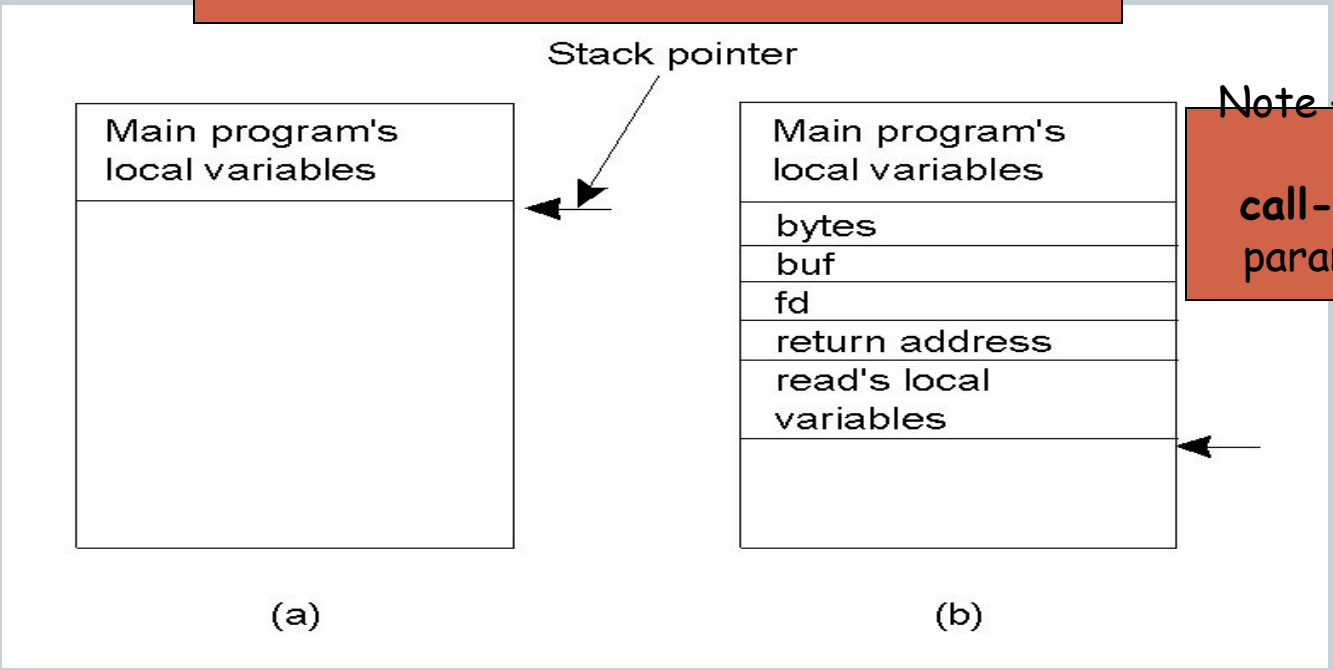
Communication

Remote Procedure Calls

a) Conventional Procedure Call

Count = read(fd, buf,bytes)



Note - **call-by-value** and **call-by-reference** parameters on the stack.

a) Parameter passing in a local procedure call: the stack before the call to read.
b) The stack while the called procedure is active.

Remote Procedure Call

- ***RPC concept :: to make a remote procedure call appear like a local procedure call.***
- The goal is to hide the details of the network communication (namely, the sending and receiving of messages).
- The calling procedure should not be aware that the called procedure is executing on a different machine.

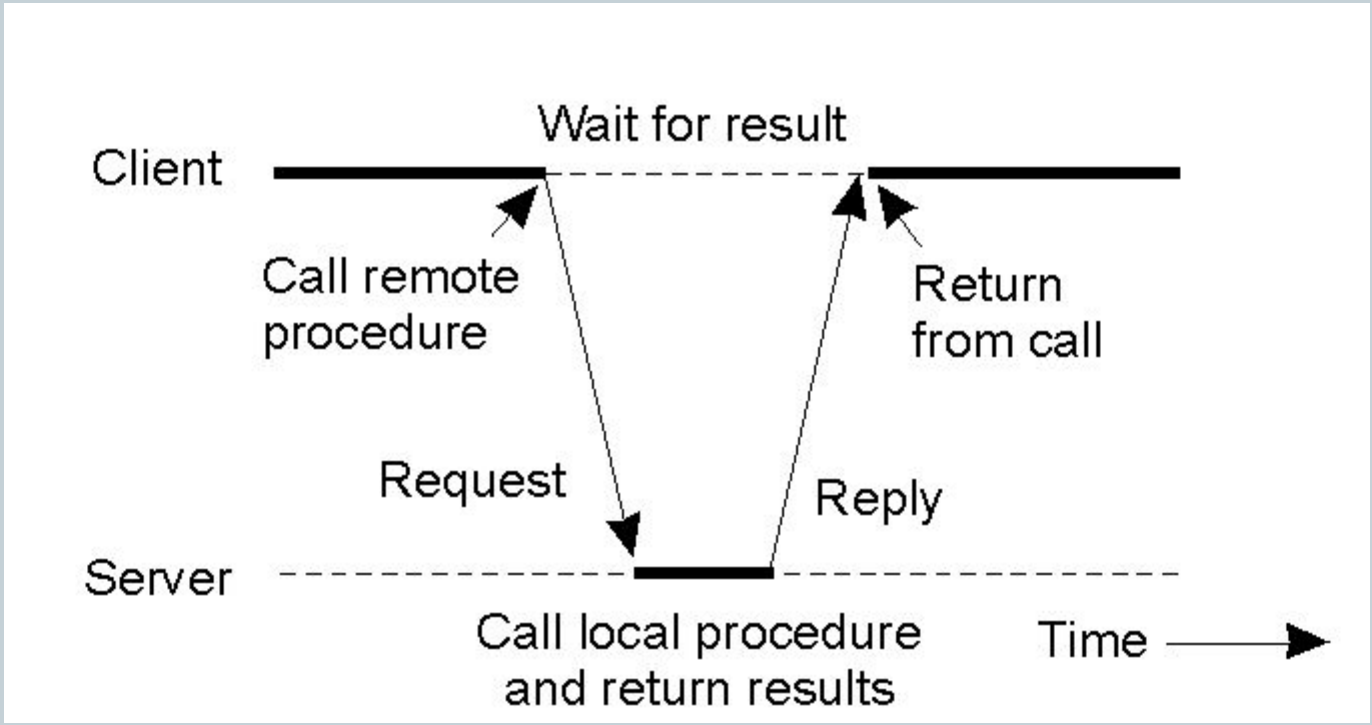
Remote Procedure Call

- When making a RPC:
  - The calling environment is suspended.
  - Procedure parameters are transferred across the network to the environment where the procedure is to execute.
  - The procedure is executed **there.**
  - When the procedure finishes, the results are transferred back to the calling environment.
  - Execution resumes as if returning from a regular procedure call.

RPC differs from OSI

- User does not open connection, read, write, then close connection – client may not even know they are using the network.
- RPC is well-suited for client-server interaction where the flow of control alternates.

