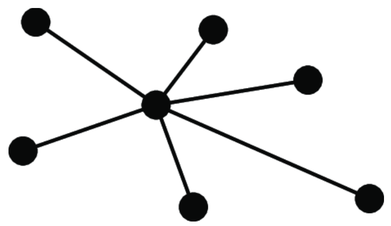
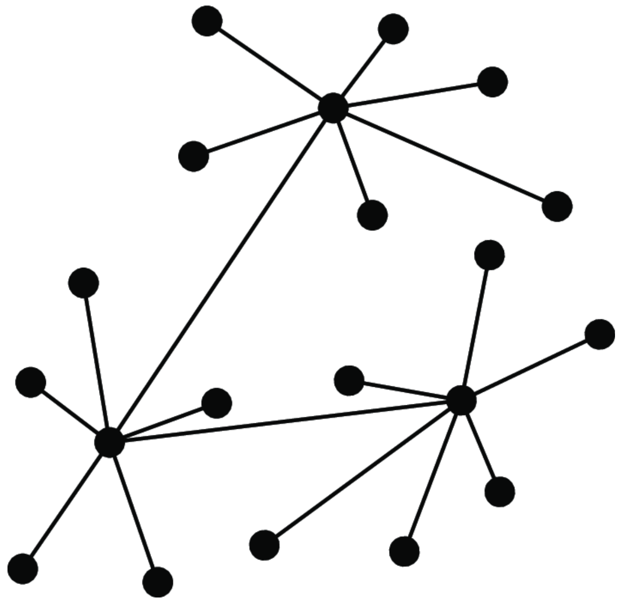
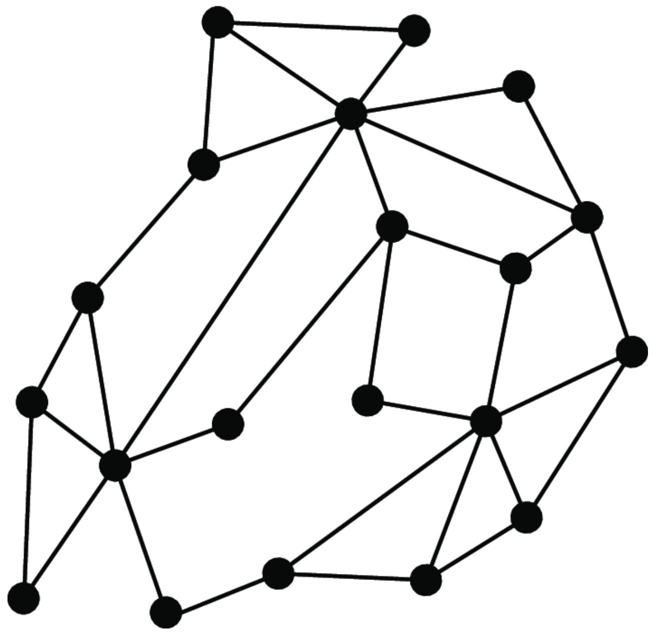
Distributed System

From networked systems to Distributed Systems


centralized


De-centralized


distributed

Distributed System

A possible definition

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

Autonomous computing elements: also referred to as nodes, can be both hardware devices and software processes.

Single coherent system: users or applications perceive a single system

Challenges:

no global clock: each node is autonomous and will thus have its own notion of time. Leads to fundamental synchronization and coordination problems.

security: in a collection of nodes how to manage group membership and authorization ?

How to collect: network options

Overlay network

Each node in the collection communicates only with other nodes in the system, its neighbors. The set of neighbors may be dynamic, or may even be known only implicitly (i.e., requires a lookup).

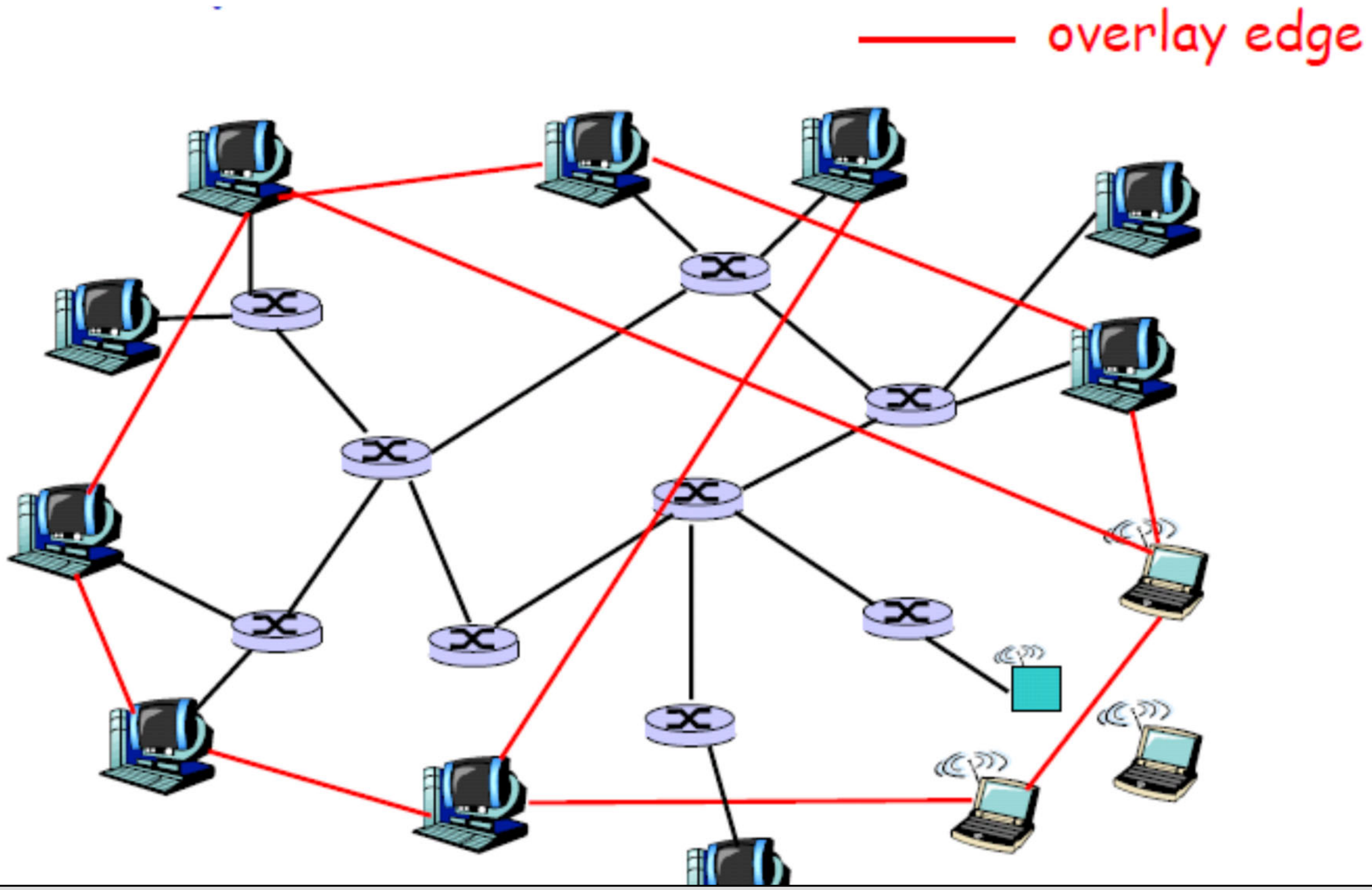
Overlay types

Well-known example of overlay networks: peer-to-peer systems.

