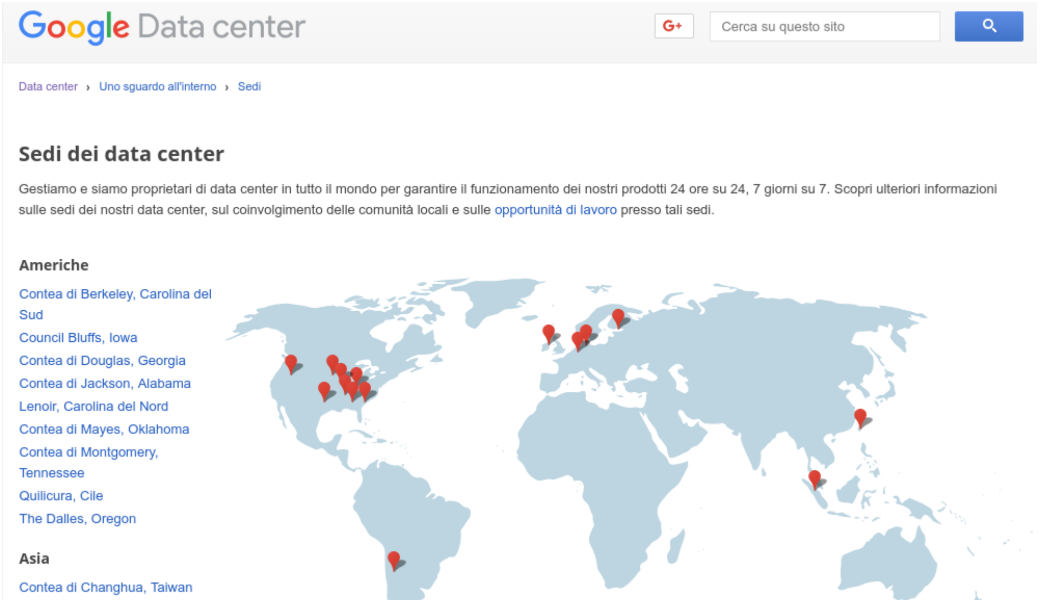
Introduction to Distributed System

Agenda

- Examples of distributed systems
- A definition of distributed system
- Design goals
- Some challenges

Reference: Chapter 1 of “Distributed Systems by M. van Steen and A. S. Tanenbaum”

https://www.google.com/about/datacenters/



Google Data center

Data center › Uno sguardo all'interno › Sedi

Sedi dei data center

Gestiamo e siamo proprietari di data center in tutto il mondo per garantire il funzionamento dei nostri prodotti 24 ore su 24, 7 giorni su 7. Scopri ulteriori informazioni sulle sedi dei nostri data center, sul coinvolgimento delle comunità locali e sulle opportunità di lavoro presso tali sedi.

Americhe

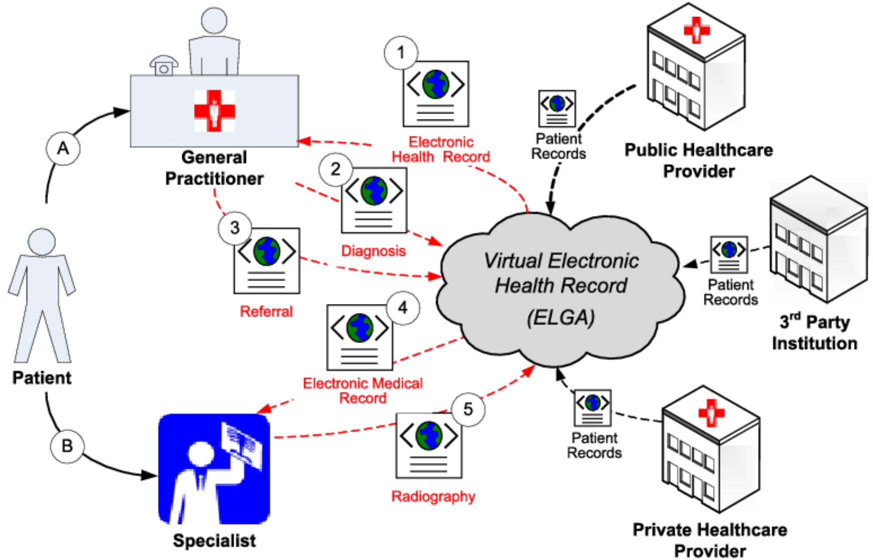
- Contea di Berkeley, Carolina del Sud
- Council Bluffs, Iowa
- Contea di Douglas, Georgia
- Contea di Jackson, Alabama
- Lenoir, Carolina del Nord
- Contea di Mayes, Oklahoma
- Contea di Montgomery, Tennessee
- Quilicura, Cile
- The Dalles, Oregon

