Distributed Systems exam answers

Tang Alexander

January 2016

Contents

1 File systems: NFS-AFS

1.1 Design NFS

The NFS server module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system. The NFS client and server modules communicate using remote procedure calls.

NFS achieves access transparancy through the virtual file system: user programs can issue file operations for local or remote files without distinction. The file identifiers used in NFS are called file handles.

NFS adopts the UNIX mountable filesystem as the unit of file grouping: one VFS structure for each remote file system mounted on the local directory.

The NFS client module cooperates with the VFS in each client machine to provide an interface for conventional application programs. It also emulates the semantics of the standard UNIX file system, effectively integrating the client.

The NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes on each request. Its file and directory operations are also integrated in a single service.

Clients make use of a mount service process to mount subtrees of remote file systems. Remote filesystems may be hard-mounted or soft-mounted in a client computer.

- **hard-mounted:** The process is suspended until the request can be completed. If the remote host is unavailable, then the NFS client module keeps trying until the request is satisfied.
- **soft-mounted:** Tries the request a small number of times and returns a failure indication if the request could not be completed. Results can become unpredictable if applications don't test for failure.

The automounter mounts a remote directory dynamically whenever an empty mount point is referenced by the client. The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each. It behaves like a local NFS server at the client machine. When the NFS client module attempts to resolve a pathname that includes one of these mount points, it passes to the local automounter a lookup() request that locates the required filesystem in its table and sends a 'probe' request to each server listed. The filesystem on the first server to respond is then mounted at the client using the normal mount service.

The resulting design provides good location and access transparancy if the NFS mount service is used properly.

1.2 Design AFS

Like NFS, AFS provides transparent access to remote shared files, but the most important design goal for AFS is to make the system scalable. The key strategy for achieving scalability is the caching of whole files in client nodes:

- **Whole-file serving:** The entire contents of directories and files are transmitted to client computers by AFS servers.
- **Whole-file caching:** Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible. When the client performs a 'close' operation, the updated contents are sent back to the server.

AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Vice is the server software that runs as a user-level UNIX process that runs in each server computer, and Venus is a user-level process that runs in each client computer.

The files available to user processes running on workstations are either local or shared. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local disks of workstations. There's a specific subtree in the directories containing all the shared files, effectively separating it from the local files. User directories are in the shared space, enabling file access from any workstation.

File system calls (such as open and close) referring to shared files are intercepted and passed to the Venus process. Venus manages the cache that holds cached copies of shared files. Each file and directory in the shared file space is identified by a unique file identifier (*fid*) which is interpretted by the Venus process. Files are grouped into volumes for ease of location and movement; volume information is stored in the fid. The Vice servers only accept requests from Venus through the use of fids.

1.3 Difference in performance

1.3.1 Caching

NFS

Read-ahead anticipates read accesses and fetches the pages following those that have most recently been read, and delayed-write optimizes writes: when a page has been altered, its new contents are written to disk only when the buffer page is required for another page. To guard against loss of data in a system crash, the altered pages are flushed to disk every 30 seconds. Clients cache the results of operations to reduce the number of requests transmitted to servers.

Bio-daemon processes are used at each client to perform read-ahead and delayed-write operations asynchronously. A bio-daemon is notified after each read request, and it requests the transfer of the following file block from the server to the client cache. In the case of writing, the bio-daemon will send a block to the server whenever a block has been filled by a client operation. Directory blocks are sent whenever a modification has occurred. Bio-daemon processes improve performance, ensuring that the client module does not block waiting for reads to return or writes to commit at the server.

AFS

Whole-file caching and the callback protocol (see 1.3.2) dramatically reduces the loads on the servers. This makes AFS much more scalable than NFS.

Because of whole-file caching, communication between Vice and Venus processes are reduced. The entire file is cached on the client machine and stays relevant until another client machine updates this file.

1.3.2 Consistency AKA cache-valication client side

NFS

In order to prevent the caches from different clients to become inconsistent, the clients are responsible for polling the server regularly to compare the cached data. A timestamp-based method is used to validate cached blocks before they are used. Since the recent updates are not always visible to clients sharing a file, the validation procedure does not guarantee the same level of consistency of files as in conventional UNIX systems.

AFS

When Vice supplies a copy of a file to a Venus process it also provides a *callback promise*, a token that guarantees it will notify the Venus process when any other client modifies the file. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a callback to each Venus process. When the Venus process receives a callback, it sets the callback promise token for the relevant file to cancelled.

