ID2201 Distributed Systems

Section A : example questions

A:1 1p. What aspect makes distributed systems most different from non-distributed systems?

$\boxed{\text{A}}$ have to handle concurrency

$\boxed{\text{B}}$ privacy and authentication

$\boxed{\textbf{C}}$ harder to handle failure

$\boxed{\text{D}}$ marshaling of data structures

A:2 1p. What is meant by access transparency?

$\boxed{\text{A}}$ remote resources are accessed using location independent names

$\boxed{\textbf{B}}$ local and remote resources are accessed using the same operations

$\boxed{\text{C}}$ a replicated resource is accessed exactly as if it was a single object

$\boxed{\text{D}}$ a resource will handle all requests equally independent of location of client

A:3 1p. What is meant by location transparency?

$\boxed{\textbf{A}}$ remote resources are accessed using location independent names

$\boxed{\text{B}}$ local and remote resources are access using the same operation

$\boxed{\text{C}}$ a replicated resources is accessed exactly as if it was a single object

$\boxed{\text{D}}$ a resource will handle all request equal independent of location of client

A:4 1p. What is meant by concurrency transparency?

$\boxed{\text{A}}$ threads are allowed to access shared data structures

$\boxed{\textbf{B}}$ processes can access resources without interfering with each other

$\boxed{\text{C}}$ a replicated resources is accessed exactly as if it was a single object

$\boxed{\text{D}}$ new nodes can be added to a system without changing the application

A:5 1p. What is meant by failure transparency?

$\boxed{\textbf{A}}$ failures are concealed for the users of a resource

$\boxed{\text{B}}$ resource errors will raise exception that can be handled by the user

$\boxed{\text{C}}$ resources can fail but only by crashing

$\boxed{\text{D}}$ a robust system where resources will not fail

A:6 1p. What is meant by replication transparency?

$\boxed{\text{A}}$ processes have knowledge of replication scheme and can take advantage of it

$\boxed{\textbf{B}}$ users of a resource access it as if it was not replicated

$\boxed{\text{C}}$ data is immutable and can be serialized on disk

$\boxed{\text{D}}$ replies to queries are copied and logged to avoid dirty reads

A:7 1p. What is significant for a client server architecture?

$\boxed{\textbf{A}}$ the client is the active part

$\boxed{\text{B}}$ servers have more execution power

$\boxed{\text{C}}$ several clients but only one server

$\boxed{\text{D}}$ the server is the active part

A:8 1p. What would we call a system where one node is always reacting on requests and other nodes only communicate with this node?

$\boxed{\text{A}}$ an asynchronous system

$\boxed{\textbf{B}}$ a client server system

$\boxed{\text{C}}$ a peer-to-peer system

$\boxed{\text{D}}$ a synchronous system

A:9 1p. What is significant for a peer-to-peer architecture.

$\boxed{\text{A}}$ no single node may initiate an operation

$\boxed{\text{B}}$ nodes are structured in a hierarchy of client and servers

$\boxed{\text{C}}$ all nodes can communicate directly with all other nodes in the network

$\boxed{\textbf{D}}$ all nodes are active and can be the initiator of operations

A:10 1p. What do we know in a synchronous system?

$\boxed{\text{A}}$ that all operations will take equal amount of time

$\boxed{\text{B}}$ exactly how long time it takes to deliver a message

$\boxed{\text{C}}$ that all messages will be delivered

$\boxed{\textbf{D}}$ the upper bound of the time to perform an operation

A:11 1p. What do we call a system where the upper bound of the time for operations and message delivery are known?

$\boxed{\text{A}}$ a fault tolerant system

$\boxed{\text{B}}$ a high speed system

$\boxed{\text{C}}$ an asynchronous system

$\boxed{\textbf{D}}$ a synchronous system

A:12 1p. What do we call a system where the maximum times for operations and message delivery <u>are not</u> known?

$\boxed{\text{A}}$ a fault tolerant system

$\boxed{\text{B}}$ a real time system

$\boxed{\textbf{C}}$ an asynchronous system

$\boxed{\text{D}}$ a synchronous system

A:13 1p. What do we call a system where the maximum times for operations and message delivery are known?

$\boxed{\text{A}}$ a fault tolerant system

$\boxed{\text{B}}$ a real time system

$\boxed{\text{C}}$ an asynchronous system

$\boxed{\textbf{D}}$ a synchronous system

A:14 1p. What is the most significant difference between a synchronous and an asynchronous distributed system?

$\boxed{\text{A}}$ how concurrency is implemented

$\boxed{\text{B}}$ location transparent operations

$\boxed{\text{C}}$ the ability to synchronize clocks

$\boxed{\textbf{D}}$ possibility of perfect failure detectors

A:15 1p. Why is it impossible to implement a perfect failure detector in an asynchronous system?

$\boxed{\text{A}}$ messages can take up to 200ms to reach its destination

$\boxed{\text{B}}$ clocks are not perfectly synchronized

$\boxed{\textbf{C}}$ maximum delay for a reply is not known

$\boxed{\text{D}}$ UDP datagrams can be dropped

A:16 1p. How can we implement a perfect failure detector in an asynchronous system?

$\boxed{\text{A}}$ use remote operating system support to signal failure

$\boxed{\text{B}}$ use heartbeats between processes

$\boxed{\textbf{C}}$ it is not possible

$\boxed{\text{D}}$ use TCP since it will reliably detect failures

A:17 1p. What is provided by UDP?

$\boxed{\text{A}}$ acknowledged delivery of messages

$\boxed{\text{B}}$ a one-way byte stream between two processes

$\boxed{\textbf{C}}$ a best effort delivery of messages to a process

$\boxed{\text{D}}$ ordered delivery of messages

A:18 1p. What does UDP give us that is not provided by IP?

$\boxed{\text{A}}$ larger messages

$\boxed{\textbf{B}}$ port addressing

$\boxed{\text{C}}$ flow control

$\boxed{\text{D}}$ congestion control

A:19 1p. What is provided by TCP?

$\boxed{\text{A}}$ a guarantee that messages will always reach its destination

$\boxed{\textbf{B}}$ a full-duplex stream between two processes

$\boxed{\text{C}}$ a best effort delivery of messages

$\boxed{\text{D}}$ transactional control and persistence

A:20 1p. What is <u>not</u> provided by TCP?

$\boxed{\textbf{A}}$ a guaranteed delivery of messages

$\boxed{\text{B}}$ a full-duplex stream between two processes

$\boxed{\text{C}}$ flow control, not to overflow the receive buffer

$\boxed{\text{D}}$ congestion control, to avoid network congestion

A:21 1p. When is the TCP window size a limiting factor for high capacity communication?

$\boxed{\textbf{A}}$ over a long fat communication link

$\boxed{\text{B}}$ over a long thin communication link

$\boxed{\text{C}}$ when a message is short

$\boxed{\text{D}}$ never

A:22 1p. Which address can be found in the TCP header?

$\boxed{\text{A}}$ there is no address field in the TCP header

$\boxed{\text{B}}$ the network IP address

$\boxed{\text{C}}$ a process identifier

$\boxed{\textbf{D}}$ the port number

A:23 1p. What is a good reason for choosing UDP rather than TCP?

$\boxed{\text{A}}$ you need to know that a message is handled by the remote application

$\boxed{\text{B}}$ you have large messages or a sequence of messages

$\boxed{\textbf{C}}$ you have small messages that should be sent with little delay

$\boxed{\text{D}}$ UDP will guarantee the delivery of a message

A:24 1p. What is a good reason for choosing TCP rather than UDP?

$\boxed{\text{A}}$ you need to know that a message is handled by the remote application

$\boxed{\textbf{B}}$ you have large messages or a sequence of messages

$\boxed{\text{C}}$ you have a small message that should be sent with little delay

$\boxed{\text{D}}$ TCP will guarantee the delivery of a message

A:25 1p. What is the maximum TCP capacity in a 100 Mbps link with 250 ms round trip latency using a 64 Kbyte window size?

$\boxed{\text{A}}$ 40 Mbps

$\boxed{\textbf{B}}$ 2 Mbps

$\boxed{\text{C}}$ 10 Mbps

$\boxed{\text{D}}$ 512 Kbps

A:26 1p. What is the maximum TCP capacity in a 100 Mbps link with 25 ms round trip latency using a 64 Kbyte window size?

$\boxed{\textbf{A}}$ 20 Mbps

$\boxed{\text{B}}$ 2 Mbps

$\boxed{\text{C}}$ 80 Mbps

$\boxed{\text{D}}$ 512 Kbps

A:27 1p. One can determine how large the receiver window should be, in for example a TCP connection, in order to maximize he capacity of a link. How is this calculated?

$\boxed{\text{A}}$ capacity of link times distance to receiver

$\boxed{\text{B}}$ latency of link squared

$\boxed{\textbf{C}}$ capacity of link times round trip latency of the link

$\boxed{\text{D}}$ time to send a message times the latency of the link

A:28 1p. The limiting factor for capacity in a TCP connection, T bps, where the link capacity is C bps, the round trip time R sec and the receiver window W bits is:

$\boxed{\text{A}}$ the greater of $W * R$ and $C * R$

$\boxed{\text{B}}$ C

$\boxed{\textbf{C}}$ the lesser of C and W/R

$\boxed{\text{D}}$ $C * R$

A:29 1p. When estimating the maximum capacity of a TCP connection you need to know:

$\boxed{\text{A}}$ the number of router hops

$\boxed{\textbf{B}}$ the minimum link capacity and the round trip

$\boxed{\text{C}}$ only the minimum link capacity

$\boxed{\text{D}}$ TCP capacity is defined by the specification RFC 1149

A:30 1p. What does it mean that you receive a TCP acknowledgment form a node?

$\boxed{\text{A}}$ A sequence of bytes have been successfully sent by your node.

$\boxed{\mathbf{B}}$ A sequence of bytes have been received by the TCP process on the node.

$\boxed{\text{C}}$ A sequence of bytes have been read by the receiving application process on the node

$\boxed{\text{D}}$ A sequence of bytes have been forwarded to the next router (that might be the final node).

A:31 1p. What is significant for synchronous communication?

$\boxed{\mathbf{A}}$ the send operation blocks and waits for the receive operation

$\boxed{\text{B}}$ a sent message will always be received

$\boxed{\text{C}}$ send operations can only be performed with certain intervals

$\boxed{\text{D}}$ sender and receiver must have synchronized clocks

A:32 1p. What is significant for asynchronous communication?

