CS 570: Modern Operating Systems

Assignment 2 Solution

1. Prove that synchronous communication could lead to deadlock.  
     
   Deadlock is a situation in which two or more processes are unable to proceed because each is waiting for the other to release a resource. In synchronous communication, processes have a fixed, predefined order in which they send and receive messages. This means that a process cannot proceed until it receives a response from another process.   
   Consider a scenario where you have three processes, A, B, and C, each sending requests to the other two processes and waiting for responses in a synchronous manner. If A sends a request to B and waits for a response, B sends a request to C and waits, and C sends a request to A and waits, you have a circular dependency. If, for some reason, one of these messages is delayed or lost, it can lead to all three processes being stuck in a waiting state, resulting in a deadlock.
2. Exercise 3.10 (give one example for each, not two) from Pradeep Sinha's book  
     
   (a) **0-reliable:** In a 0-reliable multicast, there is no guarantee of reliability. For example, in a live video streaming application where viewers can join at any time, 0-reliable multicast may be used. If some viewers miss part of the stream due to network issues or joining late, there are no mechanisms to recover the lost data for those viewers.  
   (b) **1-reliable:** In a 1-reliable multicast, there is a guarantee that at least one receiver will successfully receive the data. An example could be a stock market data feed where real-time updates are sent to multiple subscribers. The goal is to ensure that at least one subscriber receives each update to maintain market transparency.  
   (c) **M out of n reliable:** In an M out of N-reliable multicast, the goal is to ensure that a minimum of M out of N receivers successfully receive the data. For example, in a video conferencing application where the sender is broadcasting to multiple participants, the goal may be to ensure that at least 4 out of 6 participants receive the video stream successfully, even if some participants experience network issues or temporary disruptions.  
   (d) **All-reliable:** In an all-reliable multicast, the goal is to ensure that every receiver in the multicast group successfully receives the data. An example could be a software update distribution system where a company wants to ensure that every computer in its network receives critical software updates without any failures or data loss.
3. Exercise 3.13 from Pradeep Sinha's book   
     
   (a) **One-to-many communication:** In One-to-many communication, the sender will put a time stamp on the messages to be sent and all the receivers will process the messages according to the associated time stamp on that message irrespective of the order in which it receives the message. In this way, the ordering is achieved in one–to–one communication.(b) **Many-to-one:** When all the receivers send an acknowledgment back to the sender after the first message is received then this manner of the consistent ordering of messages is achieved in many-to-one communication.  
   (c) **Many-to-many:** One method to implement consistent-ordering semantics is to make the many-to-many scheme appear as a combination of many-to-one and one-to-many schemes. That is, the kernels of the sending machines send messages to a single receiver (known as a sequencer) that assigns a sequence number to each message and then multicasts it. The kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue. Messages in a queue are delivered immediately to the receiver unless there is a gap in the message identifiers, in which case messages after the gap are not delivered until the ones in the gap have arrived. This mechanism ensures consistent ordering of many-to-many communication.
4. Exercise 3.14 from Pradeep Sinha's book   
     
   **Implicit Addressing:** Implicit addressing relies on contextual information and protocols to determine the destination of a message. The recipient is identified based on its position or role within the system, without specifying its exact address explicitly.  
   *Advantages*: Implicit addressing can simplify communication by allowing processes to send messages without needing to know the specific addresses of recipients. It enables dynamic changes in the network, such as adding or removing processes, without requiring updates to message senders.

*Disadvantages*: Implicit addressing may not be precise enough for certain applications, where specific targeting or routing is necessary. Dependency on context: It relies heavily on the underlying protocols and context, which can make it harder to troubleshoot or maintain in complex systems.

**Explicit Addressing:** Explicit addressing involves specifying the exact address or identifier of the recipient process in a message. Each process has a unique identifier, such as an IP address, port number, or a globally unique identifier (GUID).

*Advantages*: Explicit addressing ensures that messages are sent directly to the intended recipient, eliminating ambiguity. It gives the sender fine-grained control over the message's destination and allows for complex routing and addressing schemes.

