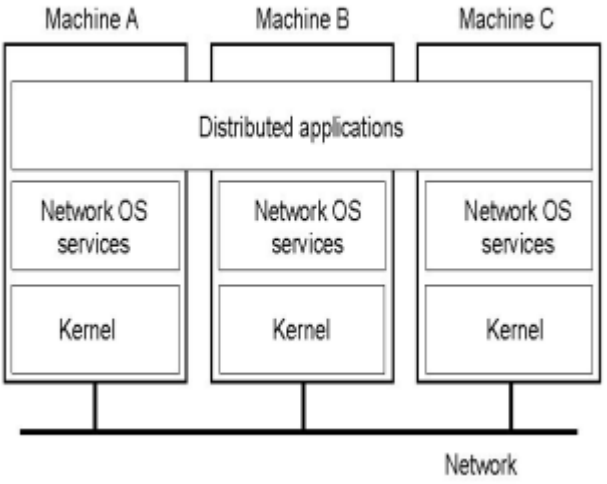
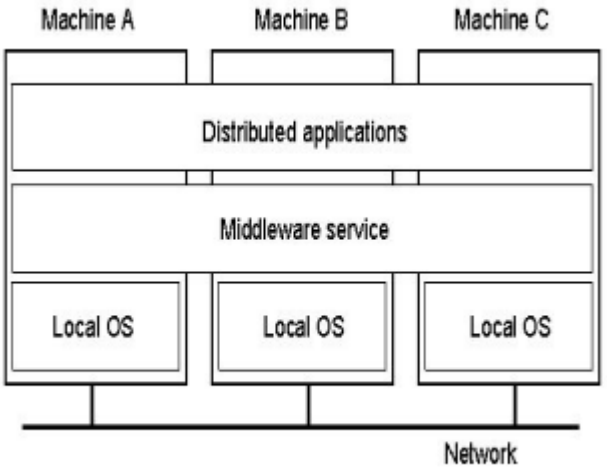
02.Modeling

The software architecture of a distributed system



It can be Network OS based: the network OS provides the communication services, different machines may have different Network OSes, masking platform differences is up to the application
programmer. You have to link all the operational services, the networking services.



Middleware based: the middleware provides advanced communication, coordination and administration services. It mask most of the platform differences.
Nowadays we use middleware in every system using it at a different degree.

Middleware: A functional view

Middleware provides “business-unaware” services through a standard API, which raises the level of the communication activities of applications.
Usually it provides:

- Communication and coordination services
  - Synchronous and asynchronous
  - Point-to-point(TCP/IP) or multicast(UDP multicast)
  - Masking differences in the network OS
- Special application services
  - Distributed transaction management, groupware and workflow services, messaging services, notification services, ...
- Management services
  - Naming, secutiry, failure handling, ...

The run-time (system) architecture of a distributed system

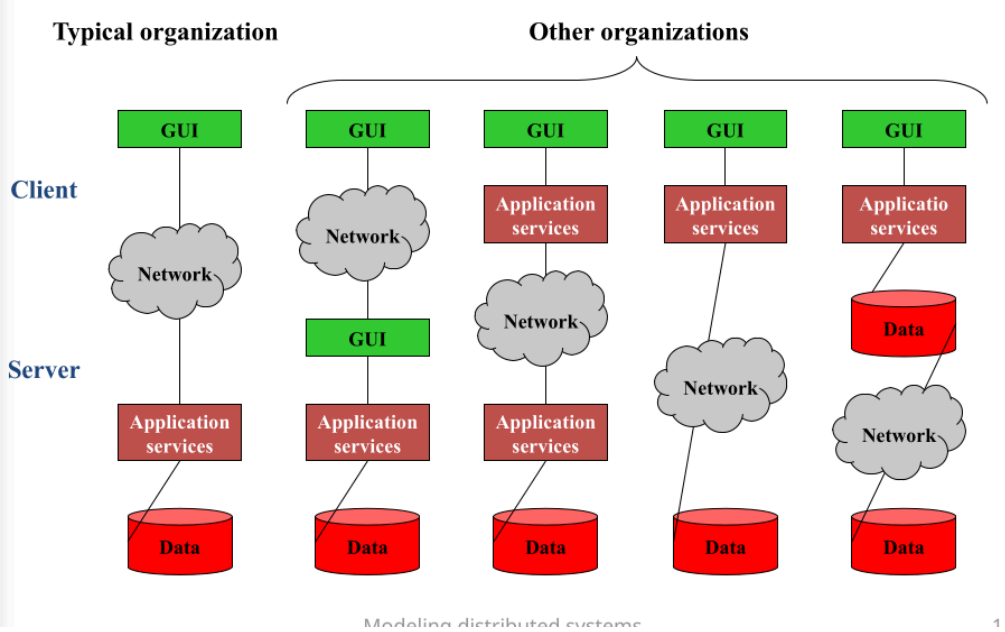
It describe how the computation is splited on the different machines. The idea is the difference is in the easyness of the distribution in the application.
Identifies the classes of components that build the system, the various types of connectors, and the data types exchanged at run-time
Modern distributed systems often adopt one among a small set of well known architectural styles