RPC between Client and Server



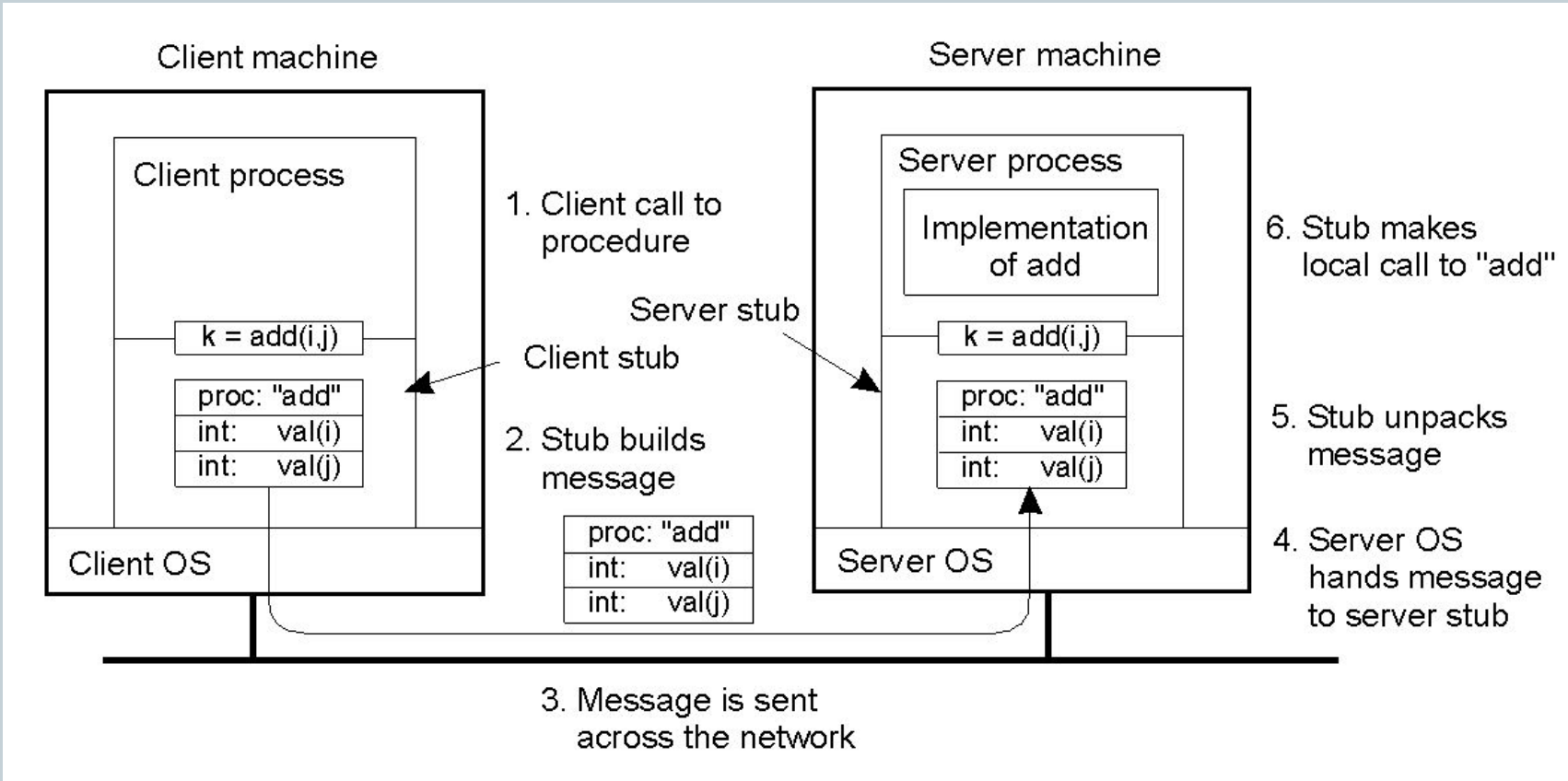
RPC Steps

1. The client procedure calls a **client stub** passing parameters in the normal way.
2. The client stub ***marshals the parameters,*** builds the message, and calls the local OS.
3. The client's OS sends the message **(using the transport layer)** to the remote OS.
4. The server remote OS gives **transport layer** message to a **server stub**.
5. The server stub **demarshals the parameters** and calls the desired server routine.

RPC Steps

6. The server routine does work and returns result to the server stub via normal procedures.
7. The server stub ***marshals the return values*** into the message and calls local OS.
8. The server OS **(using the transport layer)** sends the message to the client's OS.
9. The client's OS gives the message to the client stub
10. The client stub ***demarshals the result***, and execution returns to the client.

RPC Steps


Client machine
Server machine
Client process
Server process
Implementation of add
k = add(i,j)
k = add(i,j)
proc: "add"
int: val(i)
int: val(j)
proc: "add"
int: val(i)
int: val(j)
1. Client call to procedure
Server stub
Client stub
2. Stub builds message
proc: "add"
int: val(i)
int: val(j)
6. Stub makes local call to "add"
5. Stub unpacks message
4. Server OS hands message to server stub
Client OS
Server OS
3. Message is sent across the network

Marshaling Parameters

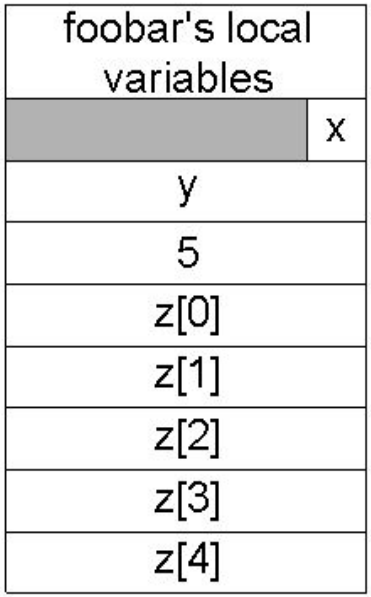
- Parameters must be marshaled into a standard representation.
- Parameters consist of simple types (e.g. integers) and compound types (e.g., C structures).
- The type of each parameter must be known to the modules doing the conversion into standard representation.
- *Call-by-reference is not possible in parameter passing*

Parameter Specification and Stub Generation

The caller and the callee must agree on the format of the message they exchange, and they must follow the same steps when it comes to passing complex data structures.

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

(a)



(b)

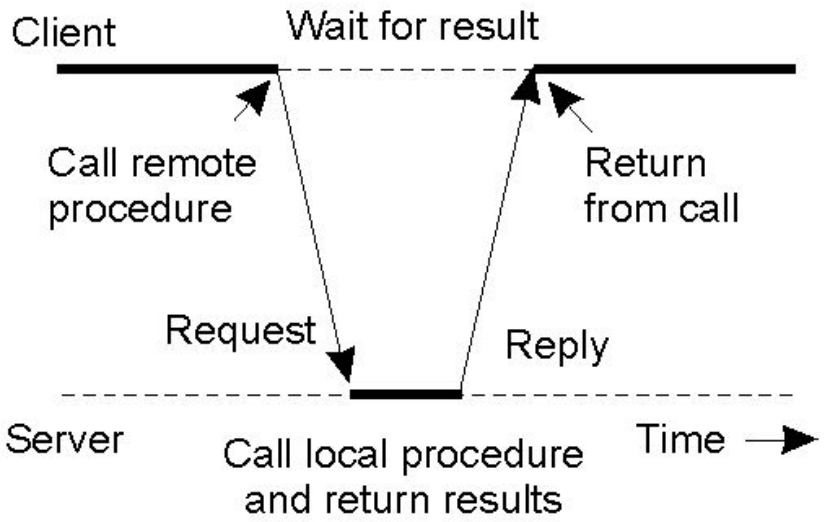
A procedure

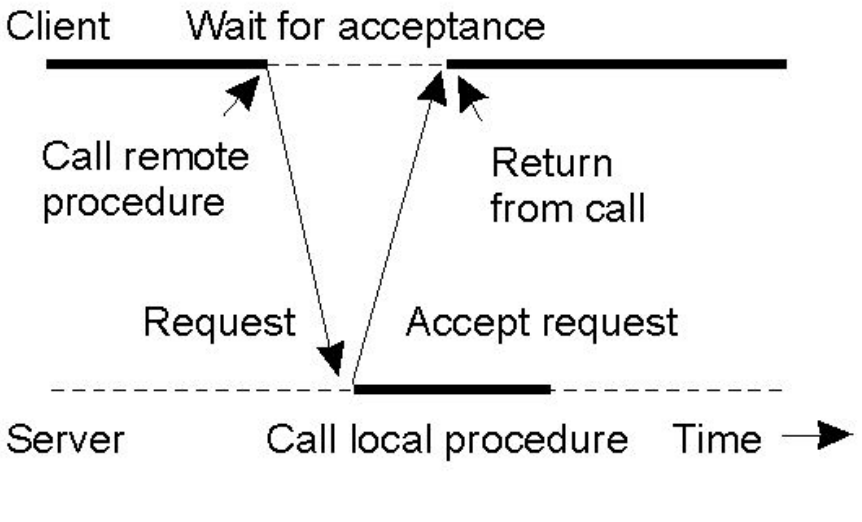
The corresponding message

RPC Details

- *An Interface Definition Language (IDL) is used to specific the interface that can be called by a client and implemented by the server.*
- *All RPC-based middleware systems offer an IDL to support application development.*

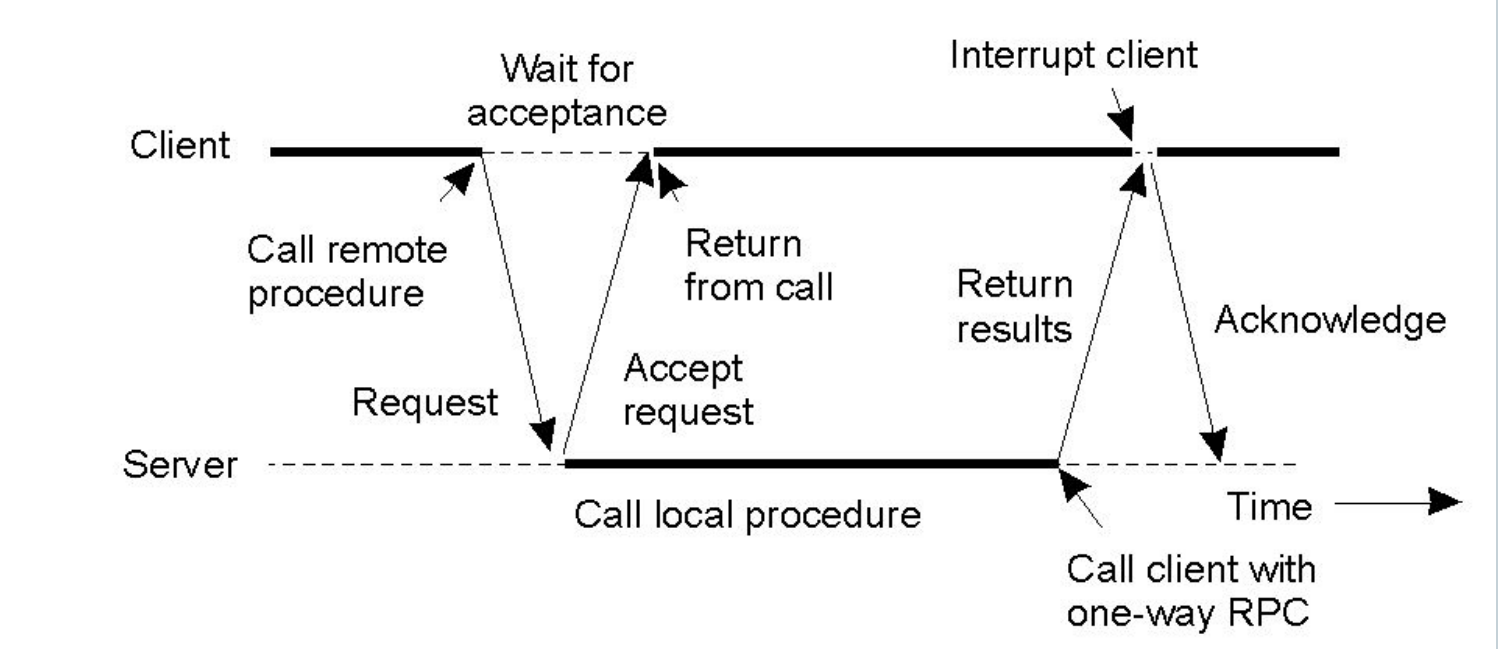
Asynchronous RPC





a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Asynchronous RPC's



A client and server interacting through two asynchronous RPCs

DCE RPC

- Developed by OSF
- DCE includes no. of services
  - Distributed file system
  - Directory service
  - Security service
  - Distributed time service (keep clock globally synchronized)
- Goal
  - To make it possible for a client to access a remote service by simply calling local procedural.
- Binding a client to a server

