

# Distributed Transaction Management – 2003

Jyrki Nummenmaa  
<http://www.cs.uta.fi/~dtm>  
[jyrki@cs.uta.fi](mailto:jyrki@cs.uta.fi)


# General information

- We will view this from the course web page.

# Motivation

- We will pick up some motivating examples from the world of electronic commerce.
- The following slides will explain discuss those examples and some of their implications.

# Electronic commerce - business-to-customer services

- Searching for product information
- Ordering products 
- Paying for goods and services 
- Providing online customer service
- Delivering services 
- Various other business-to-business services exist, but these are enough for our motivational purposes...

# Internet Commerce

- A person, running a web browser on a desktop computer, electronically purchases a set of goods or services from several vendors at different web sites.
  - This person wants either the complete set of purchases to go through, or none of them.

# Internet Commerce Example: Exhibition Hall

*PC Web browser*

*Exhibitor*

*Exhibition Hall's  
Web site*

*stands*

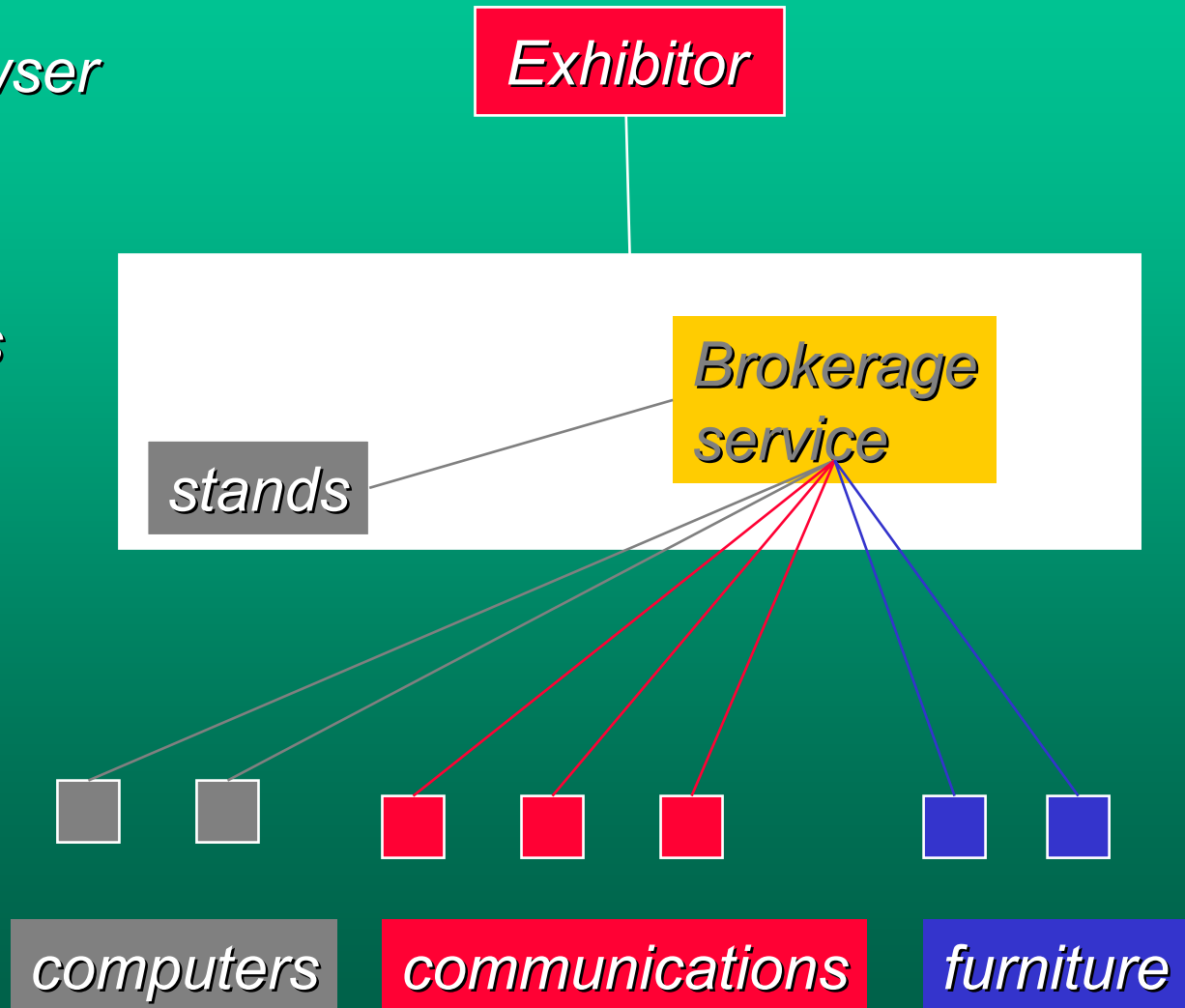
*Brokerage  
service*

*Rental  
Companies'  
Web Sites*

*computers*

*communications*

*furniture*



# Technical Problems with Internet Commerce

- Security
- Failure
- Multiple sites
- Protocol problems
- Server product limitations
- Response time
- Heterogeneous systems

# Failures: single computer

- Hardware failure
- Software crash
- User switched off the PC
- Active attack



# Failure: Additional Problems for Multiple Sites

- Network failure
  - Or is it just congestion?
  - Or has the remote computer crashed?
  - Or is it just running slowly?
- Message loss?
- Denial-of-service attack?
- Typically, these failures are partial.

# Distributed Transaction

- A set of participating processes with local sub-transactions, distributed to a set of sites, perform a set of actions.
- Server Autonomy - any server can unilaterally decide to abort the transaction.
- *All or none* of the updates or related operations should be performed.

# Subtle Difference: Transaction

- Traditional data processing (database) transaction:

set of read and update operations collectively transform the database from one consistent state to another.

- Electronic Commerce transaction:

set of operations collectively provide the user with his/her required package

# Distributed business object transaction example

- Arriving to a football stadium with a car, the customer uses a mobile terminal to buy the ticket and get a parking place.
- Business objects to
