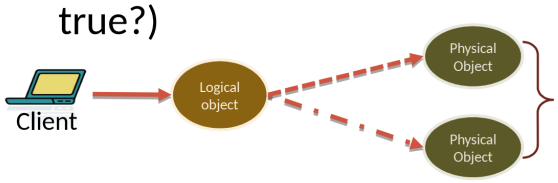
Replication

What is Replication?

- Make multiple copies of a data object and ensure that all copies are identical
- Two Types of access; reads, and writes (updates)
- Reasons, have a backup plan:
  - Handle more work (e.g. web-servers)
  - Keep data safe (fault tolerance)
  - Reduce latencies (CDN's and Caching)
  - Keep data available

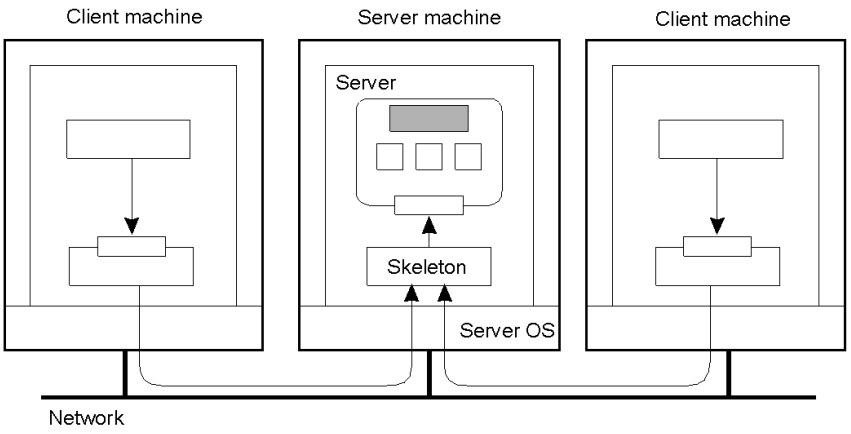
Replication requirements

- Transparency (illusion of a single copy)
  - Clients must be unaware of replication
- Consistency
  - Obtain identical results from different copies (is that true?)



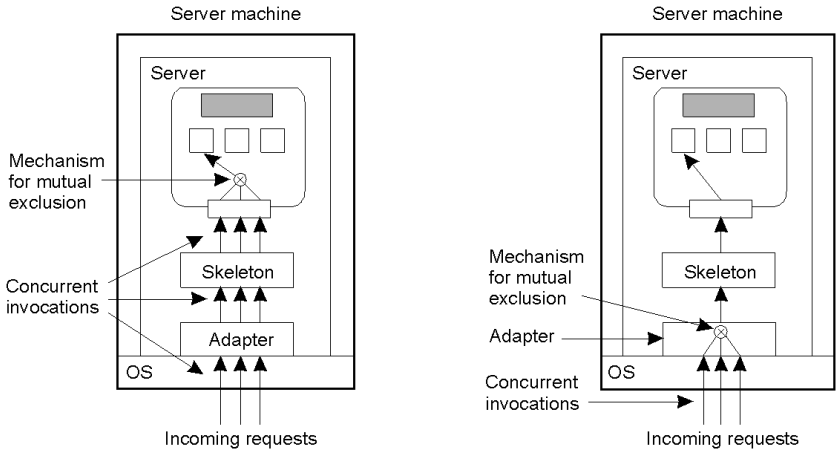
Not always identical:
- Some have received updates

Object Replication (1)



Organization of a distributed remote object shared by two different clients.

