Distributed Systems and their Software Architectures

Reference for study: Van Steen, Tanenbaum, "Distributed Systems", chapters 1,2, 4.1

Distributed System (DS): Definition

- "a computing system whose components are located on different networked computers, which communicate and coordinate their actions..." (Wikipedia)
- "a collection of autonomous computing elements that appears to its users as a single coherent system" (Tanenbaum)

⇒ DSs may include any kind of computer (from large mainframes to small devices) and **computing element** (**process**/thread/...) that can be interconnected

- ⇒ The internet and applications based on it are DSs
- ⇒ New emerging DSs: IOT, Sensor networks, Cloud-based Systems
- ⇒ In practice: most applications today are DSs

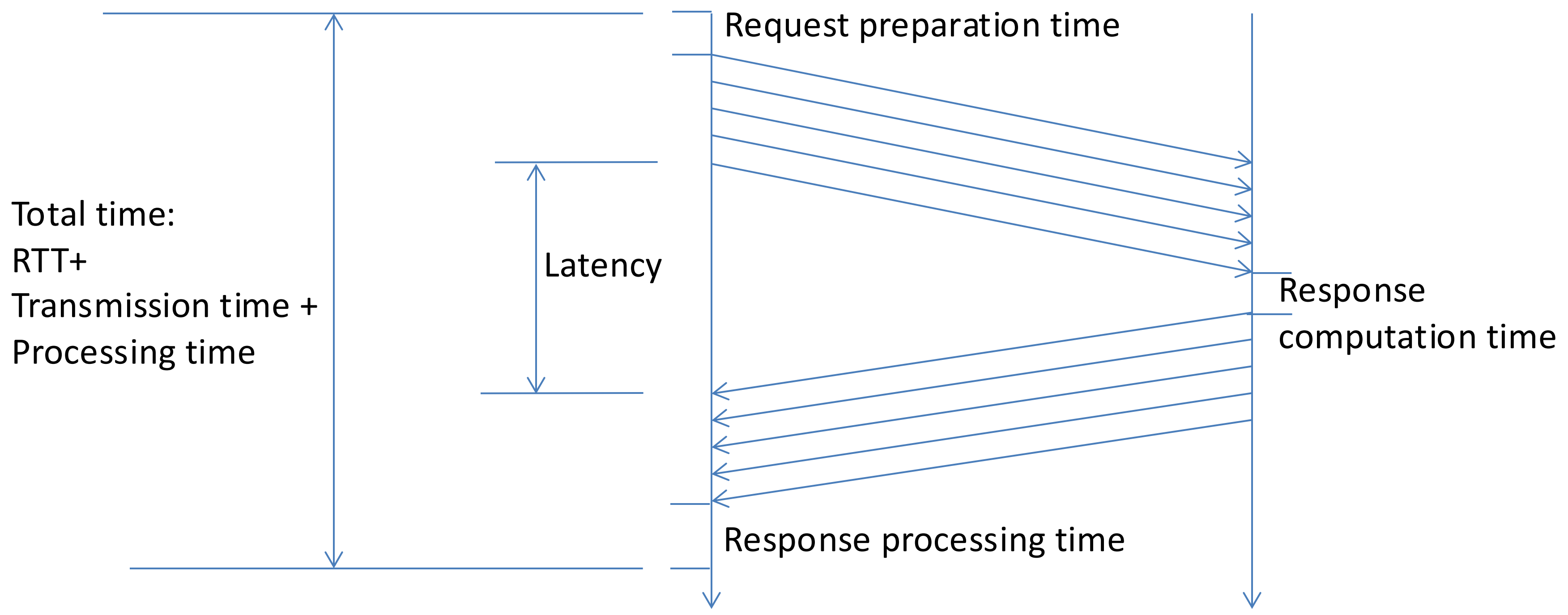
For simplicity, a computing element will be called **process** or **node**

DS Main Characterizing Properties

- Autonomy implies:
  - Asynchronous (no global clock) and concurrent
  - Heterogeneous (different HW, OS, programming language, location)
  - Possibility of partial failure
- Networked Interconnection (& geographical dispersion) imply
  - Communications take long/variable time and can fail
  - Large attack surface for malicious intruders
    - security issues are more relevant
      - unauthorized access to private data and services (e.g. eavesdropping, impersonation/ spoofing)
      - disruption or alteration of the intended system behavior and properties (e.g. man-in-the-middle, reply, DOS, code vulnerability exploits)

About Communication Time

- Each distributed interaction takes processing time + communication time
- Example: UDP request/response interaction



