Politecnico di Milano

Distributed Systems

Questions
a.a. 2018-2019

Author
Tommaso Scarlatti

Reviewer
Emanuele Chioso

December 26, 2018

Contents

1 Disclaimer

This is unofficial material. These are possible answers to the questions of the exams of the Distributed Systems course in Politecnico di Milano. The main idea is to use these questions as a support while studying the slides provided by the professor. Feel yourself free to use and enhance these questions.

2 Modeling

(3) Describe the mobile code architectural style and the type of technologies that support them.

Mobile code is an architectural style based on the ability of relocating components of an application at run-time. With components here we mean code or both code and state of the execution. There are different models depending on the original and final position of resources:

- **Remote evaluation:** site A sends a request to site B and attach its code. Site B then process the data using the received code and send it back to site A.
- **Code on demand:** site A has data and requests to site B the code and once received it executes it. A concrete example is JavaScript executed in the web browser.
- **Mobile agent:** agent A migrates to site B bringing both the code and the computational resources. Once arrived, it continues executing the code.

CREST(Computational REST) joins together the concepts of REST with mobile code.

We can differentiate between two types of mobility: *strong mobility* is the ability of a system to allow migration of both the code and the execution state of an executing unit to a different computational environment, *weak mobility* instead allows only code movement across different computational environments.

This architectural model provides great flexibility to programmers since new services/-components can be easily added or modified at run-time without stopping the application. The main drawback is that securing mobile code application is really hard.

(2) Describe and compare the data-centered (Linda) model of communication with the event-based one.

This type of architecture is data-sharing based, and access to the repository (which is passive) is usually synchronized. We have several clients which communicate with the repository through RPC. Linda is a clear example of a data-centered model: data is contained in tuples which are stored in a persistent, global shared space. Workers (clients) can insert, delete or read a copy of tuples with the following primitives: **out**(t), **rd**(p), **in**(p).

In the event-based model components collaborate by exchanging information about occurrences of events using two different primitives:

- *Publish:* a notification of an event they observe
- *Subscribe:* to events they are interested to be notified

The communication of this paradigm is purely message based, asynchronous and multicast.

Describe the service oriented architecture in general and the Web Service technology as an example of its application.

Service oriented architecture is one of the possible architectural sytle for distributed systems. It is built around the central concept of *service*, which represents loosely coupled units of funciontality. In this architecture there are three main components:

- **Providers:** export services
- **Consumers:** search for a certain service, invokes the Broker that eventually replies with the information regarding service.
- **Brokers:** keep a list of active providers and the description of the available services related to them, in order to be reached from interested consumers.

The process of invoking a set of services is known as *orchestration*.

A well known example of the service oriented architecture are *Web Services*, which is a software specifically designed to guarantee interoperable machine-to-machine interaction over a network. Different software systems often need to exchange data with each other, and a webservice is a method of communication that allows two software systems to exchange this data over the internet. Different software may use different programming languages, and hence there is a need for a method of data exchange that doesnt depend upon a particular programming language: XML. Interfaces are described using WSDL and web service operations are invoked using the SOAP protocol.

3 Communication

Describe Remote Procedure Call in general and the various architectural issue that characterize this model of communication (parameter passing, client/server binding, synchrony...).

Remote Procedure Call (RPC) is the remotely execution of a procedure on a separate machine. When the linker assembles code, in case it is an RPC it substitutes the call with a client stub version. Differently from a local call data is then packed up in a message and there is a system call to a send primitive to send the message to the server. Then the client stub call the procedure receive and starts waiting for the response. When the message reaches the server, the server stub transforms the code of the RPC in a local procedure call which is executed and sent back.

There are many issue with this communication paradigm. Firstly, **parameter passing** become hardly due to possibly structured data (which must be ultimately flattend) and different data representation across machines. For both of this problems middleware provides automated support with serialization and marshaling leveraging on a platform independent language to represent the procedure's signature: IDL. IDL raises the level of abstraction of the service definition, separating the interface from its implementation.

Binding the client to the server is another issue. Find which server host a service and where the server is located and how to establish a connection with it. Sun's RPC uses a *portmap* deamon that binds calls to server ports. The client contact directly portmap that gives back the service identifier and the available port. (This approach solves only the second problem, i.e. the client must know where the server resides. However it can use multicast query to multiple deamons). DCE's solution instead leverages on a directory server which can be contacted by the client. In this way the client must be only aware of where the directory server is located. DCE can also be distributed to ensure scalability.

Communication paradigm can be either **synchronous** (the caller is suspended until the callee is done) or **asynchronous** (with many variations depending on if the caller needs a result back). For instance, the client might be blocked waiting for acceptance from the server instead of result.

RPC can be even be **batched**. It means that if a sequence of RPCs do not require a result back, they are buffered on the client and then sent together to the server just once.

Describe and compare RPC and RMI (as general communication abstractions). Clarify the differences regarding the way parameters are handled. In particular, explain why passing by reference is problematic for RPC, how RMI addresses these difficulties, and why, viceversa, passing by value is a problem for RMI.

Remote Procedure Call (RPC) is the remotely execution of a procedure on a separate machine (see previous answer). **RMI** (Remote Method Invocation) is similar to RPC

since it shares with it the core concepts and mechanisms, but it also tries to take advantage of the OOP in a distributed setting. Sometimes we can even find RMI built on top of an RPC layer.

The main difference is that object reference in RMI can be passed around nodes in the distributed system. IDL in RPC separates interface from implementation of procedures, in RMI, following the basic OO principle, we can separate interface and implementation on two different machines. Basically, we can have the object interface on one host and the implementation on another.

In practice, there are two implementations of RMI which are *Java RMI* and *OMG Corba*. The former is single-language/single-platform, the latter is multi-language/multi-plaform. Let's see how they handle parameter passing:

- In *Java RMI*, the difference between passing by reference or by copy depends on how the parameters of the method that we are invoking are labeled: if labeled as "serializable", then the objects are serialized and passed **by copy**. Thus, is retrieved on the client as a different object and every modification has no effect on the original instance. If is labeled as "remote", the object is passed **by reference**. What happened is that if the client invokes one of the server methods and passes an object as remote object, what the server gets is a reference that goes back to the client machine and when the server invokes methods on this object, it invokes code of this method the client machine.

