Distributed System Recap

Elia Ravella

September 27, 2021

Contents

I Modeling **2**

1 The software architecture of a distributed system **2**
1.1 Network OS based 2
1.2 Middleware based 2

2 The run time architecture **2**
2.1 Client Server 2
2.2 Service Oriented Architecture 3
2.2.1 Web Services 3
2.3 Peer to Peer 3
2.4 Object Oriented 4
2.5 Data Centered 4
2.6 Event Based 4
2.7 Mobile Code 4

3 The Interaction Model **5**

4 Failure Model **5**
4.1 Failure Detection 6

II Communication **6**

5 Remote Procedure Call **6**
5.1 RPC's Interaction 7
5.2 Batching and Queuing 7

6 Remote Method Invocation **7**

7 Message Oriented Communication **7**
7.1 Publish Subscribe 8

8 Stream Oriented Communication **8**

III Naming **9**

9 Flat Naming **9**
9.1 Flat naming approach in general 9
9.2 Name Resolving Techniques 9

10 Structured Naming **10**
10.1 Structured naming systems's name resolution 10

11 Attribute Based Naming **10**

12 Removing unreferenced entities **10**
12.1 Reference counting 11
12.2 Weighted reference counting 11
12.3 Reference listing 11
12.4 Distributed mark and sweep 11

IV Synchronization and time 11

13 Absolute time solutions **11**
13.1 GPS' approach 11
13.2 Cristians' algorithm 12
13.3 Unix time solution 12
13.4 NTP 12

14 Logical time **12**
14.1 Scalar clocks 12
14.2 Vector clocks 13

15 Leader election **13**
15.1 Bully algorithm 13
15.2 Ring based approach 13

16 Collecting global state **14**
16.1 Dsitributed snapshot 14
16.2 Dijkstra Scholten termination detection 14

17 Distributed transactions **14**

18 Distributed deadlocks **15**
18.1 Distributed deadlock detection: Chandy Misra Haas 15
18.2 Distributed prevention 15

V Replication consistency 15

19 Replication models **15**
19.1 Active vs Passive 15
19.2 High availabity and consistency models 16
19.2.1 Data centric models 16
19.2.2 Client centric models 16

20 Update propagation **17**
20.1 Propagation strategies 17
20.2 Updates forwarding strategies 17

VI Fault tolerance 17

21 Agreement in process groups **17**
21.1 FloodSet algorithm 17
21.2 Byzantine failures handling 17
21.3 Impossibility of distributed agreement 18

22 Reliable group communication - faulty channels **18**
22.1 Scalable reliable multicast 18
22.2 Hierarchical feedback control 18

23 Reliable group communication - faulty processes and faulty channels **18**
23.1 Close and virtual synchrony 18
23.2 Virtually synchronized systems - message ordering 18

24 Distributed commit **19**
24.1 Two phase commit and three phase commit 19

VII Big data **19**

25 General information **19**

26 Batched approach versus stream approach **20**
26.1 Batch processing 20
26.2 Stream processing 20
26.3 Comparison 20

VIII P2P **20**

27 Design of a p2p system **20**

28 Approaches to search and lookup **21**

IX Security **21**

29 Design issues **21**

30 Certification **21**
30.1 Certification authority 21

31 Secure channel **21**

Part I
Modeling

1 The software architecture of a distributed system

With "software architecture" we describe the abstract view of a software system that helps us locate it in a certain technology stack, and how it interacts (or makes other components interact) with other systems.

1.1 Network OS based

A system can rely directly on the operative system of the machine it's working on to access the network . This makes the application platform dependent, because different machines on a network could have different operative systems.

1.2 Middleware based

A middleware is a piece of software that masks the network interaction, providing advanced primitives to work with, in order to uniform the access to the network between application. It can also provide coordination, advanced communication and administration of the network. More on the services of the middleware in the next sections.

2 The run time architecture

For run time architecture is intended the collection of software components, such as connectors, classes, data types, synchronization algorithms, that compose the software when it's up and running. All the run time architectures we'll see derives directly from the most common architectural styles used in software engineering.

2.1 Client Server

Super classic architecture for a distributed system. Peers in the network have different roles (clients, who need computations, and servers, who can perform them) and communicate usually through messages or remote proceure invocation.
Servers expose a well-known interface where users can connect through clients; these interfaces are usually passive. The client server architectural style introduces the notion of tiers: being a layered architecture, functionalities may be split into a "stack" where each layer performs one of the three main application function: data management, business logic handling, presentation. This kind of interactions has lead the way for the

2.2 Service Oriented Architecture

Taking at the extreme the concept of tier, we can imagine an app as a collection of functionalities, and split these functionalities onto different machines. Then, the client accesses only the functions it needs, contacting the right server. This architecture adds a role to the classic service consumer - service provider schema: the service broker, that takes care of orchestrating the provider's API, describe them, act as a register and index of the services and as a fixed point for clients to reach functionalities.