Message-Oriented Communication

a) Message-Oriented Transient Communication:

i) Berkeley Sockets.

ii) Message Passing Interface

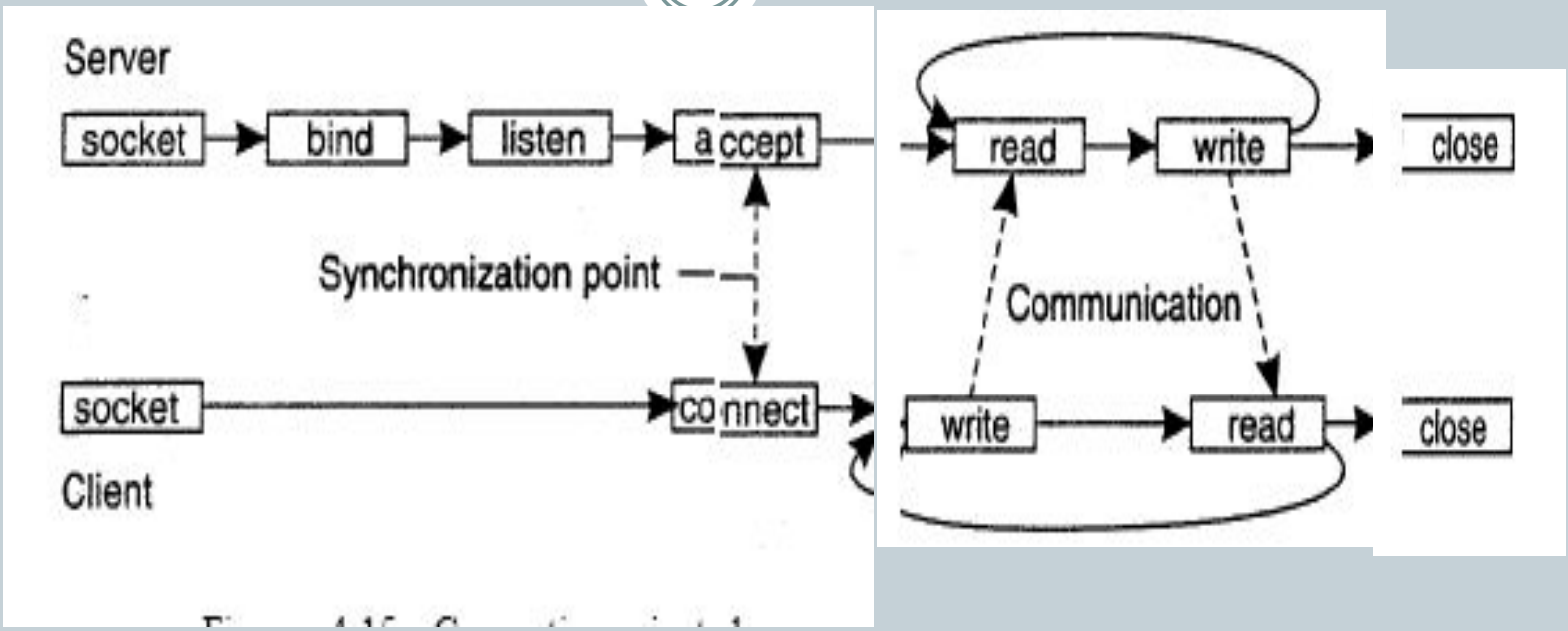
b) Message-Oriented Persistent Communication:

Message oriented transient communications
Berkeley Sockets

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

- Socket primitives for TCP/IP.

Continue…



CONNECTION ORIENTED COMMUNICATION PATTERN USING SOCKETS

;

The Message-Passing Interface (MPI)

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

- Some of the most intuitive message-passing primitives of MPI.

MPI uses the underlying network and it assumes communication take place within a known group of processes

Persistent vs. Transient Communication

Persistent communication: A message is stored at a communication server as long as it takes to deliver it at the receiver.

Transient communication: A message is discarded by a communication server as soon as it cannot be delivered at the next server, or at the receiver.

Synchronization

Clock Synchronization

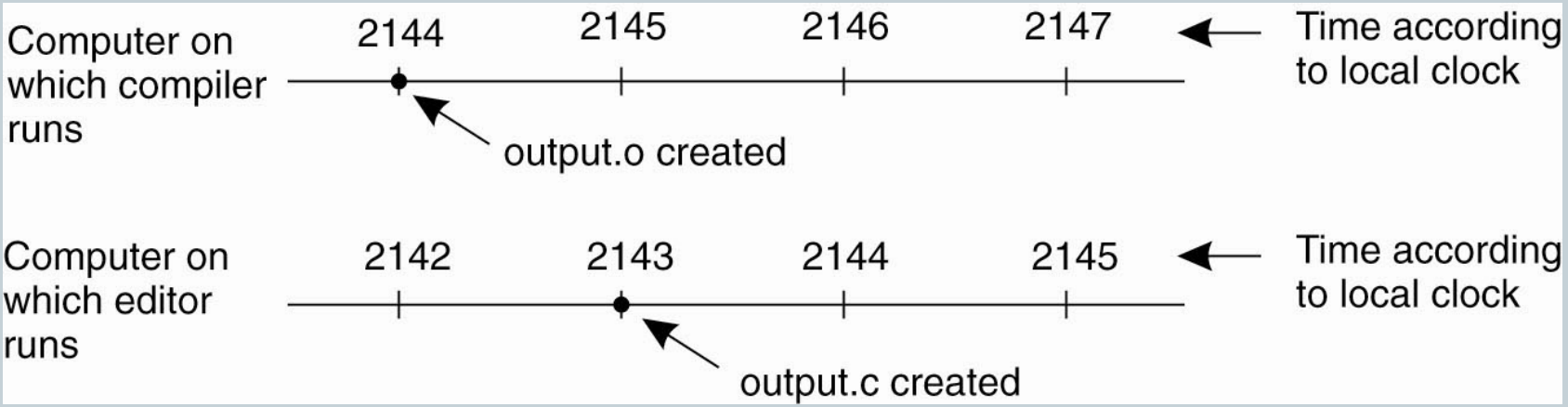
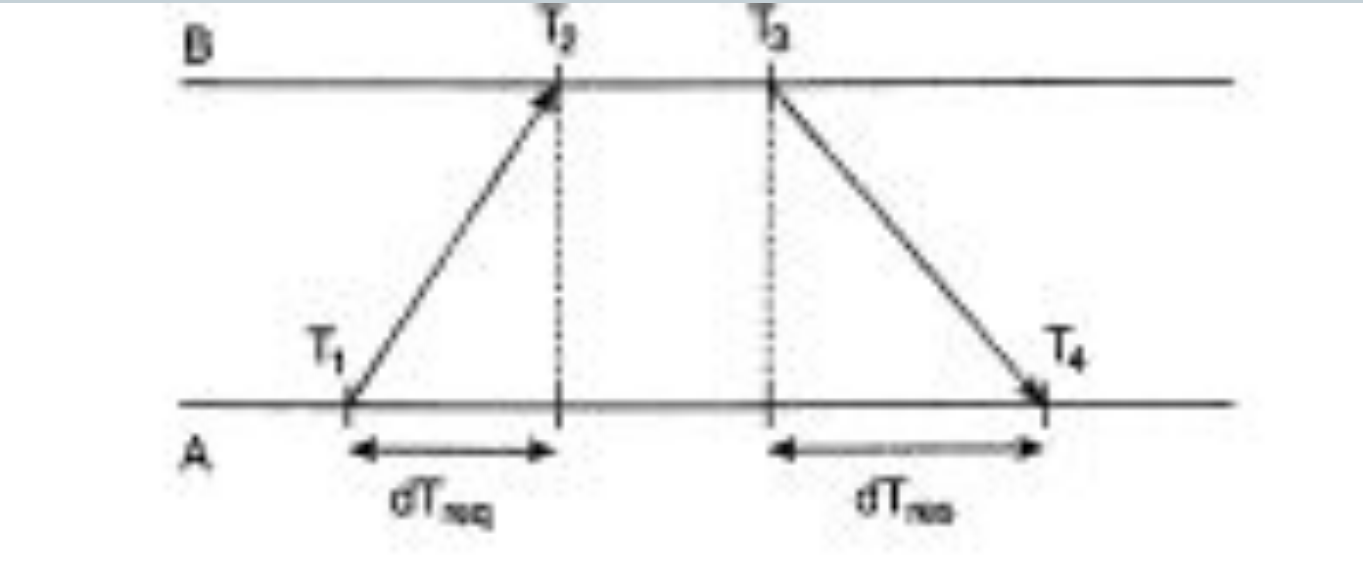


