Distributed Systems

Lesson 8
Fault Tolerance and Lab session

University of New York in Tirana
Master of Science in Computer Science
Prof. Dr. Marenglen Biba

Lesson 8

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

Lesson 8

- Part I – Fault tolerance
- Part II – Lab session on distributed object-based systems

Fault tolerance

- Concepts
- Process Resilience
- Reliable Group Communication
- Recovery

Dependable Systems

Basics

- A component provides services to clients.
- To provide services, the component may require the services from other components => a component may depend on some other component.
- Requirements for Dependability
  - **Availability:** Readiness for usage
  - **Reliability:** Continuity of service delivery
  - **Safety:** Very low probability of catastrophes
  - **Maintainability:** How easy can a failed system be repaired

Definitions

- A system is said to **fail** when it cannot meet its promises.
- An **error** is a part of a system's state that may lead to a failure.
- The cause of an error is called a **fault**. Clearly, finding out what caused an error is important.

Fault Tolerance

- Building dependable systems closely relates to controlling faults.
  - A distinction can be made between preventing, removing, and forecasting faults.

- For our purposes, the most important issue is fault tolerance, meaning that a system can provide its services even in the presence of faults.
  - In other words, the system can tolerate faults and continue to operate normally.

Types of Faults

- Faults:
  - Transient: occur once and then disappear
  - Intermittent: appears, disappears, and so on
  - Permanent: exists until the faulty component is replaced

Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 8-1. Different types of failures.

Failure Masking by Redundancy

- If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes.
- The key technique for masking faults is to use redundancy.
- Three kinds are possible:
  - Information redundancy: Example => Hamming code
  - Time redundancy: perform operation again
  - Physical redundancy: extra equipment of processes

Fault tolerance

- Concepts
- Process Resilience
- Reliable Group Communication
- Recovery

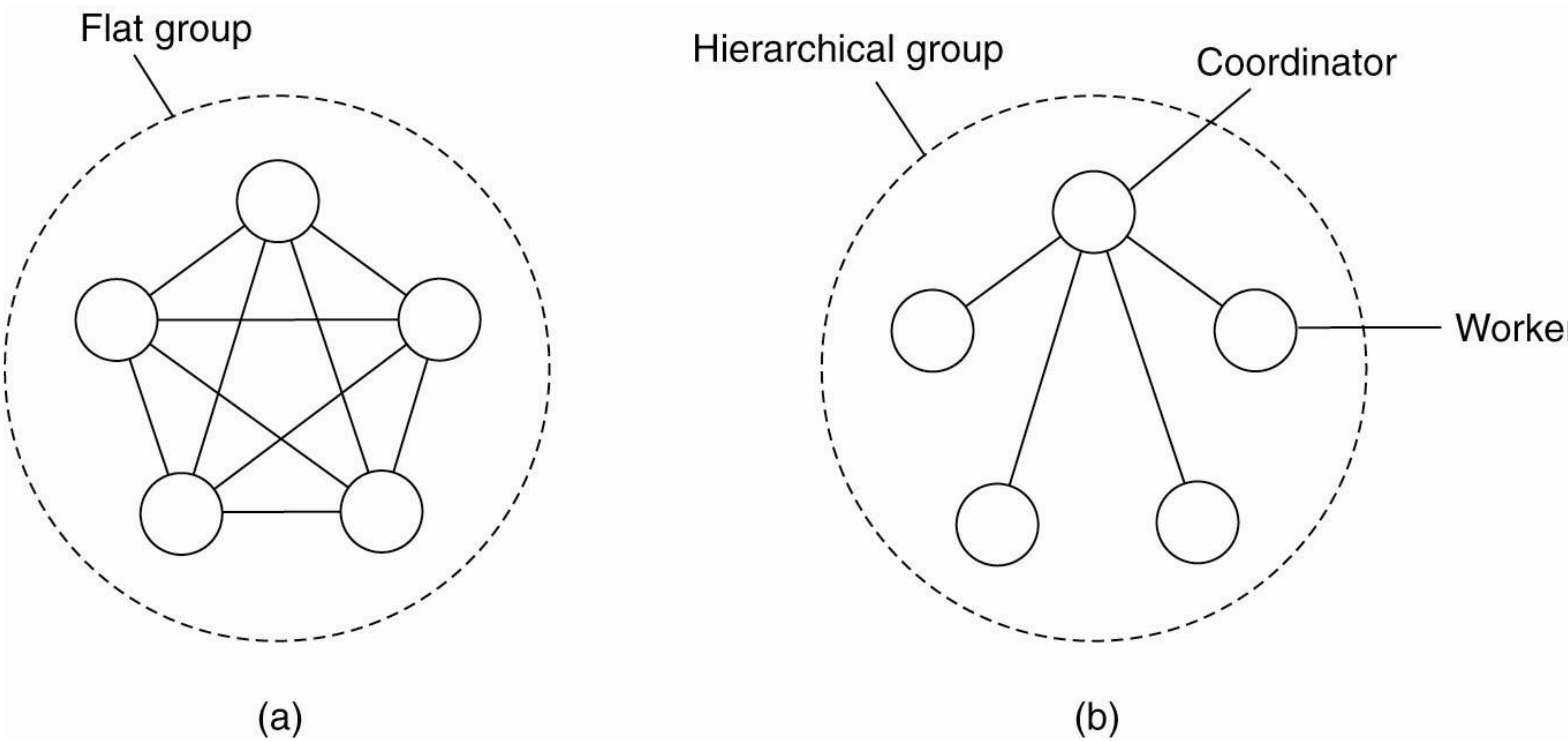
Process Resilience

Basic issue

Protect yourself against faulty processes by replicating processes into a group.

- Flat groups: Good for fault tolerance as information exchange immediately occurs with all group members
  - However, may impose more overhead as control is completely distributed (hard to implement).
- Hierarchical groups: All communication through a single coordinator => not really fault tolerant and scalable, but relatively easy to implement.

Flat Groups versus Hierarchical Groups



- Figure 8-3. (a) Communication in a flat group.
  (b) Communication in a simple hierarchical group.

Groups and failure masking

K-fault tolerant group

- When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance).

How large does a k-fault tolerant group need to be?

- Assume crash semantics => a total of *k+1* members are needed to survive *k* member failures.
- Assume arbitrary failure semantics, and group output defined by voting => a total of *2k+1* members are needed to survive *k* member failures.

Agreement in Faulty Systems

Possible cases:

1. Synchronous versus asynchronous systems.
2. Communication delay is bounded or not.
3. Message delivery is ordered or not.
4. Message transmission is done through unicasting or multicasting.