$\boxed{\text{A}}$ the receiver blocks and waits until a message arrives

$\boxed{\text{B}}$ a sent message will always be received

$\boxed{\text{C}}$ implemented over UDP

$\boxed{\mathbf{D}}$ a sender can continue without waiting for the receive operation

A:33 1p. What is true for asynchronous communication?

$\boxed{\text{A}}$ requires asymmetric clocks

$\boxed{\mathbf{B}}$ sender does not block waiting for receiver

$\boxed{\text{C}}$ sending to a process that is not in the receive state will cause an exception

$\boxed{\text{D}}$ receiver does not block waiting for sender

A:34 1p. Can a synchronous communication interface be used in an asynchronous way?

$\boxed{\mathbf{A}}$ yes, by performing the send operation in a separate thread

$\boxed{\text{B}}$ no, the send operation will block and stop the whole execution

$\boxed{\text{C}}$ no, synchronizing clocks will always take time

$\boxed{\text{D}}$ yes, by minimizing the latency for the receive operation

A:35 1p. What is the difference between the two nodes that are communicating using a stream socket API?

$\boxed{\text{A}}$ the node that connected to the server is the node that should be the first to close the connection

$\boxed{\text{B}}$ the server side should be the first to close the connection

$\boxed{\textbf{C}}$ nothing, they have equal rights and responsibilities

$\boxed{\text{D}}$ the node that connected to the server must send the first message

A:36 1p. The Erlang call `gen_tcp:recv(Socket, 0)` will return:

$\boxed{\textbf{A}}$ the first part (possibly the whole) message sent to `Socket`

$\boxed{\text{B}}$ list of 64K characters

$\boxed{\text{C}}$ all messages sent to `Socket` before closed by the sender

A:37 1p. A server using a stream socket API can communicate with several clients and separate the streams by:

$\boxed{\textbf{A}}$ creating a new stream socket for each connecting client

$\boxed{\text{B}}$ having each client tag messages with the hash of the IP address

$\boxed{\text{C}}$ assigning unique names to each client

$\boxed{\text{D}}$ handling one client at a time

A:38 1p. What is the purpose of the marshaling procedure?

$\boxed{\text{A}}$ consistency checking of type information

$\boxed{\text{B}}$ to compress data structures

$\boxed{\text{C}}$ to protect an application from unauthorized requests

$\boxed{\textbf{D}}$ to encode application layer structures in an external form

A:39 1p. What is a gossip protocol?

A A social order multicast protocol that provides causal ordering of messages in community networks of overlay routed nodes.

B A protocol that provides congestion avoidance and flow-control.

C A protocol where peers randomly exchange messages to achieve, for example, unreliable multicast.

D A reliable multicast protocol that provides FIFO order within a specified group.

A:40 1p. What is meant by *time uncoupling* in a communication framework?

A The sender does not wait for a acknowledgment from the receiver.

B Sender and receiver need not be active at the same time.

C A sender does not need to know the name or identifier of the receiver.

D Messages are resent if lost, providing a reliable service.

A:41 1p. What is meant by *space uncoupling* in a communication framework?

A The sender does not wait for a acknowledgment from the receiver.

B Sender and receiver need no be active at the same time.

C A sender does not need to know the name or identifier of the receiver.

D Messages are resent if lost, providing a reliable service.

A:42 1p. What is meant by *causal ordering* in a communication framework?

A If a client is delivered two messages *m1* and *m2*, then *m1* could not have been sent by someone after having being delivered *m2*.

B If a sender is sending two messages *m1* and *m2*, then a receiver will be delivered *m1* before *m2*.

C If two clients send messages *m1* and *m2* independently of each other, then a receiver will receive these in the real-time order in which the send operations where issues.

D Clients will be able to send messages in a *round-robin* order thus preserving inter-ordering relationships.

A:43 1p. What is meant by an idempotent operation?

A An operation that can be performed several times with the same effect as being performed once.

$\boxed{\text{B}}$ An operation that should never be performed more than once to guarantee the operational semantics.

$\boxed{\text{C}}$ An operation that does not change the state of a server, for example a read operation.

$\boxed{\text{D}}$ An operation that could possibly make the server crash and therefore should only be used if the server keeps a state in persistent memory.

A:44 1p. How are arguments passed in Java RMI?

$\boxed{\text{A}}$ call by reference only

$\boxed{\text{B}}$ call by copy only

$\boxed{\textbf{C}}$ remote objects as reference, all other as copies

$\boxed{\text{D}}$ serialized objects as reference, all other as copy

A:45 1p. Which invocation semantic is provided by Java RMI?

$\boxed{\text{A}}$ no guarantees

$\boxed{\text{B}}$ at least once

$\boxed{\text{C}}$ exactly once

$\boxed{\textbf{D}}$ at most once

A:46 1p. What level of transparency is provided by method invocation in Java RMI?

$\boxed{\text{A}}$ access transparency only since we need to explicitly describe the location

$\boxed{\textbf{B}}$ location transparency only since we need to capture remote exceptions

$\boxed{\text{C}}$ access and location transparency

$\boxed{\text{D}}$ neither access nor location transparency

A:47 1p. If a RPC call with *at least once* semantics fails, we know that:

$\boxed{\text{A}}$ the call has been executed at least once

$\boxed{\textbf{B}}$ the call might have been executed at least once

C the call has not been executed at all

D the call has either been executed once or not at all

A:48 1p. What is given by *at least once* RPC semantics?

A invocation is guaranteed to happen once

B if no failure is reported the call has been invoked at least once

C even if a failure occurs the call has been invoked at least once

D if no failure is reported the call has been invoked exactly once

A:49 1p. How do Erlang processes communicate?

A only through global storage

B synchronous message passing

C asynchronous message passing

D modifying shared data structures

A:50 1p. Does Erlang provide a form of location transparency?

A No - you always need to know the node address of a process.

B Yes - there are no explicit node addresses in the system.

C Yes - a process can use a process identifier without having to know the address of the node where the process lives.

D No - since the address of a registered process contains the IP address of the node, there is no transparency in the system.

A:51 1p. How is the destination defined in an Erlang send operation?

A as a process identifier or the ip-address and port of the receiver

B as a process identifier or a local or remote registered name

C a globally registered name

D an identifier and the binder to contact

A:52 1p. What will prevent an Erlang process from sending a message to a process on another Erlang node?

$\boxed{\mathbf{A}}$ the two nodes do not have the same secret cookie

$\boxed{\text{B}}$ the message contains process identifiers

$\boxed{\text{C}}$ the nodes are not owned by the same user

$\boxed{\text{D}}$ the destination node is not registered

A:53 1p. Message sending in Erlang provides the following semantics:

$\boxed{\text{A}}$ if a message is lost the send operation fails

$\boxed{\text{B}}$ guaranteed delivery of message

$\boxed{\text{C}}$ guaranteed, but possibly duplicated, delivery of message

$\boxed{\mathbf{D}}$ best effort FIFO delivery of message

A:54 1p. Why is a remote object passed as a reference and not as a copy in Java RMI?

$\boxed{\text{A}}$ more efficient since it requires less bytes to be sent

$\boxed{\text{B}}$ wrong, they are copied

$\boxed{\mathbf{C}}$ the object contains a mutable state that should not be duplicated

$\boxed{\text{D}}$ objects are always passed as reference

A:55 1p. Can we have a circular construction in Erlang?

$\boxed{\text{A}}$ no, data structures are always copied

$\boxed{\text{B}}$ yes, simply create a circular data structure: X = {foo, X}

$\boxed{\text{C}}$ no, Erlang is functional language without any circular structures

$\boxed{\mathbf{D}}$ yes, processes can refer to each other in a circular way

A:56 1p. What happens with Erlang processes that are suspended on a receive statement but no longer referenced by any process?

$\boxed{\text{A}}$ the will be removed by the garbage collector

$\boxed{\textbf{B}}$ they will remain in the system until explicitly killed

$\boxed{\text{C}}$ they will remain if registered, otherwise removed by the garbage collector

$\boxed{\text{D}}$ not possible, a process is always referenced by the creator of the process

A:57 1p. Assume we have three Erlang processes: A, B and C. Also assume that A sends a message, m1, to B and then sends a message, m2, to C. B is suspended, waiting for a message from A that it will send to C. What is guaranteed by the Erlang semantics.

$\boxed{\text{A}}$ C will receive m1 before m2.

$\boxed{\text{B}}$ if C receives m2 it will also receive m1

$\boxed{\textbf{C}}$ no guarantees on receiving nor ordering of messages

$\boxed{\text{D}}$ C will receive both messages but the order is not guaranteed

A:58 1p. Can we implement a RPC system with *exactly-once* semantics in an asynchronous system with non-failing nodes but unreliable networks?

$\boxed{\text{A}}$ no, since messages can be lost a reply is not guaranteed to reach the client

$\boxed{\textbf{B}}$ yes, if the client keeps re-sending a uniquely tagged request until a reply is received and a server keeps track of all handled request in order not to duplicate a request

$\boxed{\text{C}}$ yes, simply resend the request until an acknowledgment is received

$\boxed{\text{D}}$ no, we can only achieve at-most-once or at-least-once but not both

A:59 1p. What is the difference when an error is reported for an at-least-once and at-most-once remote procedure call?

$\boxed{\text{A}}$ in the at-most-once case, the remote call will not have been executed

$\boxed{\text{B}}$ nothing, in either case we don't know if the remote call has been executed

$\boxed{\text{C}}$ in the at-least-once case, the remote call will not have been executed at all

$\boxed{\textbf{D}}$ in the at-least-once case, the call could have been executed more than once

A:60 1p. How can we create a message authentication code?

$\boxed{\text{A}}$ taking the message digest of the message

$\boxed{\textbf{B}}$ taking the message digest of a shared secret and the message

$\boxed{\text{C}}$ encrypting the message using counter mode

$\boxed{\text{D}}$ encrypting the message using the public key of the receiver

A:61 1p. What is the purpose of a message authentication code?

$\boxed{\textbf{A}}$ provide integrity

$\boxed{\text{B}}$ provide non-repudiation

$\boxed{\text{C}}$ detect transmission errors

A:62 1p. What is the structure of a certificate?

$\boxed{\textbf{A}}$ a name and a public key with a signed integrity code

$\boxed{\text{B}}$ a name and a private key encrypted with a shared secret

$\boxed{\text{C}}$ a name and shared secret singed with a trusted public key

$\boxed{\text{D}}$ a public and private key encrypted with a shared secret

A:63 1p. What can be done with public-key encryption that can not be done with shared-secret encryption?

