Trustworthy Systems

A trustworthy system

- does what you want
- nothing else!

} Basic PL research, Program correctness, Program verification

- despite human and operator errors } User interfaces
- despite environmental disruptions } Fault tolerance
- despite attacks } Security

The Odd Couple

Fault-tolerance	Security
Integrity	Integrity
Availability	Availability
	Confidentiality

A working hypothesis

- Model compromised processes as Byzantine
  - Faulty processes can deviate arbitrarily (maliciously) from spec
  - Faulty processes can collude
- Build replicated services that can tolerate (a threshold of) Byzantine failures

Outline

The Rise and Fall of State Machine Replication

- State Machine Replication
- Paxos
- Byzantine agreement
- Byzantine fault-tolerance can be fast!
  - PBFT
- The Emperor is naked...

Outline

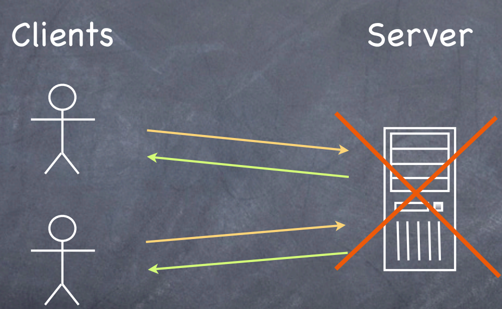
The Rise and Fall of State Machine Replication

- State Machine Replication
- Paxos
- Byzantine agreement
- Byzantine fault-tolerance can be fast!
  - PBFT
- The Emperor is naked...

Rethinking State Machine Replication

- The principle: <u>separate agreement from execution</u>
- The payoffs:
  - lower replication costs/stronger confidentiality

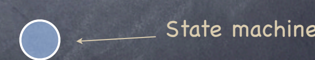
The Problem



Solution: replicate server!

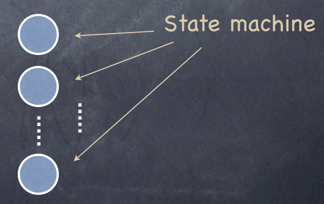
The Solution

1. Make server deterministic (state machine)



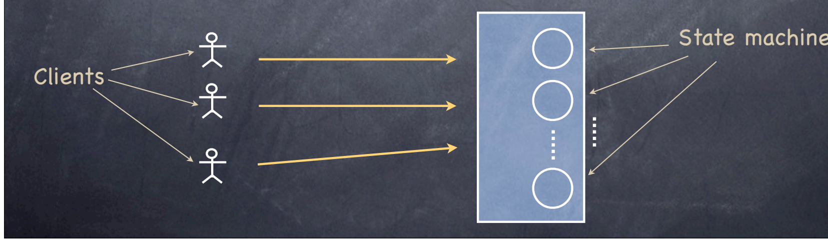
The Solution

1. Make server deterministic (state machine)
2. Replicate server



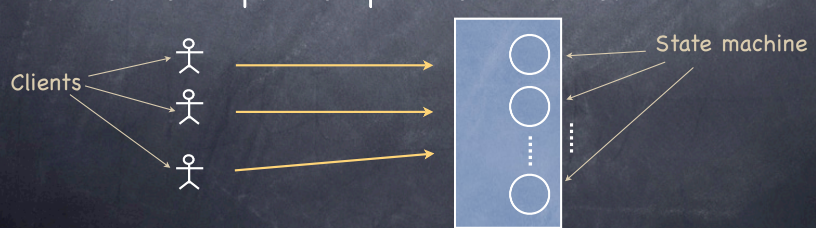
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions



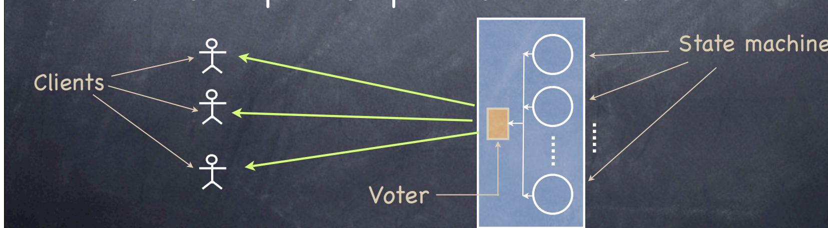
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

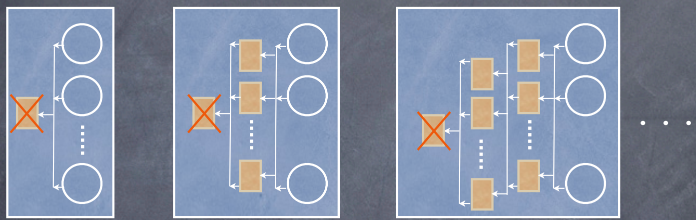


The Solution

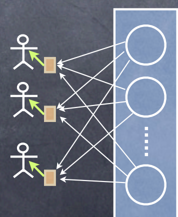
1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance



A conundrum



A: voter
and client
share fate!



Replica Coordination

All non-faulty state machines receive all requests in the same order

- AGREEMENT: Every non-faulty state machine receives every request
- ORDER: Every non-faulty state machine processes the requests it receives in the same relative order

The Part-Time Parliament

- Parliament determines laws by passing sequence of numbered decrees
- Legislators can leave and enter the chamber at arbitrary times
- No centralized record of approved decrees–instead, each legislator carries a ledger

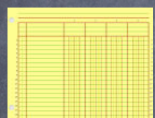
Government 101

If a majority of legislators were in the Chamber and no one entered or left the Chamber for a sufficiently long time, then

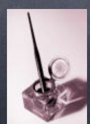
- any decree proposed by a legislator would eventually be passed
- any passed decree would appear on the ledger of every legislator

Supplies

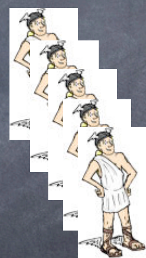
Each legislator receives



ledger



pen with indelible ink



lots of messengers

scratch paper

hourglass

Back to the future

- A set of processes that can propose values
- Processes can crash and recover
- Processes have access to stable storage
- Asynchronous communication via messages
- Messages can be lost and duplicated, but not corrupted

The Game: Consensus

SAFETY

- Only a value that has been proposed can be chosen
- Only a single value is chosen
- A process never learns that a value has been chosen unless it has been

LIVENESS

- Some proposed value is eventually chosen
- If a value is chosen, a process eventually learns it

The Players

- Proposers
- Acceptors
- Listeners

Choose a value...

1. A single acceptor

Choose a value...

1. ~~A single acceptor~~
2. A majority of acceptors (forces a single value)

Choose a value...

1. ~~A single acceptor~~
2. A majority of acceptors (forces a single value)

When should an acceptor accept?

Choose a value...

1. ~~A single acceptor~~
2. A majority of acceptors (forces a single value)

When should an acceptor accept?

- ① Acceptors must accept first received proposal
- Acceptors must accept multiple proposals

Choose a value...

1. ~~A single acceptor~~
2. A majority of acceptors (forces a single value)

When should an acceptor accept?

① Acceptors must accept first received proposal

- Acceptors must accept multiple proposals

(pid,value)

...a unique value...

② If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

...a unique value...

