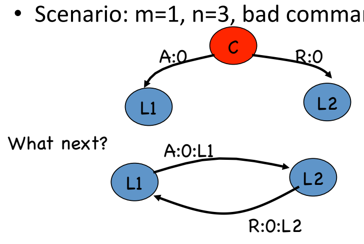
- New assumption: Cryptography
- A4) a. Loyal general’s signature cannot be forged and contents cannot be altered
- b. Anyone can verify authenticity of signature
- Simplifies problem:
  – When lieutenant i passes on signed message from j, know that i did not lie about what j said
  – Lieutenants cannot do any harm alone (cannot forge loyal general’s orders)
  – Only have to check for traitor commander
- With cryptographic primitives, can implement Byzantine Agreement with `m+2` nodes, using `SM(m)`

Signed Messages Algorithm: SM(m)

```
1. Commander signs v and sends to all as (v:0)
2. Each lieut i: keeps a set $V_i$
•     A) If receive (v:0) and no other order
•           1) Vi = v
•           2) send (V:0:i) to all (so all see what he is going to do)
•     B) If receive (v:0:j:...:k) and v not in $V_i$
•           1) Add v to $V_i$
•           2) if (k<m) send (v:0:j:...:k:i) to all not in j...k
• NOTE: send because not everybody has seen it – all nodes in vector could be faulty
• 3. When no more msgs, obey order of choose($V_i$)
   – Why? Need to make sure all loyalists have seen all variants of Vi, use deterministic choice
```

SM(1) Example: Bad Commander

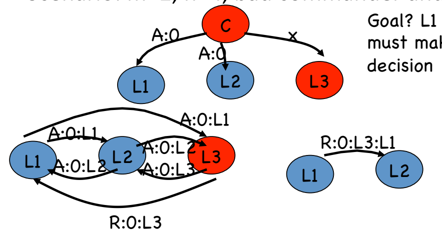
- Scenario: m=1, n=3, bad commander



V1={A,R} V2={R,A}
Both L1 and L2 can trust orders are from C
Both apply same decision to {A,R}

SM(2): Bad Commander+

- Scenario: m=2, n=4, bad commander and L3



Goal? L1 and L2 must make same decision

V1 = V2 = {A,R} ==> Same decision

Implementing Assumptions

- A1) Every message sent by nonfaulty processor is delivered correctly
  - Network failure ==> processor failure
  - Handle as less connectivity in graph
- A2) Processor can determine sender of message
  - Communication is over fixed, dedicated lines
  - Switched network???
  - Use secret-key encryption (point-to-point)
- A3) Absence of message can be detected
  - Fixed max time to send message + synchronized clocks ==> If msg not received in fixed time, use default
  - Treats late messages as byzantine failures, only tolerate a few.
- A4) Processors sign msgs such that nonfaulty signatures cannot be forged
  - Use randomizing function or cryptography to make likelihood of forgery very small
  - Public key signatures

Do byzantine faults occur

- Byzantine fault: a fault presenting different symptoms to different observers
- Yahoo study: all faults either
  - fail stop, ommission
  - correlated (many simultaneous failures)
- Honeywell study:
  - occurs in external interaction due to mis-timings
  - race conditions modifying data when sending out to replicas
- What about security attacks?

Practical Byzantine Fault Tolerance

- Use Byzantine FT to provide a service
  - How can you make a real regular service survive byzantine faults?
- Use replicated state machine model
  - Cluster of nodes (> 3f+1 for f faults)
  - Need independence for uncorrelated failures
    - multiple implementations, power supplies, networks, operating systems
  - Service must be deterministic (same start state, same transitions leads to same output)
- Clients request service from cluster, it replicates request internally
  - Uses a **signed-messages protocol** obviously
- Relax synchronous requirement for **correctness**, needed for liveness

PBFT main ideas

- Idea (from Lampson):
  - Handle normal and worst case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst case must make some progress
- Use signed messages
  - Allows quick determination of who sent a message
  - Allows forwarding messages (e.g. for view-change)
- To deal with malicious primary
  - Use a 3-phase protocol to agree on sequence number
- To deal with loss of agreement
  - Use a bigger quorum (2f+1 out of 3f+1 nodes)
- To deal with as asynchrony:
  - Need only 2f+1 *fast enough* machines
  - Treat longer delay as sign of failure, ensure *progress*

Why 3f+1 for signed msgs?