$\boxed{\text{A}}$ secure hashing

$\boxed{\text{B}}$ guarantee privacy of large messages

$\boxed{\text{C}}$ user authentication

$\boxed{\textbf{D}}$ digital signatures

A:64 1p. What is the purpose of a PKI certificate?

$\boxed{\text{A}}$ to send the private key to the receiver of a signed message

$\boxed{\text{B}}$ to protect the private key by encryption

$\boxed{\text{C}}$ to associate a shared secret to a public key

$\boxed{\textbf{D}}$ to verify the public key of a user

A:65 1p. The reason to combine symmetric and asymmetric keys during a secure session is due to:

$\boxed{\textbf{A}}$ a secure way to exchange the symmetric key is needed

$\boxed{\text{B}}$ the computational cost to use a symmetric key is too high

$\boxed{\text{C}}$ the public key is sent protected by a symmetric session key

$\boxed{\text{D}}$ none of the above

A:66 1p. How does a node authenticate a user using Kerberos?

$\boxed{\text{A}}$ the node request a password that is checked by the KDC

$\boxed{\textbf{B}}$ the user presents an encrypted ticket received from the KDC

$\boxed{\text{C}}$ the node receives a request from the KDC to contact the user

$\boxed{\text{D}}$ the node requests the password from the KDC

A:67 1p. What is a soft link in a file system?

$\boxed{\text{A}}$ a mapping of a name to a file

$\boxed{\text{B}}$ a link from a file to a path position

$\boxed{\textbf{C}}$ a path that is resolved to another path

$\boxed{\text{D}}$ a link between two files

A:68 1p. What is a hard link in a file system?

$\boxed{\textbf{A}}$ a mapping of a name to a file identifier

$\boxed{\text{B}}$ a link from a file descriptor to a path position

$\boxed{\text{C}}$ a path that is resolved to another path

$\boxed{\text{D}}$ a link between two files

A:69 1p. Can two Unix processes simultaneous write to different positions in a single file?

A Yes, the two processes will have their own file table entries.

B No, the shared i-node contains a single offset pointer.

C Only one process will have write privilege.

D Yes, but only if we operate using NFS, AFS or similar network file system.

A:70 1p. What is the purpose of the Unix lseek operation?

A find the position of string in file

B set the read/write pointer of an opened file

C find file descriptor matching name

D search for name mapped to given file descriptor

A:71 1p. How is a NFS client-side cache entry validated?

A if the call-back promise is not older than t seconds

B if the difference between the server modification time and the client modification time is less that t seconds

C if the server modification time is equal to the client modification time

D if the validity was checked less than t seconds ago or if the server modification time is equal to the client modification time

A:72 1p. How is authentication control handled in Sun NFS?

A server keeps a log and only commit changes when client does final authentication

B authentication is provided by RPC in each operation

C authentication is done when file is opened

D authentication left to client, the server will trust all client operations

A:73 1p. How does a NFS server know at what position to read and write to?

A it keeps a file table entry with a read/write position

B NFS only allow read operations and therefore does not need position information

$\boxed{\text{C}}$ not needed since all read and write operations are on a whole file

$\boxed{\textbf{D}}$ each read and write operation holds the position

A:74 1p. How is AFS client side caching implemented?

$\boxed{\textbf{A}}$ the server promise to notify the client if a file is modified by another client

$\boxed{\text{B}}$ the client will check the server status with each write operation

$\boxed{\text{C}}$ the client will check the last modified time of the server before each read

$\boxed{\text{D}}$ the consistency is checked only when a file is opened

A:75 1p. Can two client have an inconsistent view of a file using AFS?

$\boxed{\text{A}}$ no, updates are not performed unless all call-backs have been confirmed

$\boxed{\text{B}}$ yes, since consistency is only checked when the file is opened

$\boxed{\text{C}}$ no, clients will check server status with each read and write operation

$\boxed{\textbf{D}}$ yes, if a call-back message is lost a cached copy can be used although the original has been modified

A:76 1p. What is the advantage of using AFS over NFS?

$\boxed{\text{A}}$ AFS is better in a local area network

$\boxed{\text{B}}$ AFS server is stateless

$\boxed{\text{C}}$ AFS guarantees one-copy semantics

$\boxed{\textbf{D}}$ the client need not periodically check validity

A:77 1p. What is the difference between a URL and a URN?

$\boxed{\text{A}}$ URN refer to a location dependent resource

$\boxed{\text{B}}$ URL resolve into any replica of a resource matching the URI

$\boxed{\textbf{C}}$ URN resolve into any replica of a resource matching the URI

$\boxed{\text{D}}$ URLs are a subset of URNs

A:78 1p. How are inconsistencies of the resolver cache handled in the DNS architecture?

$\boxed{\mathbf{A}}$ each entry has a time-to-live

$\boxed{\text{B}}$ changes in DNS servers will be pushed to resolvers with cached values

$\boxed{\text{C}}$ the resolver will check if an entry has changed since requested

$\boxed{\text{D}}$ a server will redirect requests to the new location

A:79 1p. What is the advantage of using recursive navigation for DNS queries?

$\boxed{\mathbf{A}}$ servers can hide internal DNS hierarchy

$\boxed{\text{B}}$ less burden placed on servers, most work done by client

$\boxed{\text{C}}$ improves resolver caching

$\boxed{\text{D}}$ server does not need to hold a request state

A:80 1p. How are inconsistent cached entries removed from a DNS resolver?

$\boxed{\text{A}}$ a resolver will not cache entries that could become inconsistent

$\boxed{\text{B}}$ authorized DNS server will actively revoke previous replies

$\boxed{\text{C}}$ updated entries are pushed from servers to resolvers

$\boxed{\mathbf{D}}$ all entries have an expiration time set when cached

A:81 1p. What is the advantage of a flat name space?

$\boxed{\mathbf{A}}$ it is a simple solution

$\boxed{\text{B}}$ the space is easily divided among administrative actors

$\boxed{\text{C}}$ it is easy to handle infinite name spaces

$\boxed{\text{D}}$ names take up less room when stacked

A:82 1p. How can we make two computer clocks perfectly synchronized?

$\boxed{\text{A}}$ use atomic clocks

$\boxed{\text{B}}$ use the Stanford network synchronization protocol

$\boxed{\text{C}}$ send a message containing a time stamp every microsecond

$\boxed{\textbf{D}}$ we can not

A:83 1p. What accuracy can be provided using Christian's algorithm?

$\boxed{\text{A}}$ $\pm(T_{round} \times 2 - \text{minimum latency})$

$\boxed{\text{B}}$ $\pm(T_{round})$

$\boxed{\text{C}}$ $\pm(T_{round} \div 2)$

$\boxed{\textbf{D}}$ $\pm(T_{round} \div 2 - \text{minimum latency})$

A:84 1p. What is the purpose of the Berkeley algorithm?

$\boxed{\text{A}}$ to synchronize a node with a stratum-1 source

$\boxed{\textbf{B}}$ to perform internal synchronization

$\boxed{\text{C}}$ to compensate for wide area network delays

$\boxed{\text{D}}$ to divide a network into a hierarchy of synchronization nodes

A:85 1p. What does the reply from a NTP server contain?

$\boxed{\textbf{A}}$ send and receive time of request and send time of reply

$\boxed{\text{B}}$ the delta of the client time stamp and the server time

$\boxed{\text{C}}$ the send time of the reply

$\boxed{\text{D}}$ the latency of the request and the send time of the reply

A:86 1p. When is it better to use the Berkeley algorithm rather than NTP for clock synchronization?

$\boxed{\text{A}}$ never, NTP is always more accurate and more efficient

$\boxed{\text{B}}$ when a node needs to be well synchronized with a stratum-1 server

$\boxed{\text{C}}$ if we have a synchronous network

$\boxed{\textbf{D}}$ when implementing internal synchronization

A:87 1p. Assume that a NTP server can not respond to a received request in less than 40 ms, how does this influence the accuracy of synchronizing clients?

$\boxed{\text{A}}$ synchronization is limited by the larger of network delay and server delay

$\boxed{\textbf{B}}$ not at all

$\boxed{\text{C}}$ synchronization is limited by the network delay plus 40 ms

$\boxed{\text{D}}$ synchronization is limited by the network delay plus 20 ms

A:88 1p. At time 117 you receive a NTP reply with the following information: request sent at 82, received at 111, reply sent at 120. How should you adjust your time?

$\boxed{\textbf{A}}$ advance 16 steps

$\boxed{\text{B}}$ advance 21 steps

$\boxed{\text{C}}$ advance 9 steps

$\boxed{\text{D}}$ advance 32 steps

A:89 1p. What is true if A *happened before* B?

$\boxed{\text{A}}$ B is a consequence of A

$\boxed{\text{B}}$ A and B occurred in the same process

$\boxed{\textbf{C}}$ A must have occurred in real-time before B

$\boxed{\text{D}}$ it is unknown if A occurred in real-time before B

A:90 1p. What is true if A happened *real-time before* B?

$\boxed{\text{A}}$ B is a consequence of A

$\boxed{\text{B}}$ A and B occurred in the same process

$\boxed{\textbf{C}}$ A could have *happened before* B

$\boxed{\text{D}}$ B could have *happened before* A

A:91 1p. What is true for events A and B?

$\boxed{\textbf{A}}$ if A caused B then A *happened before* B

$\boxed{\text{B}}$ if A *happened before* B then A caused B

$\boxed{\text{C}}$ if A occurred in real-time before B then A *happened before* B

$\boxed{\text{D}}$ if A occurred in real-time before B then A caused B

A:92 1p. What can we know if we use Lamport clocks?

$\boxed{\text{A}}$ if $L(a) < L(b)$ then a *happened before* b

$\boxed{\textbf{B}}$ if $L(a) < L(b)$ then a could have caused b

$\boxed{\text{C}}$ if $L(a) < L(b)$ then a must have caused b

$\boxed{\text{D}}$ if $L(a) < L(b)$ then a occured in real-time before b

A:93 1p. What can we know if we use Lamport clocks?

$\boxed{\text{A}}$ if, and only if, $L(a) < L(b)$ then *a happened before* b

$\boxed{\text{B}}$ if, but not only if, $L(a) < L(b)$ then a *happened before* b

$\boxed{\textbf{C}}$ if, but not only if, a *happened before* b then $L(a) < L(b)$

$\boxed{\text{D}}$ if $L(a) = L(b)$ then $a = b$

A:94 1p. What can we conclude looking at the Lamport clock time stamps of two events?

