POLITECNICO
MILANO 1863

Distributed Systems
Modelling Distributed Systems

Gianpaolo Cugola

Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

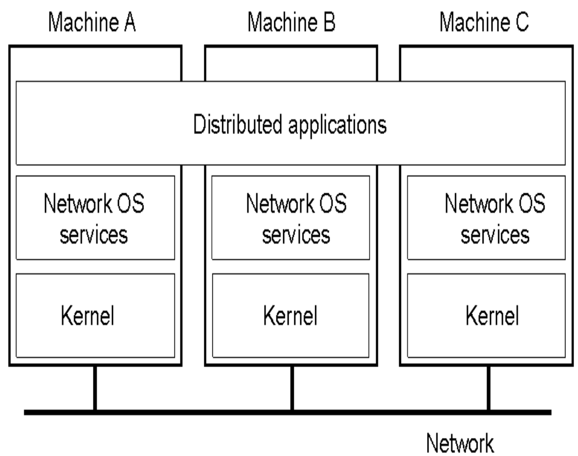
gianpaolo.cugola@polimi.it
http://home.dei.polimi.it/cugola

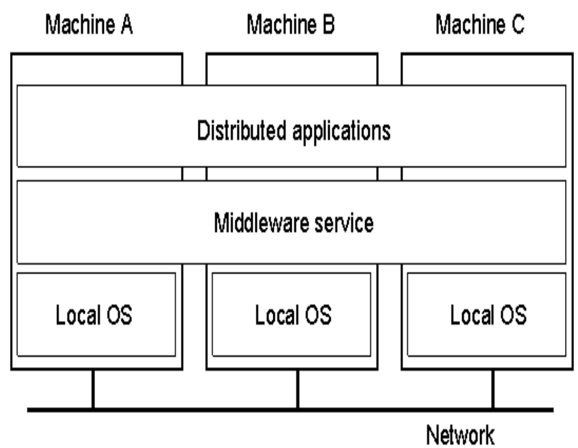
Contents

- **The software architecture of a distributed system**
- The run-time architecture
- The interaction model
- The failure model

The software architecture of a distributed system

- Network OS based
  - The network OS provides the communication services
  - Diferent machines may have different network OSes
  - Masking platform differences is up to the application programmer
- Middleware based
  - The middleware provides advanced communication, coordination, and administration services
  - It masks most of the platform differences





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 or 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, ...

Contents

- The software architecture of a distributed system
- **The run-time architecture**
- The interaction model
- The failure model

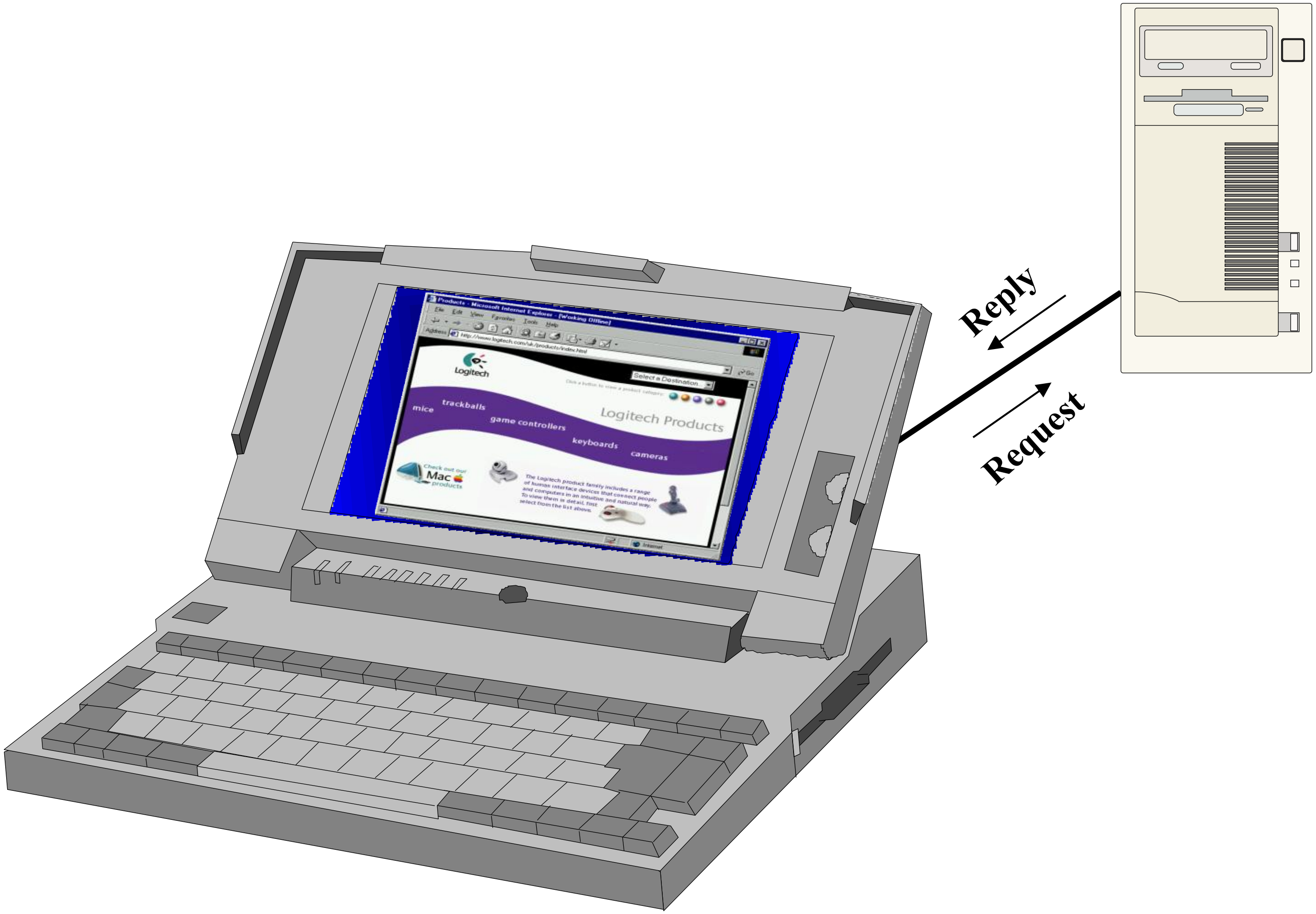
The run-time (system) architecture of a distributed system

