Distributed and pervasive system

University of Milan
Master's Degree in Computer Science
A.Y. 2022/2023

Stefano Vallodoro

Indice

1 Introduction

What is a distributed system?

Tanenmbaum's definition *Is a collection of indipendent computers that appears to its user as a single coherent system*

This definition has two important aspects. Firstly, it comprises autonomous independent components, often computers. Secondly, the user perceives the distributed system as a unified entity. This means that the autonomous components needs to collaborate

Middleware layer Distributed system is organized as a middleware that it's logically placed between applications/users and the operating system. This layer spans multiple machines and provides a uniform interface to applications, hiding any hardware/OS differences

Lamport's definition *A distributed system is one in which the failure of a computer you didn't ever know existed can render your own computer unusable*

This definition put the finger on an important issue of the distributed system: dealing with the failure.

Goal of distributed system

- *Making resource accessible*, make easy for the user and application to access remote resources, and to share them. There are many reason to want to share resources, one obvious is that economic. It's cheaper have a resource shared by several users than having to buy and maintain a separate resource for each user.
- *Distribution transparency*, the complexities should be hidden from the user who uses the distributed system
  1. *Location*, access objects without knowledge of their location
  2. *Access*, objects are accessed with the same operations regardless of whether they are local or remote
  3. *Migration*, hide that a resource might move to another location
  4. *Relocation*, it's similar to migration but while in use
  5. *Replication*, a resource is replicated without any effect
  6. *Concurrency*, consistency of shared resources is mantained despite being accessed and updated by multiple processes
  7. *Failure*, hide failure and recovery
- *Opennes*, should offer: interoperability; portability; extensibility
- *Scalability*, can be measured along three dimensions: size; geographical; administrative

1.1 Types of distributed systems

Clusters A collection of similar workstations/servers closely connected by high-speed Local-Area Network and usually running the same operating system. The goal is high performance computing tasks or high availability

- *Asymmetric approach*, there's a node that organized the work for other nodes. Examples: Google Borg, Kubernetes
- *Symmetric approach*, there is no master node. Example: Mosix

Cloud: service models *IAAS*(hardware), CPU memory and datacenters; *PAAS* (Databricks); *SAAS*, application that run on the cloud (Google Docs, Office 365, Gmail). Can be deployed in different ways: private, community, public, hybrid

Edge computing Focuses on processing data within IoT devices themselves. In practice, the idea is to move data processing from the cloud to the IoT device to reduce latency and increase data processing speed. This way, data is processed locally, and only relevant data is transmitted to the cloud for further analysis.

Fog computing Focuses on distributing data processing across multiple nodes within the IoT network. Instead of processing data only on IoT devices, Fog computing uses a set of distributed nodes within the network. This way, data can be processed more efficiently and latency can be reduced without necessarily transferring all data to the cloud.

1.2 Pervasive computing

Weiser's definition *The most profound technologies are those that disappear they weave themselves into the fabric of everyday life until they are indistinguishable from it*

Adaptivity Depending on context the system is capable to understand and change their behavior for optimize the system goal

Unconventional nodes Not designed for computing but with computing and communication capabilities. Introduces challenges such as limited resources, variability in connectivity, and variable positions. There is also high volatility due to device and communication system failures, changes in communication characteristics, and the dynamic nature of the system with a high number of nodes that may associate or dissociate.

2 Distributed system architecture

2.1 Centralized architectures

Client-server A client-server architecture is a model for organizing computing tasks where clients request and receive services from servers. The specific allocation of tasks between clients and servers can vary, such as having applications and databases hosted on the server or only having the database on the server while the application runs on the client.

Multi-tier client-server The server part is decomposed. Data manager separated from application logic. Application can change but data still stable. Microservices are an evolution of this architecture

Vertical distribution Different node for each functionality, different process in different machine. This is the main goal for microservices

Horizontal distribution Same functionality distributed on multiple nodes with load balancing

Event bus Publish-subscribe communication pattern. Is a request of interest

2.2 Decentralized architectures

Peer-to-peer All the nodes have the same functional capabilities.

Three generation of P2P systems, more conceptual than cronological

1. *First file sharing*, Napster
2. *Improved scalability, fault tolerance, anonimity*, Gnutella, BiTorrent
3. *P2P middleware*, Chord, Can

Goal of P2P middleware

- *Lookup function*, locate and communicate with any resource
- *Add and remove resources*
- *Add and remove peers*

Overlay network Logical network, over an existing lower level network, in which nodes are formed by processes and the links represents possible communication channels

Structured overlays Built with deterministic algorithm, hash table

2.2.1 Napster

It's not a pure system, there is an index server where you can find the list that actually might have the file but there's no a broker, operation does not depend on any centrally administrated system

1. File location requests
2. Lists of peers offering the file with an address IP and PORT
3. IP allow me to goes here and the PORT tell me where the Napster process
4. File delivered
5. Index update

2.2.2 Chord

Is a distributed protocol to locate nodes and objects, useful for optimize search resources in a system

Main goal Quickly mapping a resource to a node

Main idea DHT: Distributed Hash Table. DHT is the general mechanism to enabling $O(\log N)$ research

The lowest among the greater An addressing space for 1 to n bit (up to 160 bits) is fixed and each nodes is assigned one of the addresses in this space. Each resource is assigned an address taken from the same address space used for nodes and is managed (stored) by the node that have the smallest id greater than the resource.