- Client-server
- Service Oriented
- REST
- Peer-to-peer
- Object-oriented
- Data-centered
- Event-based
- Mobile code
  These models differentiate the different applications and there are difference also among them but they have the common base between all the class.

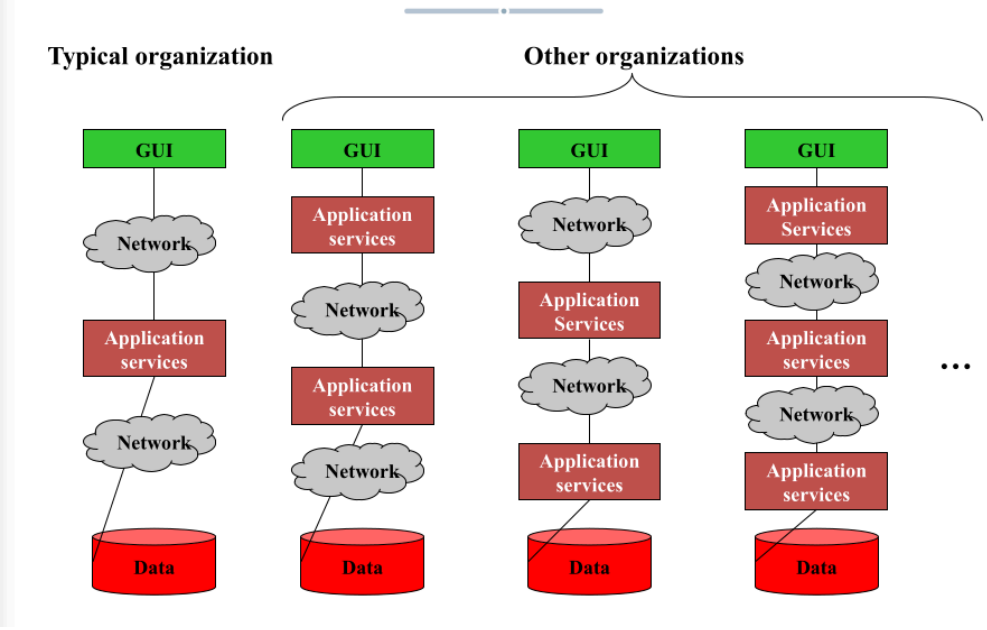
Client-Server

Components have different roles:

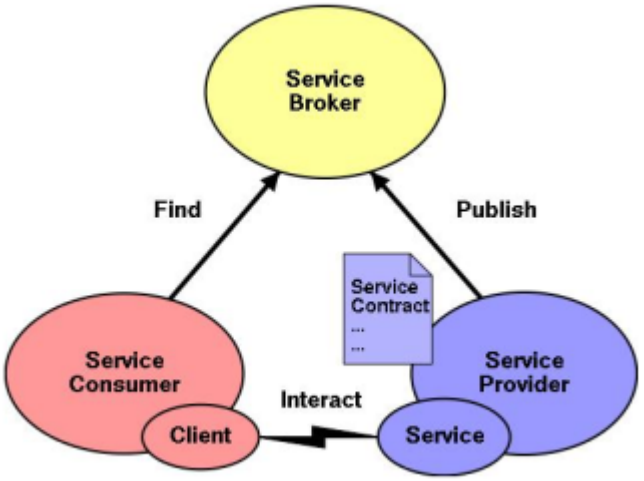
- Server: provide a set of service through a well defined API(They are passive)
- Users access those services through clients
  Communication is message based (or RPC)
  Tiers
  Often servers operate by taking advantage of the services offered by other distributed components
  In such case we have a three-tiered client-server architecture
  The services offered by a distributed application can be partitioned in three classes: User interface services, application services, storage services
  Multi-tiered client-server applications can be classified looking at the way such services are assigned to the different tiers
  Two Tiered Architecture
  A machine run the client and a machine run the server. The application layer are suddivided between the two machines depending on the needs of the application