Asia

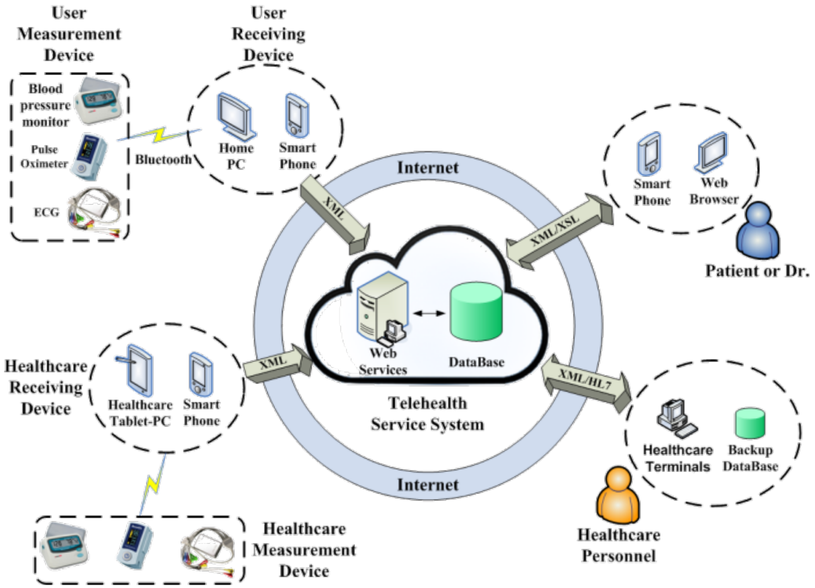
- Contea di Changhua, Taiwan

Almost Every Service We Use On Internet

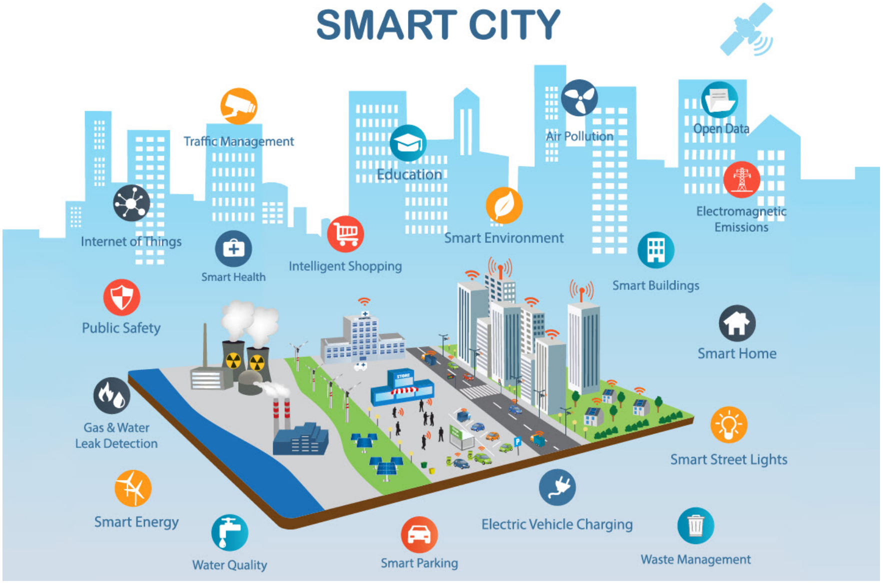
Health-care System



Telemedicine

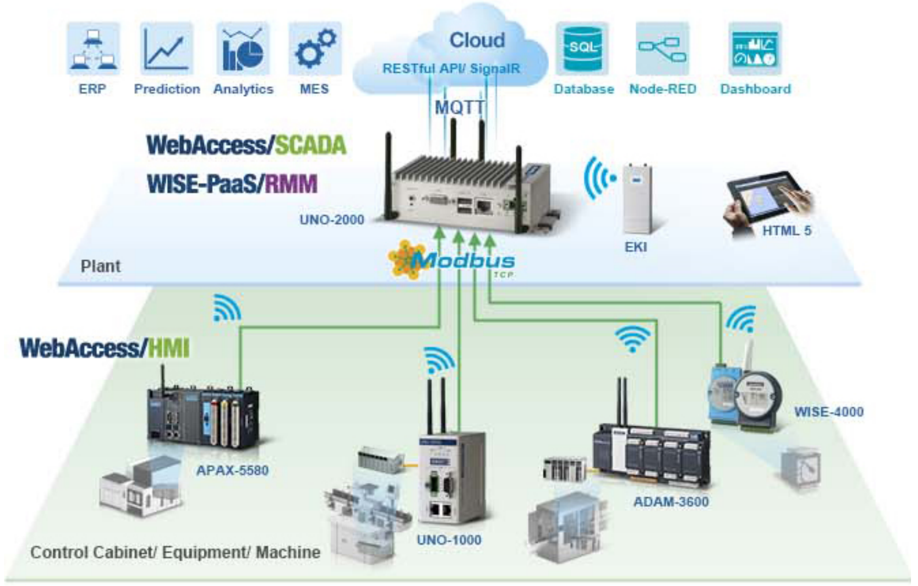


Smart City

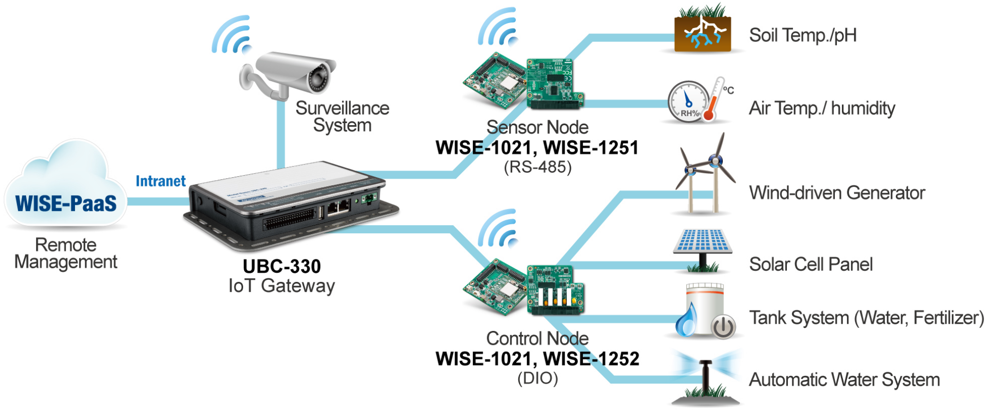


SMART CITY
Traffic Management
Education
Air Pollution
Open Data
Internet of Things
Smart Health
Intelligent Shopping
Smart Environment
Electromagnetic Emissions
Smart Buildings
Public Safety
Smart Home
Gas & Water Leak Detection
Smart Street Lights
Smart Energy
Electric Vehicle Charging
Water Quality
Smart Parking
Waste Management

Smart Factory



Smart Agriculture



What is a Distributed System?

"A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system."

(M. van Steen & A. S. Tanenbaum)

```
Received: by jumbo.dec.com (5.54.3/4.7.34)
        id AA09105; Thu, 28 May 87 12:23:29 PDT
Date: Thu, 28 May 87 12:23:29 PDT
From: lamport (Leslie Lamport)
Message-Id: <8705281923.AA09105@jumbo.dec.com>
To: src-t
Subject: distribution
```

There has been considerable debate over the years about what constitutes a distributed system. It would appear that the following definition has been adopted at SRC:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

The current electrical problem in the machine room is not the culprit--it just highlights a situation that has been getting progressively worse. It seems that each new version of the nub makes my FF more dependent upon programs that run elsewhere.

What Are Autonomic Computing Elements?