Distributed Agreement in Faulty Systems

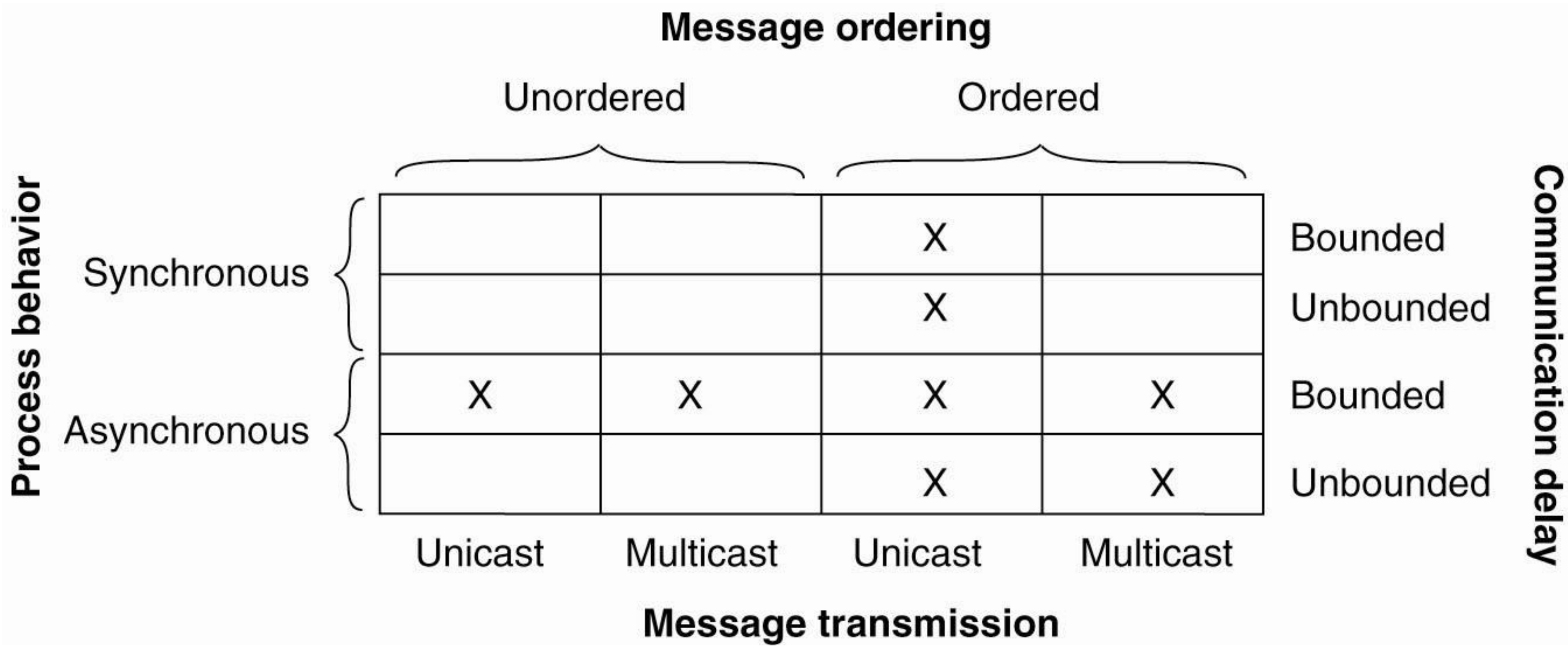
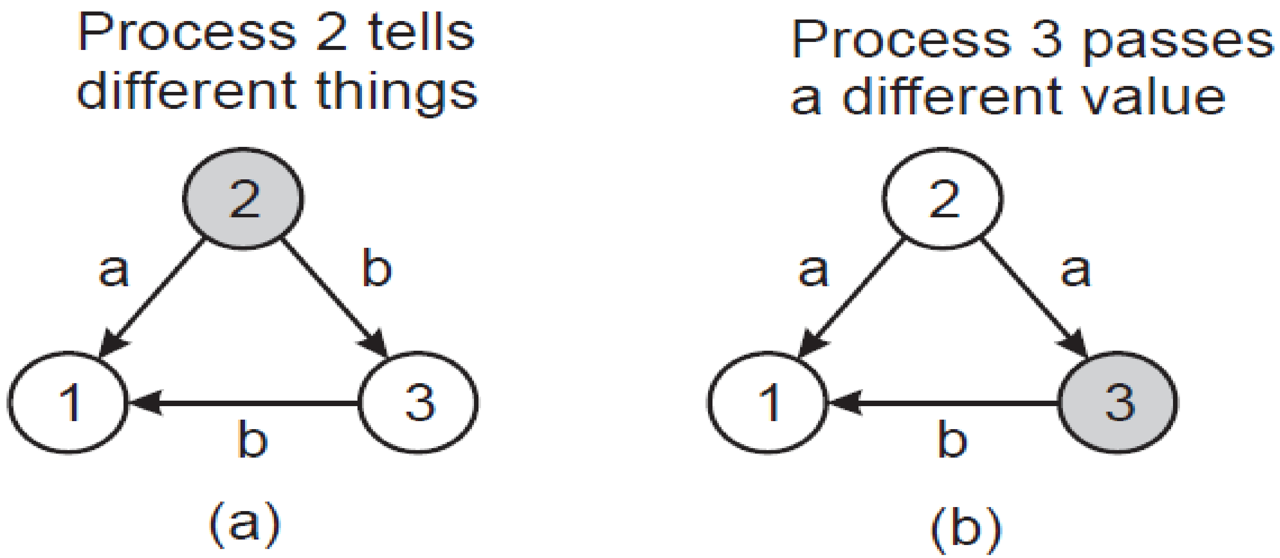


Figure 8-4. Circumstances under which distributed agreement can be reached.

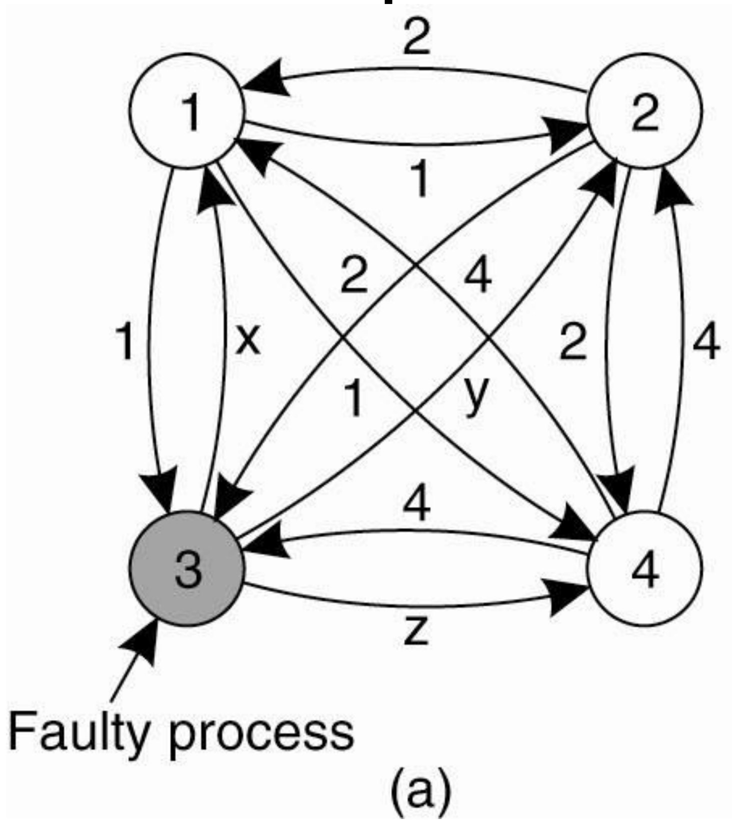
Byzantine Agreement Problem (Lamport, 1982)

Scenario

- Group members are not identical, i.e., we have a distributed computation => Nonfaulty group members should reach agreement on the same value.