- *OMG Corba* is multilanguage and multiplatform and therefore we cannot make the assumption that the object that we see by copy runs on the same platform of the sender. Thus, passing **by copy is not admitted**. The last version allows to pass by copy but only in the case both sender and receiver are written in Java e run in a Java Virtual Machine.

(2) Describe publish/subscribe in details discussing the various alternatives for the subscription language, then describe and compare the various approaches to implement a distributed (acyclic) dispatcher.

Publish/Subscribe is a form of message oriented communication. It offers a multicast and transient communication (the opposite of the message queuing paradigm, which is point-to-point and persistent) and a high degree of decoupling among components. Application components can interact with two simple primitives: publish and subscribe.

A central component in this architecture is the **event dispatcher** which is responsible for collecting all the subscriptions and for routing events to all the matching subscribers. The event dispatcher can be either centralized or distributed for scaling reasons. In the second case, usually dispatchers (message brokers) are organized in an overlay network which can be cyclic or acyclic.

The expressiveness of the **subscription language** plays an important role in the publish/subscribe architecture. There are two different types:

- *Subject-based:* (topic-based) when you subscribe you simple specify the name of the topic. And when you publish your message on a specific topic. It's a kind of group organization.
- *Content-based:* subscriptions contain expressions (event filters) that allow clients to filter events based on their content. The set of filters is determined by client subscriptions.

In a network of **dispatcher** without cycles, each client attaches one of the brokers and the various brokers are attached to each other forming an **acyclic** graph. There are three different approaches in this case:

- *Message Forwarding*: every broker stores only submissions from directly attached clients. Messages are then forwared from brokers and delivered to clients only if they are subscribed. (Optimized for many subscriptions since they don't flood the network)
- *Subscription Forwarding*: subscriptions are forwareded to all the brokers, but are never sent twice over the same link. Messages follow routes defined by subscriptions.
- *Hierarchical Forwarding*: messages and subscriptions are forwared to the root of the tree and flows downwards only if a matching subscription have been received along that route.

Describe the stream-oriented communication model and the various approaches to address the QoS issue.

In some contexts, data is organized in streams, which are sequences of data unit (e.g. text, audio, video...). There are cases in which time impacts only the performance and not the correctness of the application, for instance when we are interested only in the processing of data. In real-time applications, instead, time determines also correctness.

With QoS (Quality of Service) we usually express non functional requirements such as: required bit rate, maximum delay, maximum jitter... How can we gurantee QoS? IP is a best-effort protocol, so it doesnt offer any mechanism to control the quality of the connection. Actually it offers a differentiated services field into the header called Type of Service (TOS) which is usually discareded by routers. Therefore we need to enforce QoS at the **application layer**. At the application layer you cannot guarantee a certain level of performance but you can do something to increase the chances of reaching that level of performance. Let's have a look at three main techniques:

- *Buffering:* it is a way to control max jitter by sacrificing session setup time. Packets are not instantly delivered to the receiver, first they are stored in a buffer and then they can arrive at the destination.
- *Forward error correction:* if a packet is corrupted, instead of asking the packet again, we can employ the forwarding error correction mechanism that allows to detect and correct errors adding information to a packet such that even if the packet gets corrupted or lost we can still rebuild it.
- *Interleaving:* instead of sending packets that include consecutive frames you send packets that mix together different frames and you are also buffering them.

4 Naming

(4) Describe the problem of removing unreferenced entities in a distributed system and possible solutions to such problem.

Naming provide a global referencing service. The problem here is: how do you discover that there are entities that are no more reachable? In conventional system the common approach is to use a garbage collector. In a distributed setting we have a lack of global knowledge about whos using what, and to network failures. Several techniques have been developed to deal with this problem:

- *Reference Counting:* the object keeps track (with a counter) of how many other objects have been given references. Passing references between proceses may lead to a race condition. We can use an acknowledgment message, but it doesn't scale well. A possible solution to avoid race condition is to use an additional counter and to communicate only counter decrements. The object keeps a global and a partial counter. Every time a process references the object, the partial counter is halved. When partial = global the object is removed. This approach is called *Weighted Reference Counting.*

- *Reference Listing:* the idea here is different. Instead of keeping track of the number of references, keep track of the identities of who is referencing. We have the advantage that insertion and deletion in the list are idempotent operations and it's easier to maintain the list consistent with respect to network failures. It still suffers from race conditions when copying references.

- *Distributed Mark & Sweep:* it is useful to detect entities disconnected from the root set. Initially all nodes are marked white and then they are colored in sequence if they are reachable from the root. At the end of the process the white marked nodes are collected and deleted.

(2) Describe structured naming in general and DNS in particular.

In a structured naming system names are organized in a so called *name space.* Name space is a labeled graph with two possible types of nodes: directory nodes or leaf nodes. A leaf node stores the name of an entity. Resources are referred through *path names* which can be either absolute or relative. For scalability reasons, the name space is usually distributed over multiple name servers and it is partitioned into different layers: global, adiministrational, managerial.

DNS (Domain Name System) is a clear example of a structured name system. It translates more readily memorized domain names to the numerical IP addresses needed for locating and identifying computer services. Name space is hierarchically organized as a rooted tree and each subtree represents a "domain" and belongs to a separate authority and each name server is responsible for a particular zone. The resolution process can be:

- *Iterative:* the client contacts the root server at first which solves its portion of the name. In this approach each DNS server doesnt resolve the entire name but gives back only partial results. The client must ask several time to different name servers to solve the entire name.

- *Recursive:* the client contacts the root and the root name server returns back the entire name, by performing directly the resolution: the root name server solves the entire name.

In practice, a mixture of the two approaches is used. Caching and replication are largely used too, using IP anycast in the global name servers to forward requests to every replica.

Describe and compare the various approaches you know to implement name resolution in flat naming.

In flat naming schema names are just "flat", meaning that they are simple strings with neither structure nor content. There are several approaches to perform name resolution in this context:

- *Simple approaches:* these approaches are implemented leveraging on broadcast channels (like ARP resolution) or multicast (to reduce the number of nodes contacted). In the case of mobile nodes, we can use forward pointers: nodes leave references of their next location in their previous location.

