Distributed System Paradigms (30)

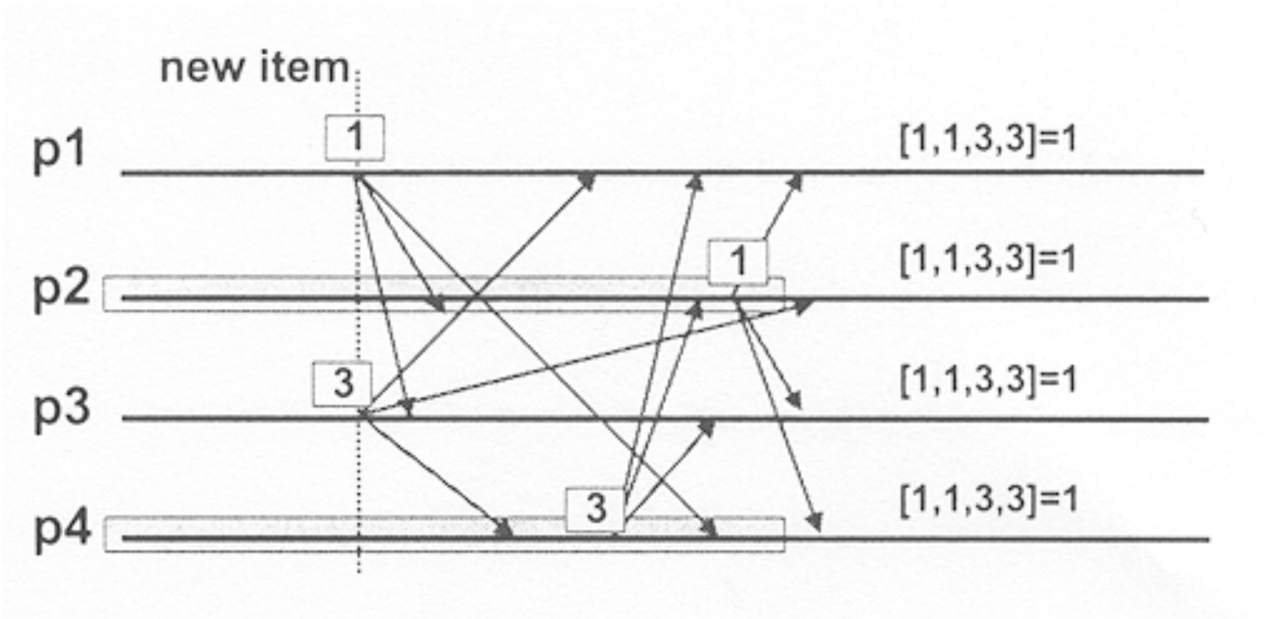
Distributed consensus

- A fundamental problem in distributed systems
- Its goal is to make a set of processes (nodes) agree on the achieved global computational state and on how to proceed

Example: Items, arriving on a belt one at a time, have to be packed by a set of packing machines
Question: Who is going to pick the next item?

Consensus can be reached by collecting a proposal from every machine, ordering the proposals according to the machine number and selecting the first proposal in that set.

A simple consensus protocol



Distributed System Paradigms (31)

a) agreement on membership (membership service)