Bizantine Agreement in Faulty Systems: Step 1



- Figure 8-5. The Byzantine agreement problem for three nonfaulty and one faulty process.
- (a) Step 1: Each process sends their value to the others (process 1 sends 1)

Bizantine Agreement in Faulty Systems: Step 2 and 3

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

(b) Step 2: the vectors that each process assembles based on (a).

(c) Step 3: consists of every process passing its vector from Fig. 8-5(b) to every other process. In this way, every process gets three vectors, one from every other process. Here, too, process 3 lies, inventing 12 new values, *a* through l.

- Processes 1, 2 and 4 come to agreement on the values for V_1,V_2,V_4.

Agreement in Faulty Systems

- Finally, in step 4, each process examines the ith element of each of the newly received vectors.
- If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked *UNKNOWN.*
- From Fig. (c) we see that 1, 2, and 4 all come to agreement on the values for v1, *v* 2, and *v* 4, which is the correct result.
- What these processes conclude regarding *v*3 cannot be decided, but is also irrelevant.
- The goal of Byzantine agreement is that consensus is reached on the value for the nonfaulty processes only.

Failure of Agreement

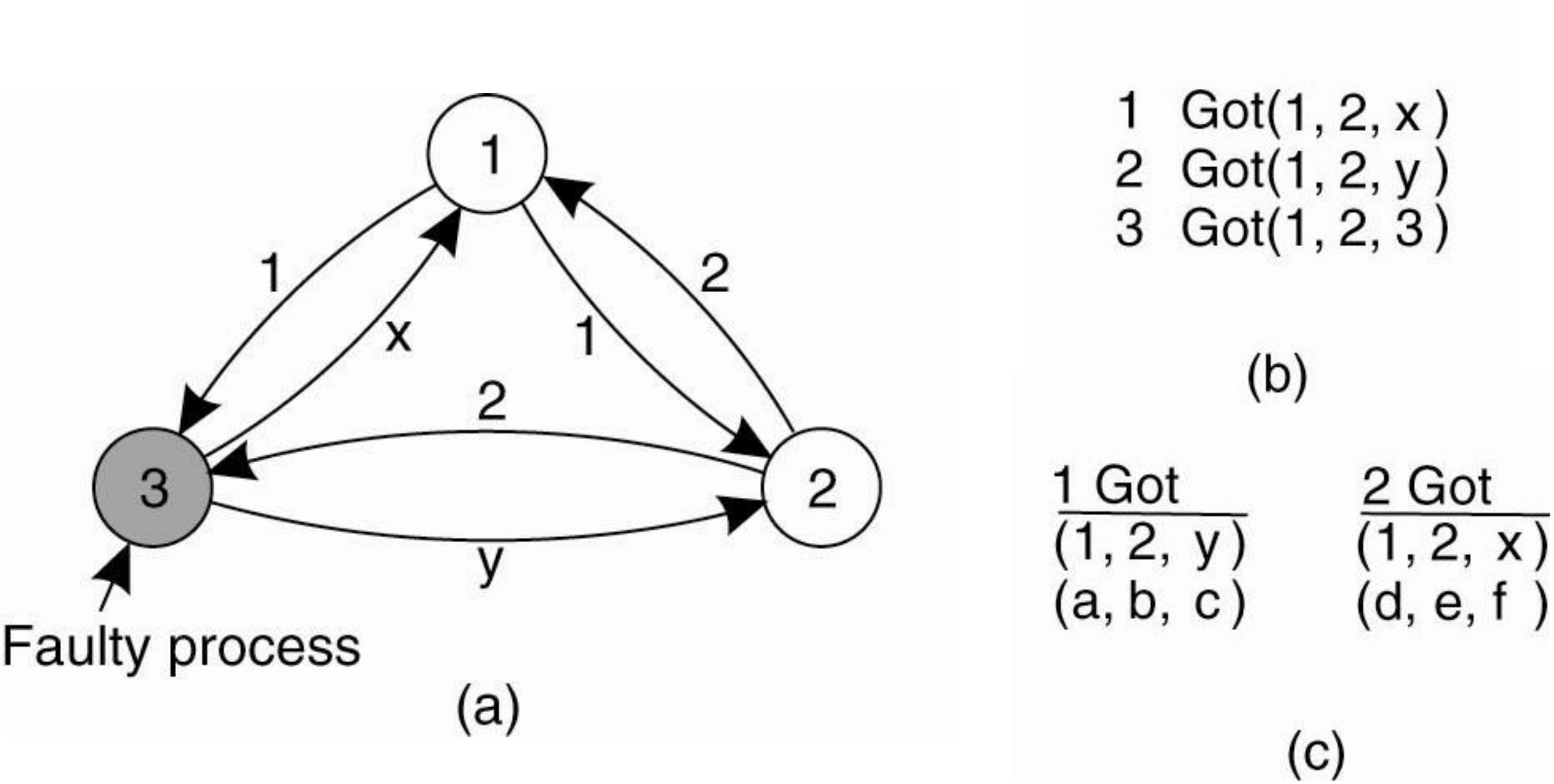


Figure 8-6. Failure of producing agreement.

Neither of the correctly behaving processes sees a majority for element 1, element 2, or element 3, so all of them are marked *UNKNOWN.*

Byzantine Agreement Requirements

- In their paper, Lamport et al. (1982) proved that in a system with *k* faulty processes, agreement can be achieved only if $2k + 1$ correctly functioning processes are present, for a total of $3k + 1$.

- Put in slightly different terms, agreement is possible only if *more* than two-thirds of the processes are working properly.