Figure. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

NTP-Network Time Protocol



The Berkeley Algorithm (1)

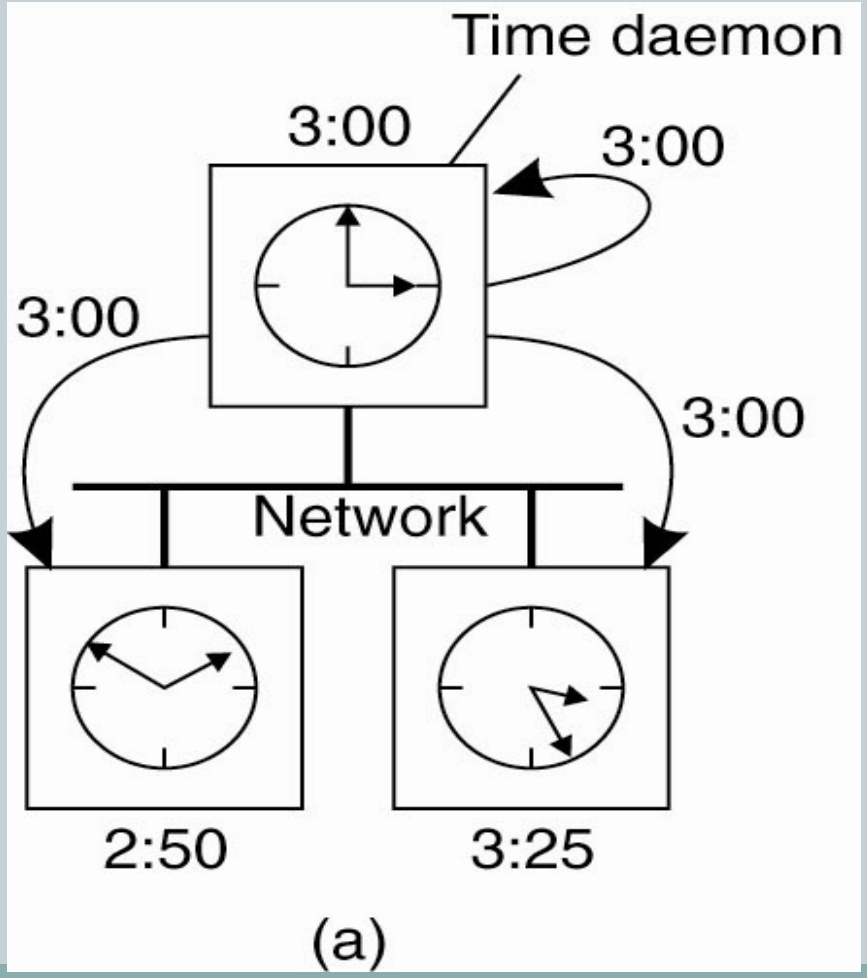


Figure. (a) The time daemon asks all the other machines for their clock values.

The Berkeley Algorithm (2)

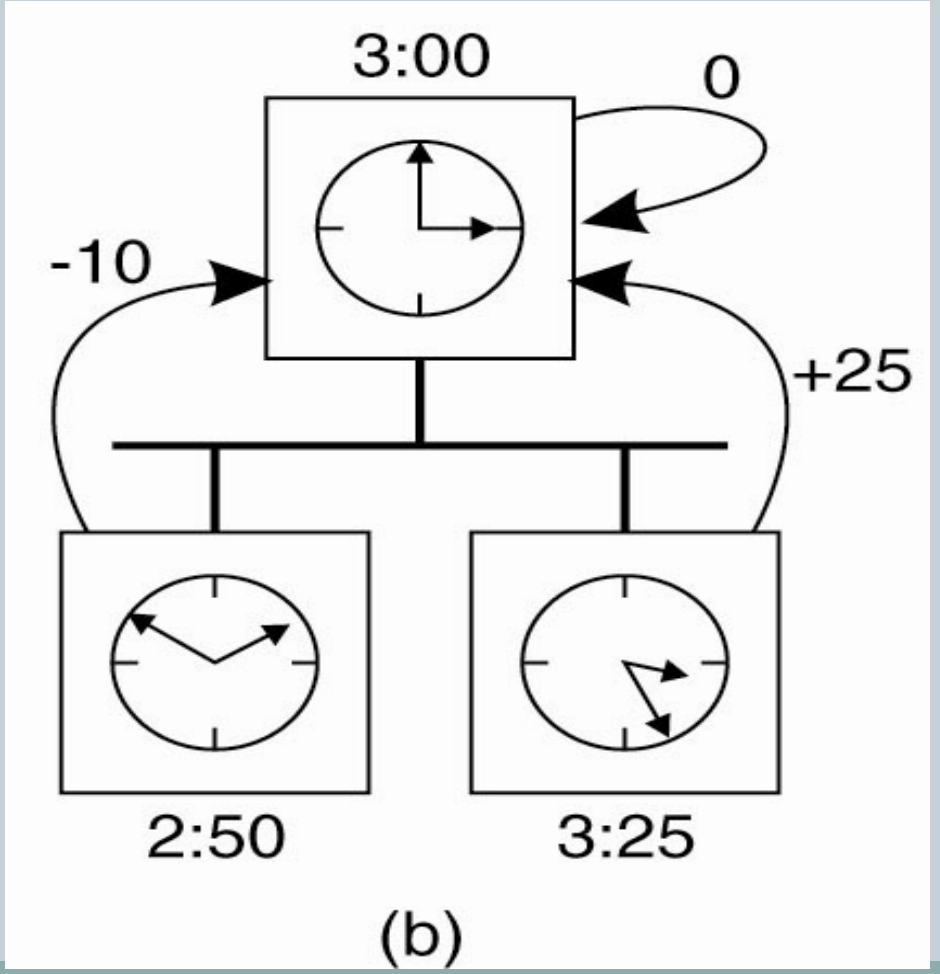
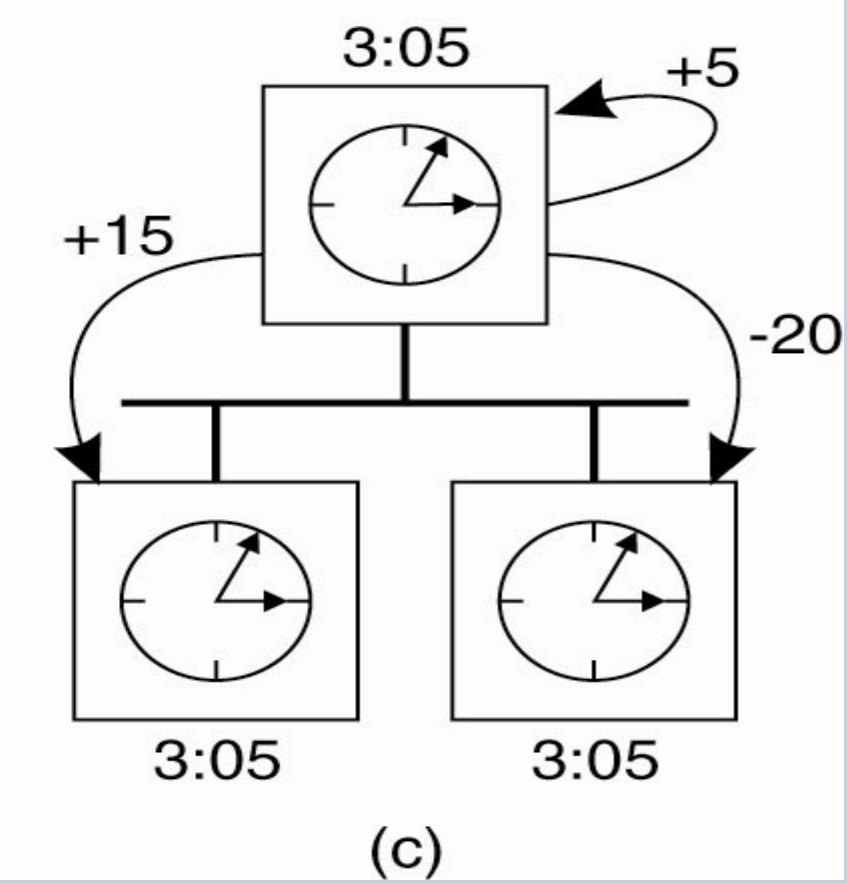


Figure .

(b) The machines answer.

The Berkeley Algorithm (3)

Figure . (c) The time daemon tells everyone how to adjust their clock.