What are?

Software processes or hardware devices (often called nodes)

Key features

- Nodes are programmed to achieve common goals
- They interact by message passing: usually a node receives a message and reacts sending a new one
- Each node is autonomous with its notion of time (no global clock)
  - Synchronization and coordination issues!

Characteristic 1: Collection of Computing Elements

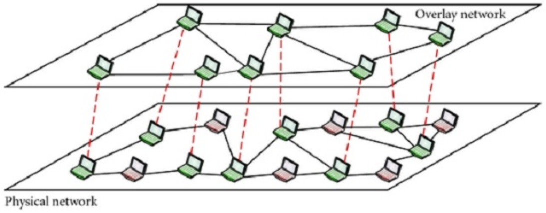
Nodes are typically organized in an Overlay Network (a graph)

- Each node in the system can communicate only with its neighbors
- The set of neighbors of a component can be static or dynamic

Membership in the overlay network

- Open group: any node can join the system and communicate with other nodes in the system
- Closed group: a mechanism is required to a node to enter the system
  - We need mechanisms to manage group membership
  - We need mechanisms that ensure a node is communicating with authorized nodes

Overlay Network $\neq$ Physical Network



- An overlay network is built upon another network (e.g. the physical one)
- An edge in an overlay network may correspond a path in the underlying network
- A overlay network is usually connected

Two kinds of overlay networks:

- Structured overlay network: a tree, ring, hypercube
- Unstructured overlay network: random network

Characteristic 2: Single Coherent System

When the user interacts with the system, the collection of nodes as a whole operates the same and it is hidden that processes and data may be spread across the network

