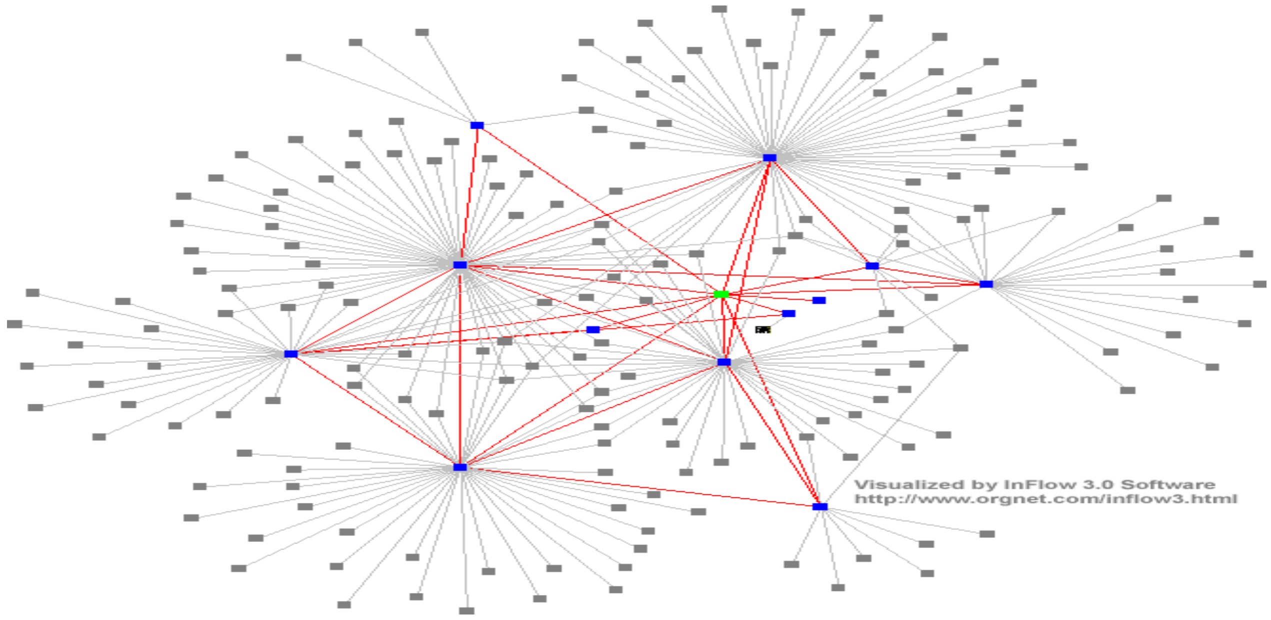
DCS Transaction Issues (Performance):

DCS Transac. Management and Concurrency Part 1


Visualized by InFlow 3.0 Software
http://www.orgnet.com/inflow3.html

Dr. Sunny Jeong. spjeong@uic.edu.cn

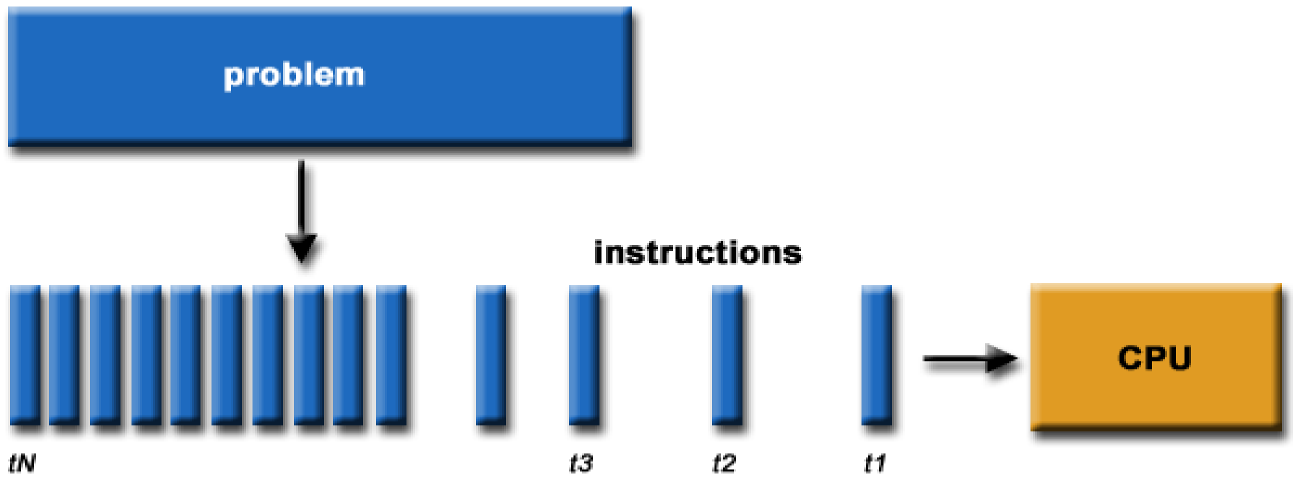
With Thanks to Prof. G. Coulouris and Prof. A.S. Tanenbaum