Address of peer One way to provide a unique identifier for a peer is to combine its IP with port number. By applying a hash function to this combination, we can obtain an address within the address space of the distributed hash table. Nodes are more more less than resources

Finger Table Is a specific implementation of a DHT. Contains a list of other nodes in the system, and the distances between them in the hash space. Specifically, the ith entry in a node's finger table contains the node that is responsible for the next 2^i hash values after the current node's identifier

Each node gets as *id* an address in the space by hashing its IP and port. Each data item gets a key (address) in the same space, a file gets a key by hashing its file name. A data item with key k is managed by the first node with $id >= k$, called *succ(k)*

If m is the number of bits in the address space each table has m entries

The formula to obtain the right number is $succ(p + 2^{i-1})$ where p is the address of the node/current peer, i the raws

Chord node insertion and deletion Chord must update the routing information when a node joins or leaves the network. Addresses of successor/predecessor nodes must be updated, resources must be moved, and finger tables must be updated. A join or leave requires several messages in the network $O(log^2 N)$

Limitation of structured overlays

- Difficult and costly to achieve, especially in dynamic environment because the peers appear and disappears very frequently
- Maintenance of the hash table
- Organize the structure when a failure happens

Unstructured overlays Built with random algorithm

- *Gossiping*
  Each node gossips with a subset of its neighbors, which in turn gossip with their neighbors, and so on, until the information has spread to a significant portion of the network
- *Superpeers*
  Keeping an index of data for a subnetwork. Each superpeer doesn't know where the resource it's locate and instead to extend on all network the request just ask to the peer that are in charge on the subnetwork. Peer organization in the network is hierarchical, some peers are more important than others. Napster's directory server is an example of superpeer.

3 Communication model

Communication is the core of distributed system because we have heterogenous nodes/machines. The only way to they can work together it's to communicate with each other

Middleware protocols All the protocols are always above the transport layer

3.1 Persistency communication

System that guarantees the delivery of a message to the destination in case the receiver cannot receive it. You have persistence communication when there is some infrastructure that take care of the messages even the recipient is not ready to handle that

- *Persistent asynchronous*, doesn't wait for any answer by the server. Put only a notification and then continue to work. In case B is not running, the message will be saved in the intermediate storage system between the server and the client
- *Persistent synchronous*, message is stored for later delivery. Must be a system that accept the request. Example: one tick on Whatsapp

Queuing system Persistent asynchronous example: message on the phone

- *Put*, append a message to a specified queue
- *Get*, until the specified queue is non-empty, remove the first message
- *Poll*, check a specified queue
- *Notify*, when a message is put into the specified queue

Message broker Helps nodes in a distributed system communicate by converting messages between different formats or protocols, translate the messages so that they can be understood by all nodes. Ensure also access transparency

Protocols *XMPP*, *MQTT: Message Queue Telemetry Transport*, *AMQP*

3.2 Transienty communications

Dual of persistency is the transienty, if the server is down it is not possible to carry out the communication

- *Transient asynchronous*, sends message and continues. Message can be send only if B is running
- *Receipt-based synchronous*, ack of received
- *Delivery-based synchronous*, wait until start elaboration
- *Response-based synchronous*, wait until he received result of his request

Berkley Sockets When you connect two processes with socket you need both processes to be active that's why it's transient. Socket is the end of a communication channel between nodes

Primitives

- *Socket*, initializes a new communication point. What the socket does is tell to the operating system to prepare some structure for this process to handle this logical connection
- *Bind*, give a port on the machine that unique identify this socket so the people outside can connect and send message
- *Listen*, prepare a memory area to queue messages and connection requests
- *Accept*, blocks code execution until a connection request arrives in the listen queue
- *Connect*, with IP and PORT
- *Send*, send messages from one machine to another
- *Receive*, receives messages from the other machine
- *Close*, closes the connection between two endpoints

3.3 Remote Procedure Call

Abstract layer for communication between nodes. It's a method of introducing transparency of access to message communication. We want to make sure that in our software we normally call, locally, a procedure that will run remotely on another machine. The main difficulty of the RPC is the passing of the parameters that include data structures, which are usually passed as references to memory areas where the data is stored. This works well on the same machine, but in remote communication the memory is not shared.

Stub Allows us to make this abstraction. It take care of marshalling and unmarshalling operations

Serialization Adjust parameters before passing them over the network

Marshalling Format RPC parameters in a message

Unmarshalling Inverse procedure

Binding a client to a server

- *Register endpoint*
- *Register service*, directory server in which was written that there's this service
- *Look up server*, searching a specific service
- *Daemon will answer to the client*, daemon can give the client the port
- *Do RPC*

4 Synchronization

Nodes must synchronize, agree on tasks to achieve common goals

4.1 Physical clock

Physical clocks are based on an oscillation of a quartz crystal properly put into tension, for this we need the battery. Associated to the crystal there are two registers, one is called *counter* and one *holding register* and at each reel the counter is decremented until it reaches 0 and generates an interrupt, then it is initialized to the value of the holding register and start again

Atomic clock Transaction of an atom of *Cesio133*. The average solar day corresponds to about 9/10 billion oscillations.

TAI International Atomic Time, an average of the atomic clocks on Earth. If computers use a number to adjust to our time then where does this time come from? It does not come from the computer clearly or otherwise not directly. What time is used as a reference in Computer Science? UTC that is TAI plus leap seconds. Leap seconds are introduced (once in a while) to keep our time in phase with the sun

Using GNSS *GPS* is the most used Global Navigation Satellite Systems. There is a server on Earth connected with an atomic clock. GNSS have one or more atomic clocks onboard

GNSS receivers listen to multiple satellites and use *trilateration* to determine their own position and UTC time deviation. The satellite sends a message that is received by the GPS receiver, calculating the time spent between sending and receiving, obtaining the distance, now it knows at what distance it is from the satellite which has a known position.

The Berkeley algorithm: internal synchronization Do not need to synchronize to an external time (UTC), only the nodes of the system are synchronized between them.