$\boxed{\text{A}}$ if $L(a) < L(b)$ then a *happened after* b

$\boxed{\text{B}}$ if $L(a) > L(b)$ then b *happened before* a

$\boxed{\text{C}}$ if $L(b) = L(a)$ then we cannot conclude anything about a and b

$\boxed{\textbf{D}}$ if $L(a) \not< L(b)$ then a did not *happened before* b

A:95 1p. What is the most that we know if we use vector clocks?

$\boxed{\textbf{A}}$ $V(a) < V(b)$ if and only if a *happened before* b

$\boxed{\text{B}}$ if $V(a) < V(b)$ then a *happened before* b

$\boxed{\text{C}}$ if a *happened before* b then $V(a) < V(b)$

$\boxed{\text{D}}$ if $V(a) = V(b)$ then a and b are unordered

A:96 1p. Which is the most natural representation of a vector clock?

$\boxed{\text{A}}$ an integer

$\boxed{\textbf{B}}$ a record with one element per process

$\boxed{\text{C}}$ a reference to a causal related process

$\boxed{\text{D}}$ on a 64-bit architecture, a double float

A:97 1p. What is the difference between a Lamport clock and a vector clock?

$\boxed{\text{A}}$ nothing, two names for the same thing

$\boxed{\text{B}}$ only difference is that Lamport clocks are more efficient since they require smaller messages

$\boxed{\textbf{C}}$ only the vector clock gives a complete description of the ”happened before order”

$\boxed{\text{D}}$ only the Lamport clock gives a complete description of the ”happened before order”

A:98 1p. When is it problematic to use vectors clocks?

$\boxed{\textbf{A}}$ when we have a dynamic set of processes

$\boxed{\text{B}}$ if nodes are not synchronized using for example NTP

$\boxed{\text{C}}$ if we have an asynchronous system

$\boxed{\text{D}}$ when we do not have an elected leader process

A:99 1p. An alternative way of implementing a vector clock would be to keep a set of the highest counters seen from each process (including own), send it along with any message, update own counter and merging the own set and received set when a message is received. This would have the following advantage:

$\boxed{\textbf{A}}$ new processes can easily be added

$\boxed{\text{B}}$ the set is easier to represent

$\boxed{\text{C}}$ nothing, it is identical to vector clocks

$\boxed{\text{D}}$ events would become totally ordered

A:100 1p. If events in a given set are to be ordered in a total order that respect the *happened before* order what is the advantage of using vector clocks?

$\boxed{\text{A}}$ only vector clocks will give us the *happened before* order

$\boxed{\text{B}}$ only Lamport clocks will give us the *happened before* order

$\boxed{\text{C}}$ if tow events are unordered only vector clocks can order them

$\boxed{\textbf{D}}$ no advantage at all

A:101 1p. Is it possible to produce a total order of a set of events stamped with a Lamport time that does not violate any causal relationship?

$\boxed{\text{A}}$ no, only vector clocks can give us the total order

$\boxed{\text{B}}$ no, causal relationships will need real-time clock time stamps

$\boxed{\textbf{C}}$ yes, but only if we know how many processes that generated the events

A:102 1p. What is the definition of a consistent cut?

$\boxed{\text{A}}$ all events in the cut are strictly ordered in a *happened before* order

$\boxed{\text{B}}$ if e *happened before* f and e is in the cut then f is in the cut

$\boxed{\text{C}}$ if e and f are in the cut then e and f have a causal order

$\boxed{\textbf{D}}$ if e is in the cut and f happened-before e then f is in the cut

A:103 1p. Why is the notion of consistent cut important?

$\boxed{\text{A}}$ it defines a state that did occur during an execution

$\boxed{\textbf{B}}$ it defines a state that possibly occurred during an execution

$\boxed{\text{C}}$ it defines the difference between synchronous and asynchronous systems

$\boxed{\text{D}}$ used to resolve two-phase locking

A:104 1p. What is the definition of a stable global state predicate?

$\boxed{\mathbf{A}}$ if a system enters a state where the predicate holds true it will remain true in all future states

$\boxed{\text{B}}$ the predicate holds true in all consistent global states

$\boxed{\text{C}}$ the predicate will eventually hold true in all linearizations

$\boxed{\text{D}}$ the predicate will never hold true in any consistent state reachable from the original state

A:105 1p. What is the definition of a unstable global state predicate?

$\boxed{\text{A}}$ if a system enters a state where the predicate holds true it will remain true in all future states

$\boxed{\text{B}}$ the predicate holds true in all consistent global states

$\boxed{\mathbf{C}}$ the predicate could hold true in a state but then be false in future states

$\boxed{\text{D}}$ the predicate will never hold true in any consistent state reachable from the original state

A:106 1p. Give an example of a stable global state predicate.

$\boxed{\text{A}}$ non of the alternatives are stable

$\boxed{\mathbf{B}}$ deadlock

$\boxed{\text{C}}$ the cargo door is open at 10.000 meters

$\boxed{\text{D}}$ A is waiting for a message from B and B is waiting for a message from A

A:107 1p. What do we know if we record a snapshot using the algorithm by Chandy and Lamport?

$\boxed{\text{A}}$ the execution passed through the state described by the snapshot

$\boxed{\text{B}}$ a non-stable predicate that is true for the snapshot has also been true during the execution

$\boxed{\text{C}}$ if the algorithm terminates then the execution will also terminate

$\boxed{\mathbf{D}}$ there is a linearizations from the state described by the snapshot and a state that did occur in the execution

A:108 1p. Assuming that we collect all state transitions of nodes and have them tagged with vector clocks what can we then do?

$\boxed{\mathbf{A}}$ for any unstable predicates determine if it possibly was true during the execution

$\boxed{\text{B}}$ for any unstable predicates determine if it was true during the execution

$\boxed{\text{C}}$ determine all stable, but no unstable, predicates

$\boxed{\text{D}}$ determine termination but not deadlock

A:109 1p. How can we detect that a non-stable predicate definitely was true during an execution?

$\boxed{\text{A}}$ let each node evaluate the predicate based on local state

$\boxed{\mathbf{B}}$ generate all consistent runs and show that the predicate is true at one point in all

$\boxed{\text{C}}$ collect a snap-shot and examine if the predicate is true

$\boxed{\text{D}}$ since the predicate is non-stable it can not be determined

A:110 1p. What is the benefit of a reliable multicast?

$\boxed{\text{A}}$ messages are delivered within a upper bound time

$\boxed{\text{B}}$ the sender will not crash during an operation

$\boxed{\mathbf{C}}$ messages are guaranteed to be delivered to all correct processes

$\boxed{\text{D}}$ messages are delivered in total order

A:111 1p. Can we implement a reliable multicast using only basic multicast?

$\boxed{\text{A}}$ no, we need a reliable failure detector

$\boxed{\text{B}}$ yes, but only if we have network supported multicast

$\boxed{\text{C}}$ no, we need a synchronous communication network

$\boxed{\mathbf{D}}$ yes, by re-sending each received message to all other nodes

A:112 1p. How can we implement uniform reliable multicast given that we have reliable point to point message passing?

$\boxed{\mathbf{A}}$ Tag each message with a unique identifier and send it to each node. When receiving a new message you forward it to all nodes in the group and then deliver it to the application layer

$\boxed{\text{B}}$ Send message to each node and wait for acknowledgment, resend if needed.

$\boxed{\text{C}}$ You can't, it's impossible to implement in an asynchronous network.

$\boxed{\text{D}}$ Tag each message with a unique identifier and send it to each node. When receiving a new message you deliver it to the application layer and then forward it to all nodes in the group.

A:113 1p. We sometimes want to include the behavior of faulty nodes in our requirements and talk about *uniform agreement* for multicast. What do we mean by uniform agreement?

$\boxed{\text{A}}$ If any node, including faulty, deliver a message then all nodes, including faulty, deliver the message.

$\boxed{\textbf{B}}$ If any node, including faulty, deliver a message then all correct nodes deliver the message.

$\boxed{\text{C}}$ If a node, including faulty, sends a message then it also delivers the message.

$\boxed{\text{D}}$ A node, including faulty, delivers a message at most once and only messages that have been sent.

A:114 1p. What is the difference between *uniform agreement* and *non-uniform agreement* for multicast?

$\boxed{\text{A}}$ In non-uniform agreement correct nodes can decide different.

$\boxed{\textbf{B}}$ Uniform agreement includes the behaviour of non-correct node.

$\boxed{\text{C}}$ Non-uniform agreement does not include the reliability requirement.

$\boxed{\text{D}}$ Uniform agreement is the combination of total and causal ordering.

A:115 1p. What is the definition of total order multicast?

$\boxed{\text{A}}$ messages are delivered in FIFO order

$\boxed{\text{B}}$ messages are delivered in real time order

$\boxed{\text{C}}$ messages are delivered in *happened before* order

$\boxed{\textbf{D}}$ messages are delivered in the same sequence

A:116 1p. What would you call a multicast service that would never deliver a message m_2 before another message m_1, if m_2 was sent as a response to m_1?

$\boxed{\text{A}}$ total order

$\boxed{\text{B}}$ FIFO order

$\boxed{\textbf{C}}$ causal order

$\boxed{\text{D}}$ random order

A:117 1p. What are the requirements for using the Bully algorithm?

$\boxed{\textbf{A}}$ we must have reliable failure detectors

$\boxed{\text{B}}$ nodes are not allowed to crash

$\boxed{\text{C}}$ a total order multicast must be available

$\boxed{\text{D}}$ we need a persistent coordinator

A:118 1p. In the Bully algorithm, having n nodes, what is the time it takes from calling an election and receiving the result? Assume that we have a multicast support in the network so we can multicast in constant time and that all decisions are in constant time.

$\boxed{\text{A}}$ depends on which node that detects that the leader is dead

$\boxed{\text{B}}$ $O(n^2)$ since each node will send $n - k$ messages forward and k backwards

$\boxed{\text{C}}$ $O(n)$, since the last node sends n messages but only waits for one

$\boxed{\textbf{D}}$ $O(1)$, since neither initiating the election nor sending out the result is depending on the number of nodes (we have multicast support)

A:119 1p. What is the advantage of a ring based election algorithm compared to the bully algorithm?

$\boxed{\textbf{A}}$ nodes can easily change priority in-between elections

$\boxed{\text{B}}$ better worst case turnaround time

$\boxed{\text{C}}$ better best case turnaround time

$\boxed{\text{D}}$ more reliable if nodes fail

A:120 1p. What is the advantage of a bully algorithm compared to a ring based algorithm?