Overview

- Introduction to parallel computing
- Amdahl's Law/Gustafson’s Law
- Benefits of using thread in DCS
- Passive replication
- Active replication
- Distributed concurrency control

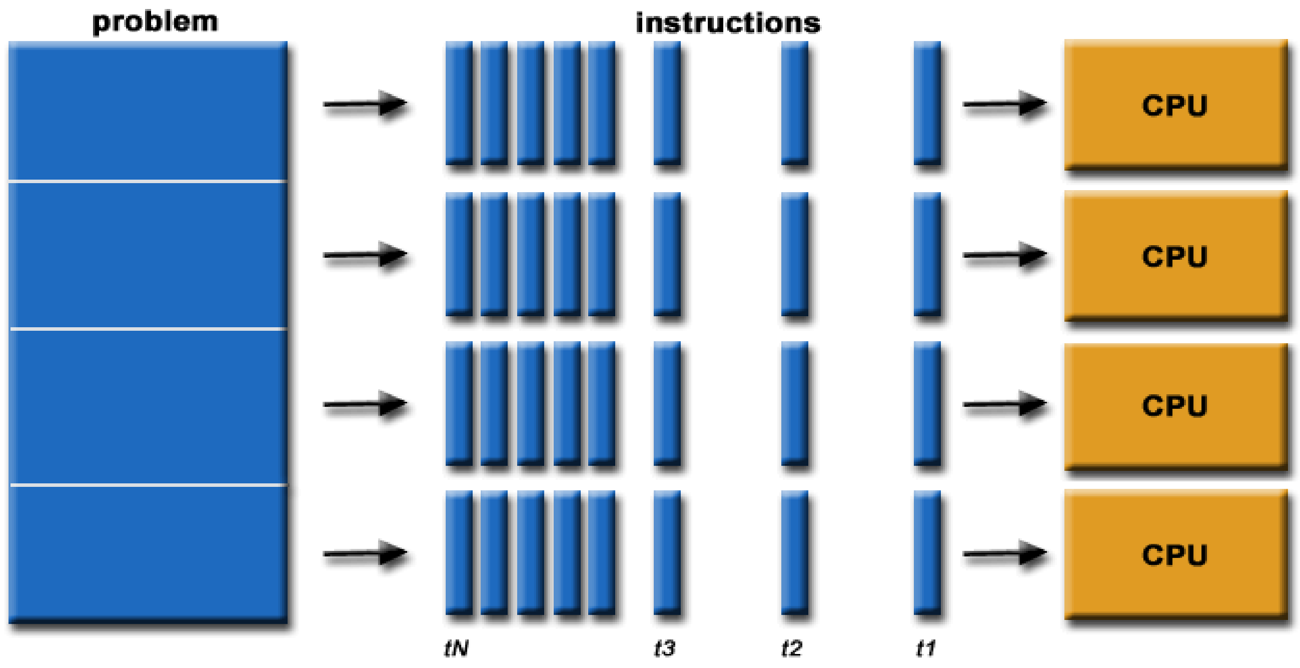
Parallel Computing

- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



Parallel Computing (1)

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Amdahl’s Law

- Gene Amdahl, chief architect of IBM's first mainframe series and founder of Amdahl Corporation and other companies found that there were some fairly stringent restrictions on how much of a speedup one could get for a given parallelized task.
- These observations were wrapped up in *Amdahl's Law*

Amdahl's Law(1)