*Disadvantages*: Explicit addressing can introduce complexity, especially in large and dynamic systems, where managing and updating addresses becomes challenging. It may not scale well in scenarios with a large number of processes, as managing and tracking unique identifiers can become cumbersome.

**Link-Based Addressing:** Link-based addressing is used in networks where processes are organized into a logical structure, such as a tree or a graph. Processes are identified based on their position or relationship within this structure.

*Advantages*: Link-based addressing often reflects the hierarchical organization of processes within a network, making it easy to navigate. It can be well-suited for systems with a hierarchical structure and a moderate number of processes.

*Disadvantages*: Link-based addressing may not be suitable for scenarios where processes do not neatly fit into a hierarchical structure or need more dynamic routing. In networks where processes frequently change positions or relationships, link-based addressing can be challenging to maintain.

(a) For communication between a server process and several client processes. The client processes send request messages to the server process, and the server process returns a reply for each client request.   
*Explicit Addressing*: In this scenario, precise communication is essential, as each client process needs to send requests to the server, and the server must return replies to the respective clients. Explicit addressing allows each client to specify the exact address of the server process, ensuring that requests are directed accurately, and replies are sent back to the correct clients.

(b) For allowing a sender process to send messages to a group of processes.  
*Implicit Addressing*: Implicit addressing is suitable for sending messages to a group of processes because it simplifies communication. The sender doesn't need to know the exact addresses of individual processes within the group. Instead, the sender can use a group identifier or rely on the context to determine the destination, making it easier to address a group of processes collectively.

(c) For allowing a sender process to send messages to a receiver process that is allowed to migrate from one node to another.

*Explicit Addressing*: When a receiver process is allowed to migrate across nodes, it's crucial to have a globally unique identifier (GUID) that remains constant regardless of the process's location. GUIDs ensure that the sender can address the receiver accurately, even as it migrates between nodes. This mechanism provides the necessary flexibility and precision.

(d) For allowing a sender process to send messages to a receiver process that is allowed to migrate from one node to another and to allow the receiver process to return a reply to the sender process.

*Explicit Addressing*: Similar to scenario (c), when the receiver process can migrate, GUIDs are suitable to maintain a consistent identifier. This ensures that the sender can reach the receiver regardless of its location. Additionally, it allows the receiver process to use the same GUID to send replies back to the sender.

(e) For allowing a client process to receive service from any one of the several server processes providing that service.

*Explicit Addressing*: In this scenario, explicit addressing can be used to address individual server processes. To select a server to receive service from, the client can dynamically choose one of the available server addresses. This mechanism allows the client to make an explicit choice among several servers providing the same service.

1. Exercise 3.16 from Pradeep Sinha's book   
     
   (a) Last One:   
   *Publish-Subscribe (Pub-Sub) Messaging with "Last Value" or "Last Writer Wins" Policy*

In a Pub-Sub system, publishers broadcast messages to a topic, and subscribers express interest in specific topics. To achieve "Last One" semantics, you can implement a "Last Value" policy. In this approach, when a publisher sends a message to a topic, only the most recent message (the "last one") is delivered to subscribers interested in that topic.

(b) At Least Once:

*Message Queues with Acknowledgments*

Use message queues to implement "At Least Once" semantics. The sender places a message in a queue, and the recipient acknowledges the receipt of the message. If no acknowledgment is received, the sender can resend the message. This approach guarantees that the message will be delivered at least once, although it may result in duplicates.

(c) Exactly Once:

*Two-Message Reliable RPC or Three-Message Reliable RPC*

Implement distributed transaction protocols like *Two-Message Reliable* or *Three-Message Reliable* to achieve "Exactly Once" semantics. These protocols ensure that an operation or message is executed exactly once, even in the presence of failures. They involve a coordinator and participants, where all participants agree on whether to commit or abort a transaction, and the coordinator ensures that the decision is consistent across all participants.

1. Exercise 3.18 from Pradeep Sinha's book  
     
   (a) For making a request to a file server to read a file: *At Least Once*: Reading a file typically doesn't have side effects or critical dependencies. Ensuring that the file content is read at least once is sufficient for this scenario.

