

ESE JULY 2023

What are the two different ways of sending data when the send primitive is invoked?

When invoking the send primitive for inter-process communication (IPC), there are two primary ways of sending data:

1. Blocking Send:

- **Description:** In a blocking send operation, the sending process is blocked until the receiving process acknowledges the receipt of the data. This means that the sender will wait until the receiver is ready to receive the data before it can proceed with its execution.
- **Use Case Example:** In a client-server architecture, if a client sends a request to the server using a blocking send, it will wait until the server processes the request and acknowledges receipt before continuing.

2. Non-blocking Send:

- **Description:** In a non-blocking send operation, the sending process continues its execution without waiting for the receiver to acknowledge or process the data. The send operation returns immediately, allowing the sender to proceed with other tasks while the data is transmitted in the background.
- **Use Case Example:** In scenarios where responsiveness is crucial, such as real-time systems, a non-blocking send allows a process to send data and continue its work without being delayed by the transmission process.

Define the following blocking/non-blocking and synchronous/asynchronous primitives:

a. Synchronous primitives b. Asynchronous primitives C. Blocking

primitives d. Non-blocking primitives

a. Synchronous Primitives:

- **Definition:** Synchronous primitives refer to operations that occur in a predetermined order, and the caller typically waits for the operation to complete before proceeding. In a synchronous operation, the flow of control is predictable and sequential.
- **Example:** A synchronous function call, where the calling process waits for the called function to complete before continuing its execution.

b. Asynchronous Primitives:

- **Definition:** Asynchronous primitives involve operations that can initiate and complete independently of the caller's execution. The caller doesn't necessarily wait for the operation to finish but may be notified later through callbacks or events.
- **Example:** Asynchronous messaging, where a process can send a message and continue its work without waiting for an immediate response.

c. Blocking Primitives:

- **Definition:** Blocking primitives are operations that cause the calling process to wait until the operation completes. The flow of control is paused, and the process remains in a blocked state until the requested action is finished.
- **Example:** A blocking I/O operation, where a process waits for data from a file or network before proceeding.

d. Non-blocking Primitives:

- **Definition:** Non-blocking primitives allow the calling process to continue its execution without waiting for the operation to complete. The operation may proceed in the background, and the caller can check later for completion or results.
- **Example:** Non-blocking I/O, where a process initiates an I/O operation and continues with other tasks without waiting for the operation to finish, checking for completion later.

What is a distributed program? What areThe different models of process communication? Explain each one in details.

Distributed Program: A distributed program is a type of computer program designed to run on multiple interconnected computers within a network. These programs are designed to achieve a common goal by dividing tasks among different machines, and they typically communicate and coordinate with each other to accomplish their objectives. Distributed programming is essential for building scalable, fault-tolerant, and efficient systems.

Models of Process Communication in Distributed Systems:

1. Message Passing:

- **Description:** In this model, processes communicate by exchanging messages. Messages can contain data, instructions, or requests. Communication can be either synchronous or asynchronous. Synchronous message passing involves the sender waiting for a response, while asynchronous message passing allows the sender to continue its work without waiting.
- **Example:** Remote Procedure Call (RPC) is a form of synchronous message passing, while message queues represent asynchronous message passing.

2. Shared Memory:

- **Description:** In shared memory communication, processes communicate by reading from and writing to shared regions of memory. Multiple processes can access and modify the shared data, and proper synchronization mechanisms (e.g., locks) are required to avoid conflicts.
- **Example:** Multithreading within a single program, where threads share a common address space.

3. Remote Procedure Call (RPC):

- **Description:** RPC is a model that allows a program to cause a procedure (subroutine) to execute in another address space (commonly on a remote machine) as if it were a local procedure call, hiding the complexities of the distributed nature of the system.
- **Example:** Java RMI (Remote Method Invocation) and gRPC (Google Remote Procedure Call) are examples of RPC frameworks.

4. Socket Programming:

- **Description:** Processes communicate over a network using sockets, which provide an endpoint for sending or receiving data. Sockets can be either connection-oriented (e.g., TCP) or connectionless (e.g., UDP).
- **Example:** Web applications use socket programming for communication between a client's browser and a server using protocols like HTTP over TCP.

5. Publish-Subscribe Model:

- **Description:** In the publish-subscribe model, processes subscribe to specific events or messages. When an event occurs, the publisher notifies all subscribers without them explicitly requesting the information. This is often used in event-driven architectures.
- **Example:** MQTT (Message Queuing Telemetry Transport) is a popular protocol for publish-subscribe communication in IoT applications.

6. Distributed Objects:

- **Description:** Distributed objects involve the exchange of objects between processes in a distributed system. Each process has a local representation of the object, and method invocations on the local object can result in remote procedure calls.
- **Example:** Common Object Request Broker Architecture (CORBA) and Java RMI provide frameworks for distributed object communication.

What is Group communication? Give any two key applications of group communication in distributed systems.

Group Communication: Group communication in distributed systems involves the coordination and information exchange among a predefined set of processes, known as a group. These processes collaborate to achieve common goals, share information, and ensure consistency across the distributed environment. Group communication is essential for building reliable and fault-tolerant distributed systems.

Two Key Applications of Group Communication in Distributed Systems:

1. Replicated Services:

- **Description:** Replicated services involve having multiple instances (replicas) of a service or application running on different nodes within a distributed system. Group communication mechanisms ensure that these replicas stay synchronized by allowing them to exchange messages and coordinate their actions.
- **Key Application Benefits:**
 - **Fault Tolerance:** If one replica fails due to a crash or other issues, the remaining replicas can continue to provide the service without interruption. This enhances system reliability.
 - **Load Balancing:** Group communication enables load balancing by distributing client requests among the available replicas. This ensures efficient utilization of resources.
- **Example Scenario:** Consider a web service replicated across multiple servers. Group communication allows these servers to share updates, ensuring that they provide consistent responses to client requests.

2. Consensus Algorithms:

- **Description:** Consensus is a fundamental problem in distributed computing where a group of processes must agree on a common decision, even in the presence of failures. Group communication plays a crucial role in facilitating the exchange of messages among processes to reach a consensus on a particular value or outcome.
- **Key Application Benefits:**
 - **Distributed Decision Making:** Consensus algorithms enable distributed processes to agree on a common decision, such as the value of a variable or the order of events, despite the possibility of individual process failures.
 - **Atomic Commitment:** In the context of distributed transactions, group communication ensures that all processes involved agree to either commit or abort the transaction, maintaining consistency across the system.
- **Example Scenario:** Imagine a distributed database system where nodes need to agree on whether to commit or abort a transaction.

Group communication ensures that all nodes reach a consensus on the outcome, preventing inconsistencies in the database.

Multicast Communication:

- **Description:** Multicast communication involves the efficient dissemination of information from one sender to multiple receivers within a group. It enables a single sender to send a message to a selected subset of processes, often referred to as a multicast group.
- **Key Application Benefits:**
 - **Scalable Communication:** Multicast reduces the network load compared to unicast (point-to-point) communication, making it more scalable for scenarios with many receivers.
 - **Efficient Broadcasting:** Ideal for scenarios where information needs to be broadcasted to a specific audience rather than the entire network.
- **Example Scenario:** In a distributed online gaming application, the game server could use multicast communication to send updates about game events (e.g., player movements, scores) to all players within a specific game session. This optimizes the use of network resources and ensures that only relevant players receive the updates.

What are the two criteria must be met by a causal ordering

Protocol? Explain with proper justification

A causal ordering protocol in distributed systems aims to establish a consistent and causal relationship among events or messages. Two key criteria must be met by a causal ordering protocol:

1. Partial Ordering of Events:

- **Explanation:** Events in a distributed system need to be partially ordered based on their causal relationship. This means that if event A causally precedes event B (i.e., A influences or triggers B), then A should be ordered before B in the causal order. However, events that are concurrent or independent should not be strictly ordered with respect to each other.
- **Justification:** Partial ordering captures the causality among events while allowing for concurrency, reflecting the fact that events may occur independently and should not be strictly ordered unless there is a causal relationship.

2. Consistent Global View:

- **Explanation:** The protocol should ensure that all processes in the distributed system have a consistent global view of the events. In other

- words, the partial ordering of events should be agreed upon by all processes, providing a unified perspective on the causal relationships.
- **Justification:** Consistency in the global view is crucial to avoid inconsistencies or misunderstandings among processes. All processes should agree on the causal relationships among events to maintain a coherent understanding of the system's behavior.

Example Justification: Consider a distributed system with multiple processes exchanging messages. If process P1 sends a message M1, and process P2 receives M1 and subsequently sends message M2, there is a causal relationship between M1 and M2. A causal ordering protocol should ensure that all processes agree that M1 causally precedes M2. However, if another process, P3, sends a message M3 concurrently with M1, these two messages may be considered concurrent in the causal order since there is no causal relationship between them.

Write a bully Election algorithms and explain it with the Help example.

The Bully Election Algorithm is a leader election algorithm used in distributed systems where processes need to elect a coordinator or leader. The algorithm ensures that the process with the highest priority becomes the leader. If the current leader fails, other processes can initiate a new election to choose a new leader.

Bully Election Algorithm:

1. Initiation:

- Any process can initiate an election by sending an Election message to all processes with higher priorities.

2. Election Message:

- Upon receiving an Election message, a process checks its own priority. If it has a higher priority, it responds with an Answer message and starts a sub-election.

3. Sub-Election:

- The process that initiates the sub-election sends Election messages to processes with higher priorities. If no higher-priority process responds, it becomes the coordinator.

4. Coordinator Status:

- The coordinator periodically sends a KeepAlive message to inform other processes about its status. If a process doesn't receive KeepAlive messages from the coordinator, it assumes the coordinator has failed and can initiate a new election.

Example: Let's consider a scenario with five processes (P1 to P5) and their respective priorities. Higher priority values indicate higher priority.

- P1 (Priority 5)
- P2 (Priority 3)
- P3 (Priority 8)
- P4 (Priority 2)
- P5 (Priority 6)

Execution:

1. Process P3 detects the failure of the current coordinator or decides to initiate an election.
2. P3 sends Election messages to P1, P2, P4, and P5 since they have higher priorities.
3. P1, P2, and P5 respond with an Answer message, indicating they have higher priorities.
4. P3 realizes that P5 has the highest priority among the respondents, so it gracefully steps down.
5. P5 becomes the coordinator and sends KeepAlive messages to inform others about its status.

If the current coordinator (P5) fails, any process that doesn't receive KeepAlive messages from P5 can initiate a new election, and the process with the highest priority among the remaining ones will become the new coordinator.

The Bully Election Algorithm ensures that the process with the highest priority is elected as the coordinator, and the system remains resilient to coordinator failures.

Explain the Lamport's distributed mutual exclusion algorithm in terms of

a. Requesting the critical section

b. Executing the critical section and

C. Releasing the critical section

The Lamport's Distributed Mutual Exclusion Algorithm is designed to ensure that multiple processes in a distributed system can access a critical section without interference. Let's break down the algorithm into the requested components:

a. **Requesting the Critical Section:**

- When a process wants to enter the critical section, it sends a request message to all other processes, including a timestamp representing its logical clock value.
- The logical clock ensures that each request is assigned a unique timestamp and helps in determining the order of requests.

b. **Executing the Critical Section:**

- Upon receiving a request, a process compares the timestamp of the incoming request with its own. The process with the lowest timestamp gets priority.
- If there's a tie in timestamps, the process with the lower process ID is given priority.
- Once a process wins the election, it can enter the critical section and perform its operations.

c. **Releasing the Critical Section:**

- After a process finishes executing the critical section, it broadcasts release messages to inform other processes that it's done.
- This allows other processes to update their information and possibly enter the critical section based on their requests and priorities.

Example: Let's consider three processes, P1, P2, and P3. P1 wants to enter the critical section.

- a. P1 sends a request message with a timestamp to P2 and P3.
- b. P2 and P3 receive the request and compare timestamps. Suppose P2 has a lower timestamp; it grants permission to P1. P1 enters the critical section.
- c. After P1 completes its critical section execution, it sends release messages to P2 and P3, allowing them to potentially enter the critical section based on their requests and timestamps.

What are the different model of deadlocks? Explain any three models.

Deadlocks are situations where a set of processes are blocked because each process is holding a resource and waiting for another resource occupied by some other process. There are different models of deadlocks, depending on how the resources and processes are represented and managed. Here are three common models of deadlocks:

- Single-resource model: In this model, each process has a unique identifier and can request one resource at a time. The system maintains a resource allocation graph, where each node represents a resource and each edge represents a request or release operation. [A deadlock occurs when there is a circular dependency among the requests, meaning that each process is waiting for a resource that is held by another process in the cycle¹.](#)
- Multiple-resource model: In this model, each process has a unique identifier and can request multiple resources at once. The system maintains a set of resources and a set of processes, where each resource has an identifier and each process has an identifier. [A deadlock occurs when there is a circular dependency among the requests, meaning that each process is waiting for one or more resources that are held by other processes in the cycle¹.](#)
- Communication model: In this model, each process has a unique identifier and can send or receive messages from other processes. The system maintains a message queue for each process, where messages are stored until they are delivered or discarded. [A deadlock occurs when there is no message in transit between any pair of processes in the set, meaning that all processes are waiting for messages from other processes in the set².](#)

I hope this helps you understand the different models of deadlocks. If you have any further questions, please feel free to ask me 😊 ...

What are the types of data indexing in P2P networking[^] Explain with example

Data indexing is a technique for organizing and locating data in a distributed system, such as a peer-to-peer (P2P) network. There are different types of data indexing, depending on how the data is mapped to the nodes or peers in the network, and how the queries are handled. Here are some examples of data indexing in P2P networking:

- Centralized indexing: This type of indexing uses one or a few central servers to store references or indexes to the data on many peers. For example, the Domain Name System (DNS) is a centralized index that maps domain names to IP addresses. [Some early P2P networks, such as Napster, also used a central directory lookup to find files shared by other peers¹.](#)

- Local indexing: This type of indexing indexes the data at various peers being scattered across other peers throughout the P2P network. To access the indexes, a structure is used in the P2P overlay to access the indexes. For example, Chord is a distributed hash table (DHT) that maps keys to values using a hash function and stores them in multiple nodes¹.
- Distributed indexing: This type of indexing requires each peer to index only its local data objects and remote objects need to be searched for. This form of indexing is typically used in unstructured overlays in conjunction with flooding search or random walk search. For example, Pastry is another DHT that supports complex queries using relational operators and text retrieval techniques¹.

I hope this helps you understand the types of data indexing in P2P networking. If you have any further questions, please feel free to ask me 😊.

What are the different distributed computing models? Explain with suitable diagram.

<https://www.ques10.com/p/2108/explain-distributed-computing-models-with-diagra-1/>

What is the global state of the distributed system? How to design the efficient method to design recording the global state of the distributed system?

<https://www.cs.uic.edu/~ajayk/Chapter4.pdf>

The global state of a distributed system is a collection of the local states of the processes and the channels. A global state computed along a consistent cut is correct. The global state of a consistent cut comprises the local state of each process at the time the cut event happens and the set of all messages sent but not yet received¹.

To design an efficient method to record the global state of a distributed system, one possible approach is to use Chandy-Lamport's global state recording algorithm². This algorithm uses a special message called MARKER to trigger each process to

record its own state and the messages it sends and receives. The MARKER travels across all communication channels and causes each process to update its local state accordingly. The final global state is obtained by concatenating all local states recorded by each process.

[Another possible approach is to use DPREU or DRRU algorithms³](#), which are based on dynamic programming and reduce the number of messages exchanged between processes. These algorithms use a data structure called DP-Tree, which stores information about each process and its neighbors in a hierarchical manner. The DP-Tree allows processes to query their neighbors without sending any messages, thus saving communication overhead. The final global state is obtained by traversing the DP-Tree from root to leaf.

I hope this helps you understand how to design an efficient method to record the global state of a distributed system. If you have any further questions, please feel free to ask me 😊...

Construct with neat diagrams and give example of different forms of

communication, such as persistent asynchronous, persistent synchronous,

transient asynchronous, receipt based transient synchronous, delivery

based transient synchronous, and response based transient synchronous

communication

1. **Persistent Asynchronous Communication:**

- **Description:** In persistent asynchronous communication, the sender does not wait for an immediate response from the receiver. The message is sent, and the sender continues its execution without blocking.
- **Example:** Sending emails is an example of persistent asynchronous communication. The sender sends an email, and the system doesn't wait for an immediate response.

2. Persistent Synchronous Communication:

- **Description:** In persistent synchronous communication, the sender waits for an immediate response from the receiver before proceeding with further actions.
- **Example:** Making a synchronous HTTP request in a web application is an example. The client sends a request, waits for the server's response, and then continues based on the received response.

3. Transient Asynchronous Communication:

- **Description:** Transient asynchronous communication involves the sender not waiting for an immediate response. However, there might be mechanisms for later acknowledgment or notification of delivery.
- **Example:** Publishing a message to a message queue (e.g., using RabbitMQ or Apache Kafka) is an example. The sender publishes a message and doesn't wait for subscribers to receive it immediately.

4. Receipt-Based Transient Synchronous Communication:

- **Description:** In receipt-based transient synchronous communication, the sender waits for an acknowledgment (receipt) from the receiver to confirm the successful delivery of the message.
- **Example:** Sending a text message with delivery receipts enabled. The sender waits for a receipt to confirm that the message has been delivered to the recipient's device.

5. Delivery-Based Transient Synchronous Communication:

- **Description:** In delivery-based transient synchronous communication, the sender waits for the actual delivery of the message to the recipient.
- **Example:** Sending a letter through postal mail. The sender waits for the physical delivery of the letter to the recipient.

6. Response-Based Transient Synchronous Communication:

- **Description:** In response-based transient synchronous communication, the sender waits for a response or reply from the receiver before proceeding.
- **Example:** Making a synchronous Remote Procedure Call (RPC). The client sends a request to the server, waits for the server to process and respond before continuing its execution.

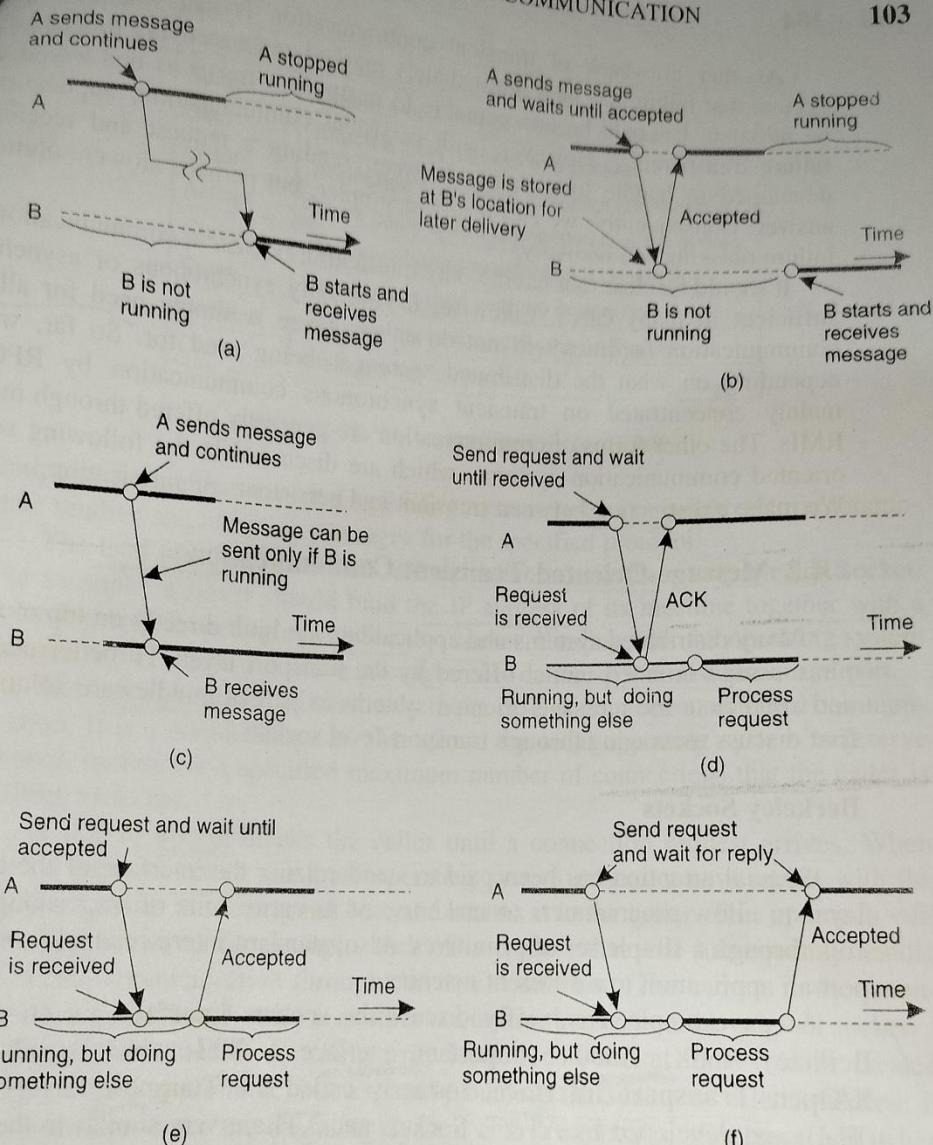


Figure 2-22. Six different forms of communication: (a) persistent asynchronous communication, (b) persistent synchronous communication, (c) transient asynchronous communication, (d) receipt-based transient synchronous communication, (e) delivery-based transient synchronous communication at message delivery, and (f) response-based transient synchronous communication.

always be immediately accessible.] For example, access can be restricted due to network or process failures. To solve such problems, proprietary solutions for persistent communication have been developed, but such solutions are not very satisfactory in light of interoperability and portability, as can easily be imagined.

Why is it difficult to keep the synchronized system of physical clocks in

distributed system? Justify your answer with appropriate examples.

Keeping a synchronized system of physical clocks in a distributed system is challenging due to various factors, including network delays, clock drift, and the asynchronous nature of distributed communication. Let's explore these challenges with examples:

1. Network Delays:

- **Challenge:** The time it takes for a message to travel between nodes in a distributed system is variable and influenced by network delays. These delays can be unpredictable and may lead to discrepancies in clock synchronization.
- **Example:** Consider two nodes, A and B, in a distributed system. If A sends a synchronization message to B, the time it takes for the message to reach B will be influenced by factors like network congestion and routing changes. This variability in transit time introduces uncertainty in clock synchronization.

2. Clock Drift:

- **Challenge:** Physical clocks on different nodes may have different clock rates due to hardware variations, temperature changes, or other factors. Over time, these differences can result in clock drift, causing clocks to deviate from the true time.
- **Example:** Two servers with slightly different clock rates will gradually drift apart. Even if they are initially synchronized, the difference in clock rates will lead to a growing time gap between them.

3. Asynchronous Communication:

- **Challenge:** Asynchronous communication, where processes operate independently and exchange messages without waiting for each other, introduces challenges in maintaining synchronized clocks. Processes may not be aware of the time it takes for a message to be transmitted, making it difficult to establish accurate synchronization.
- **Example:** In a messaging system, if Process A sends a message to Process B and continues its execution without waiting for a response, the clocks of A and B may not be perfectly synchronized due to the asynchronous nature of the communication.

4. Faults and Failures:

- **Challenge:** Distributed systems are susceptible to faults and failures, and maintaining synchronized clocks becomes more challenging when nodes can become unavailable or experience transient issues.
- **Example:** If a node in a distributed system fails and later recovers, its clock may have drifted during the downtime. Resynchronizing the

recovered node with the rest of the system can be challenging, especially if the failure duration is significant.

5. Uncertainty in Timestamps:

- **Challenge:** Assigning accurate timestamps to events in a distributed system is challenging due to the uncertainty introduced by network delays and clock drift. This uncertainty affects the consistency and reliability of timestamps.
- **Example:** In a distributed database, if events are timestamped based on local clocks, querying the database using these timestamps may yield inconsistent or inaccurate results due to variations in clock synchronization.

Hello, this is Bing. I can help you with your question about why it is difficult to keep the synchronized system of physical clocks in distributed system. Here is a summary of what I found:

- Physical clock synchronization is the process of aligning the clocks of computers or nodes in a distributed system to a common reference point, such as Universal Coordinated Time (UTC)¹.
- It is difficult to keep the synchronized system of physical clocks in distributed system because of several challenges, such as¹²:
 - Information dispersion: Distributed systems store information on machines, which may have different clock settings and sources. Gathering and harmonizing this information to achieve synchronization presents a challenge.
 - Local decision realm: Distributed systems rely on localized data for making decisions. As a result, when it comes to synchronization, we have to make decisions with information from each node, which makes the process more complex.
 - Mitigating failures: In a distributed environment, it becomes crucial to prevent failures in one node from causing disruption in synchronization. If one node fails or crashes, the other nodes may not be able to communicate or coordinate with it.
 - Temporal uncertainty: The existence of clocks in distributed systems creates the potential for time variations. Different nodes may have different clock drifts and skew, which can affect the accuracy and consistency of data exchange and task execution.

Some examples of difficulties that arise when synchronizing clocks in distributed systems are³:

- Getting two systems agree upon time: Two clocks hardly ever agree. Quartz oscillators oscillate at slightly different frequencies. This creates an ever-widening gap in perceived time. This is called as clock drift. At one point of time, the difference between the two clocks is called as clock skew.
- Synchronizing multiple nodes: If there are multiple nodes involved in a distributed system, they may have different clock settings and sources. This makes it harder to achieve consensus and coordination among them.
- Handling network delays: Network delays can affect the speed and reliability of communication between nodes. If there are network failures or congestion, the messages may not reach their destination or may arrive out of order.

I hope this helps you understand why it is difficult to keep the synchronized system of physical clocks in distributed system. If you have any other questions, please feel free to ask me 😊

Explain the Ricart Agrawala algorithm with respect to requesting the

critical section, executing the critical section and releasing the critical

section. Give the correctness of Ricart Agrawala algorithm for achieving

mutual exclusion.

The Ricart-Agrawala algorithm is a distributed mutual exclusion algorithm designed to allow processes in a distributed system to request, enter, and release a critical section in a coordinated and mutually exclusive manner. The algorithm was proposed by Glenn Ricart and Ashok Agrawala. Here's an explanation of the three key phases: requesting, executing, and releasing the critical section:

Requesting the Critical Section:

- 1. When a process P_i wants to enter the critical section:**
 - P_i sends a request message to all other processes, indicating its intention to enter the critical section.
 - The request message includes a timestamp T_i indicating the local time of P_i when the request is made.

- 2. Upon receiving a request from P_i :**
 - A process P_j compares the timestamp T_i in the request with its own timestamp T_j and replies in one of the following ways:
 - If P_j is not interested in the critical section (not requesting or currently executing), it sends an acknowledgment directly to P_i .
 - If P_j has a pending request or is currently in the critical section, it compares timestamps:
 - If T_i is smaller than T_j , P_j sends an acknowledgment.
 - If T_i is greater than T_j , P_j defers its acknowledgment until it exits the critical section.

Executing the Critical Section:

- 1. After receiving acknowledgments from all other processes:**

- P_i enters the critical section for execution.

Releasing the Critical Section:

- 1. Upon exiting the critical section:**

- P_i sends a release message to all processes, indicating that it has exited the critical section.

- 2. Upon receiving a release message from P_i :**

- A process P_j updates its timestamp to $\max(T_j, T_i) + 1$ to ensure causality and consistency.

Correctness of Ricart-Agrawala Algorithm:

The Ricart-Agrawala algorithm ensures mutual exclusion, meaning that at most one process is in the critical section at any given time. The correctness is guaranteed by the following properties:

1. No Two Processes in the Critical Section Simultaneously:

- If a process P_i is in the critical section, it has received acknowledgments from all other processes, indicating that none of them are interested in the critical section or are currently executing it.

2. Timestamp Ordering:

- The use of timestamps ensures that processes requesting the critical section are granted access in a globally agreed-upon order. Timestamps help in establishing a consistent total order among the requests.



3. Starvation-Free:

- Every process that requests the critical section eventually gets access to it. Starvation is prevented by the ordering of timestamps, which ensures fairness in process execution.

While Ricart-Agrawala provides mutual exclusion and fairness, it has drawbacks, such as potential message overhead and the need for synchronized clocks. Additionally, it assumes that communication channels are reliable, which may not always be the case in real-world distributed systems.