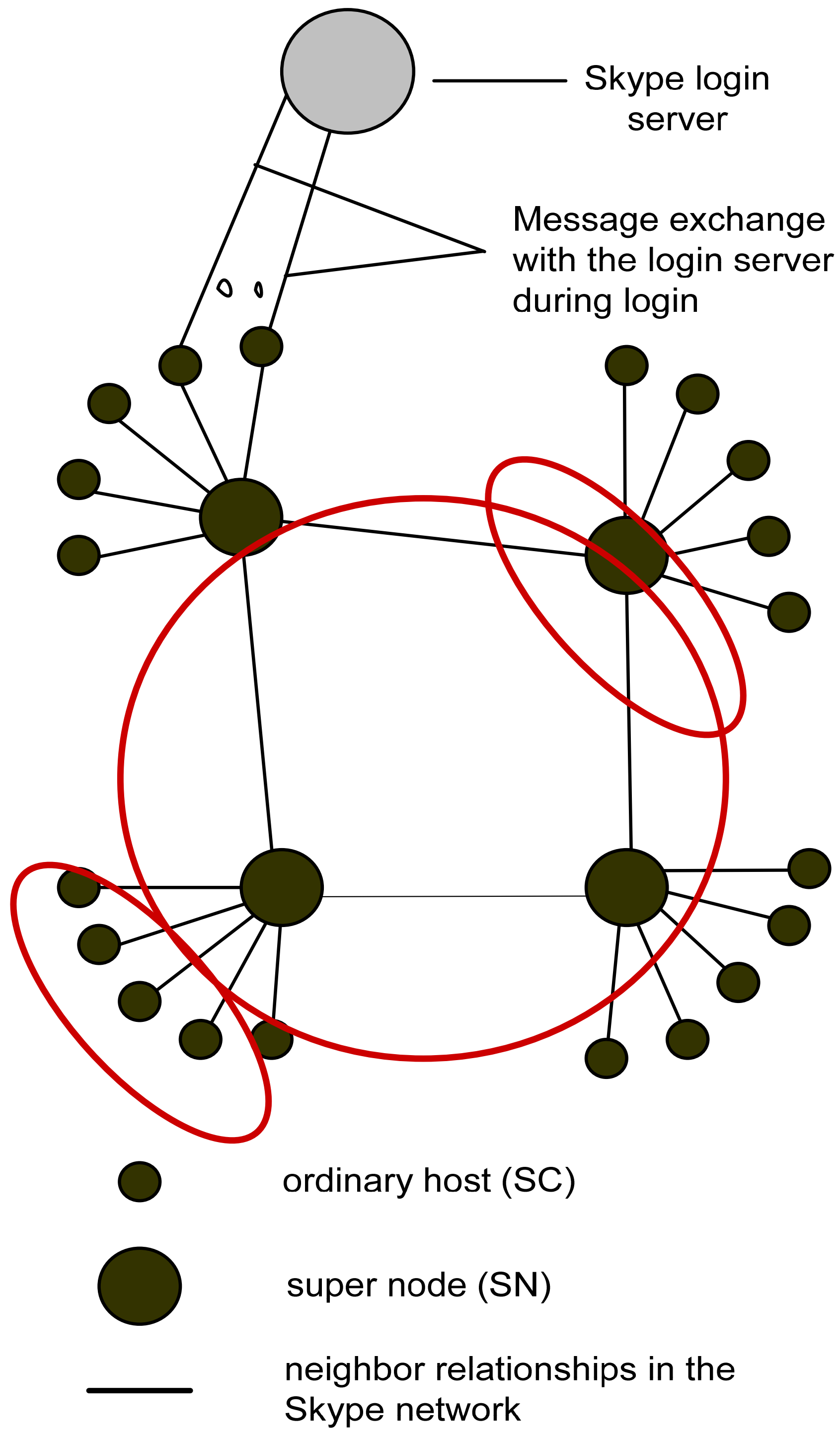
Structured: each node has a well-defined set of neighbors with whom it can communicate (tree, ring).

Unstructured: each node has references to randomly selected other nodes from the system.

Overlay Network



A P2P network is an overlay network. Each link between peers consists of one or more IP links.


Skype login server
Message exchange with the login server during login
ordinary host (SC)
super node (SN)
neighbor relationships in the Skype network

The Skype Network

- Ordinary Skype Client (SC)
  - A client Skype application
- Super Nodes (SN)
  - A client Skype application
  - Has public IP address, ‘sufficient’ bandwidth, CPU and memory
- Login server
  - Stores Skype id’s, passwords, and buddy lists
  - Used at login for authentication

The Centralization of Skype
Matt Rickard
Jun 7, 2022

- Skype was founded in 2003 by Niklas Zennström and Janus Friis. The two founders had previously created Kazaa, the peer-to-peer (p2p) music sharing protocol1. Skype was built on the idea that the cost of voice calls could be reduced by using a similar p2p protocol.
- Skype even references its p2p origins – the initial name for the project was 'Sky peer-to-peer,' which eventually was abbreviated to Skype.
- You can read an early analysis of how Skype p2p worked (An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol (2004)). Maybe the most interesting part of Skype's architecture was how it resolved network address translation (NAT) problems. Two callers behind different firewalls (e.g., on different company internal networks) may not be able to reach each other over the internet directly. Instead, these two callers would connect through a node on the internet that had sufficient bandwidth and processing power outside of both firewalls. See a deeper technical dive (part 1 and part 2).
- As Skype grew, the network started to buckle under the pressure. An hours-long outage in 2010 was related to supernodes running a bad version, then becoming overloading and failing, eventually leading to the entire network ceasing to operate (see a postmortem).
- Microsoft acquired Skype in 2011 for $8.5 billion. Shortly afterward, Microsoft changed the architecture of supernodes – first only allowing its own servers to act as supernodes and then finally deprecating the entire Skype protocol.
- Yet, P2p didn't mean censorship or tracking resistant. Skype censored and restricted its services in China to access that market. And in the U.S., it shared data with the NSA, allowing access to people's video and phone calls. This was all before the acquisition by Microsoft.

Coherent system

Essence

The collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Examples

An end user cannot tell where a computation is taking place
Where data is exactly stored should be irrelevant to an application
If or no data has been replicated is completely hidden

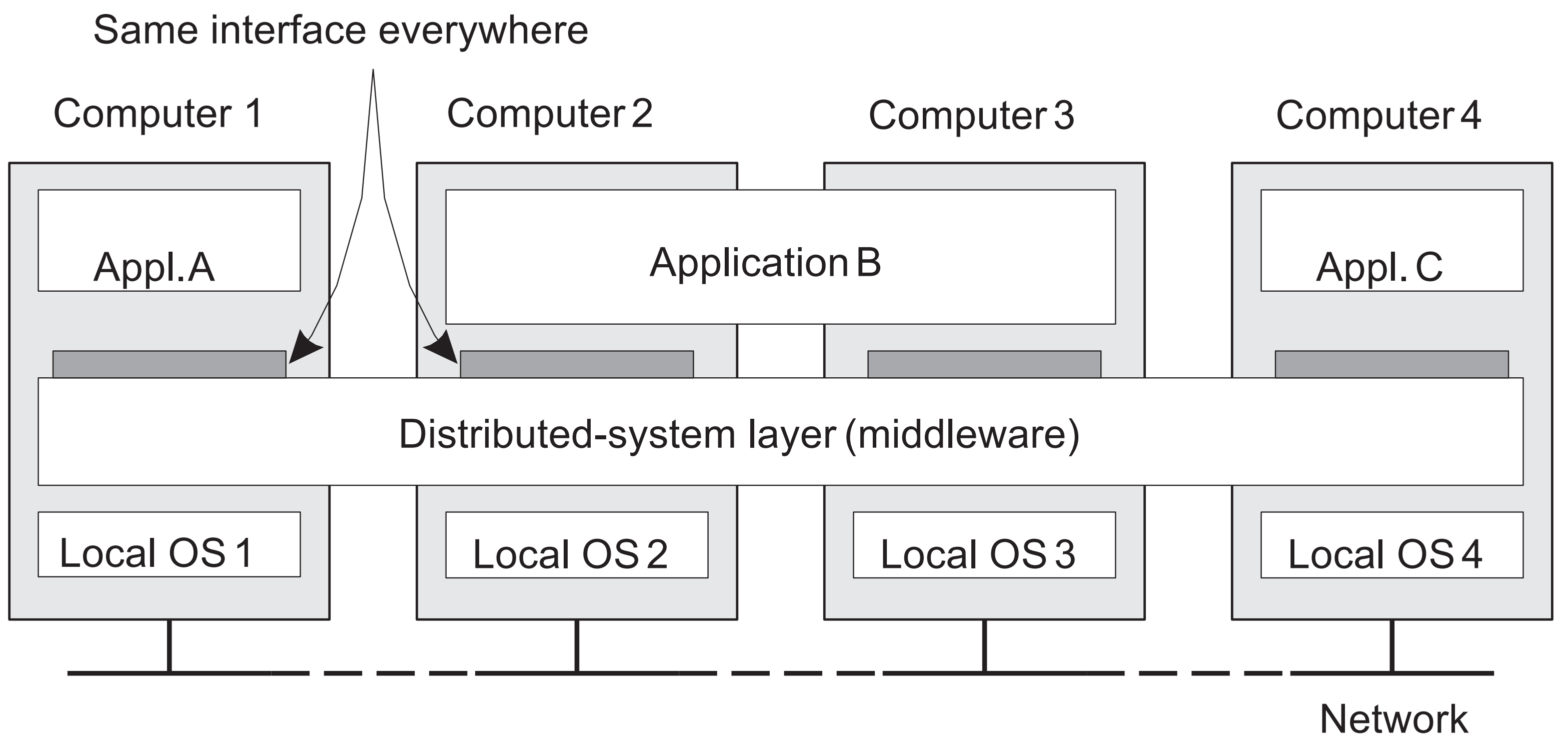
Keyword is distribution transparency

The snag: partial failures

It is inevitable that at any time only a part of the distributed system fails.

Hiding partial failures and their recovery is often very difficult and in general impossible to hide.

Middleware: the OS of distributed systems



What does it contain?

Commonly used components and functions that need not be implemented by applications separately (communication, security, accounting, resiliency).