2.2.1 Web Services

"a software system designed to support interoperable machine-to-machine interaction over a network". Simply the web adaptation of the SOA. Services are described with the Web Service Description Language and are indexed through the Universal Descritpion Discovery Integration system. The SOAP protocol implements web service interaction, that is built on top of HTTP (adding a layer on top of application layer).

REST REpresentational State Transfer is simply a set of principles, that describes how a distributed system should be built and how it should behave. These principles are:

1. Interactions are client server
2. Interaction are stateless; the only state that can be transferred is encoded in the parameters
3. Data must be explicitly marked as non / cacheable
4. Opaque layers: each component cannot see beyond the layer he's interacting with
5. Clients must support code on demand
6. Interfaces must be defined all in the same way: this makes sure that
   (a) Resources are univocally reachable through a URI
   (b) Resources are manipulated through their representation
   (c) Messages are self descriptive: each message contains enough information to describe how it should be handled
   (d) Hypermedia as the engine of the application state

2.3 Peer to Peer

P2P (is not a crime) is a model for distributed systems that does not assign explicit roles to peers connected to the network; instead tries to build (through coordination and orchestration) a system that "stand by itself" taking advantage of the edge resources of the network (i.e. the ones in the clients). P2P is a much more scalable architecture than client server. The main advantage of a peer to peer architecture is the direct exchange of resources (that can be computational power, network bandwith or plain old storage) between peers of the network.

2.4 Object Oriented

In a OO system, every component is a software object with a rigid and well defined interface. Peers pass around references of the objects to contact them, and object are invoked through messages or RMI. OO distributed systems inherit the OO programming paradigm principles, such as data obfuscation, uniform APIs and self contained components. It's a "peer to peer" model in abstract, but it's usually used to implement client server interactions.

2.5 Data Centered

A data centered model enforces a kind of "single repository" architectural style. All data available to the application is stored in a central storage facility that encapsulates all the primitives (that depend on the richness of the interface, but usually provide the classic CRUD operations) to manage the data inside it. How the central repository manages these operations depends on the implementations: usually it's a passive RPC interface that requires synchronization.

Implementation: Linda Linda is a data sharing model studied for parallel computation. It manages data encapsulating it in tuples (vectors of attributes) that all belong to a *tuple space* memorized in the central repository. The communication is persistent, content based and generative.
Like all central repository systems, these kind of platforms do not scale well.

2.6 Event Based

The event based architecture, as the client server, assigns two distinct roles to peers: publisher and subscriber. The first produces content and adds it to the system, and the second consumes the content. The contents are routed towards the client *that have explicitly expressed intereest for a certain topic*. So, a third actor lives in the system: a routing middleware that carries out all the message exchanging. The clients can subscribe to attributes of the contents as well as titles, or even perform a content based (templated) request for content: it's the middleware task to select the right contents at publishing time and routes to the clients that requested it.
This style enforces a comunication that is inherently asynchronous, message based, multicast and implicit. Publish subsribe network also enforces anonimous communication, also.

2.7 Mobile Code

With "mobile code" is intended the capability of a system to transfer the code or the state of execution between peers. We should define mobility first: *strong mobility* is the capability to move both the code and the execution state to another machine; *weak mobility* is the capability to move only the code from a machine to another.
Four models of mobile code can be defined, differentiating basiing on how code and state are handled or located.

- Code on demand: the client asks the server not for the result of a computation, but for the computation itself, then executes it locally and consumes the result
- Remote evaluation: the client produces a piece of code to be executed on the server. The state usually is kept on the server, where the code is executed
- Mobile agent: both the state of the application and the code are transferred and executed on another machine; this can be implemented through Object Oriented distributed systems, encapsulating the state in the objects' attributes
- Client server: the client asks a server to compute an operation and then consumes the result

Implementation: CREST Computational REST is a first implementation of the mobile code paradigm: it proposes a client server interaction where data is manipulated not through *representations* of it, but instead *computations* of it, like closures or continuations. CREST axioms are the simply computational-adapted axioms of REST.