What does it mean? Some examples

- A user cannot tell where a computation is taking place
- The location of data should be not relevant for an application
- It is irrelevant for an application is data have been replicated

Characteristic 2: Single Coherent System

When the user interacts with the system, the collection of nodes as a whole operates the same and it is hidden that processes and data may be spread across the network

Very hard to achieve in real life

- Systems are complex and made of many components
- Some components may fail at any time (partial failures)
- Sometimes recovery is possible but it is very hard to hide

For this reason we need some tradeo-off when we build a system: in we need to cope with partial failures and partial consistent states

Design goals of a distributed system

- When building a distributed system there are some design goals we aim at
- These goals characterize how good our systems
- Unfortunately, many of them are very difficult to achieve in practice, thus, we need trade-offs

What we want to achieve?

1. Sharing resources
2. Distribution transparency
3. Openness
4. Scalability
5. Availability
6. Modularity

Goal 1: Sharing Resources

Make easy for users to **access remote resources** and to **share them in a controlled and effective way**

Resources: they can be virtually anything

- devices: cameras, temperature sensors, printers, scanners, etc.
- computational power: CPUs, etc.
- files: video, photos, etc.
- data: temperature in a storehouse, the number of parking lot available, etc.

Examples

- BitTorrent: resources are files that users download and shares
- Google Maps: resources mainly are maps and metadata on maps

Goal 2: Distribution Transparency

- Provide users with a uniform interface and hide that processes and resources are physically spread across the network
- A distributed system that presents itself to users as it is a single computer system is said to be **transparent**

Examples

- You do not need to know which Google server you use when access to your mail
- You do not need to know if your mail are stored in a compressed form
- You do not need to know where Dropbox stores your files when you upload them

Different Kinds of Transparency

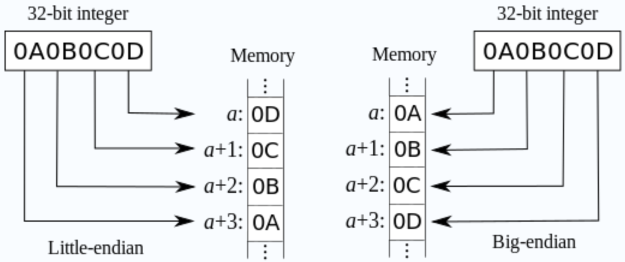
The notion of transparency is applied to different aspect of a distributed system

Kind of Transparency	What is hidden
Access	Differences in how resources are represented and accessed
Location	Where resources are located
Migration	A resource can be moved to another location
Relocation	A resource can be moved to another location when in use
Replication	A resource may be replicated possibly at different location
Concurrency	A resource may be shared by several competitive users
Failure	Failure and recovery of a resource

Access Transparency

It deals with hiding **differences** in data representation and how data can be **accessed** by users. The goal is to agree on how data is represented and accessed independently of machine architecture and OSs.

Example 1: Big-Endian vs Little-Endian



- Intel processors are little-endian
- SPARC processors are big-endian

Access Transparency

It deals with hiding **differences** in data representation and how data can be **accessed** by users. The goal is to agree on how data is represented and accessed independently of machine architecture and OSs.