  - Charge the money from a bank account
  - Give access to parking
  - Entrance to stadium (writing tickets for collection at a collection point or just giving a digital reservation document).

# Distributed business object transaction example (cont'd)

(Arriving to a football stadium...)

- Why is transactionality needed?
- All-or-nothing situation? Maybe...
- Compensational transactions are difficult - e.g. once access is given to car park, that is difficult to roll back.

# Transaction properties - Atomicity

## ■ Atomicity

- ensures that if several different operation occur within a single transaction, it can never be the case that some operations complete if others cannot complete.

# Transaction properties - Isolation

## ■ Isolation

- ensures that concurrently-executing transactions do not interfere with each other, in the sense that each transaction sees a consistent state of the data – often a database.

# Transaction properties - Durability

## ■ Durability

- ensures that unless an update transaction is rolled back, then its changes will affect the state of the data as seen by subsequently-executing transactions.



# Typical system architecture

- Front-tier clients
  - e.g. web browsers.
- Back-tier servers
  - such as database systems, message queue managers, device drivers, ...
- Middle-tier business objects
  - each typically serving one client using (and locking) a number of shared resources from a number of back-tier servers.

# Traditional system architecture

- Computers are hard-wired to each other.
- A synchronous system, where a message timeout means that a computer has crashed.
- A transparent centralised database management system, which the user can see as a single database.
- An application program can use the database as a single database, thus benefitting from transparency.

# Main transactional services

- Distributed locking is needed, if replicated data is needed for exclusive (write) access.
- Distribute Commit is needed to control the fate of the transaction in a controlled manner.
- Barrier synchronisation can be used to guarantee a consistent view of the world.

# Implementing transactional services

- As we noticed, a traditional distributed database system gives a transparent view to the system. It also takes care of concurrency.
- In a modern distributed system, the application programmer needs to implement a large part of transactional services.
- These services are complicated, and their implementation is far from being easy.

# Transaction Model

- We will quite often write "txn" instead of "transaction".