Fault tolerance

- Concepts
- Process Resilience
- Reliable Group Communication
- Recovery

Reliable multicasting

Basic model

- We have a multicast channel *c* with two (possibly overlapping) groups:
  - The sender group *SND(c)* of processes that submit messages to channel *c*
  - The receiver group *RCV(c)* of processes that can receive messages from channel *c*

Reliable multicasting

- Simple reliability: If process *P* ϵ *RCV(c)* at the time message *m* was submitted to *c*, and *P* does not leave *RCV(c), m* should be delivered to *P.*

- Atomic multicast: How can we ensure that a message *m* submitted to channel *c* is delivered to process *P* ϵ *RCV(c)* only if *m* is delivered to all members of *RCV(c)?*

Basic Reliable-Multicasting Schemes

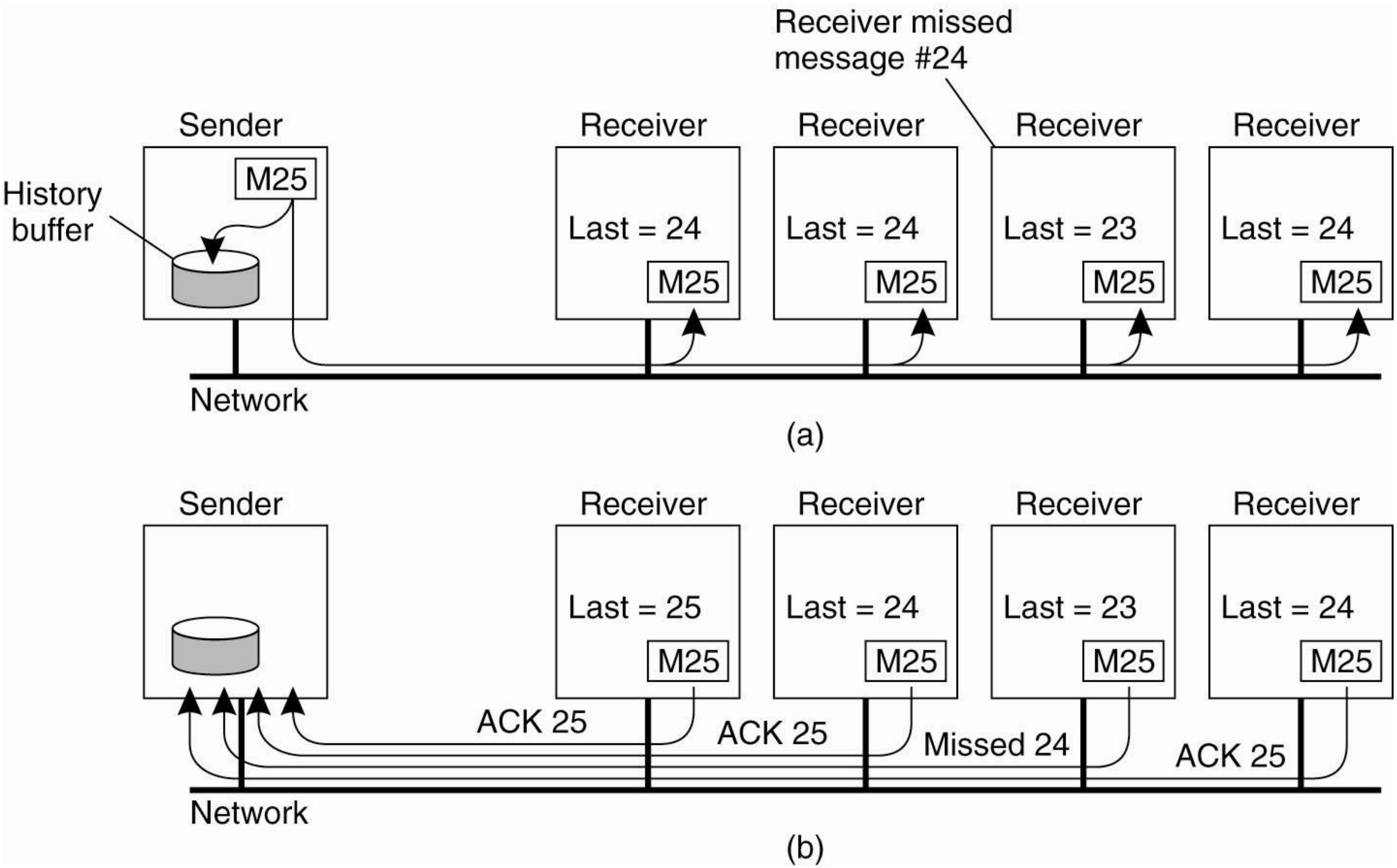


Figure 8-9.Receivers are known and assumed not to fail.
(a) Message transmission with sequence numbering. (b) Reporting feedback, missed message #24.

Reliable-Multicasting: Feedback suppression

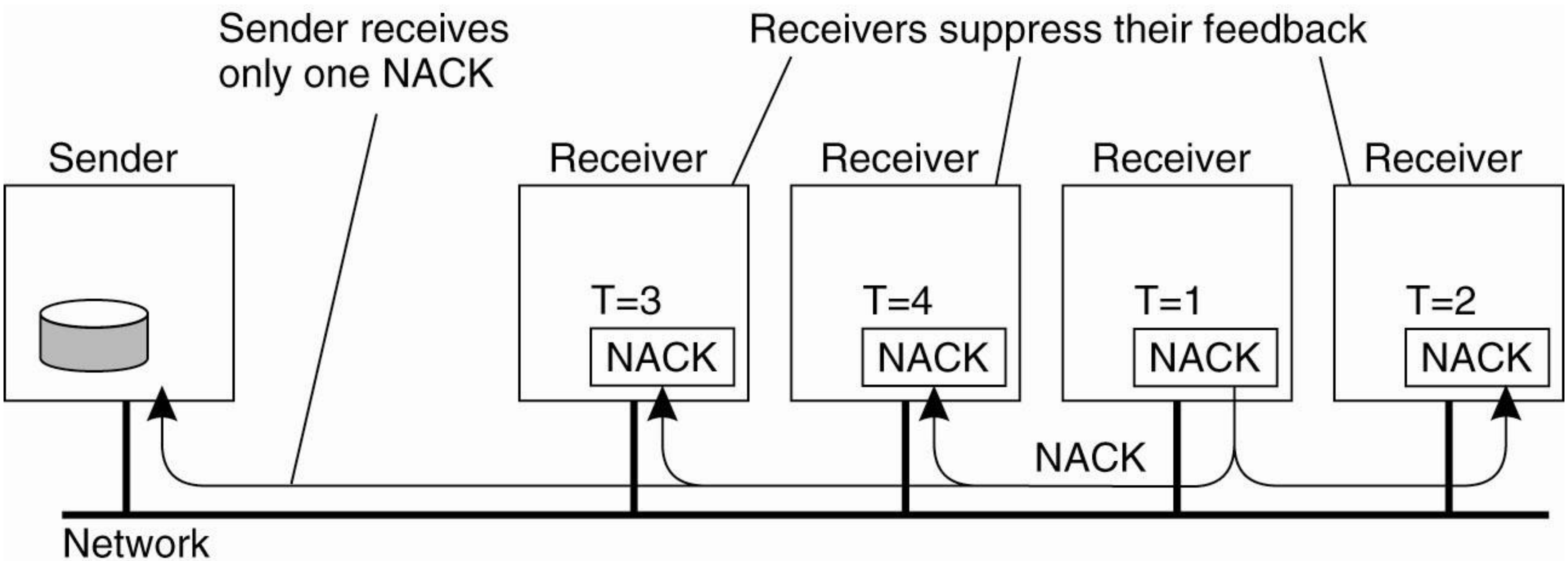


Figure 8-10. Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.