3 The Interaction Model

Two interaction styles between processes are possible:

- synchronous: the first process sends a message to the second one and stops his execution until a response is got
- asynchronous: calling for remote resources (or message sending) does not block the execution. also, time managing may differ among peers: the interaction simply does not require synchronization of any kind

Obviously, the synchronous model is much stricter than the other one because of all the boundaries it poses to the exchange of messages and information. Every solution designed for async systems, in fact, works for sync system, but the vice versa is not true.

4 Failure Model

The failure model describes the various types of error that can occur in a network. Error can be

- channel failures
- process failures

We distinguish three kind of errors:

- Omissions: data simply is not there. A process stops his execution, or a channel fails in sending the data

- Byzantine failures: a process stops acting deterministically and starts sending "wrong" data. A channel corrupts data
- Timing failures (only in synch systems): a process takes too much time computing; a channel takes too much time delivering/processing a message

4.1 Failure Detection

Failure detection is the procedure via which a process network gets aware of an error. In synchronous systems, the use of keepalive messages overcomes the most tricky part of determining who's alivo and who's not; in async models instead, shit starts hitting the fan. It has been proven that distributed agreement (so a set of processes elects a value as a correct one) in an async system *that can fail* is not virtually possible.

Part II
Communication

5 Remote Procedure Call

RPC is the mechanism adopted by various distributed systems to provide processes with the capability of calling a procedure that resides on another machine. It's inter process communication over a network. As the name suggests, RPC refers to purely procedural architectures, where there's little to no notion of application state.
One of the most complex aspects of RPC is how parameters are passed around: three main models of parameters passing are possible:

- pass by value (the value is copied and sent as is through the network)
- pass by reference (a reference to the variable is sent; this approach is very difficult to implement in a distributed context, where references, that are usually implemented by pointers, lose their meaning when sent across different processes)
- pass by copy / restore (a slight variation of pass by reference: in practice, the middleware send the value to be computed, copying it out from the reference passed as argument. At the end of the computation, the final value is written in the address)

Another difficult aspect to approach, reagrding the parameter passing, is how to deal with complex structures, that must be streamlined to bytes (serialized) is order to be passed. Also, memory representation may vary between processes or machines: the RPC middleware should take care of this aspect too, embedding the serialization and marshaling procedures in the middleware directly (enabling the programmer just to write the effective app code) and providing an Interface Definition Language to aid the description of the API.
One more ostic aspect: client server synchronization. Hardcoding the server address in the client's app code is the worst possible solution; instead, solution

like the portmap (binding a service to a port informing the portmap daemon, that can then route incoming requests or tell which port is to be called in order to obtain a certain function) or a "service broker" approach (an external actor is devoted to centralize the exchange of requests of services and the address where the clients can find the requested service)

5.1 RPC's Interaction

The base model of RPC is built as a synchronous system. However, semi-sinchronous and asynchronous implementations are possible. Most of them exploit a data type called "promise" (also called "future") that hosts the value that is remotely evaluated, but can be filled "after" the call is done, to avoid the waiting time of the network. This enforces lazy evaluation techniques.

5.2 Batching and Queuing

The call-response nature of the RPC systems is suitable for some kind of scheduling of the calls. Two models are possible: batched RPC and queued RPC. Queued RPC is pretty standard: requests are managed as they arrive to the stub. Instead, the idea behind batched RPC is to "batch together" the requests until a real-time one is issued, and then sending the whole batch to the server; this enhance performances reducing network calls.

6 Remote Method Invocation

RMI is: RPC in a object oriented world. The biggest difference/improvement is the ability to pass references around: the RMI middleware sends a skeleton of the object to the server, that uses it to access all the data structures of the sent object. The changes are then reflected in the main machine. A more complicated part of RMI wrt RPC is the logic built within the data structure? How do I serialize a function, to send an object with all his methods?
An advantage of the OO world is that there's almost no need of an IDL because the interface / implementation decoupling is built within OO languages and systems.

7 Message Oriented Communication

Instead of maintaining an active connection for a whole transaction, another approach to the exchange of information (where there is less need of a synchronous interaction, maybe) can be to encapsulate data into messages, and send them to the receiver machine that must be running a dedicated service that can recognize the message, interpret it and pass it to the above layers of the application. Message oriented systems enforces a low coupling/synchronism between sender and receiver of the messages: in some systems, not even an ACK of messages is needed. Other message oriented systems can also be persistent (opposite of *transient*): not even both process are required to be running together to interact via messages.
Two main approaches are possible when it comes to manage message-oriented

platforms: message passing and message queuing. Message passing (implemented for example through MPI) is a simple method where the middleware just takes the message and sends it through the network to the chosen receiver. The protocol can then be extended to multi-hop communication to extend the reach of a single user. Message queuing instead elevates the middleware at a central role in the system: in fact, it acts as the enpoint for all the messages, and routes them to the right receiver by itself. This approach is usually called MOM (message oriented middleware) and the "queue" regards the managing of the messages to/from clients: these messages are stored in queues (respectively of output/input to the mom).