Next time Venus opens a shared file, it will check this token first. If its value is cancelled, then a fresh copy of the file must be fetched from the Vice server, but if the token is valid, then the cached copy can be opened and used without reference to Vice.

Communication between client and server only happens when the file has been updated. Since the majority of files are not accessed concurrently, and read operations predominate over writes in most applications, the callback mechanism results in a dramatic reduction in the number of client-server interactions.

1.3.3 Implementation directory service

NFS

Clients make use of a mount service process to mount only the relevant (sub)directories of remote file systems.

AFS

Users' directories are in the shared space, enabling users to access their files from any workstation.

1.4 Discuss availability for both NFS and AFS when...

1.4.1 client fails

NFS

To guard against loss of cached data in a system crash, the altered pages are flushed to disk every 30 seconds.

AFS

When a workstation is restarted after a failure or a shutdown, Venus aims to retain as many as possible of the cached files on the local disk, but it cannot assume that the callback promise tokens are correct, since some callbacks may have been missed.

So before the use of each cached file, Venus generates a cache validation request containing the file modification timestamp to the server that is the custodian of the file. If the timestamp is current, the server responds with valid and the token is reinstated. If the timestamp shows that the file is out of date, then the server responds with cancelled and the token is set to cancelled.

1.4.2 server fails

NFS

To guard against loss of cached data in a system crash, the altered pages are flushed to disk every 30 seconds. If a server fails, then the client will find an empty mount point which makes it impossible to reach the file. In this case, the automounter will attempt to locate the required file system by searching in its table. A probe request is sent to each server listed and the file system of the first server that responds is then mounted at the client using the normal mount service.

AFS

The vice servers keep a callback list containing information about the Venus processes to which callback promises have been issued for each file. These

callback lists must be retained over server failures, so they are held on the server disks and updated using atomic operations.

1.4.3 communication fails

NFS

There are two ways to tackle this:

- **hard-mounting:** the process is suspended until the request can be completed. If the remote host is unavailable, then the NFS client module keeps trying until the request is satisfied.
- **soft-mounting:** tries the request a small number of times and returns a failure indication if the request could not be completed. Results can become unpredictable if applications don't test for failure.

AFS

Callbacks must be renewed before an open request if a time T has elapsed since the file was cached without communication from the server. This is to deal with possible communication failures, which can result in the loss of callback messages.

2 Replication

A common requirement when data are replicated is for replication transparency. That is, clients should not normally have to be aware that multiple *physical* copies of data exist.

2.1 Passive replication

Also called the *primary-backup model*.

Main purpose: Fault tolerance.
There is at any one time a single primary replica manager and one or more secondary replica managers – 'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary. No consultation with the backups is necessary, because they have all processed the same set of messages.

This system implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. If the primary fails, then the system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off.

Advantages:

- The front end requires little functionality to achieve fault tolerance. It just needs to be able to look up the new primary when the current primary does not respond.
- The primary-backup model may be used even where the primary replica manager behaves in a non-deterministic way, for example due to multi-threaded operation.

Disadvantages:

- Relatively large overheads. View-synchronous communication requires several rounds of communication per multicast, and if the primary fails then yet more latency is incurred while the group communication system agrees upon and delivers the new view.

In a variation of the model presented here, clients may be able to submit read requests to the backups, thus offloading work from the primary. The guarantee of linearizability is thereby lost, but the clients receive a sequentially consistent service.

2.2 Active replication

Main purpose: Fault tolerance.
The replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way.

The active replication system achieves sequential consistency. All correct replica managers process the same sequence of requests. This does not achieve linearizability. This is because the total order in which the replica managers process requests is not necessarily the same as the real-time order in which the clients made their requests.

Advantages:

- Handles failures well since the other servers simply continue working.

Disadvantages:

- Duplicate operations are performed for each replication manager, which can be expensive.
- Requires deterministic processes.

In a variation of the model presented here, an active replication system may be able to exploit knowledge of commutativity in order to avoid the expense of ordering all the requests. For example, any two read-only operations (from different clients) commute, and any two operations that do not perform reads but update distinct objects commute.