The Ricart-Agrawala algorithm is an optimization of Lamport's Distributed Mutual Exclusion Algorithm and follows a permission-based approach to ensure mutual exclusion in a distributed system¹². Here's how it works:

Requesting the Critical Section: When a site s_i wants to enter the critical section, it sends a timestamped REQUEST message to all other sites¹². When a site

s_j receives a REQUEST message from site s_i, it sends a REPLY message to site s_i if and only if¹²:

- Site s_j is neither requesting nor currently executing the critical section.
- In case Site s_j is requesting, the timestamp of Site s_i's request is smaller than its own request¹².

Executing the Critical Section: Site s_i enters the critical section if it has received the REPLY message from all other sites¹².

Releasing the Critical Section: Upon exiting, site s_i sends REPLY messages to all the deferred requests¹².

Correctness of Ricart Agrawala Algorithm for Mutual Exclusion: The Ricart-Agrawala algorithm achieves mutual exclusion. The proof is by contradiction². Suppose two sites s_i and s_j are executing the critical section concurrently and s_i's request has higher priority than the request of s_j. Clearly, s_i received s_j's request after it has made its own request. Thus, s_j can concurrently execute the critical section with s_i only if s_i returns a REPLY to s_j (in response to s_j's request) before s_i exits the critical section. However, this is impossible because s_j's request has lower priority². Therefore, the Ricart-Agrawala algorithm achieves mutual exclusion².

5.5.2 A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method.

Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first. Lamport's algorithm presented in

Sec. 5.2.1 is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 5-14(a).

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 5-14(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig. 5-14(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig. 5-14 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and

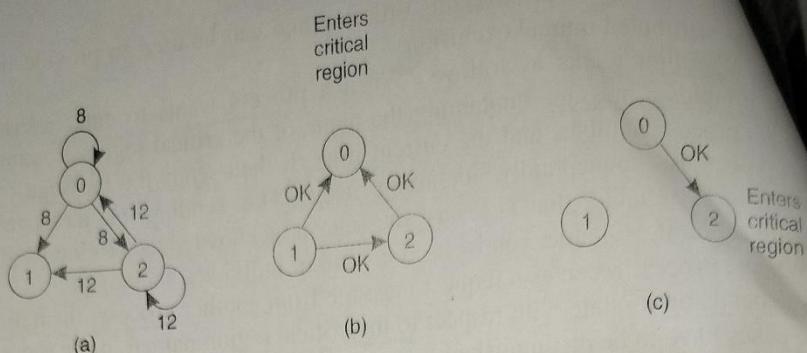


Figure 5-14. (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now $\frac{2}{n}(n - 1)$, where the total number of processes in the system is n . Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.

The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If

one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more intricate.

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust than the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

What is deadlock in distributed systems? Explain the Knaps classification of distributed deadlock detection algorithm.

A deadlock in a distributed system is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource occupied by some other process¹. Deadlocks in distributed systems are similar to deadlocks in centralized systems, but handling them in distributed systems is more complex because the resources, the processes, and other relevant information are scattered on different nodes of the system¹.

Knapp's classification categorizes distributed deadlock detection algorithms into four classes²³:

1. **Path-pushing algorithms:** These algorithms detect deadlocks by maintaining an explicit global wait-for graph (WFG). Each site sends its local WFG to all neighboring sites. The procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present²³.

- o Examples: Menasce-Muntz, Gligor and Shattuck, Ho and Ramamoorthy, Obermarck²³.

2. **Edge-chasing algorithms:** These algorithms verify the presence of a cycle in a distributed graph structure by propagating special messages called probes, along the edges of the graph. The formation of a cycle can be detected by a site if it receives the matching probe sent by it previously²³.
 - Examples: Chandy et al., Choudhary et al., Kshemkalyani–Singhal, Sinha–Natarajan algorithms²³.
3. **Diffusion computation-based algorithms:** These algorithms use echo algorithms to detect deadlocks. The deadlock detection computation is diffused through the WFG of the system. If this computation terminates, the initiator declares a deadlock²³.
 - Examples: Chandy–Misra–Haas algorithm for one OR model, Chandy–Herman algorithm²³.
4. **Global state detection-based algorithms:** These algorithms exploit the fact that a consistent snapshot of a distributed system can be obtained without freezing the underlying computation²³.

I hope this helps you understand deadlock in distributed systems and Knapp's classification of distributed deadlock detection algorithms. If you have any other questions, please feel free to ask me 😊.

Explain the models of deadlocks on distributed systems?

In distributed systems, deadlocks can occur when multiple processes or nodes are unable to proceed because each is holding resources while waiting for others to release the resources they need. Deadlock models in distributed systems help us understand the conditions under which deadlocks may arise. Here are three common models:

1. Resource Allocation Graph Model:

- **Description:** The Resource Allocation Graph is a graphical representation of the allocation and request of resources in a distributed system. Nodes represent processes, and edges represent resource requests or allocations. This model is based on the conditions necessary for a deadlock: mutual exclusion, hold and wait, no preemption, and circular wait.
- **Key Conditions:**
 - Mutual Exclusion: Resources that cannot be shared.
 - Hold and Wait: A process holds a resource and waits for others.
 - No Preemption: Resources cannot be forcibly taken from a process.
 - Circular Wait: There is a circular chain of processes waiting for resources.

- **Example:** If process P1 holds resource R1 and requests resource R2, while process P2 holds resource R2 and requests resource R1, a circular wait occurs, leading to a potential deadlock.

2. Wait-for Graph Model:

- **Description:** The Wait-for Graph model focuses on the relationships between processes waiting for resources. Nodes represent processes, and directed edges represent a waiting relationship. A cycle in the graph indicates a potential deadlock.
- **Key Condition:**
 - Circular Wait: A cycle in the wait-for graph.
- **Example:** If process P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3, and P3 is waiting for a resource held by P1, a circular wait-for graph occurs, signaling a potential deadlock.

3. State Transition Model:

- **Description:** The State Transition Model describes the possible states and transitions of processes in a distributed system. Processes can be in different states such as running, blocked, or terminated, and transitions occur based on events like acquiring or releasing resources.
- **Key Conditions:**
 - Circular Waiting: Processes are in a circular waiting state.
 - No Preemption: Resources cannot be forcibly taken from a process.
- **Example:** Consider two processes, P1 and P2, and two resources, R1 and R2. If P1 holds R1 and requests R2 while P2 holds R2 and requests R1, the system may reach a state where neither process can proceed, leading to a deadlock.

These models help in identifying the conditions and scenarios that may lead to deadlocks in distributed systems. Understanding these models is crucial for designing deadlock prevention, avoidance, and detection mechanisms to ensure the reliability and performance of distributed applications.

What are the challenges in Peer to Peer system design? How to resolve

these challenges using appropriate examples?

Peer-to-Peer (P2P) systems face several challenges due to their decentralized nature¹². Here are some of the key challenges:

1. **Efficiency:** P2P systems need to ensure efficient searching and matching of resources across the network³.
2. **Security:** Providing overall security in a P2P network is difficult as each system is independent and contains its own data⁴.
3. **Data Backup:** It is challenging to backup data as it is stored in different computer systems and there is no central server⁴.
4. **Manageability:** The decentralized nature of P2P systems raises concerns about manageability¹.
5. **Performance Guarantees:** Ensuring performance guarantees in P2P systems is a significant challenge².

To address these challenges, various solutions have been proposed:

1. **Efficient Searching and Matching:** Centralized and decentralized approaches can be used to help buyers and sellers find each other as easily and as fast as possible³. For example, Uber uses a centralized, location-based search approach for efficient matching of drivers and riders³.
2. **Security:** Security can be enhanced by adopting new distributed technologies, such as IPFS and Blockchain, which follow a full P2P model⁵.
3. **Data Backup:** Data replication strategies can be used to ensure data availability and reliability. For instance, in a P2P file sharing system, multiple copies of the same file can be stored at different peers to ensure data availability even if some peers go offline.
4. **Manageability:** Management issues can be addressed by designing proper protocols and algorithms for tasks such as resource discovery, routing, and load balancing.
5. **Performance Guarantees:** Performance can be improved by using efficient algorithms and protocols, and by optimizing resource allocation and task scheduling².

These are just a few examples of the challenges and potential solutions in P2P system design. The specific challenges and solutions can vary depending on the specific requirements and constraints of the P2P system³².

Explain the data indexing and overlays with the help of examples.

Sure, let's start with **data indexing**:

Data indexing is a technique used in distributed systems to organize and locate data efficiently¹². There are two main types of indexing models in distributed systems¹:

1. **Local Indexes:** In this model, each node in the cluster indexes the data it locally holds. This optimizes for fast indexing. However, as the data ingest increases, index maintenance locally competes with the incoming workload, and as the cluster gets larger (more nodes), the scatter-gather hits the query latency¹.
2. **Global Indexes:** In this model, an index is maintained that spans across all nodes in the cluster. This allows for efficient querying as the index provides a global view of the data. However, maintaining and updating a global index can be challenging, especially in large-scale distributed systems¹.

Now, let's move on to **overlays**:

An overlay network is a virtual network that is built on top of another network³. Nodes in an overlay network are connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network³.

For example, distributed systems such as peer-to-peer networks are overlay networks because their nodes form networks over existing network connections³. Another example is the Internet, which was originally built as an overlay upon the telephone network³.

In summary, data indexing and overlays are fundamental concepts in distributed systems that enable efficient data retrieval and network organization, respectively¹³.

What is RMI ?? How to apply parameter passing techniques like passing object by value or by reference in RMI with suitable diagram

Remote Method Invocation (RMI):

RMI, or Remote Method Invocation, is a Java API that facilitates communication between objects in different Java Virtual Machines (JVMs). It enables objects in one JVM to invoke methods on objects residing in another JVM, potentially on a different host. RMI is a powerful tool for building distributed applications where components can seamlessly communicate across a network.

Parameter Passing in RMI:

In RMI, the correct passing of parameters during remote method calls is crucial for the proper functioning of distributed applications. Two main techniques are employed: Passing by Value and Passing by Reference.

1. Passing by Value:

• Explanation:

- When an object is passed by value in RMI, a copy of the object is serialized (converted into a byte stream) and sent to the remote JVM.
- The remote JVM then deserializes the object, reconstructing it on the server side.
- Importantly, any modifications made to the object on the server side do not affect the original object on the client side.

• Use Case:

- This method is suitable for scenarios where you want to transmit the current state of an object but do not want changes on the server side to impact the client's local instance.

2. Passing by Reference:

• Explanation:

- If the object being passed implements the `java.rmi.Remote` interface, it is passed by reference.
- Instead of sending a copy of the object, RMI sends a stub (proxy) for the object to the client.
- The stub acts as a proxy, forwarding any method calls made on it to the actual remote object on the server.

• Use Case:

- This method is beneficial when you want the client to interact with the remote object as if it were a local object. Method calls on the stub are transparently forwarded to the remote object, allowing seamless communication.

Example:

Let's consider an example where we have a `Calculator` object hosted on a remote server, and a client wants to perform addition using RMI:

1. Passing by Value:

- The client sends the values to be added as parameters by value. Copies of the values are sent to the server.
- The server performs the addition and returns the result. Changes to the values on the server do not affect the client.

2. Passing by Reference:

- The client receives a stub for the **Calculator** object, which is a remote reference.
- The client invokes the **add** method on the stub as if it were a local method.
- The stub forwards the method call to the actual **Calculator** object on the server, performs the addition, and returns the result.

2.3.3 Static versus Dynamic Remote Method Invocations ESF

After a client is bound to an object, it can invoke the object's methods through the proxy. Such a **remote method invocation** or simply **RMI**, is very similar to an **RPC** when it comes to issues such as marshaling and parameter passing. An essential difference between an **RMI** and an **RPC**, is that **RMIs** generally support systemwide object references as explained above. Also, it is not necessary to have only general-purpose client-side and server-side stubs available. Instead, we can more easily accommodate object-specific stubs as we also explained.

The usual way to provide **RMI** support is to specify the object's interfaces in an **interface definition language**, similar to the approach followed with **RPCs**. Alternatively, we can make use of an object-based language such as **Java**, that will handle stub generation automatically. This approach of using predefined interface definitions is generally referred to as **static invocation**. Static invocations require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.

As an alternative, method invocations can also be done in a more dynamic fashion. In particular, it is sometimes convenient to be able to compose a method invocation at runtime, also referred to as a **dynamic invocation**. The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object. Dynamic invocation generally takes a form such as

invoke(object, method, input_parameters, output_parameters);

where *object* identifies the distributed object, *method* is a parameter specifying exactly which method should be invoked, *input_parameters* is a data structure that holds the values of that method's input parameters, and *output_parameters* refers to a data structure where output values can be stored.

For example, consider appending an integer *int* to a file object *fobject*, for which the object provides the method *append*. In this case, a static invocation would take the form

fobject.append(int)

whereas the dynamic invocation would look something like

invoke(*fobject*, *id(append)*, *int*)

where the operation *id(append)* returns an identifier for the method *append*

To illustrate the usefulness of dynamic invocations, consider an object browser that is used to examine sets of objects. Assume that the browser supports remote object invocations. Such a browser is capable of binding to a distributed object and subsequently presenting the object's interface to its user. The user could then be asked to choose a method and provide values for its parameters, after which the browser can do the actual invocation. Typically, such an object browser should be developed to support any possible interface. Such an approach requires that interfaces can be inspected at runtime, and that method invocations can be dynamically constructed.

Another application of dynamic invocations is a batch processing service to which invocation requests can be handed along with a time when the invocation should be done. The service can be implemented by a queue of invocation requests, ordered by the time that invocations are to be done. The main loop of the service would simply wait until the next invocation is scheduled, remove the request from the queue, and call *invoke* as given above.

2.3.4 Parameter Passing ES E

Because most RMI systems support systemwide object references, passing parameters in method invocations is generally less restricted than in the case of RPCs. However, there are some subtleties that can make RMIs trickier than one would initially expect, as we briefly discuss in the following pages.

Let us first consider the situation that there are only distributed objects. In other words, all objects in the system can be accessed from remote machines. In that case, we can consistently use object references as parameters in method invocations. References are passed by value, and thus copied from one machine to another. When a process is given an object reference as the result of a method invocation, it can simply bind to the object referred to when needed later.

Unfortunately, using only distributed objects can be highly inefficient, especially when objects are small, such as integers and Booleans. Each invocation by a client that is not colocated in the same server as the object, generates a request between different address spaces or, even worse, between different machines. Therefore, references to remote objects and those to local objects are often treated differently.

When invoking a method with an object reference as parameter, that reference is copied and passed as a value parameter only when it refers to a remote object. In this case, the object is literally passed by reference. However, when the reference refers to a local object, that is an object in the same address space as the client, the referred object is copied as a whole and passed along with the invocation. In other words, the object is passed by value.

These two situations are illustrated in Fig. 2-18, which shows a client program running on machine A, and a server program on machine C. The client has a reference to a local object O_1 that it uses as a parameter when calling the server program on machine C. In addition, it holds a reference to a remote object O_2 residing at machine B, which is also used as a parameter. When calling the server, a copy of O_1 is passed to the server on machine C, along with only a copy of the reference to O_2 .

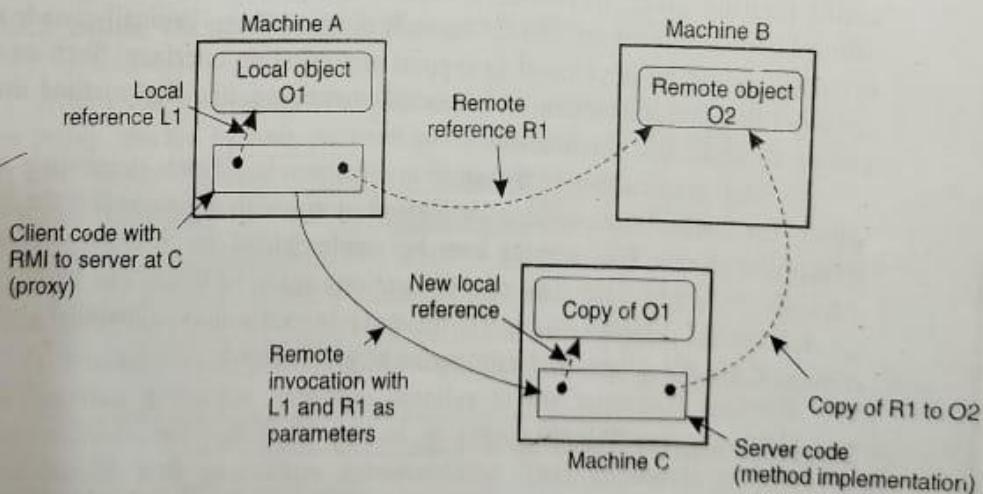


Figure 2-18. The situation when passing an object by reference or by value.

Note that whether we are dealing with a reference to a local object or to a remote object can be highly transparent, such as in Java. In Java, the distinction is

visible only because local objects are essentially of a different data type than remote objects. Otherwise, both types of references are treated very much the same (see also Wollrath et al., 1996). On the other hand, when using conventional programming languages such as C, a reference to a local object can be as simple as a pointer, which can never be used to refer to a remote object.

The side effect of invoking a method with an object reference as parameter is that we may be *copying* an object. Hiding this aspect is unacceptable, so that we are consequently forced to make an explicit distinction between local and distributed objects. Clearly, this distinction not only violates distribution transparency, but also makes it harder to write distributed applications.

A client makes the remote procedure calls to server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local operating system processing time for each send or receive operation is 0.5 milliseconds, and network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per message. Calculate the time taken by the client to generate and return from two requests: i) if it is single threaded, and ii) if it has two threads that can make requests concurrently on single processor. You can ignore context switching time.

Construct with neat diagrams and give example of different forms of communication, such as persistent asynchronous , persistent synchronous, transient asynchronous, receipt based transient synchronous , delivery based transient synchronous and response based transient synchronous communication.

What are different reasons for code migration explain in detail. Explain with diagram how client dynamically configure the server

Code migration is the process of moving programming code from one system to another¹. It can involve moving from one version of a language to a newer version, moving to a different programming language, or migrating to an entirely new platform or operating system¹. Here are some reasons for code migration:

1. **Performance Improvement:** Modern programming languages often have better efficiency than legacy languages². Therefore, migrating code can lead to more efficient execution¹².
2. **Load Sharing in Distributed Systems:** Long-running processes can be migrated to idle processors to improve system-wide performance¹³.
3. **Client-Server Systems:** If large quantities of data need to be processed, it is better to ship the data processing component to the client³. This leads to more efficiency in execution¹³.
4. **Software System Changes:** Migrating to a completely different programming language could be caused by porting to a new software system or implementing a different relational database management system (RDMS)¹.
5. **Platform or Operating System Changes:** Migrating to an entirely new platform and/or operating system (OS) changes not only the programming language, but also the machine code behind the language¹.

As for how a client dynamically configures a server, one common method is through the Dynamic Host Configuration Protocol (DHCP)⁴⁵. Here's a simplified sequence of events:

1. When a client device connects to the network, it sends out a DHCPDISCOVER message to the DHCP server⁶.
2. The DHCP server responds with a DHCPOFFER message, which includes network configuration parameters such as IP addresses, subnet masks, and gateway addresses⁴⁵.
3. The client device receives the DHCPOFFER and sends a DHCPREQUEST message back to the server to request the offered address⁵.
4. The server acknowledges the request with a DHCPACK message, completing the configuration⁵.

Unfortunately, I can't provide a diagram here, but I hope this explanation helps you understand code migration and dynamic client-server configuration. If you have any further questions, please feel free to ask me 😊.

3.4 CODE MIGRATION

So far, we have been mainly concerned with distributed systems in which communication is limited to passing data. However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. In this section, we take a detailed look at what code migration actually is. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed. To make matters concrete, we discuss the D'Agents system for mobile agents at the end of this section. Note that security issues concerning code migration are deferred to Chap. 8.

3.4.1 Approaches to Code Migration

ESE

Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another. Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.

Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modern distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, for example, a client-server system in which the server manages a huge database. If a client application needs to do many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

[This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.]

[Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.]

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and deciding in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 1.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that

protocol, would need to be linked with the client application. This requires that the software be readily available to the client at the time the application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-12. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

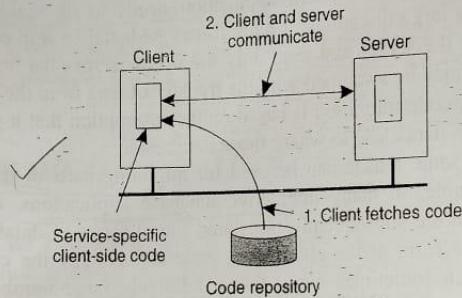


Figure 3-12. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

The important advantage of this model of dynamically downloading client-side software, is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server. There are, of course, also disadvantages. The most serious one, which we discuss in Chap. 8, has to do with security. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code

What is global state of a distributed system ? with the help of a diagram explain the distributed snapshot algorithm

The global state of a distributed system is a collection of the local states of the processes and the channels¹. A global state computed along a consistent cut is correct¹. The global state of a consistent cut comprises the local state of each process at the time the cut event happens and the set of all messages sent but not yet received¹.

The distributed snapshot algorithm, also known as the Chandy-Lamport algorithm, is used to capture a consistent global state of a distributed system². Here's how it works:

1. Any process in the distributed system can initiate this global state recording algorithm using a special message called MARKER².
2. This MARKER traverses the distributed system across all communication channels and causes each process to record its own state².
3. In the end, the state of the entire system (Global state) is recorded².

Unfortunately, I can't provide a diagram here, but I hope this explanation helps you understand the global state of a distributed system and the distributed snapshot algorithm. If you have any further questions, please feel free to ask me 😊.

5.3 GLOBAL STATE *BSK*

On many occasions, it is useful to know the global state in which a distributed system is currently residing. The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered. What exactly the local state

of a process is depends on what we are interested in (Helary, 1989). In the case of a distributed database system, it may consist of only those records that form part of the database and exclude temporary records used for computations. In our example of tracing-based garbage collection as discussed in the previous chapter, the local state may consist of variables representing markings for those proxies, skeletons, and objects that are contained in the address space of a process.

Knowing the global state of a distributed system may be useful for many reasons. For example, when it is known that local computations have stopped and that there are no more messages in transit, the system has obviously entered a state in which no more progress can be made. By analyzing such a global state, it may be concluded that we are either dealing with a deadlock (see, for example, Bracha and Toueg, 1987), or that a distributed computation has correctly terminated. An example of how such an analysis can actually be done is discussed below.

A simple, straightforward way for recording the global state of a distributed system was proposed by Chandy and Lamport (1985) who introduced the notion of a distributed snapshot. A distributed snapshot reflects a state in which the distributed system might have been. An important property is that such a snapshot reflects a consistent global state. In particular, this means that if we have recorded that a process P has received a message from another process Q , then we should also have recorded that process Q had actually sent that message. Otherwise, a snapshot will contain the recording of messages that have been received but never sent, which is obviously not what we want. The reverse condition (Q has sent a message that P has not yet received) is allowed, however.

The notion of a global state can be graphically represented by what is called a cut, as shown in Fig. 5-9. In Fig. 5-9(a), a consistent cut is shown by means of the dashed line crossing the time axis of the three processes P_1 , P_2 , and P_3 . The cut represents the last event that has been recorded for each process. In this case, it can be readily verified that all recorded message receipts have a corresponding recorded send event. In contrast, Fig. 5-9(b) shows an inconsistent cut. The receipt of message m_2 by process P_3 has been recorded, but the snapshot contains no corresponding send event.

To simplify the explanation of the algorithm for taking a distributed snapshot, we assume that the distributed system can be represented as a collection of processes connected to each other through unidirectional point-to-point communication channels. For example, processes may first set up TCP connections before any further communication takes place.

Any process may initiate the algorithm. The initiating process, say P , starts by recording its own local state. Then, it sends a marker along each of its outgoing channels, indicating that the receiver should participate in recording the global state.

When a process Q receives a marker through an incoming channel C , its action depends on whether or not it has already saved its local state. If it has not

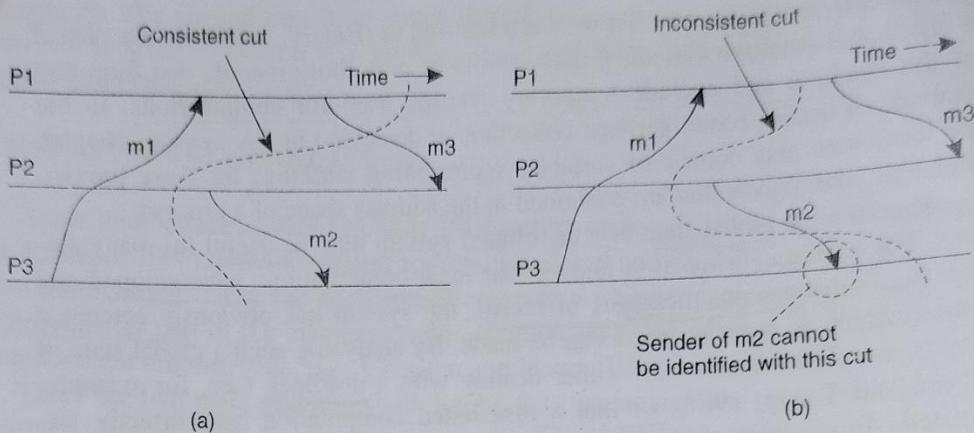


Figure 5-9. (a) A consistent cut. (b) An inconsistent cut.

already done so, it first records its local state and also sends a marker along each of its own outgoing channels. If Q had already recorded its state, the marker on channel C is an indicator that Q should record the state of the channel. This state is formed by the sequence of messages that have been received by Q since the last time Q recorded its own local state, and before it received the marker. Recording this state is shown in Fig. 5-10.

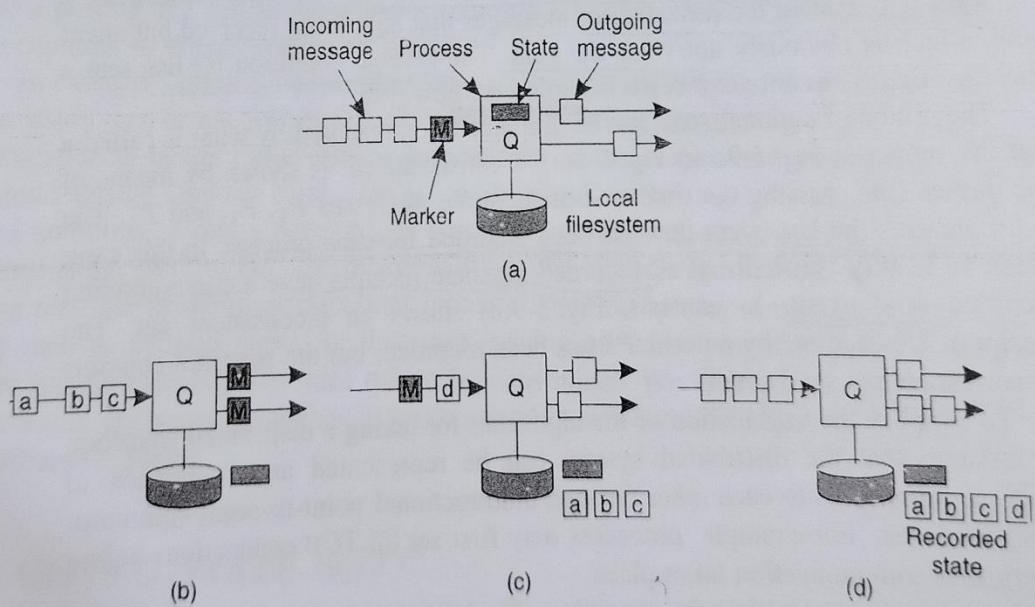


Figure 5-10. (a) Organization of a process and channels for a distributed snapshot. (b) Process Q receives a marker for the first time and records its local state. (c) Q records all incoming messages. (d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel.

A process is said to have finished its part of the algorithm when it has received a marker along each of its incoming channels, and processed each one. At that point, its recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent, for example, to the process that initiated the snapshot. The latter can then subsequently analyze the current state. Note that, meanwhile, the distributed system as a whole can continue to run normally.

It should be noted that because any process can initiate the algorithm, the construction of several snapshots may be in progress at the same time. For this reason, a marker is tagged with the identifier (and possibly also a version number), of the process that initiated the snapshot. Only after a process has received that marker through each of its incoming channels, can it finish its part in the construction of the marker's associated snapshot.

Explain how election is done when any particular system crashes using bully algorithm and ring based algorithm.