$\boxed{\text{A}}$ nodes can easily change priority in-between elections

$\boxed{B}$ better worst case number of messages

$\boxed{\textbf{C}}$ better turnaround time

$\boxed{D}$ two nodes will never be elected even if we have an unreliable failure detector

A:121 1p. What does Lamport clocks give us when implementing distributed mutual-exclusion?

$\boxed{A}$ the system is prevented from entering a dead-lock

$\boxed{B}$ requests will be granted in real time order

$\boxed{\textbf{C}}$ request are granted in *happened before* order

$\boxed{D}$ at most one process may enter the critical section at a time (safety)

A:122 1p. When does Ricart and Agrawalas mutual exclusion algorithm perform better than a central solution?

$\boxed{A}$ when there is a risk of nodes crashing

$\boxed{B}$ when the number of nodes is four or less

$\boxed{C}$ under low congestion when hardly any conflict will occur

$\boxed{\textbf{D}}$ under high congestion since only one message is needed to release and obtain the lock

A:123 1p. How are FIFO, causal and total order multicast related?

$\boxed{A}$ a total order is also a FIFO order

$\boxed{B}$ a FIFO order is also a total order

$\boxed{\textbf{C}}$ a causal order is also a FIFO order

$\boxed{D}$ a causal order is also a total order

A:124 1p. How can we guarantee that processes, that can crash, communicating over a asynchronous network can reach a non-trivial consensus?

$\boxed{\textbf{A}}$ we can not, we can only hope for the best

$\boxed{B}$ by implementing a two phase commit protocol

$\boxed{\text{C}}$ by implementing a three phase commit protocol

$\boxed{\text{D}}$ do a leader election and let the leader decide

A:125 1p. What is a dirty-read during a transaction?

$\boxed{\text{A}}$ reading a value that has been written by the same transaction

$\boxed{\text{B}}$ reading an old value that will be over written

$\boxed{\textbf{C}}$ reading a value that has not been committed

$\boxed{\text{D}}$ reading ahead of a write operation

A:126 1p. What is two-phase locking in a transaction?

$\boxed{\textbf{A}}$ not taking any locks once a lock has been released

$\boxed{\text{B}}$ taking a read lock that is strengthened to a write lock

$\boxed{\text{C}}$ reserving all locks before taking the first

$\boxed{\text{D}}$ taking locks in a strict order

A:127 1p. What does it mean that a transaction meets the atomicity property?

$\boxed{\text{A}}$ intermediate results must not be visible to other transactions

$\boxed{\textbf{B}}$ either all or no operations in the transaction are performed

$\boxed{\text{C}}$ only one datum is allowed to be updated in each transaction

$\boxed{\text{D}}$ the transaction can safely be duplicated resulting in the same server state

A:128 1p. What does it mean that a transaction meets the isolation property?

$\boxed{\textbf{A}}$ intermediate results must not be visible to other transactions

$\boxed{\text{B}}$ either all or no operations in the transaction are performed

$\boxed{\text{C}}$ effects of the transaction are isolated from server failures

$\boxed{\text{D}}$ the transaction can safely be duplicated resulting in the same server state

A:129 1p. What does it mean that a transaction meets the durability property?

$\boxed{\text{A}}$ either all or no operations in the transaction are performed

$\boxed{\text{B}}$ effects of the transaction are isolated from server failures

$\boxed{\textbf{C}}$ the effects of the transaction will remain even if we have a server crash

$\boxed{\text{D}}$ the transaction can safely be duplicated resulting in the same server state

A:130 1p. What is two-phase commit?

$\boxed{\text{A}}$ a protocol that uses a global lock that is taken by each server that wants to commit

$\boxed{\text{B}}$ a protocol where sub-transactions are allow to commit independently of each other

$\boxed{\textbf{C}}$ a protocol that ensures atomicity in a distributed transaction

$\boxed{\text{D}}$ a protocol where other transactions are allowed to see temporary value before final commit

A:131 1p. What is the problem of using optimistic concurrency control when implementing a transactional server?

$\boxed{\text{A}}$ you could end up in a dead-lock situation

$\boxed{\textbf{B}}$ hard to provide a starvation free service

$\boxed{\text{C}}$ large overhead if no conflict occurs

$\boxed{\text{D}}$ if a conflict occurs both transaction must be aborted

A:132 1p. What problem could we still have even if two transactions are serially equivalent?

$\boxed{\text{A}}$ lost updates if both transactions read the same object

$\boxed{\textbf{B}}$ dirty read if one transaction reads a value that is not yet committed

$\boxed{\text{C}}$ inconsistent retrieval if both transactions write to the same object

$\boxed{\text{D}}$ no problems since all conflicting operations are performed in the same order

A:133 1p. If the coordinator dies, what information is not enough for a set of servers to complete a two-phase-commit operation?

$\boxed{\text{A}}$ A server has received a commit message from the coordinator.

$\boxed{\textbf{B}}$ A majority of servers have sent promises to be able to commit.

$\boxed{\text{C}}$ A server has received an abort message from the coordinator.

$\boxed{\text{D}}$ A server has sent an unable to commit message to the coordinator.

A:134 1p. In a distributed transaction one often use two-phase commit, why is this better than one-phase commit?

$\boxed{\text{A}}$ We prevent dirty-read operations to cause a cascading abort scenario.

$\boxed{\text{B}}$ One-phase commit can suspend indefinitely if the the coordinator dies.

$\boxed{\text{C}}$ We prevent locks from being taken once we have released the first lock.

$\boxed{\textbf{D}}$ We ensure that if one transaction aborts then no transaction will commit.

A:135 1p. What is a phantom deadlock?

$\boxed{\text{A}}$ one that affects more than two nodes

$\boxed{\text{B}}$ one that is not cyclic

$\boxed{\textbf{C}}$ one that is detected but not stable

$\boxed{\text{D}}$ one that requires to abort more than one transaction

A:136 1p. What is a view, created by a group membership service?

$\boxed{\textbf{A}}$ the set of processes belonging to a group

$\boxed{\text{B}}$ a total order of delivered messages in a group

$\boxed{\text{C}}$ an address expansion of a multicast group

$\boxed{\text{D}}$ an access point for a client front-end

A:137 1p. In a view-synchronous group membership protocol, a process that enters the group, and is included in the delivered view, will be guaranteed to be delivered:

$\boxed{\text{A}}$ all messages in the history of the group

$\boxed{\text{B}}$ all messages starting from the active view when it issues its request to join

$\boxed{\text{C}}$ only the messages delivered before it joined

$\boxed{\textbf{D}}$ all messages starting from the view where it is included in the group

A:138 1p. Assume we have a server that is up with the probability p and use n replicated servers to increase reliability of a service. What will the probability be that the service is up, assuming that we only need one node to be up in order to provide the service?

$\boxed{\text{A}}$ $(p-1)^n$

$\boxed{\textbf{B}}$ $1-(1-p)^n$

$\boxed{\text{C}}$ $1-p^n$

$\boxed{\text{D}}$ p^n-1

A:139 1p. Why is view-synchronous communication different from reliable multicast?

$\boxed{\text{A}}$ messages are guaranteed not to be duplicated

$\boxed{\text{B}}$ we will have a lower bound on message delay

$\boxed{\text{C}}$ a message is delivered by all non-crashed processes

$\boxed{\textbf{D}}$ a message is sent and delivered only by processes belonging to the same view

A:140 1p. What is a *view* in view-synchronous communication?

$\boxed{\textbf{A}}$ A set of nodes considered to be alive.

$\boxed{\text{B}}$ The ability to determine the state of a node.

$\boxed{\text{C}}$ A central node that determines membership.

$\boxed{\text{D}}$ A combination of total and causal message delivery.

A:141 1p. In an active replicated server, how do we know that the replicas are in a consistent state?

$\boxed{\text{A}}$ they do not change state and are thus by definition consistent

$\boxed{\text{B}}$ a coordinator uses two-phase commit for each request

$\boxed{\text{C}}$ the primary replica sends update messages to all other

$\boxed{\mathbf{D}}$ they reliably receive all requests in a total order

A:142 1p. Why is it important that a primary server in a passive replicated system uses view-synchronous group communication?

$\boxed{\text{A}}$ so that requests from front-ends will be serialized

$\boxed{\mathbf{B}}$ so that all or none of the backup servers have received an update before a new primary is elected

$\boxed{\text{C}}$ to make sure backups receive updates in a total order

$\boxed{\text{D}}$ it is not necessary, its sufficient to use a reliable multicast

A:143 1p. What is sent by the primary replica manager to the backup replica manager in a passive replicated system?

$\boxed{\text{A}}$ A copy of the original request together with a unique identifier.

$\boxed{\mathbf{B}}$ A unique identifier, the state change and the response.

$\boxed{\text{C}}$ A snapshot of the state after each request has been handled.

$\boxed{\text{D}}$ A copy of the request time stamped with a vector clock that provides the information on how to order the request.

A:144 1p. Will an active replicated server with multiple front ends, that uses a total order multicaster, handle requests in causal order?

$\boxed{\text{A}}$ yes, since request are processed in total order

$\boxed{\mathbf{B}}$ not if clients can communicate with each other in between sending request and receiving reply

$\boxed{\text{C}}$ never, since total order multicast does not guarantee causal ordering

$\boxed{\text{D}}$ no, an active replicated system can never guarantee causal order

A:145 1p. A routing strategy, in a distributed hash table of n nodes, would typically have an asymptotic complexity of:

$\boxed{\text{A}}$ $O(n)$

$\boxed{\text{B}}$ $O(n^2)$

$\boxed{\textbf{C}}$ $O(log(n))$

$\boxed{\text{D}}$ $O(1)$

A:146 1p. If a node crashes with probability q and p is the risk of a node crashing during the time it takes to repair successor pointers and s is the number of successor pointers, then what is the risk of loosing a DHT ring?

$\boxed{\text{A}}$ $1 - (1 - q)^s$

$\boxed{\text{B}}$ $q(1 - (p - 1)^s)$

$\boxed{\text{C}}$ qp^s

$\boxed{\textbf{D}}$ it depends on the size of the ring

A:147 1p. What is the purpose of hashing in a DHT?

$\boxed{\textbf{A}}$ to generate uniformly distributed keys

$\boxed{\text{B}}$ to reduce the original identifiers to a smaller key space

Section B : example questions

B:1 2p. Describe why an TCP acknowledgment does not mean very much to the application layer.

Answer: An acknowledgment on the transport layer only means that a sequence of bytes have been received by the receiver. It does not mean that the application layer has read nor acted on the message.

B:2 2p. Assume you're implementing a message passing middle layer and have the option of choosing UDP or TCP as the network transport layer. What would the advantages be of choosing UDP? When would it not be practical to use?