2.3 Compare active and passive replication

- Passive replication cannot handle Byzantine failures.[1] Active replication can because the front end can collect and compare the replies it receives.
- Passive replication is linearizable, active replication is not.
- Passive replication may be used even if processes hosted by servers are non-deterministic while active replication requires deterministic processes by servers.
- In case of failure (of the primary replication manager), response for passive replication is delayed. Active replication will not decrease in performance.

2.4 Where and why is group communication used?

The notion of view-synchronous group communication is a formulation of the 'virtually synchronous' communication paradigm. The view-synchronous group communication system makes guarantees about the delivery ordering of view notifications with respect to the delivery of multicast messages and extends the reliable multicast semantics to take account of changing group views.

In passive replication, the primary uses view-synchronous group communication to send the updates to the backups. If the primary fails then the group communication system must agree upon and deliver the new view.

In active replication, the group communication system delivers the user request (that was multicasted to the group of replica managers) to every correct replica manager in the same (total) order.

2.5 Describe Coda

The Coda file system is a descendent of AFS that aims to address several requirements that AFS does not meet – particularly the requirement to provide high availability despite disconnected operation (constant data availability).

The Coda architecture

Coda runs 'Venus' processes at the client computers and 'Vice' processes at file

[1] Byzantine failures are defined as arbitrary deviations of a process from its assumed behavior based on the algorithm it is supposed to be running and the inputs it receives. Such failures can occur, e.g., due to a software bug, a (transitional or permanent) hardware malfunction, or a malicious attack.

server computers. The Vice processes are what we have called replica managers. The Venus processes are a hybrid of front ends and replica managers. They play the front end's role of hiding the service implementation from local client processes, but since they manage a local cache of files they are also replica managers.

The set of servers holding replicas of a file volume is known as the *volume storage group* (VSG). At any instant, a client wishing to open a file in such a volume can access some subset of the VSG, known as the *available volume storage group* (AVSG). The membership of the AVSG varies as servers become accessible or are made inaccessible by network or server failures.

Normally, Coda file access proceeds in a similar manner to AFS, with cached copies of files being supplied to the client computers by any one of the servers in the current AVSG. As in AFS, clients are notified of changes via a *callback promise* mechanism, but this now depends on an additional mechanism for the distribution of updates to each replica. On close, copies of modified files are broadcast in parallel to all of the servers in the AVSG.

Disconnected operation is said to occur when the AVSG is empty. Effective disconnected operation relies on the presence in the client computer's cache of all of the files that are required for the user's work to proceed. To achieve this, the user must cooperate with Coda to generate a list of files that should be cached.

While it's possible to construct a file system that relies entirely on cached copies of files in client computers, the Coda servers exist to provide the necessary quality of service as the files on the servers are more reliable.

Replication strategy

Compared with AFS, Coda enhances availability both by the replication of files across servers and by the ability of clients to operate entirely out of their caches. Both methods depend upon the use of an optimistic strategy for the detection of update conflicts in the presence of network partitions. This optimistic replication strategy relies on the attachment to each version of a file of a Coda version vector (CVV). A CVV is a vector timestamp with one element for each server in the relevant VSG. Each element of the CVV is an estimate of the number of modifications performed on the version of the file that is held at the corresponding server. The purpose of the CVVs is to provide sufficient information about the update history of each file replica to enable potential conflicts to be detected and submitted for manual intervention and for stale replicas to be updated automatically.

If the CVV at one of the sites is greater than or equal to all the corresponding CVVs at the other sites, then there is no conflict. Older replicas (with strictly smaller timestamps) include all the updates in a newer replica and they can automatically be brought up-to-date with it. When this is not the case, then there is a conflict: each replica reflects at least one update that the other does

not reflect. Coda does not, in general, resolve conflicts automatically. The file is marked as 'inoperable' and the owner of the file is informed of the conflict.

Cache coherence

The Coda currency guarantees mean that the Venus process at each client must detect the following events within T seconds of their occurrence:

- enlargement of an AVSG (due to the accessibility of a previously inaccessible server);
- shrinking of an AVSG (due to a server becoming inaccessible);
- a lost callback event.

To achieve this, Venus sends a probe message to all the servers in VSGs of the files that it has in its cache every T seconds. Responses will be received only from accessible servers. Venus is sent a volume version vector (volume CVV) in response to each probe message. The volume CVV contains a summary of the CVVs for all of the files in the volume. If Venus detects any mismatch between the volume CVVs then some members of the AVSG must have some file versions that are not up-to-date. Although the outdated files may not be the ones that are in its local cache, Venus makes a pessimistic assumption and drops the callback promises on all of the files that it holds from the relevant volume. This way, no updates will be missed because of a server that is not in the AVSG of a different client that performs an update.

