DS Lecture 7.1
Replication

October 24, 2022

Michele Albano
mialb@cs.aau.dk

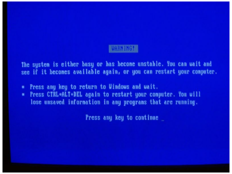
DEIS
Aalborg University
Denmark

AALBORG UNIVERSITY
DENMARK

Based on slides by Peter G. Jensen, AAU.

Goals of Replication

- Fault Tolerance
  - Transparent to user
  - Tolerates node/network failures
- High Availability
  - Service is rarely interrupted
- Performance
  - Limits of vertical scaling
  - Overcome geographic/network limits



Goals of Replication: Tolerance & Availability

Dependent

One fail = system fail.

$$uptime = (1 - p)^N$$

Independent

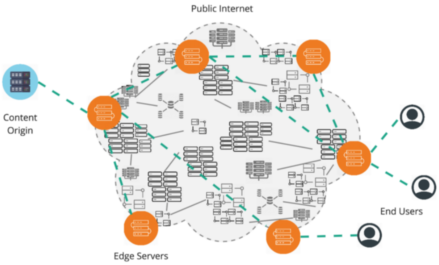
All fail = system fail.

$$uptime = 1 - p^N$$

N (p=0.05)	Availability		Yearly Downtime	
	Dep	Ind	Dep	Ind
1	95%	95%	18 days	18 days
2	90.25%	99.75%	36 days	1 day
3	85.74%	99.99%	52 days	1 h
4	81.85%	99.999%	**68 days**	**3 min**

Goals of Replication: Performance

- *Traffic on Akamai regularly peaks at more than 50 Tbps on a daily basis (2019)*
- *Google receives over 63000 searches per second on any given day (2018)*
  - Needs at least ≈ 31500 machines



Alle Shopping Billeder Bøger Mere

Ca. 276.000 resultater (0,42 sekunder)

Important Note

Caching is also replication

- Local browser cache
- Prefetching for netflix
- DNS registry

Problems

- Consensus?
  - ...or **consistency** ?
- Overhead in communication?
- Failure detection and handling?

Agenda

CAP Theorem

- Consistency
  - bank account is the same, regardless of server
- Availability
  - Bank account is always accessible, no delays
- Partition Tolerance
  - Loss of connection will not disturb bank-service

Problem

How to design such a system?

CAP Theorem

Theorem

*It is **impossible** for a distributed computer system to simultaneously provide **Consistency**, **Availability** and **Partition Tolerance**.*
*A distributed system **can satisfy any two** of these guarantees at the same time, but **not all three**.*

The Choice

Partition

The Choice

Partition

Drop Availability

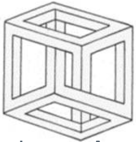
Indefinite wait

The Choice

Partition

Drop Availability

Drop Consistency



Indefinite wait

Inconsistent Answers

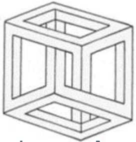
The Choice

Partition

Drop Availability

Drop Consistency

Indefinite wait



Inconsistent Answers

But wait. . .

Relaxed consistency requirements avoids impossibility.

Examples

- CP Systems
- AP Systems
- CA Systems

Examples

- CP Systems
  - Financial sector
  - Simulation (weather forecast)
  - CERN
- AP Systems
- CA Systems

Examples

- CP Systems
  - Financial sector
  - Simulation (weather forecast)
  - CERN
- AP Systems
  - Social networks
  - Search engines
  - Emails
- CA Systems

Examples

- CP Systems
  - Financial sector
  - Simulation (weather forecast)
  - CERN
- AP Systems
  - Social networks
  - Search engines
  - Emails
- CA Systems
  - Single server systems
  - Modern CPUs

Examples

- CP Systems
  - Financial sector
  - Simulation (weather forecast)
  - CERN
- AP Systems
  - Social networks
  - Search engines
  - Emails
- CA Systems
  - Single server systems
  - Modern CPUs

Application Dictates

Core/critical services are often CP.

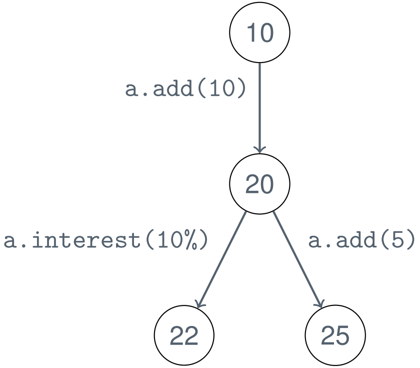
Assumptions

- Async system
- Reliable communication
- Crash-fail
- Atomic operations
- Objects are “state machines”
  - no random
  - no timer
  - no external events