About Communication Time (contd)

- Processing time is less than communication time by orders of magnitude
  - RTT
    - LAN 1ms
    - Internet/WAN 100ms
  - Transmission time

Type of line	Min. Tx Time (1 64-bytes packet)	Min. Tx time (1 1500-bytes packet)	Min Tx time (1Mbyte)
10Mbps	0.05ms	1.2ms	800ms
100Mbps	0.005ms	0.1ms	80ms
1Gbps	0.5us	0.01ms	8ms

Minimum time for typical Interactions

⇒ Minimum time for one-way notification datagram: RTT/2

⇒ Minimum time for opening TCP connection and sending notification: 1.5 RTT

⇒ Minimum time for UDP request/response: 1 RTT

⇒ Minimum time for opening TCP connection and performing request/response interaction: 2 RTT

DS Main General Requirements: Distribution Transparency

- Meaning:
  - Internal details of a DS should be kept hidden to its users
    - Where data and computation are located and how they are moved
    - Whether and where data are replicated
    - How data are internally stored/accessed/shared by the DS elements
    - Failures of elements

requires

requires

performance

reliability and fault-tolerance

- Not possible to achieve fully:
    - Latency depends on location
    - Failures may prevent transparency
- Not always convenient or desired
  - Transparency/performance tradeoff
  - Location-aware or context-aware services

DS Main General Requirements: Dynamic Membership

- DSs are generally required to support Dynamic Membership
  - there must be ways to add/remove processes
  - Processes should be able to locate the other collection elements at runtime

DS Main General Requirements: Security

- Typical security properties:
  - No unauthorized access to data or services (confidentiality, access control)
  - No possibility of interference with the system behavior (integrity, availability)
- Generally formulated as "Not easy for an attacker with certain capabilities to..."

DS Main General Requirements: Scalability

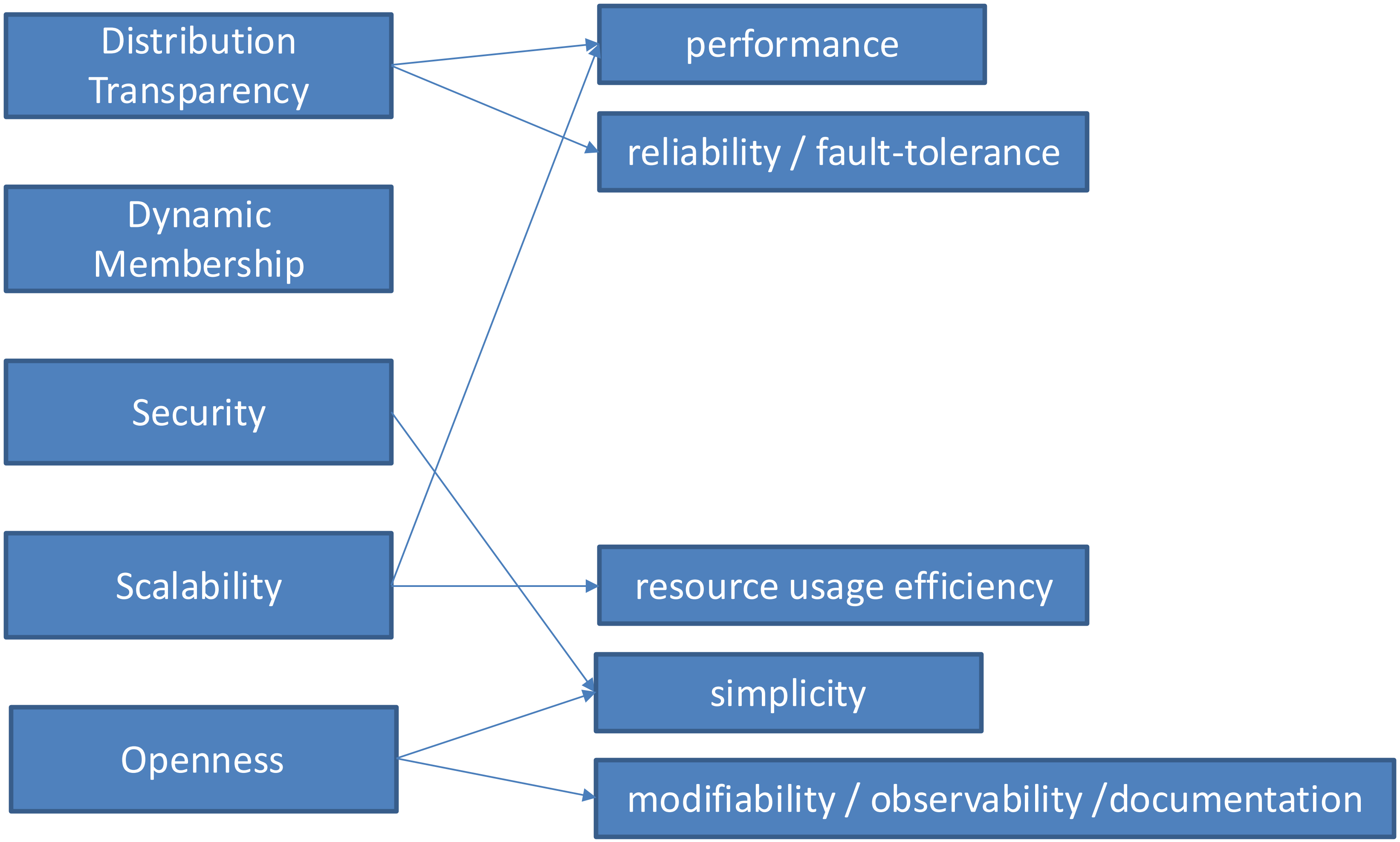
- Meaning:
  - "the extent to which a system adapts to an increase of its dimensions without losing its performance significantly" (Tanenbaum)
- For DSs, different types of scalability:
  - Size scalability (w.r.t. number of users and resources)
  - Geographical scalability (w.r.t. distance among users / components)
  - Administrative scalability (w.r.t. number of organizational units)
- Main Challenges:
  - Resource limitations $\xrightarrow{\text{requires}}$ $\boxed{\text{resource usage efficiency}}$
  - Latency and Reliability depend on distance and cannot be reduced arbitrarily