Disconnected operation

Coda therefore allows users to specify a prioritized list of files and directories that Venus should strive to retain in the cache. When disconnected operation ends, a process of reintegration begins. For each cached file or directory that has been modified, created or deleted during disconnected operation, Venus executes a sequence of update operations to make the AVSG replicas identical to the cached copy. When conflicts are detected, the cached copy is stored in a temporary location on the server, and the user that initiated the reintegration is informed.

Advantages:

- A cache miss is transparent to users and only imposes a performance penalty.
- The files in a replicated volume remain accessible to any client that can access at least one of the replicas.
- The performance of the system can be improved by sharing some of the load of servicing client requests on a replicated volume between all of the servers that hold replicas.

Disadvantages:

- Without replication, there's no significant difference in performance between AFS and Coda. However, with threefold replication the time required to complete the benchmark increases much faster than AFS as the number of users increases. The performance of Coda with replication does not scale well.
- Coda uses CVVs to check for conflicts, without regard to the semantics of the data stored in files. The approach can detect potential write-write conflicts but not read-write conflicts. These are 'potential' write-write conflicts.

Conclusion: Coda's overall approach of semantics-free conflict detection and manual resolution is sensible in many cases, especially in applications that require human judgement or in systems with no knowledge of the data's semantics.

2.6 Compare Coda and passive replication

Passive replication:

- All the clients must execute write operations through the primary replica manager.
- The primary communicates operations to all the other replica managers, which get updated accordingly.

Coda:

- Clients can push updates to different servers.
- The servers attempt to get the other servers up to date by comparing CVVs. In case of a conflict the client gets notified.

3 Transactions

3.1 What is a transaction? (explain ACID!)

A transaction is a set of operations on objects to be performed as an indivisible unit by the servers managing those objects. To prevent interference between clients performing a transaction on the same object at the same time, concurrency control and timestamp ordering is implemented.

properties of transactions: ACID

- **A**tomicity: a transaction must be all or nothing.
- **C**onsistency: a transaction takes the system from one consistent state to another consistent state.

- **I**solation: Each transaction must be performed without interference from other transactions; the intermediate effects of a transaction must not be visible to other transactions.
- **D**urability: After a transaction has completed successfully, all its effects are saved in permanent storage.

3.2 Describe the 2-phase commit protocol

The 2-phase commit protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort. This is needed over the *1-phase atomic commit protocol* so that servers may make a unilateral decision to abort a transaction when the client requests a commit. If one server aborts its part of the transaction, then the whole transaction must be aborted.

If the client requests *abortTransaction*, or if the transaction is aborted by one of the participants, the coordinator informs all participants immediately. It is when the client asks the coordinator to commit the transaction that the two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction.

- **Phase 1 (voting phase):**
    1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
    2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No*, the participant aborts immediately.
- **Phase 2 (completion according to outcome of vote):**
    3. The coordinator collects the votes (including its own).
        (a) If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
        (b) Otherwise, the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
    4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

3.3 Function of getDecision()

getDecision(trans) $\rightarrow$ *Yes/No*
Call from participant to coordinator to ask for the decision on a transaction when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

3.4 Where do you need timeouts and why?

- A participant has voted *Yes* and cannot proceed any further while its waiting for the coordinator to report on the outcome of the vote. Meanwhile the objects used by its transaction cannot be released for use by other transactions. The participant makes a *getDecision* request to the coordinator. If the coordinator has failed, the participant continues to wait as the coordinator is replaced.

- The participant has carried out all its client requests in the transaction but has not yet received a *canCommit?* call from the coordinator. If the timeout period on a lock expires, then the participant can decide to *abort* unilaterally.

- The coordinator may be delayed when it is waiting for votes from the participants. It may decide to abort the transaction after some period of time. It will then announce *doAbort* to the participants who have already sent their votes. Any incoming *Yes* votes will get ignored.

3.5 Explain distributed deadlock detection

Detect deadlock by finding cycles in the transaction wait-for graph. In a distributed system, a global wait-for graph can be constructed from the local ones. If there's a a cycle in the global wait-for graph that is not in any single local one, that's called a distributed deadlock.

Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval, and transactions may be aborted unnecessarily.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. However, centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. This solution suffers from poor availability, lack of fault tolerance and no ability to scale.

3.6 Edge chasing algorithm

Whenever a process A is blocked for some resource, a probe message is sent to all processes A may depend on. The probe message contains the process id of A along with the path that the message has followed through the distributed

system. If a blocked process receives the probe it will update the path information and forward the probe to all the processes it depends on. Non-blocked processes may discard the probe.

If eventually the probe returns to process A, there is a circular waiting loop of blocked processes, and a (distributed) deadlock is detected. To solve this, a transaction in the cycle is aborted to break the deadlock.

3.7 Priorities in distributed transactions

In *edge chasing*, every transaction involved in a deadlock cycle can cause deadlock detection to be initiated. Several transactions in a cycle initiating deadlock detection may lead to detection happening at several different servers in the cycle, with the result that more than one transaction in the cycle is aborted. To ensure that only one transaction in a cycle is aborted, transactions are given priorities in such a way that all transactions are totally ordered. For example, use timestamps. When a deadlock cycle is found, the transaction with the lowest priority is aborted. Even if several different servers detect the same cycle, they will all decide to abort the same transaction.

4 Indirect communication

Indirect communication is defined as communication between entities in a distributed system through an intermediary with no direct coupling between sender and receiver, in terms of space and/or time.

Group communication

In group communication the messages within a distributed system are sent to a group, and from there sent to all other members of the group. The sender has no knowledge of the identities of the receivers, hence the indirection. This kind of communication is called broadcasting where the sender forms a one-to-many relationship with the other members of the group.

Groups may be open or closed. In closed groups only members can multicast to it, opposed to open groups where processes outside the group can multicast to it as well.

Group communication API

- *join (group)*: A process joins the group.
- *leave (group)*: A process leaves the group.
- *fail*: A process crashed or became unreachable because of a communication failure.
- *send (group, message)*: Send a message to the group. The group indirection layer propagates the message to all other members.

Guaranties in multicast:

- *reliable multicast* guarantees to deliver:
  - *integrity:* at most once
  - *validity:* eventually
  - *agreement:* to all or no members
- *ordered multicast* guarantees to preserve:
  - *FIFO order:* senders's order
  - *causal order:* happened-before relation
  - *total order:* consistent order to all receivers

4.1 Publish-subscribe systems

It is a platform where subscribers can subscribe to certain events provided by publishers. The system matches published events against subscriptions. Subscribers then receive an update if successful matches are found.

Publishers form a one-to-many relationship with their subscribers, but the publishers do not know who is subscribed. Subscribers also **do not** need to know the publisher, as long as they can specify which kind of messages they would like to receive. Publish-subscribe systems are uncoupled in time as they provide asynchronous communication between senders and receivers. Publish-Subscribe also helps handle heterogeneity through event notifications as communication between components.

Publish-subscribe system API

- *publish (event)*: A publisher publishes an event.
- *subscribe (filter)*: A subscriber subscribes to a set of events according to the filter.
- *unsubscribe (filter)*: A susbscriber unsubscribes to a set of events according to the filter.
- *notify (event)*: Deliver events to its subscribers.
- *advertise (filter)*: A publisher declares the nature of the events it will produce.
- *unadvertise (filter)*: A publisher removes the advertisement.

4.2 Message queues

Message queues are a form of message-oriented middleware. A message queue introduces a layer of indirection between producers and consumers. Producers sends messages to a queue and consumers receive messages from these queues at any later point in time. *Message queues are uncoupled in time, but not in space.* The communication relation between a consumer and a producer is one-to-one.

Messages are usually added to the queue based on the FIFO policy, but priorities may be used as well. Message queues try to ensure reliable delivery by persisting messages: messages are eventually delivered (*validity*) and are also only sent at most once (*integrity*).

Three styles of receive are generally supported: receive, polling and notify operation.

Message queue API

- *send (message)*: A producer sends a message to the queue.
- *receive (message)*: A blocking receive operation. The consumer will block until message is available.
- *poll (message)*: Non-blocking receive. The consumer checks the status of the queue. A message is returned if available, otherwise a negative signal is returned.
- *notify (message)*: Issues an event notification when a message is available in the queue.