- *Home-based approaches:* here we have one stable home node that knows the location of the mobile node (can be replicated for stability). There are some drawbacks: home node must be always available and is fixed, leading to poor geographical scalability.

- *DHT (Distributed Hash Table):* nodes are organized in a structured overlay network which can have different topologies. The lookup is performed through the table which contains key-value pairs and it is distributed across the different nodes. Chord is an implementation of the DHT approach which relies on the fact that nodes and keys are organized in a logical ring.

- *Hierarchical approaches:* in this approach nodes are organized hierarchically in a tree structure. The root has entries for every entity and an entry point to the next subdomain. Leaves contain the address of an entity (in that domain). Lookup usually starts where the client resides and is propagated up until an entry is found. With this technique we can exploit locality, but there is a drawback: root must contain information of all entities.

5 Consistency and Replication

(2) Describe all the types of synchronization variables belonging to the different consistency models, with pros and cons for each type.

One of the problem is that the pointing time in which the synchronization takes place is not defined by the developer, he doesnt inform the system about the places in which consistency is required. A possibility is to give some predicates to the developer or to the compiler to specify when the synchronization must be enforced. So the following consistency models rely on synchronization variables:

- *Weak consistency:* it uses only one synchronization variable S and access to synchronization variable is sequentially consistent: everybody agrees on the order in which the synchronization variable is invoked. The problem with Weak consistency is that we have a single predicate to represent two things: sending my writes to everybody and synchronizing, hence obtaining the latest value. This leads to unnecessary overhead.

- *Release consistency:* it introduces different synchronization operations: *Acquire* to indicate that a critical region is about to be entered and *Release* to indicate that a critical region has just been exited. Again we have primitives and synchronization takes place only when the programmer invokes these primitives. There are two flavours of this approach: *Eager*, which means that on release all updates are pushed to all the replicas, and *Lazy*, which implies that on release nothing is sent but during acquire phase process must get the latest version.

- *Entry consistency:* explicitly associates each shared data item with a synchronization variable. It improves parallelism, enabling simultaneous access to multiple critical sections and it reduces updates overhead. On the other hand, access complexity is increased. There are two ways to access a synchronization variable: exclusive or non-exclusive mode.

Describe the client-centric consistency model.

In data-centric consistency model we made the assumption that a process interacts always with the same replica of the data. If we relax this assumption, there are consistency model that deal with this, considering consistency by the prospective of users. These are the so called Client-centric consistency models. There are some properties that are desirable:

- *Monotonic reads*: if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

- *Monotonic writes:* a write operation by a process on a data item x is completed before any successive write operation on x by the same process.

- *Read your writes:* the effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.

- *Writes follow reads:* a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read.

In client-centric consistency models, each operations gets a unique identifier (e.g. replica ID + sequence number) and two different sets are defined for each client: a **read set** which contains write identifiers relevant for the read operation performed by the client, and a **write set** which keeps the identifiers of the write performed by the client.

Describe what an epidemic algorithm is, how it works and what properties it entails. How does this compare to gossiping and other propagation strategies?

Epidemic algorithm is propagation strategy for updates in a client-centric consistency protocol. It is heavily based on theory of infectious disease spreading: updates are spread by pairwise communication among nodes and the aim is to have all the nodes infected. This algorithm is instrinsically distributed and redundant, since it is fault-tolerant, scalable and resilient to topological changes of the network. It cannot provide deterministic guarantees, but it enables relevant overhead savings.

We can distinguish between different types of nodes: *infective* are nodes that are willing to spread their updates, *susceptible* are nodes that can be infected and *removed* are update nodes not willing to re-propagate.

We have seen different propagation strategies, for instance **Anti-entropy**, in which a server chooses at random another server to propagate its updates. Communication can be push, pull or both and notifications semantic can be either positive (change seen) or negative (change missed). **Gossiping** is another strategy which works in this way: an update propagation triggers another towards a different server. If the update has already been received, the server reduce the probability to gossip further. This approach can be complemented with anti-entropy.

(2) Describe the two different approaches for primary based consistency protocols, underlying also the advantages and disadvantages of having a backup system.

In primary based consistency protocols, all writes go through a single primary server, in charge of keeping accesses consistent. We can have different variants of this protocol: local writes vs remote write and with or without a backup server.

- *Remote-Write:* it is a straightforward implementation of sequential consistency. A remote server is in charge of managing access to data. Each read/write operation is forwarded to it. The main issue is blocking client while updates propagate: we can use an asynchronous implementation but with no guarantee of sequential consistency in the presence of failures.

- *Local-Write:* the difference is that the data item requested (both read and write) is migrated to the server requesting the update, which becomes the new primary. This approach can provide better performance in case of locality and multiple accesses.

The idea is that if you want to access some data, it is highly probable that the client is the one who is working with data. Therefore I move the primary as close as possible to the client.

Describe the approaches of "active replication" and "quorum-based replication", underlying the best and worst configurations for both.

Active replication and quorum-based replication are both *Replicated-Write protocols*, characterized by the fact that write operations can be carried out at multiple replicas.

Active replication: operations are sent to each replica and we must preserve ordering through a timestamp or a centralized coordinator. In order to avoid problems with multiple invocations we can use a coordinator which is in charge to forward messages from all the replicas just once.

Quorum-based replication: is another kind of protocol in which an update occurs only if a quorum of the servers agrees on the version number to be assigned. The client contacts several replicas and set the number of replicas that it have contacted to perform read (NR) and the number of replicas that it have contacted to perform a write (NW). Typically these two properties must be respected to avoid read-write or write-write conflicts:

- $NR + NW > N$ (avoid read-write conflicts)
- $NW > N/2$ (avoid write-write conflicts)

Where N is the number of total nodes in the distributed system.

Briefly describe the Hadoop implementation of a distributed file system, and which strategies have been adopted to avoid corruption of files and allow clients to recover their data in case of a rack failure.

HDFS is a distributed file system that provides scalable and reliable data storage, and it was designed to span large clusters of servers. HDFS has a **master-slave** architecture with one single namenode and several datanodes. The namenode is in charge of managing the filesystem namespace (metadata): it keeps a namespace image, an edit log and a list of all the datanodes and which blocks they store. The master doesn't keep any file by itself. Each file uploaded to HDFS is splitted in blocks of 64 MB each for minimizing seek and maximizing transfer rate.