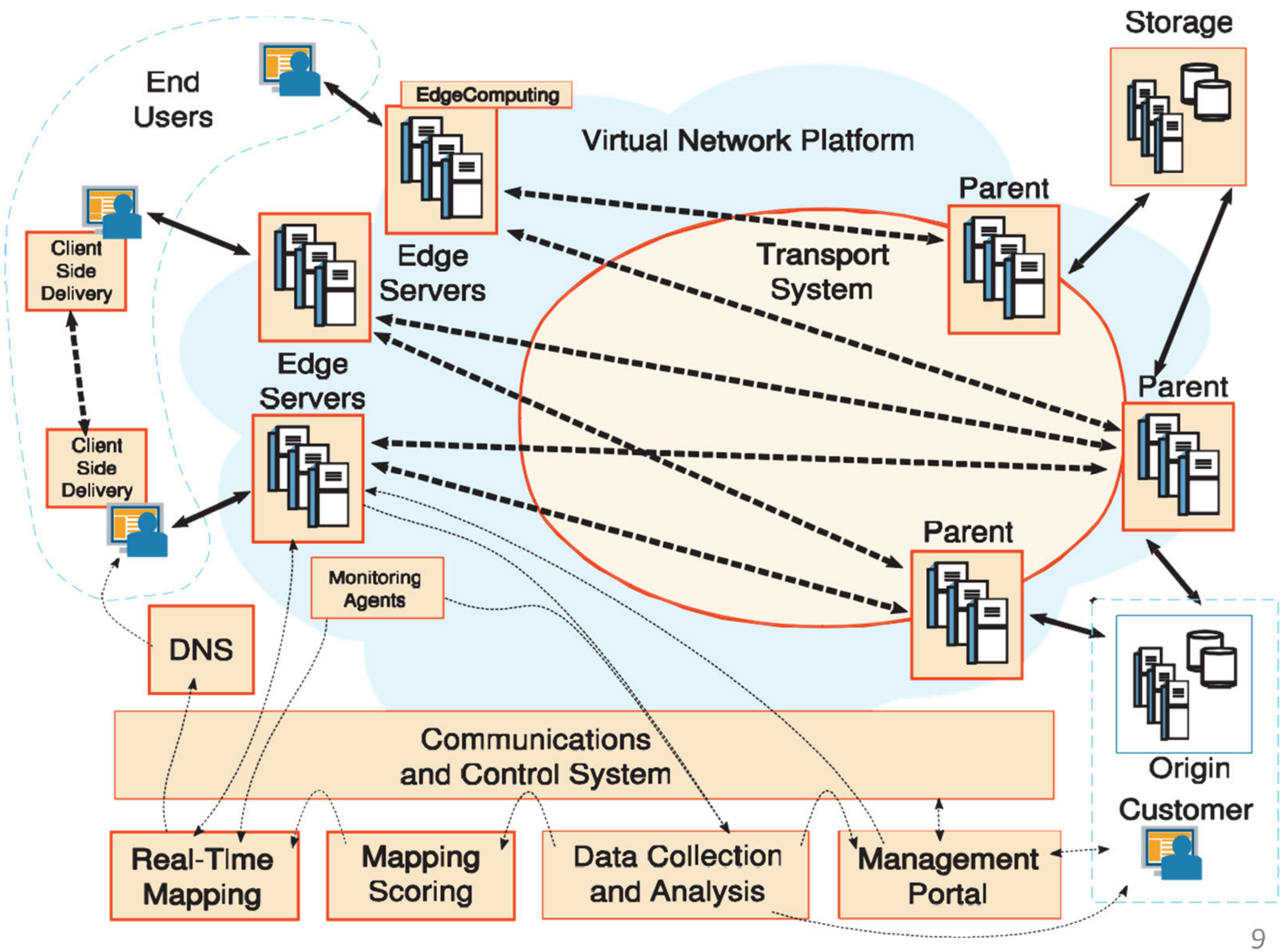
Examples of Middleware services: RPC, Transactions, Service Composition

What do we want to achieve?

Sharing of resources

some examples:

- Shared Web hosting
- Shared mail services
- Peer-to-peer systems
- Shared storage



Distribution transparency

Openness

Scalability

Distribution transparency

Transparency Type	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located must be hidden
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Objects may be both resources and processes

Degree of transparency

Observation

Aiming at full distribution transparency may be too much:

1. There are communication latencies that cannot be hidden
2. Completely hiding failures of networks and nodes is (theoretically and practically) impossible   the cost of distibuted system grow if you want to hide failures
   - You cannot distinguish a slow computer from a failing one
   - You can never be sure that a server actually performed an operation before a crash
3. Full transparency will cost performance, exposing distribution of the system
   - Keeping replicas exactly up-to-date with the master takes time
   - Immediately flushing write operations to disk for fault tolerance

Degree of transparency

Exposing distribution may be good

1. Making use of location-based services (finding your nearby friends) when dealing with users in different time zones
2. When it makes it easier for a user to understand what's going on (when e.g., a server does not respond for a long time, report it as failing).

Conclusion

Distribution transparency is a nice a goal, but achieving it is a different story, and it should often not even be aimed at.

Openness of distributed systems

se voglio eseguire il codice su diversi dispositivi l'enviroment deve essere standardizzato

What are we talking about?

Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined interfaces
- Systems should easily interoperate
- Systems should support portability of applications
- Systems should be easily extensible

An OMG® Interface Definition Language™ Publication

Interface Definition Language™

Version 4.2

OMG Document Number: formal/18-01-05 Release Date: March 2018
Standard Document URL: http://www.omg.org/spec/IDL

IPR mode: ***Non-Assert***

Simple example :

https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlExample.html

è possibile importare interfaccie e quindi avere gia un struttura del software

Openness: policies versus mechanisms

Implementing openness: policies

- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

Implementing openness: mechanisms

- Allow (dynamic) setting of caching policies
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms

On strict separation

Observation

The stricter the separation between policy and mechanism, the more we need to make ensure proper mechanisms, potentially leading to many configuration parameters and complex management.

Finding a balance

Hard coding policies often simplifies management and reduces complexity at the price of less flexibility. There is no obvious solution.

Scale in distributed systems

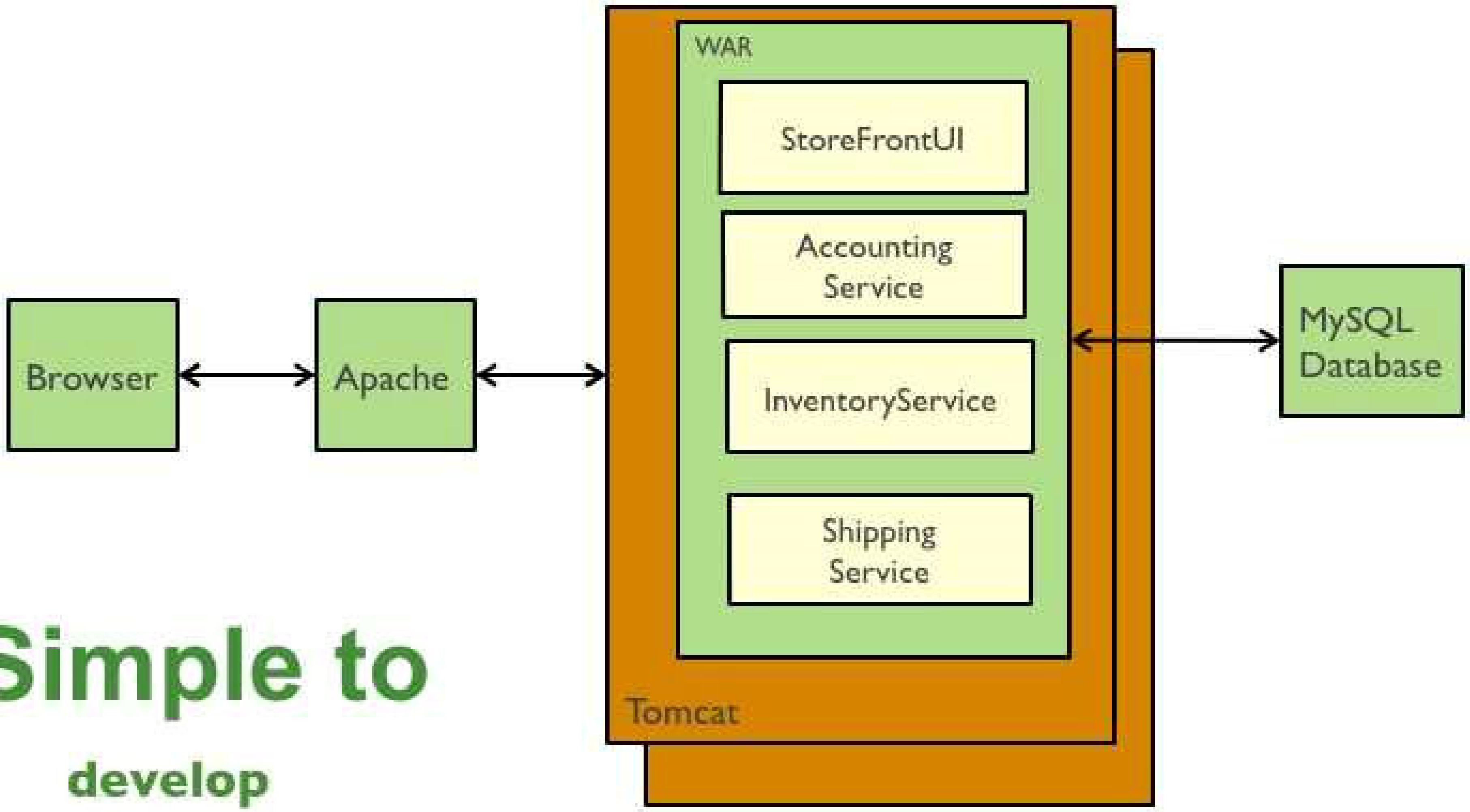
Observation

Many developers of modern distributed systems easily use the adjective "scalable" without making clear why their system actually scales.