4.3 Distributed shared memory (DSM)

The objective is to share data among computers directly (not appropriate for client-server architecture!). Each computer has a local copy of the data. This data is kept up to date by passing messages between each node over the DSM middleware.

4.4 Tuple spaces

Tuple spaces are a form of distributed shared memory. Processes communicate indirectly by placing tuples in a tuple space from which other processes can read and remove them. Space uncoupling is achieved as the sending and receiving processes may come from anywhere. Tuples may be taken from the space at any time and may even reside indefinitely in the tuple space, thus achieving time uncoupling.

Tuples space API

- *read (tuple)*: Read a tuple from the tuple space.
- *take (tuple)*: Extract (= destructive read) a tuple from the tuple sapce.

- *write (tuple)*: Write a new tuple to the tuple space.

5 RMI-JEE-GAE

For all of the following sections, keep the relation of the answers with your implemented projects in mind! Questions will be asked about it.

5.1 Persistence

5.1.1 Describe the different possibilities for persistence in JEE

JEE uses JPA (Java Persistence API). Within the JPA, the EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities. There are 2 ways to obtain an EntityManager:

- **container-managed entity managers**
  An EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction API (JTA) transaction.

  @PersistenceContext
  EntityManager em;

- **application-managed entity managers**
  The persistence context is not propagated to application components, and the lifecycle of EntityManager instances is managed by the application.

  @PersistenceUnit
  EntityManagerFactory emf;

  EntityManager em = emf.createEntityManager();

5.1.2 Describe the different possibilities for persistence in GAE

GAE can use either JPA or JDO (Java Data Object):

- Saving a JDO data object to the datastore is a simple matter of calling the *makePersistent()* method of the PersistenceManager instance.
- JPA was designed for use with traditional RDBMS (relational database management system), and so has no way to explicitly represent some of the aspects of Datastore. See section 5.1.3 for the implied limitations imposed.

5.1.3 What are the differences between the two?

Java EE focuses on multi-tier enterprise applications, while GAE focuses on scalable web applications. See table 1.

Table 1: Differences JEE-GAE

JEE	GAE
focus on multi-tier enterprise applications	focus on scalable web applications
persistence through JPA	persistence through JPA or JDO (but limited)
distributed transactions	limited transaction support
SQL database (Java DB)	NoSQL storage system (App Engine Datastore)

The NoSQL storage system is schemaless (no fixed tables). It scales horizontally (by adding nodes) in contrast to RDBMS, which scales vertically (adding resources to single node). The NoSQL storage system has weak consistency guarantees (so no ACID).

The App Engine Datastore also has the Megastore, which is a combination of the scalability of the NoSQL datastore and the convenience of traditional RDBMS. The data is represented by entities and each entity group consists of 1 root entity (with one-to-many mappings down the tree). The Megastore offers strong consistency within an entity group.

The Megastore offers limited support for JPA and JDO:

- Google-specific rules for primary keys.
- types of queries are restricted (no many-to-many relations, no join queries, ...).
- no container-managed entity manager: manually create and close EntityManager instance.
- restriction within single transactions: operate on entities in same entity group.

5.1.4 Is this typical for cloud systems?

Not sure what the question refers at. It's typical for cloud systems to focus on scalability, yes.

5.2 RMI

5.2.1 Explain remote, serializable and local class

A **Local object** is used locally; it's not transferred to another machine.

A **Serializable object** only transfers data by value. If this object is transferred to another machine, any changes made to that copy will not reflect on the original object. The copies on other machines are independent.

A **Remote object** transfers a remote reference. If this object is transferred to another machine, the changes made to that copy will also modify the original object. Data is shared.

5.2.2 What happens if you make a class serializable instead of remote?

External machines will be able to retrieve objects from that class, but they won't be able to make any changes to the original object. The original object does not share data with the copied objects.

5.3 JEE

5.3.1 How do you use a transaction in JEE? (demarcation, commit, abort, granularity, rollback, ...)

First, the user needs to define if the transaction boundaries are container-managed or bean-managed.

- **container-managed**: Transaction Demarcation (determination of transaction boundaries) is not explicit. The EJB container sets the boundaries of the transactions instead of the developer, simplifying development. Methods attributed with JEE Transactions need to be annotated with @TransactionAttribute.