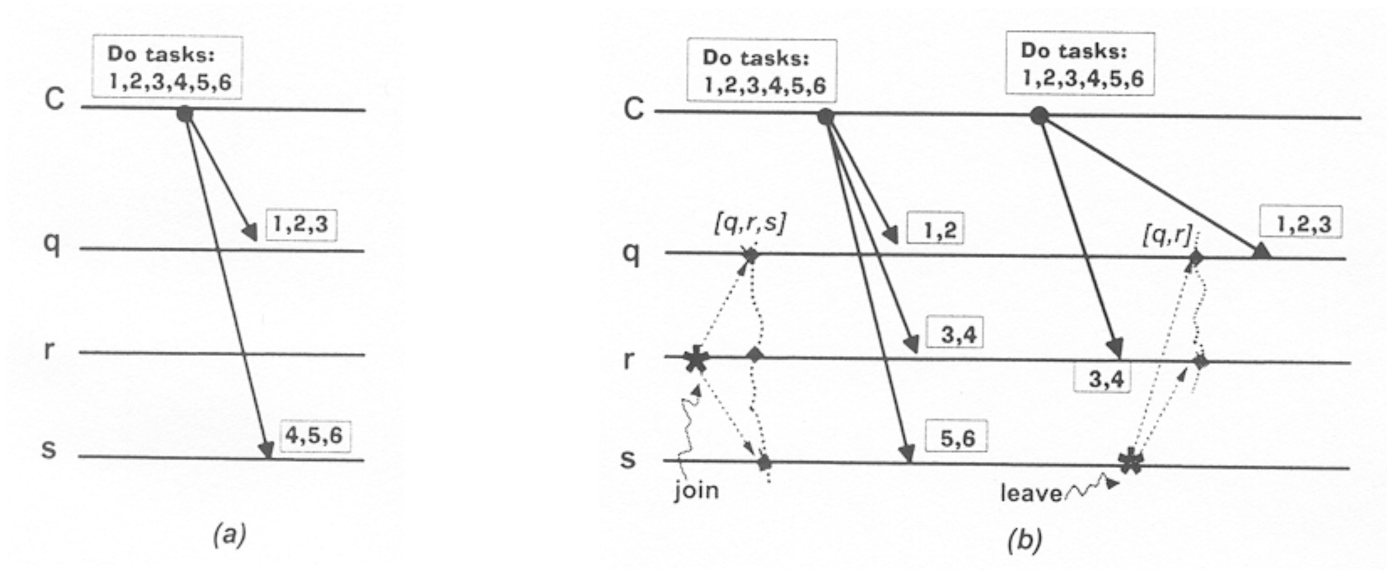
- All members of the group should reach consensus about the current membership (who is in and who not)

view: list of members (active participants) at a given time

First approach: Total order of views + Algorithm similar to the one described before, but, ...

Load balancing example: A team of servers collects requests from clients (e.g .C), and divides the work associated with each request, based on views:= membership of available servers

Ad-Hoc View Change



Problem: Requests were served by different participants based on different views

Solution: Order view changes w.r.t. messages, so that any message is received by all in the same view.

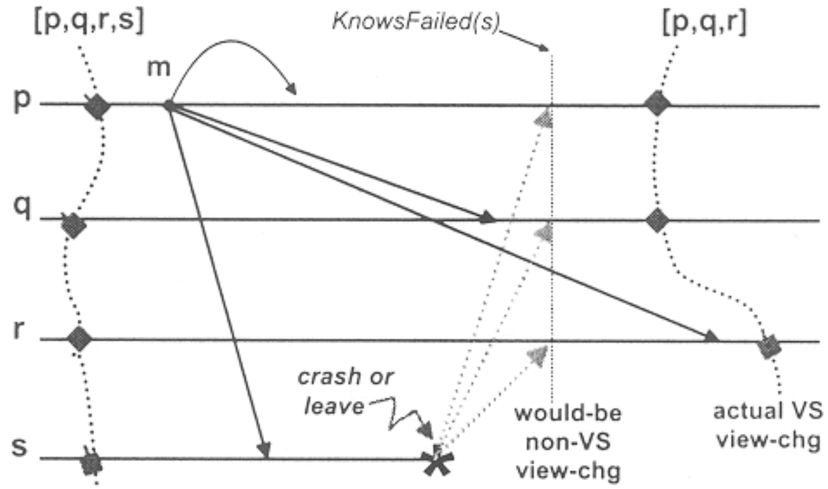
Distributed System Paradigms (32)

b) view synchrony (virtual synchrony)
Definition:
If a message *m* is delivered to a process *p* in view V^i , *m* is also delivered to *q* in view V^i *,* for all $q \varepsilon V$
--> Before installing a new view, consensus must be reached among all processes of the old view not only on the membership of V^{i+1} , but also on the set of messages to be delivered by all participants in the old view.