# Txn model - sites

- We assume that there is a set of *sites*  $S_1, \dots, S_n$ .
- All of these sites have a *resource manager* controlling the usage of the local resources.
- We may know all of these sites before the txn starts (like a site for each bookstore sub-branch) or then we may not (like when previously unknown sites from the Internet may join in).

# Txn model - participants

- The txn needs to access resources on some of these sites (without loss of generality, all of them).
- For this, there is a local transaction on each site (transaction  $T_i$  on site  $S_i$ ).
- The local transaction executes the operations required on the local site.
- To use the local resources, the local transaction talks with the local resource manager.

# Distributed Transactions

- In a distributed transaction there is a set of subtransactions  $T_1, \dots, T_k$ , which are executed on sites  $S_1, \dots, S_k$ .
- Each subtransaction manages local resources. The particular problems of managing distributed transactions vs. centralised (local) transactions come from two sources:
  - Data may be replicated to several sites. Lock management of the replicated data is a particular problem.
  - Regardless of whether the data is replicated or not, there is a need to control the fate of the distributed transaction using a distributed commit protocol.



# Failure model - sites

- Sites may fail by crashing, that is, they fail completely.
- Sometimes it is assumed that crashed sites may recover. In this case usually the resource managers and the participants have recorded their actions in persistent memory.
- Sometimes it is assumed that the crashed sites do not recover.
- Usual assumption: if a site functions, it functions correctly (instead of e.g. sending erroneous messages).

# Failure model - messages

- Messages may be delayed.
- Message transfer delays are unpredictable (asynchronous message-passing)
- Messages are transferred eventually.
- Messages between sites are not spontaneously generated.
- Messages do not change in transmission.

# Failure model - messages

- All messages arriving at a site  $S_j$  from a site  $S_i$  are processed in the order they were sent.
- It may be that the network is partitioned, that is, some sites can not exchange messages. This may continue for an unpredictable time.
  - This assumption is by default avoided, since it is a really hard one.
  - We will state it explicitly if we want it to hold.
  - However, in real world this happens.

# Asynchronous communication

- In a synchronous system, we assume that the relative speeds of processes and communication delays are bounded.
- In an asynchronous system we do not make such an assumption. This means that not receiving an expected message does not mean a failure.
- Generally, we assume here that we are dealing with an asynchronous system.

# Failure detection

- Failure is hard to detect.
- Typically, failure is assumed, if an expected message does not arrive within the usual time period.
  - Timeouts are used.
  - Delay may be caused by network congestion.
  - Or is the remote computer running slowly?
  - Mobile hosts make failure detection even harder, because it is expected behaviour if they stay unconnected for an unexpected time.

# Distributed Locking

# Mutual Exclusion (Locking)

- The problem of managing access to a single, indivisible resource (e.g. a data item) that can only support one user (or transaction, or process, or thread, or whatever) at a time.

# Desired properties for solutions

- **Safety:** Mutual exclusion is never violated. (Only one transaction gets the lock).
  - This property can not be compromised.
- **Liveness:** Each request will be granted (eventually).
  - This property should not be compromised.
- **Ordering (or Fairness):** Access to the resource should happen in the order of requests.
  - This property needs to be discussed later.