Hierarchical Feedback Control

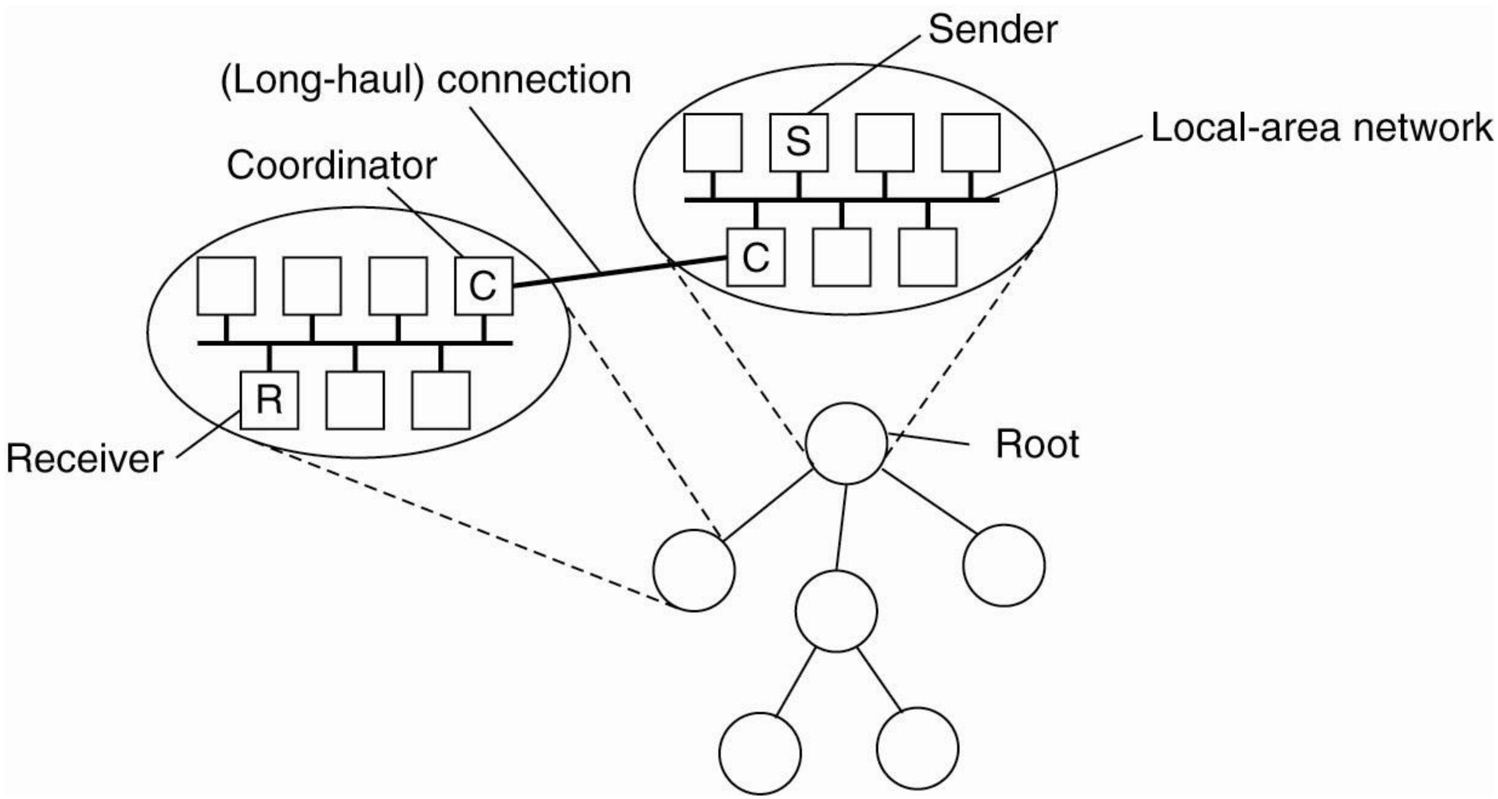


Figure 8-11. Hierarchical reliable multicasting: Each local coordinator forwards the message to its children and later handles retransmission requests.

Fault tolerance

- Concepts
- Process Resilience
- Reliable Group Communication
- Recovery

Recovery

- Checkpointing

Backward Recovery

- In backward recovery, the main issue is to bring the system from its present erroneous state back into a previously correct state.
  - To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong.
  - Each time (part of) the system's present state is recorded, a checkpoint is said to be made.

Forward Recovery

- Another form of error recovery is forward recovery.
- In this case, when the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute.
  - The main problem with forward error recovery mechanisms is that it has to be known in advance which errors may occur.

Erasure Correction

- In this approach, a missing packet is constructed from other, successfully delivered packets.
- For example, in an *(n, k)* block erasure code, a set of *k source packets* is encoded into a set of *n encoded packets,* such that *any* set of *n* encoded packets is enough to reconstruct the original *k* source packets.
- If not enough packets have yet been delivered, the sender will have to continue transmitting packets until a previously lost packet can be constructed.
- Erasure correction is a typical example of a forward error recovery approach.