Read and write operations are performed in this way: for reading, firstly client reads a list of blocks from the namenode, then the namenode guides the block search throughout the HDFS network by proximity. For file writing, namenode decide where to allocate the blocks. The first datanode receives the block, saves it locally and forwards it to the next replica. Datanodes are chosen in a smart way to balance reliablity and bandwidth. A customized strategy can be configured, but by default replicas are two nodes on the same

rack (to improve write performance) and one node on a different rack (to deal with rack failures). In case of a node failure, the failed node is removed from the list of available nodes in the name node and the under-replicated blocks are sent to new destinations.

6 Synchronization

(2) Describe the Christian's algorithm for synchronization. Under which assumptions it works at best? Why?

A common approach for synchronization protocol is to let the clients contact periodically a time server which can provide date and time with precision. The assumption is that messages are assumed to travel fast w.r.t. required time accuracy.

The possible problems are: time might run backwards on the client machine and therefore changes must be introduced gradually. Message transfer time makes the timestamp provided by the server obsolete. Assuming that the RTT (Round Trip Time) is split equally between request and response and it is averaged over several measurements:

- $T0$: client sent request
- $T1$: client received response
- I: interrupt handling time on the server
- C_{UTC}: time sent from server to client
- $RTT = T1 - T0 - I$ round trip time

we can compute the correct time at the client $T1 = C_{UTC} + RTT/2$.

Describe the different protocols you know to synchronize clocks in distributed systems.

In order to ensure synchronization in a distributed system we need distributed protocols that synchronize physical clocks. We have seen three main protocols:

- *Christian's Algorithm*: see answer before.
- *Berkeley Algorithm*: in several algorithms time server is passive, it simply responds to clients answers. In this algorithm the server plays an active role: periodically asks the connected clients their time, collects all the responses, averages it and then sends back for each client a personalized adjustment w.r.t. the mean.
- *Network Time Protocol*: NTP is used in practice over the Internet on top of UDP. It works creating an hierarchy of servers: the top most machines hold the correct time (connected to a UTC source) while the machines at the lowest level are users machines. Is a variation of the Christians algorithm which tries not only to ensure synchronization but also tries to calculate the error. Two servers at two different layers communicate:they periodically send to each other messages, characterized by two main variables:
  - o_i: estimate of the offset of clock B relative to clock A
  - d_i: measure of the accuracy of this estimate

  Basically, we need to find o, the true clock difference between the two clocks of A and B and we use the estimator o_i to approximate it.

Describe scalar clocks in general and in particular describe the Lamport's protocol to guarantee totally ordered multicast messaging (clarify the assumptions)

In many applications it is sufficient to agree on a time, even if it is not accurate w.r.t. the absolute time. What really matters is often the ordering of events rather than the times-tamp itself. Furthermore, in distributed systems, if two process do not interact with each other is not necessary to deal with synchronization (they are concurrent).

L. Lamport invented the **happens-before** relationship. It aims at capturing a potential causal ordering among events. It is potential because two events can be related by the h-b relationship even if there is no real causal connection among them or, since information can flow in other ways then message-passing, two messages can be causally related with no h-b. If two events e and e' occur in the same process and e occurs before e' then $e \rightarrow e'$. This property is transitive.

The algorithm works this way: each process keeps a scalar clock (using integers to represent the clock value) which is incremented every time the process sends a message. The message sent is timestamped with the current scalar clock of the sender. If a message is received instead, the scalar clock of the receiver is set in this way $L = max(ts, L)$.

Totally ordered multicast delivers messages in the same global order. The assumptions are that links are reliable and FIFO. Messages are sent and acknowledge using multicast and each message keeps a timestamp of the sender's scalar clock. Receivers (including the sender) store all messages in a queue, ordered according to its timestamp and eventually all the processes will have the same messages in their queues. A message is delivered to the application only when it is at the highest in the queue and all its acks have been received. Since each process has the same copy of the queue this ensure that messages are delivered in the same order globally.

(3) Describe vector clocks in general, compare them with scalar clocks and describe how the former can be used to guarantee causal delivery in a multicast communication system (clarify the assumptions you make).

A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system. Distributed systems may have no physically synchronous global clock, so a logical clock allows global ordering on events from different processes in such systems. In the **vector clocks** solution each process p_i keeps a vector V_i of N values (one for each process included itself). $V[i]$ is the number of events that have occurred at P_i.

The vectors are update following these rules: initially the vector is initialized at 0 in every position. Then if an event happens locally, the counter of the process itself is incremented. When a process i receives a message from a process j sets $V[j] = max(V[j], t)$ and then increment $V[i]$. Compared to scalar clocks they use a vector of values instead of a single counter, and in this way determine an isomorphism between the set of partially

ordered events and their timestamps. In vector clocks by looking only at the timestamps we are able to determine whether two events are causally related or concurrent, which was not possible with scalar clocks.

A slight variation of vector clocks can be used to implement **causal delivery** of messages in a totally distributed way in a multicast communication system:

- Messages and replies sent using reliable, FIFO ordered, channels
- Need *only* to preserve the ordering between messages and replies
- Vector incremented only when send a message, on receive just merge not increment
- Hold a reply until the previous messages are received

(4) Describe mutual exclusion. Define it in precise terms and show the fully distributed protocol to achieve it using scalar clocks (clarify the assumptions for the protocol to work correctly).

Mutual exclusion is a way to prevent two process to interfere when accesses some shared resources. In a distributed setting we do not have shared memory, but we have shared resources, such a printer. The assumptions for the protocol to work correctly are processes and channels reliable. The requirements are:

- *Safety property*: at most one process in the critical zone
- *Liveness property*: no deadlock no starvation
- *Optional:* if one request happened before another, the entry is granted in that order

A naive solutions consists in using a centralized server for coordinating access, but it will be a single point of failure and a bottleneck for performances.