7.1 Publish Subscribe

The perfect example of a message oriented system is a publish-subscribe kind of application: the publishers encapsulates items in messages and then sends them to the middleware, that takes care of multicasting the message to the right subscribers. Subscribers must tell the middleware which events are they interested into: to do so, a event describer language is needed (also subscription language). This language can be of various types: it can filter events basing on attributes they specify that are encoded in each event, or can specify a template for the content itself of the events and ask to receive only the ones that adhere to such template.

Event Dispatcher The central actor of the publish subscribe architecture is the event dispatcher, in other words the component that carries out all the routing of the events and of the subscription. Obviously, it should be distributed (we skip the recurring step of saying why a centralized point of routing does not work) and distributing a dispatcher poses the problem of how to route events and how to route subscriptions.

- Events: simplest approach, network flooding with events, then subscribers themselves will sort them out. Network usage is intense, in this case. Another way is to keep track (in the "local" event dispatcher) of the subscriptions of the connected clients, so to drop the event they're not interested in. This suggests a more hierarchical approach, where a single dispatcher have multiple "child nodes" and keeps track of their subscriptions
- Subscriptions: again, a hierarchical approach is better than a flood-and-pray one: dispatchers can be organized in a tree. Obviously, this may happen only in cycles-free topologies.

The event dispatcher can also make use of a DHT to route packets: this enforces a ringy overlay network to route subscriptions and events.

8 Stream Oriented Communication

Stream oriented systems are suited for situations where data comes like a continuous flow of information. Multimedia streaming is the classical example. The

major problem of stream-oriented communication (and in particular for streaming) is to concile fast transmission and QoS guarantees. This is often made by relying on a best effort protocol for transport (as UDP) and enforcing QoS at application level.

QoS at application level When basing on a "fast and unreliable" protocol (UDP) Quality of Service enhancing techniques must be carried out at the current level.

- Error recovery: techniques like interleaving packets (that mitigate the effect of burst error) or error repairing (sending additional information in the very packet to rebuild in case of soft errors) can be used to ensure a error-free communication
- Flow steadiness: to avoid listener stall (or lagging, when it comes to streaming) buffering techniques are ubiquitous: contents is buffered in the receiver before being sent to the application, then steadily served as a continous stream to the system.

Another problem to face is sender-receiver synchronization, especially in the case of multiple streams.

Part III
Naming

9 Flat Naming

9.1 Flat naming approach in general

When a system is said to use flat names for its peers it does not assign a meaning/value to the identifiers of the peers. This means that IDs of the connected hosts are just plain IDs and a the only procedure to match a name against an address (name resolving) is to have somewhere memorized this relation and to be able to access this information.

9.2 Name Resolving Techniques

The procedure of "name resolving" links and address to a name/ID. In a flat named world, various methodologies can be put in place, but they all boil down to memorize somewhere the couple { name, address } and accessing it.

- ARP approach: the network is flooded with "who knows x?" requests, then hosts that actually have a link to x responds with "I know x, he's in y"
- Hierarchical approach: ARP does not work well is big sized network, so the name resolving is centralized to a special node that memorizes all the { name, address } couples. This node is connected to all his hosts (that are the leaves of the tree) and to a parent node, that acts exactly

like him: contains the { name, address } of all peers of the subnetworks it's managing. Iterating this architecture leads us to a tree-like structure. PROBLEM: the root node should contain the information ragrding *all* peers in the network. PARTIAL SOLUTION: heavy caching.

- Home based approach: a specific case of hierarchical system, where the tree has only three levels; the leaves, the homes, the root. The idea is: every "home" knows always where a peer associated to it is, and can always return his address. The root is useless in terms of name resolving, it's just there to completely connect the tree. PROBLEM: extra step towards home increases latency. PROBLEM: mobility makes the work of the home nodes difficult.
- DHT approach: as all the DHT the idea is the same, storing the couple { name, address } in a hash table that then references other peers in the network for the lacking ones, in a ring-style fashion of overlay network.

10 Structured Naming

In general: opposing to the flat naming approach (plain names with no meaning) the structured naming approach suggests a more functional use of the identifiers of the hosts, where every possible identifier is generated according to certain predetermined rules, so that all names belongs to a structured *name space* and they're easier to lookup.