Example 1: Big Endian vs Little Endian

Big-endian is the most common format in networking: TCP/IP protocols use encode their fields in this format (network byte order)

```
#include <netinet/in.h>


uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Access Transparency

It deals with hiding **differences** in data representation and how data can be **accessed** by users. The goal is to agree on how data is represented and accessed independently of machine architecture and OSs.

Example 2: Different OSs

- A distributed systems may be made of machines running Linux and Windows
- These systems have different **naming conventions** and mechanisms for accessing resources:
  - In Linux everything is a file, e.g., `/dev/sda1`, `/proc/1/cmdline`
  - In Linux the file extension is not important

Location Transparency

- It deals with hiding to users **where** a resources is **physically located** in the system
- How we refer to resources is important to achieve location transparency

The role of naming resources

- Location transparency can be achieve by assigning **logical names** to resources
- The logical name should be **independent** as possible as of the location of the resource, e.g.,

  `https://doi.org/10.1007/s00607-016-0508-7`

  does not give info on where the document is stored. Also, if we move the document on another machine, users can still access it by using its URL.

Question: What we need to implement logical names?

Migration Transparency

- It deals with hiding that a resources can be **physically moved** in another location of the system
- The way the resources is **accessed remains unchanged**

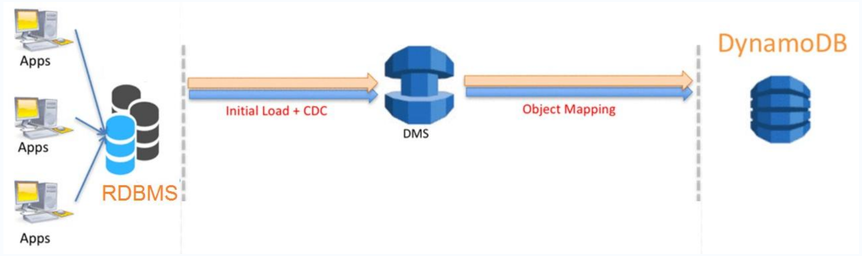
Examples:

- Communication on mobile phones when users are moving
- Tracking of goods while they are being moved from a place to another

Migration Transparency

- It deals with hiding that a resources can be **physically moved** in another location of the system
- The way the resources is **accessed remains unchanged**

Example: from RDBMS to NoSql on the Cloud (Another kind of migration)

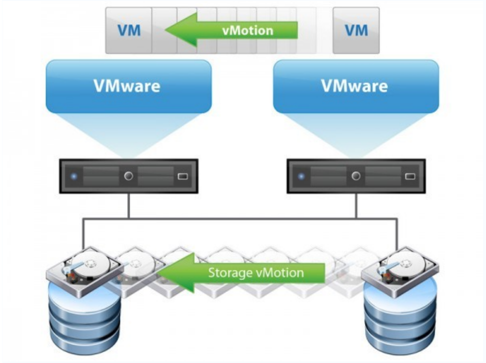


Goal: applications should not notice the change

Relocation Transparency

It deals with hiding that a resources can be **physically moved** in another location of the system when in use

Example: Relocating VMs (VMWare vMotion)



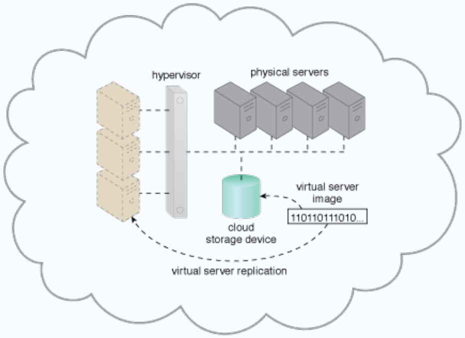
Move a running VM from a physical server to another one

Replication Transparency

- Replication plays an important role in a distributed system
- Resources may be replicated to increase the availability or the performances when resources are put near users
- Replication transparency deals with hiding that there are **several copies** of the same resource
- To achieve replication transparency is required that
  1. a system uses logical names for resources (all copies must have the same name) and
  2. it supports location transparency (Why?)

Replication Transparency

Example: Resource replication in the Cloud



Several instances of a virtual server using a single store image and several physical servers

Concurrency Transparency

- Many users may want to access and use the same resource of the system
- They can cooperates or compete
- Concurrency transparency deals with hiding that **several users** may access the same resource at the same time leaving the resource in a **consistent** state

Examples

- Two user wants to access the same file on a file server
- You want to access your email from the phone and the mail agent on your PC

Note: coordination mechanisms are required

Failure Transparency

It deals with hiding that a resource fails to work properly and that the system recovered from that failure

Examples

- If a mail server fails, users are automatically redirected to another one
- If a disk of a file server fails, users can still access their files with no losses

Note:

- Masking failures is one of the hardest issues of distributed systems
- It is very difficult to determine whether a resource has failed or if is responding slowly

Why Is Distribution Transparency Hard?

In real systems

- Communication and network latencies cannot be hidden
- Nodes may fail
- Determine if a resource failed or it is slow is hard
- Performances matter: taking all data replicas updated may be costly, e.g., when they are located on different continents
- Concurrency issues: mutual exclusion, synchronization

In practice, we are happy when we achieve a good level of distribution transparency

However, there are situations where we you want to give up some degree of distribution transparency (context/location awareness)

Goal 3: Openness

A system is open when

- It offers components, cooperates and integrates with other systems
- New resources can be dynamically added and made available to users
- It offers resources according to standard protocols

Openness: Key features

To be open a system must provides

- Standard interfaces: the syntax and the semantics of the services each component offers
- Interoperability: two components provided by different manufacturers work together relying only on the interface
- Portability: a component developed for a system A can be executed on a different system B
- Extensibility: it is relatively easy to add new components or replace existing one without affecting the rest of the system

How to define component interfaces

Interface Definition Language (IDL)

A specification language used to describe the **syntax** of services provided by a component

- Names of the available functions
- Types of the parameters
- Exceptions that can be raised

Notes

- The semantics of interfaces is through comments, however if properly specified, an interface allows a process to "talk" with another process providing that interface
- The IDL language is independent of the programming language used for implementing the component

Example of CORBA IDL