In distributed systems, when a process (often referred to as a coordinator or leader) crashes or is otherwise unable to perform its duties, an **election algorithm** is used to select a new coordinator. Two common algorithms for this purpose are the **Bully algorithm** and the **Ring algorithm**. Let's explore how each of these algorithms operates:

Bully Algorithm

The Bully algorithm is based on the principle that the process with the highest identifier (ID) should be the coordinator. [Here's how the election process works using the Bully algorithm¹:](#)

1. **Detection:** A process detects that the coordinator is not responding.
2. **Election Message:** The detecting process sends an election message to all processes with higher IDs.
3. **Wait for Response:** The detecting process waits for a response from these processes.
4. **Coordinator Election:**
 - o If no response is received within a certain time frame, the detecting process assumes that all higher ID processes are down and elects itself as the new coordinator.
 - o If a response is received, the responding process with the highest ID takes over the election process and repeats steps 2 and 3.

5. **Victory Message:** Once a process has determined that it is the highest active process, it sends a victory message to all processes with lower IDs to announce itself as the new coordinator.

Ring Algorithm

The Ring algorithm is used in systems organized in a logical or physical ring. It assumes unidirectional communication between processes. [Here's how the election process works using the Ring algorithm²:](#)

1. **Coordinator Failure Detection:** A process detects that the coordinator has failed.
2. **Election Message:** The detecting process creates an election message containing its own ID and sends it to the next process in the ring.
3. **Passing the Message:** Each process that receives the election message adds its ID to the message and passes it to the next process in the ring.
4. **Receiving Own ID:** When a process receives an election message containing its own ID, it knows that it has made a full circle.
5. **Election Completion:**
 - The process with the highest ID in the election message is elected as the new coordinator.
 - The winning process sends a message around the ring to inform all other processes of the election result.

Both algorithms ensure that a new coordinator is elected in the event of a failure. The Bully algorithm is straightforward and effective in systems where every process can communicate with every other process. The Ring algorithm is suitable for systems arranged in a ring topology, where each process communicates only with its immediate neighbor.

[For more detailed information on these algorithms, including their advantages, disadvantages, and suitability for different types of distributed systems, you can refer to the resources provided¹².](#) These resources will give you a comprehensive understanding of the algorithms and their practical applications in real-world distributed systems.

5.4 ELECTION ALGORITHMS

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process).

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be. Various algorithms are known, for example, (Fredrickson and Lynch, 1987; Garcia-Molina, 1982; and Singh and Kurose, 1994).

5.4.1 The Bully Algorithm

As a first example, consider the **bully algorithm** devised by Garcia-Molina (1982). When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P , holds an election as follows:

1. P sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, P wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. P 's job is done.

At any moment, a process can get an *ELECTION* message from one of its *OK* message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Fig. 5-11 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends *ELECTION* messages to all the processes higher than it, namely 5, 6, and 7, as shown in Fig. 5-11(a). Processes 5 and 6 both respond with *OK*, as shown in Fig. 5-11(b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

In Fig. 5-11(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. 5-11(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a *COORDINATOR* message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send all the others a *COORDINATOR* message and bully them into submission.

5.4.2 A Ring Algorithm

Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an *ELECTION* message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

Fig
5.11
expl'n

SYNCHRONIZATION

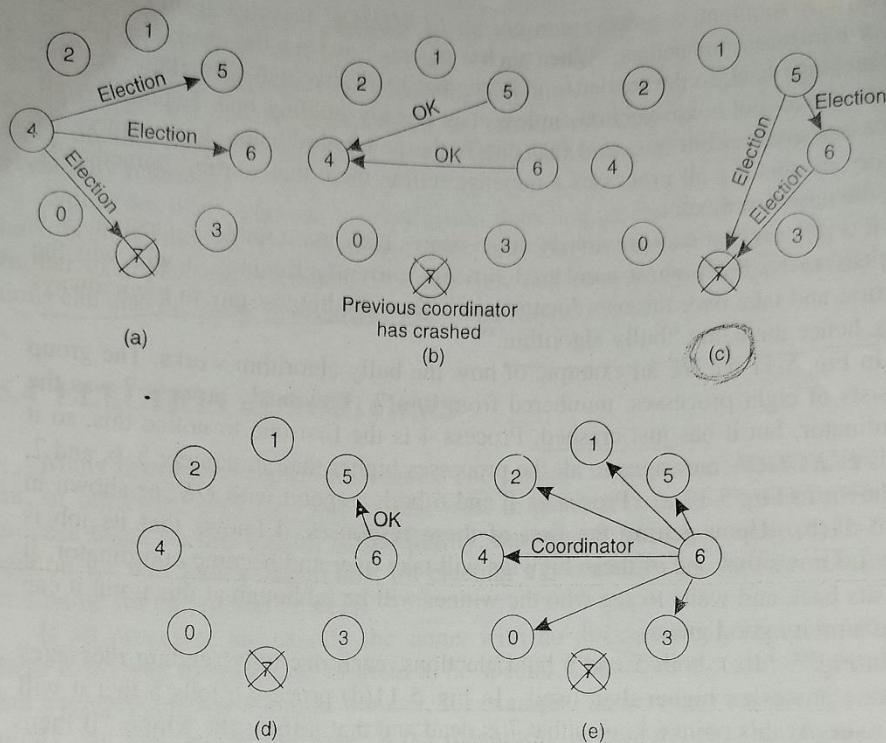


Figure 5-11. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to **COORDINATOR** and circulated once again, this time to inform everyone else who the **coordinator is** (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

In Fig. 5-12 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an **ELECTION** message and each of them starts circulating its message, independent of the other one. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into **COORDINATOR** messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this is not considered wasteful.

MUTUAL EXCLUSION

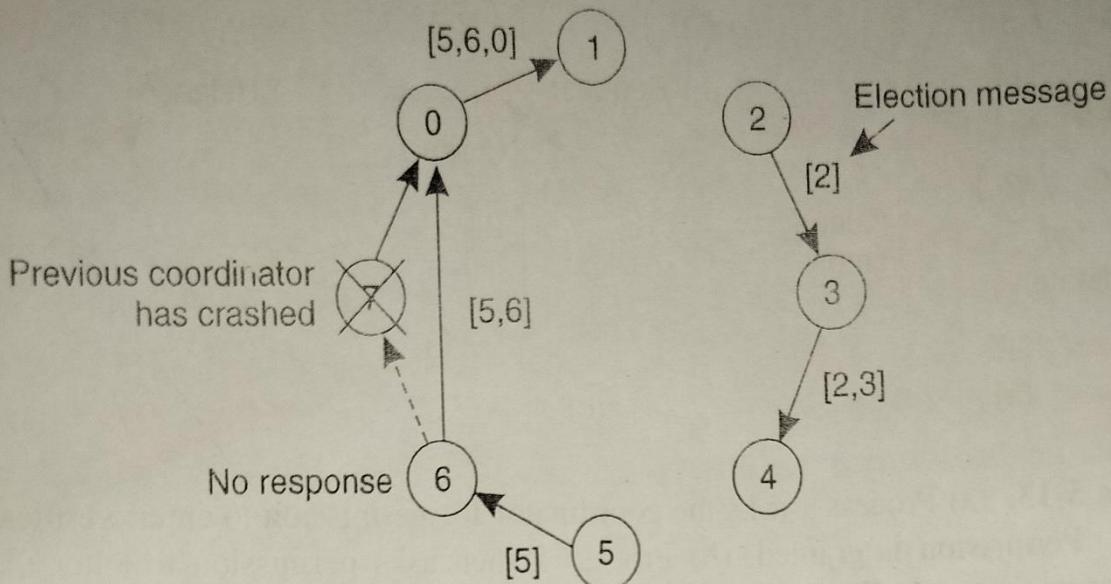


Figure 5-12. Election algorithm using a ring.

Explain the working of distributed mutual exclusion algorithm.

Distributed mutual exclusion is a critical requirement in distributed systems where only one process is allowed to execute the critical section (CS) at any given time¹². In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion¹².

There are two main types of distributed mutual exclusion algorithms: token-based and non-token-based³.

Token-Based Algorithms: A unique token is shared among all the sites. If a site possesses the unique token, it is allowed to enter its critical section³.

Non-Token-Based Algorithms: A site communicates with other sites to get their permission to enter the critical section³. An example of a non-token-based algorithm is Lamport's Distributed Mutual Exclusion Algorithm⁴. In this algorithm:

- A site s_i sends a REQUEST message to all other sites to get their permission to enter the critical section⁴.

- When a site s_j receives the REQUEST message from site s_i , it returns a REPLY message to site s_i ⁴.
- A site s_i can enter the critical section if it has received the message with timestamp larger than (ts_i, i) from all other sites and its own request is at the top of request queue i ⁴.
- When a site s_i exits the critical section, it removes its own request from the top of its request queue and sends a RELEASE message to all other sites⁴.
- When a site s_j receives the RELEASE message from site s_i , it removes the request of s_i from its request queue⁴.

These algorithms ensure that mutual exclusion is achieved in a distributed system³⁴. However, they come with their own set of challenges such as dealing with failures, ensuring fairness, and reducing message complexity³⁴. If you have any further questions, please feel free to ask me 😊.

5.5.2 A Distributed Algorithm ESF

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method.

Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first. Lamport's algorithm presented in

Sec. 5.2.1 is one way to achieve this ordering and can be used to provide time-stamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 5-14(a).

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 5-14(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig. 5-14(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig. 5-14 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and

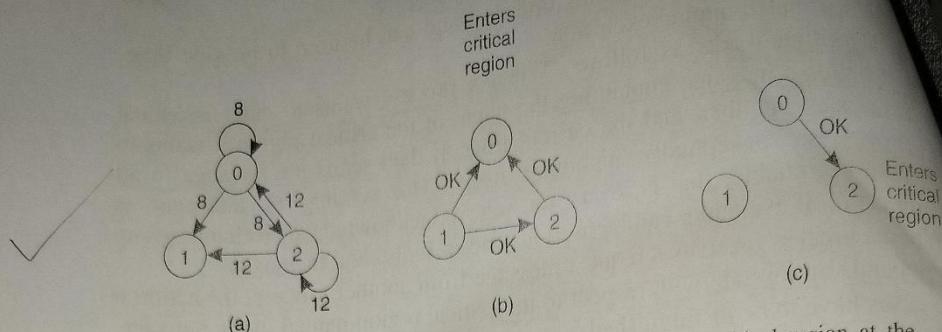


Figure 5-14. (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now $2(n - 1)$, where the total number of processes in the system is n . Best of all, no single point of failure exists.

Synchronization

Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.

Solutions

The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

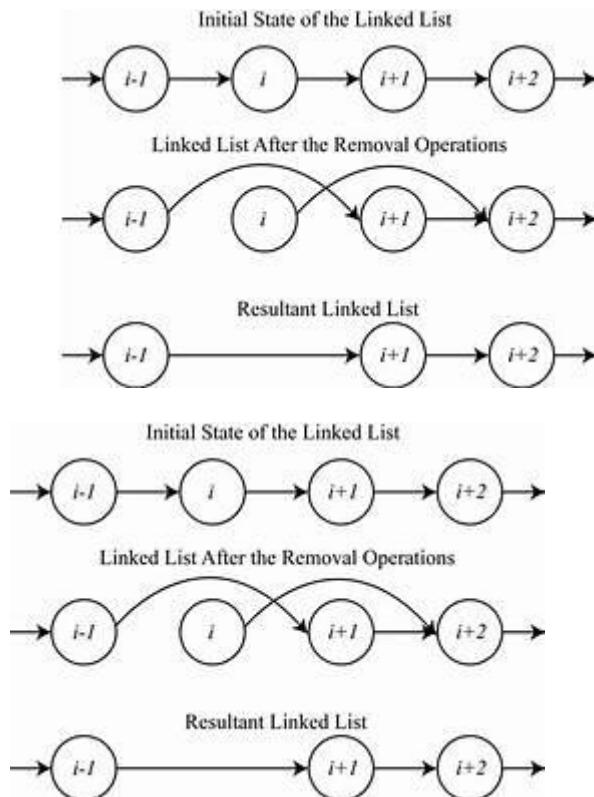
Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If

one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more intricate.

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust than the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

Give me advantages and disadvantages of distributed mutual exclusion algorithm



Explore

Distributed mutual exclusion algorithms are essential for ensuring that only one process at a time can access a shared resource in a distributed system. Here are some advantages and disadvantages of these algorithms:

Advantages:

- [Concurrency Control](#): They prevent race conditions by ensuring that only one process can enter its critical section at a time¹.
- [Fairness](#): Most algorithms are designed to provide fair access to the shared resource, ensuring that every process gets a chance to enter the critical section².
- [Scalability](#): They can be used in large-scale distributed systems without significant issues².
- [Fault Tolerance](#): Some algorithms are designed to recognize failures and continue functioning without disruption¹.
- [No Central Coordinator](#): Algorithms like Lamport's do not require a central coordinator, which avoids a single point of failure³.

Disadvantages:

- [Complexity](#): Implementing mutual exclusion in distributed systems is more complex due to the lack of shared memory and a common physical clock¹.
- [Message Overhead](#): Some algorithms, like Lamport's, require a significant number of messages to be exchanged, which can lead to high network traffic³.
- [Synchronization Delay](#): The time taken to achieve mutual exclusion can be affected by message transmission times, leading to delays³.
- [Failure Handling](#): The failure of one process can halt the progress of the entire system in some algorithms³.
- [Contention](#): High levels of contention for the shared resource can make the algorithm inefficient as it may take several rounds of communication to resolve conflicts⁴.

These algorithms play a crucial role in the design of distributed systems, and their selection should be based on the specific requirements and constraints of the system in question. [For more detailed information on distributed mutual exclusion algorithms, you can refer to the resources provided](#)¹³⁵⁴⁶².

What are the reasons for replication? Explain strict data-centric consistency model.

6.1 INTRODUCTION

In this section, we start with discussing the important reasons for wanting to replicate data in the first place. Special attention is paid to replicating objects, as this forms an increasingly important topic in modern distributed systems. Finally, we discuss replication as a technique for achieving scalability, and motivate why reasoning about consistency is so important.

6.1.1 Reasons for Replication ESE

There are two primary reasons for replicating data: reliability and performance. First, data are replicated to increase the reliability of a system. If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, imagine there are three copies of a file, and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

The other reason for replicating data is performance. Replication for performance is important when the distributed system needs to scale in numbers and geographical area. Scaling in numbers occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the work. We already came across such an example in Chap. 1, when we briefly discussed clusters of replicated Web servers.

Scaling with respect to the size of a geographical area may also require replication. The basic idea is that by placing a copy of data in the proximity of the process using them, the time to access the data decreases. As a consequence, the performance as perceived by that process increases. This example also illustrates that the benefits of replication for performance may be hard to evaluate. Although a client process may perceive better performance, it may also be the case that more network bandwidth is now consumed keeping all replicas up-to-date. We return to such trade-offs when discussing distribution protocols.

If replication helps to improve reliability and performance, who could be against it? Unfortunately, there is a price to be paid when data are replicated. The problem with replication is that having multiple copies may lead to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. Exactly when and how those modifications need to be carried out determines the price of replication.

To understand the problem, consider improving access times to Web pages. If no special measures are taken, fetching a page from a remote Web server may sometimes even take seconds to complete. To improve performance, Web browsers often locally store a copy of a previously fetched Web page (i.e., they cache a Web page). If a user requires that page again, the browser automatically returns the local copy. The access time as perceived by the user is excellent. However, if the user always wants to have the latest version of a page, he may be in for bad luck. The problem is that if the page has been modified in the meantime, modifications will not have been propagated to cached copies, making those copies out-of-date.

One solution to the problem of returning a stale copy to the user is to forbid the browser to keep local copies in the first place, effectively letting the server be fully in charge of replication. However, this solution may still lead to poor access times if no replica is placed near the user. Another solution is to let the Web server invalidate or update each cached copy, but this requires that the server keeps track of all caches and sending them messages. This, in turn, may degrade the overall performance of the server. We return to performance versus scalability issues below.

6.2.1 Strict Consistency $\Leftarrow \Leftarrow$

The most stringent consistency model is called strict consistency. It is defined by the following condition:

Any read on a data item x returns a value corresponding to the result of the most recent write on x .

This definition is natural and obvious, although it implicitly assumes the existence of absolute global time (as in Newtonian physics) so that the determination of "most recent" is unambiguous. Uniprocessor systems have traditionally observed strict consistency and uniprocessor programmers have come to expect such behavior as a matter of course. A system on which the program

```
a = 1; a = 2; print(a);
```

printed 1 or any value other than 2 would quickly lead to a lot of very agitated programmers, and for good reason too.

In a system in which data are spread across multiple machines, and which can be accessed by multiple processes, matters are more complicated. Suppose x is a data item stored only on machine B . Imagine that a process on machine A reads x at time T_1 , which means that a message is then sent to B to get x . Slightly later, at T_2 , a process on B does a write to x . If strict consistency holds, the read should always return the old value regardless of where the machines are and how close T_2 is to T_1 . However, if $T_2 - T_1$ is, say, 1 nanosecond, and the machines are 3 meters apart, in order to propagate the read request from A to B to get there before the write, the signal would have to travel at 10 times the speed of light, something forbidden by Einstein's special theory of relativity. Is it reasonable for programmers to demand that the system be strictly consistent, even if this requires violating the laws of physics?

The problem with strict consistency is that it relies on absolute global time. In essence, it is impossible in a distributed system to assign a unique timestamp to each operation that corresponds to actual global time. We can relax this situation by dividing time into a series of consecutive, nonoverlapping intervals. Each operation is assumed to take place within an interval and receives a timestamp that corresponds to that interval. Depending on how accurate clocks can be synchronized, we may now reach a situation in which there is at most one operation per interval.

Unfortunately, no guarantees can be given that at most a single operation takes place within an interval. Consequently, we still need to deal with multiple operations that take place within the same interval. Analogous to our discussion of concurrency control for distributed transactions, two operations in the same interval are said to conflict if they operate on the same data and one of them is a write operation. An important issue in defining consistency models is to define exactly what kind of behavior is acceptable in the presence of conflicting operations.

To study consistency in detail, we will give numerous examples. To make these examples precise, we need a special notation in which we draw the operations of a process along a time axis. The time axis is always drawn horizontally, with time increasing from left to right. The symbols

$W_i(x)a$ and $R_i(x)b$

mean that a write by process P_i to data item x with the value a and a read from that item by P_i returning b have been done, respectively. We assume that each data item is initially NIL . When there is no confusion concerning which process is accessing data, we omit the index from the symbols W and R .

As an example, in Fig. 6-5(a) P_1 does a write to a data item x , modifying its value to a . Note that, in principle, this operation $W_1(x)a$ is first performed on a copy of the data store that is local to P_1 , and is then subsequently propagated to the other local copies. In our example, P_2 later reads x (from its local copy of the

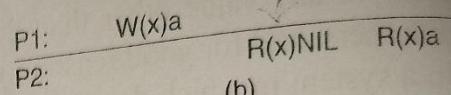
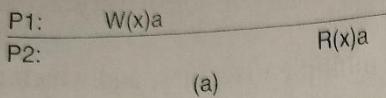


Figure 6-5. Behavior of two processes operating on the same data item. The horizontal axis is time. (a) A strictly consistent store. (b) A store that is not strictly consistent.

store and sees value a . This behavior is correct for a strictly consistent data store. In contrast, in Fig. 6-5(b), P_2 does a read after the write (possibly only a nanosecond after it, but still after it), and gets NIL . A subsequent read returns a . Such behavior is incorrect for a strictly consistent data store.

In summary, when a data store is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a data item is changed, all subsequent reads performed on that data return the new value, no matter how soon after the change the reads are done, and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the then-current value, no matter how quickly the next write is done.

In the next section, we will relax this model by considering time intervals instead of absolute time, and define precisely what is acceptable behavior for conflicting operations.

Reasons for Replication:

Replication is the process of creating and maintaining copies of data or services in multiple locations, typically distributed across a network. There are several reasons for adopting replication in distributed systems:

1. Improved Availability:

- Replication enhances system availability by providing alternative access points. If one replica is unavailable due to network issues or node failures, clients can still access other replicas.

2. Fault Tolerance:

- Replication helps in creating fault-tolerant systems. If one replica becomes unavailable or experiences a failure, other replicas can continue to serve requests, ensuring uninterrupted operation.

3. Load Balancing:

- Distributing replicas across multiple nodes allows for load balancing. Incoming requests can be distributed among replicas, preventing any single node from being overwhelmed with requests.

4. Reduced Latency:

- Placing replicas closer to users or clients can reduce latency. Users can access data or services from a nearby replica, leading to faster response times and improved user experience.

5. Scalability:

- Replication supports system scalability. As the demand for resources or services increases, additional replicas can be deployed to handle the increased load, providing a scalable and efficient solution.

6. Disaster Recovery:

- Replication aids in disaster recovery by ensuring that data or services are available even in the event of a catastrophic failure or natural disaster affecting a specific location.

7. Consistency and Performance Trade-Off:

- Replication allows for tuning the trade-off between consistency and performance. Systems can be configured to prioritize consistency by ensuring that all replicas are updated before responding to clients, or to prioritize performance by allowing some replicas to be slightly out-of-date.

Strict Data-Centric Consistency Model:

The strict data-centric consistency model is a type of consistency model in distributed systems that emphasizes strong consistency guarantees. In this model:

1. Consistency:

- All replicas of the data are consistent with each other at all times. Any read operation on any replica will return the most recent write.

2. Synchronization:

- Strict synchronization is maintained between replicas. Before a write is acknowledged, it must be replicated to all other replicas.

3. No Read Stale Data:

- Clients always read the most up-to-date data. There is no allowance for reading stale or outdated information.

4. Linearizability:

- The model is often associated with linearizability, a property that ensures that the system's behavior appears as if all operations are executed instantaneously at some point between their invocation and response.

5. Complexity and Overhead:

- Achieving strict data-centric consistency often involves significant complexity and overhead. Synchronization mechanisms, such as two-phase commit protocols or consensus algorithms, may be used to ensure consistency.

6. **Use Cases:**

- This model is suitable for applications where data consistency is of utmost importance, such as financial systems, databases, or systems dealing with critical data.

7. **Trade-Off with Performance:**

- While providing strong consistency guarantees, the strict data-centric consistency model can come with a trade-off in terms of performance. Synchronization and coordination among replicas may introduce delays.

8. **Challenges with Scalability:**

- Ensuring strict consistency across a large number of replicas can pose challenges in terms of scalability, especially as the number of nodes in the system grows.

Explain Berkeley's Algorithm and averaging algorithm used for clock synchronization with examples

The Berkeley Algorithm BSE

In Cristian's algorithm, the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries. In Berkeley UNIX, exactly the opposite approach is taken (Gusella and Zatti, 1989). Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a WWV receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Fig. 5-6.

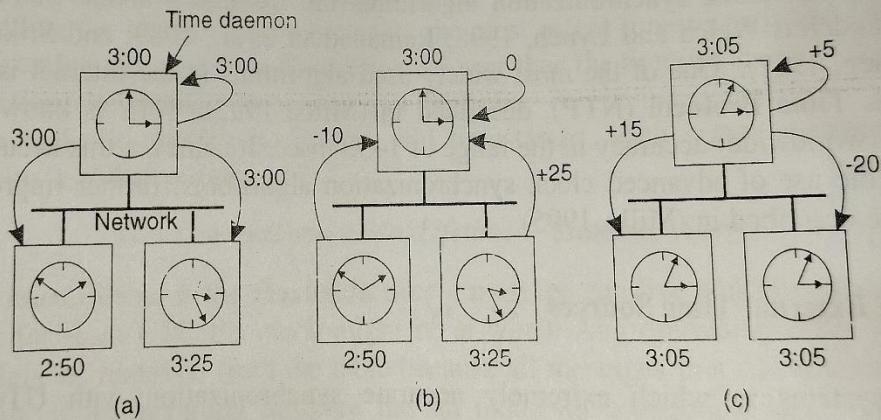


Figure 5-6. (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

In Fig. 5-6(a), at 3:00, the time daemon tells the other machines its time and asks for theirs. In Fig. 5-6(b), they respond with how far ahead or behind the time

daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Fig. 5-6(c)].

Averaging Algorithms *ESE*

Both of the methods described above are highly centralized with the usual disadvantages. Decentralized algorithms are also known. One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals. The i th interval starts at $T_0 + iR$ and runs until $T_0 + (i+1)R$, where T_0 is an agreed-upon moment in the past, and R is a system parameter. At the beginning of each interval, every machine broadcasts the current time according to its clock. Because the clocks on different machines do not run at exactly the same speed, these broadcasts will not happen precisely simultaneously.

After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval S . When all the broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all the other machines. A slight variation on this theme is first to discard the m highest and m lowest values, and average the rest. Discarding the extreme values can be regarded as self defense against up to m faulty clocks sending out nonsense.

Another variation is to try to correct each message by adding to it an estimate of the propagation time from the source. This estimate can be made from the known topology of the network, or by timing how long it takes for probe messages to be echoed.

Additional clock synchronization algorithms are discussed in the literature (e.g., Lundelius-Welch and Lynch, 1998; Ramanathan et al., 1989; and Srikanth and Toueg, 1987). One of the most widely used algorithms in the Internet is the **Network Time Protocol (NTP)**, described in (Mills, 1992). NTP is known to achieve (worldwide) accuracy in the range of 1–50 msec. It achieves this accuracy through the use of advanced clock synchronization algorithms; further improvements are described in (Mills, 1995).

Certainly! Let's start with **Berkeley's Algorithm** and then discuss an averaging algorithm used for clock synchronization.

Berkeley's Algorithm

Berkeley's Algorithm is a clock synchronization technique used in distributed systems, particularly when no machine has an accurate time source or access to UTC (Universal Time Coordinated) servers¹. Here's how it works:

1. **Master Node Selection:** A master node is chosen from the network nodes, often through an election process. [This master node acts as the time coordinator](#)¹.
2. **Time Polling:** The master node periodically pings the slave nodes to fetch their local times. [This can be done using Cristian's algorithm](#)¹.
3. **Average Time Calculation:** The master node calculates the average time difference between the times received from the slave nodes and its own system clock¹.
4. **Time Adjustment:** The master node then adjusts its clock by the average time difference and broadcasts this adjustment to the slave nodes¹.
5. **Clock Update:** Each slave node adjusts its clock by the amount received from the master node. [This helps synchronize the clocks across the network](#)¹.

Example:

Suppose we have three nodes with the following local times:

- Node A: 2:00 PM
- Node B: 2:05 PM
- Node C: 1:55 PM

The master node (Node A) calculates the average time difference:

- Node B is 5 minutes ahead.
- Node C is 5 minutes behind.

The average difference is $(5 - 5) / 2 = 0$ minutes. Node A broadcasts a message to Node B and Node C to adjust their clocks by 0 minutes, effectively synchronizing all nodes to 2:00 PM.

Averaging Algorithm for Clock Synchronization

An averaging algorithm for clock synchronization involves each node sharing its local time with other nodes and adjusting its clock based on the average time difference. [This is a distributed approach where there is no single master node](#)².

Here's a simplified version of how it might work:

1. **Local Time Sharing:** Each node sends its local time to all other nodes in the system².
2. **Average Time Calculation:** Upon receiving all the times, each node calculates the average time difference between its own time and the times received².

3. Clock Adjustment: Each node then adjusts its clock by the calculated average time difference².

Example:

Let's say we have three nodes with the following local times:

- Node X: 3:00 PM
- Node Y: 3:04 PM
- Node Z: 2:56 PM

Each node calculates the average time:

- Node X's average difference: $(4 - 4) / 2 = 0$ minutes.
- Node Y's average difference: $(-4 + 4) / 2 = 0$ minutes.
- Node Z's average difference: $(4 - 4) / 2 = 0$ minutes.

All nodes adjust their clocks by 0 minutes, maintaining their original times but now with a consensus that they are synchronized.

These algorithms help maintain time consistency across distributed systems, which is crucial for coordination and consistency of operations. [For more detailed information on these algorithms, you can refer to the resources provided¹²³⁴.](#)

DEC 2022

Identify the various problems of replication. How to improve the performance of distributed application using replication?

Problems of Replication:

1. Consistency Challenges:

- Maintaining consistency across replicas can be challenging, especially in the presence of concurrent updates. Achieving consistency without introducing significant delays or compromising performance is a complex task.

2. Concurrency Control:

- Coordinating concurrent updates from multiple replicas requires robust concurrency control mechanisms to prevent conflicts, race conditions, and ensure a correct and coherent view of the data.

3. Replica Synchronization:

- Ensuring that all replicas are synchronized and up-to-date introduces challenges. Replication protocols must handle scenarios where some

replicas may lag behind due to network delays, failures, or temporary unavailability.