- 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
  - CREST

Client-server

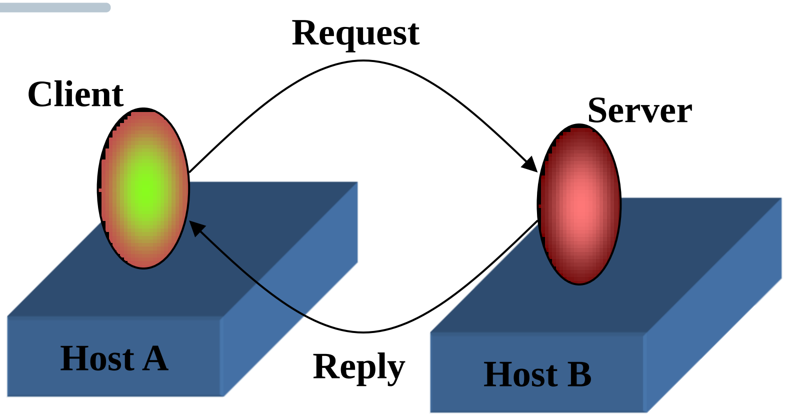
- The most common architectural style today
- Components have different roles
  - Servers provide a set of services through a well defined API
    - They are passive (just wait for client invocations)
  - Users access those services through clients
  - Communication is message based (or RPC)