Three Tiered Architecture
There are multiple machine where the layers of the application are run.



The Service Oriented Architecture



Special case of client-server architecture. In general the distinction between them is not crispy, diverse people can use diverse architecture also in base of the level of abstraction.
It looks very similar to client-server architecture as there is a component that act as a server and another one is acting as a client. The difference is that the interaction are based on service invocation. A service is a set of operations that a component offer to another(server) and one that invoke the services(client). The service is described in a service contract. The service broker is the middleman between the service provider and the service consumer. Single service broker for multiple service provider/consumer.

Web Services

Web Service: “a software system designed to support interoperable machine-to-machine interaction over a network” [W3C]
Its interface is described WSDL (Web Service Description Language)
Web service operations are invoked through SOAP, a protocol, based on XML, which defines the way messages (operation calls) are actually exchanged. Usually based on HTTP but other transport protocols can be used
It includes the set of operations exported by the web service UDDI (Universal Description Discovery & Integration) describes the rules that allows web services to be exported and searched through a registry

The REST style

Can be seen as a posteriori description of the web.
REpresentational State Transfer (REST) is both:

- A (nice) way to describe the web
- A set of principles that define how Web standards are supposed to be used
  Key goals of REST include:
- Scalability of component interactions
- Generality of interfaces
- Independent deployment of components
- Intermediary components to reduce latency, enforce security and encapsulate legacy systems

Main Constraints

- Interactions are client-server
- Interactions are stateless: state must be transferred from clients to servers. The same request can give different results as it is not saved. We can see as the state is maintained from the client, the interaction must be stateless. This is also done to permit a client to connect to different server during a sequence of interactions as the servers can be indistinguishable and act the same way.
- The data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable
- Each component cannot “see” beyond the immediate layer with which they are interacting
  - REST is layered
- Clients must support code-on-demand(optional constraint), ability of the client to execute javascript nowadays

- Components expose a uniform interface.
  Replication, caching become really simple reducing costs and increasing performance.

REST: Uniform Interface Constraints

The uniform interface exposed by components must satisfy four constraints:

- Identification of resources: each resource must have an id (usually an URI) and everything that have an id is a valid resource (including a service)
- Manipulation of resources through representations: REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types (e.g., XML), selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw resource, or is derived from the resource, remains hidden behind the interface. A representation consists of data and metadata describing the data
- Self-descriptive messages: control data defines the purpose of a message between components, such as the action being requested or the meaning of a response. It is also used to parameterize requests and override the default behaviour of some connecting elements (e.g., the cache behaviour)
- Hypermedia as the engine of application state: clients move from a state to another each time process a new representation, usually linked to other representation through hypermedia links

Peer to Peer

In a peer-to-peer applications all components play the same role, there is no distinction between clients and servers(at least at logical level).
Developed to resolve the problem that the server didn't take responsibility for something.
There could be a server that manages the connection between peers but all the communication is done between peers. There were also the opportunity that the network was becoming really good and the bandwidth was increasing, the computer at the edge were becoming more potent.
Why p2p:

- Client-server does not scale well, due to the centralization of service provision and management
- The server is also a single point of failure

- P2P leverages off the increased availability of broadband connectivity and processing power at the end-host to overcome such limitations
  P2P promotes the sharing of resources and services through direct exchange between peers
  Resources can be:
- Processing cycles (SETI@home)
- Collaborative work (ICQ, Skype, Waste)
- Storage space (Freenet)
- Network bandwidth (ad hoc networking, internet)
- Data (most of the rest)
  Fundamental difference is that it takes advantage of resources at the edges of the network.
  The changes are that end host resources have increased dramatically and broadband connectivity is now common.