(b) For making a request to a file server to append some data to an existing file:   
*Exactly Once*: Appending data to a file can be sensitive to duplication. Ensuring that the append operation occurs exactly once is essential to maintain data integrity.

(c) For making a request to a compilation server to compile a file:   
*At Least Once*: The build process can often tolerate multiple compilation attempts, and it's preferable to ensure that the request is executed at least once to guarantee the compilation process proceeds.

(d) For making a request to a database server to update a bank account:   
*Exactly Once*: Updating a bank account involves financial transactions and must maintain data consistency. Exactly once semantics are critical to ensure the integrity of such transactions.

(e) For making a request to a database server to get the current balance of a bank account:   
*At Least Once*: Reading the current balance doesn't typically have side effects or critical dependencies. Ensuring that the request is executed at least once is sufficient for this scenario.

(f) For making a request to a booking server to cancel an already booked seat:   
*Exactly Once*: Canceling a booked seat is a critical operation in a booking system. Ensuring that the cancellation request is executed exactly once is crucial to maintaining the integrity of the booking system.

1. (a) Lamport's timestamps are useful for capturing the causal relationship between events of a distributed computation.

*True*: Lamport timestamps assign a unique identifier to each event in the system such that events with a higher timestamp happened after events with lower timestamps. This allows you to establish a partial order of events, capturing the causal relationships among them.

(b) If channels are FIFO (i.e., messages sent between any pair of processes are received in the same order they were sent), then messages will be automatically causally ordered.

*False*: Consider a scenario where Process A sends Message 1 to Process B and then Message 2 to Process C. Both messages are received in the same order they were sent (FIFO). However, if there is no mechanism to establish causal relationships between Message 1 and Message 2 (e.g., Lamport timestamps or vector clocks), it does not guarantee that the messages are causally ordered.

(c) If messages are causally ordered, then channels will be ensured to be FIFO.

*False*: Causal ordering of messages ensures that events are processed in a way that respects their causal relationships. Channels can still exhibit out-of-order delivery while maintaining causal order if messages take different routes with varying delays or if the network introduces reordering.

(d) Vector timestamps are useful for determining the causal relationship between events of a distributed computation.

*True*: Unlike Lamport timestamps, vector timestamps provide a total ordering of events while preserving causal relationships. They are a more comprehensive mechanism for capturing the causal dependencies among events and ensuring a consistent global order in distributed systems.

1. Assumptions:

- Each process Pi maintains a message sequence number Si.

- Whenever Pi sends a message, it also includes the sequence number Si that is incremented by 1 after each transmission.

- A buffer (implemented as an array) is available at the receiver end (Process Pj where i≠j) in order to store any out of order (not matching with the expected sequence no) messages temporarily.

**Algorithm:**

messageReceive (message M from process Pi) {

received\_seq\_no = extractSeq (M);

if (received\_seq\_no == expected\_seq\_no) {

deliver(M)

expected\_seq\_no = expected\_seq\_no+1;

}

else

bufferMessage(M)

}

1. (a)

Case 1: Assume a and b are two events of same process P1, where a happens before b. In a process the vector timestamp always incremented after each event. So, we can conclude that VTa < VTb.  
Case 2: Let, a is an event of process P1 where b is an event of process P2. Hence, a send message m and b receive that message. In case of event b, P2 updates its vector timestamp in such a way that VTa < VTb , as m is sent from a and m cannot be received before it is sent. Thus,we can say that, VTa < VTb.  
Case 3: Let us assume, for an event c, such that a → c and c → b. By using the same argument from case 2 and applying transitivity of vector timestamps. So, we can conclude that VTa < VTb.  
From all of this arguments, we can conclude that a → b if and only if VTa < VTb .

(b)

Let us assume that VTa < VTb. It implies that each component of timestamp VTa is less than or equal to corresponding component of timestamp VTb. So, a casually proceeds b. Thus, we can say that a and b are not concurrent.

Again, let us assume that VTa > VTb which implies that each component of timestamp VTb is less than or equal to corresponding component of timestamp VTa. So, b is casually proceeds a. Thus, a and b are not concurrent. So, we can conclude that a and b are concurrent if and only if VTa ≮ VTb and VTa ≯ VTb.