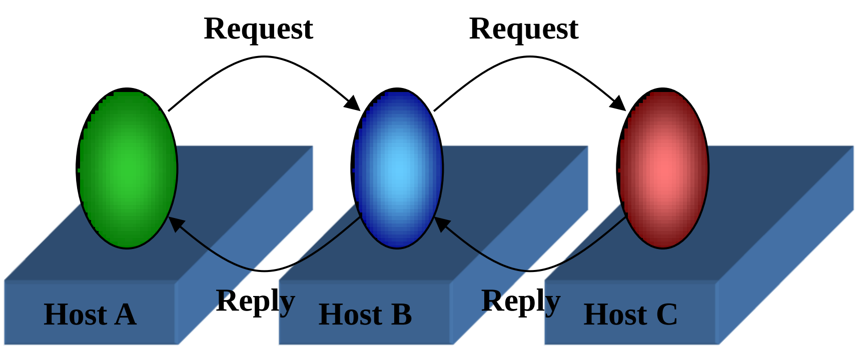
The Web is a client-server application



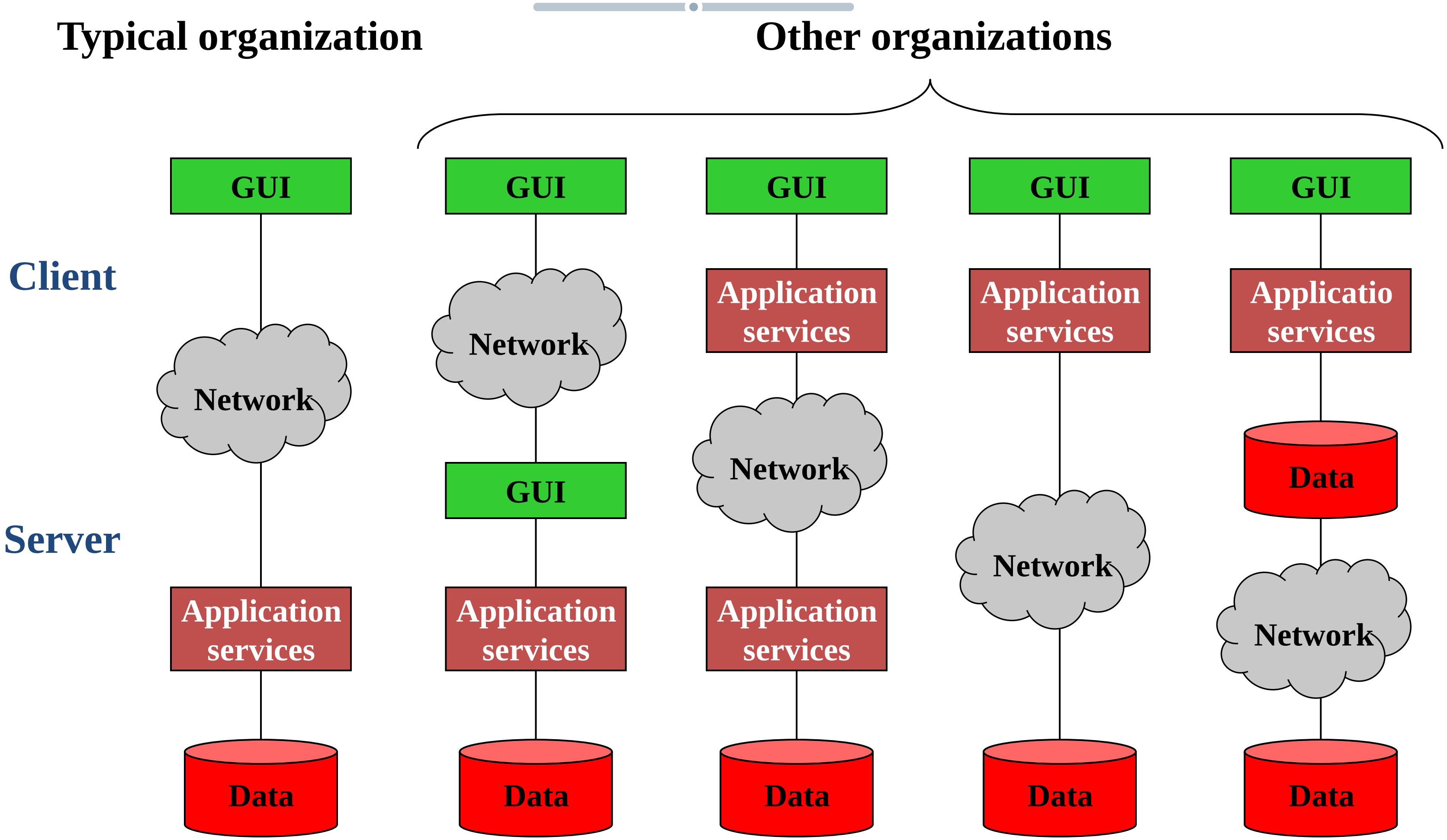
Tiers

- 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

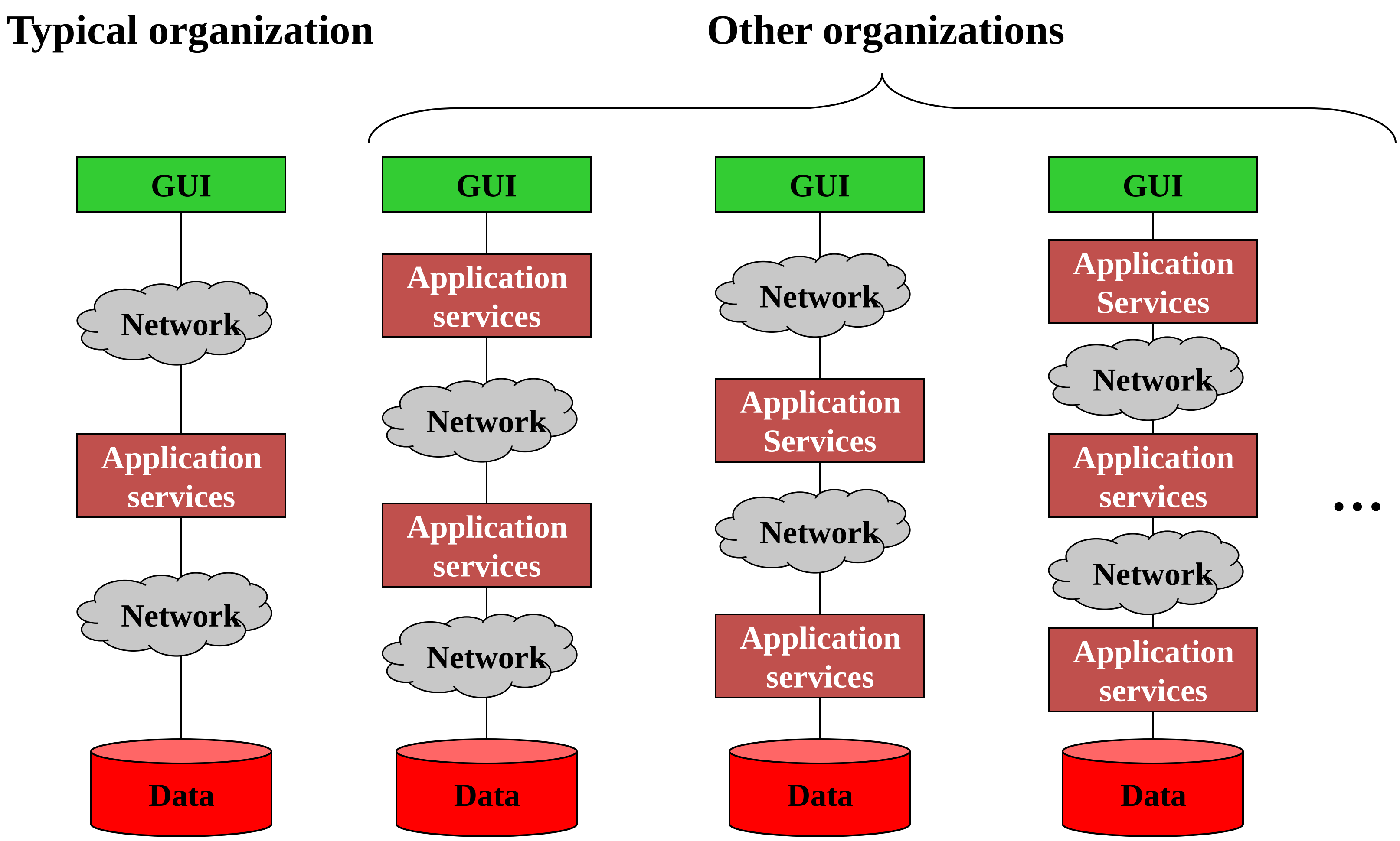




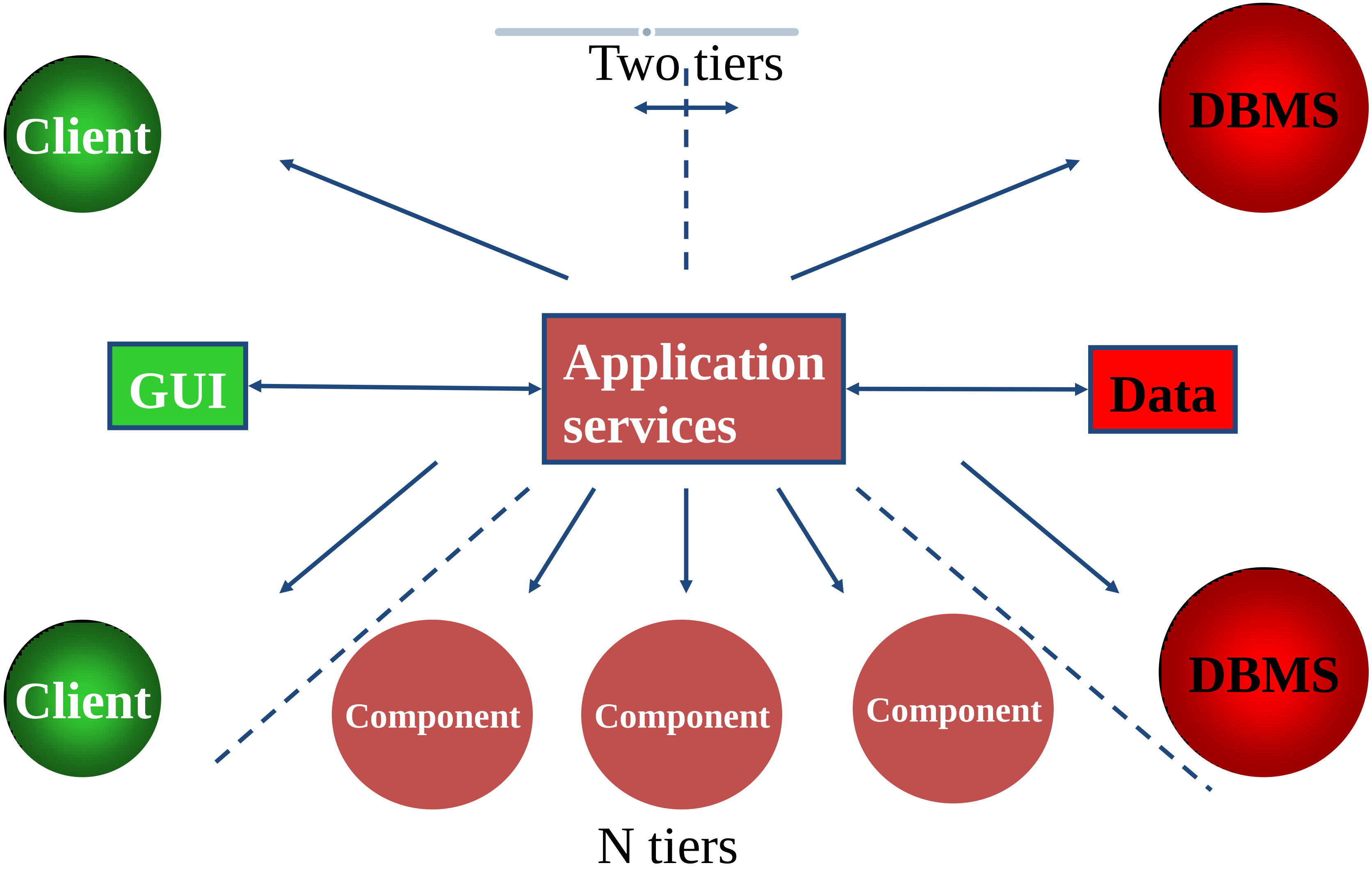
Two tiered architectures



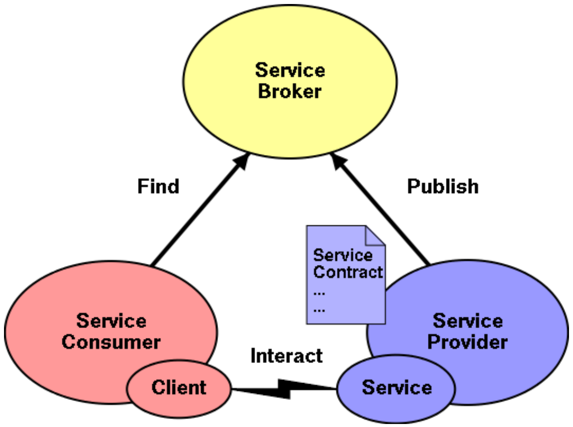
Three-tiered architectures



From two to multi-tiered architectures



The Service Oriented Architecture



- Built around the concepts of *services*, *service providers*, *service consumers*, *service brokers*
  - Services represent loosely coupled units of functionality...
  - ...exported by service providers
  - Brokers hold the description of available services to be searched by interested consumers...
  - ...which bind and invoke the services they need
  - *Orchestration* is the process of invoking a set of services in an ad-hoc workflow to satisfy a given goal
- Several incarnations
  - OSGI (Open Grid Services Infrastructure, JXTA, Jini, Web Services

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)
  - It includes the set of *operations* exported by the web service
- 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
- *UDDI* (Universal Description Discovery & Integration) describes the rules that allows web services to be exported and searched through a *registry*

The REST style

- REpresentational State Transfer (REST) is both:
  - A (nice) way to describe the web
    - by Roy Thomas Fielding, one of the authors of HTTP/1.1, co-founder and member of the Apache Software Foundation
  - A set of principles that define how Web standards are supposed to be used
    - Which often differs quite a bit from what many people actually do
- 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

REST: Main constraints

- Interactions are client-server
- Interactions are stateless
  - State must be transferred from clients to servers
- The data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable
  - The ability of caching data is key to provide scalability
- Each component cannot “see” beyond the immediate layer with which they are interacting
  - REST is layered
- Clients must support code-on-demand
  - This is an optional constraint (more on this later)
- Components expose a uniform interface

low services, the most of the information about what you want the service to do is done by encoding parameters.