10.1 Structured naming systems's name resolution

A system that makes use of structured names usually adopts a hierarchical approach to name resolution: in fact, every layer of the tree can carry out the routing of a certain part of the name. A perfect example is the filesystem: every name is composed of a tuple of identifiers that all together represents all the path from the root to the file (or directory) that name is referring to.
Systems like that (another example, the DNS) usually make use of different machines for every layer, and this pose the problem of how to efficently propagate the query? Is it better to recursively propagate the requests among all intermediate servers, or to iteratively send back to the client the address of the specific node it's requesting, descending the tree?

11 Attribute Based Naming

In a distributed system, names can also be assigned basing on the attributes of the object to be referenced. This is useful (and meaningful) in systems where search queries are attribute based, like directory services or DBMSs.

12 Removing unreferenced entities

The naming system generates names and associates them to addresses. What happens when a name becomes unreachable, i.e. when all the references to it are deleted? It must be removed from the system.

12.1 Reference counting

Focusing on object-orientedish distributed systems, a method to remove unreferenced objects is to add to the single component the ability to count how many referencers are pointing at him. This is usually implemented through a decrementing counter that's decreased every time a reference is added and increased every time a reference is returned. When the counter is back to the initial value, the objects suicides.

12.2 Weighted reference counting

To overcome the inherent limitation of reference counting, a slight modification is weighting the references. The counter does not distribute a single share of otself, but a weighted reference (a >1 value) that than can undergo the same procedure when the reference needs to be passed around again.

12.3 Reference listing

Another approach that makes explicit use of the inward references is reference listing. Instead of giving away "shares", reference listing simply suggests to list all incoming references. When the list is empty, objects suicides.

12.4 Distributed mark and sweep

Tricky cases of isolated circular referencing can't be managed by the components themselves because it's very difficult to make them aware of a problem like that. Distributed mark and sweep instead proposes a network flooding algorithm that labels referenced entities (all white, if traversed grey, if all sons traversed black) then scans the whole memory of the machines to search for unreferenced entities. PROBLEM: blocking operation.

Part IV
Synchronization and time

13 Absolute time solutions

With "absolute time solutions" or "physical clock synchronizing" are intended all the algorithms/methodologies to synchronize all process against a clock that's universal (often a real clock) and keep all the process in line with the effective flow of time.

13.1 GPS' approach

Atomic clocks are installed in every satellite and the delay between sending and receiveing a packet is kept in consideration during the calculation of the clock skew between packets. There's a complicated formula in 4 incognite to resolve to get to the actual time step.

13.2 Cristians' algorithm

The Cristians' solution consists of synchronizing all clocks against a fixed one in the network. Periodically, all peers send a "time request" to the "time server" that responds with the actual read time. The clients then add the round trip time to the received timestamp and updates its internal clock.
$clientTime = receivedTime + \frac{RTT}{2}$

13.3 Unix time solution

Unix clock synchronization works similarly to the Cristians' algorithm: the main difference is that time is not unique, but it's calculated averaging all the timestamps received from the clients, then sending to each the adjustment relative value.

13.4 NTP

Network Time Protocol is a protocol based on UDP that hierarchically distributes UTC timestamps.

14 Logical time

Often, the actual flow of time is an information that is not needed, when a logical relationship between events is maintained among processes.

14.1 Scalar clocks

A first solution is to maintain a scalar value for each process that's incremented every time a shared action (sending a message) is performed. When a process sends a message it includes its scalar clock. When a process receives a message it updates his own scalar clock to $max(ownScalar, receivedScalar) + 1$. This ensures a somewhat coherent logical flow of time in the processes.

Totally ordered multicast Scalar clocks can be used to obtain total ordering in a network. Every peer on the network timestamps the updates with his scalar clock and multicasts them to the network. All the acks for a message must be received before forwarding the message to the application. Eventually, due to the global ordering of messages the scalar clocks provide, all peers in the network receives all messages and are able to order them in the right order, assuming reliable FIFO links.

Mutual Exclusion Mutual exclusion is the condition that ensures that no processes access at the same time the same resource. Mutual exclusion can be simply achieved through the use of scalar clocks: a process floods the network with a request for a resource, attaching his scalar clocks. It will access the resource only if it receives an ACK from all other processes. If a process receives a request, three scenarios are possible:

- the process is not interested in the requested resource: it ACKs the request

- the process is holding the resource: it does not ACK the request, but will do when it releases it; the requests are memorized in a local queue ordered by clock
- the process is waiting for the resource: in this case, it must check wheter the scalar value in the request is greater or smaller than the one in his own request, and act accordingly: if it's smaller, it means that the incoming request is older than his own, so it acks it. In the opposite case, it does not.