② If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

② If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

...a unique value...

② If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

② If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

①+②=trouble

…a unique value…

- ② If a proposal with value *v* is chosen, then every higher-numbered proposal that is chosen has value *v*
- ② If a proposal with value *v* is chosen, then every higher-numbered proposal accepted by any acceptor has value *v*
- ② If a proposal with value *v* is chosen, then every higher-numbered proposal issued by any proposer has value *v*

…and only a unique value

- ② If a proposal with value *v* is chosen, then every higher-numbered proposal issued by any proposer has value *v*
- ② For any *v* and *n*, if a proposal with value *v* and pid *n* is issued, then there is a majority-set S of acceptors such that one of the following holds:
  - a. no acceptor in S has accepted any proposal numbered less than *n*
  - *b.* *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* accepted by acceptors in S

Say I do: The proposer's protocol

1. A proposer chooses a new *n* and sends <*prepare,n*> to each member of some set of acceptors, asking to respond with:
   a. A promise never again to accept a pid less than *n*, and
   b. The accepted proposal with highest pid less than *n* if any.
2. If proposer receives a response from a majority of acceptors, then it can issue <*accept*(*n*,*v*)> where *v* is the value of the highest pid among the responses, or is any value selected by the proposer if responders returned no proposals

Say I do: The acceptor's protocol

1. Always respond to *prepare* messages
2. Respond to <*accept*(*n*,*v*)> iff it has not responded to <*prepare,n'*> with *n'* > *n* ①
3. Write intended response to stable storage before sending it

Note that ① ⇒ ①

The Learning Channel

i. Each acceptor informs each learner

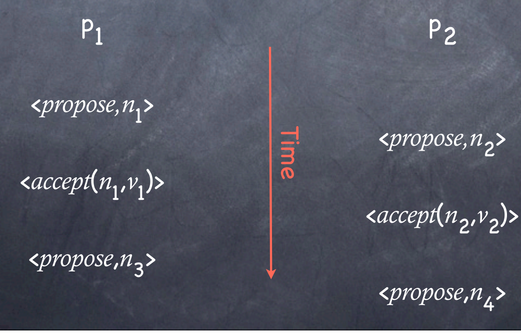
ii. Acceptors contact a distinguished learner, which informs other learners

iii. Acceptors contact a set of learners...

Don't stop me now

Liveness (surprise!) is not guaranteed:

$n_1 < n_2 < n_3 < n_4 < \ldots$



All proposers are equal, but some more so than others

- Elect a distinguished proposer
- Can't be done reliably in asynchronous systems, so...
  - real time
  - randomization

Agreement and Byzantine Generals

- One General G, a set of Lieutenants L_i
- General can order Attack (A) or Retreat (R)
- General may be a traitor; so may be some of the Lieutenants

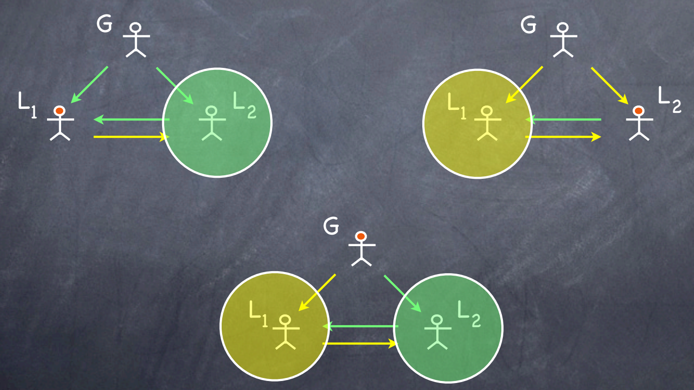
* * *

I. If G is trustworthy, every trustworthy L_i must follow G's orders

II. Every trustworthy L_i must follow same battleplan

The plot thickens...

One traitor



A lower bound (LSP82)

There is no algorithm that solves Byzantine agreement when $n \leq 3f$

A Byzantine Renaissance

- Practical Byzantine Fault-Tolerance (CL99, CL00)
  - first to be safe in asynchronous systems
  - fast! PBFT NSF only 3% slower than standard NFS on Andrew benchmark
  - uses proactive recovery to tolerate more failures over system lifetime
- BASE (RCL 01)
  - uses abstraction to reduce correlated faults

Major issue : Assumptions

- Replication algorithms make assumptions
  - behavior of faulty process
  - synchrony
  - bound of number of faults
- Service fails if assumptions are not valid
  - attacker can make service fail by making assumptions invalid
- Most earlier algorithms assume too much, and are thus vulnerable

Second issue : Performance

- Replication has performance overhead
  - Extra communication and computation
  -
- Practical algorithms require low overhead
-
- Till now : replication algorithms that do not assume too much perform poorly!

Contributions of PBFT

- Practical replication algorithm
  - Weak assumptions
  - Good performance
-
- Implementation
  - Replicated library service
  - Byzantine tolerant NFS implementation

Bad assumption : benign faults

- Most previous replication techniques assume :
  - Replicas fail by omitting/ stopping
- Invalid with malicious attacks
  - Compromised replicas may behave arbitrarily
  - Single such fault can compromise service
  - Lesser resiliency to malicious attacks!

Bad assumption : synchrony

- Synchrony : assuming known bounds on
  - Delay between steps
  - Message delays
- Assumption invalid with denial-of-service attacks
  - bad replies due to increased delays
    - system fails

Synchrony is assumed by most Byzantine fault tolerant schemes...

Issues with asynchrony

- No delay bounds
- Problem is : FLP!
- Solution in BFT:
  i. Provide safety without using synchrony
     - guarantees no bad replies
  ii. Assume eventual time bounds for liveness
     - System may not reply with active denial-of-service attack
     - But will reply when the attack ends

Bad assumption: Bound on number of faults

- Given enough time, more than f replicas are likely to malfunction
- Detection of faults is hard and slow

Bad assumption: Bound on number of faults

- Given enough time, more than f replicas are likely to malfunction
- Detection of faults is hard and slow
- Unavoidable

Bad assumption: Bound on number of faults

- Given enough time, more than f replicas are likely to malfunction
- Detection of faults is hard and slow
- Unavoidable
- Solution in BFT :
  i. Proactive recovery – periodic recovery tasks scheduled even when no faults are suspected
  ii. Frequent recoveries

High availability if at most f failures in a "window"

To summarize: THEM bad...

- Strong assumptions
  - Safety relies on synchrony - easy to break in
  - Unbounded storage - impractical
  - Absolute bound on number of faults
- Too slow to be used in practice
  - Extensive use of public key cryptography
  - High communication overhead

...BFT gooood!