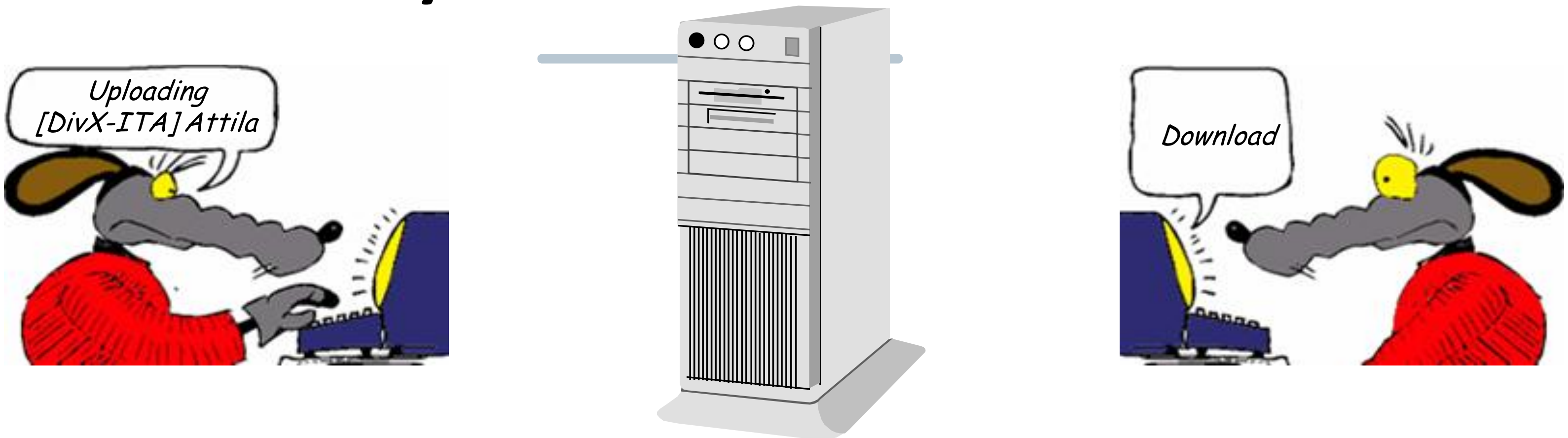
REST: Uniform interface constraints

- 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 behavior of some connecting elements (e.g., the cache behavior)
  - 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 hipermedia links

Peer-to-peer

- In a peer-to-peer applications all components play the same role
  - There is no distinction between clients and servers
- 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)

Why P2P is different



- Fundamental difference:

> *"Take advantage of resources at the edges of the network"*
> (Clay Shirky, O'Reilly)

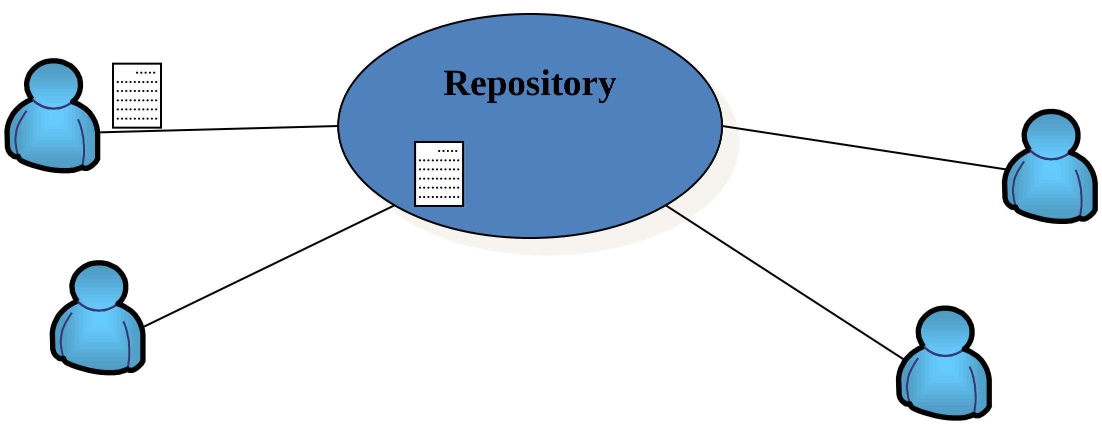
- What's changed:
  - End-host resources have increased dramatically
  - Broadband connectivity now common

Object-Oriented

- The distributed components encapsulate a data structure providing an API to access and modify it (the descripting language is indipendent by the implementation language)
  - 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