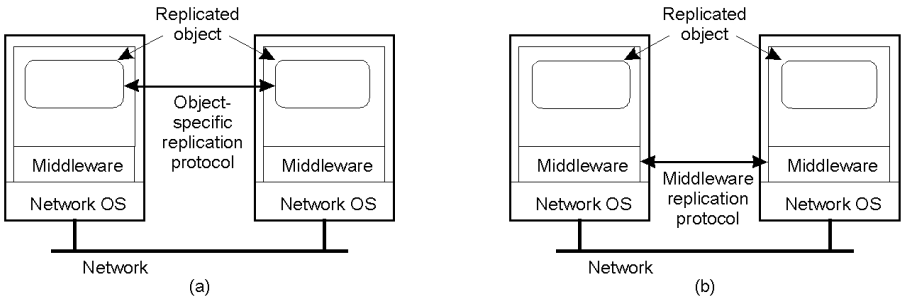
Object Replication (2)



a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

Object Replication (3)



a) A distributed system for replication-aware distributed objects.

b) A distributed system responsible for replica management

Reasons for Replication (1)

- Reliability
  - **Redundancy** is a key technique to increase availability. If one server crashes, then there is a replica that can still be used. Thus, failures are tolerated by the use of redundant components. Examples:
    - There should always be at least two different routes between any two routers in the internet.
    - In the Domain Name System, every name table is replicated in at least two different servers.
    - A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server.

Trade-off: there is some cost associated with the maintenance of different replicas.

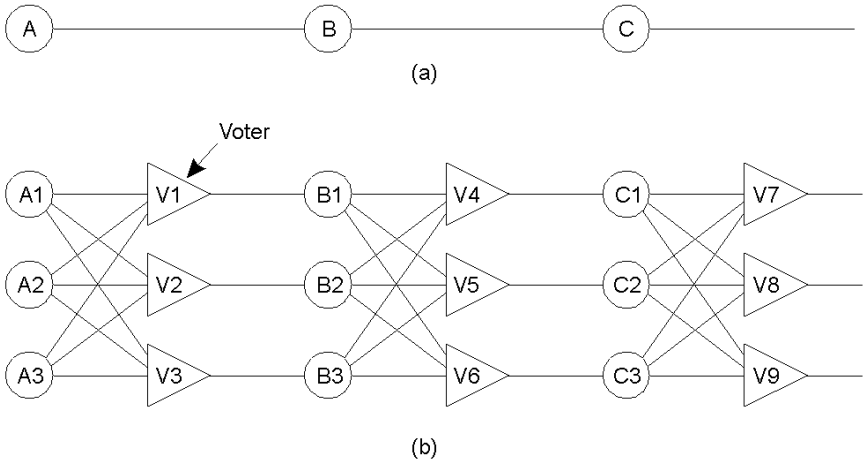
Redundancy

- Wouldn't you prefer to use an **expensive** airplane with triple-redundancy for all hardware resources [1] as opposed to a **cheap** airplane with a single-resource (no redundancy)?
- **Availability** of a replicated service over a period of time:
  - 1 – Probability(all replicas have failed)
  - Probability(all replicas have failed) = Probability(replica 1 has failed) × Probability(replica 2 has failed) × Probability (replica 3 has failed).
  - Probabilities are between 0 (=no chance/impossible) and 1 (=certainty).
- **Example**: if the probability of a system failing is 0.3 (30%), the probability of two copies of the system failing at the same time is 0.3x0.3=0.09, the probability of three copies all failing at the same time is 0.027, the probability of ten copies all failing at the same time is 0.000005905. This means that with ten copies we can have availability of 99.9994095%.
- To compute probability of failure (p) of a single node, use mean time between failures (f) and mean time to repair a failure(t): $p = 1 - t/(f+t)$
- The degree of redundancy has to strike a balance with the additional cost needed to implement it!

[1] "Triple-triple redundant 777 primary flight computer". Proceedings of the 1996 IEEE Aerospace Applications Conference.

Failure Masking by Redundancy

(see Tanenbaum, fig 8.2, p.327)



Triple modular redundancy

Reasons for Replication (2)

- Performance
  - By placing a copy of the data in the proximity of the process using them, the time to access the data decreases. This is also useful to achieve scalability. Examples:
    - A server may need to handle an increasing number of requests (e.g., google.com, ebay.com). Improve performance by replicating the server and subsequently dividing the work.
    - **Caching**: Web browsers may store locally a copy of a previously fetched web page to avoid the latency of fetching resources from the originating server. (cf. with processor architectures and cache memory)

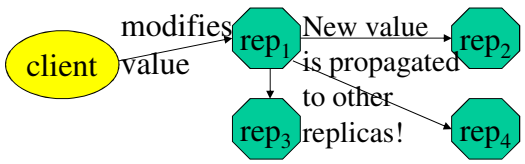
How many replicas?