Algorithm for View-Synchronous View Change

- When a process receives a message, it immediately delivers that message
- When it is time to install a new view, all processes stop sending and delivering messages
- The list of messages already delivered in this view is sent to all participants
- When each process has received all lists, it delivers all messages of the collected lists not yet delivered
- The new view is installed
- The participants resume accepting and delivering new messages

Example:



Distributed System Paradigms (33)

c) atomic broadcast

Scenario:

Response to a request not only depends on the membership, but also on the order of previous requests

--> Beside views, messages need to be totally ordered

--> An atomic broadcast protocol ensures that all messages are received by all members in the same order

Expressed in terms of a consensus problem:
All participants must agree on i) whether they delivered the message, ii) where to put it in the order of all

Simple, but inefficient algorithm (Chandra, Toueg):

- Participants do keep but not deliver all received messages
- When it is time to install a new view, all processes exchange their unordered messages kept
- Ordering of all collected messages is done by all participants using some deterministic rule
- Messages are delivered by that order
- This procedure can be done also periodically

Distributed System Paradigms (34)

c) replica determinism

Scenario:

Maintaining *identica*l copies of the same process or the same data in several nodes

What is the exact semantics behind? In its most stringent interpretation:

execute all replicas in *lock-step,* i.e. synchronized at the instruction level

This can be done only in the small scale, using hardware --> not scalable

More generic definition of *replica determinism*:

Two replicas, departing from the same initial state and subject to the same sequence of inputs should end up in the same final state and produce the same sequence of outputs.

Realization approaches often used in addition to relying on deterministic programs :

- use atomic broadcast to disseminate the inputs or
- elect one replica to decide and care on the order by which messages should be processed

---> all replicas need to be connected in order to exchange messages

Problem:

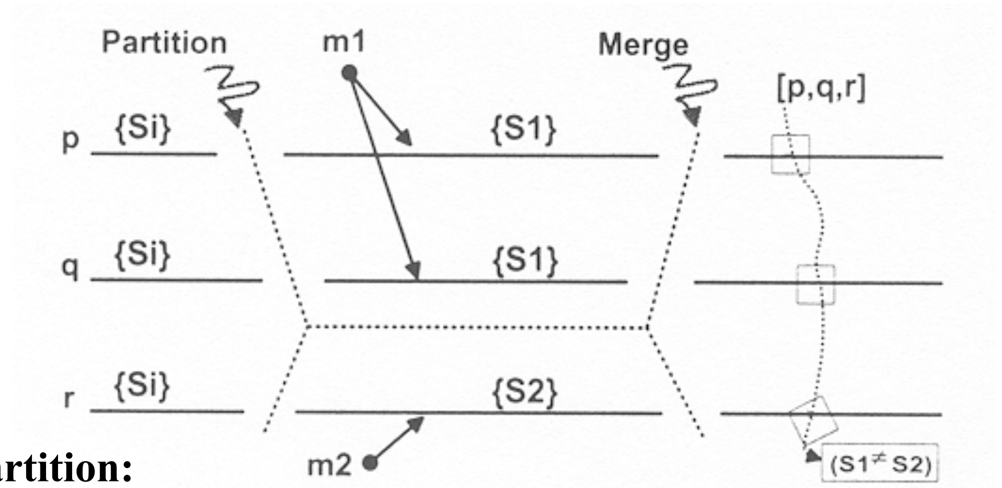
Full connectivity cannot be always ensured --->

The communication network may be split in two or more *partitions,* i.e. nodes in different partitions cannot communicate with each other --->