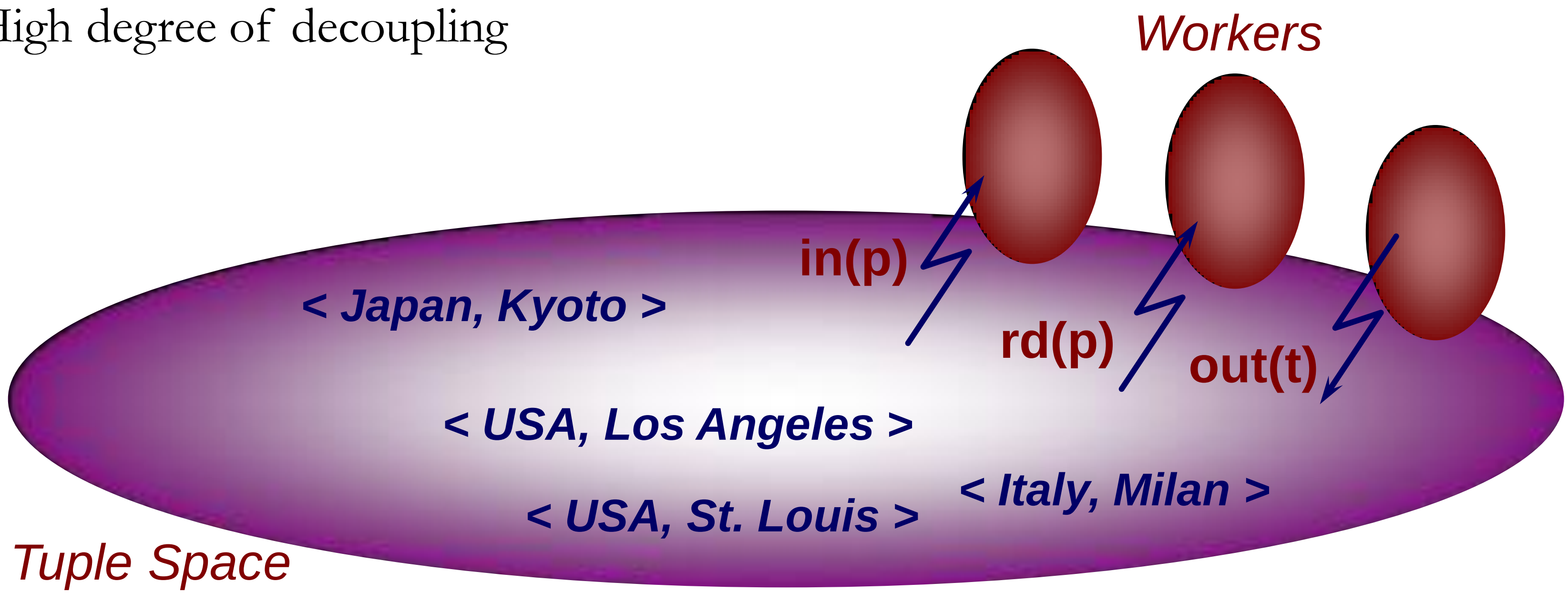
Data-centered

- Components communicate through a common (usually passive) repository
  - Data can be added to the repository or taken (moved or copied) from it
- Communication with the repository is (usually) through RPC
- Access to the repository is (usually) synchronized



Linda and tuple spaces

- Data sharing model proposed in the 80s by Carriero and Gelernter, mostly used for parallel computation
- Recently revitalized in the context of distributed computing
  - E.g., IBM TSpaces, Sun JavaSpaces, GigaSpaces
- Communication is persistent, implicit, content-based, generative
- High degree of decoupling



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

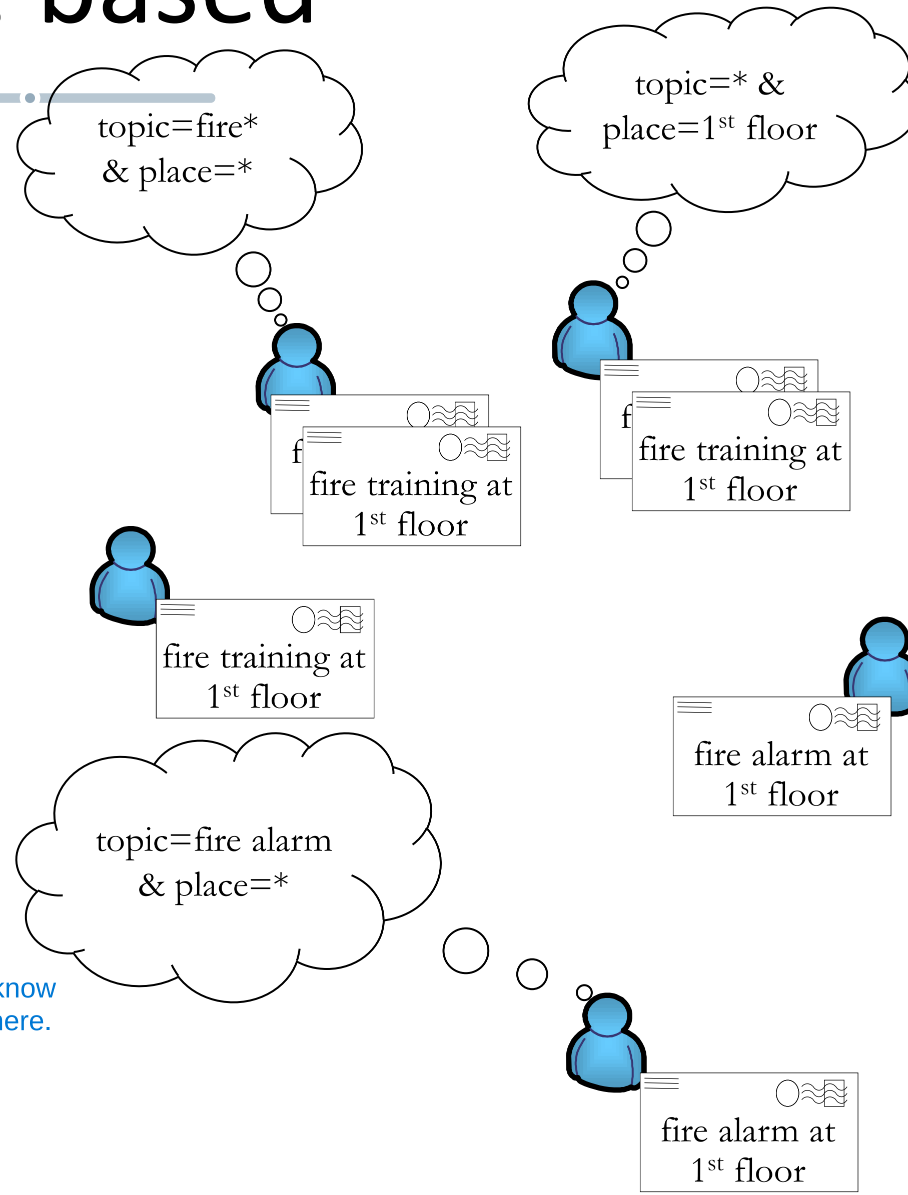
Architectural issues

- 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

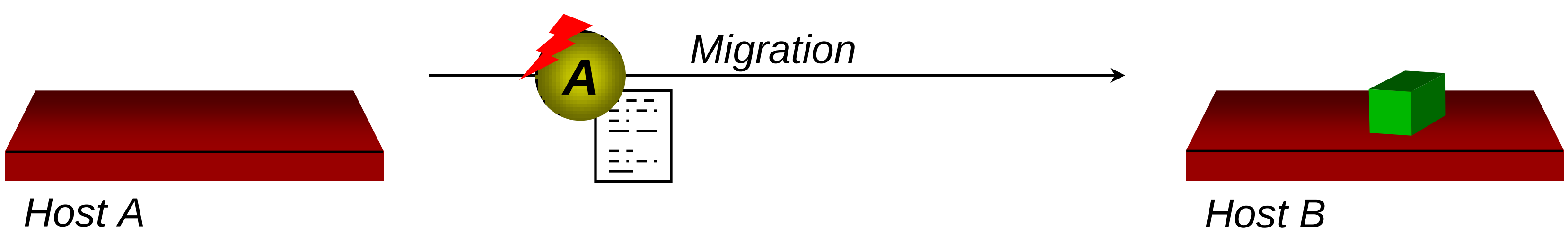
- Components collaborate by exchanging information about occurrent *events*. In particular:
  - Components *publish* notifications about the events they observe, or
  - they *subscribe* to the events they are interested to be notified about
- Communication is:
  - Purely message based
  - Asynchronous
  - Multicast
  - Implicit
  - Anonymous

when you publish you don't know how many subscribers are there.