Lamport's Logical Clocks

Logical Clocks

- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use *logical* clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the *order* in which events occur rather than the *time* at which they occurred

Event Ordering

- *Problem:* define a total ordering of all events that occur in a system
- Events in a single processor machine are totally ordered
- In a distributed system:
  - No global clock, local clocks may be unsynchronized
  - Can not order events on different machines using local times
- Key idea [Lamport]
  - Processes exchange messages
  - Message must be sent before received
  - Send/receive used to order events (and synchronize clocks)

Happened Before Relation

- If *A* and *B* are events in the same process and *A* executed before *B*, then *A* -> *B*
- If A represents sending of a message and B is the receipt of this message, then A -> B
- Relation is transitive:
  - A -> B and B -> C => A -> C
- Relation is undefined across processes that do not exchange messages
  - Partial ordering on events

Event Ordering Using *HB*

- Goal: define the notion of time of an event such that
  - If A-> B then C(A) < C(B)
  - If A and B are concurrent, then C(A) <, = or > C(B)
- Solution:
  - Each processor maintains a logical clock LC_i
  - Whenever an event occurs locally at I, $LC_i = LC_i + 1$
  - When *i* sends message to *j*, piggyback Lc_i
  - When *j* receives message from *i*
    - If $LC_j < LC_i$ then $LC_j = LC_i + 1$ else do nothing
  - Claim: this algorithm meets the above goals

Lamport's Logical Clocks

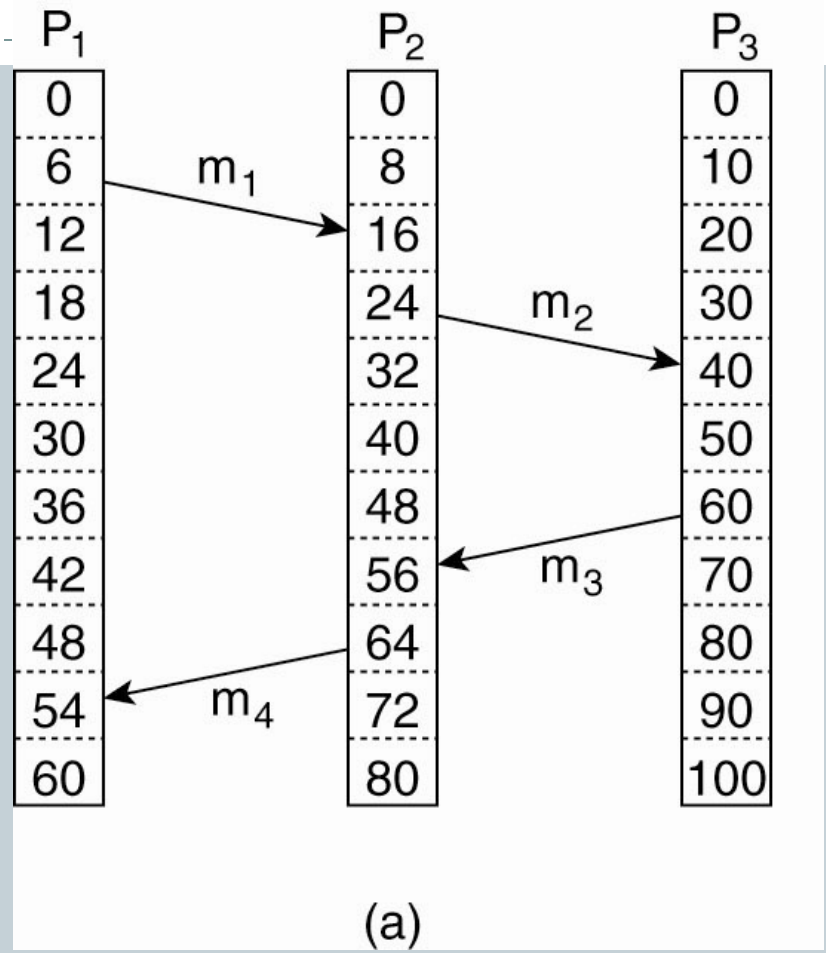


Figure . (a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks

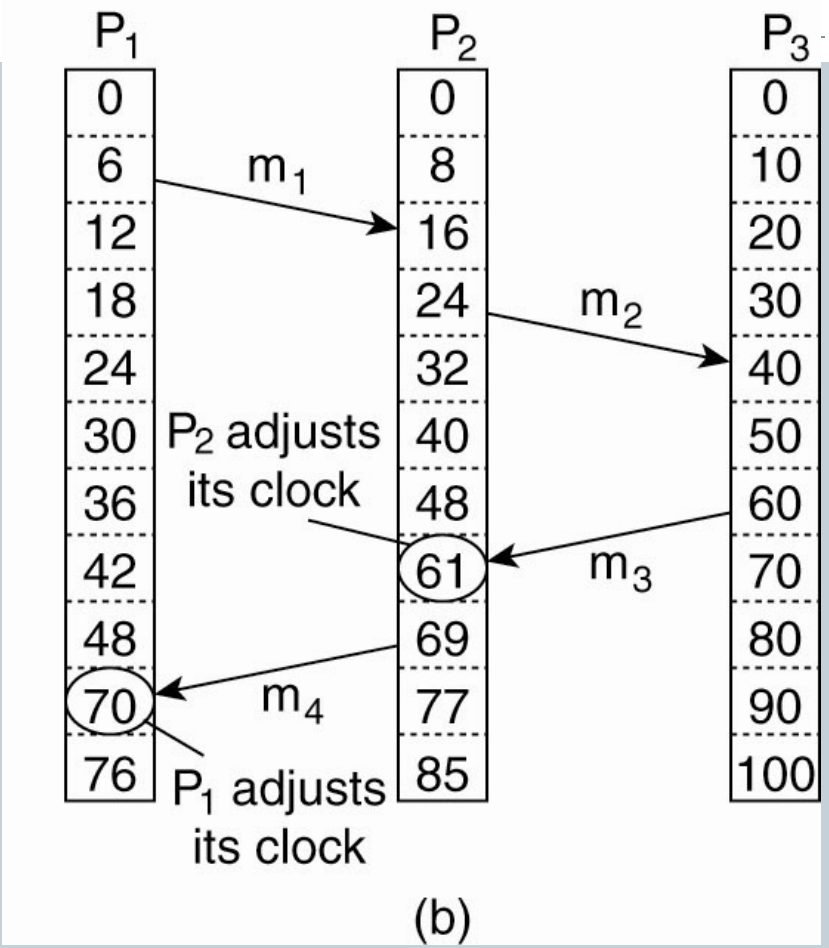


Figure . (b) Lamport's algorithm corrects the clocks.

Lamport's Logical Clocks

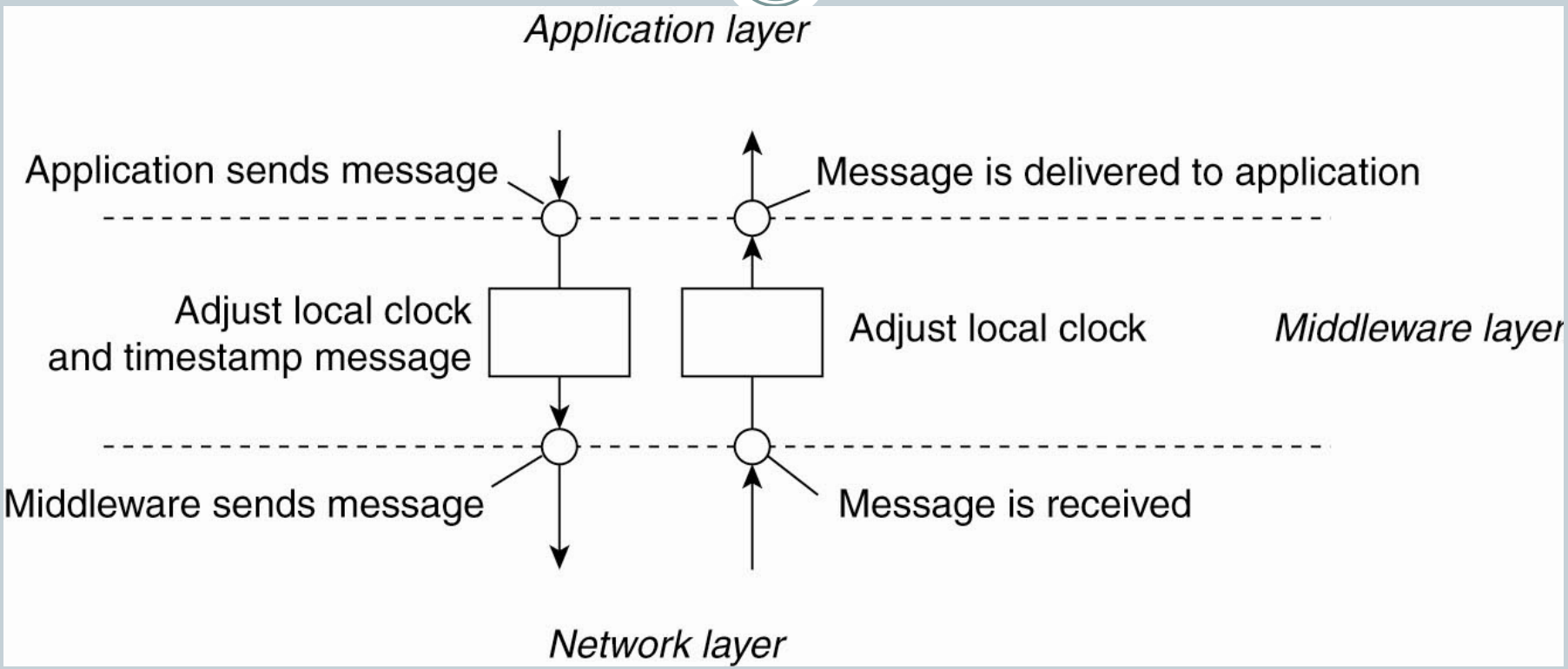


Figure . The positioning of Lamport's logical clocks in distributed systems.