- Supports complex operation requests from clients
- Safety
  - System behaves like a correct centralized service
- Liveness
  - Clients eventually receive replies to requests

BFT assumptions

- 3f+1 replicas to tolerate f Byzantine faults
- Strong cryptography
- Eventual time bounds - only for liveness

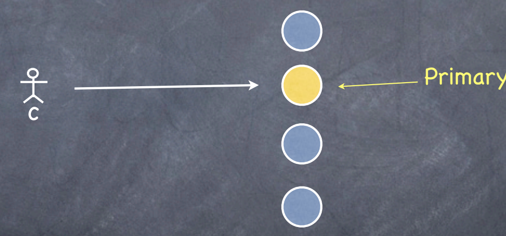
Ordering Requests

- Idea : Use quorums (remember Paxos?)
  - But now need to tolerate Byzantine faults...
- Primary-Backup

Ordering Requests

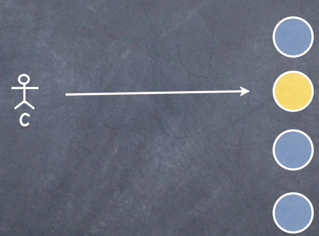
- Idea : Use quorums (remember Paxos?)
  - But now need to tolerate Byzantine faults...
- Primary-Backup
  - Protocol proceeds in Views
  - Current view designates the Primary
  - Primary orders the requests by assigning sequence numbers
  - Backups ensure correct behavior of Primary
    - Certify correct ordering by Primary
    - Trigger view change to replace faulty primary