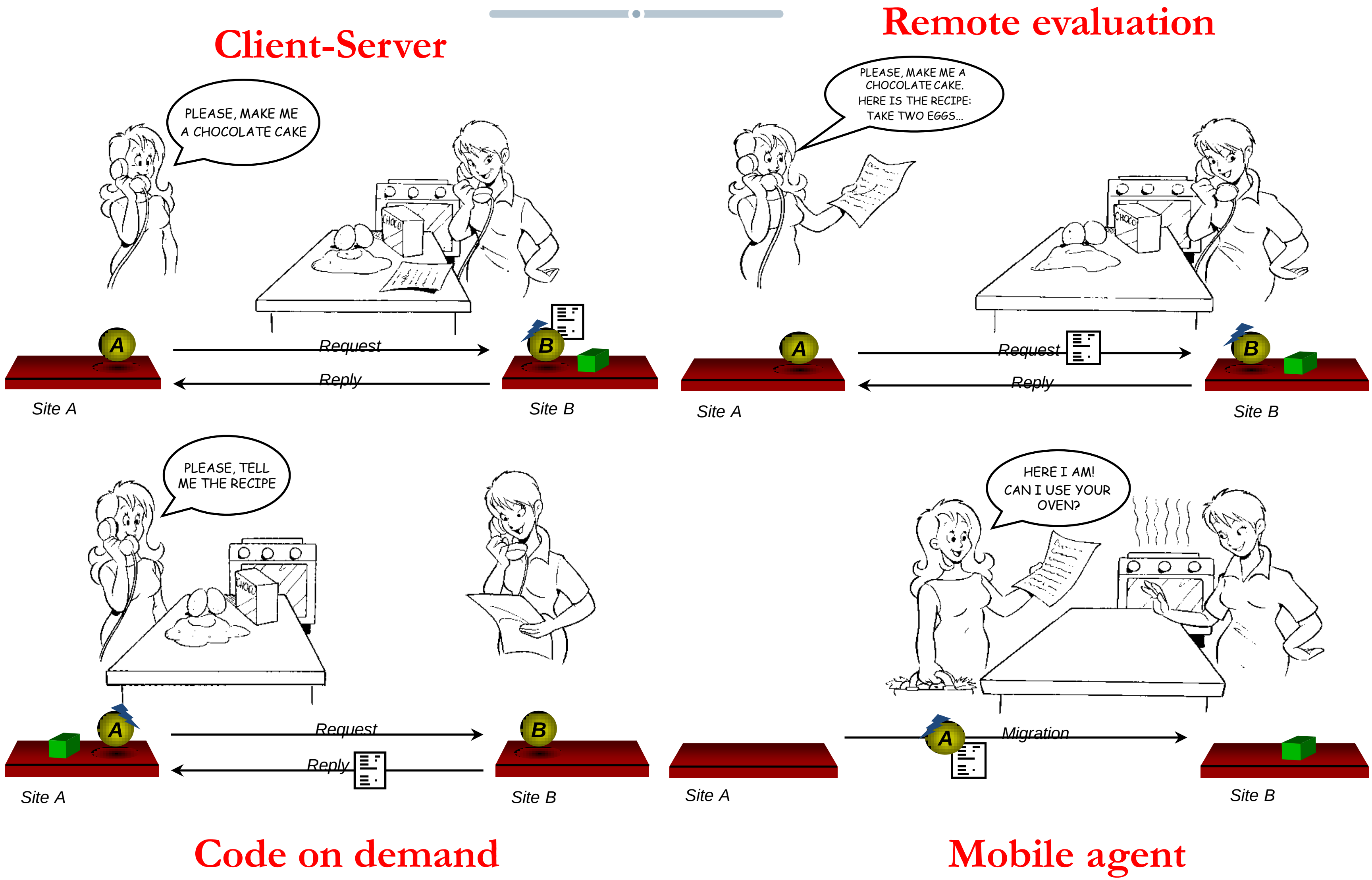
Mobile code

- 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
- Different models depending on the original and final location of resources, know-how (the code) and computational components (including the state of execution)



Mobile code paradigms

Application in which we want to do a cake.



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

CREST: REST meets mobile code

- REST is not sufficient to describe complex web 2.0 applications
  - E.g., those enabled by AJAX
- CREST (Computational REST) joins together the concepts of REST with mobile code
- Instead of “representations” interacting parties exchange “computations”
  - I.e., closures and continuations
- CREST axioms
  - A resource is a locus of computations named by an URL
  - The representation of a computation is an “expression” plus metadata to describe it
  - All computations are context-free
  - Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged
  - The presence of intermediaries is promoted

Contents

- The software architecture of a distributed system
- The run-time architecture
- **The interaction model**
- The failure 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
- Distributed systems are composed of many processes, which interact in complex ways
- The behaviour of a distributed system is described by a *distributed algorithm*
  - A definition of the steps taken by each process, *including the transmission of messages between them*
- The behavior of a distributed system is influenced by several factors: not the performance!
  - The rate at which each process proceeds
  - The performance of the communication channels
  - The different clock drift rates