1. *Time daemon asks all the other their clock values*, sends a message containing the time
2. *The machine answers*, saying the difference between the time of the message and their time
3. *Time daemon tells everyone how to adjust their clock*, daemon calculates an average of these times, it uses the average to try to minimize the movements of the clocks, and communicates how much the various clocks must move

This algorithm allows us to go back, so in case of negative delta does not bring back but the clock is slowed down until it synchronizes with others

4.2 Logical clock

It may be sufficient to share the knowledge of a partial order of events on the distributed system

Temporal precedence Each node must be aware that a certain event occurred before another

The idea An integer is used at each node as a logical clock. It's incremented every time an interesting event occurs

Happened before relationship If *a,b* are events, the expression $a \rightarrow b$ denotes the relation *a happen before b*. It is a transitive relation

Lamport's algorithm Lamport's algorithm allow us to have a partial order on events occuring

$C(a)$ is the logical clock value assigned by the process when a occurs. The value of a logical clock can only increase. Goal: if $a \rightarrow b$, then $C(a) < C(b)$

1. Before executing an event P_i executes $C_i \leftarrow C_i + 1$
2. When process P_i sends a message m to P_j, set *ts(m)* equal to C_i
3. When P_j receives message m at C_j, adjusts its own counter as

$$C_i \leftarrow max(C_j, ts(m)) + 1$$

Enforcing total order Add *process number* to the timestamp of an event. It can be represented as C(a).i, where i is the unique identifier of each process.

We can now define a *label* as clock.identifier. Modify Lamport with total order could be very useful because it can happen that two clocks are equal. To implement the multicast with total order there is a need to assume that there are no lost messages, and that messages from the same sender are received in the order in which they were sent

1. The process P_i sends the m_i message with timestamp (label) to all other processes. The message is placed in a local queue i.
2. Any incoming message from P_j is queued in its j queue according to the timestamp, and an ACK (the real key of this solution) is sent for delivery to all other processes.

3. P_j passes the message mi to the above application if:
    - m_i is at the head of the queue j.
    - m_i was received and ACKed by all other processes.
4. All processes will have the same copy of the local queue, so all messages are passed to the application in the same order in all nodes.

5 Mutual exclusion

A set of a process needs to concurrently access a shared resource, the only way in a distributed system to agree is to send messages

5.1 Centralized

1. Process 1 asks the coordinator to access the resource. Access is granted. In the request written the id of the process and the name of resource
2. Then, process 2 asks for access. The coordinator does not answer, but adds the request to the queue in FIFO order
3. When process 1 is done it notifies the coordinator that grants access to the first process in the queue

Drawbacks

- Single point of failure
- It's difficult to distinguish a problem of the coordinator process from the unavailability of the resource
- Starvation

5.2 Ricart and Agrawala

All the nodes of the system participate in the coordination

Assumption Total order of events and ACK system

1. If P wants to use a resource R it builds a message *<R, id(P), timestamp>* and sends the message to all the processes including itself
2. When a process Q receives a message we have 3 cases
   - If Q neither uses nor requires R it answer OK to P
   - If Q is using R it does not answer and it queues the request
   - If Q wants to use R but it did not yet, it compares the message timestamp with the one in the message, the earliest wins.
3. After sending out its message process P waits for OK from all processes before accessing R
4. When P is done with R it sends OK to all processes in its queue and empties the queue

Drawbacks

- Involving all processes may be a waste of resources
- We have a problem if there's a crash and no answer

5.3 Token-ring algorithm

Unordered group of processes, a software defined logic ring

1. The token is given to process 0
2. The token is sent with a message from process i to process $(i+1)mod_n$
3. A process can access the resource when it receives the token

Drawbacks If no node needs the resource, the token go ahead

6 Election

In the event of a crash, the coordinator may fail, so it takes an automatic system to elect a new coordinator in order to ensure transparency. Election of the *active* process with the highest identifier

6.1 Bully

Assumptions

- The system is synchronous
- Each process is aware of its own ID and network address, and that of every other process

Steps

1. If a process P has the highest process ID it sends a *victory message* to all other processes. However it broadcasts an *election message* to processes that have higher process IDs than itself
2. If P receives no answer becomes the coordinator
3. If P receives an *answer message* from a process with a higher process ID, it does not send any further messages and waits for a victory message. If it does not receive such a message within a certain period of time, it will restart the election process from the beginning
4. If P receives an *election message* from another process with a lower ID it sends an answer message back and if it has not already started an election, it starts the election process at the beginning

Worst case When the process with the lowest ID initiates an election. This process sends $n-1$ election messages, the next higher ID sends $n-2$ messages, and so on, resulting in $\Theta(n^2)$ messages

6.2 Chang and Roberts

Assumptions

- n processes are logically ordered in a ring. P_i process has a communication channel with $P(i+1)mod_n$
- Messages circulate clockwise without failure
- Processes have unique IDs

Steps

1. All processes are marked as *non-participant*
2. When a process P_k understands that the coordinator is not answering, it starts an election by marking itself as participant and sending to the next node in the ring a message $< Election, ID(P_k) >$
3. When a process P_m receives the election message
   - If $ID(P_k) > ID(P_m)$ forwards the message in the ring and marks itself as participant
   - If $ID(P_k) < ID(P_m)$ if P_m is non-participant it changes to participant and forwards *<Election, ID(Pm)>*, otherwise it does not forward the message
   - If $k = m$ then it is the coordinator. It marks itself as non participant and sends *<Elected, ID(Pm)>*
   - When a process P_k receives the elected message it marks itself as non-participant, stores the id of the coordinator, and unless $k = m$ it forwards the message to the next process

Drawbacks If each node only knows the next address of the ring, in case of a node crash the ring is broken. In practice each node also knows the addresses of k successors, in order to avoid the problem of crashes