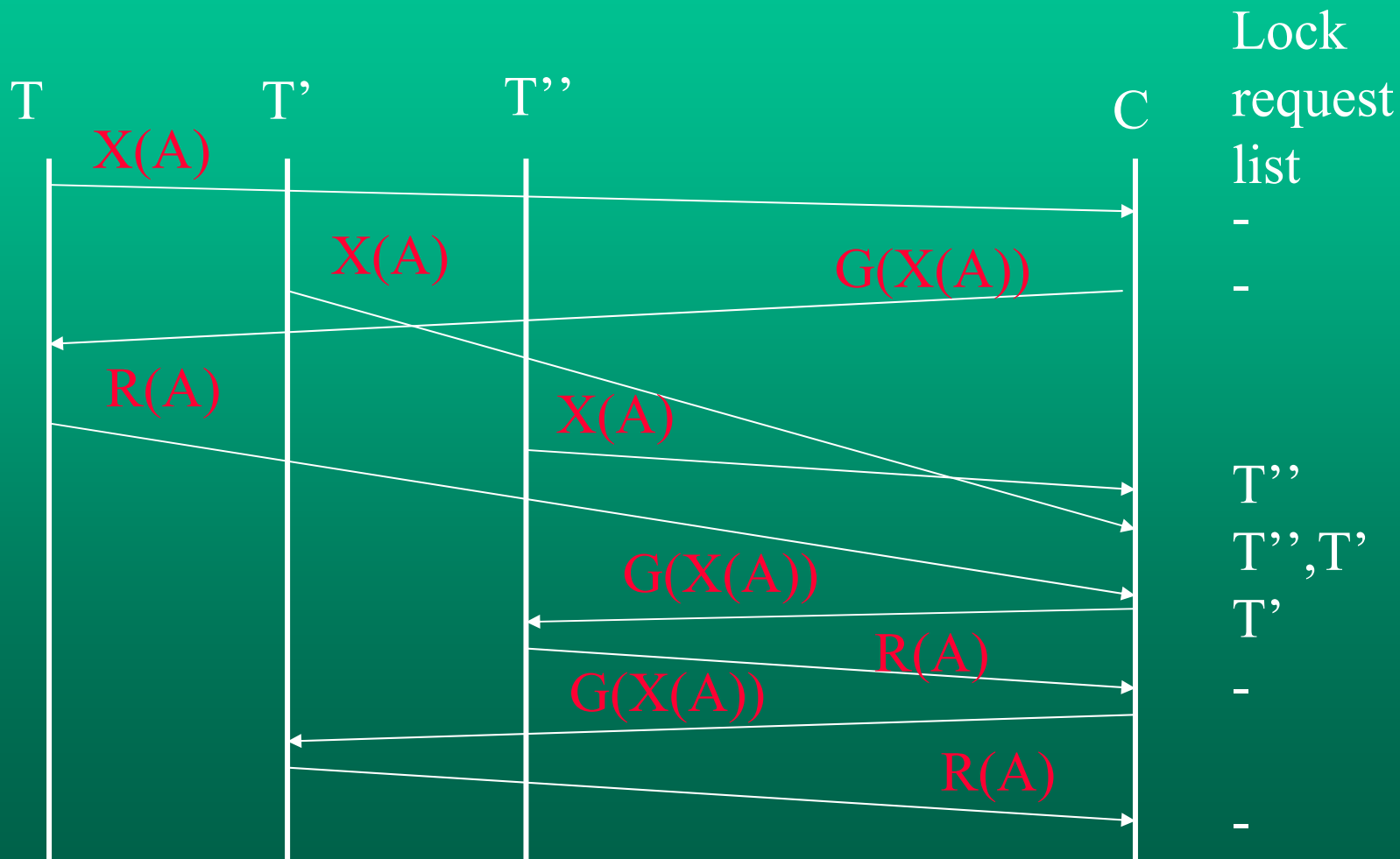
# Coordinator-based solutions / 1

- There is a coordinator to control access.
- Coordinator is a process on one of the sites. (It is none of the transactions.)
- When a transaction needs access, that transaction sends a request to coordinator. Let us write  $X(A) =$  exclusively lock  $A$ .
- The coordinator queues requests.

# Coordinator-based solutions / 2

- When the resource is available, the coordinator sends a grant message to the transaction  $T$  first in the queue.  $G(X(A)) = \text{Grant } X(A)$
- When  $T$  sees the grant message, it may use the resource.
- When  $T$  does need the resource anymore, it sends a release message to the coordinator.  $R(A) = \text{release } A.$

# An example



# Coordinator-based solutions / properties

- These coordinator-based solutions obviously have the safety and the liveness properties, if the coordinator is correctly implemented.
- We can argue that they are also fair, since requests are queued. However, the behaviour of the example is does not seem fair – and the lack of global time is a problem. More on that later.
- Since lock management is centralised, different lock types need no special attention.

# Coordinator-based solutions / weaknesses

- The system does not tolerate a crashing coordinator.
- The coordinator may become a bottleneck for performance.
- Suppose data is replicated, there is a local copy, and the coordinator is not on the local site. Then we always need to communicate over the network, which reduces the benefits of having a local copy.