Notation

`o.m(v)` = apply modifier `m` to object `o` with value `v`

`myAccount.deposit(1000)`



Requirements

- Transparent for user
- Consistent in replicated objects

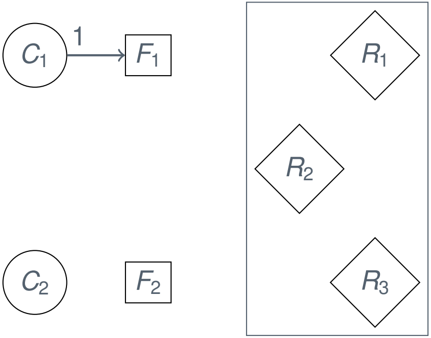
Ideal

Indistinguishable from single copy behavior

Operations

Generalized workflow

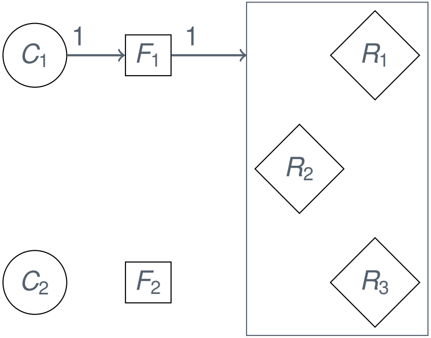
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

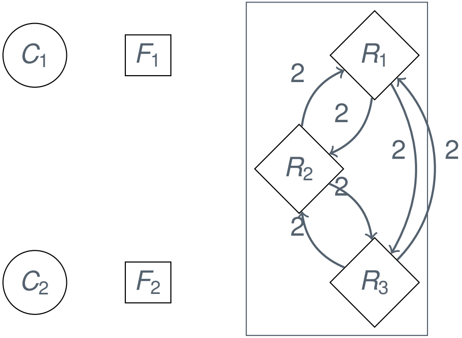
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

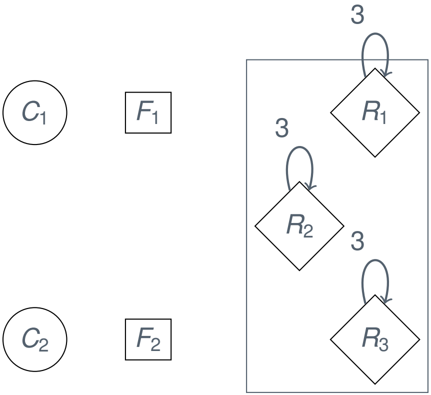
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

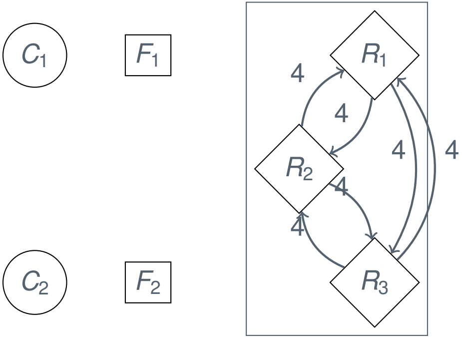
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

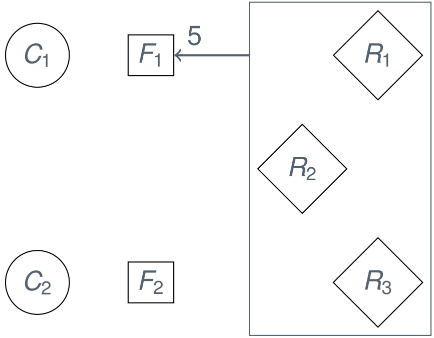
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

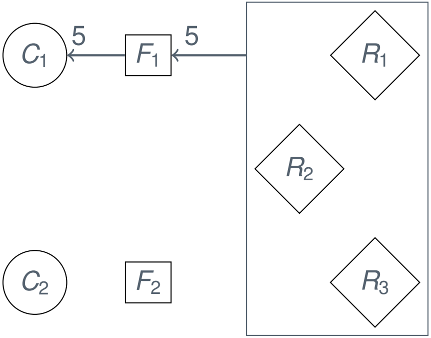
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

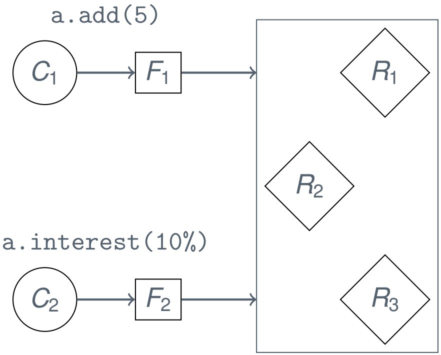
1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Operations

Generalized workflow

1. Request
2. Coordination
3. Execution
4. Agreement
5. Response



Fault Tolerance

Goal

- f-resilient replication
- No downtime
- Transparent to clients

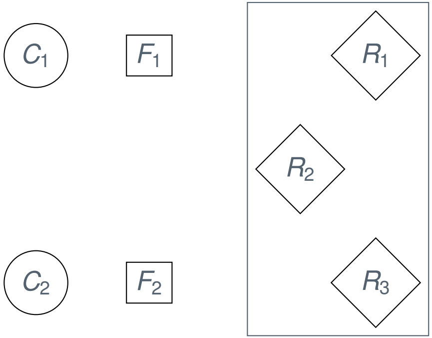
Notice

Transparent to clients is not yet formally defined.