Worst case $3N - 1$

7 Fault tolerance

Dependability Being fault tolerant is strongly related to what are called *dependable systems*, it implies

- *Availability*, run correctly at any given moment
- *Reliability*, run correctly for a long interval of time
- *Safety*, failures must be handled
- *Maintainability*

Failure type

- *Crash failure*, the processes fails by stopping but execute honestly. The most benign type of failure
- *Omission failure*, a server fails to respond to incoming request
- *Timing failure*, server answer to request too late and triggers timeout
- *Response failure*, response is incorrect
- *Byzantine failure*, arbitrary responses at arbitrary times. Normally caused by intrusion of malicious nodes in the system

State machine replication To move around the problem of failure we replicate the data having multiple copies of it. You have a server/machine, and the state reflect a sequence of operations, the log. The state machine is the component that we want to make fault-tolerant. The problem can be filter down to a problem of managing a replicated log. It will appear to clients that they are interacting with a single, reliable state machine

Information redundancy Extra bits used to recover transmitted message when noise is present

Time redundancy A transaction is repeated if aborted because failure of one of its actions

Physical redundancy Extra hardware or software/process

Process resilience *Voters* look at the outputs of the various redundant processes and determine by majority which is the correct output that will then be forwarded to the other processes. Majority is essential in nowadays algorithm for consensus

How much redundancy Where k it's the number of processes that fail

- In crash failure, where the node do not produce any value, $k + 1$
- With bad value you need at least $2k + 1$ processes
- In consensus problems without malicious nodes $3k + 1$

Consensus Each process proposes a single value and all non-faulty processes should all agree on the same value. Achieving consensus among geographically dispersed processes is a fundamental challenge in distributed computing. One well-known instance of this challenge is the *transaction commit problem.*

Byzantine problem One *commander* process proposes a value v, all non-faulty processes must agree on a value. The problem exists if the commander is not faulty

FLP Theorem Theoretical impossibility, if I am in an asynchronous system consensus is impossible. When delays in the response of messages are arbitrary there is no possible solution for the Byzantine agreement, the only solution is the assumption of partially synchronized systems

CAP Theorem No distributed system is immune to network failures, so network partitioning generally needs to be tolerated. In the presence of a partition, there are two options: *consistency* or *availability.* Choosing consistency means the system will return an error or timeout if certain information cannot be guaranteed to be up-to-date. Choosing availability means the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up-to-date. In asynchronous systems only two of these are achievable

- *Consistency*, every read receives the most recent write. Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in ACID properties
- *Availability*, system operating 100% of the time
- *Partition tolerance*, system tolerates an arbitrary number of lost messages

Google Spanner Infrastructure for managing replication. It requires replication at geographical scale, they need Paxos algorithm to maintain different replicas. First time where Paxos was used in large scale distribution.

Different dimensions for each protocol

1. *Synchrony*, async is more realistic. I don't know how much time my message spend to go to you, the clocks of different processes are not synchronized
2. *Failure model*, crash or byzantine
3. *Processing strategy*, pessimistic or optimistic. Two different approaches to ensure data consistency. The pessimistic approach, also known as the locking-based approach, is based on the idea of preventing conflicts between concurrent operations by using exclusive locks. In the optimistic approach, operations are executed without requesting preemptive locks. During the commit phase, conflicts between concurrent operations are checked, and if necessary, conflict resolution policies are applied.
4. *Participant awareness*, known or unknown
5. *Complexity metrics*

Paxos protocol Asynchronous leader-based protocools. Is a family of protocols for solving consensus in a network of unreliable or fallible processors ensuring consistency. Paxos works in three phases

1. *Prepare*, to get yourself elected as a leader. Proposer, who want to be the leader, contacts sending message to all the acceptors and asks them if they will consider its value/ballot number. Once a quorum, moves onto the second phase
2. *Accept*, if a quorum of nodes accepts this value then the value is chosen
3. *Commit*, commit the chosen value to all nodes in the cluster. Learners announce the outcome

The basic version does not cover Byzantine failure, but there's Byzantine Paxos, and tolerates the failure of k nodes with $n = 2k + 1$

Fast paxos Reduce one phase of communication assuming that the leader is elected

Raft protocol Simpler and more understandable implementation of Paxos. Equivalent to Paxos in term of fault-tolerance and performance but integrates the consensus part with the management of the log[1]. Unlike Byzantine fault-tolerant algorithms, Raft operates under the assumption that nodes trust the elected leader

A node can be in 1 of 3 states: *Follower*; *Candidate*; *Leader*. Consensus is achieved through the election of a leader. The leader is responsible for replicating

[1] `http://thesecretlivesofdata.com/raft/`

the log to the followers and regularly sending heartbeat messages to maintain its leadership. If a follower doesn't receive a heartbeat within a timeout period, it transitions to a candidate state and initiates a leader election.

- *Leader election*, all nodes initially start in the follower state. If a follower doesn't receive any communication from a leader, it transitions to the candidate state. As a candidate, it requests votes from other nodes. Each node responds with its vote. If the candidate receives votes from a majority of nodes, it becomes the leader.

  Raft uses a randomized *election timeout* (typically between 150ms and 300ms) to minimize the chance of split votes and ensure a timely resolution. This approach helps prevent multiple servers from becoming candidates simultaneously, allowing a single server to become the leader, send heartbeats, and establish its leadership before other followers can become candidates.

- *Log Replication*, every change made by a client is recorded as an entry in the leader's log. The leader replicates it to the follower nodes. Once the leader receives confirmation from a majority of followers, the entry is considered committed. Once committed on the leader node, the leader informs the followers that the entry is committed using the AppendEntries message, which is also used for heartbeats.