Scalar clocks can be used to ensure mutual exclusion in this way:

- *Requesting:* to request access to a resource a process P_i multicasts a request message m with timestamp T_m attached. When a process P_j receives m:
  - If does not hold the resource and it is not interested it acknowledges P_i
  - If it holds the resource it puts the request into a local queue ordered according to T_m
  - If does not hold the resource but it is interested in holding it and it has already sent a request, it compares T_m with the timestamp of its request and if T_m is the lowest it acknoledges P_i, otherwise it inserts the request into the local queue.

  A resource is granted to P_i when its request has been acknowledged by all the other processes.
- *Releasing:* on releasing a resource the process acknowledges all the requests queued while using that resource.

Describe leader election. The goal, possible usage scenarios, how to implement it, under which hypothesis the presented protocol works.

Many distributed algorithms require a process to act as a coordinator. We have several processes and we want to elect a leader such that every process agree. The main assumption is that *nodes are distinguishable* and the system is a *closed system*: processes known each other and their IDs but they don't know who is up and who has failed. There are several algorithms which differ on the selection process:

- *Bully election:* in this algorithm we have two additional assumptions:
  - Reliable links
  - It is possible to decide who has crashed (synchronous system)

  The algorithm works in this way: when a process P notices that the coordinator is no longer responding initiates an election process, sending an ELECT message to all the processes with a greater ID. If no one answers P wins the election and send a COORD message to all the processes. If P' responds, initiates a new election. The process with the highest ID always win the election.

- *Ring based:* the assumptions are that nodes are organized in a (physical or logical) ring topology. When a process detects a failure of the coordinator, send an ELECT message with its ID to the nearest neighbour. On receiving, if a process is not in the list of processes in the message adds its ID and propagate to the next neighbour. Otherwise change the message to COORD and propagate. On receiving a COORD message the process with the highest ID in the message is assumed to be the new coordinator.

Provide a formal definition of a "cut" and a "consistent cut". Make an example of a consistent cut and a non consistent one.

The global state of a distributed system consists of the local state of each process together with the messages in transit over the links. Since capturing the global state is impossible (no global clock) we need a way to approximate it. A **cut** is an approximation of a snapshot of a system. A cut is said to be consistent if for any event e it includes, it also includes all the events that happened before e. A consistent cut represent a situation in which the system could have been.

Formally, a cut of a system S composed of N processes can be defined as the union of the histories of all these processes up to a certain event (a history is a collection of events):

$$C =< h_1^{k_1} \cup ... \cup h_n^{k_n} >$$
$$h_i^{k_i} =< e_i^0, e_i^1, ..., e_i^{ki} >$$

A cut is said to be consistent if:

$$\forall e, f : e \in C \wedge f \rightarrow e \Rightarrow f \in C$$

To make an example is sufficient to show that a non consistent cut is one in which there is an arrow that goes across the cut.

What is problem of termination detection in a distributed system and how can we solve it?

We want to know when a computation has completed or deadlocked (i.e. no more useful work can be done). All processes should be idle and there should be no message in transit in the system. How can we detect termination?

A simple solution, proposed by Tanenbaum and then proved to be wrong, uses distributed snapshot and the notion of predecessors and successors across processes that send and receive tokens.

If we are in the case of **diffusing computation**, i.e. where all process initially are idle except for the init process, we can use the *Dijkstra-Scholten* algorithm for termination detection. The algorithm works creating a tree out of the active processes. When a node finishes processing and it is a leaf, it can be pruned from the tree. When only the root remains alive and it has completed processing we can state that the system has terminated. An idle node is activated when a message is sent to it and it is add to the tree as a child of the sender. The advantage of this algorithm is that it has a low overhead since it does not interact with never activated processes.

7 Peer to Peer

Describe the evolution of distributed systems, from client-server interactions to hybrid implementations (partly c/s, partly p2p), to full p2p implementations.

A distributed system in its most simplest definition is a group of computers working together as to appear as a single computer to the end-user. The most commonly used paradigm in todays Internet is still C/S architecture. It is based on an active component (client) which requests data from a passive component (server) which replies back with the information needed. It is a successful paradigm (Web, FTP, DNS), however, it has several drawbacks: it is hard to scale and present single point of failure.

Peer to Peer (2P2) is a completely different paradigm. There are no differences between components in the network (they are all "peers"), resources and services are shared through direct exchange between peers. In this way the paradigm can take advantage of resources at the end of the network, overcoming the problem of leaving some resources unused of C/S architecture. 2P2 paradigm quickly grown in popularity thanks to the number of end-host resources which is increased dramatically and the availability of broadband connectivity. In 2P2 nodes form a logical overlay network over a physical one and they are dynamic: they can join and leave the network at any time.

Napster was the first 2P2 file sharing application. It was a killer app: 50M downloads. It can be seen as an hybrid implementation of C/S and 2P2: end-host nodes are active contributors of the network since they share files with other peers, on the other hand there is a central server which is in charge of keeping the state of all the nodes and performing all the search processing. In KaZaA some peers are elected as "supernodes" to perform searching in a more efficient way. Gnutella and Freenet are example of fully decentralized pure 2P2 networks.

Describe the range of solutions in p2p systems regarding the problem of searching for items in the peers network: are there solutions offering any guarantees of search time?

Searching and retrieving resources is a fundamental issue in peer-to-peer systems due to their inherent geographical distribution. Basically the problem is how to direct queries towards nodes that can answer in the most efficient way.

Napster, the first 2P2 file sharing application born in the 2000, used a central server that contains the state of all the nodes (O(N)) but allowed to have a search scope and **search time of O(1)**. *Gnutella* instead, which is fully decentralized, implements the most simple algorithm for searching: flooding. Messages are sent from nodes to its neighbours and so on, bounded by a maximum number of hops (HTL Hops to live). The search scope in this case is O(N) while **search time is O(2D)** where D is the average HTL. In *KaZaA* searching is performed through an optimized version of flooding: some nodes of the network which have a good connection are "supernodes" and they retain information about the files of the other nodes. Supernodes direct searching query towards

other supernodes. KaZaA, which is a propretary protocol, offers no guarantees on the sarch scope or time.

BitTorrent is the most used file sharing network since 2007. Its searching mechanism is **out-of-band**: you need to use external services to retrieve a tracker for the file that you are willing to retrieve. A tracker is a server that keeps track of where file copies reside on peer machines, which ones are available at time of the client request, and helps coordinate efficient transmission and reassembly of the copied file.