4. **Fault Tolerance:**

- While replication enhances fault tolerance, it also introduces challenges in handling failures. Detecting and recovering from replica failures without compromising availability and consistency is crucial.

5. **Network Overhead:**

- Replicating data across a network introduces additional communication overhead. This can impact the performance of the system, especially in scenarios with high network latency or limited bandwidth.

6. **Data Access Patterns:**

- Replication strategies must consider the access patterns of the application. In scenarios with skewed access patterns or hotspots, replicas may experience uneven loads, leading to performance bottlenecks.

7. **Load Balancing:**

- Efficiently distributing the load among replicas to achieve optimal load balancing can be challenging. Improper load balancing may result in underutilized or overloaded replicas.

8. **Heterogeneity:**

- In heterogeneous environments, where replicas may have different capabilities or performance characteristics, ensuring consistent performance and behavior across all replicas becomes challenging.

Improving Performance with Replication:

1. **Load Balancing:**

- Use dynamic load balancing algorithms to distribute requests evenly among replicas. This helps avoid overloading some replicas while underutilizing others.

2. **Caching:**

- Implement caching mechanisms to reduce the need for frequent accesses to replicas. Local caches can store frequently accessed data, reducing the load on replicas and improving response times.

3. **Asynchronous Replication:**

- Consider asynchronous replication for scenarios where slight inconsistencies are acceptable. This can improve write performance by allowing the primary replica to acknowledge writes without waiting for all replicas to be updated.

4. **Consistency Levels:**

- Adjust the consistency levels based on application requirements. In scenarios where strong consistency is not critical, using eventual consistency or relaxed consistency models can improve performance.

5. Partitioning:

- Use data partitioning strategies to distribute data across replicas in a way that aligns with access patterns. This helps in avoiding hotspots and ensures a more balanced distribution of workload.

6. Parallel Processing:

- Exploit parallel processing capabilities by allowing replicas to handle requests independently whenever possible. This can be particularly effective in read-heavy scenarios.

7. Failure Detection and Recovery:

- Implement robust failure detection mechanisms to quickly identify and recover from replica failures. This helps in maintaining system availability and reliability.

8. Optimized Network Protocols:

- Use optimized network protocols and efficient data serialization formats to minimize network overhead during data replication.

9. Quorum-based Systems:

- Explore quorum-based replication systems that allow for a flexible trade-off between consistency and performance. Adjusting the quorum size can impact the number of replicas that need to agree on updates.

10. Replica Placement:

- Strategically place replicas in locations that minimize latency for end-users. This is especially important for applications with a geographically distributed user base.

Compare the Cristian's clock synchronization algorithm and Berkeley's clock synchronization algorithm.

Cristian's Clock Synchronization Algorithm:

1. Centralized Approach:

- Cristian's algorithm follows a centralized approach where one server (time server) is considered authoritative for providing the correct time.

2. Request and Response:

- The client sends a request to the time server, which responds with its current time.

3. Calculation of Offset:

- The client calculates the time offset by half of the round-trip time (RTT) of the request. The assumption is that the time taken for the request to reach the server and the response to return is roughly equal.

4. **Synchronization:**

- The client adjusts its local clock by applying the calculated time offset. This synchronization process assumes that network delays remain constant.

5. **Assumption:**

- The algorithm assumes symmetric network delays, meaning that the time it takes for a message to travel from the client to the server is the same as the time it takes for the response to travel from the server to the client.

Berkeley's Clock Synchronization Algorithm:

1. **Distributed Approach:**

- Berkeley's algorithm takes a distributed approach where each machine in the network periodically exchanges its local time with a coordinator (time daemon).

2. **Calculation of Time Offset:**

- The coordinator calculates the time offset for each machine by comparing their local time with its own. The offset is the average difference between the local time of the machines and the coordinator's time.

3. **Adjustment of Local Clocks:**

- The coordinator sends the time offset to each machine, and each machine adjusts its local clock accordingly.

4. **Regular Updates:**

- The process is repeated at regular intervals to account for changes in clock drift or network conditions.

5. **Assumption:**

- Berkeley's algorithm assumes that the time offset calculated for each machine remains relatively constant between synchronization intervals.

Comparison:

1. **Centralization vs. Distribution:**

- Cristian's algorithm relies on a central time server, making it a centralized approach. In contrast, Berkeley's algorithm distributes the synchronization process among all machines in the network.

2. **Handling Asymmetric Delays:**

- Cristian's algorithm assumes symmetric network delays, which may not hold true in all cases. Berkeley's algorithm can handle asymmetric delays more gracefully since it considers the average offset.

3. Robustness:

- Berkeley's algorithm is often considered more robust in scenarios where machines in the network experience varying clock drifts and asymmetric delays. It provides a better average estimate of the time offset.

4. Accuracy:

- Cristian's algorithm can be more accurate when network delays are relatively constant and symmetric. However, it may be sensitive to variations in network conditions.

5. Frequency of Updates:

- Berkeley's algorithm can adapt to changes more frequently as it allows for periodic updates. Cristian's algorithm typically synchronizes less frequently as it relies on explicit requests.

6. Scalability:

- Berkeley's algorithm may be more scalable in large networks as it distributes the synchronization process. Cristian's algorithm may face scalability challenges as it centralizes the synchronization on a single time server.

Cristian's Algorithm

Let us start with an algorithm that is well suited to systems in which one machine has a WWV receiver and the goal is to have all the other machines stay synchronized with it. Let us call the machine with the WWV receiver a time server. Our algorithm is based on the work of Cristian (1989) and prior work. Periodically, certainly no more than every $\delta/2\rho$ seconds, each machine sends a

SYNCHRONIZATION

248

message to the time server asking for the current time. That machine responds as fast as it can with a message containing its current time, C_{UTC} , as shown in Fig. 5-5.

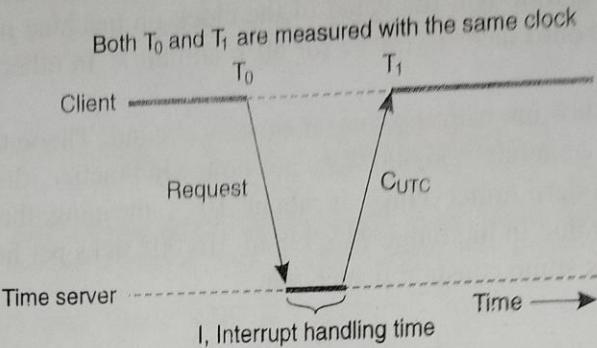


Figure 5-5. Getting the current time from a time server.

As a first approximation, when the sender gets the reply, it can just set its clock to C_{UTC} . However, this algorithm has two problems, one major and one minor. The major problem is that time must never run backward. If the sender's clock is fast, C_{UTC} will be smaller than the sender's current value of C . Just taking over C_{UTC} could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

The minor problem is that it takes a nonzero amount of time for the time server's reply to get back to the sender. Worse yet, this delay may be large and vary with the network load. Cristian's way of dealing with it is to attempt to measure it. It is simple enough for the sender to record accurately the interval between sending the request to the time server and the arrival of the reply. Both the starting time, T_0 , and the ending time, T_1 , are measured using the same clock, so the interval will be relatively accurate even if the sender's clock is off from UTC by a substantial amount.

In the absence of any other information, the best estimate of the message propagation time is $(T_1 - T_0)/2$. When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server's current time. If the theoretical minimum propagation time is known, other properties of the time estimate can be calculated.

This estimate can be improved if it is known approximately how long it takes the time server to handle the interrupt and process the incoming message. Let us

call the interrupt handling time I . Then the amount of the interval from T_0 to T_1 that was devoted to message propagation is $T_1 - T_0 - I$, so the best estimate of the one-way propagation time is half this. Systems do exist in which messages from A to B systematically take a different route than messages from B to A , and thus have a different propagation time, but we will not consider such systems here.

To improve the accuracy, Cristian suggested making not one measurement but a series of them. Any measurements in which $T_1 - T_0$ exceeds some threshold value are discarded as being victims of network congestion and thus are unreliable. The estimates derived from the remaining probes can then be averaged to get a better value. Alternatively, the message that came back fastest can be taken to be the most accurate since it presumably encountered the least traffic underway and therefore is the most representative of the pure propagation time.

The Berkeley Algorithm ESE

In Cristian's algorithm, the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries. In Berkeley UNIX, exactly the opposite approach is taken (Gusella and Zatti, 1989). Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a WWV receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Fig. 5-6.

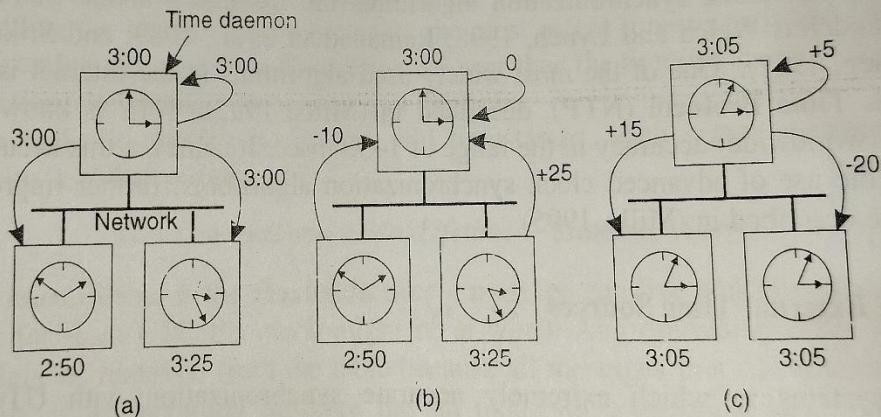


Figure 5-6. (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

In Fig. 5-6(a), at 3:00, the time daemon tells the other machines its time and asks for theirs. In Fig. 5-6(b), they respond with how far ahead or behind the time

daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Fig. 5-6(c)].

What is RMI? Compare between static and dynamic RMI.

2.3.3 Static versus Dynamic Remote Method Invocations ESF

After a client is bound to an object, it can invoke the object's methods through the proxy. Such a **remote method invocation** or simply **RMI**, is very similar to an **RPC** when it comes to issues such as marshaling and parameter passing. An essential difference between an **RMI** and an **RPC**, is that **RMIs** generally support **systemwide object references** as explained above. Also, it is not necessary to have **only general-purpose client-side and server-side stubs available**. Instead, we can more easily **accommodate object-specific stubs** as we also explained.

The usual way to provide **RMI** support is to specify the object's interfaces in an **interface definition language**, similar to the approach followed with **RPCs**. Alternatively, we can make use of an **object-based language** such as **Java**, that will handle **stub generation** automatically. This approach of using predefined interface definitions is generally referred to as **static invocation**. **Static invocations** require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.

As an alternative, method invocations can also be done in a more dynamic fashion. In particular, it is sometimes convenient to be able to compose a method invocation at runtime, also referred to as a **dynamic invocation**. The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object. Dynamic invocation generally takes a form such as

invoke(object, method, input_parameters, output_parameters);

where *object* identifies the distributed object, *method* is a parameter specifying exactly which method should be invoked, *input_parameters* is a data structure that holds the values of that method's input parameters, and *output_parameters* refers to a data structure where output values can be stored.

For example, consider appending an integer *int* to a file object *fobject*, for which the object provides the method *append*. In this case, a static invocation would take the form

fobject.append(int)

whereas the dynamic invocation would look something like

invoke(*fobject*, *id(append)*, *int*)

where the operation *id(append)* returns an identifier for the method *append*

To illustrate the usefulness of dynamic invocations, consider an object browser that is used to examine sets of objects. Assume that the browser supports remote object invocations. Such a browser is capable of binding to a distributed object and subsequently presenting the object's interface to its user. The user could then be asked to choose a method and provide values for its parameters, after which the browser can do the actual invocation. Typically, such an object browser should be developed to support any possible interface. Such an approach requires that interfaces can be inspected at runtime, and that method invocations can be dynamically constructed.

Another application of dynamic invocations is a batch processing service to which invocation requests can be handed along with a time when the invocation should be done. The service can be implemented by a queue of invocation requests, ordered by the time that invocations are to be done. The main loop of the service would simply wait until the next invocation is scheduled, remove the request from the queue, and call *invoke* as given above.

2.3.4 Parameter Passing ES E

Because most RMI systems support systemwide object references, passing parameters in method invocations is generally less restricted than in the case of RPCs. However, there are some subtleties that can make RMIs trickier than one would initially expect, as we briefly discuss in the following pages.

Explain the general structure of distributed system as middleware. State the any three services provided by middleware.

1.4.3 Middleware ESE

Neither a distributed operating system or a network operating system really qualifies as a distributed system according to our definition given in Sec. 1.1. A distributed operating system is not intended to handle a collection of *independent computers*, while a network operating system does not provide a view of a *single coherent system*. The question comes to mind whether it is possible to develop a distributed system that has the best of both worlds: the scalability and openness of network operating systems and the transparency and related ease of use of distributed operating systems. The solution is to be found in an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency. Many modern distributed systems are constructed by means of such an additional layer of what is called **middleware**. In this section we take a closer look at what middleware actually constitutes by explaining some of its features.

Positioning Middleware ESE

Many distributed applications make direct use of the programming interface offered by network operating systems. For example, communication is often expressed through operations on sockets, which allow processes on different machines to pass each other messages (Stevens, 1998). In addition, applications often make use of interfaces to the local file system. As we explained, a problem with this approach is that distribution is hardly transparent. A solution is to place an additional layer of software between applications and the network operating system, offering a higher level of abstraction. Such a layer is accordingly called **middleware**. It sits in the middle between applications and the network operating system as shown in Fig. 1-22.

Each local system forming part of the underlying network operating system is assumed to provide local resource management in addition to simple communication means to connect to other computers. In other words, middleware itself will not manage an individual node; this is left entirely to the local operating system.

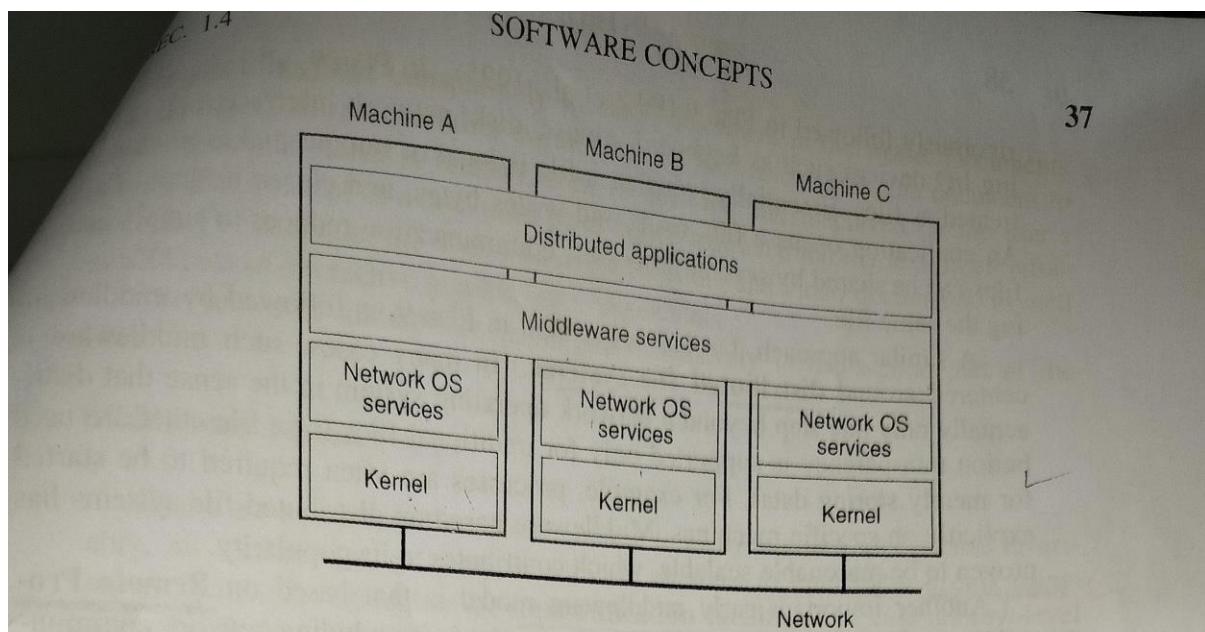


Figure 1-22. General structure of a distributed system as middleware.

An important goal is to hide heterogeneity of the underlying platforms from applications. Therefore, many middleware systems offer a more-or-less complete collection of services and discourage using anything else but their interfaces to those services. In other words, skipping the middleware layer and immediately calling services of one of the underlying operating systems is often frowned upon. We will return to middleware services shortly.

It is interesting to note that middleware was not invented as an academic exercise in achieving distribution transparency. After the introduction and widespread use of network operating systems, many organizations found themselves having lots of networked applications that could not be easily integrated into a single system (Bernstein, 1996). At that point, manufacturers started to build higher-level, application-independent services into their systems. Typical examples include support for distributed transactions and advanced communication facilities.

Of course, agreeing on what the right middleware should be is not easy. An approach is to set up an organization which subsequently defines a common standard for some middleware solution. At present, there are a number of such standards available. The standards are generally not compatible with each other, and even worse, products implementing the same standard but from different manufacturers rarely interwork. Surely, it will not be long before someone offers "upperware" to remedy this defect.

Middleware Models

To make development and integration of distributed applications as simple as possible, most middleware is based on some model, or paradigm, for describing distribution and communication. A relatively simple model is that of treating everything as a file. This is the approach originally introduced in UNIX and

rigorously followed in Plan 9 (Pike et al., 1995). In Plan 9, all resources, including I/O devices such as keyboard, mouse, disk, network interface, and so on, are treated as files. Essentially, whether a file is local or remote makes no difference. An application opens a file, reads and writes bytes, and closes it again. Because files can be shared by several processes, communication reduces to simply accessing the same file.

A similar approach, but less strict than in Plan 9, is followed by middleware centered around distributed file systems. In many cases, such middleware is actually only one step beyond a network operating system in the sense that distribution transparency is supported only for traditional files (i.e., files that are used for merely storing data). For example, processes are often required to be started explicitly on specific machines. Middleware based on distributed file systems has proven to be reasonably scalable, which contributes to its popularity.

Another important early middleware model is that based on Remote Procedure Calls (RPCs). In this model, the emphasis is on hiding network communication by allowing a process to call a procedure of which an implementation is located on a remote machine. When calling such a procedure, parameters are transparently shipped to the remote machine where the procedure is subsequently executed, after which the results are sent back to the caller. It therefore appears as if the procedure call was executed locally: the calling process remains unaware of the fact that network communication took place, except perhaps for some loss of performance. We return to remote procedure calls in the next chapter.

As object orientation came into vogue, it became apparent that if procedure calls could cross machine boundaries, it should also be possible to invoke objects residing on remote machines in a transparent fashion. This has now led to various middleware systems offering a notion of distributed objects. The essence of distributed objects is that each object implements an interface that hides all the internal details of the object from its users. An interface consists of the methods that the object implements, no more and no less. The only thing that a process sees of an object is its interface.

Distributed objects are often implemented by having each object itself located on a single machine, and additionally making its interface available on many other machines. When a process invokes a method, the interface implementation on the process's machine simply transforms the method invocation into a message that is sent to the object. The object executes the requested method and sends back the result. The interface implementation subsequently transforms the reply message into a return value, which is then handed over to the invoking process. As in the case of RPC, the process may be kept completely unaware of the network communication.

What models can do to simplify the use of networked systems is probably best illustrated by the World Wide Web. The success of the Web is mainly due to the extremely simple, yet highly effective model of distributed documents. In the model of the Web, information is organized into documents, with each document

residing at a machine transparently located somewhere in the world. Documents contain links that refer to other documents. By following a link, the document to which that link refers is fetched from its location and displayed on the user's screen. The concept of a document need not be restricted to only text-based information. For example, the Web also supports audio and video documents, as well as all kinds of interactive graphic-based documents.

We return to middleware paradigms extensively in the second part of the book.

Middleware Services ESE

There are a number of services common to many middleware systems. Invariably, all middleware, one way or another, attempts to implement access transparency, by offering high-level **communication facilities** that hide the low-level message passing through computer networks. The programming interface to the transport layer as offered by network operating systems is thus entirely replaced by other facilities. How communication is supported depends very much on the model of distribution the middleware offers to users and applications. We already mentioned remote procedure calls and distributed-object invocations. In addition, many middleware systems provide facilities for transparent access to remote data, such as distributed file systems or distributed databases. Transparently fetching documents as is done in the Web is another example of high-level (one-way) communication.

An important service common to all middleware is that of naming. Name services allow entities to be shared and looked up (as in directories), and are comparable to telephone books and the yellow pages. Although naming may seem simple at first thought, difficulties arise when scalability is taken into account. Problems are caused by the fact that to efficiently look up a name in a large-scale system, the location of the entity that is named must be assumed to be fixed. This assumption is made in the World Wide Web, in which each document is currently named by means of a URL. A URL contains the name of the server where the document to which the URL refers is stored. Therefore, if the document is moved to another server, its URL ceases to work.

Many middleware systems offer special facilities for storage, also referred to as persistence. In its simplest form, persistence is offered through a distributed file system, but more advanced middleware have integrated databases into their systems, or otherwise provide facilities for applications to connect to databases.

In environments where data storage plays an important role, facilities are generally offered for distributed transactions. An important property of a transaction is that it allows multiple read and write operations to occur atomically. Atomicity means that the transaction either succeeds, so that all its write operations are actually performed, or it fails, leaving all referenced data unaffected. Distributed transactions operate on data that are possibly spread across multiple machines.

Especially in the face of masking failures, which is often hard in distributed systems, it is important to offer services such as distributed transactions. Unfortunately, transactions are hard to scale across many local machines, let alone geographically dispersed machines.

Finally, virtually all middleware systems that are used in nonexperimental environments provide facilities for **security**. Compared to network operating systems, the problem with security in middleware is that it should be pervasive. In principle, the middleware layer cannot rely on the underlying local operating systems to adequately support security for the complete network. Consequently, **security has to be partly implemented anew in the middleware layer itself**. Combined with the need for extensibility, security has turned out to be one of the hardest services to implement in distributed systems.

A client makes a remote procedure call to a server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local operating system processing time for each send or receive operation is 0.5 milliseconds, and the network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per seconds per message. Evaluate the time taken by the client to generate and return from two requests:

- i) if it is single threaded and
- ii) if it has two threads that can make request concurrently on single processor.

Ignore the context switching time.

Q.2a

For single Thread

$$\begin{aligned}
 \text{Time per call} &= \text{compute args + marshalling + local O.S. +} \\
 &\quad \text{return local O.S. + unmarshalling} \\
 &= 5 + 0.5 + 0.5 + 0.5 + 0.5 \\
 &= 5 + 2 = 7
 \end{aligned}$$

$$\begin{aligned}
 \text{Transmission time} &= \text{request + reply} \\
 &= 3 + 3 = 6
 \end{aligned}$$

$$\begin{aligned}
 \text{Server} &= \text{local O.S. + unmarshalling + process req. +} \\
 &\quad \text{+ marshall reply + local O.S. reply.} \\
 &= 0.5 + 0.5 + 10 + 0.5 + 0.5 \\
 &= 12
 \end{aligned}$$

$$\begin{aligned}
 \text{Total time for one request} &= \text{client + transmission + server} \\
 &= 7 + 6 + 12 = 25 \text{ milliseconds}
 \end{aligned}$$

∴ For two requests,

$$\text{Time taken} = 25 \times 2 = 50 \text{ milliseconds.}$$

(i) For two Threads

$$\begin{aligned}
 \text{Client time} &= \text{compute + unmarshal + local O.S.} \\
 &= 0.5 + 0.5 + 0.5 = 6
 \end{aligned}$$

$$\begin{aligned}
 \text{For server, first request arrives at} \\
 \text{client time + transmission time} &= 6 + 3 = 9
 \end{aligned}$$

To compute the time taken by server 13

$$\begin{aligned}
 &= \text{local O.S. + unmarshal + compute reply} \\
 &= 0.5 + 0.5 + 10 = 11
 \end{aligned}$$

∴ First request is completed by
9 + 11 = 20 milliseconds.

Extra time to marshall reply & to for local OS
 $= 0.5 + 0.5 = 1$

So the first request is ready to be sent back to the client by 21 milliseconds.

Since we have two threads by this time second request has already arrived & is ready to compute.

Time to compute second request is =
compute reply + marshall reply + local OS send.

$$= 10 + 0.5 + 0.5 = 11$$

$$\text{So far time taken} = 21 + 11 = 32$$

By this time the reply to the first request has reached the client

so the time taken to reach the client is.

transmission time + local OS + unmarshal response

$$= 3 + 0.5 + 0.5 = 4$$

: Total time taken for whole process =
 $32 + 4 = 36 \text{ milliseconds.}$

In the client server model implemented using a simple RPC mechanism, after making RPC request, a client keeps waiting until reply is received from the server for its request. It would be more efficient to allow the client to perform other jobs while the

server is processing the request. Describe three mechanisms that may be used in this case to allow a client to perform other jobs while the server is processing its request.

Synchronous RPC with diagram 3 marks

Asynchronous RPC with diagram 4 marks

Deferred RPC with diagram 3 marks

Asynchronous RPC

As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned. This strict request-reply behavior is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and have done useful work just after requesting the remote procedure to be called. Examples of where there is often no need to wait for a reply include: transferring money from one account to another, adding entries into a database, starting remote services, batch processing, and so on.

To support such situations, RPC systems may provide facilities for what are called **asynchronous RPCs**, by which a client immediately continues after issuing the **RPC request**. With asynchronous RPCs, the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure. The reply acts as an acknowledgement to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server's acknowledgement. Fig. 2-12(b) shows how client and server interact in the case of asynchronous RPCs. For comparison, Fig. 2-12(a) shows the normal request-reply behavior.

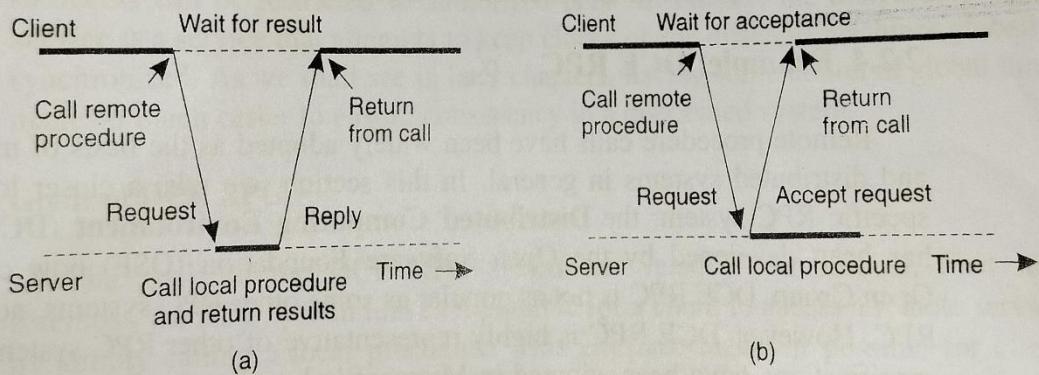


Figure 2-12. (a) The interaction between client and server in a traditional RPC.
(b) The interaction using asynchronous RPC.

Asynchronous RPCs can also be useful when a reply will be returned but the client is not prepared to wait for it and do nothing in the meantime. For example, a client may want to prefetch the network addresses of a set of hosts that it expects to contact soon. While a naming service is collecting those addresses, the client may want to do other things. In such cases, it makes sense to organize the communication between the client and server through two asynchronous RPCs, as shown in Fig. 2-13. The client first calls the server to hand over a list of host

names that should be looked up, and continues when the server has acknowledged the receipt of that list. The second call is done by the server, who calls the client to hand over the addresses it found. Combining two asynchronous RPCs is sometimes also referred to as a **deferred synchronous RPC**.

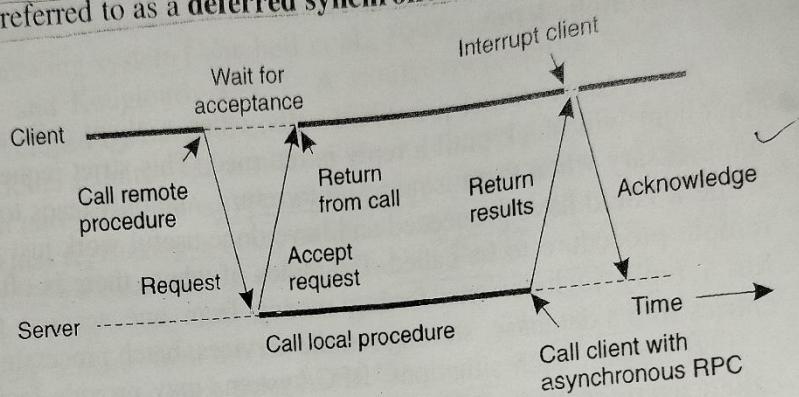
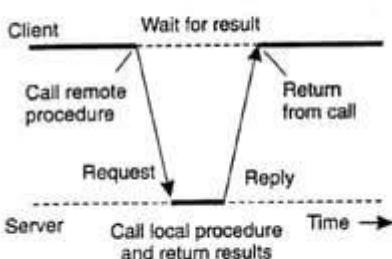


Figure 2-13. A client and server interacting through two asynchronous RPCs.