  If the leader crashes, the logs can become inconsistent. The new leader resolves this by comparing its log with each follower's log, finding the last matching entry, and replacing the follower's log with its own entries from that point onward.

In Raft, the *State Machine Safety rule* is maintained by ensuring that a candidate can only win an election if its log contains all the committed entries. When a candidate requests a vote from a voter, it includes information about its own log. If the voter's log is more up-to-date than the candidate's log, the voter will not grant its vote to the candidate. This mechanism guarantees the safety of the state machine by ensuring that only candidates with the most up-to-date log can become leaders.

8 Pervasive system

The concept of transparency of distributed systems here is extreme, even the element that makes calculation and communication, the node, in fact disappears.

Switch from mobile nodes to *sensors* that have computational capabilities and are connected to Internet

Distributed pervasive systems An extension of distributed systems with the following characteristics

- *Unconventional nodes*, nodes that are not explicitly designed for computing but are equipped to perform computing and communication functions
- *Adaptivity*, considers the current context

Mobile computing introduces new problems: limited resources; variance in connectivity; variable position; different types of interfaces

Smart device A device to be defined as smart must have the following characteristics: connected to Internet; run algorithms, locally or remotely, to interpret data and understand context; offer personalized services based on context

8.1 Data acquisition

Transducers Devices that transform one form of energy into another. Sensor captures a physical phenomenon and translates it into electricity, the actuators the opposite

Sensors - Input transducers Acquire measurements from the physical world

- *Physicists*, motion that measure acceleration and rotation along three axes; environmental; position sensors
- *Virtual*, services that provide context data to remote clients

Actuators - Output transducers Make actions that affect the physical world. They are usually powered

Base stations Collect data from distributed sensing and actuating devices, may act as a gateway between networks

Data management Saving battery is the biggest problem in this context, I may need to make *continuous queries*, that is, queries that continue over time to retrieve information

8.1.1 Basic methods

Batch processing Buffering the data acquired by the sensor on the device and then sending it to the base station or remote server. No sensor-side computation but high communication costs

Sampling Do not use all sensor data

Overlapping sliding windows *Time windows* are defined and real-time computation sensor side is made on these. The *overlap* has the task of capturing the phenomena that occur at the edge of the time windows

8.1.2 Advanced methods

Improve data quality and save energy

In-network query processing Build an overlay network in the form of a reverse tree, by aggregating the data into intermediate nodes of the tree. In this way I reduce the amount of data transmitted to the base station

Duty-cycling The radio that transmits the data remains in *sleep mode* for most of the time, if you have a given "abnormal" the radio *wake up* and sends the data. Coordinated sleep/wakeup scheduling algorithm

Model-based/data-driven approaches The data present a strong spatio-temporal correlation. The goal is to reduce the number of samples, read or sent, while maintaining good data quality

1. *Data cleaning*, eliminate outliers from the data. This step is done by the base station and not by the sensor. If a data doesn't follow this function, it's an outlier, I don't send it. The most likely sensor readings are compared with a statistical model, and anomalies are detected by comparing the raw sensor data with the model values.
2. *Data acquisition*, only send data that doesn't follow that pattern, that is, that surpasses a certain δ. Example: temperature
3. *Query processing*, processing queries by accessing or generating minimal amount of data

4. *Data compression*, eliminate redundancy memorizing the extremes of a range of value. Example: Poor man's compression-MidRange, used to reduce the amount of data I have to send

9 Context awareness

Traditional software applications cannot understand the context of a request, so users must make explicit requests with various parameters

Context *A series of circumstances or facts surrounding a particular event or situation.* In our case the event is a request for a mobile service

Temporal context Context history, enables deriving new context, predicting new context

Adaptiveness Captures context data to automatically adapt behavior. Very important because there can be: changes in network connectivity; battery changes; changes in the environments

- *Adapt functionality*, change data flow, hide or expose functionality
- *Adapt data*, more or less accurate/quality

Obtaining context

- *Low level context*, directly acquired by sensors or explicit preferences indicated by the user in their profiles
- *High level context*, obtained through inference methods on low-level context information. What the user is doing or the user's *mood* can be deduced from the activity that the user follows in the day. Knowing what the user is doing allows you to adapt the interface.

Context representation The same context information could be used by multiple applications and even shared. A formal representation of the information is necessary to process it automatically

- *Flat models*, no historical data, no reasoning
- *DB-Based models*, formal models such as ER or Context Modelling Language are used that allow a more effective reasoning on data
- *Ontological models*, formal specification of a shared conceptualization

10 Distributed Ledger Technology and blockchain

It's a distributed log. An example of DLT is Blockchain, it was born in 2008 with the paper *Bitcoin: a peer to peer electronic cash system*, Satoshi Nakamoto.

The most important feature of this technology is the introduction of a historical distributed, immutable/append-only, fault-tolerant and Byzantine fault-tolerant register

DLT All the previous system basing on assumption that processes would fail by crashing. In the Blockchain world we're interested in malicious failures because you are storing data on untrusted components. Each node also contains the same copy of data, which must therefore be synchronized.

- *Permissionless/public DL*, any node in the network can participate in validating transactions and creating new blocks, without needing specific authorization or permissions. There is no central authority
- *Permissioned/private DL*, only a restricted group of authorized nodes have permission to validate transactions and create new blocks. Is controlled and managed by a central authority

Problem of consensus Nodes must find a consensus on the history of data in the blockchain. This consensus is given by the blocks and their order

Blockchain Blockchain is a permissionless ledger of transactions. To maintain quality and decentralization without using a coordinator, a system of digital signatures is employed. Transactions are digitally signed with the sender's private key and broadcasted to all nodes in the blockchain. When a node receives a transaction, it is considered valid. Validated transactions, however, are still regarded as pending as they are not yet part of the chain.