(or, how do we trade cost with availability?)

- **Capacity Planning**: the process of determining the necessary capacity to meet a certain level of demand (a concept that extends beyond distributed computing)
- Example problem:
  - Given the cost of a customer waiting in the queue;
  - Given the cost of running a server;
  - Given the expected number of requests;
  - How many servers shall we provide to guarantee a certain response time if the number of requests does not exceed a certain threshold?
- Problems like this may be solved with mathematical techniques. **Queuing theory**, which refers to the mathematical study of queues (see Gross & Harris, Fundamentals of queuing theory) may be useful.

The price to be paid…

- Besides the cost (in terms of money) to maintain replicas, what else could be against replication?
- Consistency problems:
  - If a copy is modified, this copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. When and how those modifications need to be carried out determines the price of replication. Example (4 replicas):

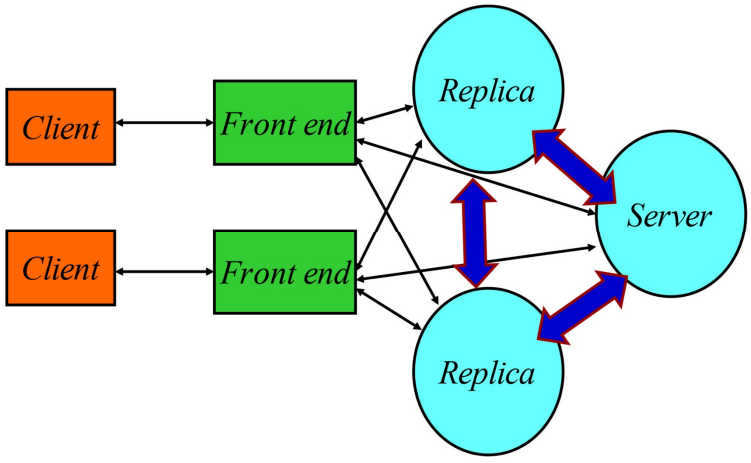


The cure may be worse than the disease!!!

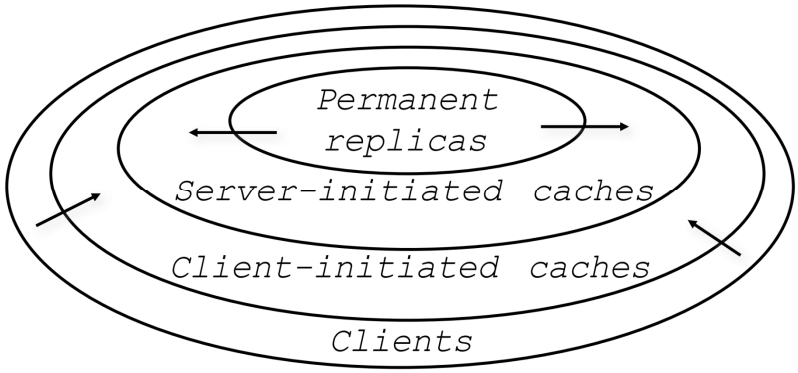
The price to be paid… (cont)

- Keeping multiple copies consistent may itself be subject to serious scalability problems!
- In the example before, when an update occurs it needs to be propagated to all other replicas. No other processes should read the same value from the other replicas before the update happened… However:
  - What if it is unlikely that there will ever be a request to read that same value from other replicas?
  - At the same time with the update, there is a request to read this value from another process – which one came first?
- Global synchronisation takes a lot of time when replicas are spread across a wide area network.
- **Solution**: Loosen the consistency constraints! So, copies may not be always the same everywhere… To what extent consistency can be loosened depends on the access and update patterns of the replicated data as well as on the purpose for which those data are used.