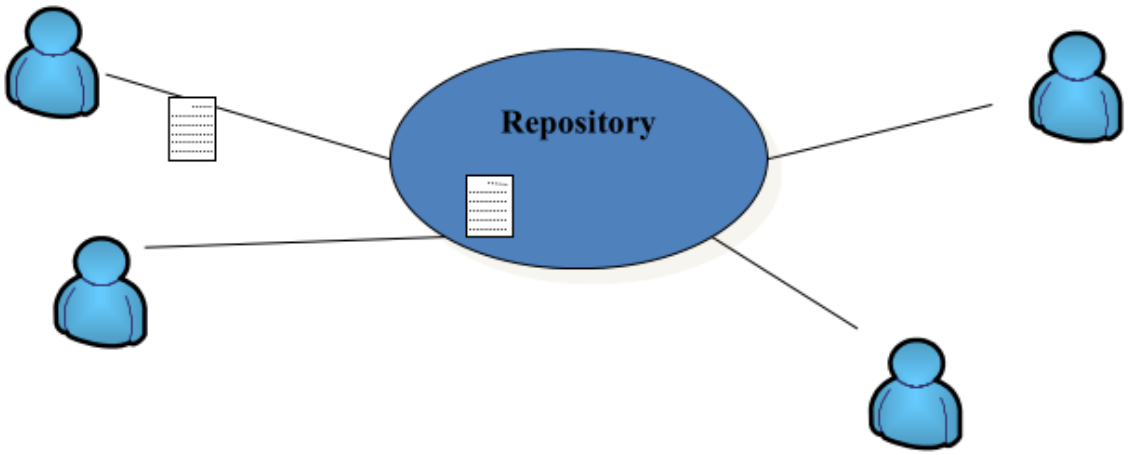
Object Oriented

The distributed components encapsulate a data structure providing an API to access and modify it
Each component is responsible for ensuring the integrity of the data structure it encapsulates
The internal organization of such data structure is hidden to the other components (who may access it only through the API mentioned above)
Components interact through RPC
Its a “peer to peer” model
But its often used to implement client-server applications
Pros

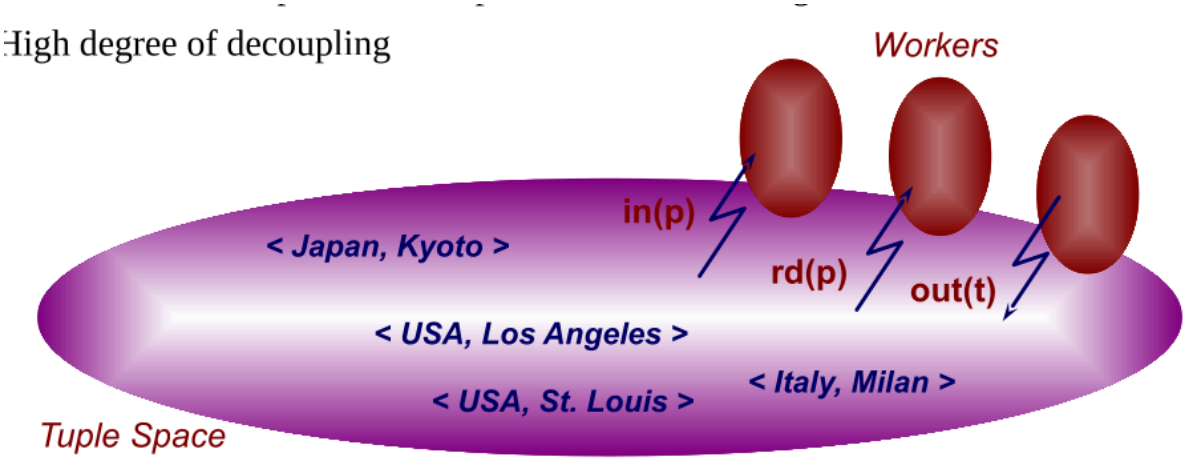
- Information hiding hides complexity in accessing/managing the shared data
- Encapsulation plus information hiding reduce the management complexity(E.g., the objects that build the server may be moved at run-time to share the load)
- Objects are easy to reuse among different applications
- Legacy components can be wrapped within objects and easily integrated in new applications
  The objects are on multiple machines, they need to invoke different objects on different machines.
  Each component is not only a peer but is an object, need a description

language to describe the method offered by the objects as they can be written in different programming languages.