Structure of the blockchain Transactions in blockchains are grouped into blocks with a timestamp and the entire data history stored in each block through a *Merkel tree*. Inside each block there is also a *nonce* and a *prevhash*, reference to its predecessor

Double spending Some transactions may be contradicting each other, in this case consensus is required

Consensus The challenge is therefore to have for the nodes a consensus on the blocks and their sequence, each node must have the same copy of the chain

- Calculate the hash of each transaction and a block

- Compute the hash of a block including the hash of the prevhash in the chain
- Include a trick to make the computation of the block hash expensive but very easy to verify
- The difficulty of the problem is increased

Miner's problem Find a number to be assigned to the field as well as the total block so that the hash of the entire block is smaller than a certain number or has a particularly property

Proof Of Work The algorithm used for consensus.

Within the same blockchain there could be several different chains between peers: to solve it, a majority vote is made, choosing the chain shared by multiple peers. It's the heart of the blockchain, if there wasn't a consensus, everyone would have a different idea of the transactions. Branches that are not part of the prevailing chain are no longer developed

- The fastest growing chain becomes the longest and most reliable
- If a malicious node wants to change a transaction of an intermediate block in the chain, it must recalculate the hash of the block itself and all its successors and prevail over the rest of the nodes in the network
- The blockchain is considered safe until at least 50% of the nodes are not malicious

Smart contract Generalization of the use of blockchain. Pieces of software that define a contract that works independently that is stored within the blockchain becoming immutable

11 Privacy

11.1 Privacy in mobile apps and services

Linking identity with sensitive data

Location Based Service Re-identification can also be based on frequent patterns of locations. Example: My LBS requests (for car sharing for example) originate from a specialised hospital every Tuesday.

In a social network context co-location may become private information. Protection of location and absence privacy becomes trickier

11.2 Privacy protection regulation and techniques

GDPR: EU General Data Protection Regulation

- *Privacy by default*, default privacy settings should be the most protective
- *Privacy by design*, new processes must be designed with data protection in mind
- *Right to be forgotten*, individuals have the right of correcting or deleting their personal data
- *Pseudonymization*, it separates sensitive data from the data respondents keeping the mapping between them accessible only to selected authorised entities
- *Data minimization*, acquire personal data only at the precision strictly necessary for the service

Security principles

- *Confidentiality*, only authorized parties can access data
- *Integrity*, data should not be altered without authorization
- *Availability*
- *Transparency*
- *Unlinkability*, privacy-relevant data cannot be linked across domains. K-anonymity
- *Intervenability*, data rectification or access

k-anonymity Each released record cannot be associated to less than k possible respondents

Location k-anonymity The principle of k-anonymity requires that the individual must be indistinguishable from $k-1$ other potential issuers of the LBS request.

Fake location Simultaneous requests from $n+1$ locations (one of them being the real one) and discard n results

Spatial cloaking Spatial cloaking is an anonymization technique used in the context of k-anonymity. It involves enlarging the location of the requesting user to include other users' positions

The geographic position of the user needs to be enlarged to a cloaking region that includes $k-1$ other users before sending the request to the LBS. This region can be computed by a trusted anonymization service that knows many user positions, such as a mobile operator. Therefore, even in the worst-case scenario where an untrusted LBS provider can identify all k users in the reported area, they can only determine that one of them searched for something, and there is only a chance of $\frac{1}{k}$ that this user was x.

Differential privacy A technique for data privacy protection used to safeguard the privacy of individuals participating in a data collection or analysis process. It involves adding statistical noise to the original data in order to mask personal information

12 Laboratory

12.1 Multi-threaded Servers and Thread Synchronization

Client-Server Paradigm A client asks a specific service to a server

- Server creates a *listening socket* specifying the service port
- Client creates a socket specifying the server's address and the service port
- When an incoming connection is received, the server creates an *established socket* and communication takes place on it

Iterative Servers Can handle one request at time. The requests are queued and the server will sequentially accept and handle each request

Threads Different execution sequences within the same process sharing the same memory space. To create a new thread in Java

- *Thread inheriting*, extend the `Thread` class and override the `run()` method with the thread's code. Then we create an instance of the custom Thread class and call its `start()` method to begin execution of the thread in a new thread context.

```
1         //creation
2         public class MyThread extends Thread{
3             public void run()
4         }
5
6         //invocation
7         MyThread thread = new MyThread()
8         thread.start()
```

- *Implementing Runnable*, implement the `Runnable` interface, define a constructor for the thread, and implement the `run()` method. To start the thread, we create a Thread object and pass the instance of our Runnable class to its constructor. Then, we call the start() method on the Thread object.

```
1         //creation
2         public class RunnableThread implements Runnable{
3             public void run()
4         }
5
6         //invocation
7         Thread thread = new Thread(new RunnableThread())
8         thread.start()
```

The main difference between extending the Thread class and implementing the Runnable interface in thread creation lies in inheritance and flexibility. Extending the Thread class allows direct inheritance of all Thread functionalities, while implementing the Runnable interface requires creating a separate instance of the Thread class and passing the Runnable instance to its constructor. Implementing the Runnable interface provides more flexibility as it allows the child class to extend other classes if needed and promotes better separation of concerns.

After the `run()` method finishes, the thread also terminates its execution.

Concurrent Servers: dispatcher-worker model Different clients can concurrently send different requests to the port the server is listening on. It runs a different thread for each connection with a client

Concurrent programming The threads of a program have access to the same memory space, allowing for efficient data exchange. However, this can lead to problems such as *thread interference*[2] and *memory inconsistency*[3]. To prevent these issues, thread synchronization is necessary.