It should be noted that variants of asynchronous RPCs exist in which the client continues immediately after sending the request to the server. In other words, the client does not wait for an acknowledgement of the server's acceptance of the request. We refer to such RPCs as **one-way RPCs**. The problem with this approach is that if reliability is not guaranteed, the client cannot know for sure whether its request will be processed. We return to these matters in Chap. 7.

Synchronous RPC



- **Definition of RPC:**

- Software communication protocol for requesting services from a program on another computer.
- Client-server model, where the client initiates the request, and the server provides the service.
- Synchronous operation with the use of lightweight processes or threads for concurrent execution.

- **IDL in RPC:**

- Interface Definition Language (IDL) describes the software component's API.
- Acts as a bridge between machines using different operating systems and languages.

- **How RPC Works:**

- Compilation includes a stub in the code representing the remote procedure.
- Client runtime program contacts a name server to find the server's address.
- Request is sent to the server through the network, and response-request protocols handle the return.

- **Execution Flow in RPC:**

- Calling environment is suspended during the remote procedure call.
- Procedure parameters are transferred to the remote environment for execution.
- Results are sent back to the calling environment, and execution resumes as if from a local procedure call.

- **Example:**

- A client program offloads a complex computation to a server using RPC.
- RPC handles communication, execution, and result retrieval seamlessly.

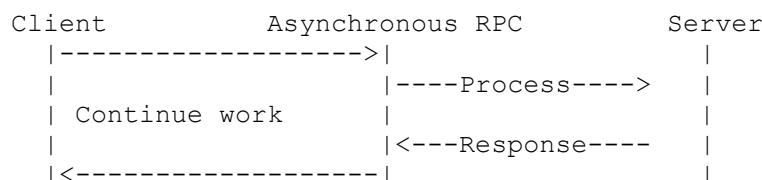
- **Opinion:**

- RPC simplifies distributed computing, abstracting network complexities.
- Allows developers to focus on functionality rather than intricate network details.

Asynchronous RPC

Asynchronous RPC allows the client to make a request and then continue with other tasks without waiting for the server's response. The client can later check for the server's response or be notified when it is ready. This mechanism improves efficiency by not blocking the client's execution.

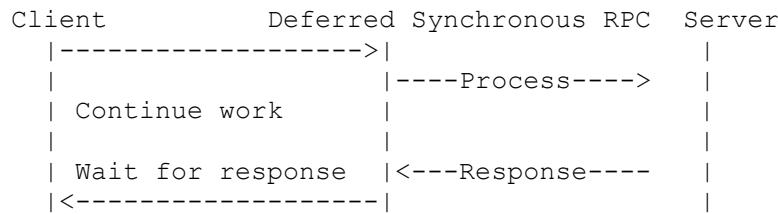
Diagram:



Deferred Synchronous RPC

Deferred Synchronous RPC is a hybrid approach where the client sends a request and continues with its execution. The client defers the waiting for the server's response to a later point in time when the result is needed. This allows the client to perform other jobs while the server processes the request.

Diagram:



These mechanisms provide different ways to handle the waiting time associated with RPCs, allowing clients to utilize their time more efficiently while waiting for server responses. [For more detailed information on these RPC mechanisms, you can refer to the resources provided¹²³.](#)

What actions to be taken with respect to the references to local resources when migrating the code to another machine? Justify these actions with example.

Resource-to machine binding			
Process-to-resource binding	Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)
	By value	CP (or MV, GR)	GR (or CP)
	By type	RB (or GR, CP)	RB (or GR, CP)

- Actions to be taken with respect to the references to local resources when migrating code to another machine.
- GR: establish global system-wide reference
- MV: move the resources
- CP: copy the resource

Process to resource binding and resource to machine binding table with actions 4 marks

Explanations with example 6 marks

3.4.2 Migration and Local Resources ESE

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult, is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else. An example of such a binding by identifier is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address. In the same line of reasoning, references to local communication endpoints also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated. In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment.

Finally, fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. An example of a fixed resource is a local communication endpoint. Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-14.

		Resource-to-machine binding		
		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
 MV Move the resource
 CP Copy the value of the resource
 RB Rebind process to locally available resource

Figure 3-14. Actions to be taken with respect to the references to local resources when migrating code to another machine.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to establish a global reference.

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy, is when migrating a process that is making use of a local communication endpoint. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and

install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change *their* global reference, and send messages to the new communication endpoint at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement distributed shared memory mechanisms as discussed in Chap. 1. Obviously, this is not really a viable solution.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding, the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

When migrating code to another machine in a distributed system, it's essential to address the references to local resources to ensure the code functions correctly in the new environment. Here are the actions to be taken with respect to the references to local resources, along with justifications and examples:

Actions and Justifications:

- 1. Global Reference Establishment (GR):** Establish a global system-wide reference for resources that can be accessed globally. This ensures that the migrated code can refer to resources regardless of its physical location.

Example: If the code interacts with a web service, update the service's URL to a global endpoint that the migrated code can access from anywhere.

2. **Resource Movement (MV):** Move unattached resources that are exclusively used by the program being migrated. This action is taken to maintain the integrity of the data and the functionality of the code.

Example: If the code relies on specific data files, these files should be moved to the new machine to ensure the code continues to operate without modification.

3. **Resource Copying (CP):** Copy the value of resources that cannot be moved but can be duplicated. This allows the migrated code to function with a local copy of the necessary resources.

Example: Configuration files or libraries that are required for the code to run can be copied to the target machine.

4. **Rebinding (RB):** Rebind process references to locally available resources for fixed resources that cannot be moved. This action is necessary to adapt the code to the new environment's resources.

Example: If the code references a local printer for output, the reference should be updated to point to a printer available on the new machine's network.

Detailed Explanation with Examples:

- **Global Reference Establishment (GR):** For resources like databases or APIs that are accessible over the network, the code should be updated to use the global references. This might involve changing local database connection strings to point to a globally accessible database server.

Justification: Using global references allows the code to remain functional and agnostic to the underlying infrastructure changes.

- **Resource Movement (MV):** When the code relies on specific files or executables that are not tied to the original machine, these should be moved along with the code. This ensures that the dependencies of the code are met on the new machine.

Justification: Moving resources maintains the operational context of the code, preventing errors due to missing files or executables.

- **Resource Copying (CP):** In cases where resources like configuration files or environment-specific data cannot be moved, creating a copy on the target machine is necessary. The code can then be adjusted to use these local copies.

Justification: Copying resources ensures that the code has access to the necessary data and configurations without disrupting the original system's setup.

- **Rebinding (RB):** For resources that are inherently local, such as printers, scanners, or other peripherals, the code must be updated to interact with the equivalent resources available on the new machine.

Justification: Rebinding allows the code to integrate with the new environment seamlessly, utilizing the available local resources effectively.

By taking these actions, the migrated code can continue to function correctly and efficiently in the new environment, leveraging the available resources while maintaining its original functionality¹². For more detailed information on code migration and handling local resources, you can refer to the resources provided¹².
These resources will give you a comprehensive understanding of the concepts and their practical applications in real-world distributed systems.

Suppose there are three processes A, B and C. All clock runs at the same rate but initially A's clock reads 10, b's clock reads 0 and C's reads 5. At time 10 by A's clock, A sends message to B, this message takes 4 units of time to reach B. B then waits one unit of time and then sends a message onto C which takes 2 units of time to reach C. Assuming that the system implements Lamport's timestamps draw a picture illustrating the timestamps for the message and explain how the timestamps are obtained.

Diagram 4 marks

First message 3 marks

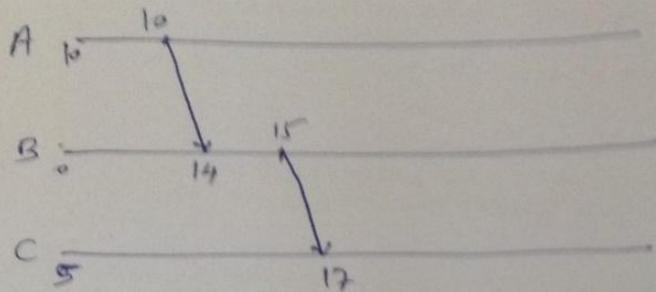
Second message 3 marks

Initially

$$A \rightarrow 10$$

$$B \rightarrow 0$$

$$C \rightarrow 5$$



First message

A sends message at 10

$$\text{Time} + B = 0$$

When message arrives at B

$$\begin{aligned}\text{Time at } B &= \max(A, B) + \text{comm. time.} \\ &= \max(10, 0) + 4 \\ &= 10 + 4 \\ &= 14 \text{ units.}\end{aligned}$$

Second message

Time at B = 14 but it waits one time unit &
then send.

∴ after B sends message at 15

$$\text{Initial Time} + C = 5$$

∴ It message received at C.

$$\begin{aligned}\text{The Time at } C &= \max(B, C) + \text{comm. time.} \\ &= \max(15, 5) + 2 \\ &= 15 + 2 \\ &= 17.\end{aligned}$$

What are the reasons of distributed systems are being more popular and useful?

Distributed systems have become increasingly popular and useful for several reasons:

1. **Resource Sharing:** They enable the sharing of hardware, software, and data resources across different locations¹.
2. **Scalability:** Distributed systems can easily scale to accommodate more users and resources, improving the responsiveness of the system¹.
3. **Fault Tolerance:** They are designed to continue operating properly even if some components fail, ensuring reliability¹.
4. **Concurrency:** Distributed systems naturally support concurrent processing, allowing multiple activities or functionalities to be performed simultaneously by separate users in different locations¹.
5. **Improved Performance:** By dividing tasks among multiple machines, distributed computing can make better use of resources and complete tasks faster, especially with large data sets².
6. **Flexibility:** They allow for more flexible system configurations and can be adapted to a wide range of applications and use cases³.
7. **Cost-Effectiveness:** Distributed systems can be more cost-effective than centralized systems, as they can leverage existing infrastructure and reduce the need for expensive, high-powered machines².
8. **Geographical Distribution:** They support applications that are inherently distributed, such as social media platforms, where information is shared among geographically dispersed users¹.
9. **Transparency:** To the end-user, distributed systems appear as a single, unified system, providing a seamless experience despite the complex underlying architecture².

These reasons contribute to the growing trend of applications migrating to the cloud, where distributed systems offer improved performance, scalability, and fault tolerance compared to traditional centralized systems⁴. For more detailed information on the advantages of distributed systems, you can refer to the resources provided¹²⁴³.

1.1 DEFINITION OF A DISTRIBUTED SYSTEM ESF

Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others. For our purposes it is sufficient to give a loose characterization:

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

This definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think they are dealing with a single system. Both are essential. We will come back to these points later in this chapter after going over some background material on both the hardware and the software.

Instead of going further with definitions, it is perhaps more useful to concentrate on important characteristics of distributed systems. One important characteristic is that differences between the various computers and the ways in which they communicate are hidden from users. The same holds for the internal organization of the distributed system. Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

Distributed systems should also be relatively easy to expand or scale. This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers actually take part in the system as a whole. A distributed system will normally be continuously available, although perhaps certain parts may be temporarily out of order. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications.

To support heterogeneous computers and networks while offering a single system view, distributed systems are often organized by means of a layer of software that is logically placed between a higher-level layer consisting of users and

What is client centric consistency model? The mobile user accessing different replicas of distributed database apply the eventual consistency with respect to this scenario and explain.

Client centric consistency model 2 marks

Eventual consistency with some scenario 4 marks

Explanation 4 marks

6.3 CLIENT-CENTRIC CONSISTENCY MODELS

The consistency models described in the previous section aim at providing a systemwide consistent view on a data store. An important assumption is that concurrent processes may be simultaneously updating the data store, and that it is necessary to provide consistency in the face of such concurrency. For example, in the case of object-based entry consistency, the data store guarantees that when an object is invoked, the invoking process is provided with a copy of the object that reflects all changes to the object that have been made so far, possibly by other processes. During the invocation, it is also guaranteed that no other process can interfere, that is, mutual exclusive access is provided to the invoking process.

Being able to handle concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, sequential consistency may possibly be guaranteed only when processes use synchronization mechanisms such as transactions or locks.

In this section, we take a look at a special class of distributed data stores. The data stores we consider are characterized by the lack of simultaneous updates, or when such updates happen, they can easily be resolved. Most operations involve reading data. These data stores offer a very weak consistency model, called eventual consistency. By introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

6.3.1 Eventual Consistency ESE

To what extent processes actually operate in a concurrent fashion, and to what extent consistency needs to be guaranteed, may vary. There are many examples in which concurrency appears only in a restricted form. For example, in many database systems, most processes hardly ever perform update operations but only read data from the database. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes.

As another example, consider a worldwide naming system such as DNS. The DNS name space is partitioned into domains, where each domain is assigned to a naming authority, which acts as owner of that domain. Only that authority is allowed to update its part of the name space. Consequently, conflicts resulting from two operations that both want to perform an update on the same data never occur (i.e., write-write conflicts). The only situation that needs to be handled are read-write conflicts. As it turns out, it is often acceptable to propagate an update in a lazy fashion, meaning that a reading process will see an update only after some time has passed since the update took place.

Yet another example is the World Wide Web. In virtually all cases, Web pages are updated by a single authority, such as a webmaster or the actual owner of the page. There are normally no write-write conflicts to resolve. On the other hand, to improve efficiency, browsers and Web proxies are often configured to keep a fetched page in a local cache, and to return that page upon the next request. An important aspect of both types of Web caches is that they may return out-of-date Web pages. In other words, the cached page that is returned to the requesting client is an older version compared to the one available at the actual Web server. As it turns out, many users find this inconsistency acceptable.

These examples can be viewed as cases of (large-scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent. This form of consistency is called **eventual consistency**.

Data stores that are eventually consistent thus have the property that in the absence of updates, all replicas converge toward identical copies of each other. As we will discuss later in this chapter, eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas. Write-write conflicts are often relatively easy to solve when assuming that only a small group of processes can perform updates. Eventual consistency is therefore often cheap to implement. Specific implementation details will be discussed later in this chapter.

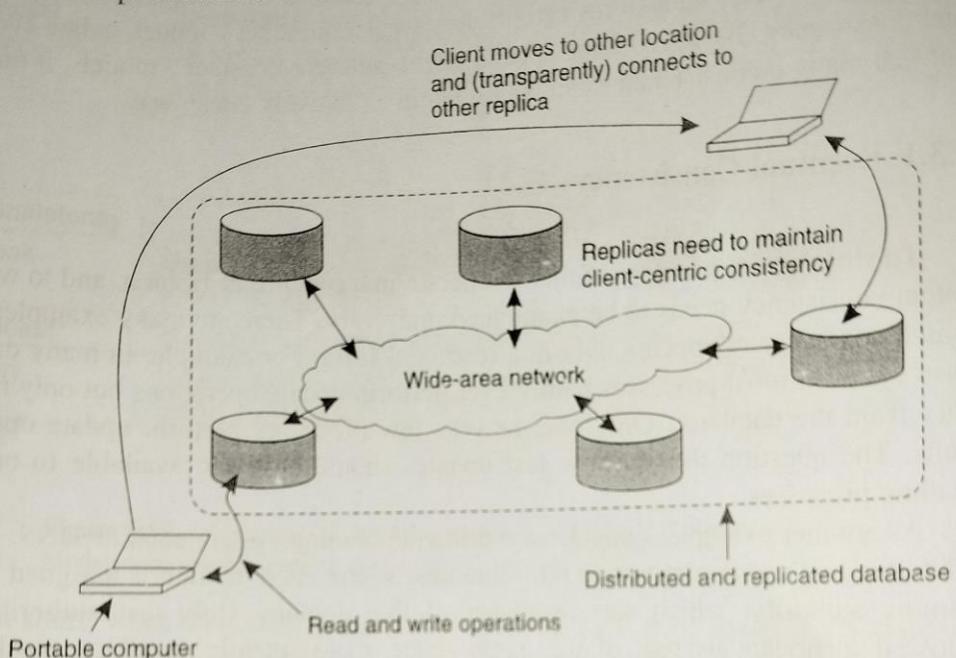


Figure 6-19. The principle of a mobile user accessing different replicas of a distributed database.

Eventual consistent data stores work fine as long as clients always access the same replica. However, problems arise when different replicas are accessed. This is best illustrated by considering a mobile user accessing a distributed database as shown in Fig. 6-19.

The mobile user accesses the database by connecting to one of the replicas in a transparent way. In other words, the application running on the user's portable computer is unaware on which replica it is actually operating. Assume the user performs several update operations and then disconnects again. Later, he accesses the database again, possibly after moving to a different location or by using a different access device. At that point, the user may be connected to a different replica than before, as shown in Fig. 6-19. However, if the updates performed previously have not yet been propagated, the user will notice inconsistent behavior. In particular, he would expect to see all previously-made changes, but instead, it appears as if nothing at all has happened.

This example is typical for eventually consistent data stores and is caused by the fact that users may sometimes operate on different replicas. The problem can be alleviated by introducing **client-centric consistency**. In essence, client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client. No guarantees are given concerning concurrent accesses by different clients.

Client-centric consistency models originate from the work on Bayou (see, for example, Terry et al., 1994; and Terry et al., 1998). Bayou is a database system developed for mobile computing, where it is assumed that network connectivity is unreliable and subject to various performance problems. Wireless networks and networks that span large areas, such as the Internet, fall into this category.

Bayou essentially distinguishes four different consistency models. To explain these models, we again consider a data store that is physically distributed across multiple machines. When a process accesses the data store, it generally connects to the locally (or nearest) available copy, although, in principle, any copy will do just fine. All read and write operations are performed on that local copy. Updates are eventually propagated to the other copies. To simplify matters, we assume that data items have an associated owner, which is the only process that is permitted to modify that item. In this way, we avoid write-write conflicts.

Client-centric consistency models are described using the following notations. Let $x_i[t]$ denote the version of data item x at local copy L_i at time t . Version $x_i[t]$ is the result of a series of write operations at L_i that took place since initialization. We denote this set as $WS(x_i[t])$. If operations in $WS(x_i[t_1])$ have also been performed at local copy L_j at a later time t_2 , we write $WS(x_i[t_1];x_j[t_2])$. If the ordering of operations or the timing is clear from the context, the time index will be omitted.

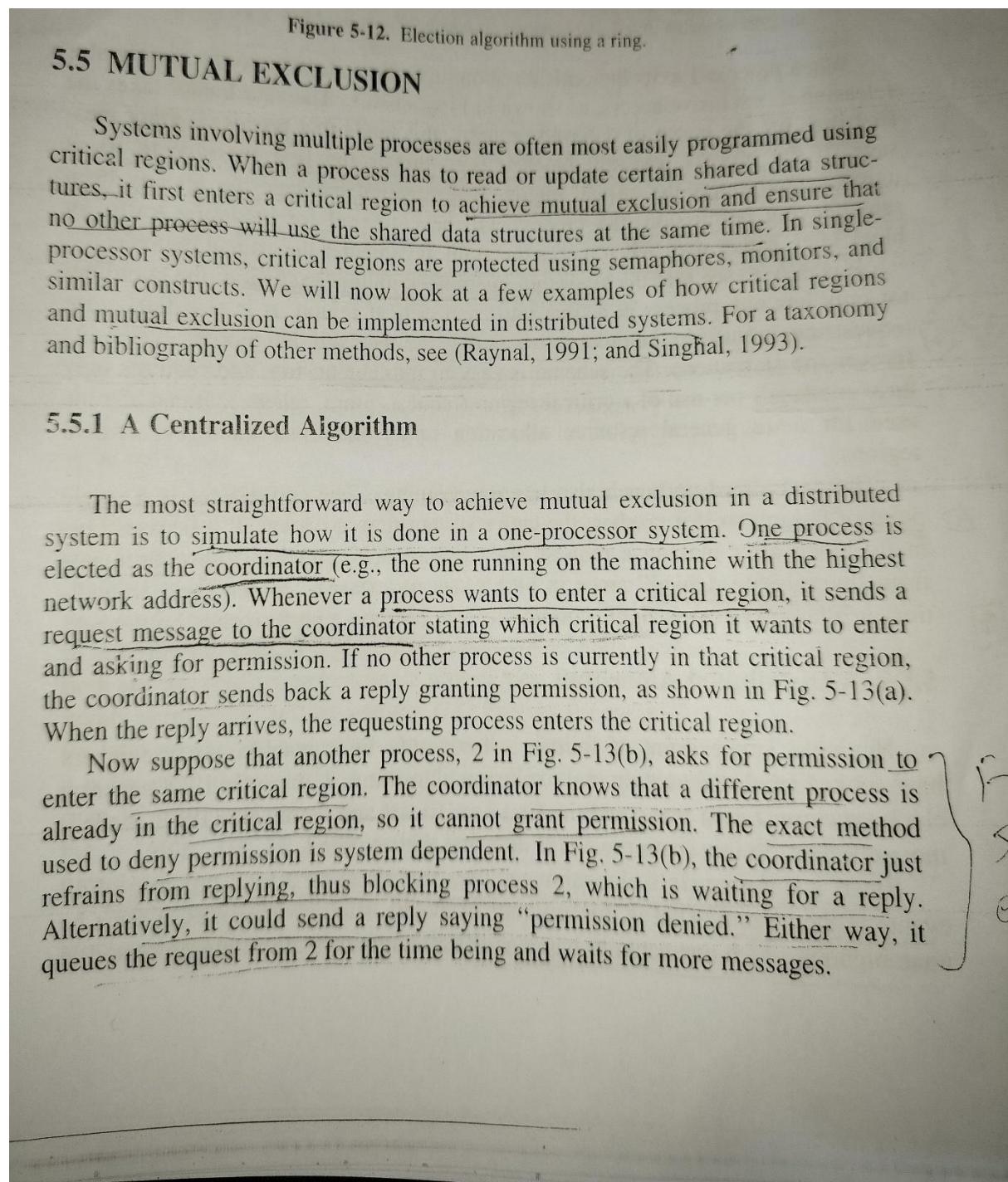
How mutual exclusion algorithm satisfy the requirement of mutual exclusion and starvation using centralized approach and distributed approach? Give the example of centralized mutual exclusion algorithm for 3 processes and one coordinator process showing request, reply, and release messages. How it is advantages to use a timestamp for distributed mutual exclusion? Justify your answer using 4 processes with timestamp for distributed mutual exclusion.

Requirement of Mutual exclusion using centralized and distributed approach 2 marks

Centralized mutual exclusion diagram with explanation 3 marks

distributed mutual exclusion diagram with explanation 3 marks

advantages 2 marks



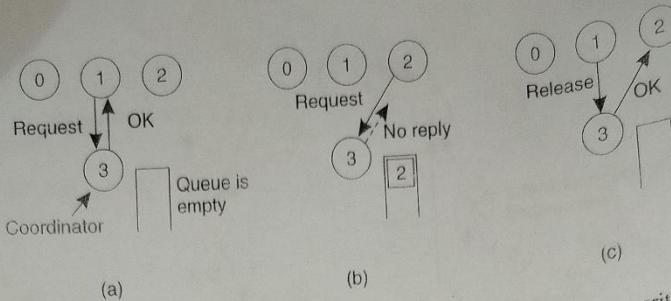


Figure 5-13. (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. 5-13(c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

5.5.2 A Distributed Algorithm ESE

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport's 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method.

Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first. Lamport's algorithm presented in

Sec. 5.2.1 is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 5-14(a).

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 5-14(b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig. 5-14(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig. 5-14 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and

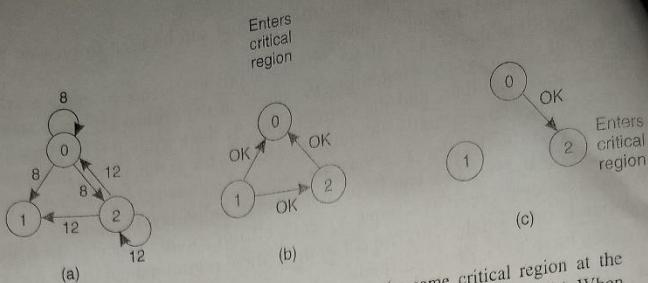


Figure 5-14. (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now $2(n - 1)$, where the total number of processes in the system is n . Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.

The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If

one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more intricate.

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust than the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

5.5.3 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. 5-15. Here we have a bus network, as shown in Fig. 5-15(a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig. 5-15(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process k to process $k+1$ (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around

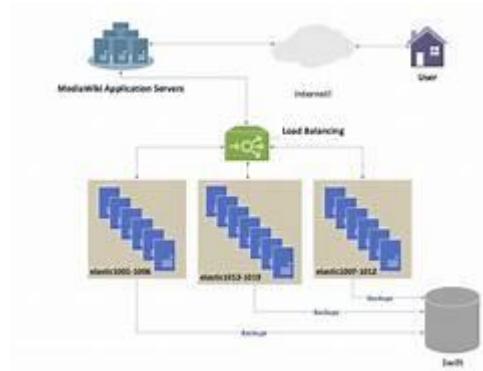
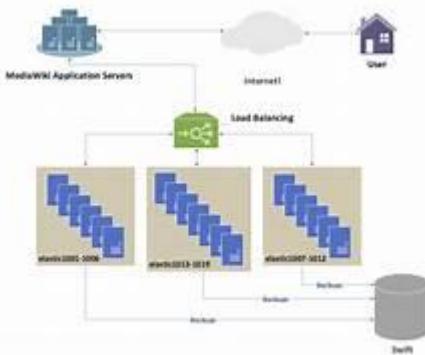
Suppose you have decided to use the high-low policy as the process transfer policy of a load balancing algorithm for distributed system. Select the suitable method that you will use in your implementation for choosing high mark and low mark

values. Do these threshold values have to be same for all processors in the system? Give reasons for your answer.

Threshold policy for process transfer 6 marks

Selecting the threshold 2 marks

Reasons 2 marks



Explore

Threshold Policy for Process Transfer

The **high-low policy**, also known as the **double-threshold policy**, is a method used in load balancing algorithms for distributed systems to decide when to transfer processes from heavily loaded nodes to lightly loaded nodes¹. The policy uses two threshold values:

- **High Mark:** If a node's load exceeds this value, it attempts to transfer processes to a less loaded node.
- **Low Mark:** If a node's load is below this value, it can accept more processes.

The goal is to maintain the system's load between these two marks, avoiding both underutilization and overloading of nodes.

Selecting the Threshold

When implementing the high-low policy, the method for choosing high mark and low mark values can be either **static** or **dynamic**:

- **Static Method:** Predefined threshold values are set for each node based on its processing capabilities. This method does not adapt to changes in the system's workload.
- **Dynamic Method:** Threshold values are calculated based on the average workload and a predefined constant. This method allows the system to adapt to varying workloads and can provide better load balancing.

Reasons for Threshold Values Variance

Threshold values **do not have to be the same** for all processors in the system. Here are the reasons:

- **Differing Capabilities:** Processors may have different computational capabilities, memory sizes, and I/O capacities. [Setting the same threshold for all would not account for these differences²](#).
- **Workload Variability:** Different nodes may handle different types of workloads. Some may be more CPU-intensive, while others may be I/O-bound. Tailoring thresholds to the type of workload can improve efficiency.
- **System Dynamics:** In a dynamic system, the workload can change over time. A static, uniform threshold may not be optimal as it cannot adapt to these changes.

[By setting individualized threshold values, the load balancing algorithm can more effectively distribute the workload according to the unique characteristics and current state of each processor in the distributed system¹](#). This approach can lead to better overall system performance and resource utilization.

MSE OCT 22

What is the scalability in distributed systems? Which are the different dimensions to measure the scalability of distributed system? What are the scalability problems and solutions/techniques? Explain with example.

ESR INTRODUCTION

1.2.4 Scalability ability to handle increased workload for extending system by repeatedly applying a cost-eff. strategy Worldwide connectivity through the Internet is rapidly becoming at least as common as being able to send a postcard to anyone anywhere around the world. With this in mind, scalability is one of the most important design goals for developers of distributed systems.