Data Centered



A bunch of components distributed in the network. All the interaction is mediated by something that is logically centralized, no direct interaction by the components.
This idea started from Linda and tuple spaces, a single synchronous tuple space.
You may write or read a pattern from the tuple space.



The advantage is that the interaction is not directed and so it is less chained together.

Linda in a nutshell

Data is contained in ordered sequences of typed fields (tuples)
Tuples are stored in a persistent, global shared space (tuple space)
Standard operations:

- out(t): writes the tuple t in the tuple space

- rd(p): returns a copy of a tuple matching the pattern (or template) p, if it exists; blocks waiting for a matching tuple otherwise. If many matching tuples exist, one is chosen non-deterministically
- in(p): like rd(p), but withdraws the matching tuple from the tuple space
- Some implementations provide also an eval(a), which inserts the tuple generated by the execution of a process a
  Many variants:
- Asynchronous, non-blocking primitives (probes): rdp(p) and inp(p), return immediately a null value if the matching tuple is not found
- Bulk primitives: e.g., rdg(p)
- …
  Some of the non-standard primitives have non-trivial distributed implementations
- E.g., if atomicity is to be preserved, probes require a distributed transaction

Architectural Issues

The tuple space model is not easily scaled on a wide-area network

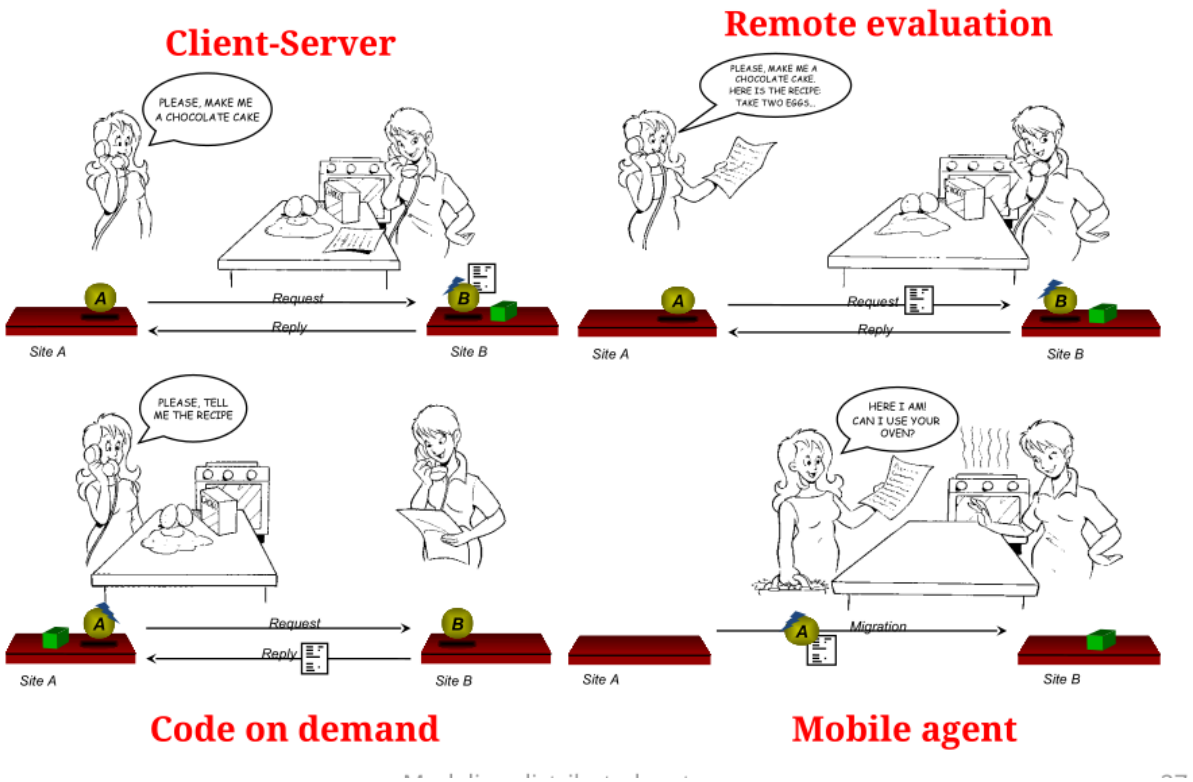
- How to store/replicate tuples efficiently
- How to route queries efficiently
  The model is only proactive
- Processes explicitly request a tuple query, reactive/asynchronous behavior must implemented with an extra process and a blocking operation
  As a consequence, commercial implementations:
- Provide only client access to a server holding the tuple space. Instead of a fully distributed, decentralized implementation
- Introduce reactive primitives(e.g., notify allows to register a listener, invoked when a matching tuple is written)