Checkpointing

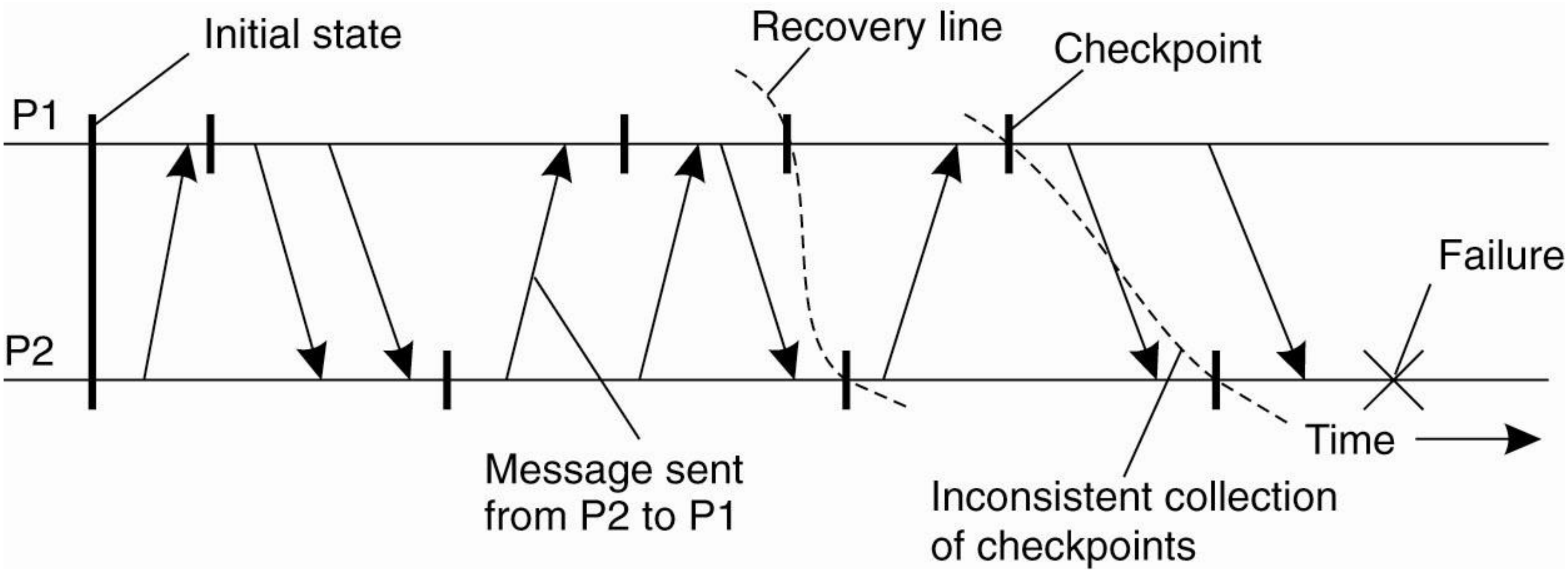


Figure 8-24. A recovery line.

We need to record a consistent global state, also called a distributed snapshot.

In a distributed snapshot, if a process *P* has recorded the receipt of a message, then there should also be a process *Q* that has recorded the sending of that message.

After all, it must have come from somewhere.

Independent Checkpointing and the domino effect

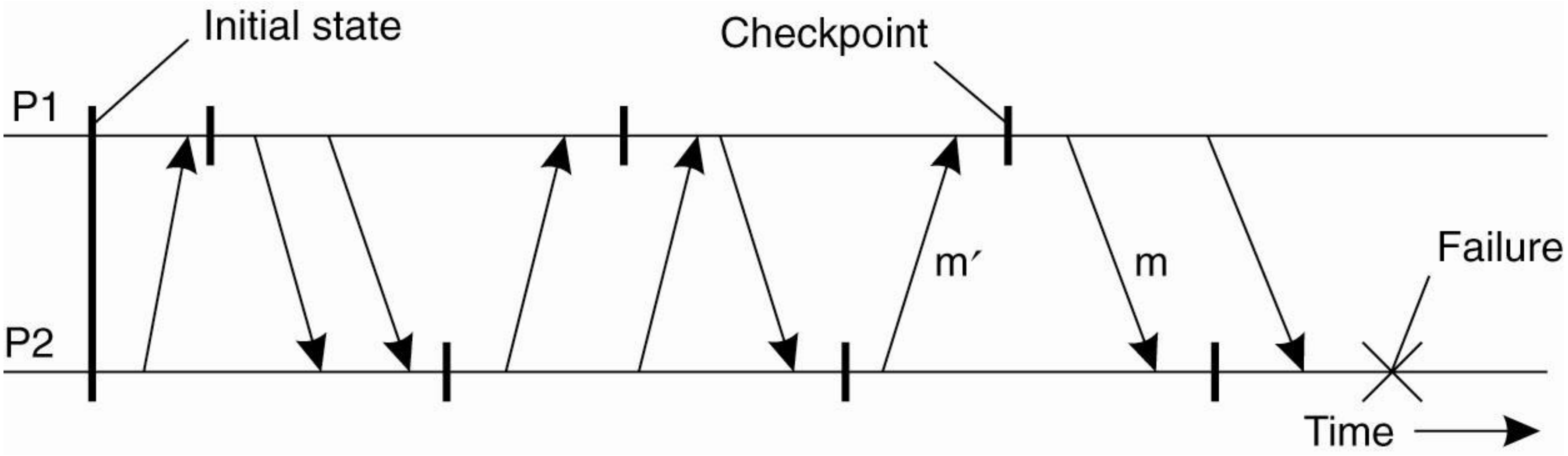


Figure 8-25. The domino effect.

Coordinated Checkpointing

- In coordinated checkpointing all processes synchronize to jointly write their state to local stable storage.
  - The main advantage of coordinated checkpointing is that the saved state is automatically globally consistent, so that cascaded rollbacks leading to the domino effect are avoided.
- A solution is to use a two-phase blocking protocol.
  - A coordinator first multicasts a *CHECKPOINT_REQUEST* message to all processes.
  - When a process receives such a message, it takes a local checkpoint, queues any subsequent message handed to it by the application it is executing, and acknowledges to the coordinator that it has taken a checkpoint.
- When the coordinator has received an acknowledgment from all processes, it multicasts a *CHECKPOINT_DONE* message to allow the (blocked) processes to continue.