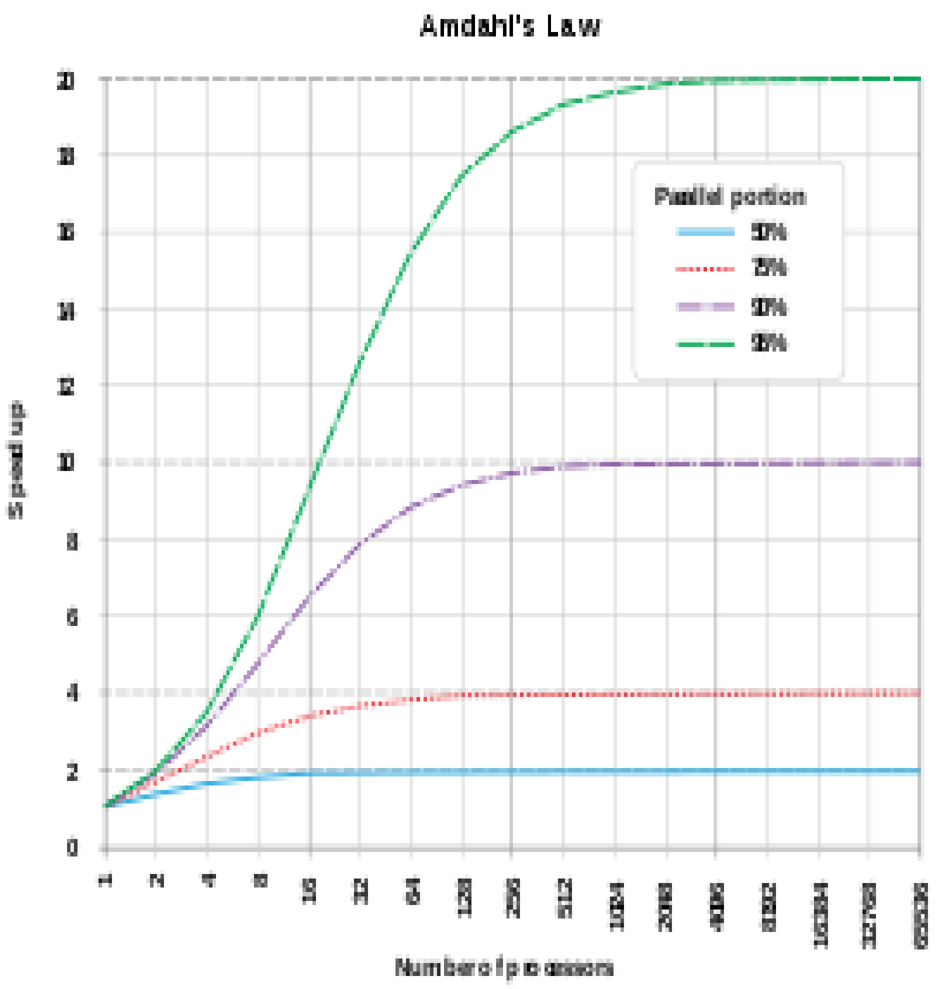
- The overall performance of a system is a result of the interaction of all of its components.
- System performance is most effectively improved when the performance of the most heavily used components is improved.
- This idea is quantified by Amdahl's Law:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

where S is the overall speedup; f is the fraction of work performed by a faster component; and k is the speedup of the faster component.

Amdahl’s Law(2)

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1-Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$



Amdahl’s Law(3)

- Amdahl’s Law gives us a handy way to estimate the performance improvement we can expect when we upgrade a system component.

Case

- On a large system, suppose we can upgrade a CPU to make it 150% faster for $10,000 or upgrade its disk drives for $7,000 to make them 250% faster.
- Processes spend 70% of their time running in the CPU and 30% of their time waiting for disk service.
- An upgrade of which component would offer the greater benefit for the lesser cost?

Amdahl's Law(4)

- The processor option offers a 130% speedup:

$$f = 0.70, \quad k = 1.5 \qquad S = \frac{1}{(1 - 0.7) + 0.7/1.5}$$

- And the disk drive option gives a 122% speedup:

$$f = 0.30, \quad k = 2.5 \qquad S = \frac{1}{(1 - 0.3) + 0.3/2.5}$$

- Each 1% of improvement for the processor costs \$333, and for the disk a 1% improvement costs \$318.

The Bottom line of Amdhal’s law

- The point that Amdahl was trying to make was that using lots of parallel processors was not a viable way of achieving the sort of speed-ups that people were looking for.

- i.e. it was essentially an argument in support of investing effort in making single processor systems run faster.

Gustafson’s Law (1)

Gustafson’s Law says that if you apply *P* processors to a task that has serial fraction *f*, scaling the task to take the same amount of time as before,
the speedup is