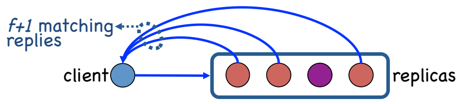
- Not just computing internal result, but outside world (client) needs to see result, needs to tell faulty from non-faulty outcome
    - BG model: local node needs to know what to do
  - Must be possible to proceed after contacting (n-f) replicas, since f replicas might be faulty and not respond
- Also possible that the f replicas that did not respond are not faulty (but slow …), so f of those that did reply are faulty
  - Asynchrony assumption
- So need enough of a majority in responses to make right decision: n-2f > f → n > 3f

PBFT Strategy

- Primary runs the protocol in the normal case
- Replicas *watch* the primary and do a view change if it fails
  - Elect new trustworthy leader
- Note: not replicating for scalability, just fault tolerance

Algorithm Overview

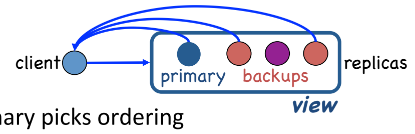
State machine replication:
- deterministic replicas start in same state
- replicas execute same requests in same order
- correct replicas produce identical replies



Hard: ensure requests execute in same order

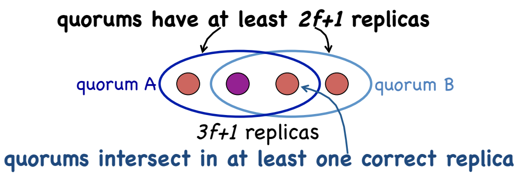
Ordering Requests

Primary-Backup:
- View designates the primary replica



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

Quorums and Certificates



quorums have at least *2f+1* replicas
quorum A
quorum B
3f+1 replicas
quorums intersect in at least one correct replica

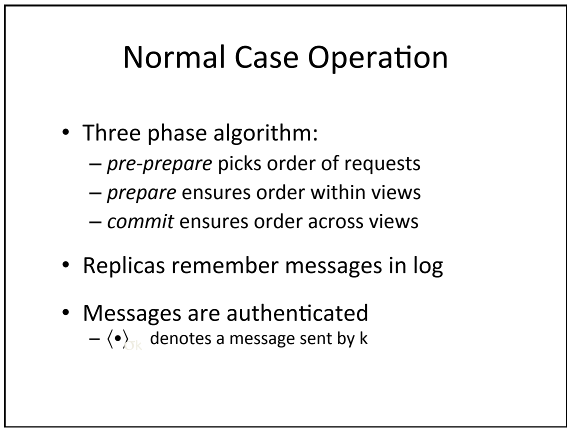
- **Certificate** ≡ set with messages from a quoru
- Algorithm steps are justified by certificates

Replica state

- A replica id i (between 0 and N-1)
  - Replica 0, replica 1, …
- A view number v#, initially 0
- Primary is the replica with id
  i = v# mod N
- A log of <op, seq#, status> entries
  - Status = pre-prepared or prepared or committed

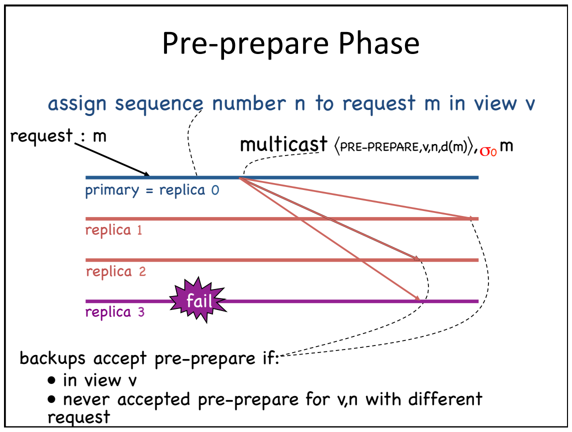
View Change

- Replicas watch the primary
- Replicas request a view change when one node is slow or misbehaving
  - When enough replicas ask for view change, it starts (act like pre-prepare messages)
- Commit point for new view: when 2f+1 replicas have prepared
  - Just f nodes cannot trigger a view change



Normal Case Operation

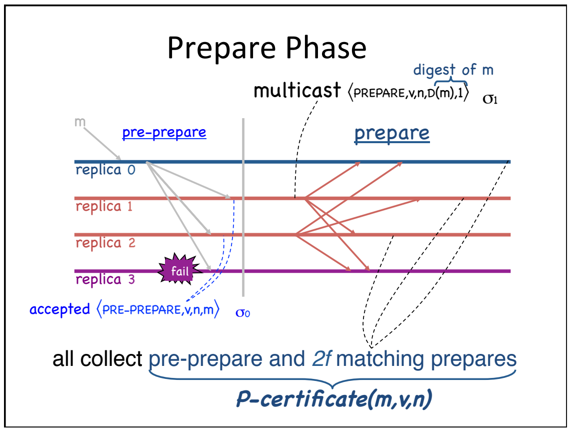
- Three phase algorithm:
  - *pre-prepare* picks order of requests
  - *prepare* ensures order within views
  - *commit* ensures order across views
- Replicas remember messages in log
- Messages are authenticated
  - $\langle \bullet \rangle_{\sigma_k}$ denotes a message sent by k



Pre-prepare Phase

assign sequence number n to request m in view v
request : m
multicast $\langle$PRE-PREPARE,v,n,d(m)$\rangle_{\sigma_0}$ m
primary = replica 0
replica 1
replica 2
replica 3
fail

backups accept pre-prepare if:
- in view v
- never accepted pre-prepare for v,n with different request