Event Based

Allow components to interacts by publishing or subscribing events. Subscriptions are permanent if not manually removed. If someone publish an event it is automatically sent to every user that is subscribed to it. The interaction is multicast, mediated by the transport layer that totally mediated by the event bus. The communication is implicit as the centre decide how to do it and is anonymous as you only subscribe to topics. Called event based because it was initially used to transport events.

Mobile Code

A style based on the ability of relocating the components of a distributed application at run-time(Only the code or both the code and the state)
In general the code of the application remain fixed. The idea is to migrate code at runtime.
Three mobile code paradigm+client-server for standard:



- Remote Evaluation: you evaluate the code remotely and ask someone doing it giving them the code(Ex. Google COlab, AWS, printing on Postscript)
- Code on Demand: I request some code and execute locally(Ex. REST, javascript execution)
- Mobile Agent: the process(code+state of execution) move between the users in the middle of execution

Mobile code technologies

Strong mobility is the ability of a system to allow migration of both the code and the execution state of an executing unit to a different computational environment. Very few systems (usually research based) provide it
Weak mobility is the ability of a system to allow code movement across different computational
environments. Provided by several mainstream systems including Java, .Net, the Web

Mobile code in practice

Pros:

- The ability to move pieces of code (or entire components) at run-time provides a great flexibility to programmers
  - New versions of a component can be uploaded at run-time without stopping the application
  - Existing components can be enriched with new functionalities
  - New services can be easily added
  - Existing services can be adapted to the client needs

  Cons:
- Securing mobile code applications is a mess

The interaction model

Distributed algorithms

Traditional programs can be described in terms of the algorithm they implement. Steps are strictly sequential and (usually) process execution speed influence performance, only
The distributed systems have multiple processes and in particular you need the processes to interact only through message busses. A distributed algorithm is composed of two things: the definition of the steps taken by each process and the transmission of messages between them
The communication is fundamental to have a distributed system. The behaviour is influenced by the rate at which process proceeds, the performance of the communications channels and the difference clock drift rates.

Interaction Model

To formally analyze the behaviour of a distributed system we must distinguish (at least in principle) between:

- Synchronous distributed systems:
  - The time to execute each step of a process has known lower and upper bounds
  - Each message transmitted over a channel is received within a known bounded time

    - Each process has a local clock whose drift rate from real time has a known bound
- Asynchronous distributed systems:
  - There are no bounds for process execution speeds, message transmission delays, clock drift rates
  Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one (but the vice versa is clearly false).
  We mostly assume that systems are synchronous

The Failure Model

Both processes and communication channels may fail. The failure model defines the ways in which failure may occur to provide a better understanding of the effects of failures
We distinguish between:

- Omission failures
    - Processes: fail stop (other processes may detect certainly the failure) vs. Crash
    - Channels: send omission, channel omission, receive omission(Drop of packages)
- Byzantine (or arbitrary) failures
    - Processes: may omit intended processing steps or add more(different execution of code)
    - Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once()
- Timing failures (apply to synchronous systems, only)
  - Occur when one of the time limits defined for the system is violated
  Processes most of the time have omission failures. Channels typically at physical level do byzantine failures, that also for the protocol(checksum on the messages) that permit to transform byzantine failures in omission failure easily.
  FLP Theorem demostrated that in an asynchronous system there isn't a way for agreement.
  It matters for committing or aborting a transaction in a distributed database(e.g. withdraw at the ATM). Agree on values of replicated, distributed sensors. Agree on whether a system component is faulty. We solve this in practice changing the

assumptions(e.g. make links reliable enough), reduce the guarantees(e.g. only probabilistic instead of deterministic)