Answer: By choosing UDP one would avoid the three-way handshake that is required to set up the communication in TCP. This would mean that messages could be sent with less overhead. If messages could be larger than one UDP packet (more than 4K byte) one would have to re-implement much of the TCP protocol on top of UDP. Moreover, if messages between two nodes was frequent and often two-way, the initial cost of setting up a TCP connection would not be a significant overhead.

B:3 2p. At local time 117 you receive a NTP reply with the following information: request sent at 82, received at 111, reply sent at 120. How should you adjust your local time?

Answer: You calculate the offset by $[(T2 - T1) + (T3 - T4)]/2$. In this example it will give you an offset of +16.

B:4 2p. When defining a request message layer, one has the option of choosing to provide *at-most-once* or *at-least-once* semantics. Describe an advantage with the *at-least-once* semantics but also describe what it means to the application layer.

Answer: In an *at-least-once* layer the implementation is free to resend messages if they are possibly lost. This will be able to hide some network problems and thus provide a more reliable service. The application layer needs to adapt to the situation where a request could have been duplicated by the messaging layer. This can be done by only using idempotent operations.

B:5 2p. Briefly describe a situation that meets the requirements of FIFO order multicast but not to total order multicast.

Answer: Two processes, A and B, multicast two messages each, a_1, a_2 and b_1 and b_2 and these are delivered in the order a_1, a_2, b_1 and b_2 by one node and as a_1, b_1, a_2 and b_2 by another node.

B:6 2p. Briefly describe a situation that meets the requirements of total order multicast but not to FIFO order multicast.

Answer: Two processes, A and B, multicast two messages each, a_1, a_2 and b_1 and b_2 and these are delivered in the order a_2, a_1, b_1 and b_2 by both nodes. Both nodes see the same order, i.e. total-order, but the messages from A were not delivered in FIFO order.

B:7 2p. Describe briefly how is NFS client-side caching is implemented and how file blocks are validated in a read operation.

Answer: NFS validation implemented using a time stamp scheme. If the time stamp was validated less then t seconds ago the cached copy is trusted. If the last validation is older, the modification time is retrieved from the server. If the file has not been modified on the server, validation succeeds. If the file has been modified a new copy is retrieved.

B:8 p. Describe the rules to maintain a Lamport clock.

Answer: Increment a counter in each operation, add it to each outgoing message. In a receive operation you update the counter to the maximum of its current value and the counter value of the message before increment the counter.

B:9 2p. In a passive replicated system we have a primary that uses a view synchronous multicaster to deliver state changes to the back-end replicas. Why do we use a view synchronous communication layer?

Answer: The view synchronous layer is needed, first of all to elect a new leader if the primary dies. It is also needed to guarantee that all backup replicas have actually been delivered the multicasted state changes before a new leader is elected.

B:10 2p. In an active replication implementation of a fault tolerant server the group service will multicast messages in total order, why is this important and how does the total order help?

Answer: The total order is important since the state of replica managers need to be consistent. The total order guarantees that the replicas see the requests in the same order and thus do the same sequence of state transitions.

B:11 2p. If you need to replicate a server that mainly handles compute intensive read request i.e. request that do not change the state of the server, what would then be the best strategy for replication, active or passive, and why?

Answer: If the request is compute intensive it would be better to use a passive replicated system. Then only the primary will do the computation and since it is a read request the message to the passive replicas need not include a state change.

In an active replication scheme that multicast all request all machines would be heavily loaded. If the front-end can determine that it is a read request it could possibly send this to only one replica thus implementing load balancing. We could then improve the throughput of the system but it requires that the front-end can determine the type of request (which might not be possible).

B:12 2p. Give one reasons why we are interested in examining a global state of a distributed system. Describe a problem that can be determine by looking at the global state but not by looking at each local node independently?

Answer: Reasons could for example be: garbage collection, deadlock detection, termination detection or debugging.

For garbage collection we could have a structure that describes a circle around the nodes, each node wants to keep it since it is referenced from an external structure.

When looking at dead-lock each node can not decide if it is in a dead-lock situation or if a message in transit will allow it to move forward, similar with termination.

When debugging we often need to determine if a global invariant holds but if we can not decide what our state is how can we determine if it holds?

In all cases the problem is that we can not simply gather the states from all nodes and determine if a predicate holds true. We might have messages in transit that carry information that will alter the situation. It is important to realize that to detect a dead-lock we do not only need to know that node A is waiting for node B and that node B is waiting form node A, we also need to know that there is no message in transit between node A and B that will break the dead-lock.

B:13 2p. Show by example how a dead-lock situation is falsely detected, a phantom lock, if one only consider information gathered from the nodes in a system one by one.

Answer: One node, A, could report that it is suspended waiting for a lock held by another node B. Before we have time to query B, the lock is released and B moves into a state where it is waiting for A. Now B will report that is is suspended waiting for A.

B:14 2p. Describe an efficient algorithm that allows you to detect a distributed dead-lock.

Answer: When suspended on a lock, send a token forward to the porcess that holds the lock. The token is forwarded by any process that is suspended waiting for a lock. If the token finds its way back to the initial process then we have a dead-lock.

Taking a snap-shot of the whole system does of course work but can not be regarded as efficient.

B:15 2p. Describe briefly quorum based mutual exclusion and its advantages.

Answer: A process is allowed to enter a critical section if it can acquire the votes of a quorum (could be a majority of the nodes). In contrast to an algorithm where one would have to obtain the acknowledgment from all nodes, one need only talk to a subset.

Quorum based solutions are fault tolerant in that they do not require all nodes to be alive for the algorithm to make progress.

B:16 2p. In a quorum based algorithm the quorum is often chosen to be the majority of processes. You could however choose other quorums, what is the most important criteria when you divide nodes into quorums?

Answer: That any two quorums overlap in at least one node so that it prevents two quorums to be formed.

B:17 2p. Describe the usage of a *leaf set* in a DHT and why it is important.

Answer: The leaf set is the closest neighbours to a node in a the DHT ring. It is maintained to ensures that the routing algorithm will terminate and find the requested node even in the event of failures.

B:18 2p. In a DHT a large numbers of nodes are connected in a logical ring structure. In order to maintain the ring even if nodes dies, one will keep track of a number of successors, not only the closest one. The question is how many successors to keep track of, the more successors we keep track of the more resilient we are to node failures, but it also means more links to check. What do we need to consider when deciding how many successors to keep track of?

Answer: Apart from estimating the risk of node failure we must consider the number of nodes in the ring. The larger the ring the higher is the risk of consecutive nodes dying; what works for a hundred nodes does not work for a thousand nodes.

B:19 2p. A distributed hash table uses a hashing of object names as the key for each object. Assuming we have names on objects that could be used as keys, for example phone numbers of subscribers, what would the problem be if the phone numbers were used instead of the hash of the numbers?

Answer: The objects, for example subscribers with phone numbers, might not be evenly distributed over the key space. This means that if we divide the key space equally between servers, we will have some servers with uneven load.

B:20 2p. When using *two-phase commit* in a distributed transaction the system could get stuck if the coordinator dies. There are however situations were the participants in the transaction safely can determine if they should commit or abort. Describe situation were the system is stuck and an situation where the participants can decide even if the coordinator is dead.

Answer: The system is stuck if all participants have promised to commit but have not received a decision from the coordinator. If the coordinator has told one of the participants to either commit or abort this is a safe decision for all to make.

B:21 2p. Describe with a message diagram the difference between a service that is *linearizable* and one that only provides *sequential consistency.* Why would we settle for an service that only gives us sequential consistency?

Answer: A linearizable service respects real-time order; if one node performs an operation in real-time before another node performs an operation, then these operations must be executed in this order.

Sequential consistency is easier to implement since we only have to respect the program order (we still have to maintain a sequential order of all operations).

B:22 2p. What would the advantage be to use a gossip protocol when building a replicated service? What would we maybe have to sacrifice?

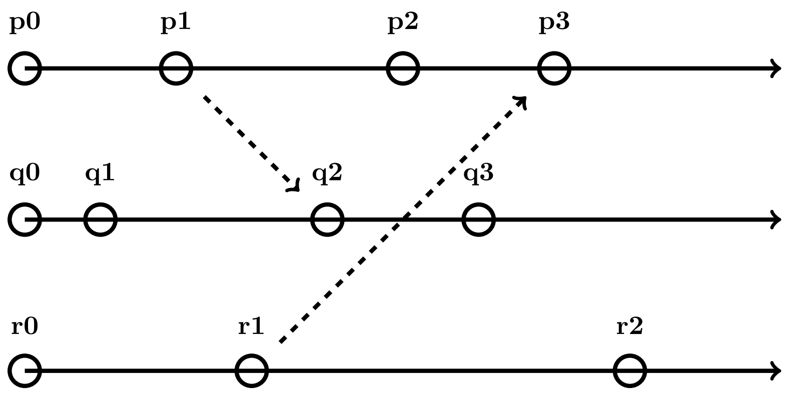
Answer:

B:23 2p. Why would you use a snap-shot algorithm, what is collected and what can you determine?

Answer: The snap-shot will collect a consistent global state. It does so by collecting local states and all messages that are in transit i.e. that have been sent but not yet arrived.

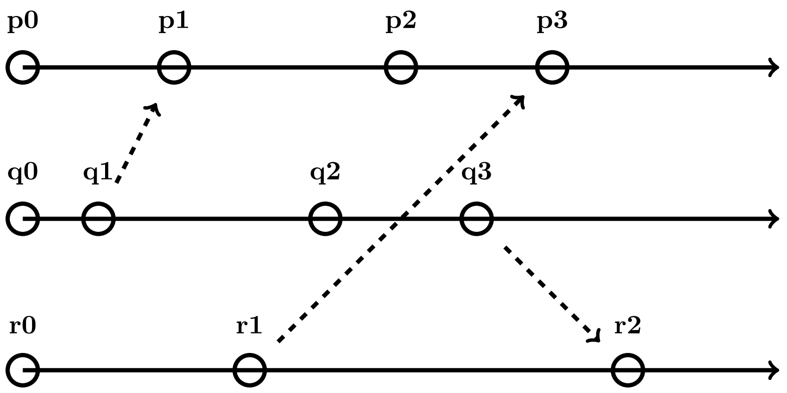
If one can prove that a stable predicate holds in the generated global state then it is also true in the global state of the real execution.

B:24 2p. Draw two lines, one that represents a *consistens cut* and one that represents an *inconsistent cut.*



Answer: The consistent cut should not include any receive event if the send event is not included. The opposite holds for the inconsistent cut.