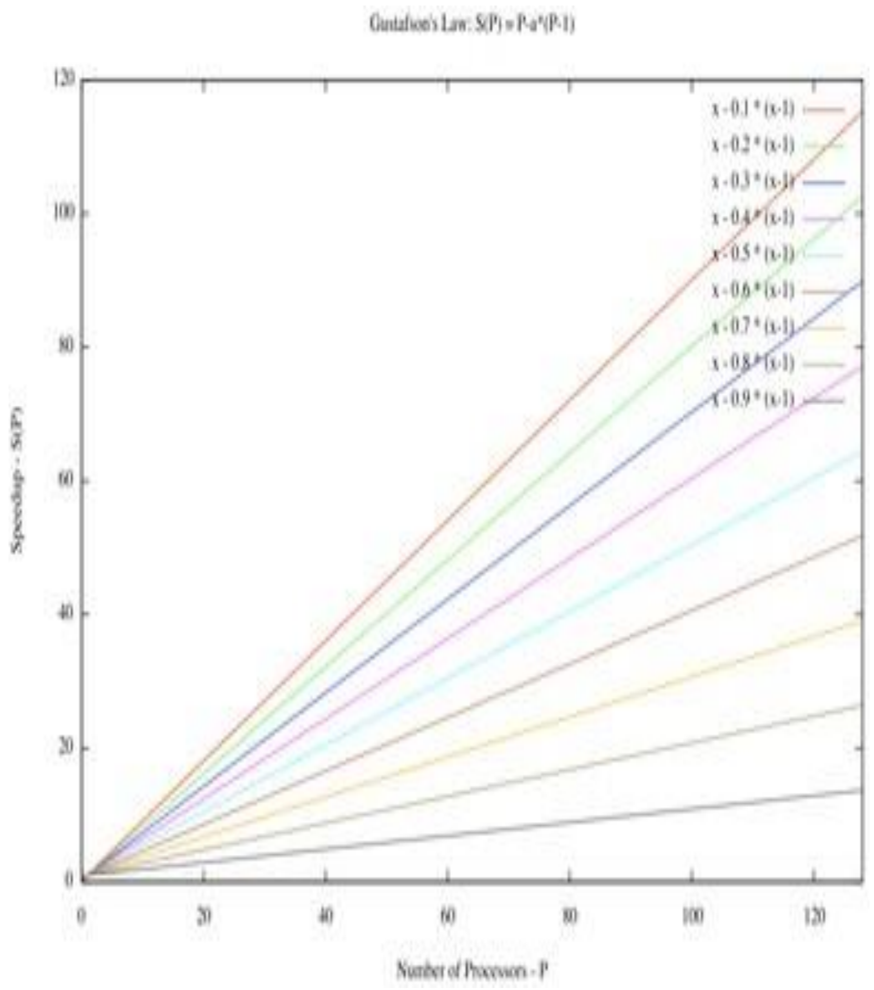
S == f+P(1−f)P−f(P−1)

Alternatively, S can be expressed
$S(P)=P- \alpha (P-1)$

where "P" is the number of processors,
"S" is the speedup, and alpha is the non-parallelizable part of the process.

Gustafson’s Law (2)

- For example, if the non-parallelised part is 50% of the total process and 128 CPUs are used, the performance improvement will be
- 128-0.5*(128-1) = 64.5 times
- Why are there non parallelised part in both?



Superlinear Speedup

You will get used to seeing $S_p < p$

On the other hand, it is also possible that $S_p > p$

This seemingly impossible condition is called *superlinear speedup*

It is quite rare in real life, but it really can happen that a program runs more than p times as fast on p processors

This can happen for a variety of reasons, some technological, and some more philosophical

https://annals-csis.org/Volume_8/drp/498.html

Superlinear Speedup(1)

Analysis

The first technological reason is due to cache memory

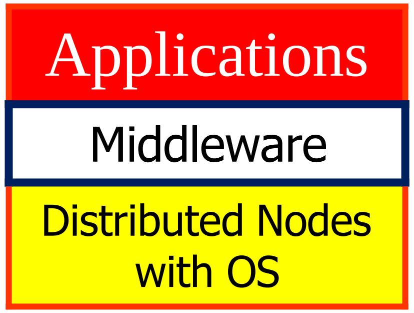
Cache memory is a lot faster than main memory so if you can fit your problem entirely in cache, it will run faster

For example, the Core i7: perhaps 200 cycles to access main memory, compared to 2 cycles for a L1 cache hit

p processors might have *p* times the cache of a single processor, so a problem spread across the processors might well fit in the larger amount of cache available

Building Distributed Systems : Support from OS

- Middleware is a layer of software (system) between Applications and Operating System (OS) powering the nodes of a distributed system.
- The OS facilitates:
  - Encapsulation and protection of resources inside servers;
  - Invocation of mechanisms required to access those resources including concurrent access/processing.



Introduction to replication

- replication can provide the following
- performance enhancement
  - e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.
  - replication of read-only data is simple, but replication of changing data has overheads
- fault-tolerant service
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - if *f* of *f*+1 servers crash then 1 remains to supply the service
  - if *f* of 2*f*+1 servers have byzantine faults then they can supply a correct service
- availability is hindered by
  - server failures
    - replicate data at failure- independent servers and when one fails, client may use another. Note that caches do not help with availability(they are incomplete).
  - network partitions and disconnected operation
    - Users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