```
module Bank
{
  typedef float CashAmount;  // Type for cash
  typedef string AccountId;  // Type for account ids

  interface Account
  {
  readonly attribute AccountId  account_id;
  readonly attribute CashAmount balance;

  void withdraw(in CashAmount amount) raises (InsufficientFunds);

  void deposit(in CashAmount amount);
  }
}
```

Flexibility

- An open system is also a flexible system
- To achieve flexibility is required that the system is organized as a collection of small and easily replaceable components
- Monolithic systems tend to be not flexible enough
- Another way to improve flexibility is to separate policies and mechanisms

Policies vs mechanisms

What are they?

- The policies specifies what is to be done
- The mechanism specifies how it is to be done

The idea

We can adapt the behaviour of a system by only changing the policy and without affecting the mechanisms which implement that policy

Examples

To achieve a better performances it is easier to change

- the priority of threads/processes rather than changing the scheduling algorithm
- the size of a cache or how long the cache store a piece of data, rather than changing how data are stored and accessed

What are policy and what mechanisms?

- Description of the resources that a component may access

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code (mechanism)
- Implement different encryption algorithms

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code (mechanism)
- Implement different encryption algorithms (mechanism)
- Specify which QoS level to use when the bandwidth varies

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code (mechanism)
- Implement different encryption algorithms (mechanism)
- Specify which QoS level to use when the bandwidth varies (policy)
- Specify different level of secrecy for communication

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code (mechanism)
- Implement different encryption algorithms (mechanism)
- Specify which QoS level to use when the bandwidth varies (policy)
- Specify different level of secrecy for communication (policy)
- Provide adjustable QoS parameters per data stream

What are policy and what mechanisms?

- Description of the resources that a component may access (policy)
- Access control list for each resource saying which component may access it (mechanism)
- The operations that a downloaded code may perform (policy)
- Support different level of trust of mobile code (mechanism)
- Implement different encryption algorithms (mechanism)
- Specify which QoS level to use when the bandwidth varies (policy)
- Specify different level of secrecy for communication (policy)
- Provide adjustable QoS parameters per data stream (mechanism)

Advantages of separating policies and mechanisms

The separation of mechanism and policy provides flexibility to a system

- If the interface between mechanism and policy is well defined, the change of policy may affect only a few parameters
- If interface between these two is vague or not well defined, it might involve much deeper change to the system.