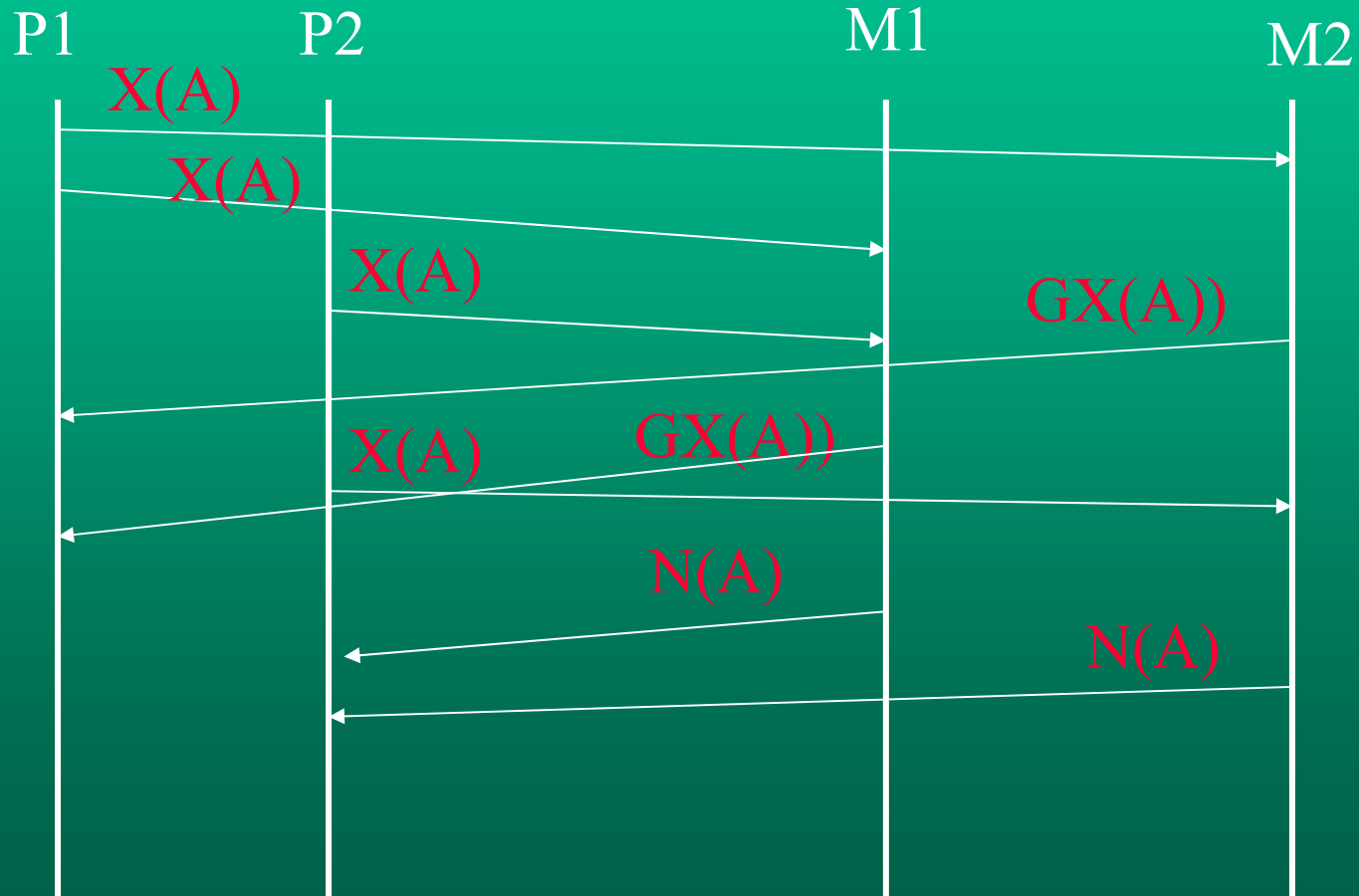
# Primary copy for replicated data

- If data is not replicated, then to use a data item, you must contact the site containing the item.
- If the resource manager at that site acts as the coordinator giving locks for its items, communication is simple.
- If the data is replicated, then we can have a "primary copy", which is accessed for locking. The resource manager at the site of the primary copy is the coordinator.