B:25 2p. Draw two lines, one that represents a *consistens cut* and one that represents an *inconsistent cut.*



Answer: The consistent cut should not include any receive event if the send event is not included. The opposite holds for the inconsistent cut.

Section C : example questions

C:1 4p. What we are often looking for is what caused something to happen in a distributed system. To our help we have Lamport clocks, vector clocks and real-time clocks. How are these clocks related to each other and how are they related to causality? Describe the pros and cons of using these clocks, and how good they are in helping us track down the events that caused something to happen.

Answer: If we have no knowledge of the application layer the best thing we can do is to track *happened-before* order. The vector clocks would give us a perfect picture of the happened-before order and all events that could possibly have been the cause.

A Lamport clock will give us a set of events that is larger i.e. we will have events in the set that did not happened-before.

If we have perfect synchronized clocks the real-time clock will give us all events that are in the happen-before set. It will also give us events that happened real-time before but not necessarily happen-before.

If the real-time clocks are out of sync we might fail to include all events that are in the happend-before set and thus fail to find all events that are in causal realationship.

Vector clocks will take up space and are complex to manage, especially in dynamic systems.

C:2 4p. Assume that you should implement a group of processes with a replicated state. The processes should receive request from other processes and reply to these. Requests can possibly change the state and you must of course make sure that the replicated state is updated in exactly the same order in all replicas. This is simple to solve since you have been given a total order reliable multicaster to use. Show how the multicaster is used to synchronize requests to the replicas.

The hard problem is to allow new nodes to join the group; how is this done? The only API for the multicaster is `join` and `send`. The multicaster does not know about state and will happily let any process enter the group. How will a new node receive the state and how do we guarantee that the node is synchronized with the other replicas?

Answer: All request should be sent to the multicaster and only replied to when the multicaster deliver the message. One would need some tagging of requests so that only the original receiver of the request replies, all other only perform the necessary updates.

To join the system a node will first request to join. The newly joined node will then send a request for the state using the multicaster. It will ignore all messages delivered by the multicaster until it sees its own message. It will then save all messages until it sees the first reply with the state. The saved request can then be applied to the state to obtain a state that is up to date.

C:3 4p. Assume that you're given a middle-ware that implements total order reliable multicast. A process can join a group and then receive all messages from the group from that point in time. How would you implement a replicated state using the middle-ware?

The middle-ware is the only layer that you have for process communication and you are not allowed to modify it. The process that you want to replicate has an internal mutable state and should reply to read and write requests. You should be able to add new processes to the system that will then be included in the replication scheme. It is assumed that clients can send request to any process in the group.

Answer: The trick is to send a request to all other members of the group asking for the state. A newly joined process will discard received messages until it sees its own request. It will then save messages until it sees a state message. The saved messages can then be applied to the state.

Any request from a client is multicasted to the group, possibly with information on who should reply.

C:4 4p. Describe two different ways how we can implement total order multicast using only b-multicast with no link-layer multicast support. Compare the two strategies.

Answer: The two different strategies are: a centralized scheduler and a distributed consensus. In the scheduler solution one should describe how the message could be sent directly to the nodes with a unique reference and how the scheduler then assigns a sequence number to the reference. The distributed consensus implementation is based on sending out the message and then collect proposals for its sequence number. The highest proposed sequence number is then chosen and distributed to all nodes.

When comparing the two strategies one should look at how well they can handle failure of nodes. The centralized solution is of course vulnerable but does not rely on all nodes in the network. The distributed strategy has no single point of failure but multiple points of failure. We need to know if nodes join or leave the group.

The number of messages are also relevant. The centralized solution have far fewer messages, only two broadcast operations. The distributed solution needs two broadcast operations plus proposal messages from all nodes.

C:5 4p. Describe an implementation of a *total order reliable multicast.* The set of processes is static and known to all processes in the group. The network is is not reliable; messages can be lost, duplicated and delivered in any order. Processes can crash but will behave correct until they crash and will not do anything after a crash. You are allowed to use a perfect failure detector without having to describe how this is implemented.

Your implementation should consist of a set of layers where each layer provides a certain functionality.

Answer:

C:6 4p. You should implement a distributed messaging system (no central server) for a known set of client nodes. You can see it as a chat service where all chat clients communicate with each other.

What is important is that when the messages are presented to the user, they are presented in a logical order i.e. we don't want to see a comment on a message if we have not seen the original message. Assume that all nodes are correct and describe an implementation.

Now assume that we have a dynamic set of clients. Clients can join the group and leave the group. Joining and leaving the group should be done in total order with respect to all messages. How would you have to change your implementation to support this?

Answer: If we have a known set of clients its easy to use vector clocks to tag all messages. The clocks are merged when messages are received and updated only when new messages are multicasted to the group. This will allow all clients to see if they have all messages needed in order to safely display the new message.

In order to add new nodes to the system we need to have identifiers on the nodes to impose total order. A node will send out a request to add a new member. All nodes acknowledge this message and then stop multicasting more messages until the new member is confirmed. A node that is waiting for acknowledgments and receives another member request will have to sort out which request that should be treated first.

C:7 4p. A friend of yours has implemented a distributed mutual exclusion lock for a known, fixed number of processes. A process must acquire the lock before entering the critical section. The protocol is very simple; a process sends request and enters a state waiting for ok messages from all other processes. If it receives other requests while in the waiting state it will simply put them on hold until it has executed the critical section.

Your friend has some problems with dead-locks, something he detects and handles with time-outs. Describe to him what the problem is and a simple way to at least handle the dead-lock problem while preserving the distributed architecture. Also warn him that although your simple solution will prevent dead-locks from occurring it might have other problems. Describe these problems and give your friend some hints on how to handle them.

Answer: The simple extension to the protocol is to let each process have a priority. A process in the waiting state must allow a process with higher priority to enter the lock. In order to prevent to locks from entering the critical section at the same time it need also send yet another request to the process. The problem is of course that processes with low priority might suffer from starvation. To solve this one could add a Lamport clock to each process. The time stamp will decide priority and will be incremented for each received request.

C:8 4p. When implementing the Ricart and Agrawalas algorithm for mutual exclusion, it is important to not only send a Lamport time stamp with each request but also a process identifier. Why is it important to also include a process identifier and show by an example what could go wrong if we only use a time stamp.

Answer: The process identifier is used to break deadlock situations, two request can have the same time stamp and thus cause either starvation or dead lock

C:9 4p. Describe *two phase commit*: how it works and when it is used. Also describe a scenario where the protocol could stall and strategies on how one could recover from this situation.

Answer:

C:10 4p. You have been assigned to implement a logging process in a distributed system. The process should receive messages from all processes the system and write these to a file. The system should run continuously and the log file is used to monitor the execution and determine the order of events. Describe a solution and its limitations.

Answer: The problem here is of course to first of all capture the happened-before order. This could be done using Lamport clocks but the problem we then have is that we do not know when it is safe to write messages to the file; there might always be a message with a lower Lamport time stamp arriving late. This can be solved by probing the processes and thereby advance their Lamport clocks. Vector clocks would solve this problem since we could determine if a message is depending on an event that has not been reported yet. One problem we would have when using Vector clocks is that we could have to settle for a fixed number of processes.

C:11 4p. When implementing a logging service you have the alternative to use *vector clocks* or *Lamport clocks*. What are the pros and cons of choosing vector clocks? Describe two scenarios, one where you would definitely choose vector clocks and another where you would not.

Answer: The pros of vector clocks is that we do not have to wait for messages from processes that are not active. The cons is the problem with handling too many processes or handling a dynamic set of processes. One example where it is good to use vector clocks is when we have a fixed set of processes but some are not active for long time periods. Vector clocks should not be used in systems where we have a large dynamic set of processes that are constantly sending messages.

C:12 4p. You have been assigned to implement a locking service in a distributed system. A central server should receive requests from a known set of processes and grant them a lock. The problem is that it should do so guaranteeing causal order i.e. if a process, A, sends a request and then sends a message to another process, B, that then sends a request, the system should make sure that the process A, is given the lock before process B.

You are in control of the messaging system that the processes use to communicate and can adapt this to your needs.

Answer: The problem here is of course to first of all capture the happened-before order. This could be done using Lamport clocks but the problem we then have is that we do not know when it is safe to grant a lock; there might always be a message with a lower Lamport time stamp arriving late. The solution is to use Vector clocks.

C:13 4p. Assume we have a non-deterministic implementation of a server (uses a multi-threaded implementation that could deliver different replies depending on scheduling of threads) and we want to build a replicated server using either passive or active replication architecture. Would this be possible and what would the problems be with each strategy?

Answer: The main problem that must be addressed is in the active replication architecture. Active replication relies on the fact that servers are deterministic state machines and that they of course will have the same state if they receive the same requests. To use the existing implementation one would have to make it deterministic or at least make sure that the servers were left in a consistent state after each request even if the replies differed. The passive replication strategy does not rely on a deterministic server since the primary server will update the backup servers with the new state that it computed.

C:14 4p. There are many ways to implement a distributed lock service: a single sequencer, a ring based algorithm or a completely distributed as proposed by Ricart and Agrawala. The algorithm proposed by Ricart and Agrawala has its disadvantages when it comes to fault tolerance and number of messages but it does have one advantage over both ring based or sequencer based solutions. Explain this advantage and under which circumstances it could be important.

Answer: The algorithm has a synchronization delay of one message. In an environment when we have high lock contention we will be able to switch faster between a node that holds the lock and the next node in turn. This is true even if only two nodes compete for the lock.

C:15 4p. We have implemented a distributed service where nodes communicate only through multicast messages. Since it is important to preserve total order a central sequencer is used that handles all multicast messages.

Multicast messages are not very frequent but they need to be delivered quickly. To minimize delay they are sent one by one using UDP. What must be taken care of if we want the sequencer to implement a reliable multicast protocol that provides total and FIFO order. Assume that the nodes are correct i.e. they will not crash.

How can your implementation be changed if we drop the requirement on FIFO order?

Answer: An answer should reflect on the requirements and argue what needs to be handled in order to solve each one of them. The problem is of course that UDP does not give us reliability nor FIFO order by default so this has to be handled manually. Reliability must take care of lost packets but need not handle ordering. The FIFO requirement could be handled both at the sequencer and at the receiving clients (trickier) but it could speed up delivery.

C:16 4p. Assume that we have a distributed system where processes communicate by sending messages. We have an *unstable predicate* that we want to determine if it is true during an execution. How would we go about to answer this question? Outline what an implementation would look like and what limitations there would be. What could you do to limit the amount of messages and the size of messages?