DS Main General Requirements: Openness

- Meaning:
  - The extent to which a system offers components that can easily be used by, or integrated into other systems (of different types)

Related requirements:
simplicity, modifiability, observability, documentation

DS Main General Requirements: Summary

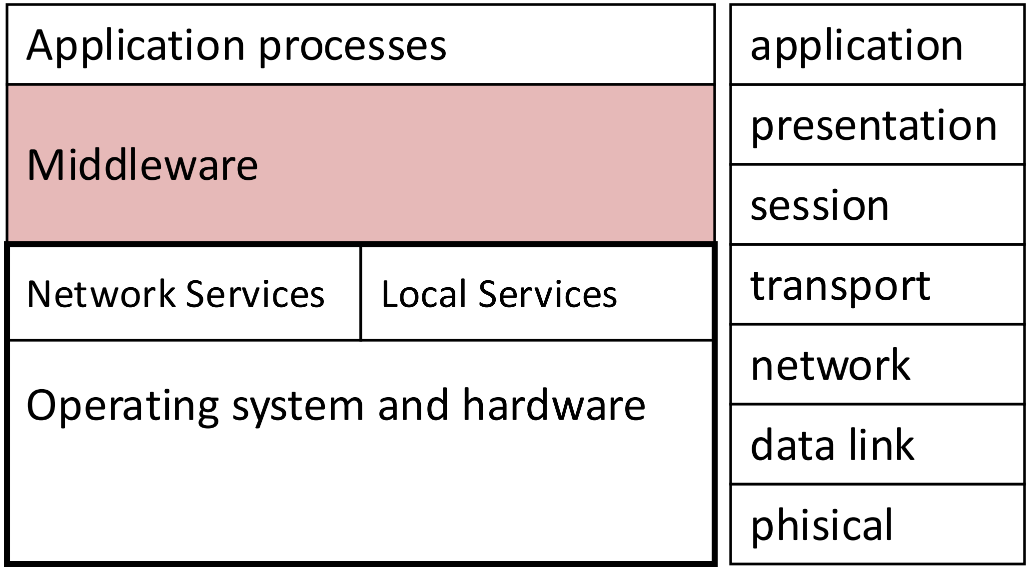


DS: General Software Architecture

- The software architecture of a DS
  - is based on the layered (OSI) model
  - is made of a collection of processes
    - physically executed on possibly different network hosts
    - interacting according to a stack of protocols
- The OSI layers relevant for the DS programmer are the application-oriented layers (5,6,7)
- A common way of managing the complexity of the many requirements is to build applications on top of a software layer called middleware

Middleware

- Stands between applications and O.S.
- Solves several challenging requirements of distributed software