Client-Service interactions



$\langle REQUEST,o,t,c\rangle_{\sigma_c}$

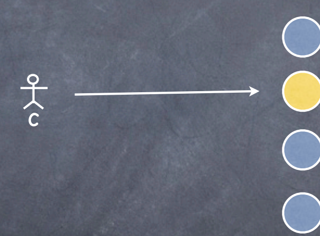
Client-Service interactions



$\langle REQUEST,o,t,c\rangle_{\sigma_c}$

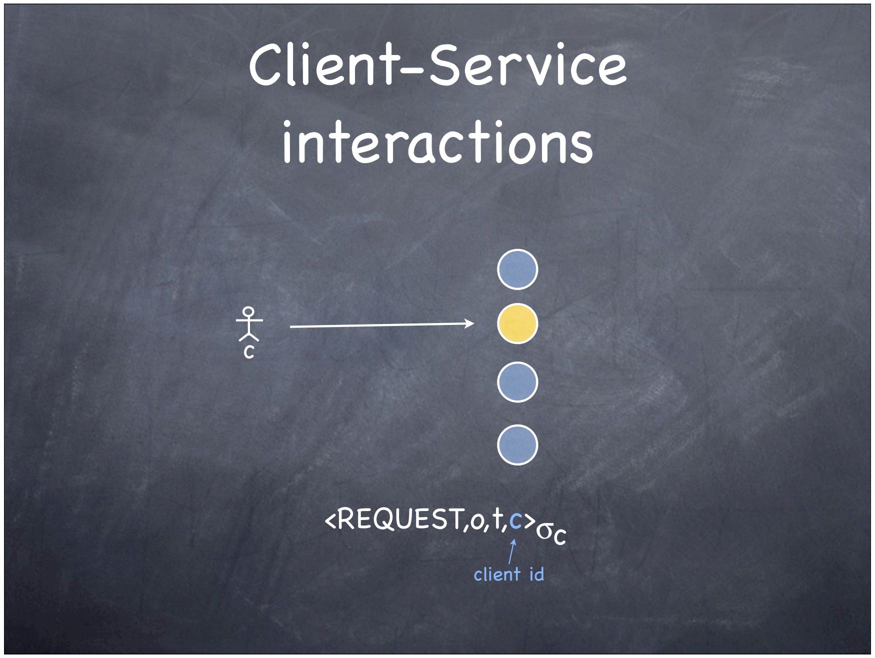
state machine operation

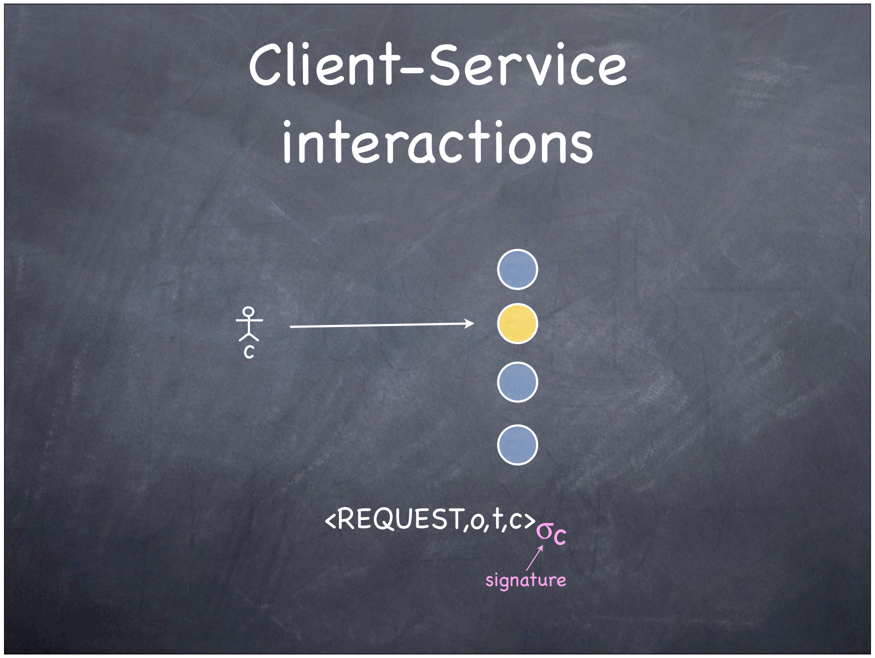
Client-Service interactions

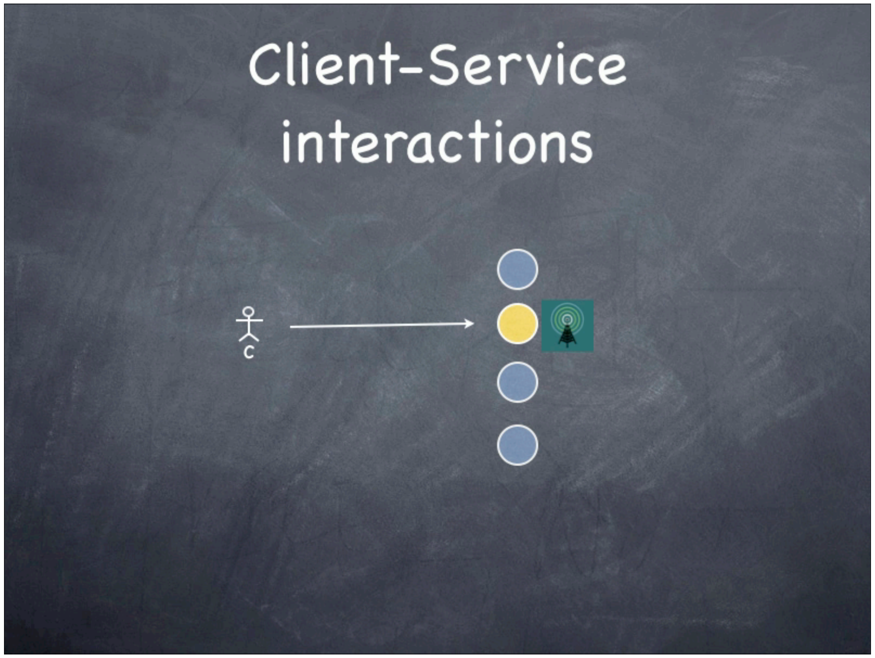


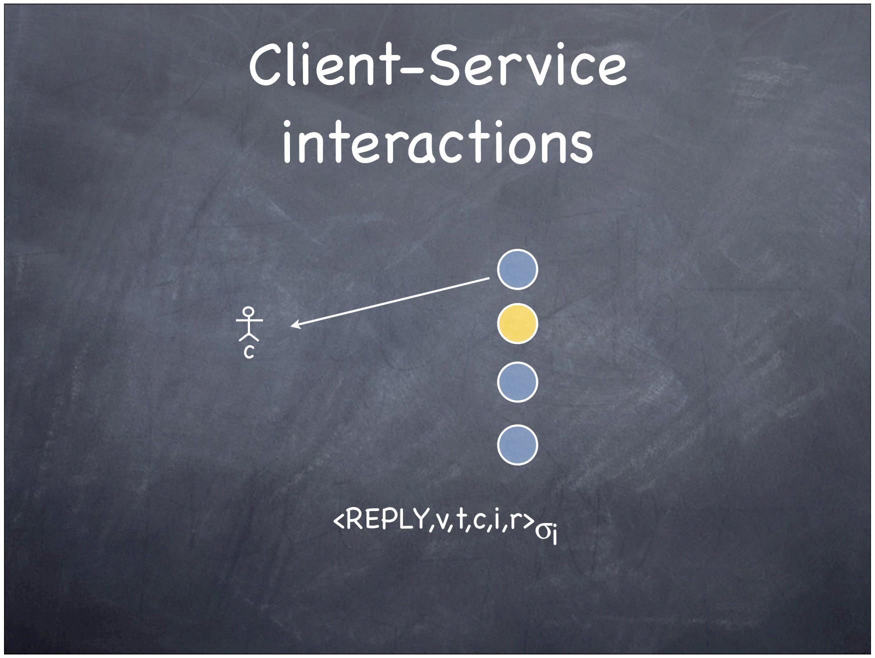
$\langle REQUEST,o,t,c\rangle_{\sigma_c}$