Scalability of a system can be measured along at least three different dimensions (Neuman, 1994). First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.

Scalability Problems

When a system needs to scale, very different types of problems need to be solved. Let us first consider scaling with respect to size. If more users or resources need to be supported, we are often confronted with the limitations of centralized services, data, and algorithms (see Fig. 1-3). For example, many services are centralized in the sense that they are implemented by means of only a single server running on a specific machine in the distributed system. The problem with this scheme is obvious: the server can simply become a bottleneck as the number of users grows. Even if we have virtually unlimited processing and storage capacity, communication with that server will eventually prohibit further growth.

Unfortunately, using only a single server is sometimes unavoidable. Imagine that we have a service for managing highly confidential information such as medical records, bank accounts, personal loans, and so on. In such cases, it may be best to implement that service by means of a single server in a highly secured separate room, and protected from other parts of the distributed system through special network components. Copying the server to several locations to enhance performance may be out of the question as it would make the service more vulnerable to security attacks.

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book -
Centralized algorithms	Doing routing based on complete information

Figure 1-3. Examples of scalability limitations.

Just as bad as centralized services are centralized data. How should we keep track of the telephone numbers and addresses of 50 million people? Suppose that

each data record could be fit into 50 characters. A single 2.5-gigabyte disk would provide enough storage. But here again, having a single database would undoubtedly saturate all the communication lines into and out of it. Likewise, imagine how the Internet would work if its Domain Name System (DNS) was still implemented as a single table. DNS maintains information on millions of computers worldwide and forms an essential service for locating Web servers. If each request to resolve a URL had to be forwarded to that one and only DNS server, it is clear that no one would be using the Web (which, by the way, would probably solve the problem again).

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

The trouble is that collecting and transporting all the input and output information would again be a bad idea because these messages would overload part of the network. In fact, any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results must be avoided. Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally or internationally is tricky.

Geographical scalability has its own problems. One of the main reasons why it is currently hard to scale existing distributed systems that were designed for local-area networks is that they are based on synchronous communication. In this form of communication, a party requesting service, generally referred to as a client, blocks until a reply is sent back. This approach generally works fine in LANs where communication between two machines is generally at worst a few

hundred microseconds. However, in a wide-area system, we need to take account that interprocess communication may be hundreds of milliseconds, three orders of magnitude slower. Building interactive applications using synchronous communication in wide-area systems requires a great deal of care (and not a little patience).

Another problem that hinders geographical scalability is that communication in wide-area networks is inherently unreliable, and virtually always point-to-point. In contrast, local-area networks generally provide highly reliable communication facilities based on broadcasting, making it much easier to develop distributed systems. For example, consider the problem of locating a service. In a local-area system, a process can simply broadcast a message to every machine, asking if it is running the service it needs. Only those machines that have that service respond, each providing its network address in the reply message. Such a location scheme is unthinkable in a wide-area system. Instead, special location services need to be designed, which may need to scale worldwide and be capable of servicing a billion users. We return to such services in Chap. 4.

Geographical scalability is strongly related to the problems of centralized solutions that hinder size scalability. If we have a system with many centralized components, it is clear that geographical scalability will be limited due to the performance and reliability problems resulting from wide-area communication. In addition, centralized components now lead to a waste of network resources. Imagine that a single mail server is used for an entire country. This would mean that sending an e-mail to your neighbor would first have to go to the central mail server, which may be hundreds of miles away. Clearly, this is not the way to go.

Finally, a difficult, and in many cases open question is how to scale a distributed system across multiple, independent administrative domains. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security.

For example, many components of a distributed system that reside within a single domain can often be trusted by users that operate within that same domain. In such cases, system administration may have tested and certified applications, and may have taken special measures to ensure that such components cannot be tampered with. In essence, the users trust their system administrators. However, this trust does not expand naturally across domain boundaries.

If a distributed system expands to another domain, two types of security measures need to be taken. First, the distributed system has to protect itself against malicious attacks from the new domain. For example, users from the new domain may have only read access to the system's file service in its original domain. Likewise, facilities such as expensive image setters or high-performance computers may not be made available to other users. Second, the new domain has to protect itself against malicious attacks from the distributed system. A typical example is that of downloading programs such as applets in Web browsers. Basically, the new domain does not know what to expect from such foreign code, and

may therefore decide to severely limit the access rights for such code. The problem, as we shall see in Chap. 8, is how to enforce those limitations.

Scaling Techniques

Having discussed some of the scalability problems brings us to the question of how those problems can generally be solved. Because scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network, there are basically only three techniques for scaling: hiding communication latencies, distribution, and replication (see also Neuman, 1994).

Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester's side. Essentially, this means constructing the requesting application in such a way that it uses only **asynchronous communication**. When a reply comes in, the application is interrupted and a special handler is called to complete the previously issued request. Asynchronous communication can often be used in batch-processing systems and parallel applications, in which more or less independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

However, there are many applications that cannot make effective use of asynchronous communication. For example, in interactive applications when a user sends a request he will generally have nothing better to do than to wait for the answer. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. A typical case where this approach works is accessing databases using forms. Normally, filling in forms is done by sending a separate message for each field, and waiting for an acknowledgement from the server, as shown in Fig. 1-4(a). For example, the server may check for syntactic errors before accepting an entry. A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form, as shown in Fig. 1-4(b). This approach of shipping code is now widely supported by the Web in the form of Java applets.

Another important scaling technique is **distribution**. Distribution involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. A good example of distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organized into a tree of **domains**, which are divided into **nonoverlapping zones**, as shown in Fig. 1-5. The names in each zone are handled by a single name server. Without

INTRODUCTION

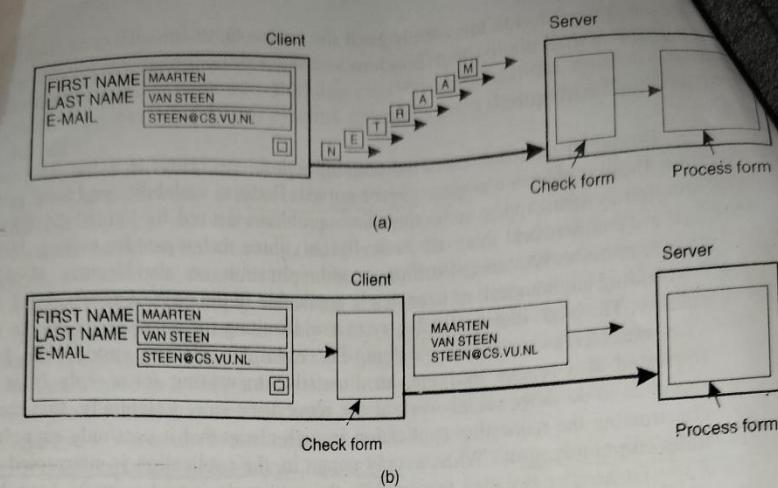


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

going into too many details, one can think of each path name being the name of a host in the Internet, and is thus associated with a network address of that host. Basically, resolving a name means returning the network address of the associated host. Consider, for example, the name *nl.vu.cs.flits*. To resolve this name, it is first passed to the server of zone *Z1* (see Fig. 1-5) which returns the address of the server for zone *Z2*, to which the rest of name, *vu.cs.flits*, can be handed. The server for *Z2* will return the address of the server for zone *Z3*, which is capable of handling the last part of the name and will return the address of the associated host.

This examples illustrates how the *naming service*, as provided by DNS, is distributed across several machines, thus avoiding that a single server has to deal with all requests for name resolution.

As another example, consider the World Wide Web. To most users, the Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL. Conceptually, it may even appear as if there is only a single server. However, the Web is physically distributed across a large number of servers, each handling a number of Web documents. The name of the server handling a document is encoded into that document's URL. It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

✓ Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually replicate components across a

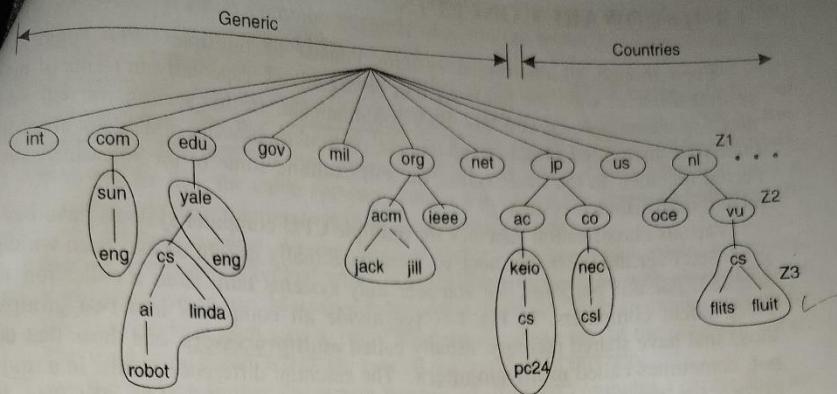


Figure 1-5. An example of dividing the DNS name space into zones.

distributed system. Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before.

Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource, and not by the owner of a resource.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to consistency problems.

To what extent inconsistencies can be tolerated depends highly on the usage of a resource. For example, many Web users find it acceptable that their browser returns a cached document of which the validity has not been checked for the last few minutes. However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges. The problem with strong consistency is that an update must be immediately propagated to all other copies. Moreover, if two updates happen concurrently, it is often also required that each copy is updated in the same order. Situations such as these generally require some global synchronization mechanism. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way. Consequently, scaling by replication may introduce other, inherently nonscalable solutions. We return to replication and consistency in Chap. 6.

An experimental file server is up 75% of the time and down for 25% of the time due to bugs. How many times does this file server have to be replicated to give an availability of at least 99% ?

- A. 2
- B. 4
- C. 8
- D. 16

Lets us check each option

- a) if there are 2 replications, probability of failing both at same time = $.25 \times .25 = .0625 = 6.25\%$
Availability = $100 - 6.25 = 93\%$ //Not the answer
- b) In case of 4 replications probability of failing all at same time = $(0.25)^4 = 0.0039 = 0.39\%$
Availability = $100 - 0.39 \sim 99\%$ //Hence the answer

No need to check furthur as all options are more than 4

Answer is **B**

What is meant by transparency in distributed systems? What are the kinds of transparency exist in distributed systems? Explain each one with examples

an actually prove you physically possess the card by using a card reader.

Another security problem is that of tracking communication to build up a preference profile of a specific user (Wang et al., 1998). Such tracking explicitly violates privacy, especially if it is done without notifying the user. A related problem is that increased connectivity can also lead to unwanted communication, such as electronic junk mail often called spam. In such cases, what we may need is to protect ourselves using special information filters that select incoming messages based on their content.

1.2.2 Transparency ESE

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be **transparent**. Let us first take a look at what kinds of transparency exist in distributed systems, and then address the question whether transparency is always required.

Transparency in a Distributed System

The concept of transparency can be applied to several aspects of a distributed system, as shown in Fig. 1-2.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Figure 1-2. Different forms of transparency in a distributed system (ISO, 1995).

Access transparency deals with hiding differences in data representation and the way that resources can be accessed by users. For example, to send an integer

from an Intel-based workstation to a Sun SPARC machine requires that we take into account that Intel orders its bytes in **little endian** format (i.e., the high-order byte is transmitted first), and that the SPARC processor uses **big endian** format (i.e., the low-order byte is transmitted first). Other differences in data representation may exist as well. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.

An important group of transparency types has to do with the location of a resource. **Location transparency** refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of such a name is the URL <http://www.prenhall.com/index.html>, which gives no clue about the location of Prentice Hall's main Web server. The URL also gives no clue as to whether *index.html* has always been at its current location or was recently moved there. Distributed systems in which resources can be moved without affecting how that resource can be accessed are said to provide **migration transparency**. Even stronger is the situation in which resources can be relocated while they are being accessed without the user or application noticing anything. In such cases, the system is said to support **relocation transparency**. An example of relocation transparency is when mobile users can continue to use their wireless laptop while moving from place to place without ever being (temporarily) disconnected.

As we shall see, replication plays an important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. **Replication transparency** deals with hiding the fact that several copies of a resource exist. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called **concurrency transparency**. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use

of transactions, but as we shall see in later chapters, transactions are difficult to implement in distributed systems.

A popular alternative definition of a distributed system, due to Leslie Lamport, is "You know you have one when the crash of a computer you've never heard of stops you from getting any work done." This description puts the finger on another important issue of distributed systems design: dealing with failures. Making a distributed system **failure transparent** means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made, as we will discuss in Chap. 7. The main difficulty in masking failures lies in the inability to distinguish between a dead resource and a painfully slow resource. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot conclude that the server is really down.

The last type of transparency that is often associated with distributed systems is **persistence transparency**, which deals with masking whether a resource is in volatile memory or perhaps somewhere on a disk. For example, many object-oriented databases provide facilities for directly invoking methods on stored objects. What happens behind the scenes, is that the database server first copies the object's state from disk to main memory, performs the operation, and perhaps writes that state back to secondary storage. The user, however, is unaware that the server is moving state between primary and secondary memory. Persistence plays an important role in distributed systems, but it is equally important for nondistributed systems.

Degree of Transparency

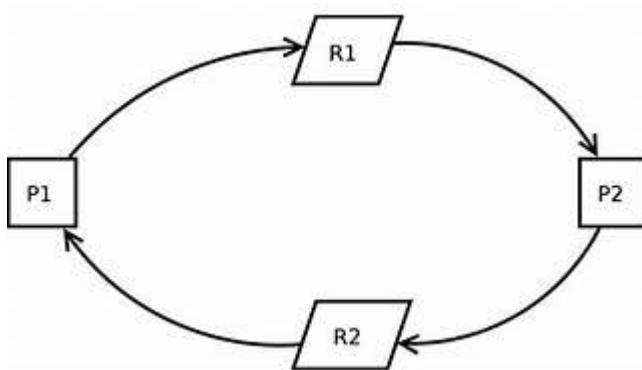
Although distribution transparency is generally preferable for any distributed system, there are situations in which attempting to blindly hide all distribution aspects from users is not always a good idea. An example is requesting your electronic newspaper to appear in your mailbox before 7 A.M. local time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.

Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother Nature will not allow it to send a message.

ESE MAY 2021

What is a phantom deadlock? What might be the reason for phantom deadlock in a distributed system? Suppose that

centralized deadlock detection scheme in which the transfer on request method is used to transfer the information from the local coordinator to the central coordinator. Give an example to show that the algorithm may still detect a phantom deadlock.



A phantom deadlock in distributed systems refers to situations where multiple processes or threads get stuck and can't move forward because of conflicts in synchronization and resource allocation¹. This happens when different tasks are being executed at the same time, causing each process to wait for a resource that is held by another process¹. Unlike deadlocks, where resources are explicitly locked and blocked, phantom deadlocks often occur due to dependencies that are not immediately obvious¹.

Phantom deadlocks usually happen because of a combination of factors like synchronization mechanisms, strategies for allocating resources, and the inherent complexities of distributed systems¹. In distributed systems, multiple nodes may request resources from each other such as files, locks, or data to carry out tasks¹. Due to network delays and the asynchronous nature of communication, nodes might not receive responses to their resource requests¹. This can lead to communication delays¹. Sometimes nodes may lack time and comprehensive information about the system's resource allocation and utilization¹. As a result, they might misinterpret the information due to knowledge¹.

In a centralized deadlock detection scheme, one site is chosen as a control site. This site is responsible for deadlock detection². It has control over all resources of the system². If a site requires a resource, it requests the control site, the control site allocates and de-allocates resources, and maintains a wait-for graph². And at a regular interval of time, it checks the wait-for graph to detect a cycle². If the cycle exists then it will declare the system as deadlock otherwise the system will continue working².

Let's understand it better with an example of a phantom deadlock¹. In this diagram, the arrows show the relationships of waiting. For instance, P1 is waiting for R2, P2 is waiting for R3, and P3 is waiting for R1¹. This circular pattern suggests a deadlock¹. However, due to the asynchronous nature of distributed systems and the delay in propagating local information, the deadlock detection algorithm might identify a deadlock that does not really exist³. Such situations are called phantom deadlocks and they lead to unnecessary aborts³.

Why is heterogeneity unavoidable in many distributed systems?

What are some of common types of incompatibilities encountered in heterogeneous distributed systems? What are the common issues with which the designer of a heterogeneous distributed system must deal?

Heterogeneity is often unavoidable in distributed systems due to the diverse requirements of different applications¹. Hardware and software are generally tailored to a specific application area¹. For instance, supercomputers and multiprocessors excel at compute-intensive tasks but struggle with user interface and device I/O¹. On the other hand, personal computers and workstations are frequently equipped with excellent user interfaces¹. Many applications necessitate complex user interfaces, specific I/O devices, and large amounts of computational power¹. As a result, integrating diverse machines into a coherent distributed system and sharing resources among them is very desirable¹.

Common types of incompatibilities encountered in heterogeneous distributed systems include:

1. **Data Compatibility and Conversion:** Different byte-ordering and floating-point representations may be used by machines with different architectures¹. Data that is sent from one machine to another must be converted to the destination machine's format¹.
2. **Block Size Selection:** The data transmission unit (block) must be transformed according to the data type of its contents¹.

Designers of a heterogeneous distributed system must deal with several issues, including:

1. **Heterogeneity:** This applies to the network, computer hardware, operating system, and implementation of different developers².

2. **Openness**: The degree to which new resource-sharing services can be made available to the users².
3. **Scalability**: The system should remain efficient even with a significant increase in the number of users and resources connected².
4. **Security**: The security of an information system has three components: Confidentiality, integrity, and availability².
5. **Failure Handling**: Corrective measures should be implemented to handle cases when some faults occur in hardware and the software program².
6. **Concurrency**: There is a possibility that several clients will attempt to access a shared resource at the same time².
7. **Transparency**: The distributed system should be perceived as a single entity by the users or the application programmers rather than a collection of autonomous systems².

A server is designed to perform simple integer arithmetic operations (addition, subtraction, multiplication and division).

Clients interact with this server by using an RPC mechanism.

Describe the contents of the call and reply messages of this RPC application, explaining the purpose of each component.

In a Remote Procedure Call (RPC) mechanism, the client and server communicate through call and reply messages¹²³. Here's a description of the contents of these messages in the context of a server performing simple integer arithmetic operations:

Call Message:

1. **Transaction Identifier (xid)**: This is a unique identifier for each transaction⁴. It helps in matching the reply message to the corresponding call message².
2. **Message Type**: This indicates whether the message is a call or a reply⁴. In this case, it would be set to "CALL".
3. **Procedure Identifier**: This specifies the procedure to be invoked on the server. For example, it could be an identifier for addition, subtraction, multiplication, or division.
4. **Procedure Parameters**: These are the parameters required for the procedure. For arithmetic operations, these would be the integers on which the operation is to be performed¹.

Reply Message:

1. **Transaction Identifier (xid):** This matches the xid in the call message, allowing the client to correlate the reply with the original request².
2. **Message Type:** This indicates whether the message is a call or a reply⁴. In this case, it would be set to “REPLY”.
3. **Reply Status:** This indicates whether the procedure was executed successfully or not¹.
4. **Return Values:** If the procedure was executed successfully, this contains the result of the operation¹.

The purpose of these components is to facilitate communication between the client and the server, allowing the client to invoke procedures on the server and receive the results. The client and server may be on different systems, possibly with different architectures and data representation types¹. The RPC mechanism abstracts these details, making network communication appear as if it were a local procedure call⁵. This greatly simplifies the development of distributed applications⁶.

How would you incorporate persistence asynchronous communication into model of communication based on RMI to remote objects? In Java Remote object invocation how to pass object by reference or by value give an example with justification.

Incorporating persistent asynchronous communication into a model of communication based on Remote Method Invocation (RMI) involves a few steps¹²:

1. **Persistent Communication:** In persistent communication, messages are stored until the receiver is ready¹. This can be achieved by implementing a message queue or buffer at the sender’s end¹. When a message is sent, it is stored in this buffer until the receiver is ready to process it¹.
2. **Asynchronous Communication:** In asynchronous communication, the sender continues immediately after it has submitted the message¹. This can be achieved by implementing non-blocking methods or using separate threads for sending and receiving messages¹.

In Java RMI, objects can be passed either by value or by reference³⁴⁵⁶:

1. **Pass by Value:** If an object does not implement the `Remote` interface but is `Serializable`, then it will be serialized and passed by value³. This means a copy of the object is created and sent to the remote method³.

-
2. **Pass by Reference:** If an object implements the `Remote` interface, then it will be passed by reference³. This means a reference to the object (a stub) is sent to the remote method³. Any changes made to the object through this reference will affect the original object³.

Here's an example:

```
public interface MyService extends Remote {  
    public MyHandle createX(SomeSerializableObj param1) throws  
RemoteException;  
}  
  
public interface MyHandle extends Remote {  
    void doOne();  
    void doTwo();  
}
```

In this example, `SomeSerializableObj` is passed by value to the `createX` method, while `MyHandle` is passed by reference⁴. The `createX` method returns a reference to a `MyHandle` object, allowing the client to interact with that object on the server⁴.

processes operating on different distributed proxies by using synchronized methods. Instead, explicit distributed locking techniques will have to be used.

Java Remote Object Invocation ESE

As the distinction between local and remote objects is hardly visible at the language level, Java can also hide most of the differences during a remote method invocation. For example, any primitive or object type can be passed as a parameter to an RMI, provided that the type can be marshaled. In Java terminology, this means that it must be **serializable**. Although, in principle, most objects can be serialized, serialization is not always allowed or possible. Typically, platform-dependent objects such as file descriptors and sockets, cannot be serialized.

The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference. In other words, a local object is first copied after which the copy is used as parameter value. For a remote object, a reference to the object is passed as parameter instead of a copy of the object, as was also shown in Fig. 2-18.

In Java RMI, a reference to a remote object is essentially implemented as we explained in Sec. 2.3.2. Such a reference consists of the network address and endpoint of the server, as well as a local identifier for the actual object in the server's address space. That local identifier is used only by the server. As we also explained, a reference to a remote object also needs to encode the protocol stack that is used by a client and the server to communicate. To understand how such a stack is encoded in the case of Java RMI, it is important to realize that each object in Java is an instance of a class. A class, in turn, contains an implementation of one or more interfaces.

In essence, a remote object is built from two different classes. One class contains an implementation of server-side code, which we call the *server class*. This class contains an implementation of that part of the remote object that will be running on a server. In other words, it contains the description of the object's state, as well as an implementation of the methods that operate on that state. The server-side stub, that is, the skeleton, is generated from the interface specifications of the object.

The other class contains an implementation of the client-side code, which we call the *client class*. This class contains an implementation of a proxy. Like the skeleton, this class is also generated from the object's interface specification. In its simplest form, the only thing a proxy does is to convert each method call into a message that is sent to the server-side implementation of the remote object, and convert a reply message into the result if a method call. For each call, it sets up a connection with the server, which is subsequently torn down when the call is finished. For this purpose, the proxy needs the server's network address and endpoint

as mentioned above. This information, along with the local identifier of the object at the server, is always stored as part of the state of a proxy.

Consequently, a proxy has all the information it needs to let a client invoke methods of the remote object. In Java, proxies are serializable. In other words, it is possible to marshal a proxy and send it as a series of bytes to another process, where it can be unmarshaled and used to invoke methods on the remote object. In other words, a proxy can be used as a reference to a remote object.

This approach is consistent with Java's way of integrating local and distributed objects. Recall that in an RMI, a local object is passed by making a copy of it, while a remote object is passed by means of a systemwide object reference. A proxy is treated as nothing else but a local object. Consequently, it is possible to pass a serializable proxy as parameter in an RMI. The side effect is that such a proxy can be used as a reference to the remote object.

In principle, when marshaling a proxy, its complete implementation, that is, all its state and code, is converted to a series of bytes. Marshaling the code like this is not very efficient and may lead to very large references. Therefore, when marshaling a proxy in Java, what actually happens is that an implementation handle is generated, specifying precisely which classes are needed to construct the proxy. Possibly, some of these classes first need to be downloaded from a remote site. The implementation handle replaces the marshaled code as part of a remote-object reference. In effect, references to remote objects in Java are in the order of a few hundred bytes.

This approach to referencing remote objects is highly flexible and is one of the distinguishing features of Java RMI (Waldo, 1998). In particular, it allows for object-specific solutions. For example, consider a remote object whose state changes only once in a while. We can turn such an object into a truly distributed object by copying the entire state to a client at binding time. Each time the client invokes a method, it operates on the local copy. To ensure consistency, each invocation also checks whether the state at the server has changed, in which case the local copy is refreshed. Likewise, methods that modify the state are forwarded to the server. The developer of the remote object will now have to implement only the necessary client-side code, and have it dynamically downloaded when the client binds to the object.

Being able to pass proxies as parameters works only because each process is executing the same Java virtual machine. In other words, each process is running in the same execution environment. A marshaled proxy is simply unmarshaled at the receiving side, after which its code can be executed. In contrast, in DCE for example, passing stubs is out of the question, as different processes may be running in execution environments that differ with respect to language, operating system, and hardware. Instead, a DCE process first needs to (dynamically) link in a locally available stub that has been previously compiled specifically for the process's execution environment. By passing a reference to a stub as parameter in an RPC, it is possible to refer to objects across process boundaries.

How GIOP and IIOP used in CORBA? What is the role of ORB? Give the example of interface and implementation repository used in CORBA. What are the functions of interfaces

used in CORBA at client and server side?

In the Common Object Request Broker Architecture (CORBA), the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP) play crucial roles¹²:

1. **GIOP:** This is a general protocol that defines how Object Request Brokers (ORBs) communicate in CORBA¹. It's designed to be simple, scalable, low-cost, and general enough to be implemented on any connection-oriented transport protocol¹.
2. **IIOP:** This is a specific implementation of GIOP for use over the Internet². It provides a mapping between GIOP messages and the TCP/IP layer². All CORBA products are required to support IIOP¹.

The Object Request Broker (ORB) is a key component of CORBA³⁴⁵⁶⁷. It handles the communication, marshaling, and unmarshaling of parameters, making the process transparent for CORBA server and client applications³. The ORB also manages the Interface Repository, a distributed database containing IDL interface definitions³.

The Interface Repository provides persistent storage for interface definitions specified in OMG IDL⁸. It manages and provides access to a collection of object definitions⁸. The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects⁹.

Interfaces in CORBA specify a software boundary between a service implementation and its clients¹⁰¹¹. They can involve attributes, operations, and type definitions¹⁰¹¹. On the client side, the ORB offers services like providing interface definitions from the Interface Repository and constructing invocations for use with the Dynamic Invocation Interface³. On the server side, the ORB manages the activation and deactivation of objects, among other tasks³.

Suppose there are three processes A,B and C. All clock runs at the same rate but initially A's clock reads 10, B's clock reads 0 and C's clock reads 5. At time 10 by A's clock, A sends message to B, this message takes 4 units of time to reach B. B then waits one unit of time and then sends a message onto C which takes 2 units of time to reach C. Assuming that the system implements Lamport's timestamps draw a picture illustrating the timestamps for the message and explain how the timestamps are obtained.

Choose an example of an application for which causal consistency is the most suitable consistency model.

SEC. 6.2

DATA-CENTRIC CONSISTENCY MODELS

305

is t , then it is always true that $r + w \geq t$. In other words, for any sequentially consistent store, changing the protocol to improve read performance makes write performance worse, and vice versa. For this reason, researchers have investigated other (weaker) models. In the following sections we will discuss some of them.

6.2.3 Causal Consistency ESE

The causal consistency model (Hutto and Ahmad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. We already came across causality when discussing vector timestamps in the previous chapter. If event B is caused or influenced by an earlier event, A , causality requires that everyone else first see A , then see B .

Consider a memory example. Suppose that process P_1 writes a variable x . Then P_2 reads x and writes y . Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x read by P_2 (i.e., the value written by P_1). On the other hand, if two processes spontaneously and simultaneously write two different variables, these are not causally related. When there is a read followed later by a write, the two events are potentially causally related. Similarly, a read is causally related to the write that provided the data the read got. Operations that are not causally related are said to be concurrent.

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

As an example of causal consistency, consider Fig. 6-9. Here we have an event sequence that is allowed with a causally-consistent store, but which is forbidden with a sequentially consistent store or a strictly consistent store. The thing to note is that the writes $W_2(x)b$ and $W_1(x)c$ are concurrent, so it is not required that all processes see them in the same order.