•

Requirements for replicated data

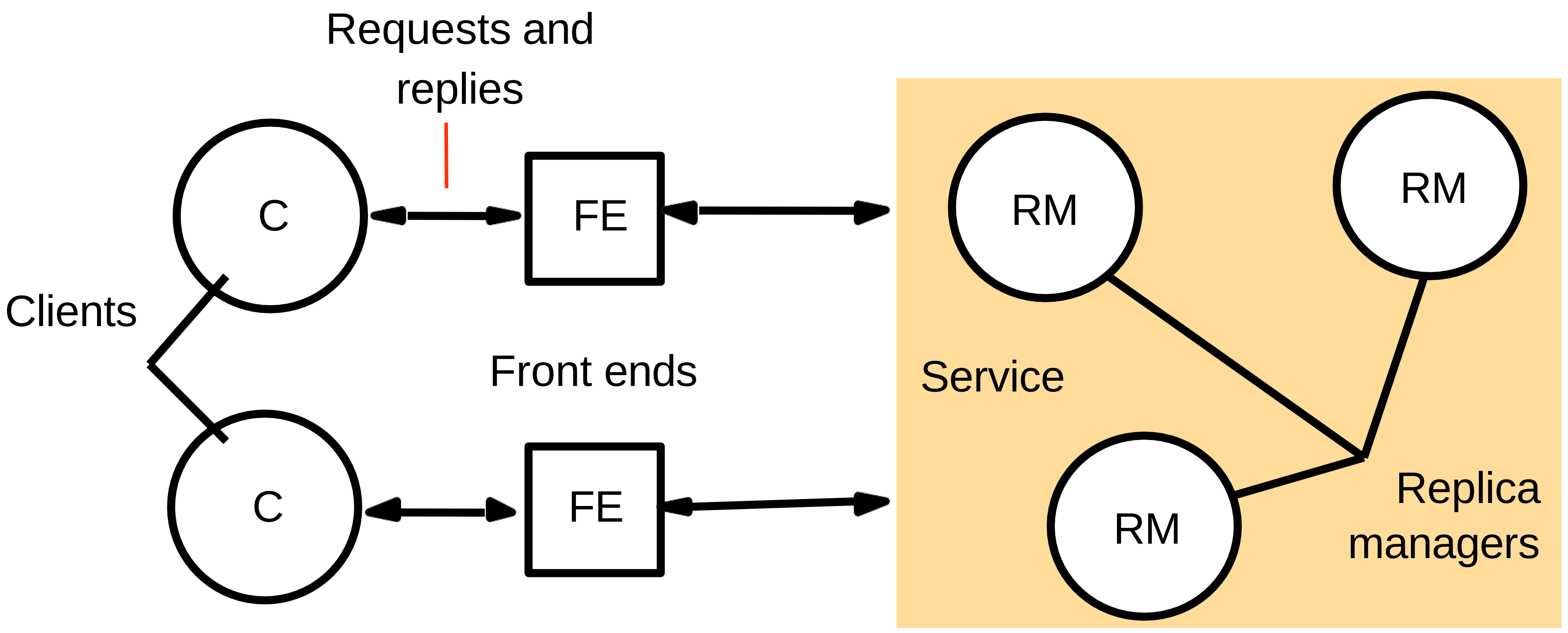
What is replication transparency?

- Replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result
- Consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results. These issues are addressed in Bayou and Coda.

System model

- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally assume no network partitions
- replica managers
  - an RM contains replicas on a computer and access them directly
  - RMs apply operations to replicas recoverably
    - i.e. they do not leave inconsistent results if they crash
  - objects are copied at all RMs unless we state otherwise
  - static systems are based on a fixed set of RMs
  - in a dynamic system: RMs may join or leave (e.g. when they crash)
  - an RM can be a *state machine*, which has the following properties:

A basic architectural model for the management of replicated data



Five phases in performing a request

- issue request
  - the FE either
    - sends the request to a single RM that passes it on to the others
    - or multicasts the request to all of the RMs (in state machine approach)
- coordination
  - the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)
- execution
  - the RMs execute the request (sometimes tentatively)
- agreement
  - RMs *agree* on the effect of the request, .e.g perform 'lazily' or immediately
- response
  - one or more RMs reply to FE. e.g.
    - for high availability give first response to client.
    - to tolerate byzantine faults, take a vote