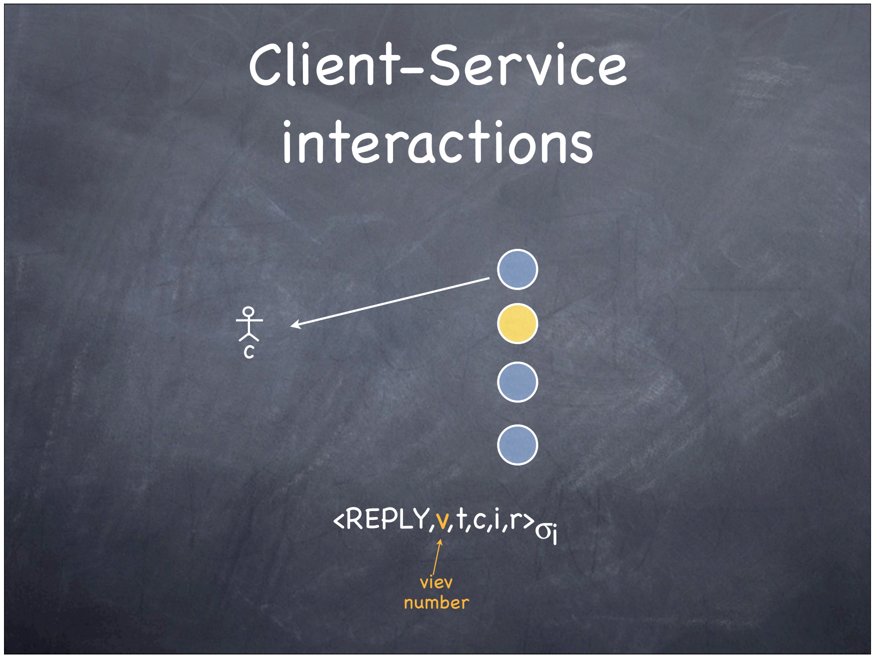
timestamp

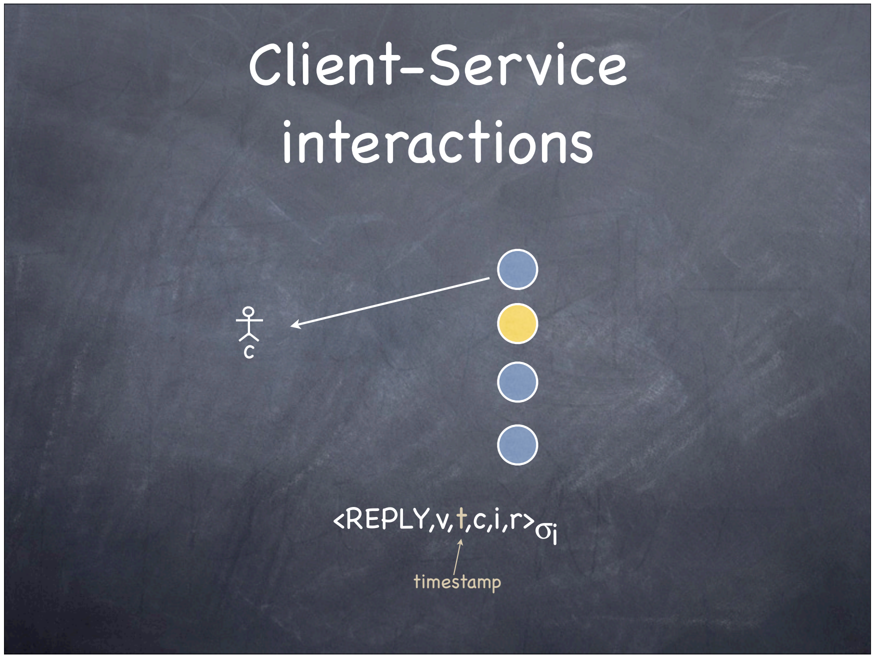

Client-Service interactions
c
<REQUEST,o,t,c>σc
client id

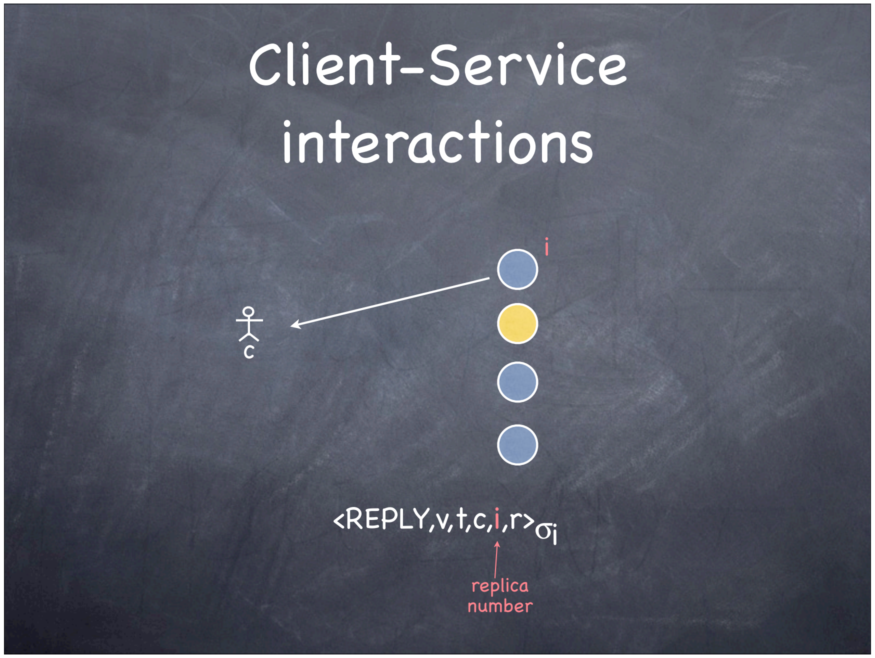

Client-Service interactions
c
<REQUEST,o,t,c>σc
signature

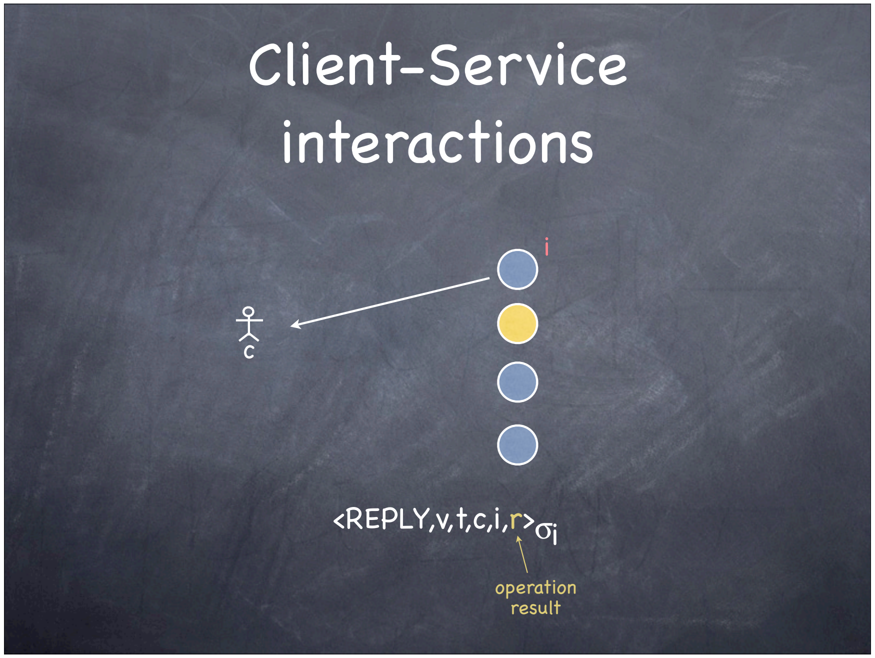

Client-Service interactions
c


Client-Service interactions
c
<REPLY,v,t,c,i,r>σi


Client-Service interactions
c
<REPLY,v,t,c,i,r>$_{\sigma_i}$
viev number


Client-Service interactions
c
<REPLY,v,t,c,i,r>$_{\sigma_i}$
timestamp


Client-Service interactions
i
c
<REPLY,v,t,c,i,r>$_{\sigma_i}$
replica number


Client-Service interactions
i
c
<REPLY,v,t,c,i,r>$_{\sigma_i}$
operation result

Client-Service interactions

Before accepting r, c waits for f+1 replies with same t and r from different replicas

Troubleshooting

- If c times out waiting for reply, it broadcasts its request to all replicas
- If replica has already computed response, it just returns it
- Otherwise, replica forward request to primary
- If primary does not multicast, it is eventually suspected

Quorums and Certificates

- Quorums contain at least 2f+1 replicas
- Any two quorums intersect in at least one correct replica
- Always one quorum available with non-faulty replicas
- Certificate: set of messages from a quorum which guarantees or certifies a certain property
- Algorithm steps are justified by certificates

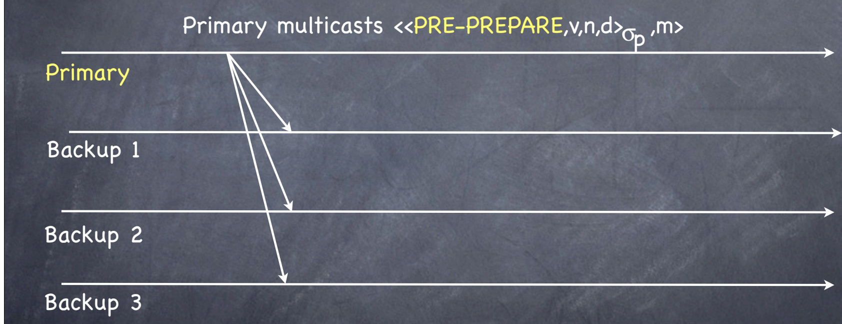
Algorithm Components

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

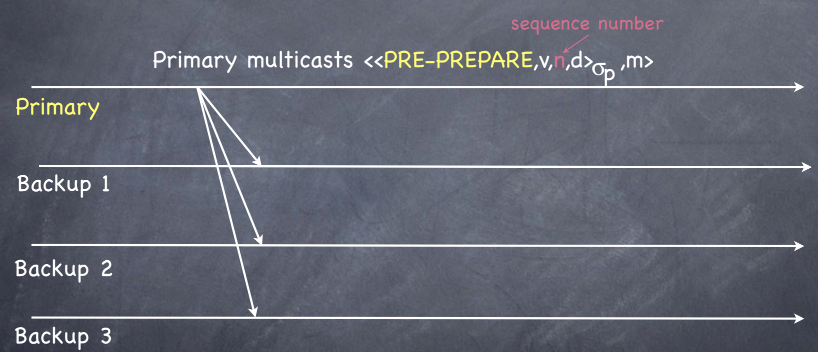
Normal case operation

- 3 phase algorithm :
  i. Pre-prepare phase picks order of requests
  ii. Prepare phase ensure ordering of requests within views
  iii. Commit phase ensures order across views
- Replicas remember messages on stable log
- Messages are authenticated