At least three components

- Number of users and/or processes (size scalability)
- Maximum distance between nodes (geographical scalability)
- Number of administrative domains (administrative scalability)

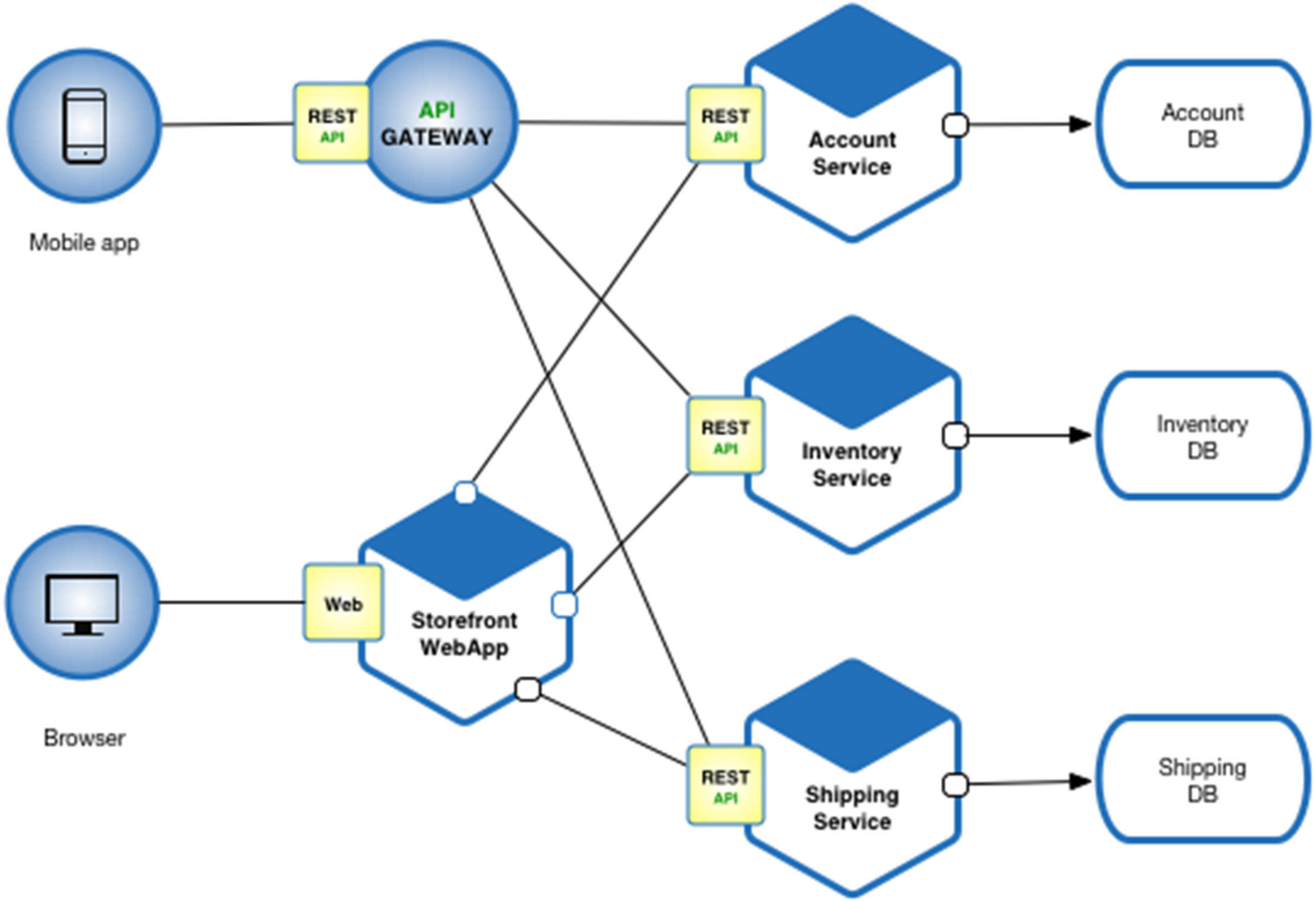
Traditional web application architecture



Simple to

- **develop**
- **test**
- **deploy**
- **scale**

diverse componenti su diversi nodi in caso di necessità



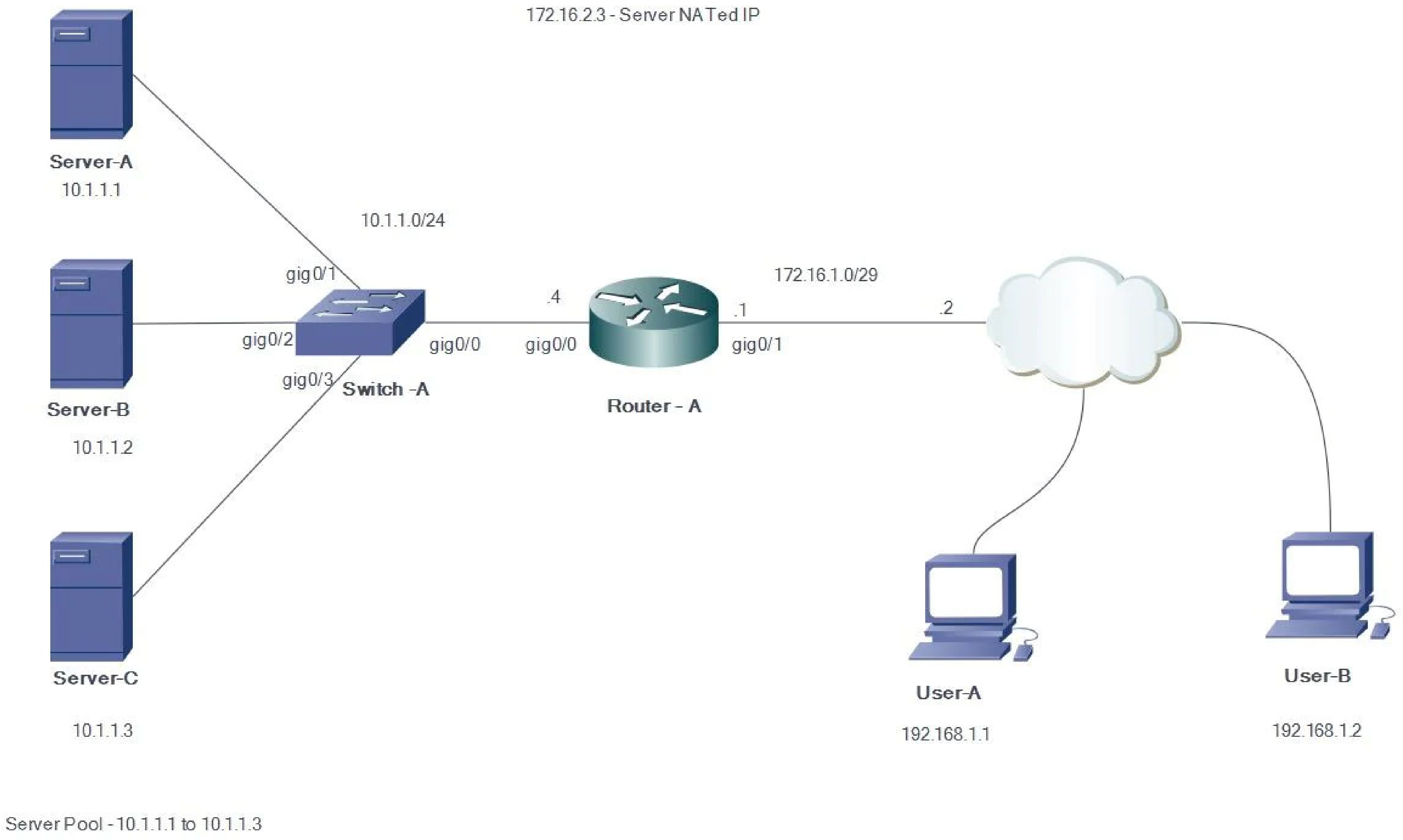
Size scalability

Observation

Most systems account only, to a certain extent, for size scalability. Often a solution: multiple powerful servers operating independently in parallel. Today, the challenge still lies in geographical and administrative scalability.