Describe the evolution of P2P algorithms, especially in the four basic operations: join, publish, search and fetch.

In the first 2P2 file sharing application, *Napster*, joining, searching and publishing relued completely on a centralized server which was in charge to keep the state of all the available nodes. Only fetching was performed directly between peers.

Gnutella and *Freenet*, which are pure 2P2 architectures, solve the problem of joining by selecting a subset of nodes (neighbours) to be contacted on startup (for Gnutella PING/PONG messages are used). Searching is performed with a simple flooding algorithm in the case of Gnutella while Freenet uses an hill-climbing method with backtracking leveraging also on caching properties of the nodes to optimize the queries. Publishing in Gnutella is not even needed. Freenet tries to put "close" file together in order to create a small world network model. Fetching in both cases is performed sending files all the way back to the client.

KaZaA relies on supernodes (subset of nodes of the network with high bandwidth) to perform joining, publishing and searching, which is done using flooding but only between supernodes. *BitTorrent*, on the other hand, uses a tracker server to perform joining and publishing. Searching is out-of-band in this protocol. Fetching is performed directly between peers: can fetch simultaneously from multiple peers.

Describe the BitTorrent protocol in detail, in particular the "choking" mechanism: what problem is tackled with this approach? Is this problem treated also by other protocols?

BitTorrent is the most used file sharing network since 2007. It allows many people to download the same file without slowing down everyone elses download. It is focused on providing fast fetching rather than searching.

In BitTorrent we can distinguish two different types of peers: *leeches*, peers that have only portions of the file and *seeders* which instead have the entire file. Together leeches and seeders form a swarm. Files are broken down into smaller fragments (usually 256KB in size) to start uploading as soon as possible. Chunks of file are not downloaded in sequential order and therefore they need to be assembled by the receiving machine.

When a peer joins the network it has to contact a tracker server which contains a list of available peers, then it can start downloading from these peers (both seeders and leechers). Searching instead is performed out-of-band, meaning that the client must rely on

third-party sevices (e.g. the web) to retrieve a .torrent file, a meta-data file describing the file(s) to be shared. It holds:

- Name and size of the file(s)
- Address of the tracker
- Address of the peers
- Checksum of all blocks

Chocking is a special mechanism of the BitTorrent protocol which tries to solve the problem of free riders: peers that only download without contributing to the network. Choking is a temporal refusal to upload: every 10s a choking evaluation is performed and each peer un-chokes a fixed number of peers. The evaluation is based on the download rate between peers. The protocol implements also an "optimistic" un-choking mechanism that is uploading to a peer regardless of the current download rate. This allows to discover currently unused connections.

Several other protocols tried to deal with the problem of free riders in different ways. *Napster* uses an user-based approach: "I will not upload to you if you don't share anything worth". *eMule* uses a credit system: the more you upload the more credits you get. Priority in the queues is based on credtis. *KaZaA* has a Partecipation Level system which is flawed, as it relies on the client accurately reporting their Participation Level and therefore it is easy to cheat.

Compare the network model of Freenet with DHT approaches, in particular with Chord.

Freenet is a P2P application designed to ensure true freedom of communication over the Internet. It was originally developed by Ian Clarke in 1999. It allows anybody to publish and read information with complete anonymity. Freenet is built with the aim of no centralized control or network administration. The protocol works in this way: on startup, client contact a few other nodes it knows about it and gets its own unique id. Searching is performed through a query for file id using a steepest-ascent hill-climbing search with backtracking. If the query finds a file, it returns it all the way back to the sender. Insertion of new data (publishing) can be handled similarly to searching: file is sent toward the node which stores other files whose id is closest to file id. In this way inserted data is routed in the same way as a request would.

Each node in the Freenet model keeps a routing table associating the key with the data source and during searching data is cached along the way. In Chord, each node stores items and keeps a finger table which is able to perform searching in $O(log(n))$ thanks to the way items are stored in nodes. This approach doesn't exploit caching. Freenet network tend to be a "small world" and "closed" files id tend to be stored on the same node. Information migrate towards area of demand and files are prioritized according to popularity.

In Freenet intelligent routing makes queries relatively short and keeps search scoping small but still we don't have no provable guarantees. Chord, on the other hand, guarantees to find the item in at most $log(N)$ steps, where N is the number of nodes.

8 Fault tolerance

Describe and discuss the kind of failures (the failure model) that may happen in a distributed system.

If we consider a distributed system as a collection of processes that communicate with one another and with their clients, not adequately providing services means that processes, communication channels, or possibly both, are not doing what they are supposed to do. We can distinguish between three types of failures:

- *Omission failures:*
  - Processes: fail-safe (sometimes wrong but easily detectable output), fail-stop (detectable crash), fail-silent (undetectable crash).
  - Channels: send omission, channel omission, receive omission.
- *Timing failures:* (only for synchronous systems)
  - Both: occur when one of the time limits defined for the system is violated.
- *Byzantine failures:* (arbitrary failures)
  - Processes: may omit processing steps or add more.
  - Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once.

A system is said to be *fault tolerant* if it can provide its services even in the presence of faults.

Describe the failure model for a distributed system in general (processes and channels). Which is the most common type of failure for channels? Why this type of failures which are very hard to manage for processes are not so relevant/problematic for channels?

See previous answer for a detailed explanation of what are the failure model that may happen in a distributed system.

Byzantine failures are really hard to manage for processes. If the system is asynchronous, Fisher et al. (1985) demonstrated that agreement is impossible between a group of processes even if just one is faulty. The main problem is that we cannot distinguish a slow process from a dead one. For channels instead byzantine failures are not so problematic. Messages can be resent if they are missing or if they are sent twice is the process that ensure that the second one is discarded.

Describe the problem of agreement in a group of processes under the various conditions (type of admitted failures, async vs sync systems...).