# Voting-based algorithms

- We assume here that we know a set of resource managers (say,  $M_1, \dots, M_n$ ), which hold a replicated data item.
- When transaction  $T$  needs access to the shared resource, it will send a message to  $M_1, \dots, M_n$  asking for the permission.
- Each  $M_1, \dots, M_n$  will answer Yes or No.
- $T$  waits until the replies are in.
- If there are enough Yes votes,  $T$  will get the lock.

# A voting example





# Which resource managers to consult?

- In principle, it could be enough to ask only a subset (like a majority) of processes for a permission.
- This subset could be statically defined, given a data item.
- However, as it might be advantageous to contact near-by resource managers, the set may well depend on who is asking.

# How many processes to ask?

- Suppose we have  $n$  processes, and we consult  $k$  processes for an exclusive lock (write-lock) and  $m$  processes for a shared lock (read-lock).
- To avoid two simultaneous exclusive locks, we must have  $k > n/2$ .
- To avoid simultaneously having an exclusive and a shared lock, must have  $k + m > n$ .
- If read-operations dominate, then we may choose  $m=1$  and  $k= n$ .

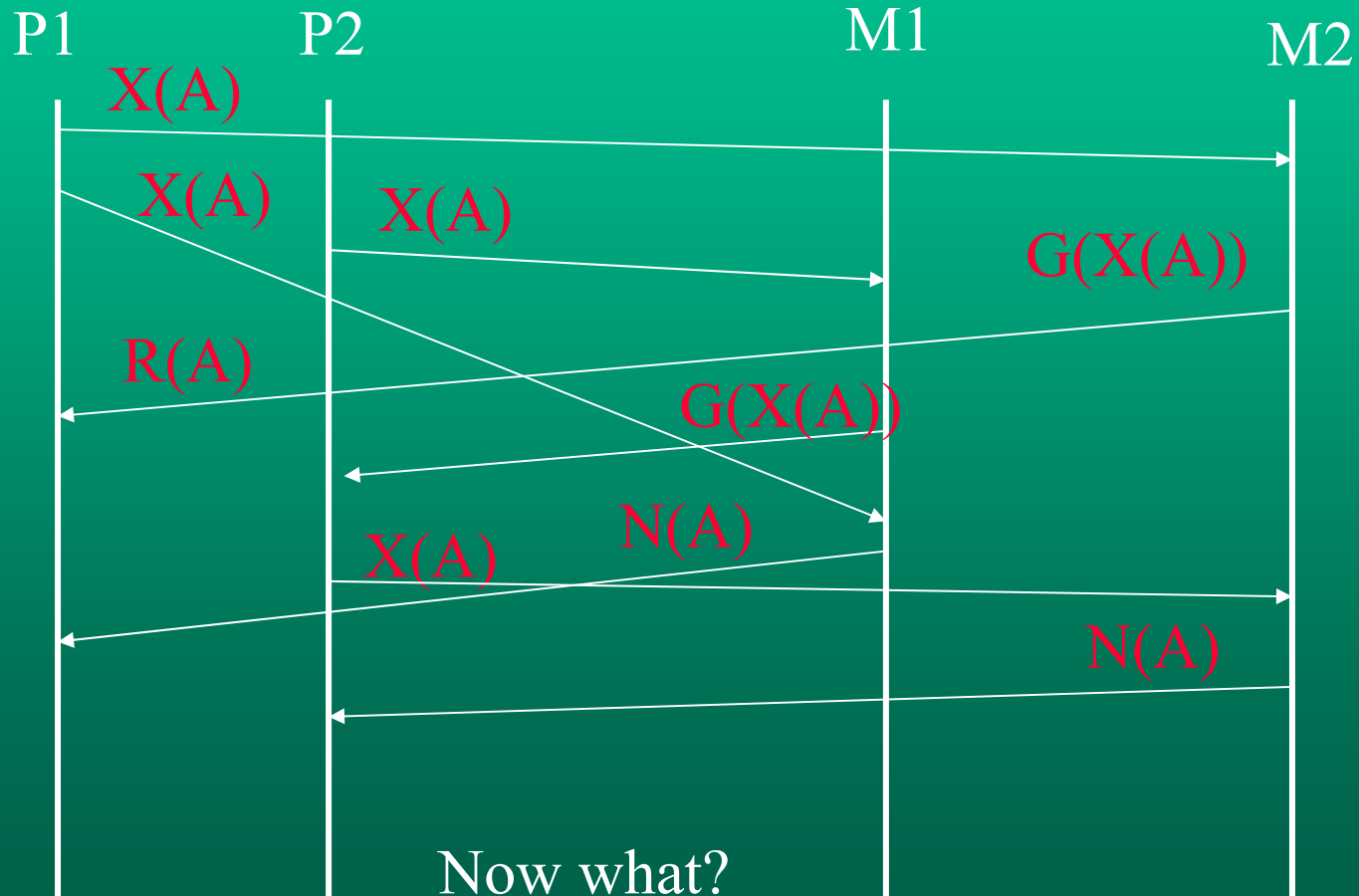
# Example

- Suppose we operate an airline with offices in Tampere, Stockholm and London.
- It seems reasonable to replicate timetables and use  $m=1$ ,  $k=n$ , since that information does not change that often.
- For ticket booking, primary copy seems more appropriate. By statistical analysis we may get to know, where people (geographically) people book which flights, to choose the placement of each primary copy.

# Who needs to give permission?

- If we need a permission from all resource managers, then we do not tolerate site failures (all the downsides of having a coordinator plus all the extra effort of contacting all the resource managers).
- Generally, a majority is enough.
- There are also ways other than simple majority or unanimous vote, but one has to be careful to preserve the mutual exclusion.

# A problematic voting



# Analysis for voting

- Safety – apparently ok.
- Liveness – this far there is nothing to stop the previous slide situation repeating over and over.
- Fairness – similarly, nothing appears to guarantee fairness.
- -> Some improvements are necessary.