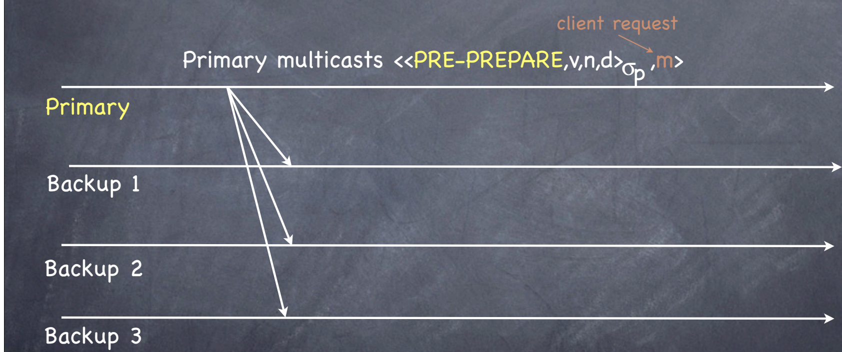
Pre-prepare



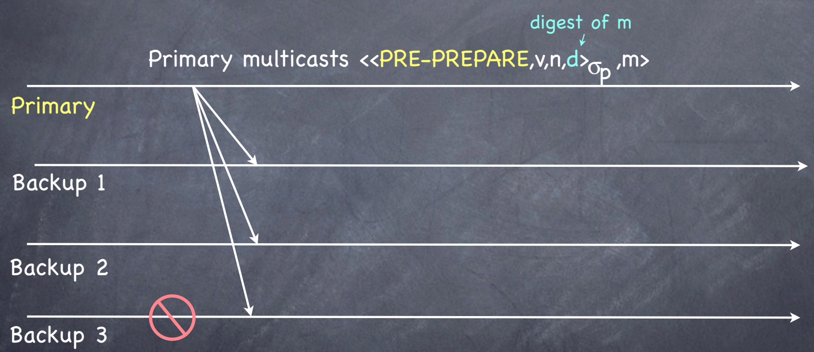
Pre-prepare



Pre-prepare



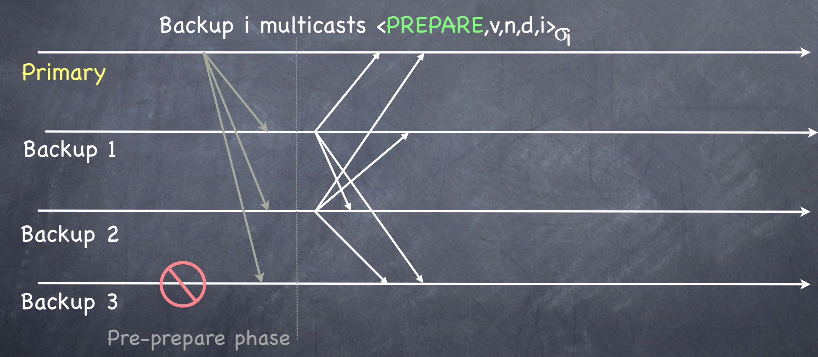
Pre-prepare



Backup i accepts PRE-PREPARE if:

- PRE-PREPARE is well formed
- i is in view v
- i has not accepted another PRE-PREPARE for v,n with a different d
- n is between two water-marks

Prepare



Prepare Certificate

- P-certificates ensure total order within views

Prepare Certificate

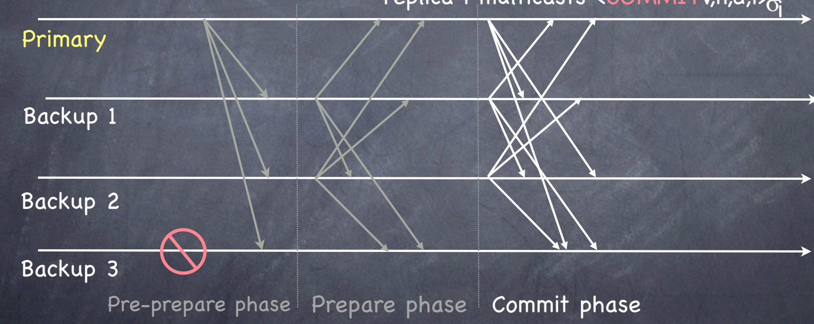
- P-certificates ensure total order within views
- Replica produces P-certificate(m,v,n) iff its log holds:
  - The request m
  - A pre-prepare for m in view v with sequence number n
  - 2f prepares from different backups that match the pre-prepare

Prepare Certificate

- P-certificates ensure total order within views
- Replica produces P-certificate(m,v,n) iff its log holds:
  - The request m
  - A pre-prepare for m in view v with sequence number n
  - 2f prepares from different backups that match the pre-prepare
- A P-certificate(m,v,n) means that a quorum agrees with assigning sequence number n to m in view v
  - NO two non-faulty replicas with P-certificate(m_1,v,n) and P-certificate(m_2,v,n)

Commit

After receiving a P-certificate, replica i multicasts <COMMIT v,n,d,i>$_{\sigma_i}$



Commit Certificate

- A replica has a C-certificate(m,v,n) if:
  - It had a P-certificate(m,v,n)
  - Log contains 2f+1 matching commits from different replicas
- Replica executes a request after it gets C-certificate for it, and has cleared all previous requests

A useful invariant

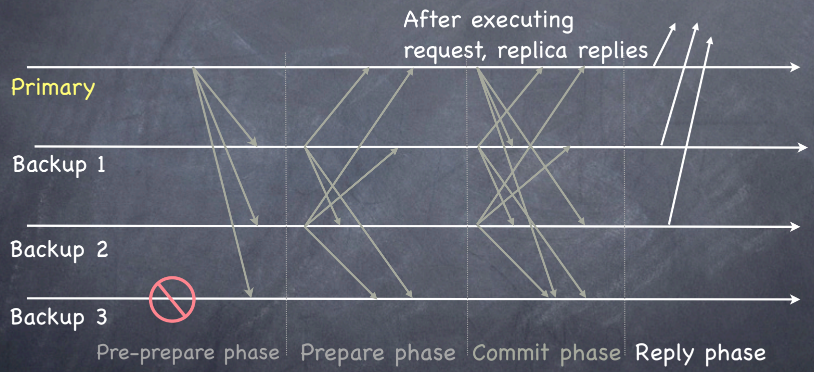
Some replica has C-certificate(m,v,n) ≡
f+1 correct replicas have a P-certificate

It ensures the following properties:

i. Non-faulty replicas agree on sequence number of requests that commit locally even across view changes

ii. If non-faulty replica builds C-certificate, eventually f+1 non-faulty replicas do so

Reply



Garbage Collection

- Truncate Log with Certificate
  - Each replica periodically checkpoints state and builds certificate to prove state is correct
  - Multicasts $\langle CHECKPOINT, n, d, i \rangle_{\sigma_i}$

Garbage Collection

- Truncate Log with Certificate
  - Each replica periodically checkpoints state and builds certificate to prove state is correct
  - Multicasts $\langle CHECKPOINT, n, d, i \rangle_{\sigma_i}$

    last request

Garbage Collection

- Truncate Log with Certificate
  - Each replica periodically checkpoints state and builds certificate to prove state is correct
  - Multicasts $\langle CHECKPOINT, n, d, i \rangle_{\sigma_i}$

    state digest

Garbage Collection

- Truncate Log with Certificate
  - Each replica periodically checkpoints state and builds certificate to prove state is correct
  - Multicasts <CHECKPOINT,n,d,i>$_{\sigma_i}$
  - CK-Certificate ≡ 2f+1 checkpoint messages for same n,d from different i's
  - CK-certificate used in view changes
  - CK-certificate advances low, high watermarks

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
    - n: last proved ckpt

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
    - P: {P-certificates held by i for requests with sn > n}

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
- When primary j for v+1 receives 2f VIEW-CHANGE:
  - multicasts <NEW-VIEW,v+1,V,O>$_{\sigma_j}$

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
- When primary j for v+1 receives 2f VIEW-CHANGE:
  - multicasts <NEW-VIEW,v+1,V,O>$_{\sigma_j}$

    {2f+1 VIEW-CHANGE messages}

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
- When primary j for v+1 receives 2f VIEW-CHANGE:
  - multicasts <NEW-VIEW,v+1,V,O>$_{\sigma_j}$

    set of PRE-PREPARE messages

View changes

- If primary in view v times out, replica i :
  - stops accepting messages (except CHEKPOINT,VIEW-CHANGE,NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1,n,CK-cert,P,i>$_{\sigma_i}$
- When primary j for v+1 receives 2f VIEW-CHANGE:
  - multicasts <NEW-VIEW,v+1,V,O>$_{\sigma_j}$
  - appends messages in O to its log
  - enters view v+1

O's

A set of <PRE-PREPARE,v+1,n,d>$_{\sigma_j}$,
for all n: min-s < n ≤ max-s, where

- min-s = sn of latest proved checkpoint in V
- max-s = sn of latest P-certificate in V
-

O's

A set of <PRE-PREPARE,v+1,n,d>$_{\sigma_j}$,
for all n: min-s < n ≤ max-s, where

- min-s = sn of latest proved checkpoint in V
- max-s = sn of latest P-certificate in V
-

$$d = \begin{cases} \text{digest of m with P-certificate<m,v,n> (if any)} \\ d^{null} \end{cases}$$

Safety

- Within a view, replicas agree on sn of requests for which a C-certificate can be built

Safety

- Within a view, replicas agree on sn of requests for which a C-certificate can be built
- Across views?