NOTE: a token ring network can also achieve mutual exclusion, in the case of a single circulating token.

14.2 Vector clocks

Vectorized extension of the scalar clocks: a process does not hold anymore his internal (global) state, but the state of all the processes in his pool, memorized as scalar clocks in a vector. The updating mechanism is the same as scalar clocks but replicated on all elements in the vector.

Causal delivery There can be situation in which totally ordered multicast is a stronger condition than required. Vector clocks can be used to implement causal delivery: the update function for each process present an increment only on sending a message (not more also on receiving, the value on receive is only updated) and the vectorized clocks carry enough information to grant a causal relationship between messages.

15 Leader election

A coordinator is needed when performing certain parallel computation. If no leader is designated (or the current one fails), an algorithm to identify it must be developed and embedded in all the processes of the pool.
Leader election is the procedure that make all processes agree on the new coordinator. The minimal assumption that must be made is that nodes are *distinguishable*. Also, the system must be *closed*, so every node knows the IDs of all the other.

15.1 Bully algorithm

When a process crash, the process that detects the fail sends to all the higher-pid processes a ELECT message. If he gets no responses he auto-elects himself and sends a COORD message to all processes. If a process have an higher PID than the one in the received ELECT message he responds and sends another ELECT message, with his pid now.

15.2 Ring based approach

When the coordinator fails, the process that detects the failure sends to his *successor* an ELECT message with his PID. When an ELECT message is received, a process checks if their PID is inside, and if it is, it changes the message to

COORD with attached the highest PID in the packet. In two rounds of the ring a new leader is established.

16 Collecting global state

For global state it's generally intended the set of states of the single processes in a network, *plus* the state of the channels, at a given moment. To extend the normal "state" concept to a network, we introduce the *cut*: a *cut* is a set of events (of different processes) that is *consistent* when for every event all his predecessors are in the cut. This means that a "message sent" event can always appear in a cut, but a "message received" can appear only if the "message sent" correlated is present.

16.1 Dsitributed snapshot

Lamport's algorithm to record global state: it relies on *reliable FIFO channels and strongly connected networks* and works this way

1. A process decides to start a snapshot, records its internal state and sends to all the connected peers a snapshot token
2. If a process receive a token, it closes the channel where he received it (so it will not accept further packets from that channel) records his internal state, broadcasts the tokens and starts recording the messages cominng from the opened channels
3. The distributed snapshot ends when all channels of all processes are closed.

It has been proven that this method records a consistent cut.

16.2 Dijkstra Scholten termination detection

For diffusing computation (where every process is awaken when a message is received) telling a termination from a deadlock is no easy task. Dijkstra Scholten algorithm works by constructing a tree (daring today) out of the cuncurrent processes, and telling nodes to suicide if all their leaves are idle and the node itself is idle.

17 Distributed transactions

Transaction: sequence of basic data operation with the ACID properties. Atomicity, Consistency, Isolation, Durability. Transactions can be

- flat
- nested
- distributed: flat transactions for distributed data

Atomicity approaches:

- private workspace

- writeahead log: local modifications, distributed agreement on global modify

Cuncurrency control approach:

- 2PL or Strict 2PL
- Optimistic timestamping: stamp *only* data items with start time of transaction; at commit, if any items have been changed since start, rollback.
- Pessimistic timestamping: every *transaction* have a timestamp, and every operation must arrive "at the right time" (logically speaking) to be performed. For example, a write is committed only if its timestamp is the "newest" among the last read/write on the target data piece. (So, double timestamping: both data *and* transaction)

18 Distributed deadlocks

Deadlock: the condition in which a network of processes holds and requests resources in a "crossed" manner that implies the halting of all them, with no possibility of destalling. Obviously, distributed systems suffer from this problem, adding the complexity of discovering the deadlock among *different machines*.

18.1 Distributed deadlock detection: Chandy Misra Haas

A deadlock can be visualized on a resource graph by a cycle in the arcs. Using this property, the CMH algorithm works sending to all processes that are holding requested resources a probe message, and asking them to do the same. If the probe "completes the cycle" that process suicides.

18.2 Distributed prevention

Distributed prevention is the name that is usually used to define the design choices that are made to avoid deadlocks. One of those could be a "global timestamping" of processes that allows them to request an already occupied resource only if their timestamp is "older".