Consistency Models

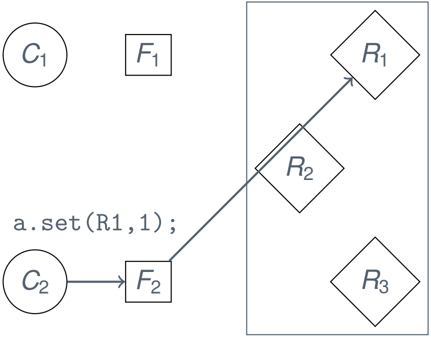
- Strong consistency
  - In real-time, after update *A*, everybody will see the modification done by *A* when reading

Inconsistency



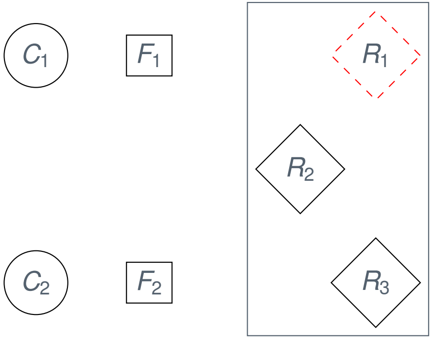
1. Initial a=0, b=0

Inconsistency



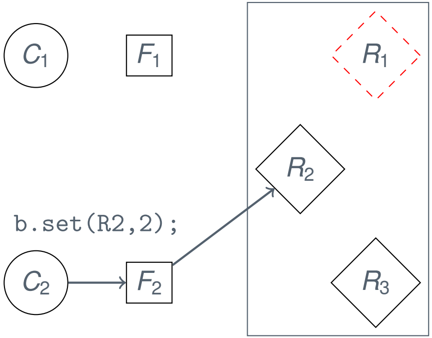
1. Initial `a=0, b=0`
2. `C2:  a.set(R1,1)`

Inconsistency



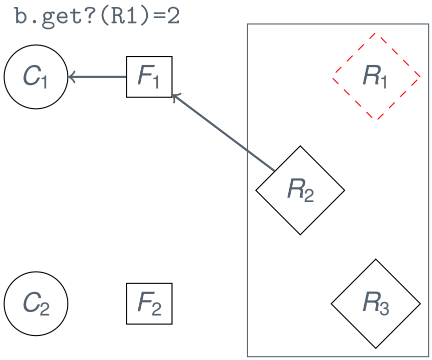
1. Initial `a=0, b=0`
2. `C2:   a.set(R1,1)`
3. `R1:   Crash`

Inconsistency



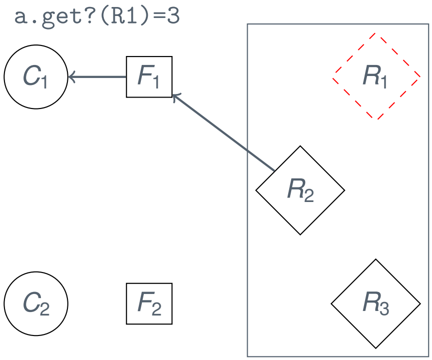
1. Initial a=0, b=0
2. C2: a.set(R1,1)
3. R1: Crash
4. C2: b.set(R2,2)

Inconsistency



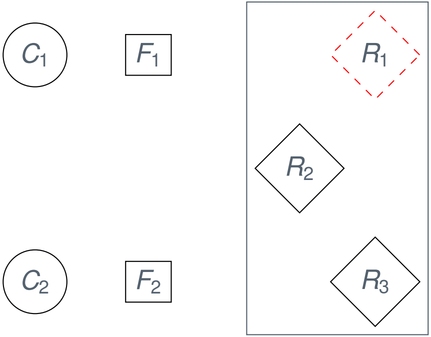
1. Initial `a=0`, `b=0`
2. `C2:  a.set(R1,1)`
3. `R1:  Crash`
4. `C2:  b.set(R2,2)`
5. `C1:  b.get?(R2)` $\rightarrow 2$

Inconsistency



1. Initial a=0, b=0
2. `C2:   a.set(R1,1)`
3. `R1:   Crash`
4. `C2:   b.set(R2,2)`
5. `C1:   b.get?(R2)` $\rightarrow$ 2
6. `C1:   a.get?(R2)` $\rightarrow$ 0

Inconsistency



1. Initial a=0, b=0
2. C2: a.set(R1,1)
3. R1: Crash
4. C2: b.set(R2,2)
5. C1: b.get?(R2) $\rightarrow$ 2
6. C1: a.get?(R2) $\rightarrow$ 0

Inconsistent!

Desired Temporal Consistencies

- if I write a value, I will see that (or a newer value) on a subsequent read
- if I read twice, the value returned on the second read is at least as new as from the first read
- if data is related (questions and answers), I expect this to be reflected in a consistent manner
  - . . . no constraints on unrelated data!

Linearizability (Lamport)

C_i operations

$o_1^i, o_2^i, \ldots, o_n^i$ for some operation $o \in O$

Timestamp

Let $T(o_n^i)$ be the timestamp of o_n^i.