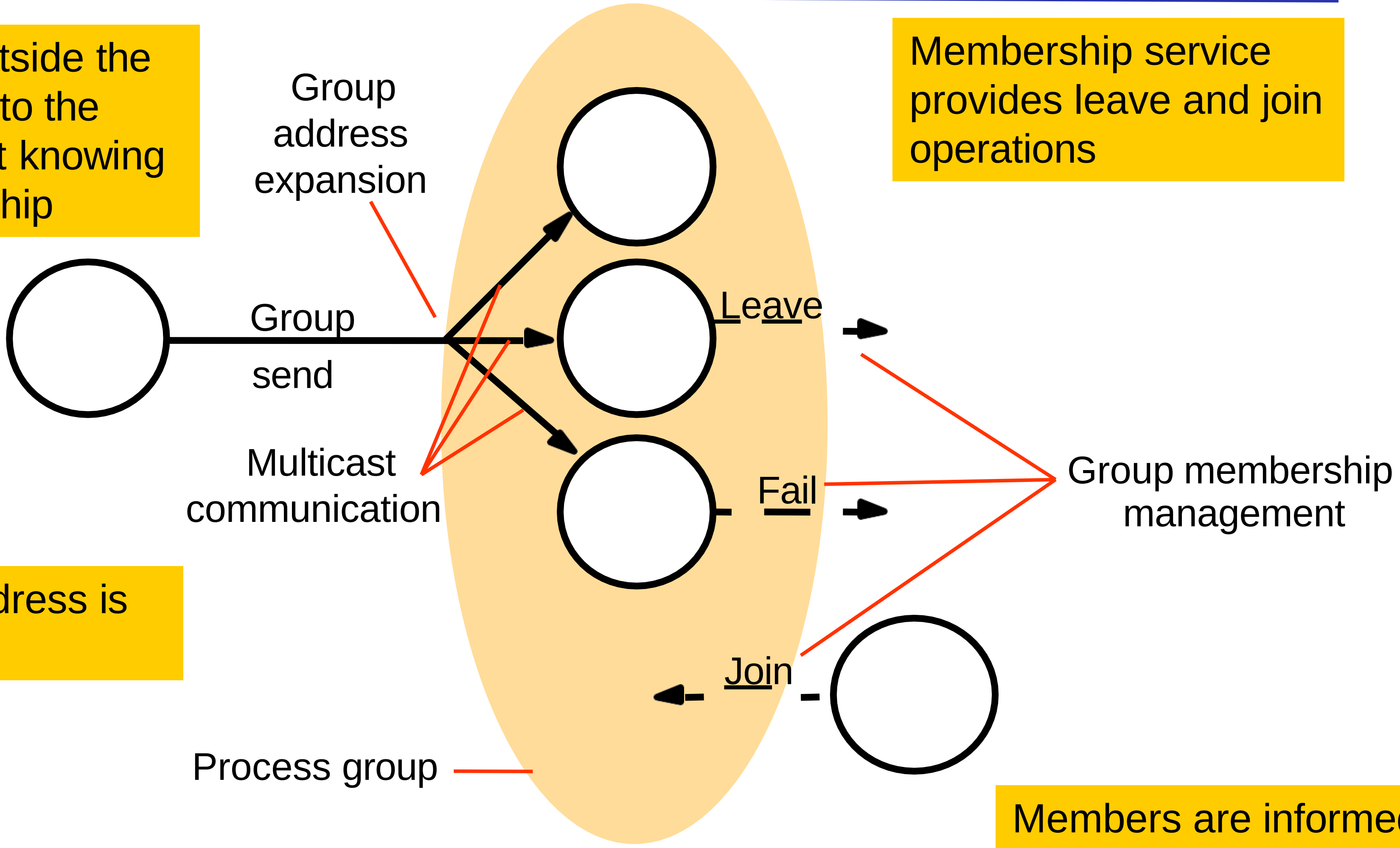
Group communication

- process groups are useful for managing replicated data
  - but replication systems need to be able to add/remove RMs
- group membership service provides:
  - interface for adding/removing members
    - create, destroy process groups, add/remove members. A process can generally belong to several groups.
  - implements a failure detector
    - which monitors members for failures (crashes/communication),
    - and excludes them when unreachable
  - notifies members of changes in membership
  - expands group addresses
    - multicasts addressed to group identifiers,
    - coordinates delivery when membership is changing
- e.g. IP multicast allows members to join/leave and performs address expansion, but not the other features

Services provided for process groups

A process outside the group sends to the group without knowing the membership

Membership service provides leave and join operations



The group address is expanded

Members are informed when processes join/leave

Failure detector notes failures and evicts failed processes from the group

We will leave out the details of view delivery and view synchronous group communication