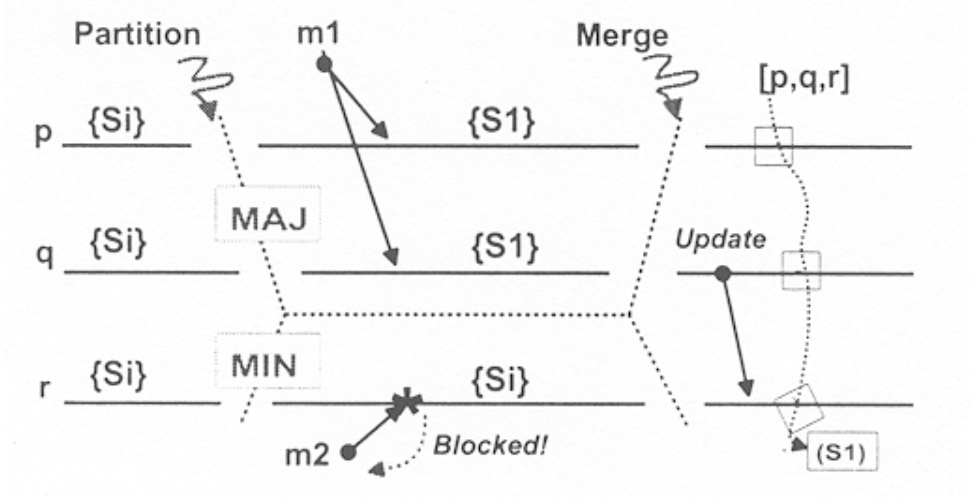
The state of replicas in different partitions is likely to diverge, making it impossible to reconcile later on.

Distributed System Paradigms (35)

State Divergence with Partitioning:



Using a Primary Partition:



- consistency/availability trade-off
- ensuring strong replica consistency is very expensive, since it requires all updates to be totally ordered

Distributed System Paradigms (36)

9. Concurrency

Fundamental property of distributed systems allowing to increase utilization and efficiency.

Let O := $\{o_1, \ldots\ o_m\}$ be a set of operations and X := $\{x_1, \ldots, x_n\}$ be a set of objects where $o_i(x_j)$ with $1 \leq i \leq m$ and $1 \leq j \leq n$ describes the execution of operation o_i on object x_j. Then, a computation C := $[o_i(x_j), \ldots, o_{i'}(x_{j'})]$ is defined by any finite sequence of operations from O on objects from X where |C| denotes the length of the computation, i.e. the number of operations to be executed on X.

Now, let us consider several, say p, computations C_k ($1 \leq k \leq p$) operating on a common object set O. Then, $o_i^k(x_j)$ denotes the execution of operation o_i on x_j within computation C_k.

S := $[s_1, \ldots, s_l]$ is called a *schedule* where each s_i represents an operation $o_i^k(x_j)$ and $l = \sum_k |C_k|$, meaning that S contains all operations of the p computations to be executed in S.