Linearizability

An interleaving $\ldots, o_5^i, o_{100}^j, o_6^i \ldots$ (with $i \neq j$) is linearizable if

- arrive at a (single) correct copy of the object (from specification)
- the order is consistent with real time
  - $T(o_5^i) \leq T(o_{100}^j) \leq T(o_6^i)$.

Linearizability
Problems

Implementation

- Sync hardware clock on multiple machines
- Guess maximal network delay *D*
  - keep operation in hold-back queue until age *D*
  - keep hold-back queue sorted

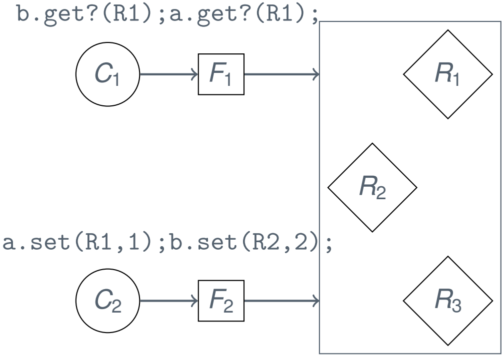
Drawbacks

- No accurate clock synchronization algorithm
  - Reasonably accurate versions exists (depends on *D*)
- No hard deadline in async setting

Consistency Models

- Strong consistency
  - In real-time, after update *A*, everybody will see the modification done by *A* when reading
- Weak consistency
  - What is the ordering, disregarding real-time?
  - “reasonably consistent”

Interleavings



a.set(R1, 1)	b.set(R2, 2)	b.get?(R1)	a.get?(R1)
a.set(R1, 1)	b.get?(R1)	b.set(R2, 2)	a.get?(R1)
a.set(R1, 1)	b.get?(R1)	a.get?(R1)	b.set(R2, 2)
b.get?(R1)	a.set(R1, 1)	b.set(R2, 2)	a.get?(R1)
b.get?(R1)	a.set(R1, 1)	a.get?(R1)	b.set(R2, 2)
b.get?(R1)	a.get?(R1)	a.set(R1, 1)	b.set(R2, 2)

Sequential Consistency (Lamport)

C_i operations

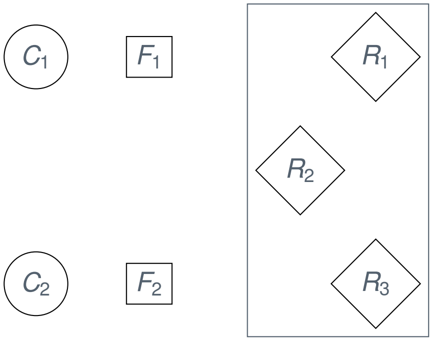
$o^i_1, o^i_2, \ldots, o^i_n$ for some operation $o \in O$

Sequential Consistency

An interleaving $\ldots, o^i_a, o^j_b, o^i_c \ldots$ (with $i \neq j$) is sequentially consistent if

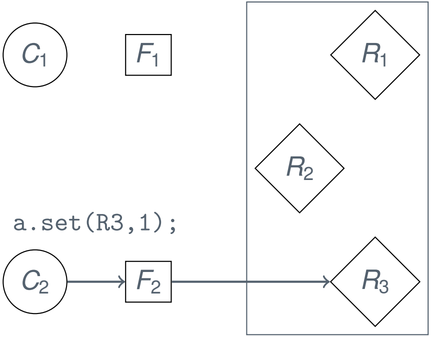
- arrive at a (single) correct copy of the object (from specification)
- the order respects causality of C_i.
  - $a < c$, i.e. from C_i, o^i_a was sent before o^i_c.

Sequentially Consistent



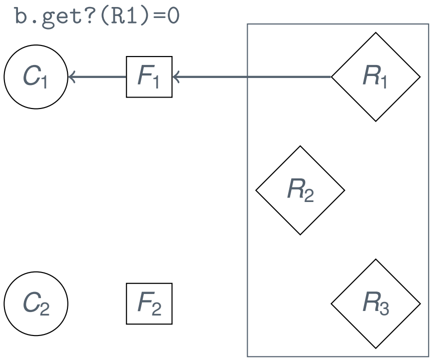
1. Initial `a=0, b=0`

Sequentially Consistent



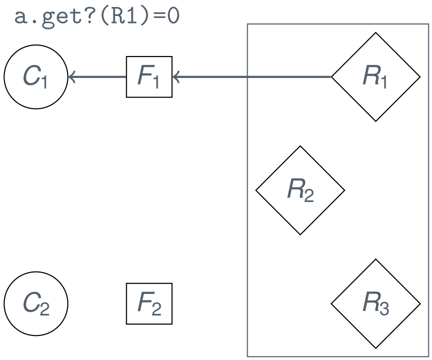
1. Initial `a=0, b=0`
2. `C2:   a.set(R3,1)`