Replication Architecture



Placement of Replicas



Placement of Replicas

- Permanent replicas
  - Clusters of servers
  - Geographically dispersed web mirrors (Akamai)
- Server-initiated caches
  - Placement of hosting servers
- Client initiated caches
  - enterprise proxies or web browser caches

Update propagation in replicas

- Push based propagation
  - A replica pushes the update to the others
  - May push the new data or parameters of the update operation
- Pull-based propagation
  - A replica requests another replica to send the newest data it has
- Pushing data vs. pushing updates
  - Pushing updates reduces traffic

Types of ordering adapted to replication

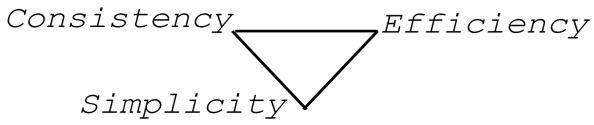
FIFO– if a client issues **r** and then **r'**, any correct Replica Manager that handles **r'** handles **r** before it

Causal– if the issuing of **r** happened-before issuing **r'**, then any correct Replica Manager that handles **r'** handles **r** before it

Total – if a correct Replica Manager handles **r** before **r'**, then any correct RM that handles **r'** handles **r** before it

Consistency issues

- A contract between the client developer and a provider of the replicated service
  - The provider guarantees that the data will be updated according to some consistency criteria
  - The application developer will need to devise applications with these criteria in mind
- "Ideal consistency": system behavior is indistinguishable from a non-replicated system
- The consistency-efficiency-simplicity triangle

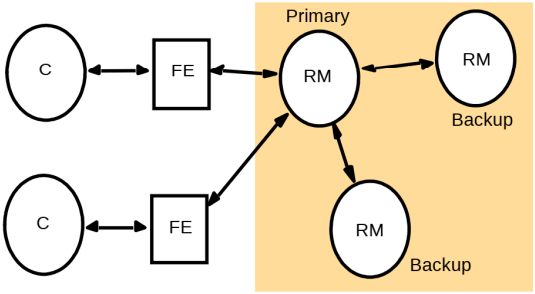


Homework

- Study different consistency models used in replication

Passive (primary-backup) replication

- One *primary* replica manager, many backup replicas
  - If primary fails, backups can take its place (election!)
- Implements linearizability if:
  - A failing primary is replaced by a unique backup
  - Backups agree on which operations were performed before primary crashed
- View-synchronous group communication!

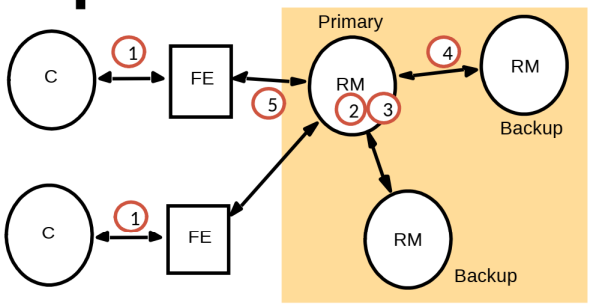


Passive (primary-backup) replication

- One server plays a special primary role
  - Performs all the updates
  - May propagate them to backup replicas eagerly or lazily
  - Maintains the most updated state
- Backup servers may take off the load of processing client requests but only if stale results are ok
- Implementable without deterministic operations
- Typically easier to implement than active replication
- Less network traffic during the normal operation but longer recovery with possible data loss
- Several sub-schemes (cold backup, warm backup, hot standby)

Steps of passive replication

1. Request
   - Front end issues request with unique ID
2. Coordination
   - Primary checks if request has been carried out, if so, returns cached response
3. Execution
   - Perform operation, cache results
4. Agreement
   - Primary sends updated state to backups, backups reply with Ack.
5. Response
   - Primary sends result to front end, which forwards to the client

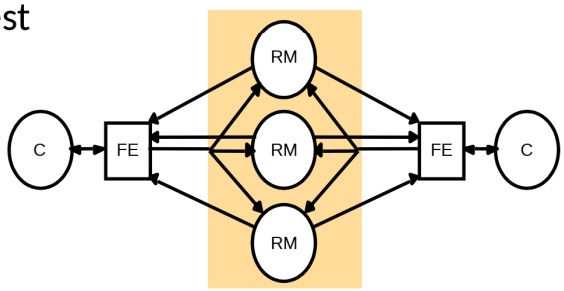


What happens if the primary RM crashes?