(c)

We know that, two vector timestamps are said to be equal if and only if every corresponding

component in their vector timestamps are equal. In fig4 we can see that the corresponding

component in the vector timestamps are not equal. Now, if a ≠ b then we can say that VTa ≠ VTb,

since any difference in the event ordering would be reflected in their respective timestamps and

thus VTa is not equal to VTb.

1. Exercise 4.3 from Pradeep Sinha's book   
     
   In a conventional procedure call model, it's common for the caller and callee procedures to use global variables to communicate with each other because they typically run within the same address space or memory context. In this model, both the caller and callee share the same memory, so they can easily access global variables to exchange information.  
   However, in the Remote Procedure Call (RPC) model, the caller and callee procedures may execute on different machines or in separate address spaces. In RPC, the caller and callee often run in different address spaces, which means they have separate memory spaces. Using global variables for communication between remote procedures could compromise system security and isolation. RPC involves communication over a network. Data sent between processes in RPC is typically serialized, transmitted over the network, and then deserialized at the destination. In distributed systems, multiple processes may concurrently call the same remote procedure. If global variables were used for communication, handling concurrent access and updates to those variables would be complex.  
   To address these challenges and ensure safe and reliable communication between remote procedures, RPC models use more structured and standardized mechanisms. These mechanisms include parameter passing, message serialization, and well-defined interfaces that abstract the communication process. These mechanisms enable remote procedures to communicate without the need for global variables.
2. Exercise 4.5 from Pradeep Sinha's book   
     
   Designing transparent Remote Procedure Call (RPC) mechanisms is a complex task, and achieving complete transparency is often challenging due to several fundamental issues. The primary goal of RPC is to make remote procedure calls look like local calls. However, the network introduces latency, potential failures, and varying communication costs. Achieving complete transparency over the network is challenging. While RPC mechanisms can hide many network-related details, they cannot completely eliminate network-specific issues such as delays, timeouts, and packet loss. Application developers may need to handle these issues explicitly.

Transparent RPC mechanisms often aim to abstract away the complexities of handling failures. However, failures can occur at various levels, including network failures, server crashes, and timeouts. Transparent RPC mechanisms must serialize parameters and results to transmit them over the network. This serialization may not support all data types or complex data structures transparently. RPC systems aim to work across heterogeneous environments with different architectures, platforms, and programming languages. Achieving complete transparency in such diverse settings is challenging. Transparent RPC should allow clients to call remote procedures without knowing their physical location. Achieving complete location transparency in highly dynamic environments or in cases where services migrate across nodes can be complex.

In summary, while RPC mechanisms aim to provide transparency and make remote calls appear as if they are local, achieving complete transparency is often not feasible due to inherent challenges like network characteristics, failures, data serialization, heterogeneity, and dynamic environments. Developers must be aware of these limitations and be prepared to handle them explicitly in their applications. Transparent RPC mechanisms can greatly simplify distributed application development, but they cannot completely eliminate the need for understanding and managing the distributed nature of the system.

1. Exercise 4.12 from Pradeep Sinha's book   
     
   **Stateful Servers**: Stateful servers maintain information about the current state of a client's session or interaction. They remember past interactions with clients and store data related to those interactions. Each client request is processed in the context of its previous interactions.

1. Stateful servers store information about client sessions, allowing clients to continue where they left off.

2. Managing session state requires more infrastructure and potentially introduces scalability challenges.

2. Stateful servers can be more efficient in terms of processing requests since they have context about the client's state.

**Stateless Servers:** Stateless servers do not retain information about client sessions between requests. Each client request is processed independently without any reference to previous interactions. All necessary information must be included in the client request.  
1. Stateless servers do not maintain any session-specific data between requests.

2. They are simpler to design, implement, and scale because there is no need to manage client session data.

3. Stateless servers may require redundant data transmission since clients need to include all necessary context with each request.