Sequentially Consistent



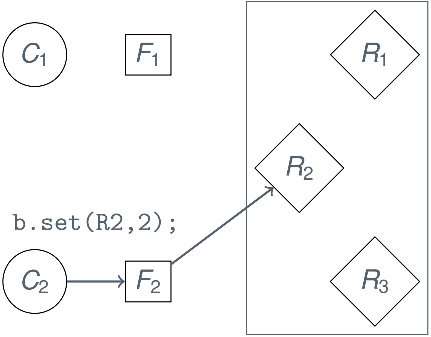
1. Initial a=0, b=0
2. C2: a.set(R3,1)
3. C1: b.get?(R1) $\rightarrow$ 0

Sequentially Consistent



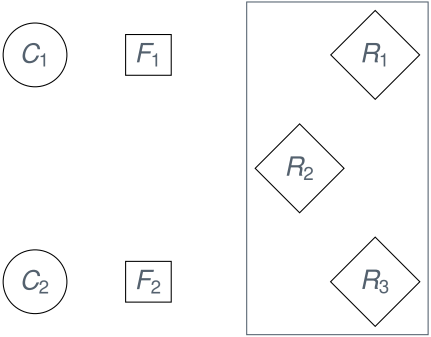
1. Initial `a=0`, `b=0`
2. `C2:` `a.set(R3,1)`
3. `C1:` `b.get?(R1)` $\rightarrow$ 0
4. `C1:` `a.get?(R1)` $\rightarrow$ 0

Sequentially Consistent



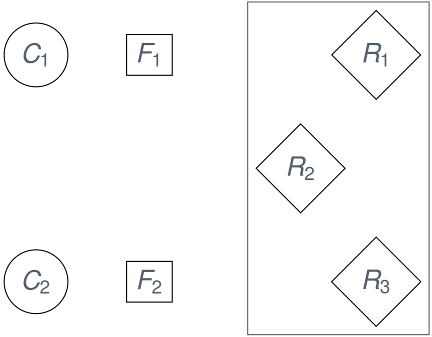
1. Initial `a=0`, `b=0`
2. `C2:    a.set(R3,1)`
3. `C1:    b.get?(R1)` $\rightarrow$ 0
4. `C1:    a.get?(R1)` $\rightarrow$ 0
5. `C2:    b.set(R2,2)`

Sequentially Consistent



1. Initial `a=0, b=0`
2. `C2:  a.set(R3,1)`
3. `C1:  b.get?(R1)` $\rightarrow$ 0
4. `C1:  a.get?(R1)` $\rightarrow$ 0
5. `C2:  b.set(R2,2)`

Sequentially Consistent



1. Initial `a=0, b=0`
2. `C2:   a.set(R3,1)`
3. `C1:   b.get?(R1)` $\rightarrow$ 0
4. `C1:   a.get?(R1)` $\rightarrow$ 0
5. `C2:   b.set(R2,2)`

Sequentially Consistent

Not Linearizable

Replication Architectures for Fault Tolerance

Read-only replication

- Immutable files
- Cache-servers

Passive replication (primary/secondary)

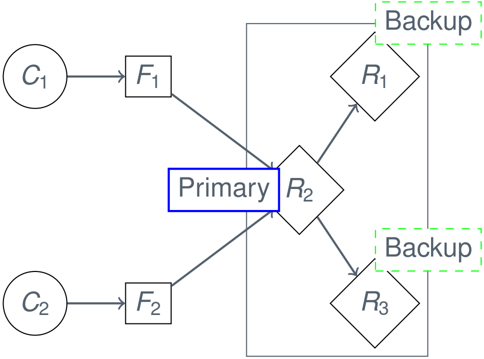
- High consistency
- Banks?

Active Replication

- Fast failover mechanism
- Workload distribution

Passive Replication

1. **Request:** Through primary replica
2. **Coordination:** Primary dictates
3. **Execution:** Apply to primary
4. **Agreement:** Send value to backups
5. **Response:** Reply after backups
   `ACK`



Passive Replication

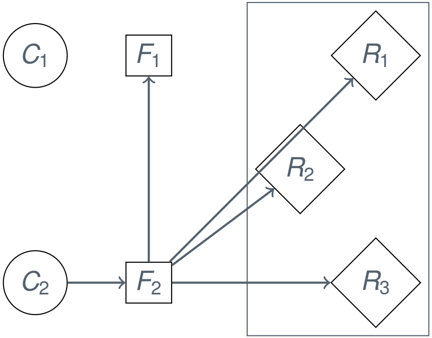
- "just follow primary"
- Up to $n - 1$ crashes
- No byzantine failure
- Linearizable (wrt. clock of primary)
- Large overhead of failure

Note

Sacrifice linearizability => offload reads to backups!

Active Replication

1. **Request:** Fs totally ordered reliable multicast to all replicas (and Fs)
2. **Coordination:** Requests delivered in total order
3. **Execution:** Execute as received
4. **Agreement:** Not needed
5. **Response:** Byzantine, wait for $(n/2)$ agreements, otherwise send first response.