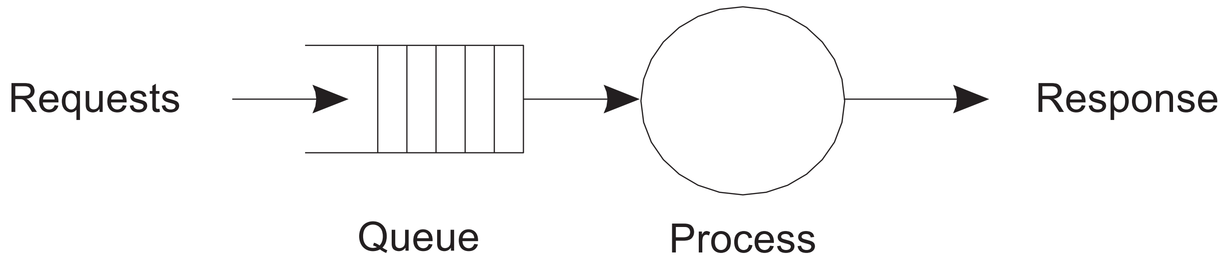
Root causes for scalability problems with centralized solutions

- The computational capacity, limited by the CPUs
- The storage capacity, including the transfer rate between CPUs and disks
- The network between the user and the centralized service


172.16.2.3 - Server NATed IP
Server-A
10.1.1.1
10.1.1.0/24
gig0/1
Server-B
gig0/2
gig0/3
Switch -A
10.1.1.2
gig0/0
.4
gig0/0
Router - A
.1
gig0/1
172.16.1.0/29
.2
Server-C
10.1.1.3
User-A
192.168.1.1
User-B
192.168.1.2
Server Pool - 10.1.1.1 to 10.1.1.3

Formal analysis

A centralized service can be modeled as a simple queuing system



Assumptions and notations

The queue has infinite capacity $\Rightarrow$ arrival rate of requests is not influenced by current queue length or what is being processed.
Arrival rate requests: λ requests per second
Processing capacity service: μ requests per second

Fraction of time having *k* requests in the system

$$p_k = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k$$

Formal analysis

Utilization U of a service is the fraction of time that it is busy

$$U = \sum_{k>0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \quad \Rightarrow \quad p_k = (1-U)\, U^k$$

Average number of requests in the system

$$\overline{N} = \sum_{k\geq 0} k \cdot p_k = \sum_{k\geq 0} k \cdot (1-U)U^k = (1-U) \sum_{k\geq 0} k \cdot U^k = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U}$$

Average throughput

$$X = \underbrace{U \cdot \mu}_{\text{server at work}} + \underbrace{(1-U)\cdot 0}_{\text{server idle}} = \frac{\lambda}{\mu} \cdot \mu = \lambda$$

Formal analysis

Response time: total time take to process a request after submission

$$R = \frac{\overline{N}}{X} = \frac{S}{1 - U} \Rightarrow \frac{R}{S} = \frac{1}{1\text{-}U}$$

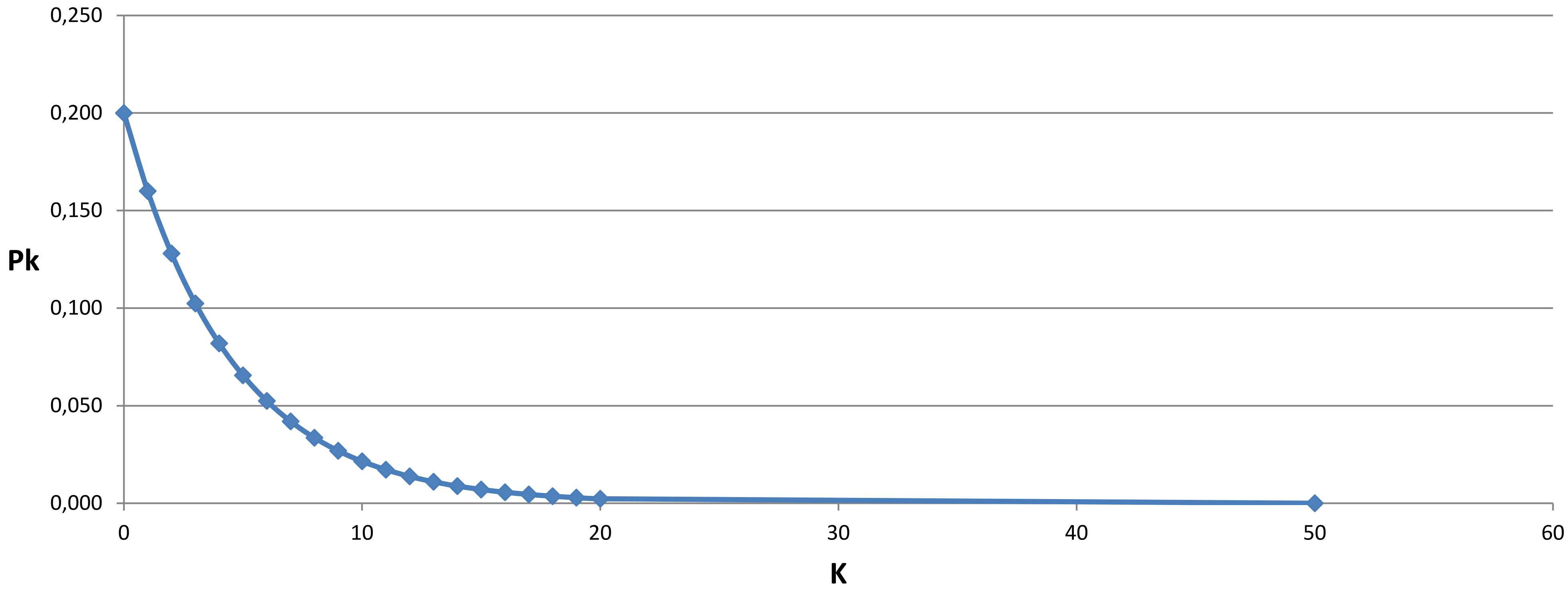
with $S = \frac{1}{\mu}$ being the service time.

Observations

If U is small, response-to-service time is close to 1: a request is immediately processed

If U goes up to 1, the system comes to a grinding halt.

Solution: decrease S.



ServiceCapacity.xlsx

Problems with geographical scalability

posizioni dei server in diverse aree giografiche

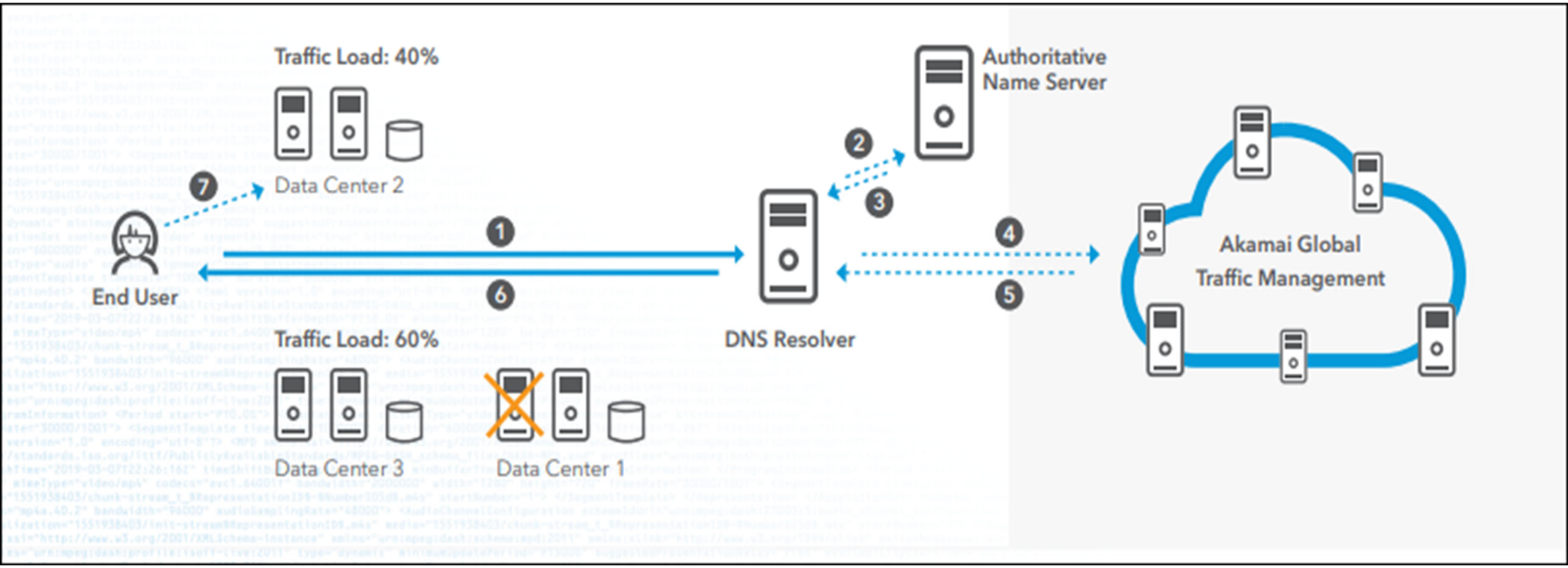
Cannot simply go from LAN to WAN: many distributed systems assume synchronous client-server interactions: client sends request and waits for an answer. Latency may easily prohibit this scheme.