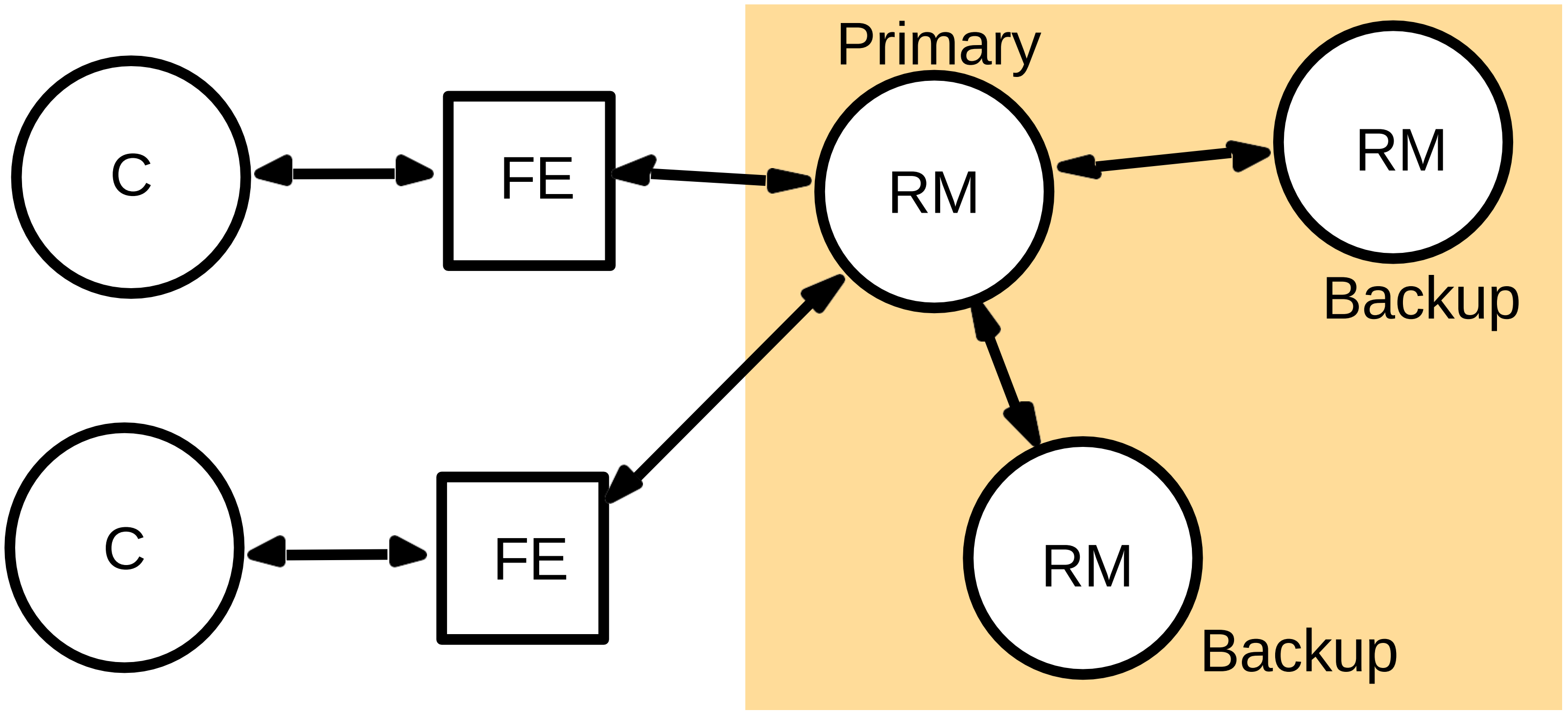
- A full membership service maintains *group views*, which are lists of group members, ordered e.g. as members join group.
- A new group view is generated each time a process joins or leaves the group.
- The idea is that processes can 'deliver views' (like delivering multicast messages).
  - ideally we would like all processes to get the same information in the same order relative to the messages.
- *view synchronous group communication* with reliability.
  - all processes agree on the ordering of messages and membership changes,
  - a joining process can safely get state from another member.
  - or if one crashes, another will know which operations it had already performed
  - This work was done in the ISIS system

Fault-tolerant services

- provision of a service that is correct even if *f* processes fail
  - by replicating data and functionality at RMs
  - assume communication reliable and no partitions
  - RMs are assumed to behave according to specification or to crash
  - intuitively, a service is correct if it responds despite failures and clients can’t tell the difference between replicated data and a single copy
  - but care is needed to ensure that a set of replicas produce the same result as a single one would.

•

The passive (primary-backup) model for fault tolerance



- There is at any time a single primary RM and one or more secondary (backup, slave) RMs
- FEs communicate with the primary which executes the operation and sends copies of the updated data to the result to backups
- if the primary fails, one of the backups is promoted to act as the primary

Passive (primary-backup) replication. Five phases.

- The five phases in performing a client request are as follows:
- 1. Request:
  - a FE issues the request, containing a unique identifier, to the primary RM
- 2. Coordination:
  - the primary performs each request atomically, in the order in which it receives it relative to other requests
  - it checks the unique id; if it has already done the request it re-sends the response.
- 3. Execution:
  - The primary executes the request and stores the response.
- 4. Agreement:
  - If the request is an update the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
- 5. Response:
  - The primary responds to the FE, which hands the response back to the client.

•

Passive (primary-backup) replication (discussion)

- This system implements linearizability, since the primary sequences all the operations on the shared objects
- If the primary fails, the system is linearizable, if a single backup takes over exactly where the primary left off, i.e.:
  - the primary is replaced by a unique backup
  - surviving RMs agree which operations had been performed at take over
- view-synchronous group communication can achieve this
  - when surviving backups receive a view without the primary, they use an agreed function to calculate which is the new primary.
  - The new primary registers with name service
  - view synchrony also allows the processes to agree which operations were performed before the primary failed.
  - E.g. when a FE does not get a response, it retransmits it to the new primary
  - The new primary continues from phase 2 (coordination -uses the unique identifier to discover whether the request has already been performed.