P1:	W(x)a		W(x)c	
P2:	R(x)a	W(x)b		
P3:	R(x)a		R(x)c	R(x)b
P4:	R(x)a		R(x)b	R(x)c

Figure 6-9. This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

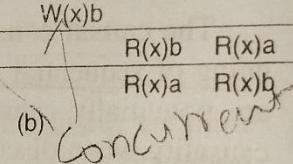
Now consider a second example. In Fig. 6-10(a) we have $W_2(x)b$ potentially depending on $W_1(x)a$ because the b may be a result of a computation involving

the value read by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order. Therefore, Fig. 6-10(a) is incorrect. On the other hand, in Fig. 6-10(b) the read has been removed, so $W_1(x)a$ and $W_2(x)b$ are now concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered, so Fig. 6-10(b) is correct.

P1:	$W(x)a$		
P2:		$R(x)a$	$W(x)b$
P3:			$R(x)b$
P4:			$R(x)a$

(a)

P1:	$W(x)a$		
P2:		$R(x)a$	$W(x)b$
P3:			$R(x)b$
P4:			$R(x)a$



(b)

Figure 6-10. (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.

Implementing causal consistency requires keeping track of which processes have seen which writes. It effectively means that a dependency graph of which operation is dependent on which other operations must be constructed and maintained. One way of doing this is by means of vector timestamps, as we discussed in the previous chapter. We return to the use of vector timestamps to capture causality later in this chapter.

How to construct message queuing model is used in message oriented persistent communication? What is the relationship between the queue level addressing and network level addressing is used in message queuing system?

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. In practice, most communication servers are directly connected to each other. In other words, a message is generally transferred directly to a destination server. In principle, each application has its own private queue to which other applications can send messages. A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue. No guarantees are given about when, or even if, the message will actually be read, which is completely determined by the behavior of the recipient.

These semantics permit loosely-coupled communication. There is thus no need for the receiver to be executing when a message is being sent to its queue. Likewise, there is no need for the sender to be executing when its message is picked up by the receiver. The sender and receiver can execute completely independently of each other. In fact, once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us four combinations with respect to the execution mode of the sender and receiver, as shown in Fig. 2-26.

In Fig. 2-26(a), both the sender and receiver are executing during the entire transmission of a message. In Fig. 2-26(b), only the sender is executing, while the receiver is passive, that is, in a state in which message delivery is not possible. Nevertheless, the sender can still send messages. The combination of a passive sender and an executing receiver is shown in Fig. 2-26(c). In this case, the receiver can read messages that were sent to it, but it is not necessary that their respective senders are executing as well. Finally, in Fig. 2-26(d), we see the situation that the system is storing (and possibly transmitting) messages even while sender and receiver are passive.

Messages can, in principle, contain any data. The only important aspect is that messages are properly addressed. In practice, addressing is done by providing a systemwide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications. An effect of this approach is that the basic interface offered to applications can be extremely simple, as shown in Fig. 2-27.

The put primitive is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. As we explained, this is a nonblocking call. The get primitive is a blocking call by which an authorized

Fig 2-26
expl

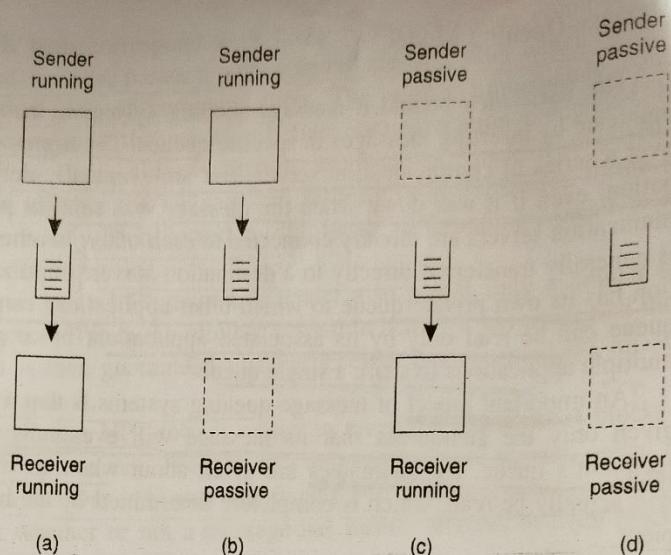


Figure 2-26. Four combinations for loosely-coupled communications using queues.

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 2-27. Basic interface to a queue in a message-queuing system.

process can remove the longest pending message in the specified queue. The process is blocked only if the queue is empty. Variations on this call allow searching for a specific message in the queue, for example, using a priority, or a matching pattern. The nonblocking variant is given by the poll primitive. If the queue is empty, or if a specific message could not be found, the calling process simply continues.

Finally, most queuing systems also allow a process to install a handler as a *callback function*, which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver's side that continuously monitors the queue for incoming messages and handles accordingly.

General Architecture of a Message-Queuing System

Let us now take a closer look at what a general message-queuing system looks like. One of the first restrictions that we make, is that messages can be put only into queues that are local to the sender, that is, queues on the same machine, or no worse than on a machine nearby such as on the same LAN. Such a queue is called the source queue. Likewise, messages can be read only from local queues. However, a message put into a queue will contain the specification of a destination queue to which it should be transferred. It is the responsibility of a message-queuing system to provide queues to senders and receivers and take care that messages are transferred from their source to their destination queue.

It is important to realize that the collection of queues is distributed across multiple machines. Consequently, for a message-queuing system to transfer messages, it should maintain a mapping of queues to network locations. In practice, this means that it should maintain a (possibly distributed) database of queue names to network locations, as shown in Fig. 2-28. Note that such a mapping is completely analogous to the use of the Domain Name System (DNS) for e-mail in the Internet. For example, when sending mail to the logical *mail* address *steen@cs.vu.nl*, the mailing system will query DNS to find the *network* address (i.e., the IP address) of the recipient's mail server to use for the actual message transfer.

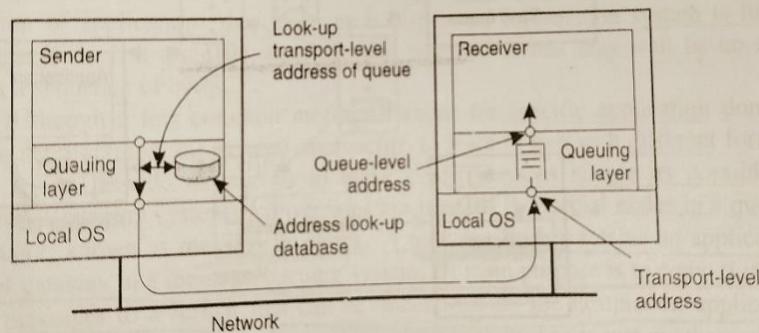


Figure 2-28. The relationship between queue-level addressing and network-level addressing.

Queues are managed by queue managers. Normally, a queue manager interacts directly with the application that is sending or receiving a message. However, there are also special queue managers that operate as routers, or relays: they forward incoming messages to other queue managers. In this way, a message-queuing system may gradually grow into a complete, application-level, overlay network, on top of an existing computer network. This approach is similar to the construction of the early MBone over the Internet, in which ordinary user processes were configured as multicast routers. These days, many routers already support multicasting, often rendering overlay multicasting less useful.

Relays can be convenient for a number of reasons. For example, in message-queuing systems, there is no general naming service available that dynamically maintain queue-to-location mappings. Instead, the topology of the queuing network is static, and each queue manager needs a copy of the queue-to-location mapping. It is needless to say that in large-scale queuing systems, this approach can easily lead to network-management problems.

One solution is to use a few routers that know about the network topology. When a sender A puts a message for destination B in its local queue, that message is first transferred to the nearest router, say R_1 , as shown in Fig. 2-29. At that point, the router knows what to do with the message and forwards it in the direction of B . For example, R_1 may derive from B 's name that the message should be forwarded to router R_2 . In this way, only the routers need to be updated when queues are added or removed, while every other queue manager has to know only where the nearest router is.

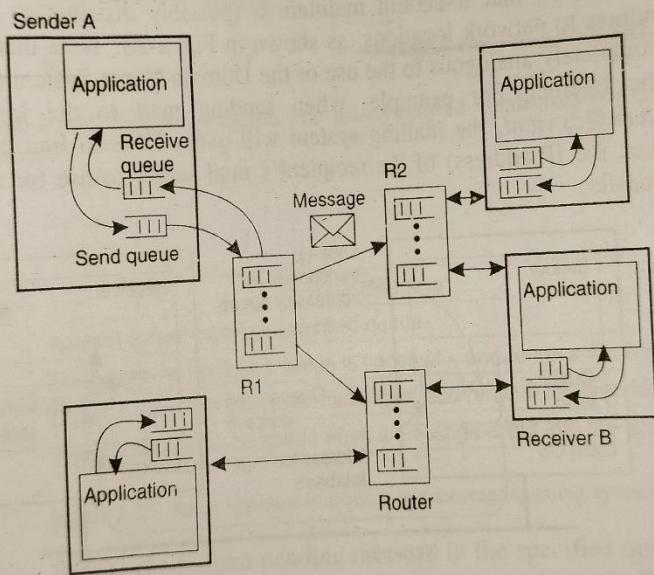


Figure 2-29. The general organization of a message-queuing system with routers.

(1) Relays can thus generally help build scalable message-queuing systems. However, as queuing networks grow, it is clear that the manual configuration of networks will soon become unmanageable. The only solution is to adopt dynamic routing schemes as is done for computer networks. In that respect, it is somewhat surprising that such solutions are not yet integrated into some of the popular message-queuing systems.

(2) Another reason why relays are used is that they allow for secondary processing of messages. For example, messages may need to be logged for reasons of

security or fault tolerance. A special form of relay that we discuss in the next section is one that acts as a gateway, transforming messages into a format that can be understood by the receiver.

- 3) Finally, relays can be used for multicasting purposes. In that case, an incoming message is simply put into each send queue.

Message Brokers

An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed information system. Integration requires that applications can understand the messages they receive. In practice, this requires the sender to have its outgoing messages in the same format as that of the receiver.

The problem with this approach is that each time an application is added to the system that requires a separate message format, each potential receiver will have to be adjusted in order to produce that format.

An alternative is to agree on a common message format, as is done with traditional network protocols. Unfortunately, this approach will generally not work for message-queuing systems. The problem is the level of abstraction at which these systems operate. A common message format makes sense only if the collection of processes that make use of that format indeed have enough in common. If the collection of applications that make up a distributed information system is highly diverse (which it often is), then the best common format may well be no more than a sequence of bytes.

Although a few common message formats for specific application domains have been defined, the general approach is to learn to live with different formats, and try to provide the means to make conversions as simple as possible. In message-queuing systems, conversions are handled by special nodes in a queuing network, known as message brokers. A message broker acts as an application-level gateway in a message-queuing system. Its main purpose is to convert incoming messages to a format that can be understood by the destination application. Note that to a message-queuing system, a message broker is just another application, as shown in Fig. 2-30. In other words, a message broker is generally not considered to be an integral part of the queuing system.

A message broker can be as simple as a reformatter for messages. For example, assume an incoming message contains a table from a database, in which records are separated by a special end-of-record delimiter and fields within a record have a known, fixed length. If the destination application

How the code is executed on the CPU and GPU memory in CUDA? How the allocating and deallocating of device memory space via `cudaMalloc` and `cudaFree` takes place?

In CUDA, the execution of code and the management of memory involve several steps:

1. [**Execution of Code:** The execution of any CUDA program involves three main steps¹:](#)
 - o **Host-to-Device Transfer:** Copy the input data from host memory (CPU) to device memory (GPU).
 - o **Execution:** Load the GPU program and execute it, caching data on-chip for performance.
 - o **Device-to-Host Transfer:** Copy the results from device memory back to host memory.
2. [**Memory Allocation and Deallocation:**](#)
 - o [`cudaMalloc`: This function is used to allocate an object in the device's global memory². It takes two parameters: the address of a pointer to the allocated object and the size of the allocated object in terms of bytes².](#)
 - o [`cudaFree`: This function is used to free an object from the device's global memory².](#)

Here is a simple example of how you might use these functions in a CUDA program:

```
const size_t SIZE = 900; // width and height of an image
float *abc; // Declare a pointer to a float

// Allocate memory on the GPU
cudaMalloc((void **)&abc, SIZE * SIZE * sizeof(float));

// ... Perform operations on abc ...

// Free the allocated memory
cudaFree(abc);
```

In this example, `cudaMalloc` is used to allocate a block of memory on the GPU that can hold `SIZE * SIZE` floats. [The `cudaFree` function is then used to deallocate that memory once it is no longer needed³.](#)

Please note that CUDA does not natively support the allocation of 2-dimensional arrays. [If you need a 2-dimensional array, you will need to either manually compute the index into a 1-dimensional array or allocate an array of pointers, each of which points to a 1-dimensional array³.](#)

Remember that efficient memory management is crucial for achieving good performance in CUDA applications. Always be sure to free any memory that you allocate once you're done using it to prevent memory leaks. Also, try to minimize data transfers between the host and the device, as these can be expensive operations.

The process migration facility of distributed system does not allow free migration of processes from one node to another but has restrictions regarding which node's processes can be migrated to which other nodes of the system. What might be the reasons behind imposing such a restriction? What are the action to be taken with respect to the references to the local resources when migrating the code to another machine?

Process migration in a distributed system is a complex task and there could be several reasons for imposing restrictions on which node's processes can be migrated to which other nodes:

1. **Dynamic Load Balancing:** It allows processes to take advantage of less loaded nodes by migrating from overloaded ones¹².
2. **Availability:** Processes that reside on a failure node can be migrated to other healthy nodes¹².
3. **System Administration:** Processes that reside on a node that is to be undergone system maintenance can be migrated to other nodes¹².
4. **Data Locality:** Processes can take advantage of locality of data or other special capabilities of a particular node¹².
5. **Mobility:** Processes can be migrated from a handheld device or laptop computer to a more powerful server computer before the device gets disconnected from the network².
6. **Fault Recovery:** The mechanism to halt, transport and resume a process is technically useful to support fault recovery in mission-critical or transaction-based applications².
7. **Node Capabilities:** Not all nodes might have the same capabilities or resources, and processes might be restricted based on the requirements of the process³.
8. **Refusal of Destination System:** Sometimes the destination system may refuse to accept the migration of a process to itself. If a workstation is idle, a process may have been migrated to it. Once the workstation is active, it

may be necessary to evict (recover) the migrated processes to provide adequate response time³.

These restrictions help ensure that the distributed system operates efficiently and effectively, balancing the load across nodes, preserving data locality, and maintaining system stability¹²³.

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult, is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else. An example of such a binding by identifier is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address. In the same line of reasoning, references to local communication endpoints also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated. In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment.

Finally, fixed resources are intimately bound to a specific machine or cannot be moved. Fixed resources are often local devices. An example of a fixed resource is a local communication endpoint.

Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-14.

Process-to-resource binding	Resource-to-machine binding		
	Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)
	By value	CP (or MV,GR)	GR (or CP)
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)
GR Establish a global systemwide reference			
MV Move the resource			
CP Copy the value of the resource			
RB Rebind process to locally available resource			

Figure 3-14. Actions to be taken with respect to the references to local resources when migrating code to another machine.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to establish a global reference.]

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy, is when migrating a process that is making use of a local communication endpoint. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and

install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change *their* global reference, and send messages to the new communication endpoint at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement distributed shared memory mechanisms as discussed in Chap. 1. Obviously, this is not really a viable solution.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding, the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.4.3 Migration in Heterogeneous Systems

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own architecture and operating system.

A programmer is writing an application for release consistent system. However an application needs sequential consistency to produce correct results. What precautions must the programmer

take?

When writing an application for a release consistent system that needs sequential consistency, the programmer must take the following precautions¹:

1. **Proper Synchronization:** The programmer must ensure that the program is properly synchronized¹. This means that synchronization primitives like locks, semaphores, or monitors are used correctly to control the order of access to shared resources¹.
2. **Acquire and Release Operations:** In a release consistent model, the actions of entering and leaving a critical section are classified as acquire and release¹. The programmer must explicitly code these operations into the program¹.
3. **Order of Operations:** The programmer must ensure that before an access to a shared variable is performed, all previous acquires by this processor must have completed¹. Similarly, before a release is performed, all previous reads and writes by this process must have completed¹.
4. **Processor Consistency:** The acquire and release accesses must be processor consistent¹. This means that the memory accesses performed by each processor appear in a single, sequentially consistent order².

If these conditions are met and the program is properly synchronized (i.e., processors implement acquire and release properly), the results of any execution will be exactly the same as they would have been executed following sequential consistency¹. In effect, accesses to shared variables are separated into atomic operation blocks by the acquire and release primitives so that races and interleaving between blocks will be prevented¹.

How would you incorporate persistence asynchronous communication into model of communication based on RMI to remote objects?

Incorporating persistent asynchronous communication into a model of communication based on Remote Method Invocation (RMI) involves a few steps¹²:

1. **Persistent Communication:** In persistent communication, messages are stored until the receiver is ready¹. This can be achieved by implementing a message queue or buffer at the sender's end¹. When a message is sent, it is stored in this buffer until the receiver is ready to process it¹.
2. **Asynchronous Communication:** In asynchronous communication, the sender continues immediately after it has submitted the message¹. This can be achieved by implementing non-blocking methods or using separate threads for sending and receiving messages¹.

In Java RMI, you could achieve this by creating a separate thread for each remote method invocation³. The thread would handle the communication with the remote object, allowing the main application thread to continue executing without waiting for the remote method to complete³. To ensure persistence, you could use a local buffer or a database to store the messages until they are processed¹.

However, it's important to note that incorporating persistent asynchronous communication into RMI may require significant changes to the application's architecture and could introduce additional complexities¹². It's also worth noting that while this approach can improve the responsiveness of the application, it may also increase the risk of data inconsistencies if not properly managed¹².

ESE IT JAN 22

What is weak consistency model? Write down its three properties.

Using time diagram show the valid and invalid sequence of event.

There are six possible outcomes: P_1 is killed, P_2 is killed, or both processes can be killed. With FIFO consistency, however, both processes can be killed. This result can occur if P_1 reads $R_1(y)0$ before it sees P_2 's $W_2(y)1$ and P_2 reads $R_1(x)0$ before it sees P_1 's $W_1(x)1$. With a sequentially consistent data store, there are six possible statement interleavings, and none of them results in both processes being killed.

6.2.5 Weak Consistency PSE

Although FIFO consistency can give better performance than the stronger consistency models, it is still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Not all applications require even seeing *all* writes, let alone seeing them in order. Consider the case of a process inside a critical section writing records to a replicated database. Even though other processes are not supposed to touch the records until the first process has left its critical section, the database system has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all copies of the database.

A better solution would be to let the process finish its critical section and then make sure that the final results are sent everywhere, not worrying too much whether all intermediate results have also been propagated to all copies in order, or even at all. In general, this can be done by introducing what is called a synchronization variable. A synchronization variable S has only a single associated operation synchronize(S), which synchronizes all local copies of the data store. Recall that a process P performs operations only on its locally available copy of the store. When the data store is synchronized, all local writes by process P are propagated to the other copies, whereas writes by other processes are brought in to P 's copy.

Using synchronization variables to partly define consistency leads to what is called weak consistency (Dubois et al., 1988). Weak consistency models have three properties:

1. Accesses to synchronization variables associated with a data store, are sequentially consistent.
2. No operation on a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

The first point says that all processes see all operations on synchronization variables in the same order. In other words, if process P_1 calls `synchronize(S1)` at the same time that process P_2 calls `synchronize(S2)`, the effect will be the same as if either `synchronize(S1)` preceded `synchronize(S2)`, or vice versa.

The second point says that synchronization "flushes the pipeline." It forces all writes that are in progress or partially completed or completed at some local copies but not others to complete everywhere. When the synchronization is done, all previous writes are guaranteed to be done as well. By doing a synchronization after updating shared data, a process can force the new values out to all other local copies of the store.

The third point says that when data items are accessed, either for reading or writing, all previous synchronizations will have been completed. By doing a synchronization before reading shared data, a process can be sure of getting the most recent values.

Unlike the previous consistency models, weak consistency enforces consistency on a group of operations, not on individual reads and writes. This model is most useful when isolated accesses to shared data are rare, with most accesses coming in clusters (many accesses in a short period, then none for a long time).

Another important distinction with the previous consistency models is that we now limit only the *time* when consistency holds, rather than limiting the *form* of consistency. In fact, we could say that with weak consistency, sequential consistency is enforced between groups of operations instead of between individual operations. Synchronization variables are used to delimit those groups.

```

int a, b, c, d, e, x, y;           /* variables */
int *p, *q;                      /* pointers */
int f(int *p, int *q);           /* function prototype */

a = x * x;                       /* a is stored in a register */
b = y * y;                       /* b as well */
c = a * a * a + b * b + a * b;   /* used later */
d = a * a * c;                   /* used later */
p = &a;                          /* p gets the address of a */
q = &b;                          /* p gets the address of b */
e = f(p,q);                     /* function call */

```

Figure 6-14. A program fragment in which some variables may be kept in registers.

The idea of having memory be wrong is nothing new. Many compilers cheat too. For example, consider the program fragment of Fig. 6-14, with all the variables initialized to appropriate values. An optimizing compiler may decide to compute a and b in registers and keep the values there for a while, not updating their memory locations. Only when the function f is called does the compiler have

to put the current values of a and b back in memory, because f might try to access them. This is a typical compiler optimization.

Having memory be wrong is acceptable here because the compiler knows what it is doing (i.e., because the software does not insist that memory be up-to-date). Clearly, if a second process existed that could read memory in an unconstrained way, this scheme would not work. For example, if during the assignment to d , the second process read a , b , and c , it would get inconsistent values (the old values of a and b , but the new value of c). One could imagine a special way to prevent chaos by having the compiler first write a special flag bit saying that memory was out-of-date. If another process wanted to access a , it would busy wait on the flag bit. In this way one can live with less than perfect consistency provided that synchronization is done in software and all parties obey the rules.

Now let us consider a somewhat less far-fetched situation. In Fig. 6-15(a) we see that process P_1 does two writes to a data item, and then synchronizes (indicated by the letter S). If P_2 and P_3 have not yet been synchronized, no guarantees are given about what they see, so this sequence of events is valid.

P1:	W(x)a	W(x)b	S
P2:		R(x)a	R(x)b
P3:		R(x)b	R(x)a

(a)

P1:	W(x)a	W(x)b	S
P2:		R(x)a	
P3:			

(b)

Figure 6-15. (a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

Fig. 6-15(b) is different. Here P_2 has been synchronized, which means that its local copy of the data store is brought up to date. When it reads x , it must get the value b . Getting a , as shown in the figure, is not permitted with weak consistency.

If we can think of replicating a remote object across several machine, we need to solve the problem of how to protect the object against simultaneous access by multiple clients. What are the two solution to this problem

Consider a distributed remote object that is shared by multiple clients, as shown in Fig. 6-1. Before we can even think of replicating a remote object across several machines, we need to solve the problem of how to protect the object against simultaneous access by multiple clients. There are basically two solutions to this problem (see, for example, Briot et al., 1998).

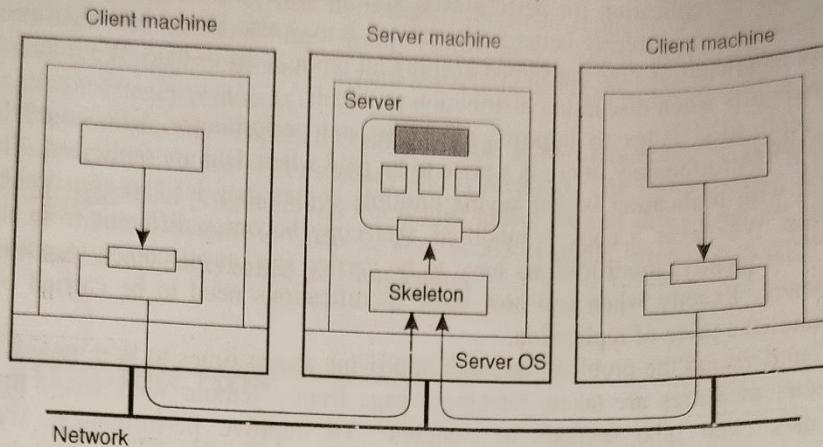


Figure 6-1. Organization of a distributed remote object shared by two different clients.

Sol 1

The first solution is that the object itself can handle concurrent invocations. As an example, we explained that a Java object can be constructed as a monitor by declaring the object's methods to be synchronized. Assume that two clients simultaneously invoke a method of the same object, leading to two concurrent threads at the server where the object resides. In Java, if the object's methods have been synchronized, only one of those two threads is allowed to proceed while the other is blocked until further notice. Different levels of concurrency may exist, but the important issue is that the object itself implements the means to handle concurrent invocations. This principle is shown in Fig. 6-2(a).

Sol 2

The second solution is that the object is completely unprotected against concurrent invocations, but that instead, the server in which the object resides is made responsible for concurrency control. In particular, by using an appropriate object adapter, it becomes possible to ensure that concurrent invocations will not leave the object in a corrupted state. For example, such an object adapter is one that uses a single thread per object, effectively serializing all accesses to each object it manages, as shown in Fig. 6-2(b).

Replicating a shared remote object without taking any special measures regarding the handling of concurrent invocations may lead to consistency problems. These problems are caused by the fact that the replicas need additional synchronization to ensure that concurrent invocations are performed in the correct order at each of the replicas. An example of this synchronization problem was the

What are the reason for replication? What are the two issues in replication?

Replication in distributed systems is primarily used for the following reasons¹²³⁴⁵:

1. **Fault Tolerance:** If a node stops working, the distributed network will still work fine due to its replicas¹.
2. **Load Sharing:** Loads on a server are shared among different replicas¹.
3. **Data Availability:** It enhances the availability of the data. If the replicas are created and data is stored near to the consumers, it would be easier and faster to fetch data¹.
4. **Performance Improvement:** By having replicas, workloads can be distributed across them⁵.

However, replication also introduces two key issues¹⁶⁷⁸:

1. **Consistency Maintenance:** Keeping all replicas consistent is a major challenge. Any change made to one replica needs to be reflected in all others¹. This can increase resource consumption and time complexity¹.
2. **Replica Placement:** Deciding where, when, and by whom replicas should be placed is a complex problem⁷. The goal of data replication solutions is to create a state close to optimal in these systems⁹. Therefore, there is a growing need for algorithms that determine the optimal replication configuration in the system according to the available resources⁹.

How Mutual Exclusion is handled in Distributed System? Compare the Centralized, Distributed and Token Ring Algorithms with respect to Message per entry/exit, Delay before entry (in message times), Problems

5.5.4 A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In Fig. 5-16 we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm.

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires $n - 1$ request messages, one to each of the other processes, and an additional $n - 1$ grant messages, for a total of $2(n - 1)$. (We assume that only point-to-point communication channels are used.) With the token ring algorithm, the number is variable. If every process constantly

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	
Distributed	$2(n - 1)$	$2(n - 1)$	Coordinator crash
Token ring	1 to ∞	0 to $n - 1$	Crash of any process Lost token, process crash

Figure 5-16. A comparison of three mutual exclusion algorithms.

wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. 5-16 we show the former case. It takes only two message times to enter a critical region in the centralized case, but $2(n - 1)$ message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to $n - 1$ (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

5.6 DISTRIBUTED TRANSACTIONS

A concept that is strongly related to mutual exclusion is that of a transaction. Mutual exclusion algorithms ensure that a shared resource such as a file, printer, and so on, is accessed by at most one process at a time. Transactions have in common that they also protect a shared resource against simultaneous access by several concurrent processes. In particular, transactions are used to protect

Consider the behavior of two machines in a distributed system.

Both

have clocks that are supposed to tick 1000 times per millisecond. One

of them actually does, but the other ticks only 990 times per millisecond. If UTC updates come in once a minute, what is the maximum clock skew that will occur?

The clock skew can be calculated by comparing the difference in the clock rates of the two machines.

Given that one machine ticks 1000 times per millisecond (the correct rate), and the other ticks 990 times per millisecond, the difference in their rates is 10 ticks per millisecond.

Since there are 1000 milliseconds in a second and 60 seconds in a minute, the maximum clock skew that will occur before the next UTC update comes in is:

$$10 \text{ ticks/ms} \times 1000 \text{ ms/s} \times 60 \text{ s} = 600,000 \text{ ticks}$$

So, the maximum clock skew that will occur is 600,000 ticks.

Show the relation between clock time and UTC when clocks tick at different rates.

In a distributed system, each machine has its own physical clock that ‘ticks’ based on physical processes¹. However, these clocks are imperfect and can gain or lose time over time, a process known as clock drift¹. This means that clocks can tick at different rates, creating an ever-widening gap in perceived time¹.