Thread Synchronization - Synchronized Every object instance in Java has an associated *intrinsic lock* that is also called *monitor*. If a method is declared as `synchronized`, before it is executed, the method acquires the intrinsic lock. This ensures that only one thread can access the object at a time.

When a thread executes a synchronized method on an object, the following happens

- The value of the intrinsic lock associated with the object is checked
- If is available, the value is changed to not available and the method is executed
- Once the method finishes executing, the value of the intrinsic lock is changed back to available
- If the lock is not available, the thread *waits* until the lock becomes available

To synchronize primitive types, we can create dummy objects that are used only for their lock. Also a static method can be defined synchronized, which means that the method is locked not on a specific object instance, but on the class itself. This means that only one thread can execute the synchronized static method for a given class at any given time.

[2]Thread interference occurs when two or more threads simultaneously access the same shared variable and try to modify it concurrently

[3]Memory inconsistency occurs when threads access the same variables without synchronization and without taking care of the access order. Therefore, a thread may see a non-updated value of a shared variable, which can lead to logical errors.

The use of synchronized grants two properties:

1. *Mutual exclusion*, only one thread at a time can obtain a specific lock
2. *Visibility*, the changes applied to the shared data before the lock is released must be visible to the threads that will acquire the lock later

Deadlock Reciprocal waiting. Possible solution: wait with a timeout parameter

Volatile This keyword can be assigned to variables and it grants visibility but not mutual exclusion, making it a *lighter version of synchronized*. The volatile keyword ensures that reads and writes to a volatile variable are done directly on the main memory, bypassing the thread's local cache. This guarantees that changes made to the volatile variable by one thread are immediately visible to other threads.

12.2 Machine to Machine Communication

In a distributed system, processes that runs on different machines connected through a network have to communicate. The communication occurs through messages exchange, messages are exchanged through sockets

It is necessary to agree on a common format to represent data

Marshalling and unmarshalling When data is transmitted, it needs to be assembled in a specific format that is suitable for transmission. This process is called *marshalling*, and it is performed by the party transmitting the data. On the other hand, the party receiving the data performs the *unmarshalling* process, which involves translating the received data into a readable format. For this process to work seamlessly, both parties need to agree on a common external data format that ensures interoperability. Three commonly used formats for this purpose are

- *XML: eXtensible Markup Language*, is a standard for representing data in a textual format, offering the advantage of creating customized subsets (dialect) of the language to accurately describe specific information. Its powerful query system based on XPath and XQuery allows extracting only desired information from large structured data sets.
- *JSON: Javascript Object Notation.* Is a lightweight data-interchange format. Newer designed specifically for Javascript, but independently of it. This standard is both human-and-machine-readable and easier to read than XML.
- *Protocol Buffers*

Gson library A Google library used to convert Java objects into JSON representations and viceversa. It works with any type of pre-existing Java object, even objects we don't have the code of

- `toJson()` method to convert an object to a JSON string
- `fromJson()` method to convert a JSON string to an instance of an object

12.2.1 Google's protocol buffer

Google has suggested a way to encode structured information in a faster and more efficient manner than with conventional XML or JSON formats. This involves converting the data into binary form, which allows for smaller file sizes and faster processing times.

How to use protocol buffer

1. Define the format of the messages to be serialized throught a `.proto` file, these files must be shared among the programs that have to communicate
2. Build Gradle to run the Protocol Buffers compiler that automatically generates the marshalling and unmarshalling code
3. Then, it's possible to write code to send and receive messages

The language To define a message in our model, we need to use the `message` keyword. Every message is a record that contains different fields and it can contain other messages. Each part of the message needs to be labeled as either

- `required`, mandatory
- `optional`, meaning it may or may not be present
- `repeated`, allowing for a variable number of occurrences

Tags To identify the fields in the binary format and match them during serialization and deserialization. Tags are represented by integer numbers, with tags from 1 to 15 being encoded through 1 byte and tags from 16 to 2047 requiring 2 bytes. It's important to note that tags must not be changed once a message is being used within a system.

```
1  //protocol buffer example
2  syntax = "proto3";
3
4  message Actor {
5  string name = 1;
6  int32 age = 2;
```

```
7           string gender = 3;
8           repeated Movie movies = 4;
9           }
10
11          message Movie {
12          string title = 1;
13          int32 year = 2;
14          repeated string genres = 3;
15          }
```

```
1           //object building
2           Actor actor = Actor.newBuilder()
3               .setName("Tom Hanks")
4               .setAge(64)
5               .setGender("Male")
6               .addMovies(
7                   Movie.newBuilder()
8                       .setTitle("Forrest Gump")
9                       .setYear(1994)
10                      .addGenres("Drama")
11                      .build(),
12                  Movie.newBuilder()
13                      .setTitle("Cast Away")
14                      .setYear(2000)
15                      .addGenres("Drama")
16                      .addGenres("Adventure")
17                      .build()
18              )
19          .build();
```

12.3 RPC: Remote Procedure call

The Remote Procedure Call paradigm allows clients to call remote procedures as if they were local procedures, without having to manage the details of network communication.

When a client requests a remote procedure, a local *stub* is created that represents the remote procedure. This stub hides the complexity of network communication and allows the client to call the remote procedure as if it were a local function. Once the stub is called, the remote procedure's parameters are transferred to the server over a network connection. The server processes the parameters and returns the results to the client over the same network connection.

gRPC Framework RPC system developed and widely used by Google. Services defined through Protocol Buffers.

On the server side, we need to create the interface and set up a gRPC server to handle requests from clients. Meanwhile, on the client side, we only need to use the stub methods provided by the server to make requests.