Example: Totally Ordered Multicasting

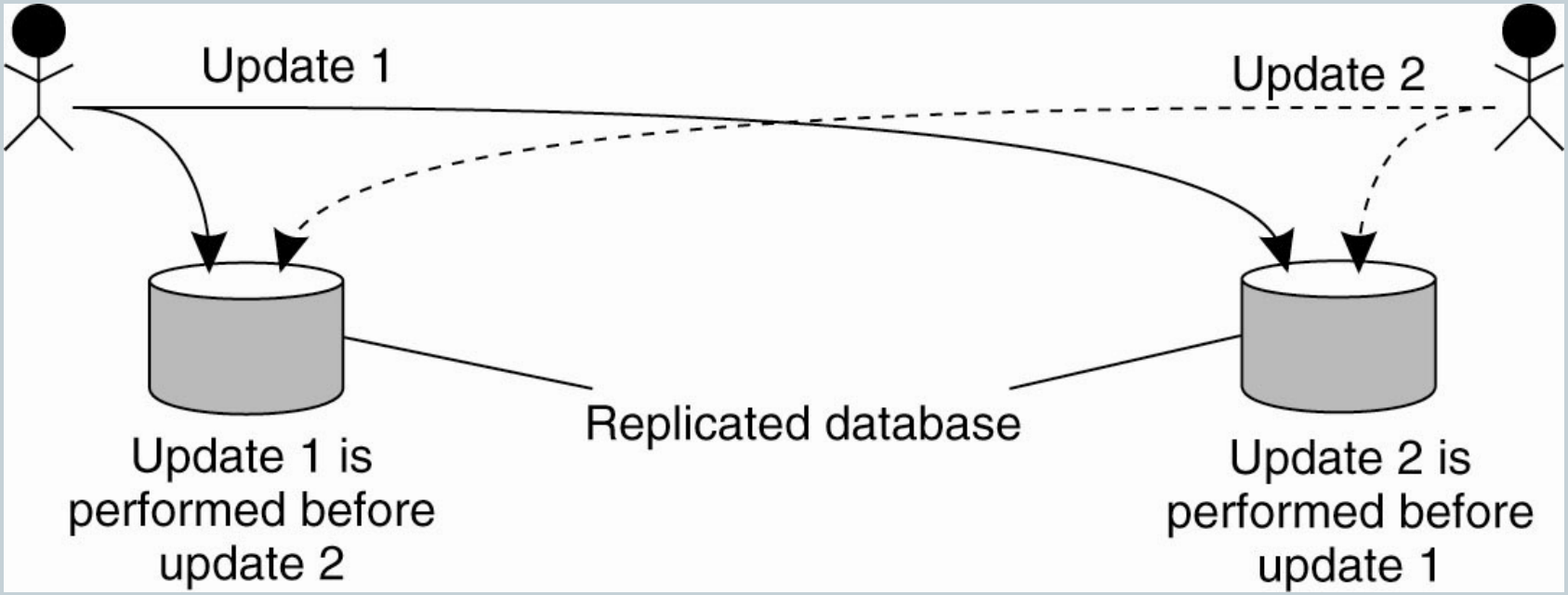


Figure . Updating a replicated database and leaving it in an inconsistent state.

Election Algorithms

Election Algorithms

- Many distributed algorithms need one process to act as coordinator
  - Doesn't matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (aka *leader election*)
- Examples: take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
- Types of election algorithms: Bully and Ring algorithms

Bully Algorithm

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election, OK, I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with n processes

Bully Algorithm Details

- Any process *P* can initiate an election
- *P* sends *Election* messages to all process with higher Ids and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, it treats sender an coordinator

The Bully Algorithm (1)

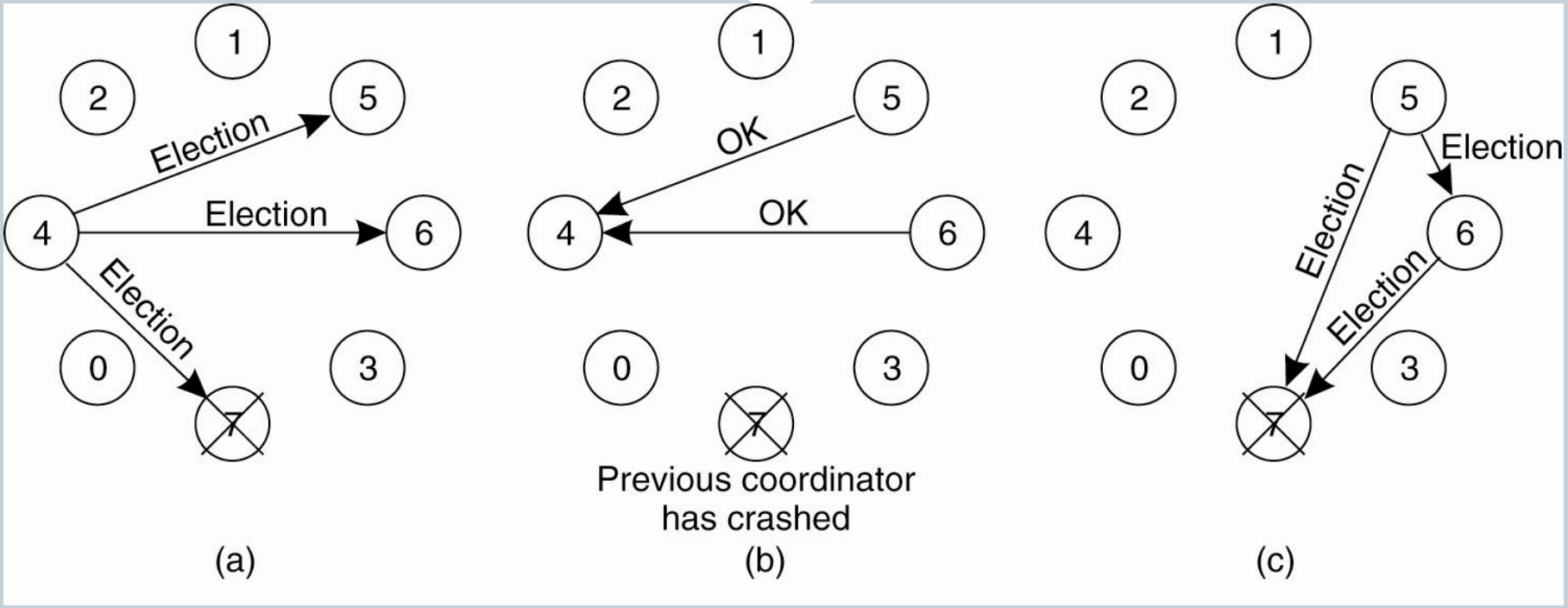


Figure. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

The Bully Algorithm (2)

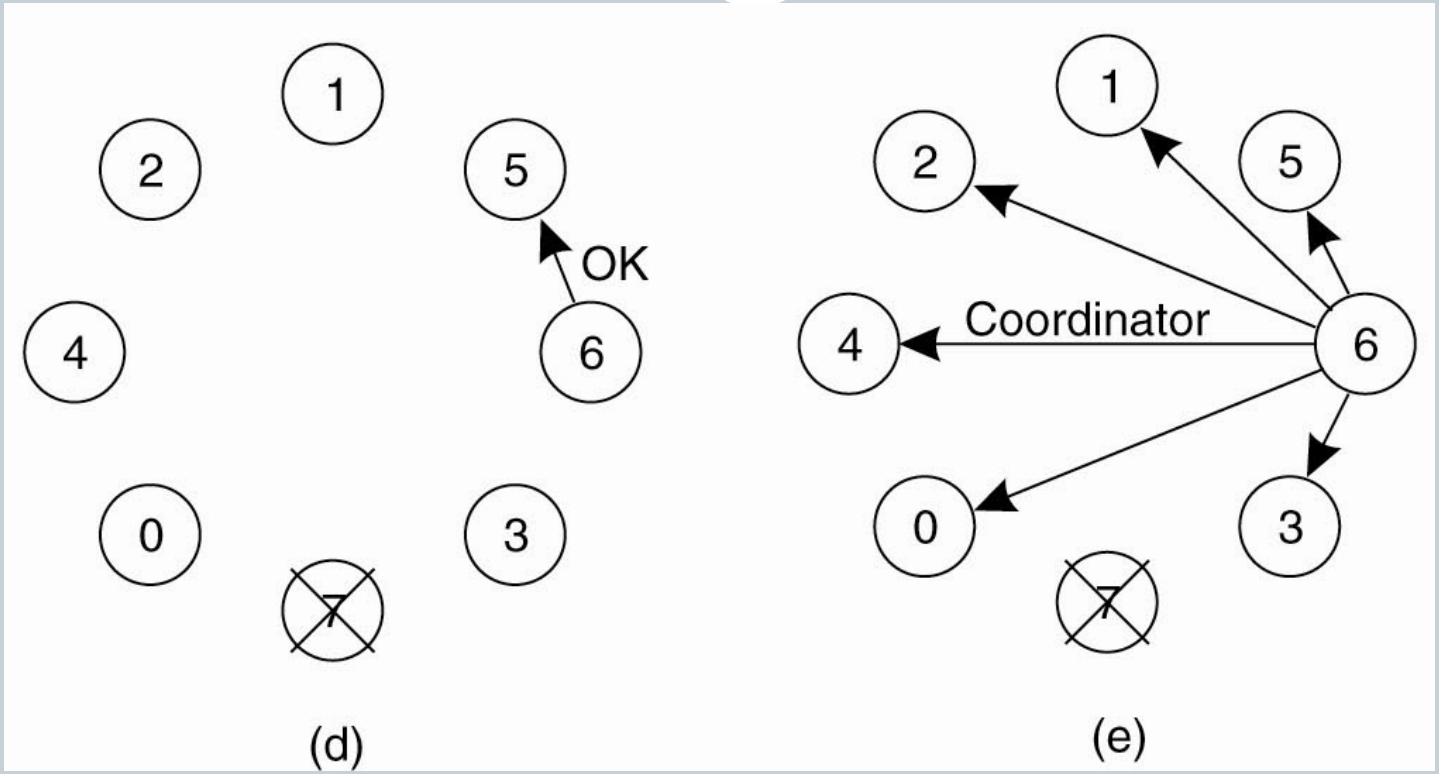


Figure. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Ring-based Election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
  - Select process with highest ID
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