WAN links are often inherently unreliable: simply moving streaming video from LAN to WAN is bound to fail.

Lack of multipoint communication, so that a simple search broadcast cannot be deployed. Solution is to develop separate naming and directory services (having their own scalability problems).

Example of geographical scalability : AKAMAI

distribuiva dataset in diverse parte del mondo e poi veniva destribuita la richiesta ricevuta a seconda di disponibilita dei singoli server



Problems with administrative scalability

Essence

Conflicting policies concerning usage (and thus payment), management, and security

Examples

Computational grids: share expensive resources between different domains.

Shared equipment: how to control, manage, and use a shared radio telescope constructed as large-scale shared sensor network?

Exception: several peer-to-peer networks

File-sharing systems (based, e.g., on BitTorrent)
Peer-to-peer telephony (Skype)
Peer-assisted audio streaming (Spotify)

Note: end users collaborate and not administrative entities.

Techniques for scaling

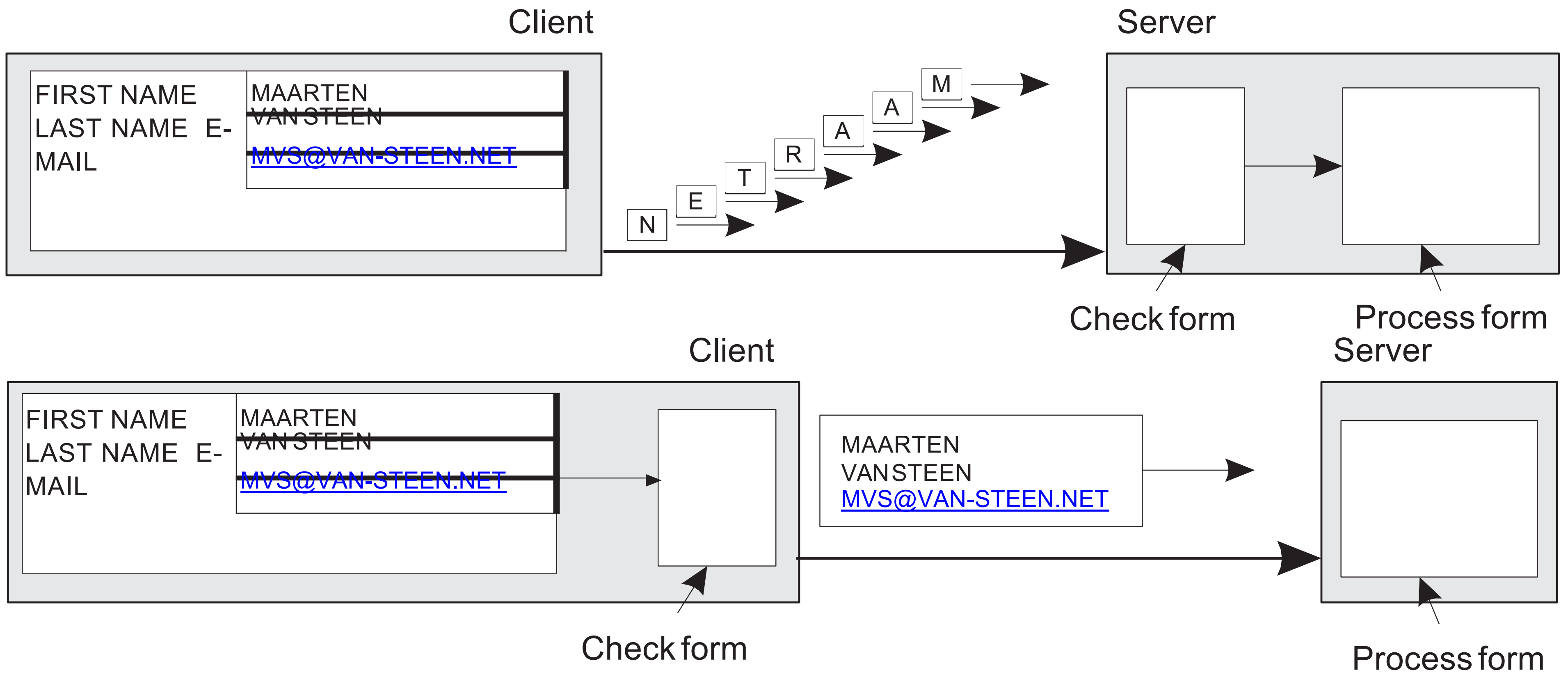
Hide communication latencies

- Make use of asynchronous communication
- Have separate handler for incoming response
- Problem: not every application fits this model

Techniques for scaling

un esempio è questo si puo delegare al client certi tipi di operazioni , nell'esempio la validita dell'email

Facilitate solution by moving computations to client



Techniques for scaling

Partition data and computations across multiple machines

- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)

Replication and caching: Make copies of data available at different machines

- Mirrored Web sites (Content Delivery networks, e.g. Akamai)
- Web caches (in browsers and proxies)
- File caching (at server and client)
- Replicated file servers and databases
- (e.g. Infinispan)

muovere i dati piu vicino all'utente

Scaling: The problem with replication

avendo varie copie dei dati puo succedere che i dati non siano allineati , devono essere aggiornati tutti

Applying replication is easy, except for one thing

- Having multiple copies (cached or replicated), leads to inconsistencies: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires global synchronization on each modification.
- Global synchronization precludes large-scale solutions.

Observation

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but tolerating inconsistencies is application dependent.

Developing distributed systems: Pitfalls

Observation

Many distributed systems are needlessly complex caused by mistakes that required patching later on. Many false assumptions are often made.

False (and often hidden) assumptions

The network is reliable

The network is secure

The network is homogeneous

The topology does not change

Latency is zero

Bandwidth is infinite

Transport cost is zero

There is one administrator

Three types of distributed systems

nodi con una grande potenza di calcolo

High performance distributed computing systems

Distributed information systems

Distributed systems for pervasive computing

un grandissimo numero di nodi

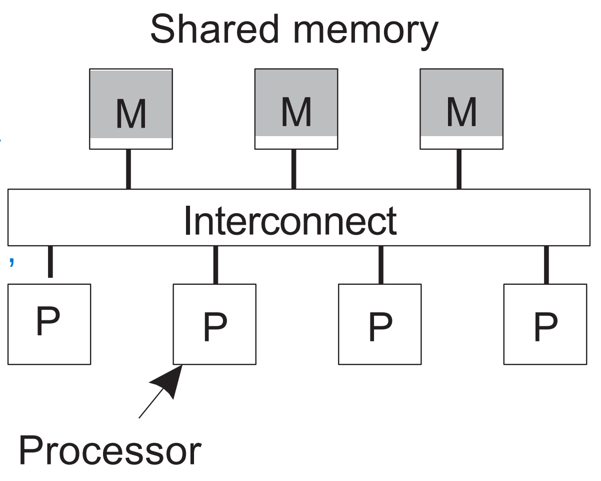
Parallel computing

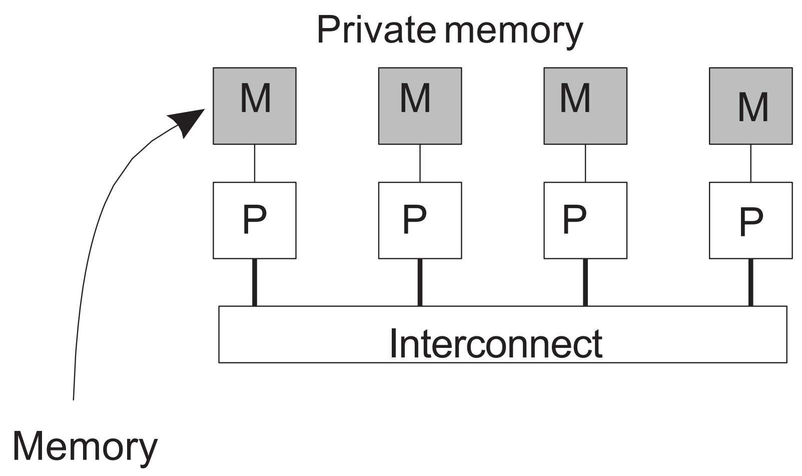
Observation

High-performance distributed computing started with parallel computing

Multiprocessor and multicore versus multicomputer

memoria separate dalla potenza coputazionale, piu efficente la della gestione della memoria , meno ripetizioni ecc





tutto nella stessa location, migliori performance

metodo usato dalle schede grafiche, ogni nodo coputazionae ha la propria memoria

Distributed shared memory systems

Observation

al contempo due nodi possono richiedere lo stesso data, questo puo dare problemi di concorrenza

Multiprocessors are relatively easy to program in comparison to multicomputers, yet have problems when increasing the number of processors (or cores).