Active Replication

- Sequentially consistent
- RTO-multicast
  - Impossible in async
  - Expensive otherwise
- Handles byzantine nodes
  - assuming signed messages, $(n/2) - 1$ failures
- Failover = cheap
  - Just exclude failed from group
  - “same procedure”
- Read can be trivially distributed

Availability

Availability VS Fault Tolerance

- We care less about consistency
- Higher uptime = better
- Faster response times

Example

- Read-only: caches
- Most web-scaled services
  - Youtube, facebook, stackoverflow, . . .

We study. . .

. . . the gossip architecture

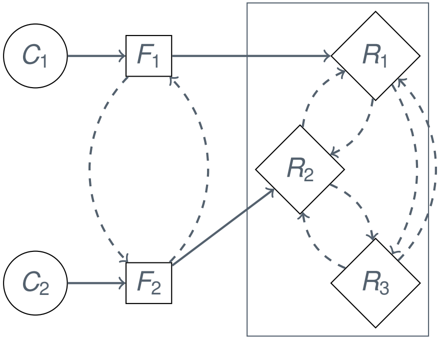
Gossip Architecture

Operations

- Read
  - no state change
- Write (Update)
  - can change state of object

Relaxed Consistency

- *R*'s apply operations "eventually" with specific order
- Client may receive outdated data
- . . . though never older than clients current data



Gossip Architecture

Reads

Causal ordering

Writes

Choice of clients

- **Causal order**
- Forced (Total + Causal) order
- Immediate ordering
  - Immediate-ordered updates: applied in a consistent order relative to any other update at all replica managers
  - Forced-order update and a causal-order update that are not related by the happened-before relation may be applied in different orders at different replica managers

Gossip Architecture
Idea

Vector clocks, vector clocks everywhere

Track “number of unique updates R_i has seen of object from *some* frontend” as a vector.

- Each entry in vector-clock corresponds to R_i
  - R_i updates own index in vector on update from some F_i
  - Keep messages from future in hold-back queue
  - Avoid duplicates
- Frontends keep track of “last known” time-stamp
  - Frontends label their reads/writes with last-known time-stamp
  - Receive new timestamp updates from R_i (or via gossip).

Gossip Architecture

Phases

1. **Request:** *F*s forwards to a single *R* (or more)
2. **Coordination:** Queue request until order is respected
3. **Execution:** Execute in correct order
4. **Agreement:**
   - Wait for gossip
   - Request missing data
5. **Response:**
   - Read: await coordination
   - Write: immediately

Gossip Architecture

Frontend View

Frontend

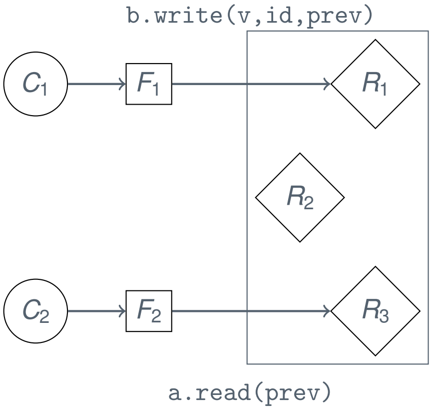
Keep a vector timestamp *prev*

On read/write operation *o* from client

1. Send $(o, prev)$ to some R_i
2. Wait for response
3. Received *new* is merged with *prev*.
4. Gossip/piggyback with other clients

Notice

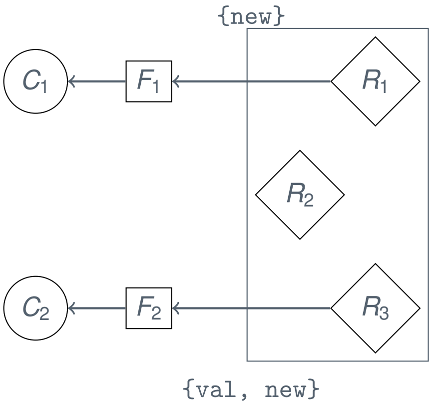
F_i may communicate with different R_j each time.



Gossip Architecture
Frontend View

Frontend
Keep a vector timestamp *prev*
On read/write operation *o* from client
1. Send $(o, prev)$ to some R_i
2. Wait for response
3. Received *new* is merged with *prev*.
4. Gossip/piggyback with other clients

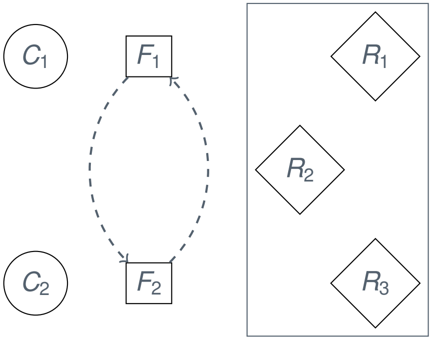
Notice
F_i may communicate with different R_j each time.



Gossip Architecture
Frontend View

Frontend
Keep a vector timestamp *prev*
On read/write operation *o* from client
1. Send $(o, prev)$ to some R_i
2. Wait for response
3. Received *new* is merged with *prev*.
4. Gossip/piggyback with other clients

Notice
F_i may communicate with different R_j each time.