A Ring Algorithm

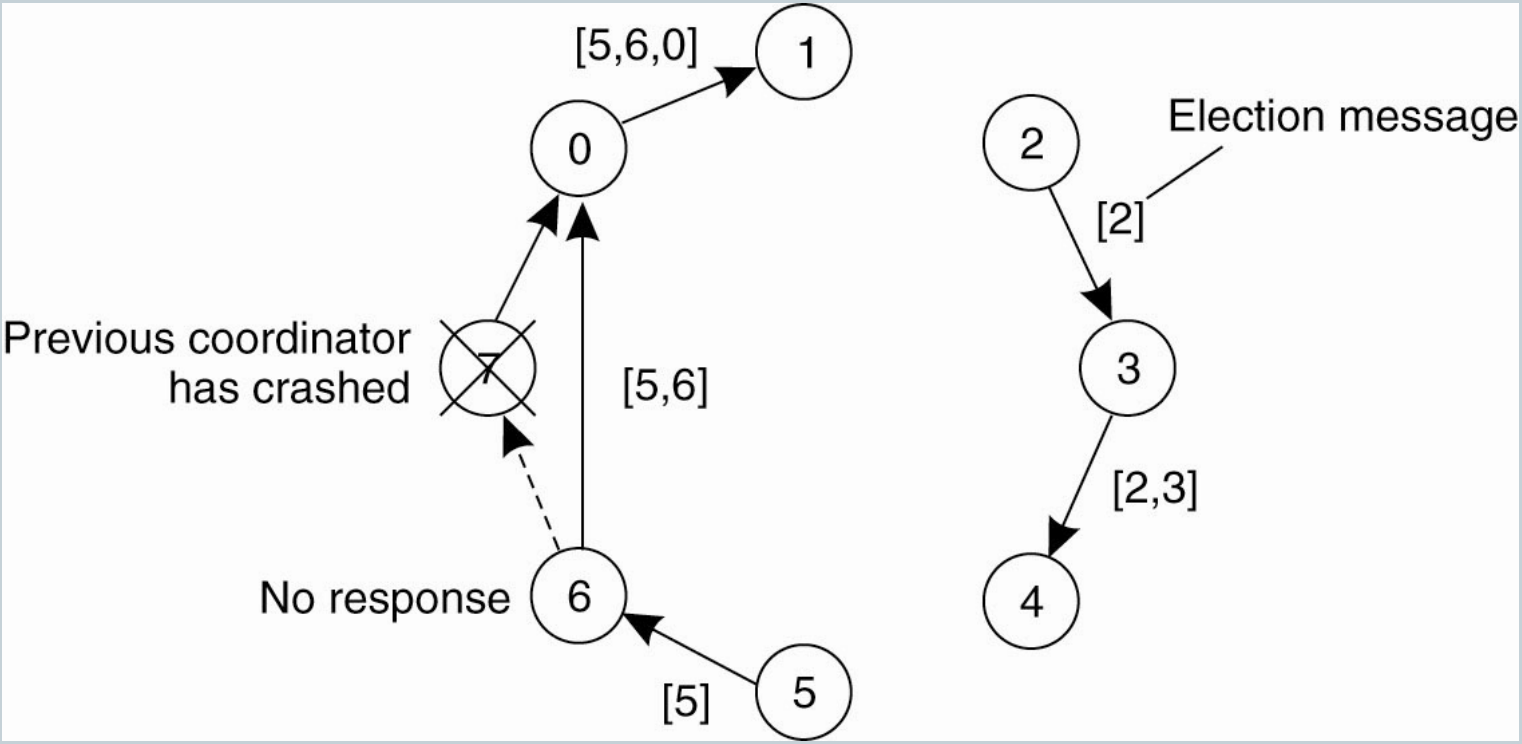


Figure 6-21. Election algorithm using a ring.