End of PART I

- Readings
  - Distributed Systems, Chapter 8

Part II – Lab Session

Distributed Objects

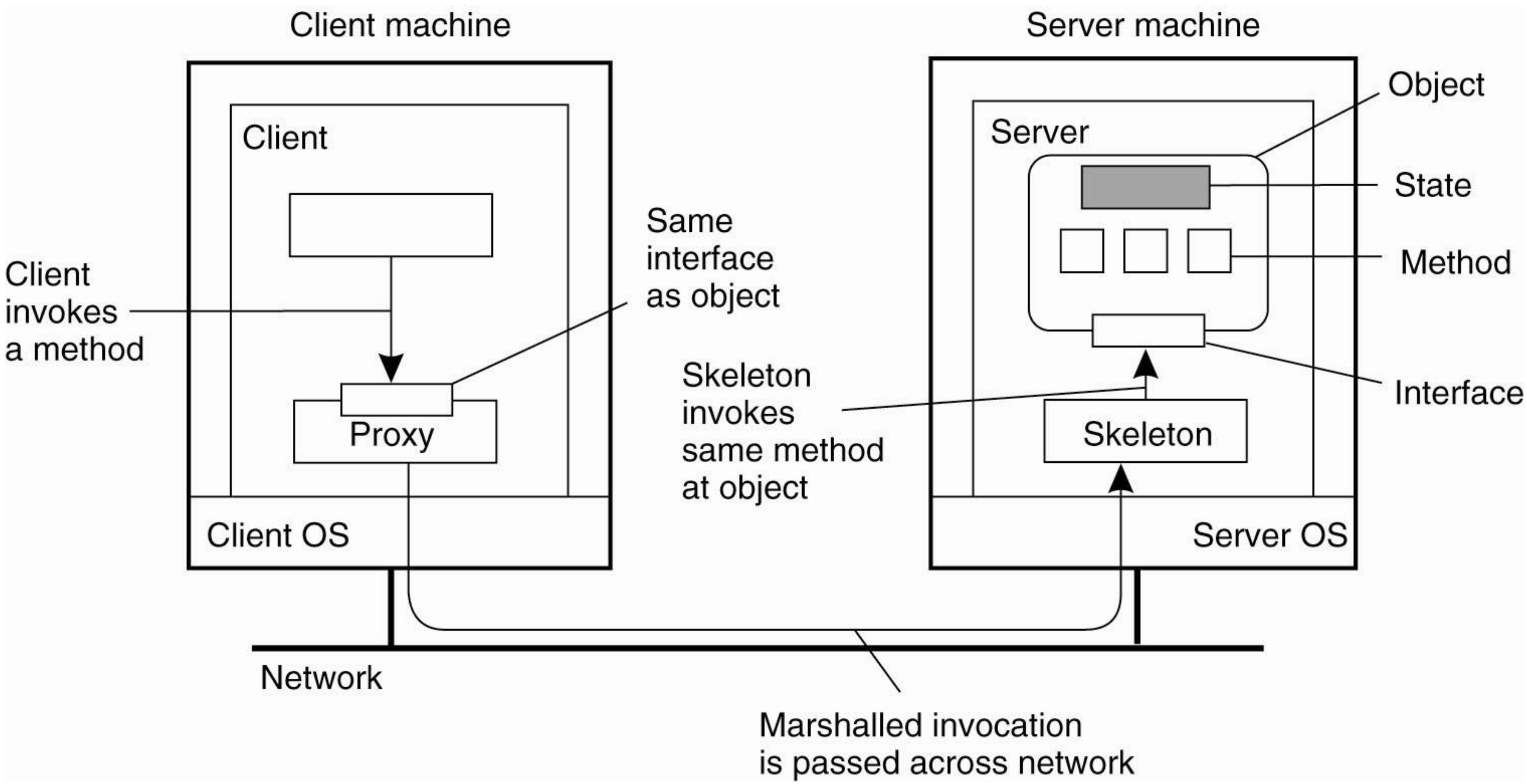
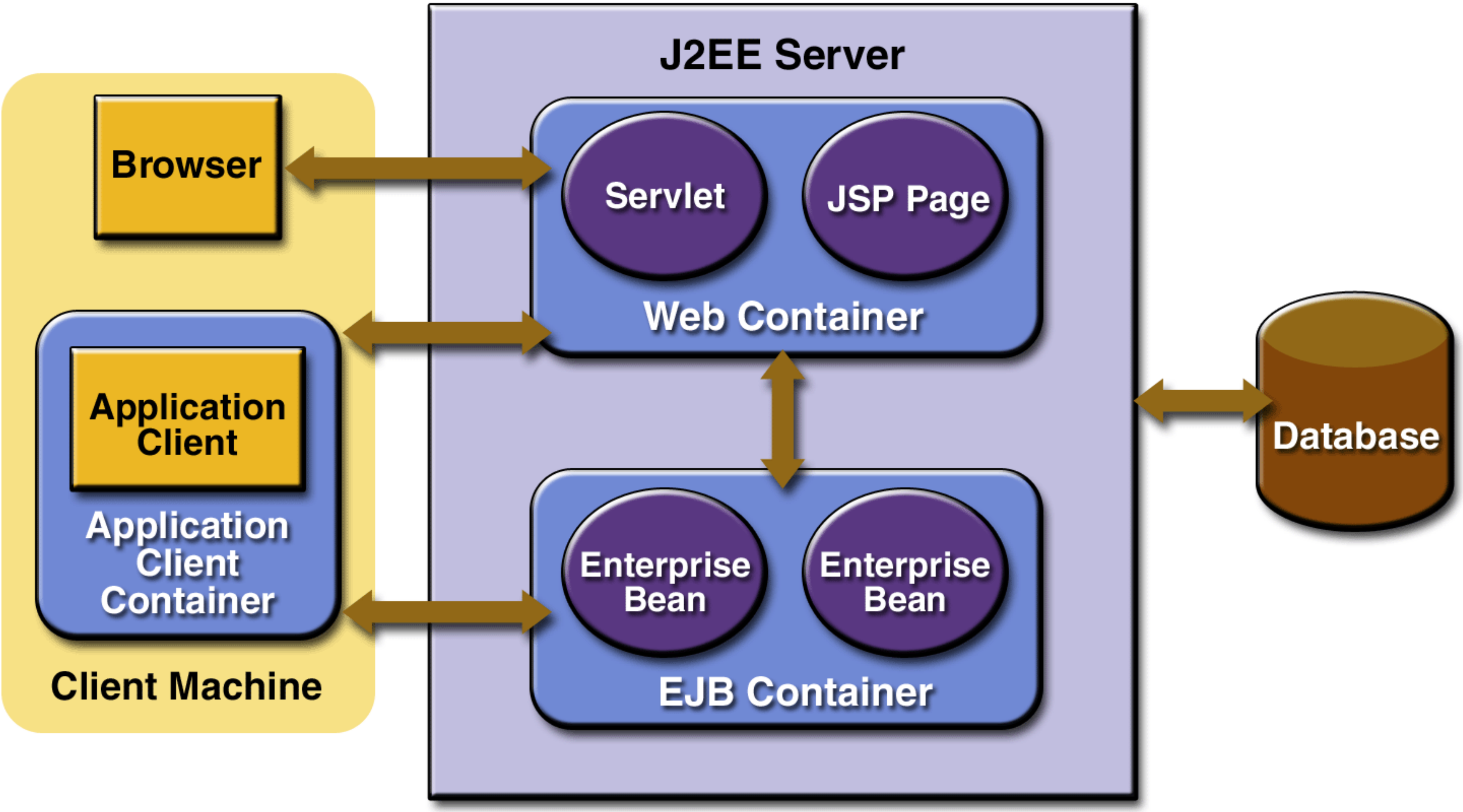


Figure 10-1. Common organization of a remote object with client-side proxy.