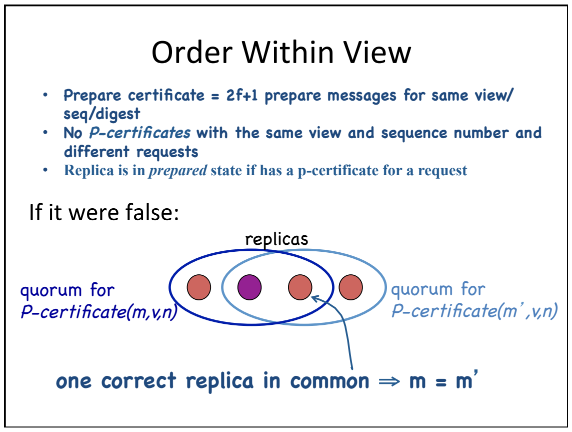


Prepare Phase

digest of m
multicast $\langle$PREPARE,v,n,D(m),1$\rangle_{\sigma_1}$
m
pre-prepare
prepare
replica 0
replica 1
replica 2
replica 3
fail
accepted $\langle$PRE-PREPARE,v,n,m$\rangle_{\sigma_0}$

all collect pre-prepare and *2f* matching prepares

P-certificate(m,v,n)



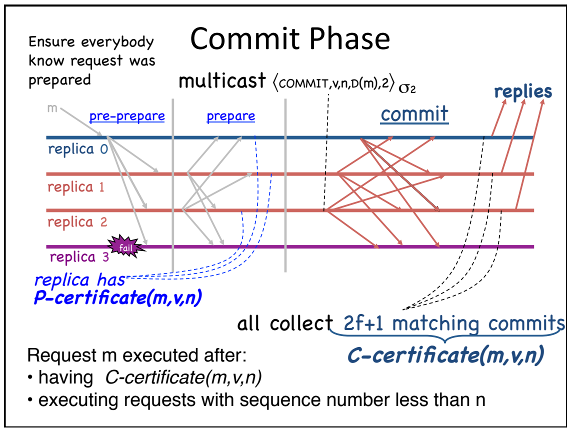
Order Within View

- **Prepare certificate = 2f+1 prepare messages for same view/ seq/digest**
- **No *P-certificates* with the same view and sequence number and different requests**
- **Replica is in *prepared* state if has a p-certificate for a request**

If it were false:

replicas
quorum for P-certificate(m,v,n)
quorum for P-certificate(m',v,n)

one correct replica in common ⇒ m = m'



Commit Phase

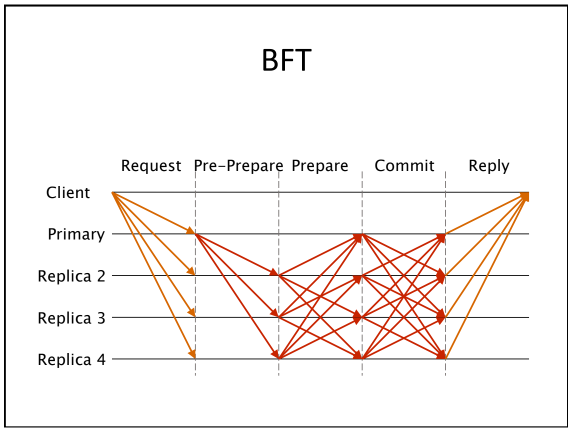
Ensure everybody know request was prepared

multicast $\langle$COMMIT,v,n,D(m),2$\rangle_{\sigma_2}$
replies
m
pre-prepare
prepare
commit
replica 0
replica 1
replica 2
replica 3
fail
replica has P-certificate(m,v,n)

all collect 2f+1 matching commits

C-certificate(m,v,n)

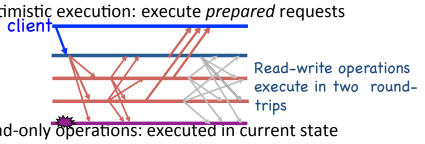
Request m executed after:
- having *C-certificate(m,v,n)*
- executing requests with sequence number less than n



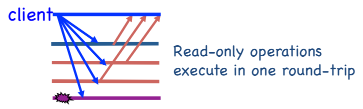
BFT

Communication Optimizations

- HMAC for digest – no public key
  - Public key only for view change, where messages are forwarded
- Digest replies: send only one reply to client with result
- Optimistic execution: execute *prepared* requests



- Read-only operations: executed in current state



View Change

- Replicas watch the primary
- Request a view change
  - send a do-viewchange request to all
    - Include proof you saw all previous pre-prepare messages
  - new primary requires 2f+1 requests
  - sends new-view with this certificate
- Need 2f+1 view-change requests to prevent faulty nodes from triggering frequent changes
- Rest is similar
- **Key point: handle failure separately from normal case!**

Performance

- PBFT NFS file server runs about the same speed as normal NFS
  - Why?
- Answers:
  - NFS bottlenecked by disk, PBFT NFS leaves data in memory
  - PBFS introduces new message latency + encryption, but not that much compared to disk
  - NFS not CPU bound, so ample CPU to do extra encryption
  - Only one client, so synchronous workload – not look at scalability

Additional Issues

- State transfer
- Checkpoints (garbage collection of the log)
- Selection of the primary
- Timing of view changes
- Under failure situations, throughput drops to zero while views change