•

Discussion of passive replication

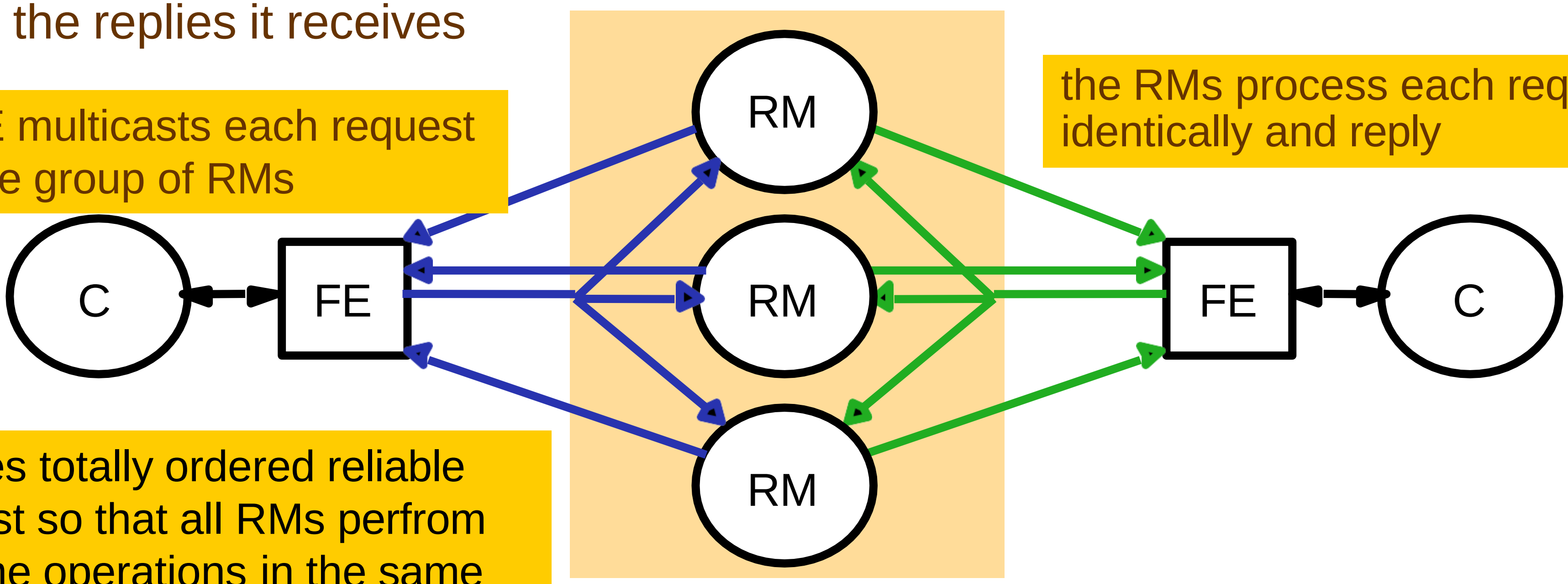
- To survive *f* process crashes, *f*+1 RMs are required
  - it cannot deal with byzantine failures because the client can't get replies from the backup RMs
- To design passive replication that is linearizable
  - View synchronous communication has relatively large overheads
  - Several rounds of messages per multicast
  - After failure of primary, there is latency due to delivery of group view
- variant in which clients can read from backups
  - which reduces the work for the primary
  - get sequential consistency but not linearizability
- Sun NFS uses passive replication with weaker guarantees
  - Weaker than sequential consistency, but adequate to the type of data stored
  - achieves high availability and good performance
  - Master receives updates and propagates them to slaves using 1-1 communication. Clients can uses either master or slave
  - updates are not done via RMs - they are made on the files at the master

•

Active replication for fault tolerance

- the RMs are *state machines* all playing the same role and organised as a group.
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- If an RM crashes it has no effect on performance of the service because the others continue as normal
- It can tolerate byzantine failures because the FE can collect and compare the replies it receives

a FE multicasts each request to the group of RMs



the RMs process each request identically and reply

Requires totally ordered reliable multicast so that all RMs perfrom the same operations in the same order

Active replication - five phases in performing a client request

- Request
  - FE attaches a unique *id* and uses *totally ordered reliable multicast* to send request to RMs. FE can at worst, crash. It does not issue requests in parallel
- Coordination
  - the multicast delivers requests to all the RMs in the same (total) order.
- Execution
  - every RM executes the request. They are state machines and receive requests in the same order, so the effects are identical. The *id* is put in the response
- Agreement
  - no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast.
- Response
  - FEs collect responses from RMs. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Active replication - discussion

- As RMs are state machines we have sequential consistency
  - due to reliable totally ordered multicast, the RMs collectively do the same as a single copy would do
  - it works in a synchronous system
  - in an asynchronous system reliable totally ordered multicast is impossible – but failure detectors can be used to work around this problem. How to do that is beyond the scope of this course.
- this replication scheme is not linearizable
  - because total order is not necessarily the same as real-time order
- To deal with byzantine failures
  - For up to *f* byzantine failures, use 2*f*+1 RMs
  - FE collects f+1 identical responses
- To improve performance,
  - FEs send read-only requests to just one RM