J2EE


J2EE Server
Browser
Servlet
JSP Page
Web Container
Application Client
Application Client Container
Enterprise Bean
Enterprise Bean
EJB Container
Database
Client Machine

Example: Enterprise Java Beans

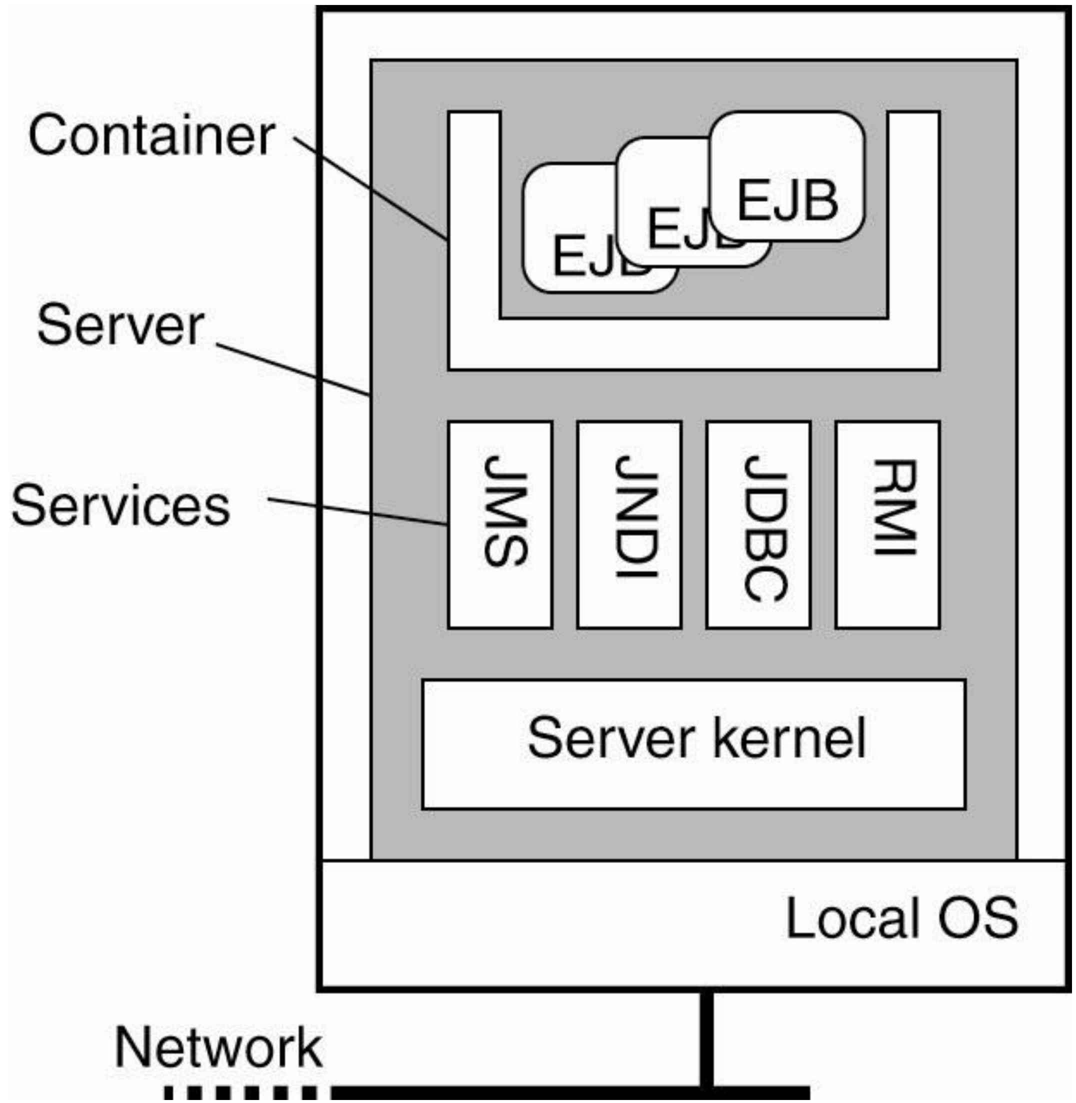


Figure 10-2. General architecture of an EJB server.

Enterprise JavaBeans (EJB)

- A managed, server-side component architecture for modular construction of enterprise applications.
- The EJB specification is one of several Java APIs in the Java EE specification.
- EJB is a server-side model that encapsulates the business logic of an application.
- The EJB specification intends to provide a standard way to implement the back-end 'business' code typically found in enterprise applications (as opposed to 'front-end' interface code).
- EJBs are intended to handle such common concerns as persistence, transactional integrity, and security in a standard way, leaving programmers free to concentrate on the particular problem at hand.

The container

- The important issue is that an EJB is embedded inside a container which effectively provides interfaces to underlying services that are implemented by the application server.
- The container can more or less automatically bind the EJB to these services, meaning that the correct references are readily available to a programmer.
- Typical services include those for remote method invocation (RMI), database access (JDBC), naming (JNDI), and messaging (JMS).

Java EE Application Server

- The Enterprise JavaBean specification defines the roles played by the EJB container and the EJBs as well as how to deploy the EJBs in a container.

- To deploy and run EJB beans, a Java EE Application server can be used, as these include an EJB container by default.

Persistent Vs. Transient Objects

- A persistent object is one that continues to exist even if it is currently not contained in the address space of any server process.
  - In practice, this means that the server that is currently managing the persistent object, can store the object's state on secondary storage and then exit.
  - Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests.

Persistent Vs. Transient Objects

- A transient object is an object that exists only as long as the server that is hosting the object.
  - As soon as that server exits, the object ceases to exist as well.

- To take the discussion away from middleware issues, most object-based distributed systems simply support both types.

Remote Objects

- A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is *not* distributed: it resides at a single machine.
- Only the interfaces implemented by the object are made available on other machines.
- Such objects are also referred to as remote objects.

Session Beans

- A *session bean* represents a single client inside the J2EE server.

- To access an application that is deployed on the server, the client invokes the session bean's methods.

- The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

Session Beans

- A session bean is similar to an interactive session.
- A session bean is not shared - it may have just one client, in the same way that an interactive session may have just one user.
- Like an interactive session, a session bean is not persistent.
  - That is, its data is not saved to a database.
  - When the client terminates, its session bean appears to terminate and is no longer associated with the client.

Stateful Session Beans

- The state of an object consists of the values of its instance variables.
- In a stateful session bean, the instance variables represent the state of a unique client-bean session.
- Because the client interacts ("talks") with its bean, this state is often called the *conversational state*.
- The state is retained for the duration of the client-bean session.
- If the client removes the bean or terminates, the session ends and the state disappears.

Stateless Session Beans

- A stateless session bean does not maintain a conversational state for a particular client.

- When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation.

- When the method is finished, the state is no longer retained.

Practical Session: EJBs

- What do you need?
  - J2EE SDK
    - JSDK will install also Glassfish Server
  - NetBeans (or Eclipse)
    - This may come with his own Glassfish Server setup

Practical Session: EJBs

- Examples in Netbeans
  - Cart
    - EJB (stateful bean)
  - Counter
    - Facelets + EJB (singleton bean)
  - Converter
    - Java Servlets + EJB (stateless bean)
  - HelloService
    - Web Service + EJB (stateless bean)
  - Timer
    - Automatic: time-out every minute
    - Programmatic: time-out every N seconds from the setting of the timer, ex. 8 seconds.

Develop a simple Bank Manager Bean

- Develop the application based on Lab Manual on EJBs given in class

End of Lesson 8

- Readings
  - Distributed Systems, Chapter 10
- Lab Manual on EJBs given in class