Goal 4: Scalability

The idea

The capability of a system to grow and manage an increasing amount of work

Good scalability?

- A single server for all users

Goal 4: Scalability

The idea

The capability of a system to grow and manage an increasing amount of work

Good scalability?

- A single server for all users No
- A centralized routing algorithm

Goal 4: Scalability

The idea

The capability of a system to grow and manage an increasing amount of work

Good scalability?

- A single server for all users No
- A centralized routing algorithm No
- A system with a web server in Europe and a database in Australia

Goal 4: Scalability

The idea

The capability of a system to grow and manage an increasing amount of work

Good scalability?

- A single server for all users No
- A centralized routing algorithm No
- A system with a web server in Europe and a database in Australia No

Note: Usually, scalability is considered by developers the most important design goal, but do not forget the other ones

Scalability Dimensions

Scalability can be measured in various dimensions

- Size scalability: numbers of users and/or nodes/processors/resources
- Administrative scalability: numbers of organization or administrative domains sharing a single distributed system
- Load scalability: ability to expand and contract resource pools to accommodate heavier or lighter workloads
- Functional scalability: ability to support new functionality at minimal efforts
- Geographic scalability: ability to maintain performance and usability regardless of the distance between nodes

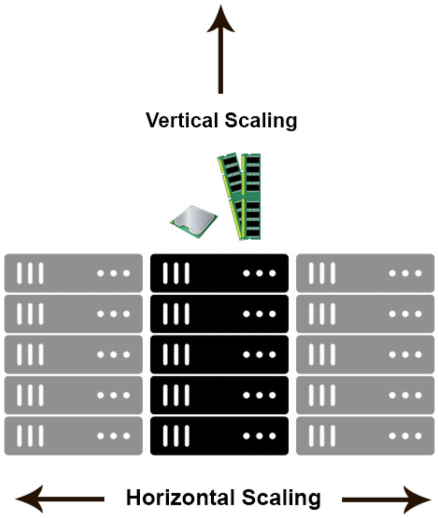
Size scalability

The ability of a system to work effectively when the numbers of requests or resources increase

Examples

- A search engine continues to work effectively even when the number of indexed documents increase
- A routing protocol scales w.r.t. network size, if the size of the routing table grows as $O(\log N)$, where N is the number of nodes in the network
- A peer-to-peer system scales when the number of messages to locate a file is independent of the number of node in the network (see Gnutella vs BitTorrent)

Two Approaches to Size Scalability: Horizontal and Vertical Scaling



- **Horizontal scaling:** adding more nodes to a system
- **Vertical scaling:** adding more resources to existing nodes

Approaches to scaling

- Asynchronous communication
- Move computation to the client
- Decrease the number of messages
- Caching and replication
- Partitioning tasks and resources

Hiding Communication Latency

Avoid waiting for responses to remote services as much as possible

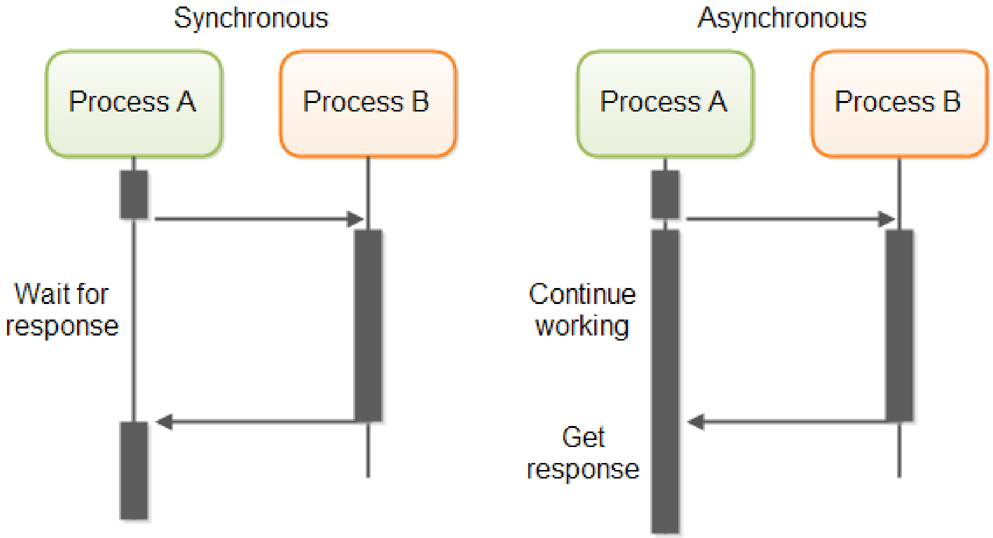
Synchronous communication

The sender blocks until the receiver picks up the message — there is a rendezvous

Asynchronous communication

The sender drops the message and continues its execution

Synchronous Communication vs Asynchronous Communication



An Example of Synchronous Request

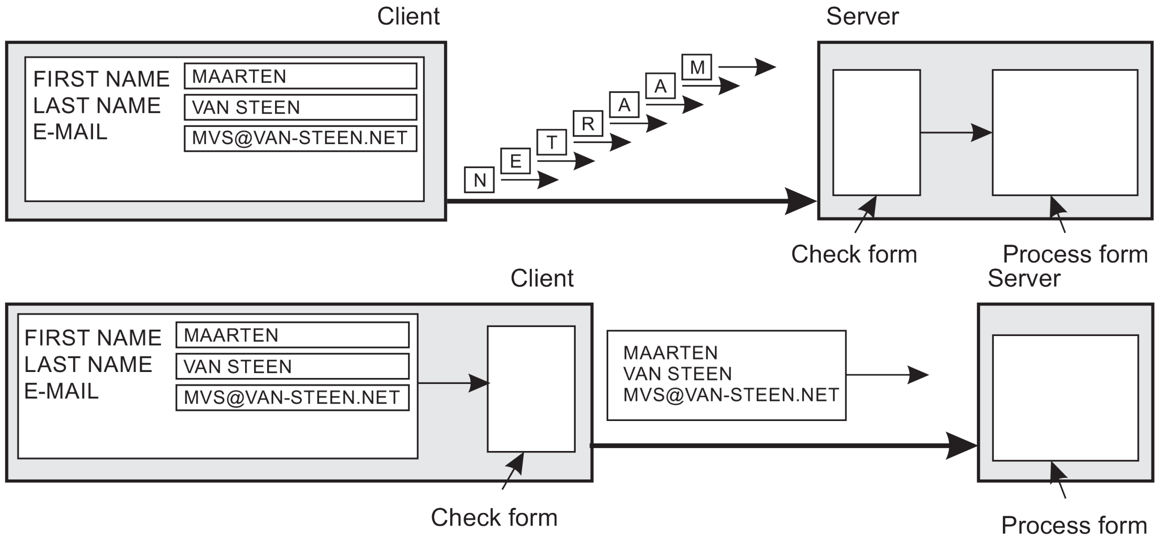
```
import requests