A number of tasks may require that the members of a group agree on some decision before continuing, but since we are dealing with faults, we want all the non faulty processes to reach an agreement. The decision is taken by the processes themselves rather than by an external observer (e.g. voting mechanism). This is also known as the *consensus problem*. A set of properties must hold:

- *Agreement*: no two processes decide on different values
- *Validity*: if all processes start with the same value v, then v is the only possible decision value
- *Termination:* all non faulty processes eventually decide

In the most simple case we consider a **synchronous system** in which all processes evolve in synchronous round, with reliable channels and processes that may have **omission failures**: it is possible to reach an agreement in at least $f + 1$ rounds with f being a bound on the number of failures using the *Floodset* algorithm. The algorithm works in this way:

- A default value v_0 is specified, then each process keeps a variable W which is a subset of V initialized with its start value.
- For $f + 1$ rounds each process sends W to all other processes and adds the received set to its W.
- After $f+1$ rounds, if $|W| = 1$ decide on the only element of W, otherwise if $|W| > 1$ decide on v_0.

If we introduce also **byzantine failures**, the problem can be described in terms of armies and generals (firstly formulated by L. Lamport): communication is perfect, but some generals are traitors. Lamport (1982) showed that if there are m traitors, *2m+1* loyal generals are needed for an agreement to be reached, for a total of *3m+1* generals. The algorithm works in this way: each node send a value to all the other process. The processes collect all the values in a vector, send this vector to the others and then they compute the vector using majority for each vector position.

Let's now consider **asynchronous systems**: Fischer et a1.(1985) proved that in a distributed system in which messages cannot be guaranteed to be delivered within a known, finite time, no agreement is possible if even one process is faulty. The problem with such systems is that arbitrarily slow processes are indistinguishable from crashed ones (i.e., you cannot tell the dead from the living).

(2) Describe the various alternatives for reliable communication when process are reliable but links are not.

We want to achieve a reliable group communication. Our assumptions are that groups are fixed, and processes non-faulty. Therefore, all group members should receive the

multicast (not necessarily in the same order). There are several alternatives to ensure a reliable communication:

- *Positive acknowledgments:* the sender sends its message to all the members and they reply with an ACK message if they receive it. This approach leads to an ACK implosion.
- *Negative acknowledgments:* if a process doesn't receive a message, it sends a NACK in multicast. The other processes on receiving the NACK suppress their feedback. In this way the sender receives just one NACK but everyone must process NACKs.
- *Hierachical Feedback Control:* in this approach receivers are organized in groups headed by a coordinator and groups are organized in a tree routed at the sender. The main drawback is that the hierachy tree has to be constructed and mantained.

Discuss virtual Synchrony.

Virtual synchrony is an interprocess message passing technology which solves the *atomic multicast problem*. The problem is the following: we want a message be delivered either to all the members of a group of processes or to none, and that the order of messages be the same at all receivers (e.g. an update in a database with replicas). Ideally we would like *close synchrony*:

- Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order
- A multicast to a process group to be delivered to its full membership. The send and delivery events should be considered to occur as a single, instantaneous event

Unfortunately, close synchrony cannot be achieved in the presence of failures.

Virtual synchrony is a form of reliable multicast which follows these (weaker) requirements:

- Crashed processes are purged from the group and have to re-join
- Messages from non-faulty processes are delivered to all the other processes
- Messages from a failing process are processed either by all correct members or by none
- Relevant messages are delivered in a consistent order with respect to other multicasts and with respect to each other

The *group view* is the set of processes to which a multicast should be delivered as seen by the sender at sending time. A view changes every time a process join or leave the group. We can see view changes as another from of multicast message. Therefore multicasts take place in epochs separated by group membership changes.

Retaining the virtual synchrony property, we can identify different orderings for the multicast messages:

- Unordered multicast

- FIFO multicast
- Causally-ordered multicast

Describe briefly when we build the rollback-dependency graph and the checkpoint dependency graph, how we build them and the general goal these algorithms solves.

When processes resume working after a failure, they have to be taken back to a correct state. Since we are in a distributed setting, we do not have a global state of the system from which we can recover the state of each process. Therefore, each process periodically and independently records its own state (the rectangles in the diagram). Our goal is to produce a recovery line, a set of checkpoints which together form a consistent cut.

Both rollback and checkpoint dependency graph are built starting from a graph that shows the processes dependencies between intervals. When a process sends a message to another, it includes also information about its current checkpoint interval, therefore it creates a dependency between these two intervals. Both graphs change the interval dependencies into checkpoint dependencies: rollback creates a dependency (an arrow) between the ending checkpoint of the interval of the sender and the ending checkpoint of the interval of the receiver. Checkpoint graph instead is built creating a dependency from the starting checkpoint of the sender and the ending checkpoint of the receiver.

The two algorithms work in this way:

- *Rollback dependency graph:* the recovery line is computed by marking the nodes corresponding to the failure states and then marking all those which are reachable from one of them. Each process then rolls back to the last unmarked checkpoint.
- *Checkpoint dependency graph:* take the latest checkpoints that do not have dependencies among them.

9 Big Data

Consider the architecture of a system for processing large volumes of data on a cluster of machines. Define and compare the following design/architectural choices: batched vs streaming, pipelined vs scheduled. In particular, discuss their benefits or limitations in terms of latency, throughput and elasticity.

Data stream processing is a computer programming paradigm based on two concepts: streams of data that flow through and processing operators. In this context, a platform for Big Data can have two key design/architectural choices:

- Batched vs Straming (data) - Throughput / latency
- Pipelined vs Scheduled (operators) - Load balancing / fault tolerance

In *batch* computations the stream is divided into batches of data and each operator process a batch at a time. It might introduce some delay since we have to wait until a batch is fulfilled with data. On the other hand, processing elements in batch is typically more efficient since data is transferred in larger blocks. Throughput is increased in this way. In *streaming* computations instead each element gets processed as soon as it is available. We have lower delay since results are ready after the computation time of the operator but we might have lower throughput. A typical tradeoff is usign *microbatches*. This technique is implemented in Apache Spark: uses batches of small dimension to have a high throughput with acceptable delay.