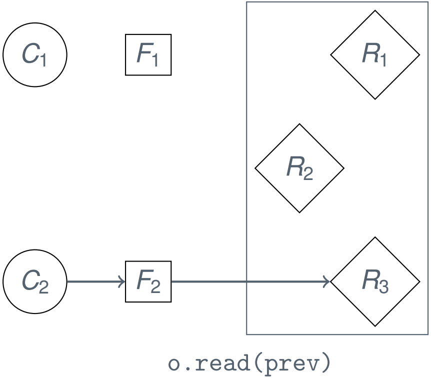
Gossip Architecture
Replication Managers View

Replication Managers - Read

Value v, value timestamp vts ...

On read operation o from F_2

1. Got $(o, prev_i)$ from F_i
2. if $prev \leq vts$
   - return (v, vts) instantly



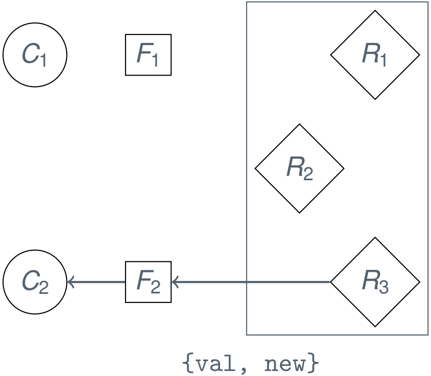
Gossip Architecture
Replication Managers View

Replication Managers - Read

Value v, value timestamp vts ...

On read operation o from F_2

1. Got $(o, prev_i)$ from F_i
2. if $prev \leq vts$
   - return (v, vts) instantly



Gossip Architecture

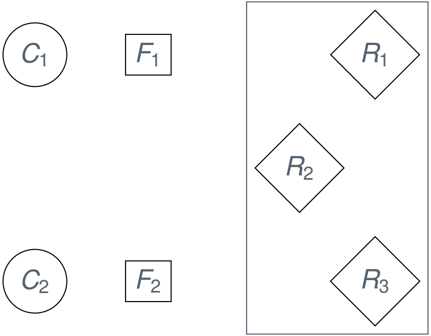
Replication Managers View

Replication Managers - Read

Value v, value timestamp vts ...

On read operation o from F_2

1. Got $(o, prev_i)$ from F_i
2. if $prev \leq vts$
   - return (v, vts) instantly



Example of vector-clock use

$prev = (1, 2, 3)$ and $vts = (1, 1, 1)$ =

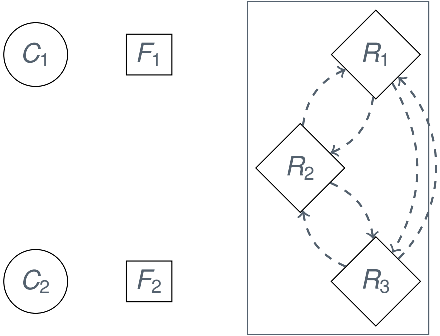
Missing 1 update from R_2 and 2 updates from R_3

Gossip Architecture
Replication Managers View

Replication Managers - Read

Value v, value timestamp vts ...
On read operation o from F_2

1. Got $(o, prev_i)$ from F_i
2. if $prev \leq vts$
   - return (v, vts) instantly
3. Otherwise, wait for gossip
4. ...or request missing



Example of vector-clock use

$prev = (1, 2, 3)$ and $vts = (1, 1, 1)$ =
Missing 1 update from R_2 and 2 updates from R_3

Gossip Architecture

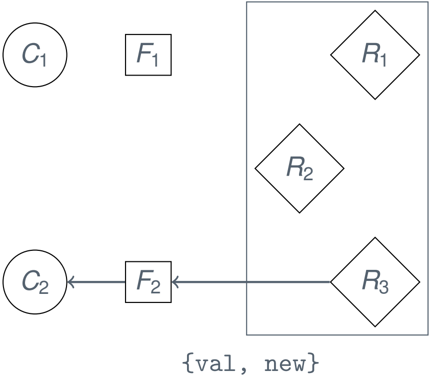
Replication Managers View

Replication Managers - Read

Value v, value timestamp vts ...

On read operation o from F_2

1. Got $(o, prev_i)$ from F_i
2. if $prev \leq vts$
   - return (v, vts) instantly
3. Otherwise, wait for gossip
4. ...or request missing
5. Reply when $prev \leq vts$



Example of vector-clock use

$prev = (1, 2, 3)$ and $vts = (1, 1, 1) =$

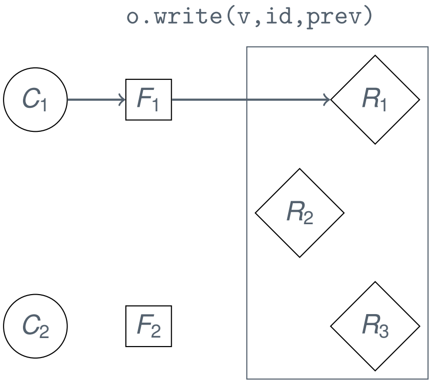
Missing 1 update from R_2 and 2 updates from R_3