num_requests = 20

for i in range(num_requests):
  response = requests.get('http://example.org/')
  print("Response " + str(i) + ": " + str(response.status_code))
```

Before starting a new request the previous one must be completed

An Example of Asynchronous Request

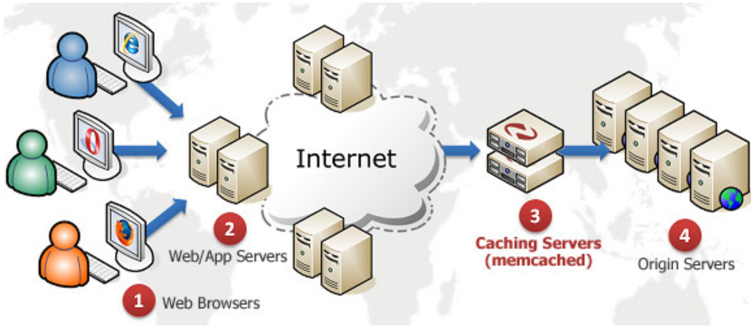
```
import asyncio
import requests

async def main():
    loop = asyncio.get_event_loop()
    futures = [
        loop.run_in_executor(
            None, requests.get, 'http://example.org/'
        )
        for i in range(20)
    ]
    for response in await asyncio.gather(*futures):
        print("Response: " + str(response.status_code))

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Move Computations to the Client



Caching and Replication

Make copies of data available on different components of the system

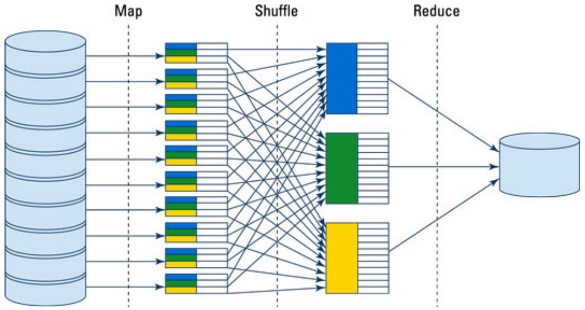


Note: Having multiple copies (cached or replicated) of data requires keeping them **consistent** (synchronization) on each modification

Examples of distributed caching system: memcached, EHCache

An Example of Partitioning Tasks

MapReduce is a framework for processing large datasets using a many nodes



- **map:** applies a computation to local data
- **shuffle:** redistributes data based on the output key
- **reduce:** process each group of output data

Goal: Availability

The intuition

Every request received by a non-failing node in the system must result in a response

Note: There is no bound on how long the system may run before providing an answer

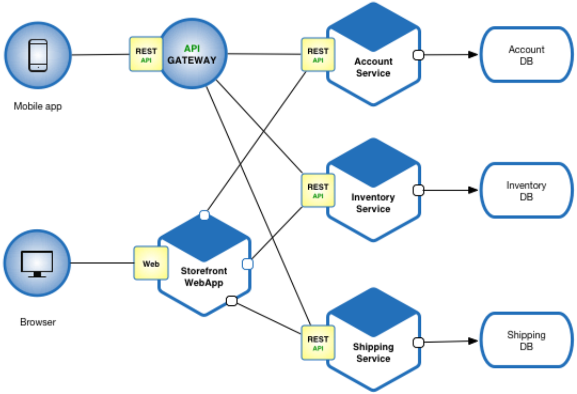
Two implicit assumptions on the system

1. Failure (crash, unavailability, etc.) must be detected as soon as possible
2. Recovery procedures must be initiated as soon as a failure is detected

Goal: Modularity

- Your application is made of many simpler parts (modules)
- Each module has
  - A well defined interface specifying how other modules can interact with it
  - A functional and implementation-independent specification
  - Written in a different programming language and provided by a different vendor

An example, Service Oriented Architecture



Challenges

Achieving a good level of **distributed transparency**, **openness**, **scalability**, **availability**, **modularity** is hard

Typical questions

- What is the right interface or abstraction?
- How to partition the system into modules for scalability and modularity?
- How to detect failures and manage them?
- How do nodes coordinate to achieve their tasks?

Other Challenges

- Security:
  - Component authentication
  - Isolate misbehaving components
  - DoS attacks
  - Information flow
- Implementation challenges:
  - What's the bottleneck?
  - How to reduce the load on bottleneck resources?
- Quality of service:
  - Throughput
  - Responsiveness
  - Load balancing

Misconception/False assumption

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- There is one administrator
- There is shared knowledge

Conclusion

What we have discussed

- Examples of distributed systems
- A definition of distributed system
- Design goals
- Some challenges

Reference: Chapter 1 of "Distributed Systems by M. van Steen and A. S. Tanenbaum"