# How to re-start after not getting a lock?

- Apparently, something needs to be done to avoid repeating the situation where no-one gets the lock.
- If we re-start requesting locks, we can tell younger transactions to wait longer before re-starting.
- However, new transactions may always step in to stop the oldest transaction from getting the lock -> this is not the solution.

# Queueing the requests?

- Instead of just answering the lock requests, the resource managers can also maintain a lock request list.
- Put the oldest transaction T first in the list and answer no-one Yes before T has either got and released the lock or canceled the lock request.
- Now, eventually T should get the lock and we are able to get liveness.



# Using timestamps – basic idea:

- Give each transaction a timestamp
- Execute the transactions' reads and writes.
- If there is a conflict (impossible event compared to serial execution based on timestamps), roll back the younger transaction, which is then free to restart.

# Using timestamps – examples

- T1 starts
  - T2 starts
  - T2 writes X
  - T1 is to read X – conflict, as T2 should have not have written this value!
  - Roll back T2, if it still exists. Otherwise roll back T1.
  - Multiversioning solves this.
- T1 starts
  - T2 starts
  - T2 reads X
  - T1 is to write X – conflict, as T2 should have read this new value!
  - Roll back T2, if it still exists. Otherwise roll back T1.
  - Multiversioning does not solve this!

# Distributed timestamps?

- Can be used similarly as centralised timestamps with the exception that we must be able to order timestamps globally.
- Same trick: clock time + site id: if local clock times are equal, use site id.

# Ordering things

- Fairness in both the coordinator-based and voting-based protocol as well as timestamping seems to depend on ordering the transactions by their age.
- However, we would need synchronised clocks to do this. Perfect synchronisation or clocks is not possible. Good synchronisation can sometimes be assumed.
- Next time we will study logical ordering events and possibly deadlock management.

# Programming

- It is important to get started with the programming part.
- Study the basis for this (unless you already know).
- For our purposes, network access with sockets seems appropriate.
- For Java, see the following tutorial:  
<http://java.sun.com/docs/books/tutorial/networking/index.html>