In case of streaming computation, at system start up, the operators are instantiated and allocated to physical nodes. Then the data flows from node to node where are processed. Operators form a processed *pipeline* which can be used to achive both data and task parallelism. In the case of batch computations, operators can be scheduled (instantiated on a node) on demand, when a batch is available for processing. Load balancing is easier in the case of scheduled execution, it can change number of instances at runtime. In streaming computation we can adopt techniques that are similar to scheduling algorithms in operating systems. What happens in the two cases if a node fails? In case of scheduled operators, the task is simply rescheduled. In pipelined processing, periodically checkpoint to (distributed) file system and in the case of failure, reply from the last checkpoint.

What is the difference between HPC and cluster computing?

What is Map-Reduce?

MapReduce is a programming model for cluster computing introduced by Google in early 2000s. It enables application programs to be written in terms of high-level operations on data, splitting the computation in two different phases: Map and Reduce. The former processes individual elements and for each of them produces in output one or more $< key, value >$ pairs. The latter processes all the values with the same key and outputs a value. The developers need only to specify the behaviour of these two functions.

The platform is in charge to control:

- *Scheduling:* allocates resources for mappers and reducers (master-slave architecture)
- *Data distribution:* moves data from mapper to reducer (tries to exploit data locality)
- *Fault tolerance:* transparently handles the crash of one or more nodes (map and reduce must be re-executed)

Typical MapReduce applications are found in tasks in which a sequence of these steps is required, series of data transformation and iterating processes. It is a fixed paradigm with an high overhead but is simple to use from the developer point of view and is good for large-scale data analysis.

10 Security

Describe what is access control, how it works and the two main ways of implementing it in a distributed system.

Access control means managing access rights. In a non distributed system is quite easy: each user has rights to use resources. In distributed systems is not trivial: we should create an account for each user on each machine or use a centralized approach. A better approach is using a reference monitor which mediates requests from subjects to access objects.

Conceptually, we have an access control matrix which is a sparse matrix. We can implement it in two different ways:

- *ACL: Access Control List.* Each object has its own. A client create an access request r as subject s and the server is in charge to check. One drawback of ACL is that it occupies great memory. We can use groups to build a hierarchy in ACL.
- *Capabilities Lists*: each user has its own capabilities list. Client create access request r for object o passing its capability C.

Describe the logical key hierarchy approach, and for which problem it is used.

In the context of secure group communication, we need a way to preserve the integrity of the system even when a process joins or leaves the group. Basically, two properties must be preserved:

- Backward secrecy: process cannot decrypt message before joining
- Forward secrecy: process cannot decrypt message after leaving

Logical key hierarchy is an approach for efficient key distribution: leaves are members with keys, in the root we have the DEK (Data Encryption Key) and in the other nodes KEK (Key Encryption Keys). Each member knows each key up to the root. In case of leaving we change all the keys of the leaving member and we can diffuse efficiently the new keys, encrypting each key with the children.

Describe the use of symmetric and asymmetric encryption algorithms in the secure protocols we studied. What are the "session keys"? Why do we use them?

UNCLEAR: does he want a list of all algorithms: RSA, DH, Digital Signature... and their applications?

In order to provide secure communication in a distributed system we need to build a secure channel. In a secure channel authentication and message integrity should go together. A message is no longer useful if it has been altered or if I don't know its source. Authentication needs shared information between the authenticator and the party: this

information is a shared *authorization key* which can be either symmetric or asymmetric. Authentication can be ensured using a challenge-response schema between parties.

Once authentication is set up, usually a *session key* (symmetric) is exchanged to provide integrity and possibly confidence of following messages. Session keys are useful to limit the wearing of the main key (used for authentication), meaning that an active attacker can solicit use of the key. After session is closed the session key must be destroyed.

Describe the problem of Trust when designing and using the layers that build a distributed application (from the network up to the application itself). What is a Trust Computing Base? Where the security mechanism should be put?

When considering the protection of a (distributed) application, there are essentially three approaches that can be followed:

- Focus on the data
- Focus on the possible operations invoked on the data
- Focus on the users that have access to the data

In order to enforce a security policy, you need security mechanisms to be put somewhere in the protocols stack. If you don't trust security at a low level you can build your mechanisms at a high level. On the other hand, if you put them on a high level they might depend on lower level mechanisms. Therefore you need to trust on them. This set is called Trusted Computing Base (TCB).

What is a certification authority? Why there are also different approaches to the same problem? (i.e. the "Web of Trust"?)

A digital signature ensures that plaintext was encrypted with a certain key but it says nothing about who is using that private key. Public-key certificates solve this problem: a tuple (identity, public key, CA sign) is created. The CA digitally signs files called digital certificates, which bind an identity to a public key.

The public key of the CA is assumed to be well-known. The basic idea is that pervasive information is hard to alter. Usually CA are organized in a hierarchical way: CA uses its private key to sign the certificate, its public key must be trusted by another CA. Therefore we need a Top level CA which is a trusted element. In order to distribute this trusted element we need a central authority that releases it. Another approach is PGP Pretty Good Privacy (web of trust) where users can authenticate other users by signing their public key with their own.

Why Kerberos is able to use a simplified Needham-Schroeder implementation?

The Needham-Schroeder protocol is one of the two key transport protocols intended for use over an insecure network. These are:

- *Symmetric Key Protocol*: based on a symmetric encryption algorithm. It forms the basis for the **Kerberos** protocol. This protocol aims to establish a session key between two parties on a network, typically to protect further communication.
- *Public-Key Protocol*: based on public-key cryptography. This protocol is intended to provide mutual authentication between two parties communicating on a network, but in its proposed form is insecure.

Needham-Schroeder protocol uses nonces, which are arbitrary numbers that can be used only once in communication.

Describe the Diffie-Helman protocol. Which problem does it try to solve? Why is it only a partial solution? Which other solutions have been proposed to solve the problem?

For authentication protocols we need a shared secret keys between each party or with the KDC. Diffie-Helman is an algorithm which guarantees to exchange symmetric keys on an insecure channels. It is a way to transform a secret key exchange in a public key exchange. It is useful since it relaxes the assumptions of confidentiality and integrity to only integrity. It is only resistant to passive attacks, while is vulnerable to active opponents (e.g. man in the middle attack).

The algorithm works as follow: g and n are publicly known numbers (with some math properties). Alice picks x and sends $g^x mod n$ to Bob. Bob picks y and sends back $g^y mod n$. Then they both compute $g^{xy} mod n$ and in this way they have a shared secret.