  The transaction ends just before the method ends. A transaction can either end with the statement fully executing or a rollback, undoing the effects of all statements in the transaction. Rollback can be triggered in two ways. If a system exception (not application exception) is thrown, the container automatically rolls back transaction. Alternatively, user can use *setRollbackOnly* to explicitly direct rollback.

- **bean-managed**: Transaction Demarcation is explicit. In other words, the developer must mark the boundaries of the transaction (start and end, as well as commit and rollback).

DO NOT mix and match bean-managed and container-managed commands! This means, do not invoke methods like commit and setAutoCommit in container-managed transactions. Likewise, do not invoke setRollBackOnly in bean-managed transactions. A method should belong to at most one transaction.

The granularity of transactions refers to how much data one transaction should handle. A very fine-grained transaction would commit after a single action for example, while a coarse-grained transaction might perform a lot of actions before committing.

5.3.2 Give 3 TransactionAttributes and explain their meaning

All 6 of them are listed, take your pick:

- **Required**:
  - If the client (=caller) is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction.

- If the client is not associated with a transaction, the container starts a new transaction before running the method.

- **RequiresNew**:
  - If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:
    1. Suspend the client's transaction
    2. Start a new transaction
    3. Delegate the call to the method
    4. Resume the client's transaction after the method completes
  - If the client is not associated with a transaction, the container starts a new transaction before running the method.

- **Mandatory**:
  - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction.
  - If the client is not associated with a transaction, the container throws the TransactionRequiredException.

- **NotSupported**:
  - If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:
    1. Suspend the client's transaction before invoking the method
    2. Resume the client's transaction after the method completes
  - If the client is not associated with a transaction, the container does not start a new transaction before running the method.

- **Supports**:
  - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction.
  - If the client is not associated with a transaction, the container does not start a new transaction before running the method.

- **Never**:
  - If the client is running within a transaction and invokes the enterprise bean's method, the container throws an exception.
  - If the client is not associated with a transaction, the container does not start a new transaction before running the method.

5.3.3 Explain briefly how a 2-phase commit could have been used instead

2-phase commit explained in section 3.2. In context of the car rental company (crc) exercises, when confirming all quotes of a client, each crc server can confirm whether they were able to confirm the quote. If any server has issues, the coordinator can ask all the other crcs to abort the transaction, failing all the quote confirmations requested by the client.

5.3.4 Explain the difference between bean-managed and container-managed transactions

See section 5.3.1.

5.3.5 When is a 2-phase commit protocol necessary?

It's necessary when a transaction spanned over multiple servers should execute everything or abort it all.

5.4 GAE

5.4.1 Does GAE belong to IaaS, PaaS or SaaS platforms? Explain why

- **SaaS (Software as a Service)**: uses the web to deliver applications that are managed by a third-party vendor and whose interface is accessed on the clients' side. Most SaaS applications can be run directly from a web browser without any downloads or installations required, although some require plugins.
- **PaaS (Platform as a Service)**: used for applications, and other development, while providing cloud components to software. What developers gain with PaaS is a framework they can build upon to develop or customize applications. PaaS makes the development, testing, and deployment of applications quick, simple, and cost-effective. With this technology, enterprise operations or a third-party provider can manage operating systems, virtualization, servers, storage, networking, and the PaaS software itself. Developers, however, manage the applications.
- **IaaS (Infrastructure as a Service)**: are self-service models for accessing, monitoring, and managing remote datacenter infrastructures, such as compute (virtualized or physical), storage, networking, and networking services (e.g. firewalls). Instead of having to purchase hardware outright, users can purchase IaaS based on consumption, similar to electricity or other utility billing.

GAE offers automatic scaling and load balancing, without delivering a real product to the end user. The target audience are user application developers, thus GAE belongs to PaaS.

5.4.2 Consider you have a design in JEE. Can you easily map it to GAE? What would you have to change? Give two examples.

Not easy, depending on the JEE design regarding the persistent objects in the database. The differences are already explained in section 5.1.3, but two important ones are mentioned again here:

- **Persistence**: GAE does not support many-to-many relation mappings. So the relations would have to transformed into a tree structure (one-to-many relations). The types of queries that can be performed on a RDBMS are very limited as well: more advanced JPQL queries from JEE won't work anymore.
- **Transactions**: GAE does not support container-managed transactions and has no knowledge about entity groups. Each transaction needs to be started, committed, rolled-back and closed manually by the user. Additionally, a single transaction can only operate on one entity group.