Gossip Architecture
Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i

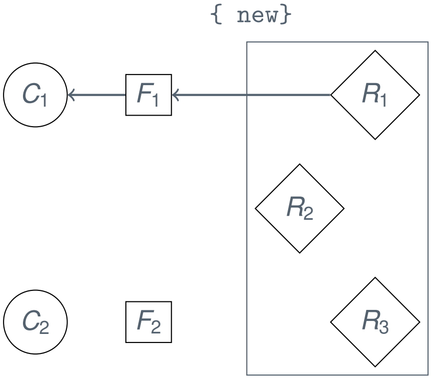


Gossip Architecture
Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts

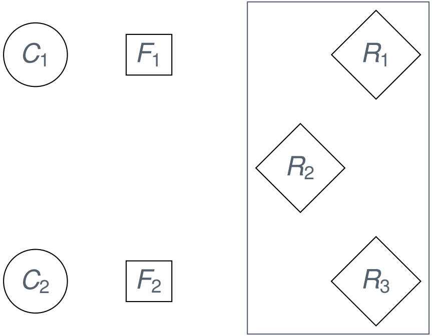


Gossip Architecture
Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts
3. Increment rts_i



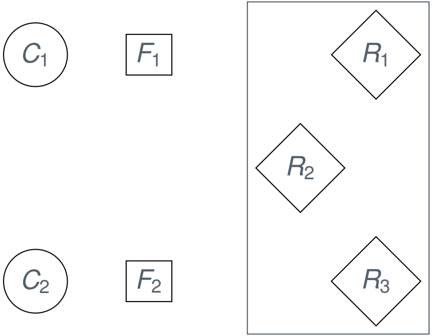
Gossip Architecture

Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts
3. Increment rts_i
4. Let $prev' = prev$ but with $prev'_i = rts_i$



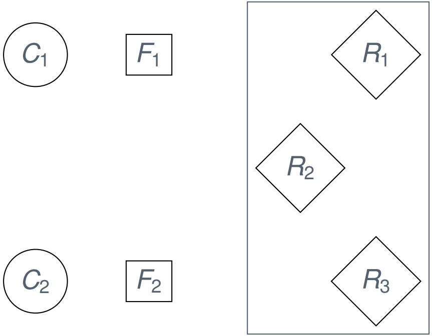
Gossip Architecture

Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts
3. Increment rts_i
4. Let $prev' = prev$ but with $prev'_i = rts_i$
5. Store in log with $prev'$ as time-stamp



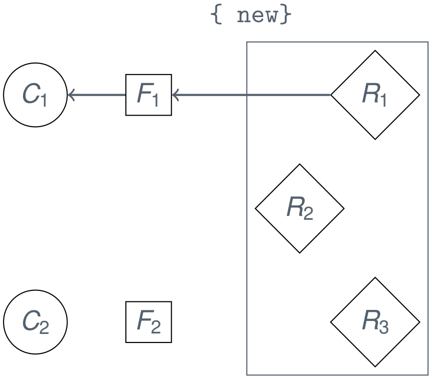
Gossip Architecture

Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts
3. Increment rts_i
4. Let $prev' = prev$ but with $prev'_i = rts_i$
5. Store in log with $prev'$ as time-stamp
6. Return $prev$ with $prev_i = rts_i$ to F_i

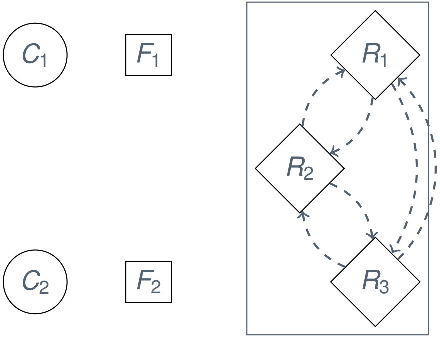


Gossip Architecture
Replication Managers View

Replication Managers - Write

$(v, vts, log, rts, executed, \ldots)$

1. Got $(v, id, prev)$ from F_i
2. If $id \in executed$ return rts
3. Increment rts_i
4. Let $prev' = prev$ but with $prev'_i = rts_i$
5. Store in *log* with $prev'$ as time-stamp
6. Return *prev* with $prev_i = rts_i$ to F_i
7. Gossip, execute and cleanup *log* in causal order



Gossip Architecture
Replication Managers View

Replication Managers - Execute and Gossip

$(v, vts, log, rts, executed, \ldots)$

On read/write operation o from F_2

1. Wait for entry in *log* to become stable
   - $entry.prev \leq vts$
   - Keep track of executed op-ids, skip duplicates
2. Clear out log when all are guaranteed to have delivered
3. Merge own & senders time-stamp on gossip

Details

Frequency of gossip

- Minutes, hours or days
- Depend on the requirement of application

Topology

- Random
- Deterministic: investigate known clocks?
- Topological: Mesh, circle, tree
- Geographical

Discussion

- Works even with network partition
  - . . . but may need conflict resolution
- More *R*'s = more gossip
- Larger delays between gossip
  - Larger consistency gaps
  - Higher latency
- Good when conflicting updates are rare