While stateful servers may provide an easier programming paradigm and can be more efficient in certain cases, they come with increased complexity, potential scalability challenges, and reduced fault tolerance. Stateless servers offer advantages in terms of scalability, fault tolerance, simplicity, and client flexibility, making them a preferred choice for many distributed applications, especially those designed for the web and cloud environments. Ultimately, the choice between stateful and stateless servers depends on the specific requirements and trade-offs of the application.  
Stateless servers are inherently easier to scale horizontally because they don't maintain session state. This makes it simpler to distribute client requests across multiple servers to handle high loads. Stateless servers are more fault-tolerant because they do not rely on maintaining session state. If one server fails, a new server can take over without losing client session information. Stateless servers work well with load balancers, which can distribute requests evenly among multiple servers. Load balancing is critical for achieving high availability and efficient resource utilization. Stateless servers simplify application logic and reduce the complexity of handling session state. They also ensure that each request is treated independently, which can lead to more predictable behavior. Stateless servers empower clients to manage their session state, giving clients more control and flexibility. Clients can easily switch between servers if necessary. Some communication protocols, such as HTTP, are inherently stateless. Designing applications to be stateless aligns well with these protocols, making it easier to build web-based and RESTful services.

1. Exercise 4.14 from Pradeep Sinha's book   
     
   To design remote procedures offered by a server in a way that ensures interleaved and concurrent requests from different clients do not interfere with each other we can employ several strategies and techniques.  
   Implement synchronization mechanisms to protect critical sections of code that access shared resources or data. Use techniques like mutexes, semaphores, or locks to ensure that only one thread or process can access a shared resource at a time. This prevents data corruption and race conditions. Implement synchronization mechanisms to protect critical sections of code that access shared resources or data. Use techniques like mutexes, semaphores, or locks to ensure that only one thread or process can access a shared resource at a time. This prevents data corruption and race conditions.
2. Exercise 4.19 (do not worry about last-one semantics) from Pradeep Sinha's book  
     
   (a) For making a request to a time server to get the current time:  
   *Last-One*

Last-One semantics ensure that you receive the last response from the server, which corresponds to the most up-to-date time.  
  
(b) For making a request to a node's resource manager to get the current status of resource availability of its node:  
*May-Be*  
Resource availability may change frequently, but missing occasional updates may not have significant consequences. Using May-Be semantics allows for efficient querying of resource status without excessive overhead.  
  
(c) For periodically broadcasting the total number of current jobs at its node by a process manager in a system where process managers of all nodes mutually cooperate to share the overall system load:  
*Last-Of-Many*  
Last-Of-Many semantics ensures that the broadcasted information is not lost, and all cooperating process managers eventually receive the total number of current jobs. This is important for load balancing and coordination.

(d) For making a request to a computation server to compute the value of an equation:  
*At-Least-Once*  
It's important to ensure that the computation request is received and processed by the server. At-Least-Once semantics guarantee that the request is executed at least once, even if there are network or server failures.

(e) For making a request to a booking server to get the current status of seat availability:  
*May-Be*  
Seat availability may change frequently, and missing occasional updates is generally acceptable. May-Be semantics provide efficient querying without adding significant overhead.

(f) For making a request to a booking server to reserve a seat:  
*Exactly-Once*  
Booking a seat should ensure that the reservation is processed only once to avoid double bookings and ensure consistency.  
  
(g) For making a request to a file server to position the read-write pointer of a file to a specified position  
*May-Be*  
Positioning the read-write pointer is typically an idempotent operation, and missing occasional updates may not have significant consequences.  
  
(h) For making a request to a file server to append a record to an existing file  
*Exactly-Once*  
Appending a record should be performed exactly once to avoid data duplication or corruption.

(i) For making a request to a name server to get the location of a named object in a system that does not support object mobility:  
*Last-One*  
In a non-mobile system, the location of named objects is relatively stable, and it's important to receive the most recent information.  
  
(j) For making a request to a name server to get the location of a named object in a system that supports object mobility:  
*At-Least-Once*  
In a system with object mobility, the location of objects may change frequently due to migration. At-Least-Once semantics ensure that you eventually receive location updates even if objects move.