Part V
Replication consistency

19 Replication models

19.1 Active vs Passive

A passive replicated system acts like a non replicated system (so single bottleneck for all operations) with backup copies (followers) that are cascadingly updated when the master propagates the new values of the data. To obtain K-fault tolerance, at least K+1 sopies must be active. NO SHARING OF

WORKLOAD.
An active replicated system, instead, allows read and write operations from all copies. To ensure consistency, a coordination is needed:

- single leader systems make use of a single replica to coordinate among the others. While a read operation can be issued to any replica, all writes must pass through / be authorized by the leader. This architecture is suitable in particular cases in which it's not so costly to reach the leader for confirmation.
- multiple leaders systems, on the other hand, work by "replicating the leader". To avoid single-leader bottlenecks, a fleet of leaders can be deployed, that can speed up write operations. They must be synchronized among themselves also.
- leaderless system instead trust a more democratic approach to committing write operations: "a read on a certain value and a write operation can be performed iff enough replicas agrees on the value being read/written"

19.2 High availabity and consistency models

A system is *highly available* if it does not require blocking interactions in order to complete operations.

19.2.1 Data centric models

Strict consistency Strict consistency systems are difficult to build: the idea is that in a distributed system, no errors can happen and all peers are aware of the nature and timing of all operations in the network, so to build a consistent model (of data) locally.

Serializability There are few systems that *need* strict consistency. Serializability is another "strict" model, that does not need "all knowing" peers. In fact, the only information peers should agree on is the order in which a certain set of operations (that can be of different peers on different data) is carried out.

Causal consistency Causal consistency systems preserves the order of the "logically connected" operations. So, the read-followed-by-write relationship and the write-followed-by-write realtionships must be agreed on among all peers.

FIFO consistency Least strict models, preserves only the order or write operations.

19.2.2 Client centric models

Read your writes If a client writes variable x with value A at time t1, for all times t2 $>$ t1 it can't read an older value of x

Monotonic reads If a value x has been read on time t1, no older value can be read on time t2 $>$ t1

Monotonic writes Write operations are carried out before another write operation on the same variable is completed.

Write follows reads A write operation on x performed after a read on variable x is guaranteed to take place on the same or more recent value of x

20 Update propagation

20.1 Propagation strategies

Two approaches are possible when dealing with "who updates who"

1. Push upgrades ("Hey you! I got upgrades!")
2. Pull upgrades ("Hey you! Have you got updates for me?")

20.2 Updates forwarding strategies

A server has an update to spread. It can

Anti enthropy approach Select a single neighbour and update it

Gossiping Send an update to a server, that then sends to a server ecc. When the update comes back, reduces the probability to propagate it

Broadcasting Just a special case of gossiping at the end of the day

Part VI
Fault tolerance

21 Agreement in process groups

21.1 FloodSet algorithm

The floodset algorithm is applicable to synchronous systems that also evolves synchronously. It resolves the agreement problem in a fail-silent scenario: in total, we are considering a very small group of distributed systems (fully synchronous, no timing failures, no byzantine failures). It's very simple, a set of values to be agreed on (values for a single variable) is passed around. If after n rounds (n number of faults to be proof-safe from) there are more than one values in the set, the returned value is the default one.

21.2 Byzantine failures handling

If we add to the FloodSet hypothesis the possibility of byzantine failures things get messy. Lamport showed that a minimum of 3m+1 processes are needed to be m-fault tolerant.

21.3 Impossibility of distributed agreement

If we add asynchonicity it has been formal proven that distributed agreement is impossible.

22 Reliable group communication - faulty channels

22.1 Scalable reliable multicast

NACK approach: instead of sending an ACK on every packet received, every process sets a timer that resets when a packet is received. If the timer reaches zero, a Negative ACK is sent. This reduces the number of packet on the network.

22.2 Hierarchical feedback control

Super classic hierarchical control. Networks are divided in subgroups, each connected through a coordinator, that can apply the strategy he prefers in the subnetwork he coordinate. The coordinator subgroup is managed separately.

23 Reliable group communication - faulty processes and faulty channels

23.1 Close and virtual synchrony

Close synchrony is the ideal model for synchrony: every process in a multicast group sees all the events in real time and all in the same order. Close synchrony cannot be achieved in presence of faulty processes.
Virtual synchrony is a model so that

1. crashed processes are kicked and must rejoin
2. all messages from non faulty processes are processed by all non faulty processes
3. messages from a faulty process are processed by all or by none
4. global ordering of relevant messages: "view changes" (so changes in the process pool structure) are send in a consistend order wrt the other multicasted messages

All multicasts must take place between view changes.