Solution: Try to implement a shared-memory model on top of a multicomputer.

Example through virtual-memory techniques

Map all main-memory pages (from different processors) into one single virtual address space. If process at processor *A* addresses a page *P* located at processor *B*, the OS at *A* traps and fetches *P* from *B*, just as it would if *P* had been located on local disk.

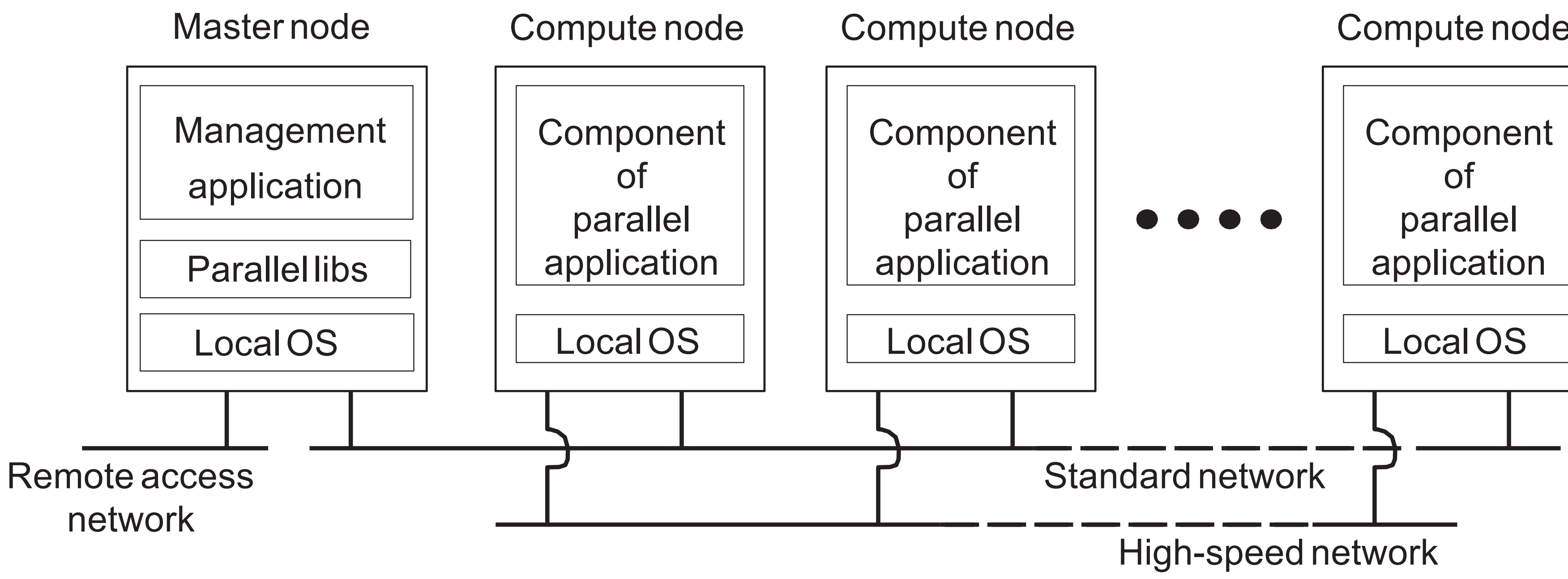
Problem

Performance of distributed shared memory could never compete with that of multiprocessors, and failed to meet the expectations of programmers. It has been widely abandoned by now.

Cluster computing

Essentially a group of high-end systems connected through a LAN

- Homogeneous: same OS, near-identical hardware
- Single managing node

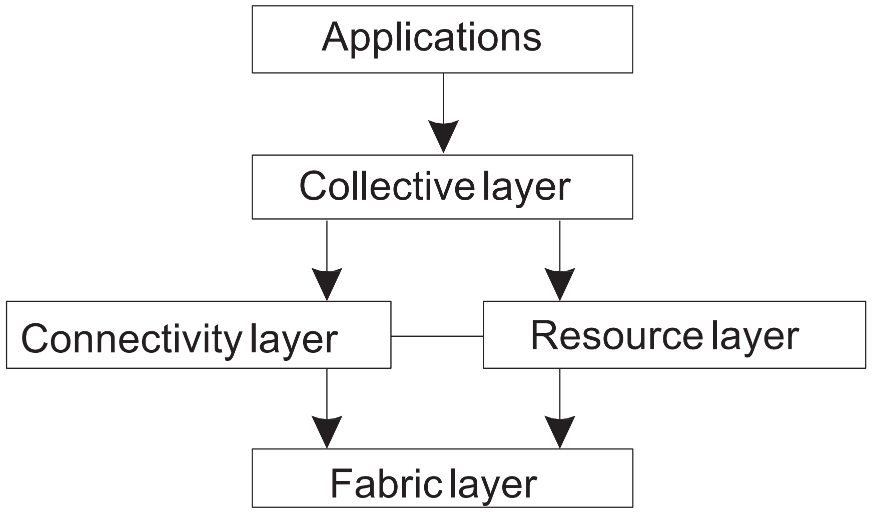


Grid Computing: lots of nodes from everywhere

Heterogeneous gestione di vari nodi

Dispersed across several organizations

Can easily span a wide-area network



The layers

Fabric: Provides interfaces to local resources (for querying state and capabilities, locking, etc.)

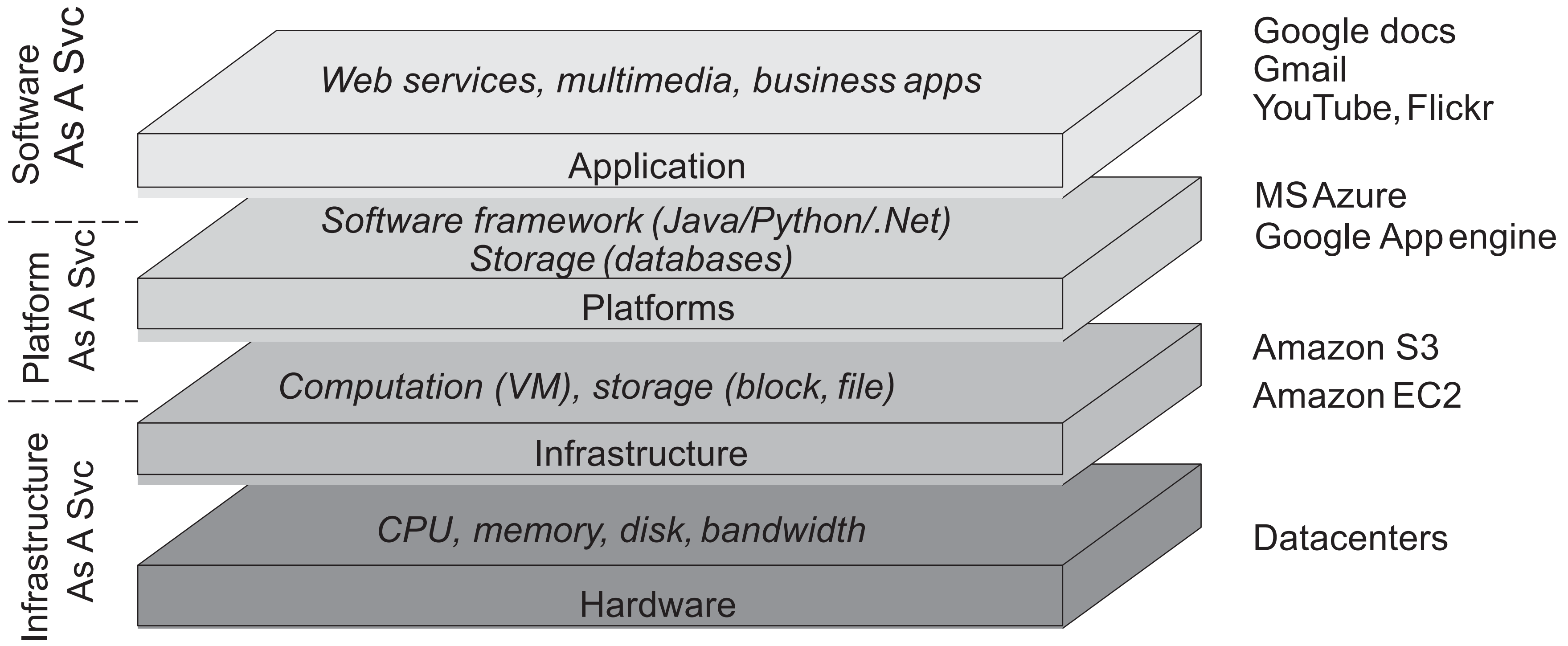
Connectivity: Communication/transaction protocols, e.g., for moving data between resources. Also various authentication protocols.

Resource: Manages a single resource, such as creating processes or reading data.

Collective: Handles access to multiple resources: discovery, scheduling, replication.

Application: Contains actual grid applications in a single organization.

Cloud computing



Distributed Information Systems: integrating applications

Situation

Organizations confronted with many networked applications, but achieving interoperability was painful.