- ❑ Before agreement
- ❑ After agreement

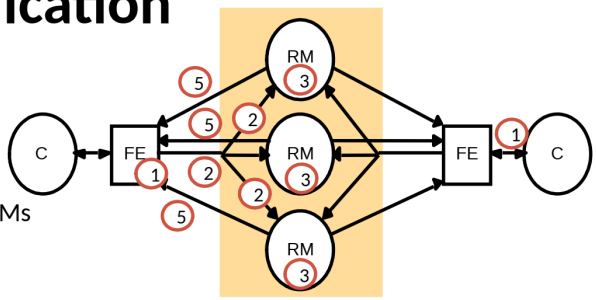
Active replication

- ❑ RMs play equivalent roles
- ❑ All replica managers carry out all operations
- ❑ Front ends multicast one request at a time (FIFO)
- ❑ Requests are totally ordered
- ❑ Implements sequential consistency
- ❑ Tolerate Byzantine failures



Steps of active replication

1. Request
   - Front end adds unique identifier to request, multicasts to RMs
2. Coordination
   - Totally ordered request delivery to RMs
3. Execution
   - Each RM executes request
4. Agreement
   - Not needed
5. Response
   - All RMs respond to front end, front end interprets response and forwards response to client

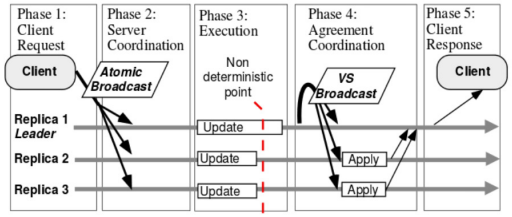


Comparing active and passive replication

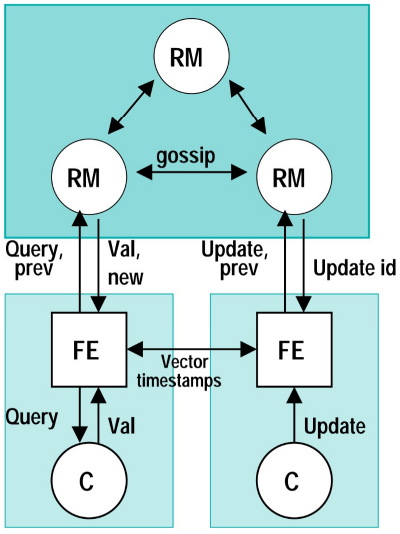
- Both handle crash failures (but differently)
- Only active can handle arbitrary failures
- Passive may suffer from large overheads
- Optimizations?
  - Send "reads" to backups in passive
    Lose linearizability property!
  - Send "reads" to specific RM in active
    Lose fault tolerance
  - Exploit commutativity of requests to avoid ordering requests in active

Semi Active Replication

- Intermediate soluyion between Active and Passive replication
- Main difference with active replication
  - each time replicas have to make a non-deterministic decision, a process, called the leader , makes the choice and sends it to the followers



The Gossip Architecture



The gossip service front end handles client operations using an application-specific API and turns them into gossip operations (queries or updates). RM updates are **lazy** in the sense that gossip messages may be exchanged only occasionally. Each front end keeps a **vector timestamp** *prev*, reflecting the latest data values accessed by the client/front end. When clients communicate directly they piggyback their vector timestamps, which are then merged.

Gossip Architecture

Processing requests

1. *Request:* The front end sends the request to a RM. Queries are usually synchronous but updates are asynchronous: the client continues as soon as the request is passed to the front end and the front end propagates the request in background.
2. *Update response:* The RM replies to the front end as soon as it has received an update.
3. *Coordination:* The RM that receives a request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages.
4. *Execution:* The RM executes the request.
5. *Query response:* If the request is a query the RM responds at this point.
6. *Agreement:* The RMs only coordinate via gossip messages.