Safety

- Within a view, replicas agree on sn of requests for which a C-certificate can be built
- Across views?
  - C-certificate(m,v,n) ⇒ 2f+1 P-certificate(m,v,n)
  - $\langle \text{NEW-VIEW}, v+1, V, O \rangle_{\sigma_j}$ accepted ⇒ 2f+1 VIEW-CHANGE
  - At least 1 correct replica in v+1 has P-certificate(m,v,n) !

Liveness

Install new views conservatively :

Try maximizing period T where 2f+1 correct replicas are in the same view

Increase T exponentially until some request executes

Communication Optimizations

i. One replica sends response, other send digests

ii. Replicas may optimistically execute requests for which hold a P-certificate
  - return tentative response
  - client needs 2f+1 tentative responses to accept

iii. Read Only requests
  - replicas execute in current state
  - client accepts if it receives 2f+1 responses
  - otherwise, send regular R/W request

Fast Authentication

- Use MACs instead of digital signatures
- MAC is 1000x faster than PK signatures
- Public key cryptography used to setup MAC keys, VIEW-CHANGE and NEW-VIEW messages
- Non-trivial
  - MAC less powerful than signatures
  - Receiver cannot prove authenticity to others...

Back to the Dark Ages

- Too many replicas

  Who cares? Machines are cheap...

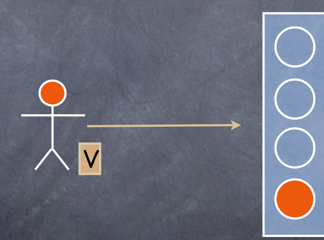
  But achieving independent failures is expensive

  – Independently failing hardware

  – Independently failing software!

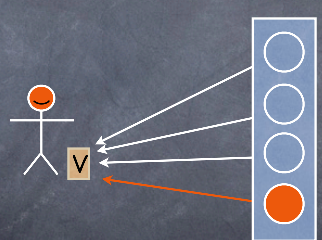
Back to the Dark Ages

- No confidentiality



Back to the Dark Ages

- No confidentiality



Rethinking State Machine Replication

Not Agreement + Order

but rather Agreement on Order + Execution

Rethinking State Machine Replication

Not Agreement + Order

but rather Agreement on Order + Execution

Benefits

- ~~3f+1~~ 2f+1 state machine replicas

Rethinking State Machine Replication

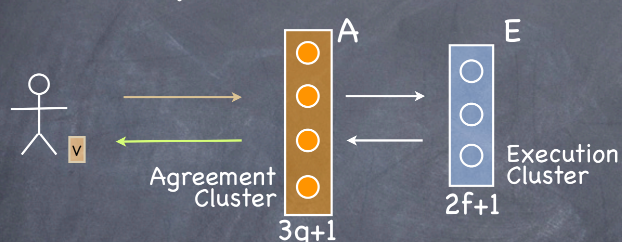
Not Agreement + Order

but rather Agreement on Order + Execution

Benefits

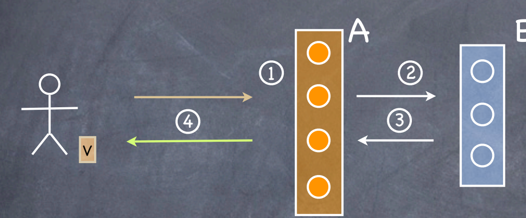
- ~~3f+1~~ 2f+1 state machine replicas
- Replication ~~hurts~~ helps confidentiality

Separation reduces replication costs



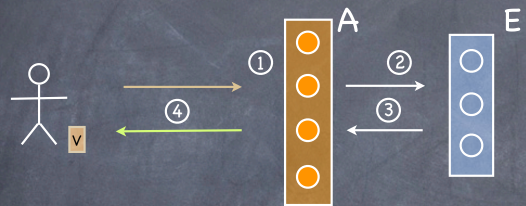
- Not all nodes are created equal!
  - Nodes in E: expensive
    - (different across applications and within same application)
  - Nodes in A: cheap
    - (simple and reusable across applications)

The implementation...



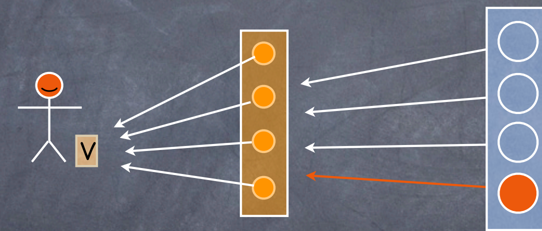
1. A assigns unique sequence number to request
2. <request, rsn>$_A$: request is certified unique
3. E executes in rsn order
4. <reply, rsn>$_E$: reply is certified unique

...is simple

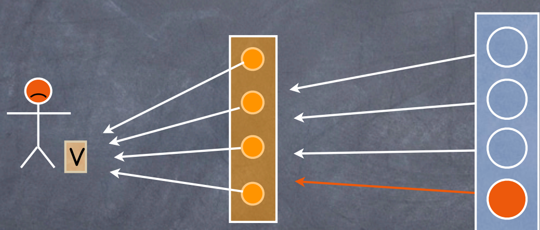


- Separating agreement and execution is easy
  - No need to change agreement protocol
  - Just forward request instead of executing
- Just a couple of subtle points
  - To handle message loss, implement retransmission in E
  - Retransmission occurs only if a message is really lost

Separation enables confidentiality



Separation enables confidentiality

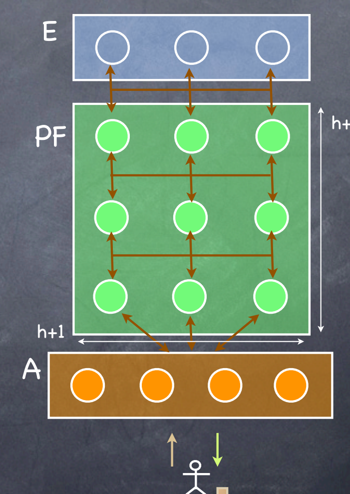


Agreement nodes can filter incorrect replies

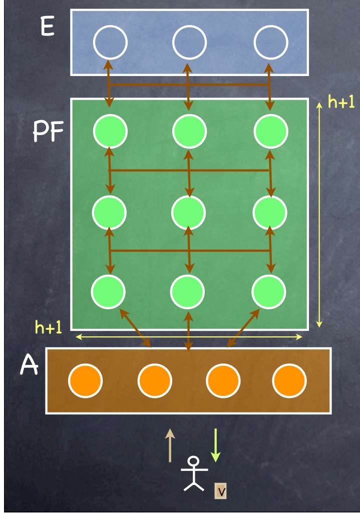
The Privacy Firewall

Three design principles:

1. Use redundant filters for fault tolerance
2. Restrict communication
3. Eliminate nondeterminism

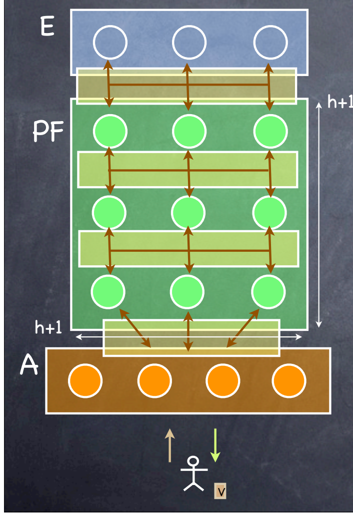


Inside the PF



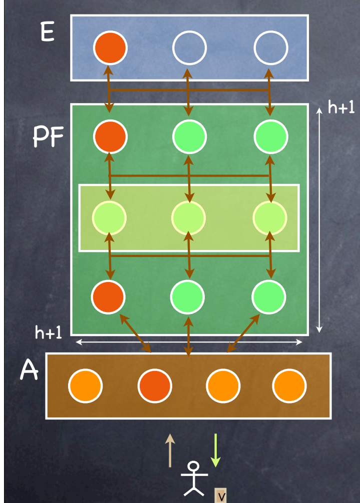
- (h+1)^2-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - h+1 rows → one is correct
- Live
  - h+1 columns → one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Inside the PF



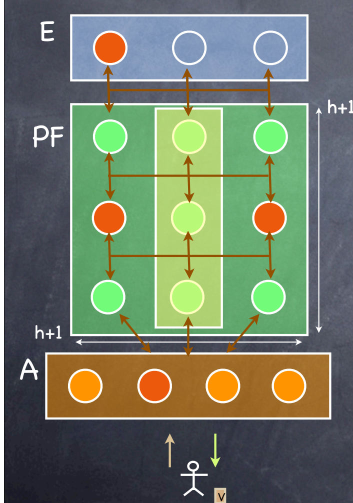
- (h+1)^2-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - h+1 rows → one is correct
- Live
  - h+1 columns → one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Inside the PF



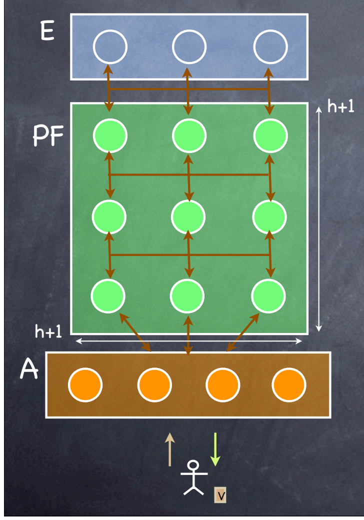
- (h+1)^2-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - h+1 rows → one is correct
- Live
  - h+1 columns → one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Inside the PF



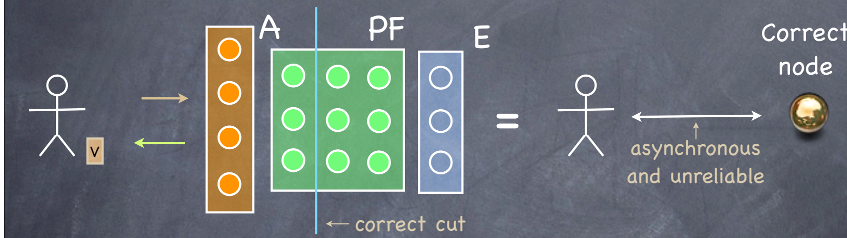
- (h+1)^2-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - h+1 rows → one is correct
- Live
  - h+1 columns → one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Inside the PF



- $(h+1)^2$-filter grid tolerates h Byzantine failures
- A filter only communicates with filters immediately above or below
- Each filter checks both reply and request certificates
- Safe
  - h+1 rows → one is correct
- Live
  - h+1 columns → one is correct
- Restricts nondeterminism
  - threshold cryptography for replies
  - cluster A locks rsn
  - controlled message retransmission

Privacy Firewall guarantees



Output-set confidentiality

Output sequence through of correct cut is a legal sequence of outputs produced by a correct node accessed through an asynchronous, unreliable link

Timing Attacks

- Faulty node in E can influence response latency

Timing Attacks

- Faulty node in E can influence response latency

Timing Attacks

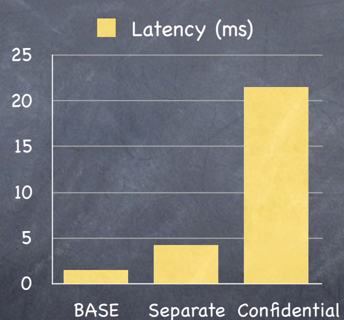
- Faulty node in E can influence response latency

- Information theoretic confidentiality appears impossible without synchrony

Prototype

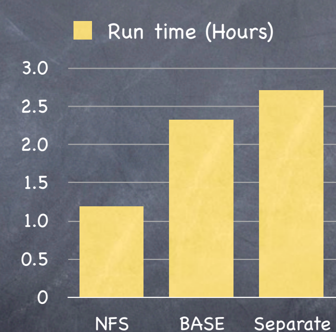
- Built on top of BASE (RCL '01)
- Implements BFT-confidential NFS
- 10 machines: 1 client, 4 in A and PF, 2 in A, 3 in E
  - 128 MB RAM, 100 Mbps switch
- Tolerates one fault in each of E, A, and PF
- Limitations
  - No uninterruptible power supply
  - The nodes in E are identical
  - Communication not physically restricted

Micro-Benchmark (req/resp: 40B/4KB)



No optimizations

Modified Andrew Benchmark (MAB 500)



Confidentiality adds an extra 16%

Conclusions

Trustworthy distributed systems through BFT

- A new architecture for state machine replication
  - separates agreement from execution
  - reduces the number of expensive replicas
  - improves confidentiality
  - may lead to more efficient algorithms

Conclusions

Trustworthy distributed systems through BFT

- A new architecture for state machine replication
  - separates agreement from execution
  - reduces the number of expensive replicas
  - improves confidentiality
  - may lead to more efficient algorithms
- Chinó: Quorum Systems
  - single replica may not know entire state...
  - but a quorum of replicas will
  - very active research area

Conclusions

Trustworthy distributed systems through BFT

- A new architecture for state machine replication
  - separates agreement from execution
  - reduces the number of expensive replicas
  - improves confidentiality
  - may lead to more efficient algorithms
- Quorum Systems
  - single replica may not know entire state...
  - but a quorum of replicas will
  - very active research area
- Are these the Emperor's new clothes?