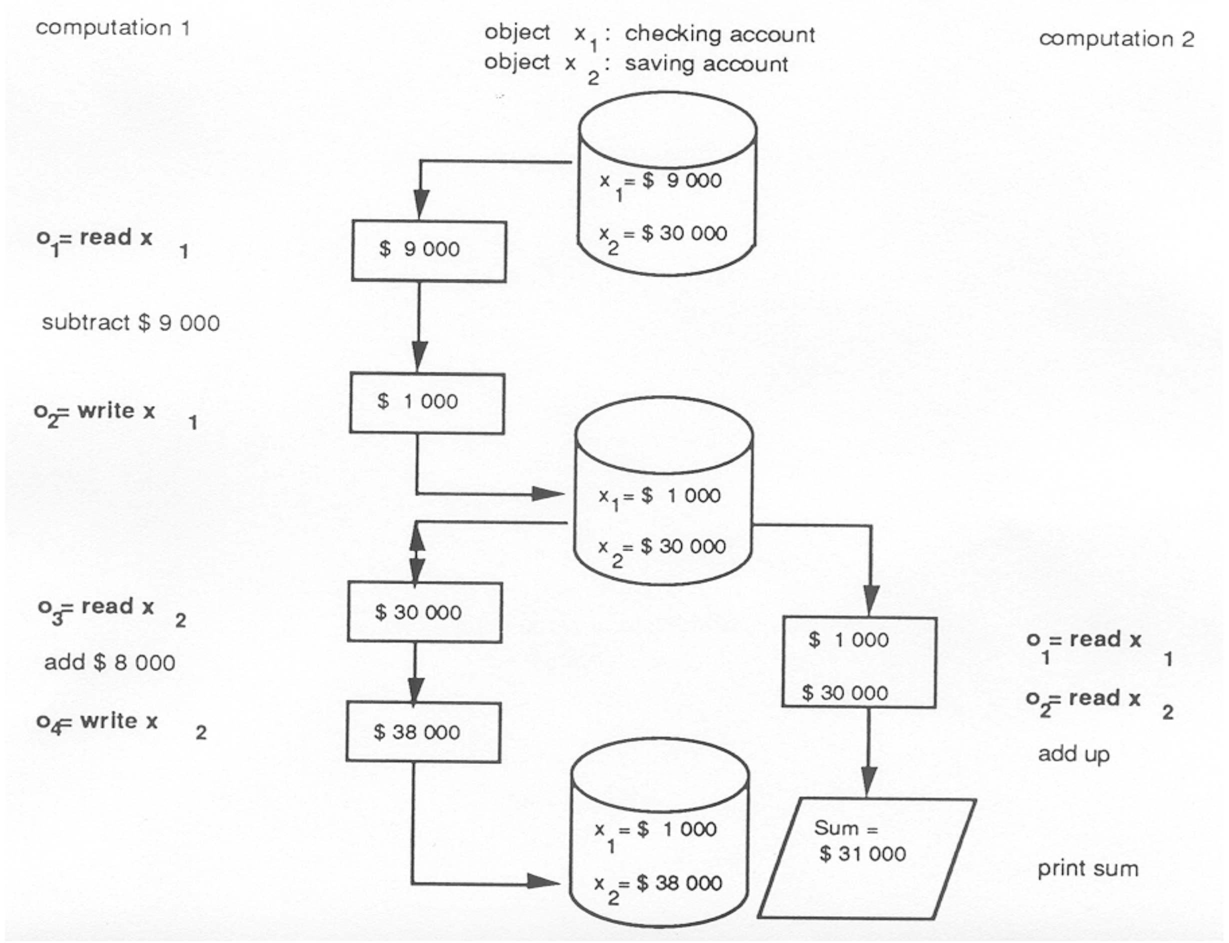
Let *a* and *b* be two events. According to the time they occur, t(a) and t(b), resp., we can define a relation < on a and b as follows: a < b if t(a) < t(b).

A schedule is concurrent if it comprises at least two computations C_i and C_k such that $o_1^i < o_{|Ck|}^k$ and $o_1^k < o_{|Ci|}^i$, meaning that the first operation in both computations is executed before the last operation of any of the two computations has been executed ---> computations cannot be ordered

Two operations $o_i^k(x_j)$ and $o_{i'}^{k'}(x_{j'})$ belonging to two different computations C_k and $C_{k'}$ in a schedule S are in *conflict* if j = j', i.e. if they have to operate on the same object.

Distributed System Paradigms (37)

Incorrect execution of the schedule $[o_1^1(x_1), o_2^1(x_1), o_1^2(x_1), o_2^2(x_2), o_3^1(x_2), o_4^1(x_2)]$



C_2 failed to deliver a correct result because it has become dependent on the effects of another computation C_1 although both of them were intended to be totally independent from each other.