Server side, for each service defined in the proto file, it is necessary to implement its logic. Client side it is only required to specify the service address and call the stub methods

Multiplexing Communication channel based on HTTP2, it leverages multiplexing: different requests and responses can be asynchronously received with a single TCP connection. Mopre advanced than REST that is based on only HTTP verbs

Synchronous vs Asynchronous RPC methods can be used in both synchronous and asynchronous ways. Synchronous calls are blocking, which means the client waits for a response from the server before continuing. Asynchronous calls don't block the thread, allowing the client to continue processing while waiting for a response from the server. gRPC supports both synchronous and asynchronous stubs.

RPC Types

1. ***Unary***, single request single response

```
rpc Greeter (HelloRequest) returns (HelloResponse) {}
```

2. ***Server Streaming***, single request stream of responses

```
rpc LotsOfReplies (HelloRequest) returns (stream HelloResponse) {}
```

3. ***Client Streaming***, stream of requests single response

```
rpc LotsOfGreetings (stream HelloRequest) returns (HelloResponse) {}
```

4. ***Bidirectional streaming***, streams are independent, client and server can write in any order

```
rpc BidiHello (stream HelloRequest) returns (stream HelloResponse) {}
```

StreamObserver class Is used for asynchronous communication over a stream, to receive asynchronous notifications from a stream of messages. It provides three methods to manage stream communication

- `onNext(V value)`, receive a value from the stream
- `onError(Throwable t)`, receives an exception generated by the stream
- `onCompleted()`, receives a notification that the stream communication has ended successfully

12.4 REST: Representational State Transfer

Software architectural style that defines a set of rules to follow for creating web services. These HTTP methods provide a uniform interface that can be used to communicate with web services using different programming languages and systems. REST architecture relies over four HTTP methods, each resource is identified by a *URI*: Uniform Resource Identifier and in each operation the client transmits or receives a representation of a resource

CRUD → HTTP
Create → POST
Read → GET
Update → PUT
Delete → DELETE

Is stateless, each client request is complete and independent and the HTTP header and body contain all the parameters necessary for the server to perform the requested operation. The problem with stateful approach is that the server responds to the client's requests must always be the same because it has to store the client's state. This fact limits the system scalability, load-balancing, and failover.

Jersey Open-source library for web service REST development. Enables to map HTTP requests to Java methods

```
1    @Path("/hello")
2    public class HelloWorld {
3    @GET
4    @Produces(MediaType.TEXT_PLAIN)
5    public String sayHello() {
6        return "Hello world!";
7    }
8    }
```

Annotations are used to define the interaction between Jersey and our code. Each class manages all actions for a specific level of the URI tree. Each URI consists of two parts

- *prefix*, defined at the configuration level
- *suffix*, that changes according to the resources on which we want to apply the actions

JAXB Automatic *marshalling* and *unmarshalling* of JavaBeans objects in JSON or XML. Integrated in Jersey.

Use of annotations for: define the XML root (`@XMLRootElement`); Specify variable types if different from those used in Java; The names to be assigned to the elements, if they differ from those specified in the code

`@Produces` allows specifying the format returned by a method

`@Consume` allows specifying in which format the data is expected from the client

12.5 MQTT: Message Queue Telemetry Transport

Machine-to-machine communication protocol, designed for devices with resource constraints (e.g., IoT devices).

Publish-subscribe pattern MQTT is based on the Publish-subscriber pattern. They never contact each other directly, a broker filters all incoming messages from publishers and distributes them to subscribers.

Asynchronicity Processes are not blocked while waiting for a message or publishing a message

MQTT client and broker MQTT client may be any thing connected to the Internet (from microcontrollers to a massive server). Both publisher and subscriber are MQTT clients.

MQTT broker is responsible for managing various aspects such as user authentication, connection handling, session maintenance, and subscriptions. The primary function of the broker is to receive messages from publishers and then transmit them to the subscribers.

MQTT topics MQTT messages are published to *topics*. Topics consist of one or more topic levels, separated by a forward slash

$$iot/sensor/temperature$$

Topics are a great way to organise the data flows through the network. We could use MQTT and use topics to subdivide the data, in this way the clients can subscribe only to the topics they are interested in. Suppose that we are dealing with several sensors deployed accross multiple sites, we could use MQTT and use topics to subdivide the data

$$site1/position \ ; \ site1/temp$$

$$site2/position \ ; \ site2/temp$$

QOS: Quality Of Service It is an agreement between the sender and receiver regarding the guarantee of message delivery. There are three levels of QoS:

- QOS 0: At most once. It means that the message can be lost or delivered to the recipient only once without any guarantees of further delivery attempts.
- QOS 1: At least once. It means that the message will be delivered to the recipient at least once, but it may be duplicated during delivery in case of missing acknowledgment (PUBACK).
- QOS 2: Exactly once. It means that the message will be delivered to the recipient exactly once, without any duplications or losses.

Persistent session If the connection between the client and the broker is interrupted, the topics to which the client subscribed will be lost. To avoid this problem, the client can request a persistent session when connecting to the broker (using the `cleanSession` flag). Persistent sessions save all relevant information for the client on the broker: subscription topics of the client; all (new) messages in QOS1 or 2 that the client has not yet confirmed.

Retained message A retained message is a regular MQTT message with the retained flag set to true. Each client that subscribes to a topic pattern matching the topic of the retained message receives the retained message immediately after subscribing.

MQTTCallback It enables a MqttClient to work asynchronously. Optional for publishing, mandatory for subscribing

- `messageArrived(String topic, MqttMessage message)`
- `connectionLost(Throwable cause)`
- `deliveryComplete(IMqttDeliveryToken token)`

Last will and testament If the broker detects an unexpected client disconnection, it sends the Last Will message to all subscribed clients of the corresponding topic