Answer: We could use vector clocks and let each process send its state, tagged with a vector time-stamp, to a monitor process. The monitor would group the states into possible *global consistent states*. These states are then ordered forming a latices. If the predicate that we are interested in is true for some nodes in this lattice so that it is impossible to find a path from the original state to the final state without going through a true state then the predicate was definitely true during the execution.

In order to limit the number of messages and the size of the messages, the processes need only send in a state when the parameters of the predicate change. Moreover they need only send in the information that is relevant to determine the predicate. This will of course only work if the predicate is known aforehand.

C:17 4p. A friend of yours wants some advice on how to implement a server based application to keep track of a "tennis ladder". The tennis ladder is a competition at the local tennis club where all players are ranked in a list ordered by whom they have beaten. Any one can schedule a game by challenging a player that is at most two positions higher on the list. A player that is being challenged can not deny the opponent a game. All games are played on the same time on Saturday afternoons.

The interface should be web based so all players can schedule a game from home. Some simple authentication scheme is needed to prevent someone from schedule games for others.

Your friends first attempt was chaotic. He knew that all members were gentlemen and that they would not challenge someone who was already scheduled but, if one player challenged a second player and a third player, at more or less the same time, challenged the first player, a confirmed game could be lost.

The second attempt used a locking scheme with one lock per player. In order to schedule a game one would grab the lock of oneself and the player that one wanted to challenge, and if both were free schedule a game. This simple worked without any deadlocks but had the effect that some players logged in and took the lock of themself to prevent anyone to be able to challenge them.

First explain why the first solution does not work. Then explain why the locking works without causing a deadlock. Last, and most important, describe a better solution to the problem so that players can challenge each other and no one can prevent others from challenging them.

Answer: The chaotic behavior in the beginning is a consequence of having no concurrency control whatsoever. A game that had been scheduled by one player could be overwritten by another player. Locks can introduce deadlocks if locks can create a circular dependency. Since we here only can challenge players in front of us this is not possible.

A better solution is to implement optimistic concurrency control. A player could then not prevent other players from challenging them.

C:18 4p. You're building a high performance transaction server using time-stamp concurrency control. Describe what information you save for each object in the data-base and the rules for the read and write operations.

When will a transaction suspend and when will it have to abort? Why would it be an advantage to save multiple committed values and how would you make sure that you're not storing data that is no longer needed?

Answer: A short description that shows how tentative write operations are tied and how successful read operations are recorded. Transactions will abort if they issue a read operation or write operation that arrives "too late". It will only suspend if it tries to read an uncommitted value.

If one would allow a sequence of committed values one would allow more read operations to be performed even if they arrive late. We could keep track of the transaction with the lowest time-stamp and always truncate sequences so that earlier values are removed. This could be done by each transaction or by a separate daemon that would go through all data items.

C:19 4p. You're building a transactional database and have some options in how to implement the concurrency control. What would the pros and cons be to implement an optimistic concurrency control scheme? When would it be most useful?

Answer: Optimistic concurrency control does not use client held locks and will thus not dead-lock. One client can not prevent others from proceeding. Optimistic concurrency control can have problem with starvation. We can not guarantee that a client will be able to validate a transaction. Furthermore we need to do the validation in sequential order, something that could limit performance.

Optimistic concurrency control is most useful when we have clients that are outside our control; we can not trust them to take locks since they could cause the system to dead-lock.

C:20 4p. Assume we have a service that consists of a very large media data-base containing several millions of key indexed images (think Google Earth). We have a limited set of producers that inserts new images to the data-base but this does not happen very often. We also have millions of concurrent clients that only read information from the database. The producers and clients do not communicate with each other in any other way. How would you use distribution to increase the performance of this service. Is it important that the producers and clients do not communicate out-side of the data-base.

How would your implementation have to change if the producers also read from the data-base and their actions dependent on the information in the data-base. This could for example be only add an image if it contains less cloud than the one we have or add a daylight image if the image to the east is day light, to create a view of a rising sun?

Answer: An answer should explain why it is important that the producers and client do not communicate with each other. The discussion should lead to an suggested implementation where the data base could be distributed with out requirements on synchronization for example dividing the database into regions and using client side caching to increase performance.

The answer should discuss how reading producers could introduce inconsistency and how a transactional data-base could solve the problem. Also how client side caching could become inconsistent if new images are shown but not the images that resulted in the new image.

C:21 4p. A large scale key value store is implemented as a distributed hash table with entries replicated on three nodes for high availability. A read operation must be confirmed by R nodes and a write operation by W nodes. Depending on the requirements of the system one can choose to R and W so that the desired properties are meet. What are the pros and cons of different R and W values?

Outline a system that uses W equal to one and R equal to three. What will the problems be? Can we detect and possibly resolve problems that occur? What if R is two?

Answer:

C:22 4p. Jules Verne wrote about a world deep inside the earth with trees, lakes and animals. He forgot to mention the inhabitants of this world called Lamponians. The Lamponians lived in isolated villages and rarely traveled. The problem with living in this world deep inside the earth was that there is no sun or stars to track and thus very hard to keep track of time.

The interesting thing was how the Lamponians had organized their postal system. The mails was carried by pigeons between post offices (pigeons never got lost but they could take some time to deliver mail). Despite the problem of keeping time, the postal system could provide some sort of time.

When you handed in a letter to be delivered at the local post office you would get a slip back, stating at what time the letter was sent. You could collect letters when ever you wanted but had to confirm that you had picked them up. Each letter was stamped with a code that that could be used to determine when it was sent. As an example of how this could be used we can look at the football matches.

Every now and then (more or less a week, but no one knows since it is hard to keep track of time), the Lamponians went to one of four football matches. There was limited number of seats on each match and the Lamponians had to send in an application to go to one of the matches. They all had good friends in other villages whom they wanted to see a game with, so they did send a lot of mails before deciding what game to go to.

The Lamponians did want to watch a game with friends but easily became upset. If a Lamponian sent in an application and then told her friends that this was the best game, she would be very upset if she did not get a ticket while one of her friends, who applied only after being told that it was the best game, did get a ticket. In order not to upset anyone the Lamponian Football Association would wait until all Lamponians had sent in their applications and only then sort things out using the time codes on the letters. In some cases they didn't really know who sent their letters first and had to toss a coin to decide who should get the ticket. Everything worked, or at least no one complained. The only problem was that they had to wait for applications from all Lamponians before allocating tickets.

Explain how the postal system was constructed and how it could keep track of time. Suggest a solution so that the Lamponian Football Association could distribute tickets without waiting for all applications (you should not upset anyone).

You might wonder how Lamponians could turn up for a game in time if they had no clocks and this is a mystery, don't worry about this.

Answer:

C:23 4p. Assume that you have four uniquely named processes communicating in an asynchronous system. Assume also that they will behave correctly, that no messages are lost and that we have all the time in the world; how can you then implement a consensus protocol?

Assume that messages can get lost, how does this change the situation?

Assume that no messages are ever lost but we have a deadline to meet and that all processes have to decide what to do before the deadline, how does this change the situation?

Answer: In the first scenario all processes simply send their estimate and they all follow the one with the first name in some agreed order. Since all messages will eventually arrive and we have all the time in the world there is no problem.

If messages can get lost nothing changes but we have to implement a protocol that acknowledge messages and resend them if lost. We can thus build a layer that provides re-sending of messages and use the same algorithm as before.

In the last scenario the problem can not be solved since we can not tell for sure that all other processes know about the decision. Take the simple example of two nodes that have to act before a given time. They can send acknowledgment to each other but they can never be sure that the acknowledgment has arrived in time. Note that a we can reach an agreement but we can not know for sure that all nodes know about the decision in time.

C:24 4p. A historian wants to investigate the origin of a brilliant idea: who actually was first and how the idea evolved. As material to this investigation he has a set of articles, unfortunately without dates but of course with very good references (still no dates). Several papers describe the idea but they could of course be written independently from each other. How will he be able to trace the idea and determine the relationship? Assuming he is given two papers that describe the idea, how can he determine who was first without looking at the other papers? How does this relate to vector clocks?

Assuming that an author wrote articles one by one, and numbered them; could we show that two articles from two authors did happened in real-time order although we don't have a reference dependencies between the two articles?

Assume we want to reward papers presenting original ideas. Design a very simple scheme that will allow us to reward a paper from a set of submitted papers so that we will never give a reward to a paper if there is another submitted paper with the same idea that the first paper depends on.

Answer: The articles can be placed in a partial order based on the references. If we are given two articles that do not reference each other then their relationship can not be established just by looking at the two papers. Only by looking at all papers, or at least all papers reachable from the papers, can we determine if two articles are related.

The system is not identical to vector clocks since it only keeps a vector of its immediate dependencies. If a reference list would include the reference lists (recursively) of all its references then it would be similar to a vector clock.

Assume author Alice writes article A-1 and A-2, and that author Bob writes a paper B-1 with a reference to A-2. Then we know that A-1 was written in real-time before B-1 but there is no direct reference relationship. The real-time order is inferred by the numbering of articles by each author and the references.

A simple scheme to keep track of original ideas is to implement Lamport clocks. We do not need the power of vector clocks since we only need to guarantee that we do not award the wrong paper. Assume that all authors stamp their articles with their Lamport time. Their Lamport clock is updated when they make references to papers. If we have two articles, A and B, that both describe the same idea and A has time stamp 345 and B has time stamp 456. Then we should give the prize to A. It could be that B is referring to A or some paper that refers to A but it can not be that A is referring to B nor any paper that refers to B. So, our Lamport clocks allow us to play it safe.

C:25 4p. You're held prison by a king with a strange sense of humor. In his castle he has a set of communicating processes that use Lamport clocks to keep track of time. The processes will output a slip of paper for each event it executes, the paper is stamped with the process identifier and the Lamport time when the event was execute.

One day the king grabs a hand full of slips and comes down to the prison cell. He then says - If you can place these slips of paper in a essay row without violating the logical order in which they occurred I will set you free in the morning, if you fail he will kill you. However, if you can prove to him that it is impossible to do without risking your life he will not only set you free but also give you half of the kingdom and let you marry his son/daughter (or something else that you rather want to do).

What will you do, take a chance and place down the cards in one row or can you walk out of there looking forward to become the future king?

Answer: There is of course a way of lining the cards up in a row without violating the "happen before order". Place them in this order and walk free. If you try to prove to the king that it is impossible you're likely to stay in the prison for ever.