23.2 Virtually synchronized systems - message ordering

Possible ordering strategies for multicas messages:

1. unordered multicast.
2. causal ordering: the logic relationships among messages (as receive-send) are maintained, and FIFO.

3. fifo ordering: messages sent in a particular order from a peer are received in the same order from all others.

Total ordering A system is said to implement total ordering if all processes receive the same set of messages all in the same order. If A receives x before y, there's no way B receives y before x

VS implementation

1. An external component detects a view change (process failure / channel break) and issues a view change to all the pool
2. All peers flush to the network the set of *unstable messages*, the messages received that have not be acknowledged by the whole pool yet, and marks them as stable. When all processes have received an ACK for all the unstable messages they can restart communication.

24 Distributed commit

24.1 Two phase commit and three phase commit

Distributed commit and distribute agreement suffer the same problems: all processes (non faulty) must agree on a single value ecc ecc.

Two phase commit Every process acts as a FSA with 4 states: init, wait, abort, commit. The initiator (or transaction manager) sends out a vote request, and when it receives all vote-commit messages broadcasts a global-commit. A peer in the init state waits for a vote-request and then either vote for commit and waits or aborts.
This method stalls if not all processes votes.

Three phase commit As two phase, but non blocking, and also can detect and overcome process failures. A state is added (pre commit) that signals the condition of having received only commits but not from all peers. If it's the coordinator that fails, a peer can deide what to do basing on which state has been reached by other participants.

Part VII
Big data

25 General information

Big data is the term that describes all the data collected from various systems as a whole. Properties of big data:

- Velocity: data is collected / generated *fast*
- Variety: not only file formats, also the semantic of data can be various

- Volume: lots of data
- Veracity: possible incorrectness

26 Batched approach versus stream approach

26.1 Batch processing

Data is collected and organized in batches that are indipendently elaborated and passed around.

26.2 Stream processing

Data is a continuous stream of information. Operations are arranged as ”filters” or ”mappers” that modifies this stream until it reaches the desired format for output

26.3 Comparison

Latency Batched approaches suffer of high latency in the case of intermitting data flow, because of the need of organize data in batches. Stream oriented approaches are less latent, due to the intrinsic ”elaborate on the fly” architecture.

Throughput Batched systems are more efficient in term of throughput, due to the scheduling that can be applied to batches to optimize the flow of data.

Load balancing Batched systems (for the same reasons of their advantages in throughput) can apply more sophisticated load balancing operations.

Elasticity Stream oriented systems are very rigid: the operation must take place in a determined order at a certain time, and scheduling cannot be applied.

Fault tolerance You kidding me? Resuming a stream oriented computation after a failure is much more complicated than a batched one.

Part VIII
P2P

27 Design of a p2p system

A p2p system has no centralized actors all clients rely on to carry out a specific operations. Instead, all peers are interconnected and they exchange directly resources, wheter this are data, storage, bandwidth and so on.

28 Approaches to search and lookup

Searching for a resource is a critical point in a p2p system. Three classical approaches:

Centralized search Napster's solution: all operations are fully decentralized but search and lookup, that go through a fixed point of the network that has all the information.

Hierarchical approach Kazaa's approach: same as napster, but the central fixed point is itself a distributed system of peers that acts as a single one. A client connects always to a central fixed peer to search, and then to the single peer to exchange resources.

Fully decentralized approach - query flooding Gnutella adopts a different style when it comes to network organization: no central authority (at all) and network flooding for request. To avoid network overflow, search packets include an HopsToLive field.

Part IX
Security

29 Design issues

Security can be placed in 3 different places in a distribute network: on the data, encypting it; on the operation thatcan be performed on data, to limit the changes and threats to it; on the user, limiting their capabilities.

30 Certification

30.1 Certification authority

What is a CA? It's a well known entity that ensures the veridicity of a certificate, so a tuple name + key used to associate a person to a key. CA are *well known* in the sense that the power of their role relies on the universality of the CA identity information. Checking all keys manually is a hell, so CA adopts a hierarchical approach. Every certificate has an expiration date.

31 Secure channel

Secure channels provide protection from spoofing and data alteration. The authentication happens in a challenge response fashion, where each peer asks the other to encrypt some data with a shared key, then verifies if the data is encrypted correctly. A solution with an asymmetric key is the classic HTTPs approach. A Key Distribution Center can avoid key wearing and can lift the network from the burden of generating and destroying the keys.