Those are unrelevant when talking about a traditional algorithm

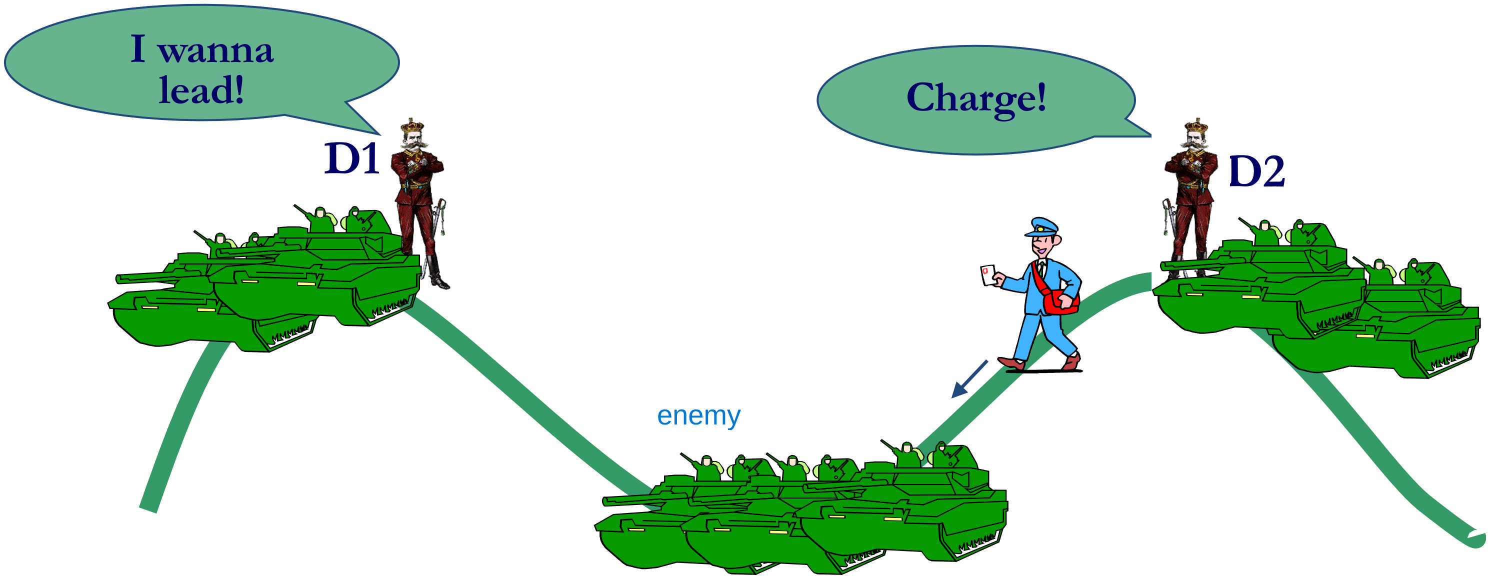
Interaction model

- To formally analyze the behavior 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

Usually systems are asynchronous

- Any solution that is valid for an asynchronous distributed system is also valid for a synchronous one (but the vice versa is clearly false)

The pepperland example



- The pepperland divisions are safe as long as they remain in their encampments
- If both charge at the same time they win, otherwise they loose
- Generals need to agree on:
  - Who will lead the charge
  - When the charge will take place
- We consider the case when messengers are able to walk from an hill to another without being captured by the enemies

The pepperland example

- Even in asynchronous pepperland it is possible to agree on who will lead the charge
  - How?
- Charging together is a different issue
  - It is not possible in asynchronous pepperland
    - If the leader sends a messenger to the other general saying "charge!" the messenger may take three hours or just five minutes to reach the other general
    - Also differences on each division's clock do not allow strategies based on sending a message with the time to charge
  - In synchronous pepperland it is possible to determine the maximum difference between charge times
    - Let min and max be the range of message transmission times
    - The leader sends a message "charge!", wait min minutes then charge
    - On receiving the "charge!" message the other general immediately charge
    - The second division may charge later then the first one but no more that (max-min) minutes
    - If we know that the charge will last longer then the victory is guaranteed

Contents

- The software architecture of a distributed system
- The run-time architecture
- The interaction model
- **The failure model**

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
  - Byzantine (or arbitrary) failures
    - Processes: may omit intended processing steps or add more
    - 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

Failure detection in pepperland

- How to detect if one of the two divisions has been attacked and defeated by the enemies?
- Easy in synchronous pepperland:
  - Each division periodically send a messenger to the other saying "I am still here"
  - When no messengers arrive for longer than max minutes we can conclude that the other division has been defeated
- What about asynchronous pepperland?
  - We cannot distinguish wheather the other division has been defeated or the time for the messenger t cross the valley is just very long

Agreement in “failing pepperland”

- Suppose the messengers can be captured by enemies
- Can the two generals send messengers so that they both consistently decide to charge or surrender?
  - Reaching an agreement on one of the two possible decisions requires the successful arrival of at least one message
  - Consider scenario A in which the fewest delivered messages that will result in agreement to attack are delivered
  - Let scenario B be the same as A except that the last message delivered in A is lost in B, and any other messages that might be sent later are also lost
  - Suppose this last message is from General 1 to General 2
  - General 1 sees the same messages in both scenarios, so he definitely attacks
  - However, the minimality assumption of A implies that General 2 cannot also decide to attack in scenario B, so he must make a different decision
  - Hence General 1, not being sure its last message arrived, has wrongly decided to attack (both in scenarios A and B)
  - The problem is unsolvable

Impossibility of distributed consensus in practice

- Formally demonstrated by Fischer, Lynch, Patterson in 1985
- Does it really matter in real life? Yes!!!
  - Commit or abort a transaction in a distributed database
    - E.g., when you withdraw money at the ATM
  - Agree on values of replicated, distributed sensors
  - Agree on whether a system component is faulty
- How is it solved in practice?
  - Change the assumptions
    - E.g, make links reliable (enough)
  - Reduce the guarantees:
    - E.g., only probabilistic instead of deterministic