Basic approach

A networked application is one that runs on a server making its services available to remote clients. Simple integration: clients combine requests for (different) applications; send that off; collect responses, and present a coherent result to the user.

Next step

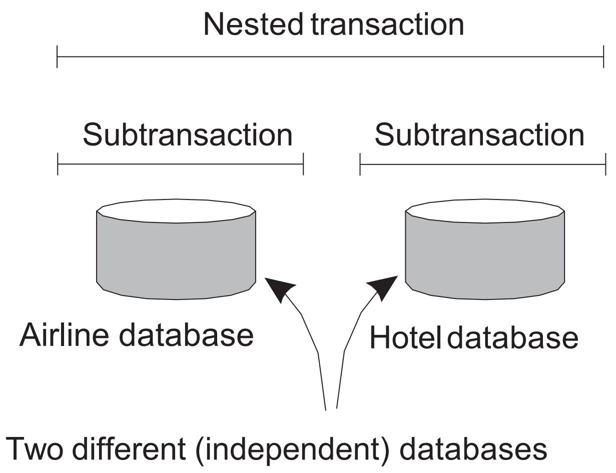
Allow direct application-to-application communication, leading to Enterprise Application Integration.

Example EAI: (nested) transactions

Transaction

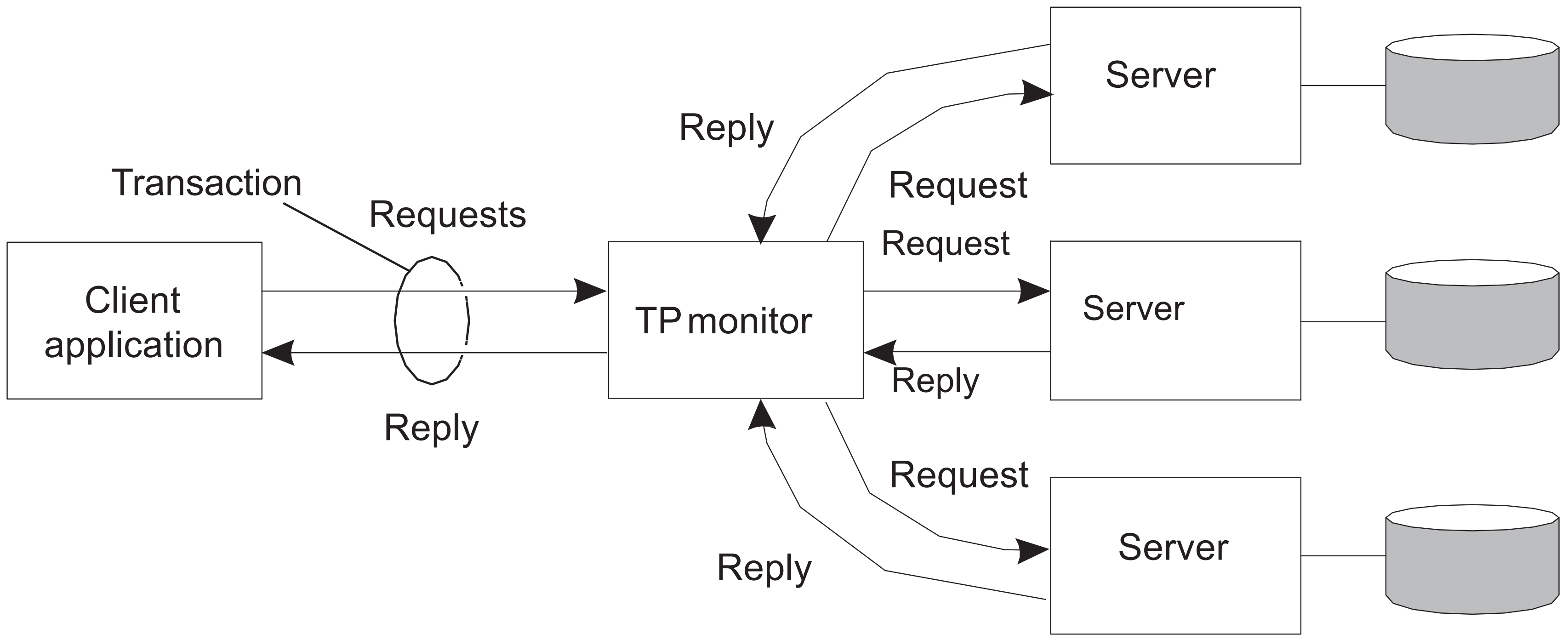
Primitive	**Description**
BEGIN TRANSACTION	Mark the start of a transaction
END TRANSACTION	Terminate the transaction and try to commit
ABORT TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Issue: all-or-nothing



ACID
Atomic: happens indivisibly (seemingly)
Consistent: does not violate system invariants
Isolated: not mutual interference
Durable: commit means changes are permanent

TPM: Transaction Processing Monitor



Observation

In many cases, the data involved in a transaction is distributed across several servers. A TP Monitor is responsible for coordinating the execution of a transaction.

How to integrate applications

File transfer: Technically simple, but not flexible:

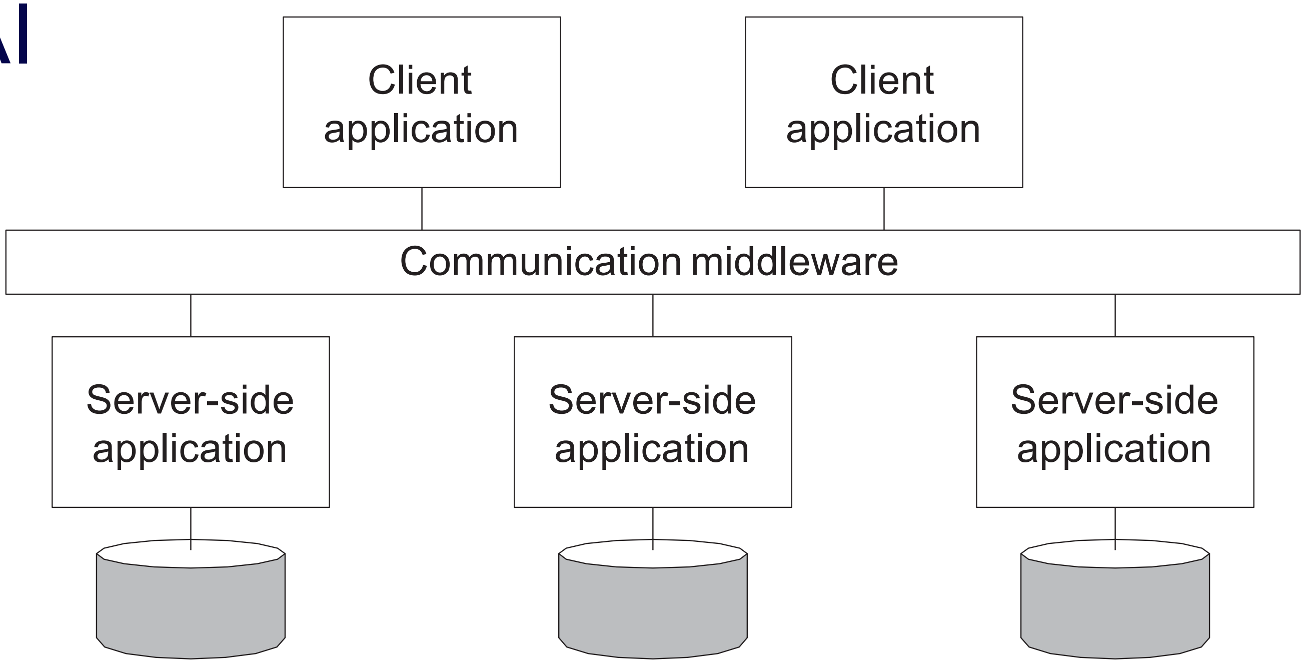
Figure out file format and layout
Figure out file management
Update propagation, and update notifications.

Shared database: Much more flexible, but still requires common data scheme next to risk of bottleneck.

Remote procedure call: Effective when execution of a series of actions is needed.

Messaging: RPCs require caller and callee to be up and running at the same time. Messaging allows decoupling in time and space.

Middleware and EAI



Middleware offers communication facilities for integration

Remote Procedure Call (RPC): Requests are sent through local procedure call, packaged as message, processed, responded through message, and result returned as return from call. Similar approach for Remote Metod Invocation (RMI) based on objects instead of procedure calls.

Message Oriented Middleware (MOM): Messages are sent to logical contact point (published), and forwarded to subscribed applications.

Distributed pervasive systems

Observation

Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system, characterized by the fact that the system naturally blends into the user’s environment.

Three (overlapping) subtypes

- Ubiquitous computing systems: pervasive and continuously present, i.e., there is a continuous interaction between system and user.
- Mobile computing systems: pervasive, but emphasis is on the fact that devices are inherently mobile.
- Sensor (and actuator) networks: pervasive, with emphasis on the actual (collaborative) sensing and actuation of the environment.