Coordinated Universal Time (UTC) is a time standard that is based on International Atomic Time (TAI) with leap seconds added at irregular intervals to compensate for Earth’s slowing rotation²³. UTC provides a stable reference time scale that can be used to synchronize the clocks in a distributed system²³.

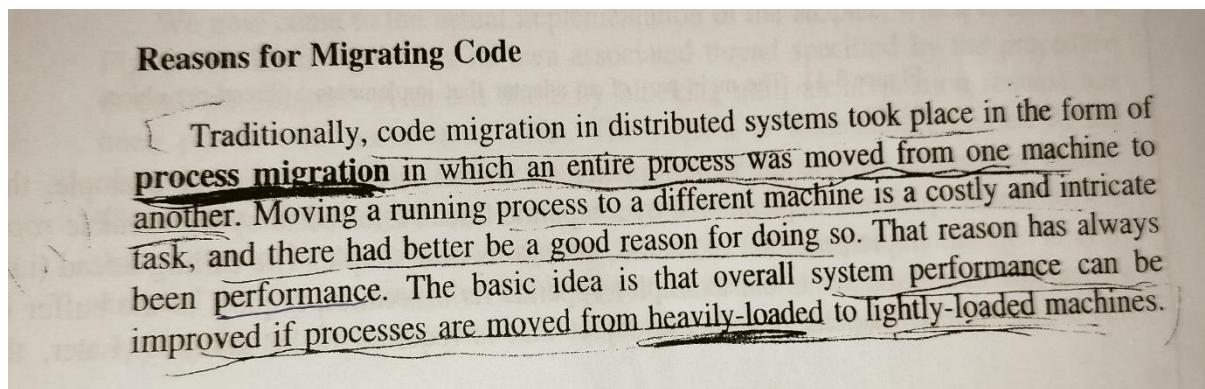
When a machine’s clock ticks at a different rate than UTC, it can have positive or negative drift with respect to UTC¹. To deal with this drift, the machine needs to resynchronize periodically¹. This resynchronization can be achieved by making small adjustments to the clock’s tick rate¹. If the clock is fast, it is made to run slower until it matches UTC, and if it is slow, it is made to run faster until it matches UTC¹.

So, the relationship between a machine's clock time and UTC when clocks tick at different rates is a dynamic one, with the machine's clock time being continuously adjusted to minimize the skew with UTC¹. This process of clock synchronization aims to minimize the difference, or skew, between the machine's clock and UTC¹.

What are reason for migrating code?

Describe the following

- a) Code segment
- b) Resource segment and
- c) Execution segment



Define

- i **Clock drift** - Clock drift refers to several related phenomena where a clock does not run at exactly the same rate as a reference clock¹. That is, after some time the clock "drifts apart" or gradually desynchronizes from the other clock¹. All clocks are subject to drift, causing eventual divergence unless resynchronized¹.
- ii. **Clock skew** – When a system has n computers and all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of synchronization and give different values when read out. This difference in time values is called clock skew.

Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modern distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, for example, a client-server system in which the server manages a huge database. If a client application needs to do many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

[This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.]

[Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.]

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and deciding in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 1.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that

protocol, would need to be linked with the client application. This application requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-12. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

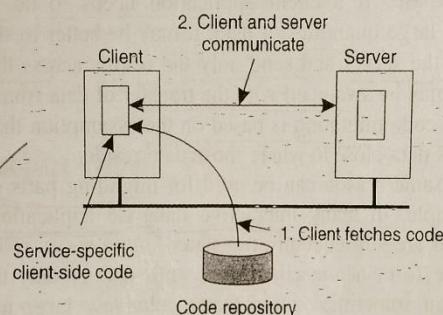


Figure 3-12. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

The important advantage of this model of dynamically downloading client-side software, is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server. There are, of course, also disadvantages. The most serious one, which we discuss in Chap. 8, has to do with security. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code

described in (Fuggetta et al., 1998). In this framework, a process consists of three segments. The code segment is the part that contains the set of instructions that make up the program that is being executed. The resource segment contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on. Finally, an execution segment is used to store the current execution state of a process, consisting of private data, the stack, and the program counter.

What is group communication? Give any two key application area of group communication in distributed systems

What is distributed object? Draw the organization of distributed object and explain its components.

2.3.1 Distributed Objects *FS*

The key feature of an object is that it encapsulates data, called the state, the operations on those data, called the methods. Methods are made available through an interface. It is important to understand that there is no legal way process can access or manipulate the state of an object other than by invoking methods made available to it through an object's interface. An object may implement multiple interfaces. Likewise, given an interface definition, there may be several objects that offer an implementation for it.

This separation between interfaces and the objects implementing these interfaces is crucial for distributed systems. A strict separation allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Fig. 2-16, is commonly referred to as a distributed object.

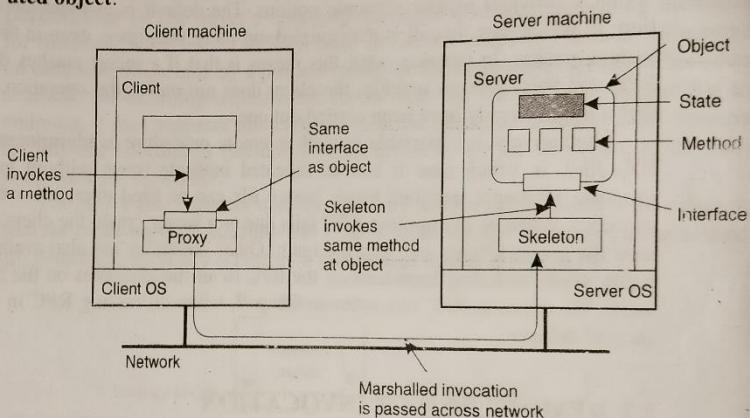


Figure 2-16. Common organization of a remote object with client-side proxy.

When a client binds to a distributed object, an implementation of the object's interface, called a proxy, is loaded into the client's address space. A proxy is analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a server stub, often referred to as a skeleton, which unmarshals them to proper method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.

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

2.3 REMOTE OBJECT INVOCATION

interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects. As we shall see in later chapters, in a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces.

In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the

necessary processing, assuming that the data needed are in a cache

in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded

<https://gateoverflow.in/324493/andrew-tanenbaum-os-edition-exercise-question-17-page-no-175>

Let's calculate the number of requests per second a single-threaded server can handle first.

The time taken to process a request in the single-threaded server is the sum of the time taken to get a request for work, dispatch it, and do the necessary processing (15 msec), and the time taken for a disk operation, which is needed one-third of the time ($75 \text{ msec} * 1/3$). This gives us:

$$\begin{aligned}\text{Time per request} &= 15 \text{ msec} + 13 * 75 \text{ msec} = 40 \text{ msec} \\ \text{Time per request} &= 15 \text{ msec} + 31 \\ *75 \text{ msec} &= 40 \text{ msec}\end{aligned}$$

The number of requests per second the server can handle is the reciprocal of the time per request (in seconds):

$$\begin{aligned}\text{Requests per second} &= 1 / \text{Time per request} = 1 / 40 \text{ msec} = 25 \text{ requests/sec} \\ \text{Requests per second} &= 1 / \text{Time per request} = 1 / 31 \text{ msec} = 25 \text{ requests/sec}\end{aligned}$$

For a multithreaded server, the situation is a bit more complex. If we assume that the server has enough threads to handle all incoming requests simultaneously, then it can handle a disk operation (which takes 75 msec) while continuing to process other requests. Therefore, the server is not limited by the time taken for disk operations. In this case, the server can handle:

$$\begin{aligned}\text{Requests per second} &= 1 / (15 \text{ msec} + 31 \text{ msec}) = 66.67 \text{ requests/sec} \\ \text{Requests per second} &= 1 / (15 \text{ msec} + 31 \text{ msec}) = 66.67 \text{ requests/sec}\end{aligned}$$

Please note that this is a simplified analysis and actual performance can vary based on many factors such as the number of available threads, the overhead of context switching, the performance of the disk subsystem, and so on. It's also important to note that while multithreading can increase throughput, it can also increase

complexity and the potential for synchronization issues. Therefore, the decision to use a single-threaded or multithreaded server should be based on the specific requirements and constraints of your application.

What is group communication? Give any two key application area of group communication in distributed systems

Group communication is the act of sending and receiving messages to multiple members of a group¹²³⁴. In a business or distributed system environment, groups often use this type of communication to exchange ideas, determine goals, and motivate other members¹²³⁴. The size of groups can vary, but they typically comprise at least three members and up to 20 members¹²³⁴. When groups communicate effectively, they can share necessary information as they work to accomplish mutual goals¹²³⁴.

Key application areas of group communication in distributed systems include:

1. **Reliable Dissemination of Information:** Group communication is used to reliably disseminate information to potentially large numbers of clients⁵⁶. This is particularly useful in applications where updates or notifications need to be sent to multiple recipients simultaneously⁵⁶.
2. **Support for Collaborative Applications:** Group communication provides the infrastructure needed for collaborative applications⁵⁶. These applications allow multiple users to work together on shared tasks, such as document editing, software development, or decision-making processes⁵⁶.

Compare Distributed OS, Network OS, and Middleware based distributed systems with respective following parameters:

Degree of transparency

Same OS on all node

No. of copies of OS

Basis for communication

Resource management

vi. Scalability

vii. Openness

Here's a comparison of Distributed Operating Systems (DOS), Network Operating Systems (NOS), and Middleware-based Distributed Systems (MDS) based on the parameters you mentioned¹²:

Parameter	Distributed OS	Network OS	Middleware-based DS
Degree of Transparency	<u>High¹</u>	<u>Low¹</u>	<u>Varies¹</u>
Same OS on all nodes	<u>Yes¹</u>	<u>No¹</u>	<u>No¹</u>
No. of copies of OS	<u>One per node¹</u>	<u>One per node¹</u>	<u>One per node¹</u>
Basis for Communication	<u>Message passing and shared memory¹</u>	<u>File-based¹</u>	<u>Depends on middleware¹</u>
Resource Management	<u>Managed by the OS¹</u>	<u>Managed locally at each node¹</u>	<u>Depends on middleware¹</u>
Scalability	<u>Less scalable¹</u>	<u>More scalable¹</u>	<u>Depends on middleware¹</u>
Openness	<u>Varies¹</u>	<u>Varies¹</u>	<u>Typically high¹</u>

Please note that these are general characteristics and actual implementations can vary. Also, the degree of transparency, scalability, and openness in Middleware-based Distributed Systems can greatly depend on the specific middleware used¹.

Many networked systems are organized in terms of a back office and a front office. How does organizations match with the coherent view we demand for a distributed system?

In many networked systems, the organization is split into a **back office** and a **front office**¹²³⁴⁵. The back office typically refers to the administrative and support functions of an organization, such as accounting, human resources, and IT¹²³⁴⁵. The front office, on the other hand, refers to the customer-facing functions, such as sales, marketing, and customer service¹²³⁴⁵.

In a distributed system, we demand a coherent view, meaning that the system should appear to its users as a single, integrated entity, rather than a collection of independent components. This is often referred to as transparency.

The back office and front office can fit into this coherent view in the following ways:

1. **Data Consistency:** Both the back office and front office need to work with consistent data. For example, the sales data that the front office uses should be consistent with the financial data that the back office uses.
2. **Process Integration:** The processes in the back office and front office need to be integrated. For example, when a sale is made (a front office process), the inventory should be updated (a back office process).
3. **Communication:** The back office and front office need to communicate effectively. This can be facilitated by middleware, which provides a communication layer between different parts of the distributed system.

By ensuring data consistency, process integration, and effective communication, the back office and front office can both contribute to the coherent view of the distributed system.

ESE AUG 2023

What are the different goals of distributed systems?

<https://www.ques10.com/p/2123/state-the-goals-of-distributed-system/>

When to use vector clock? What is the difference between Lamport clock

and vector clock? With suitable example explain Lamport clock and vector clock.

Vector Clocks are used in distributed systems to provide a causally consistent ordering of events¹². They are particularly useful in systems where it's important to determine if any two arbitrarily selected events are causally dependent or concurrent¹³². For example, they are often used in multi-node databases and distributed message queues².

Lamport Clocks and Vector Clocks are both logical clocks used to order events in a distributed system⁴. Here are the key differences between them:

1. **Causal Relationships:** Vector clocks can determine if any two arbitrarily selected events are causally dependent or concurrent⁴. Lamport clocks cannot do this⁴.
2. **Compactness:** Lamport clocks are more compact⁴. Vector clocks require overhead proportional to the number of nodes⁴.

Now, let's look at examples of how each of these clocks work:

Lamport Clock Example⁵⁶⁷⁸⁹:

Consider two processes, P1 and P2, with events E11, E12, E13 in P1 and E21, E22, E23 in P2. The events are as follows:

- E11: P1 sends a message M1 to P2.
- E12: P1 receives a message M2 from P2.
- E13: P1 sends a message M3 to P2.

The Lamport timestamps for these events would be:

- E11: 1 (P1 sends M1)
- E21: 1 (P2 receives M1, increments its clock to $\max(1, 1) + 1 = 2$)
- E12: 2 (P1 receives M2, increments its clock to $\max(2, 1) + 1 = 3$)
- E13: 3 (P1 sends M3)
- E22: 2 (P2 receives M3, increments its clock to $\max(3, 2) + 1 = 4$)
- E23: 4 (P2 has an internal event)

Vector Clock Example¹

Explain working of JAVA RMI with example.

2.3.6 Example 2: Java RMI ESE

In DCE, distributed objects have essentially been added as a refinement of remote procedure calls. Instead of identifying a remote procedure in a server, a client identifies a remote procedure in a server's object. The lack of a proper systemwide object reference mechanism emphasizes that we are indeed dealing with a mere refinement of RPCs.

Let us now take a look at distributed objects from a completely different perspective. In Java, distributed objects have been integrated into the language. An important goal was to keep as much of the semantics of nondistributed objects as possible. In other words, the Java language developers have aimed for a high degree of distribution transparency. However, as we shall see, Java's developers have also decided to make distribution apparent where a high degree of transparency was simply too inefficient, difficult, or impossible to realize.

The Java Distributed-Object Model

Java also adopts remote objects as the only form of distributed objects. Recall that a remote object is a distributed object whose state always resides on a single machine, but whose interfaces can be made available to remote processes. Interfaces are implemented in the usual way by means of a proxy, which offers exactly the same interfaces as the remote object. A proxy itself appears as a local object in the client's address space.

There are only a few, but subtle and important differences between remote objects and local objects. First, cloning local or remote objects is different. Cloning a local object O results in a new object of the same type as O , and with exactly the same state. Cloning thus returns an exact copy of the object that is cloned. These semantics are hard to apply to a remote object. If we were to make an exact copy of a remote object, we would not only have to clone the actual object at its server, but also the proxy at each client that is currently bound to the remote object. Cloning a remote object is therefore an operation that can be executed only by the server. It results in making an exact copy of the actual object in the server's address space. Proxies of the actual object are thus not cloned. If a client at a remote machine wants access to the cloned object at the server, it will first have to bind to that object again.

A more important difference between local and remote objects in Java is the semantics of blocking on an object. Java allows each object to be constructed as a *monitor*, by declaring a method to be **synchronized**. If two processes simultaneously call a synchronized method, only one of the processes will proceed while the other will be blocked. In this way, we can ensure that access to an object's internal data is completely serialized. As in monitors, a process can also be blocked inside an object, waiting for some condition to become true, as was also explained in Chap. 1.

Logically, blocking in a remote object is simple. Suppose that client A calls a synchronized method of a remote object. To make access to remote objects look always *exactly* the same as to local objects, it would be necessary to block A in the client-side stub that implements the object's interface and to which A has direct access. Likewise, another client on a different machine would need to be blocked locally as well before its request can be sent to the server. The consequence is that we need to synchronize different clients at different machines. As we discuss in Chap. 5, distributed synchronization can be fairly complex.

An alternative approach would be to allow blocking only at the server. In principle, this works fine, but problems arise when a client crashes while its invocation is being handled by the server. As we shall see in Chap. 7, we may require relatively sophisticated protocols to handle this situation, and which that may significantly affect the overall performance of remote method invocations.

Therefore, the designers of Java RMI have chosen to restrict blocking on remote objects only to the proxies (Wolfrath et al., 1996). In practice, this means

that remote objects cannot be protected against simultaneous access from processes operating on different proxies by using synchronized methods. Instead, explicit distributed locking techniques will have to be used.

Java Remote Object Invocation ESE

As the distinction between local and remote objects is hardly visible at the language level, Java can also hide most of the differences during a remote method invocation. For example, any primitive or object type can be passed as a parameter to an RMI, provided that the type can be marshaled. In Java terminology, this means that it must be serializable. Although, in principle, most objects can be serialized, serialization is not always allowed or possible. Typically, platform-dependent objects such as file descriptors and sockets, cannot be serialized.

The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference. In other words, a local object is first copied after which the copy is used as parameter value. For a remote object, a reference to the object is passed as parameter instead of a copy of the object, as was also shown in Fig. 2-18.

In Java RMI, a reference to a remote object is essentially implemented as we explained in Sec. 2.3.2. Such a reference consists of the network address and endpoint of the server, as well as a local identifier for the actual object in the server's address space. That local identifier is used only by the server. As we also explained, a reference to a remote object also needs to encode the protocol stack that is used by a client and the server to communicate. To understand how such a stack is encoded in the case of Java RMI, it is important to realize that each object in Java is an instance of a class. A class, in turn, contains an implementation of one or more interfaces.

In essence, a remote object is built from two different classes. One class contains an implementation of server-side code, which we call the *server class*. This class contains an implementation of that part of the remote object that will be running on a server. In other words, it contains the description of the object's state, as well as an implementation of the methods that operate on that state. The server-side stub, that is, the skeleton, is generated from the interface specifications of the object.

The other class contains an implementation of the client-side code, which we call the *client class*. This class contains an implementation

How are the replications used in distributed systems?

Replication in distributed systems involves making redundant copies of resources, such as data or processes, while ensuring that all the copies are identical¹. This is done to improve the reliability, fault-tolerance, and performance of the system².

Here are some ways replications are used in distributed systems:

1. **Fault Tolerance:** If a node stops working, the distributed network will still work fine due to its replicas¹.
2. **Load Sharing:** Loads on a server are shared among different replicas¹.
3. **Data Availability:** It enhances the availability of the data. If the replicas are created and data is stored near to the consumers, it would be easier and faster to fetch data¹.
4. **Performance Improvement:** By having replicas, workloads can be distributed across them¹.

There are two main types of replication strategies used in distributed systems¹:

- **Active Replication:** The request of the client goes to all the replicas. It is to be made sure that every replica receives the client request in the same order else the system will get inconsistent¹.
- **Passive Replication:** The client request goes to the primary replica, also called the main replica. There are more replicas that act as backup for the primary replica¹.

However, replication also introduces complexity and cost of data updates³. In many cases, data storage systems that exploit replication use relaxed consistency criteria³. This survey describes different replication schemes and discusses several consistency models, protocols, and techniques designed to support consistency in replicated systems³.

What are the desirable characteristics of message passing? What are the different issues in IPC by message passing? How is group communication handled in distributed systems?

Desirable Characteristics of Message Passing¹²³⁴:

1. **Simplicity:** Message passing systems should be easy to use and understand¹.
2. **Uniform Semantics:** The system should provide a consistent set of operations for sending and receiving messages¹.
3. **Efficiency:** The system should minimize the overhead of message passing to ensure high performance¹.
4. **Reliability:** The system should ensure that messages are delivered accurately and reliably¹.

5. **Flexibility:** The system should be adaptable to a variety of applications and use cases¹.
6. **Security:** The system should provide mechanisms to protect the confidentiality and integrity of messages¹.

Issues in IPC by Message Passing⁵⁶⁷⁸⁹:

1. **Sender and Receiver Identification:** Determining who is sending and receiving the message⁵.
2. **Number of Receivers:** Determining whether there is a single receiver or multiple receivers⁵.
3. **Guaranteed Delivery:** Ensuring that the intended recipient received the message⁵.
4. **Reply Handling:** Determining whether it is required to wait for a reply from the sender⁵.
5. **Failure Handling:** Handling catastrophic events such as a communication link failure or node crash⁵.
6. **Buffering:** Deciding what to do with the message if the receiver is not ready to take it⁵.
7. **Synchronization:** Synchronization of communicative processes is a critical issue in the communication structure⁵.

Group Communication in Distributed Systems¹⁰¹¹¹²:

Group communication in distributed systems refers to the communication medium used by nodes (computers or processes) to commute information and coordinate their actions¹⁰. It involves transferring and entering messages between nodes to achieve various goals such as coordination, synchronization, and data sharing¹⁰. There are three types of group communication in a distributed system¹⁰:

1. **Broadcast Communication:** When the host process tries to communicate with every process in a distributed system at the same time¹⁰.
2. **Multicast Communication:** When the host process tries to communicate with a designated group of processes in a distributed system at the same time¹⁰.
3. **Unicast Communication:** When the host process tries to communicate with a single process in a distributed system at the same time¹⁰.

**What is the difference between physical clock and logical clock?
Explain Christians algorithm and Berkeleys algorithm with the help of example.**

Physical Clock vs Logical Clock:

A physical clock is a physical process combined with a strategy for measuring that process to record the progression of time¹. For example, the revolution of the Earth measured in solar days is a physical clock¹. However, physical clocks are not always precise, so we can't rely on physical time to order events².

A logical clock, on the other hand, is a mechanism for capturing chronological and causal relationships in a distributed system³. Distributed systems may have no physically synchronous global clock, so a logical clock allows global ordering on events from different processes in such systems³. A logical clock is not necessarily to maintain the same notion of time as a reliable watch. Instead, it is to keep track of information pertaining to the order of events⁴.

Cristian's Algorithm:

Cristian's Algorithm is a clock synchronization algorithm used to synchronize time with a time server by client processes⁵. Here's how it works:

1. The process on the client machine sends the request for fetching clock time (time at the server) to the Clock Server at time t_1 .
2. The Clock Server listens to the request made by the client process and returns the response in form of clock server time.
3. The client process fetches the response from the Clock Server at time t_2 and calculates the synchronized client clock time using the formula given below⁵:

$$T_{CLIENT} = T_{SERVER} + 2(t_2 - t_1)$$

where T_{CLIENT} refers to the synchronized clock time, T_{SERVER} refers to the clock time returned by the server, t_1 refers to the time at which request was sent by the client process, and t_2 refers to the time at which response was received by the client process⁵.

Berkeley's Algorithm:

Berkeley's Algorithm is a clock synchronization technique used in distributed systems⁶. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess a UTC server⁶. Here's how it works:

1. An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves⁶.
2. Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm⁶.
3. Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network⁶.

What are the difficulties are faced while using process migration over code migration? Explain code migration in heterogeneous systems with the help of example.

Process migration and code migration are two techniques used in distributed systems to improve performance, availability, and resource utilization¹². However, they each have their own challenges:

Process Migration Challenges¹:

- **State Transfer:** The entire state of the process, including the program counter, registers, and memory, must be transferred¹.
- **Resource Usage:** Processes may have open files, network connections, or other resources that need to be managed during migration¹.
- **Communication Overhead:** After migration, a process may need to communicate with other processes on its original node, leading to increased network traffic¹.
- **System Heterogeneity:** Differences in system architecture, operating system, or even system configuration can make process migration more complex¹.

Code Migration Challenges²:

- **Code Transfer:** Only the code segment is typically transferred, not the entire process state².
- **Resource Management:** The migrated code may need to access resources that are not available on the target node².
- **Security:** Transferring code between nodes can have security implications, including the risk of code tampering or unauthorized access².

In heterogeneous systems, code migration can be more complex due to differences in system architecture, operating system, or programming language support². One approach to handle this is through the use of **intermediate language or bytecode**, which can be interpreted or compiled on different platforms². For example, Java applets use this approach for code migration². The applet's Java bytecode is transferred over the network and executed by a Java Virtual Machine (JVM) on the target node, allowing the code to run on various types of systems².

Another example is the D'Agents system, a research middleware platform that supports various forms of code migration². An agent in D'Agents is a program that can migrate between heterogeneous platforms, and it can be written in Tcl, Scheme, or Java². This allows the agent to execute on a variety of systems, demonstrating how code migration can be used in a heterogeneous environment².

How is data centric consistency model is used in distributed systems? How consistent ordering of operations takes place in distributed systems?

Data-Centric Consistency Model:

The Data-Centric Consistency Model is a set of principles and protocols used in distributed systems to define how data is accessed and updated, ensuring that all nodes in the system agree on the state of shared data¹. It primarily focuses on the consistency guarantees provided to clients regarding the data they access and manipulate¹. Here are some common data-centric consistency models²¹:

- **Strong Consistency**: All nodes in the system agree on the order in which operations occurred. Reads will always return the most recent version of the data, and writes will be visible to all nodes immediately after they occur²¹.
- **Eventual Consistency**: Updates propagate through the system and eventually, all replicas will be consistent¹.
- **Causal Consistency**: All processes see causally-related shared accesses in the same order²¹.
- **Read-Your-Writes Consistency**: This ensures that a process that writes a value to some data item will always read the latest written value¹.
- **Monotonic Reads and Writes**: If a process reads the value of a data item, any successive reads on that data item by that process will always return the same or a more recent value¹.

Consistent Ordering of Operations:

Consistency models in distributed systems refer to the guarantees provided by the system about the order in which operations appear to occur to clients². Specifically, it determines how data is accessed and updated across multiple nodes in a distributed system, and how these updates are made available to clients². Here are some common techniques for ensuring consistent ordering of operations²³⁴:

- **Sequential Consistency**: The result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program²³.
- **Linearizability (Strict Consistency)**: All nodes in the system agree on the order in which operations occurred. Reads will always return the most recent version of the data, and writes will be visible to all nodes immediately after they occur²³.

- **Causal Consistency:** All processes see causally-related shared accesses in the same order²³.
- **Distributed Consensus Algorithms:** Algorithms like Paxos and Raft are used to achieve total ordering in distributed systems. These algorithms ensure that all nodes in the system agree on the order of events by coordinating a leader node that is responsible for ordering events⁴.

Why is load balancing required in distributed systems? Explain any two load balancing algorithms with examples in distributed systems.

Load balancing is a critical aspect of distributed systems as it helps optimize resource utilization, minimize response time, and ensure high availability and reliability¹². It involves distributing workloads across multiple computing resources to prevent any single resource from being overwhelmed¹². By ensuring that the workload is spread evenly among all available resources, load balancing can reduce server response time, maximize throughput, and provide a better user experience¹².

Here are two common load balancing algorithms used in distributed systems:

1. **Round Robin Algorithm:** This is one of the simplest methods for distributing client requests across a group of servers¹. When a request comes in, it is sent to the next server in the list. The algorithm then moves that server to the bottom of the list. For example, if there are three servers (Server 1, Server 2, Server 3), the first request goes to Server 1, the second to Server 2, the third to Server 3, and the fourth back to Server 1¹.
2. **Least Connections Algorithm:** This algorithm directs traffic to the server with the fewest active connections¹. This is useful in situations where sessions have a large variance in their processing time. For example, if there are three servers (Server 1 with 2 connections, Server 2 with 3 connections, Server 3 with 1 connection), the next request would go to Server 3, as it has the least number of active connections¹.

Please note that the choice of load balancing algorithm depends on the specific requirements and constraints of your distributed system¹.

What is the role of middle ware in distributed systems?

<https://www.educative.io/answers/what-is-the-role-of-middleware-in-distributed-systems>

In distributed systems, middleware is a software component that provides services between two or more applications and can be used by them¹. It sits between the application and the network¹².

Here are some key roles of middleware in distributed systems:

1. **Common Services:** Middleware provides common services such as authentication, authorization, compilation for best performance on particular architectures, input/output translation, and error handling¹.
2. **Communication:** Middleware enables one or more kinds of communication or connectivity between applications or application components in a distributed network². It simplifies connectivity between applications, application components, and back-end data sources².
3. **Modularization:** Middleware can be modularized from the application so it has better potential for reuse with other applications running on different platforms¹.
4. **Interoperability:** Middleware provides services that enable different applications and services to communicate using common messaging frameworks². It also provides services that enable components written in multiple languages to talk with each other².
5. **Integration:** Middleware allows organizations to implement their processes seamlessly by integrating all components of the enterprise¹.

Remember, there are numerous types of middleware designed for different scenarios, each having its own set of responsibilities¹. Therefore, it's vital for enterprises to first identify what type of solution they require before making a purchasing decision¹.