Mutual Exclusion: A Centralized Algorithm

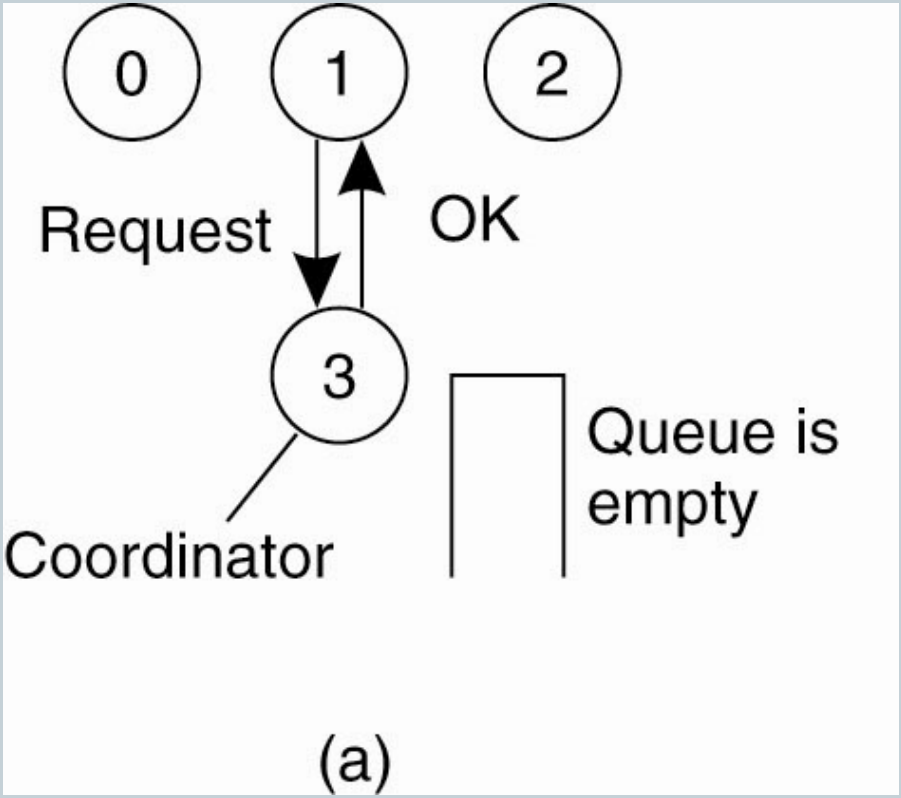


Figure . (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

Mutual Exclusion: A Centralized Algorithm

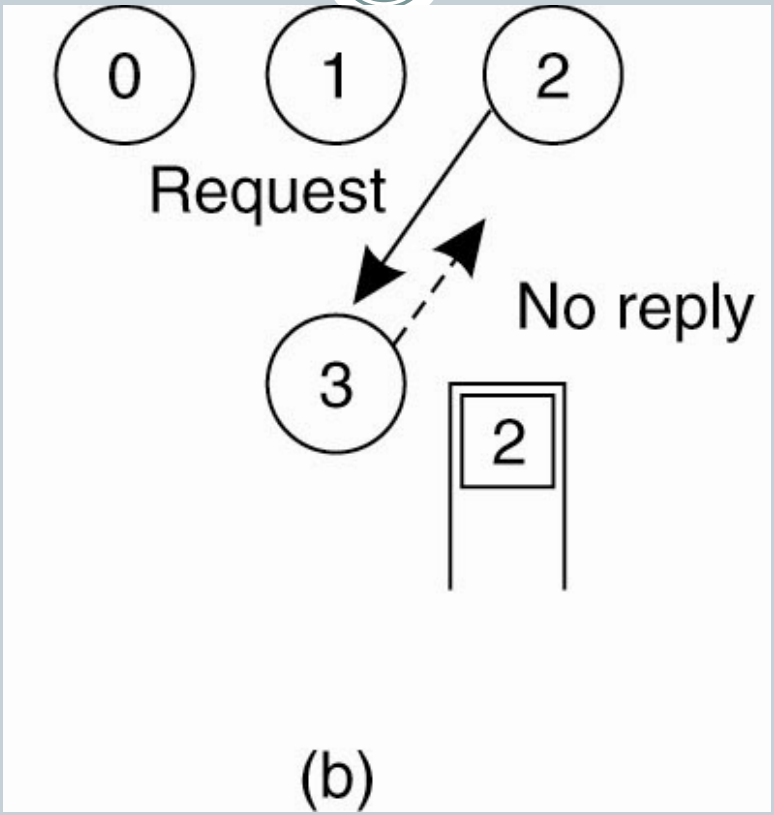


Figure . (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

Mutual Exclusion : A Centralized Algorithm

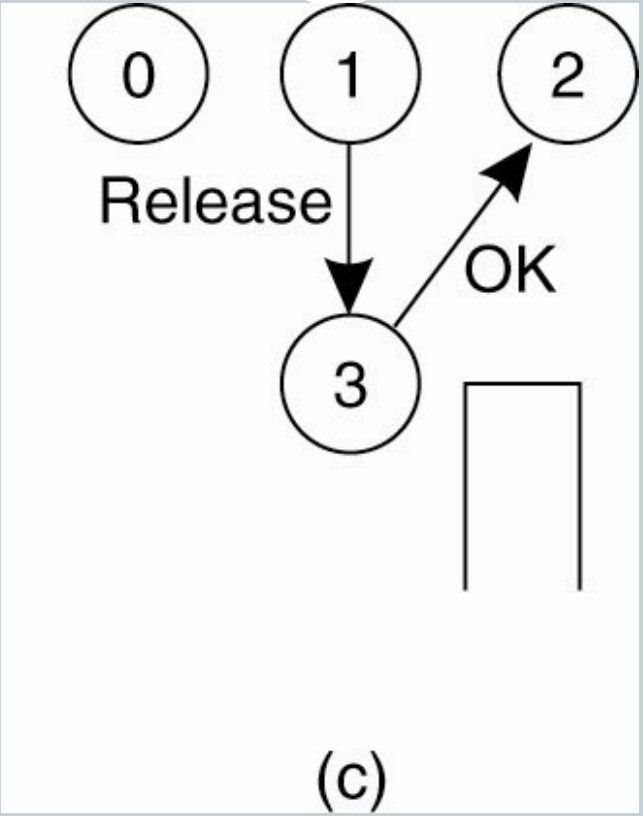


Figure . (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

- It is easy to implement & requires only three messages per resource(Request, Grant, Release)
- Drawback:

  1. Coordinator is a single point of failure.

  2. Single Coordinator may become performance bottleneck

A Distributed Algorithm

- When process wants to access shared resource ,it sends message to all processes in a group.
- Message consist of name of resource, process no & current time.

A Distributed Algorithm

Three different cases:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

A Distributed Algorithm

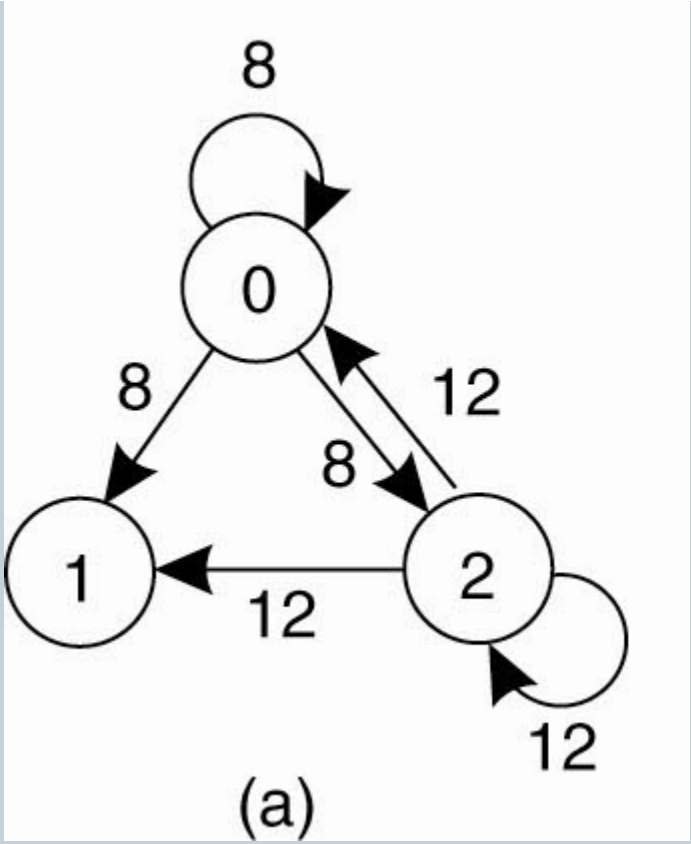


Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

A Distributed Algorithm (3)

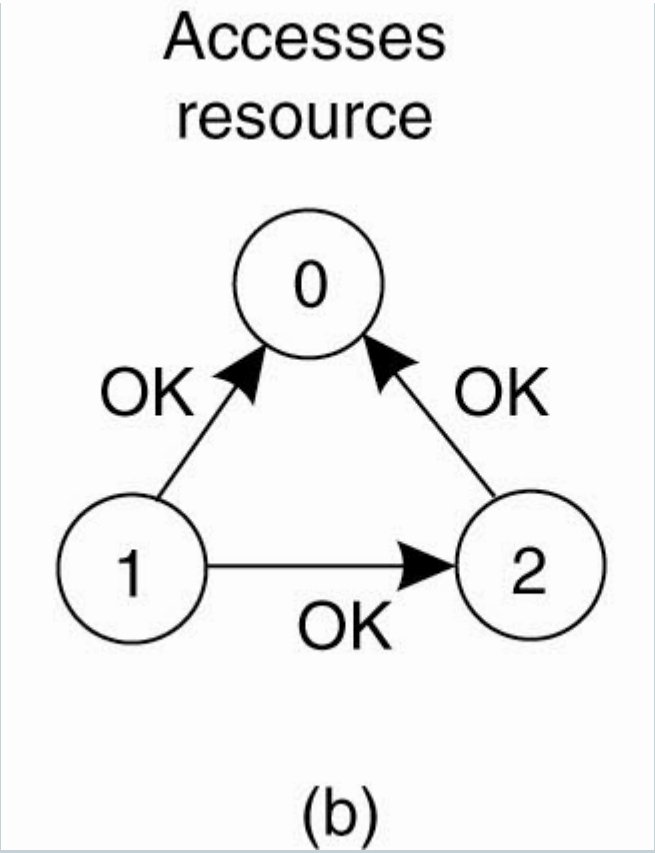


Figure 6-15. (b) Process 0 has the lowest timestamp, so it wins.

A Token Ring Algorithm

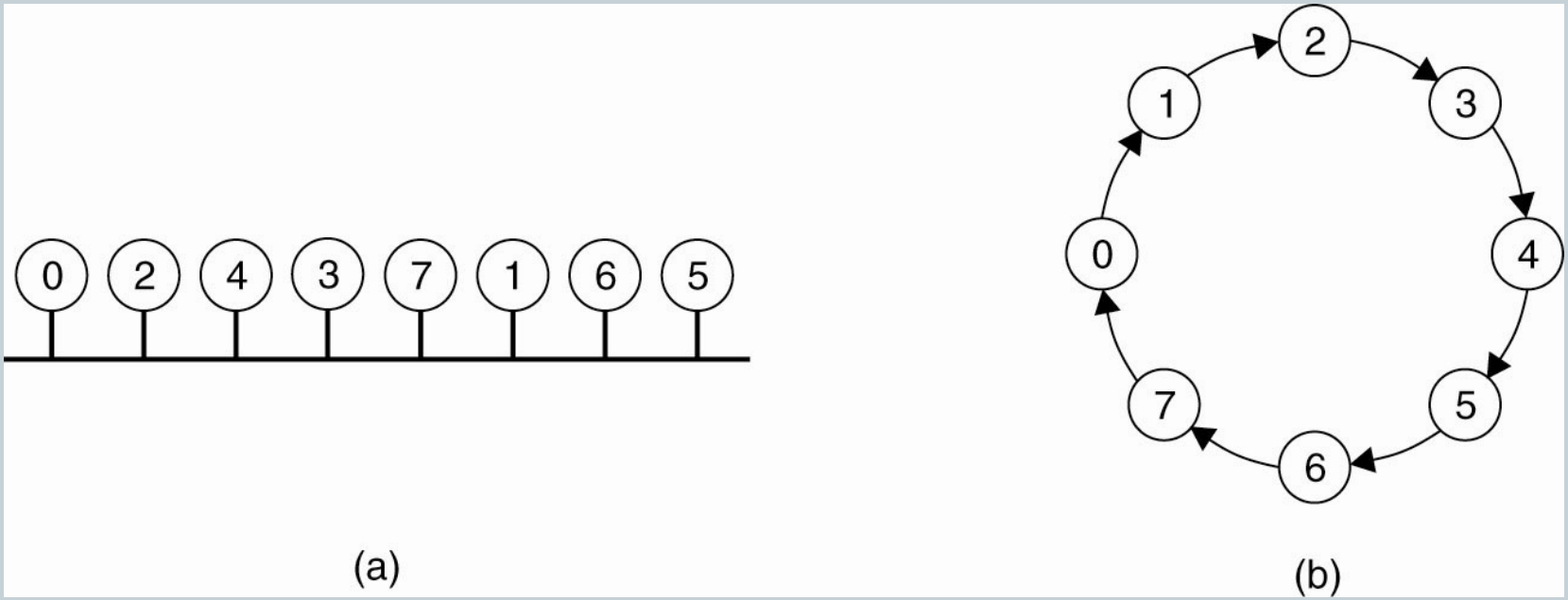


Figure 6-16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Decentralized Algorithm

- Use voting
- Assume n replicas and a coordinator per replica
- To acquire lock, need majority vote $m > n/2$ coordinators
  - Non blocking: coordinators returns OK or "no"
- Coordinator crash => forgets previous votes