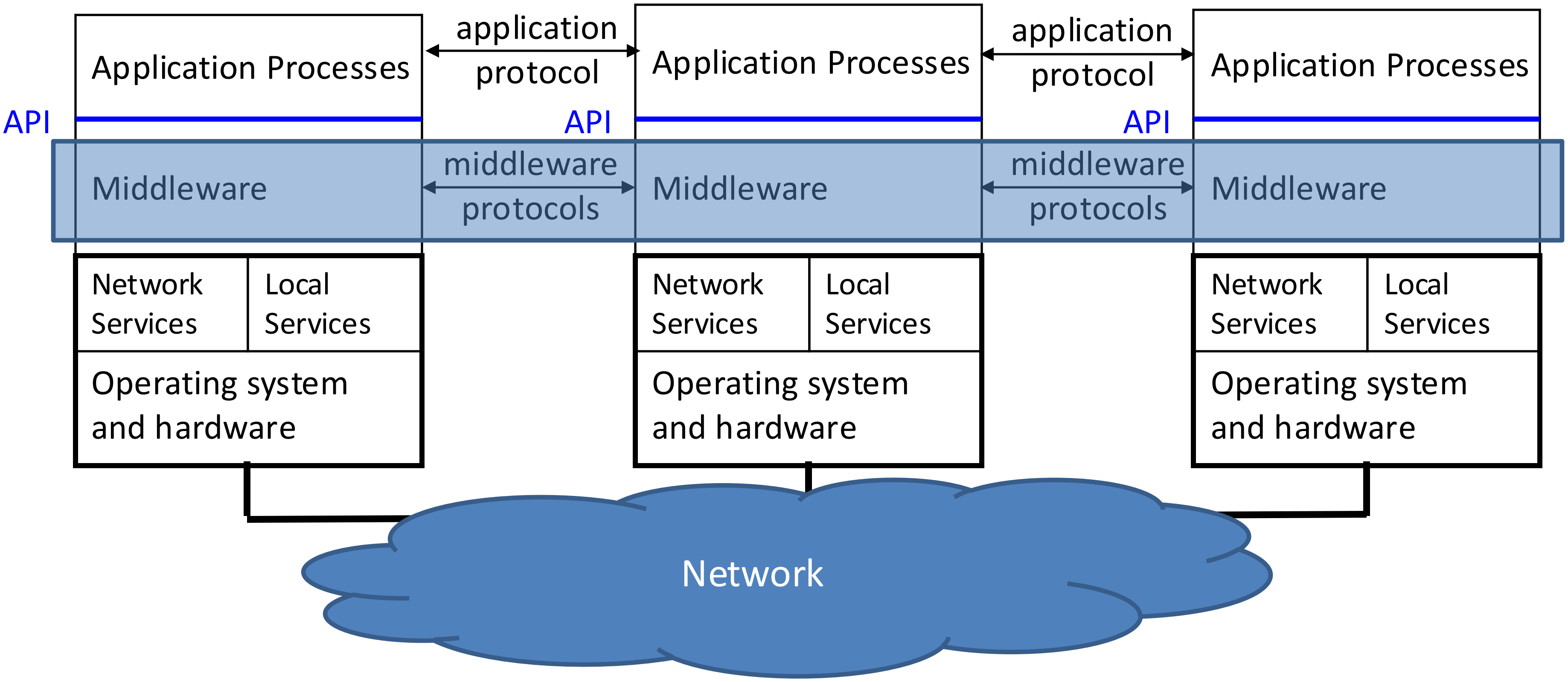
Middleware

- Provides **business-unaware** services for coordination and communication among remote processes
- **hides** communication through the network, process and host heterogeneity, security issues, etc.
- Examples of software that can be classified as middleware:
  - web browser
  - database driver
- Examples of software that cannot be classified as middleware:
  - airline reservation system (business aware)

Typical Middleware Services

- **Interaction Services**:
  - information exchange, connection management, session management, deadlock avoidance mechanisms, etc.
- **Services for accessing specific applications**:
  - database access, transaction processing, distributed object management
- **Management, Control and Administration Services**:
  - directory, security, performance monitoring, etc.

Middleware Layer



DS: Main Design Goals/Strategies

- Limit interactions (distribution transparency, scalability)
  - "the best application performance is obtained by not using the network" (R. Fielding)
- Use middleware solutions as far as possible (do not re-invent the wheel) => **reuse**
- Build applications relying only on APIs (not on the middleware internals) => **information hiding**

Strategies for Scalability

- Scaling up
  - improving the **capacity** of the computing/storage/communication elements (e.g., replace CPUs with more powerful ones)
- Scaling out
  - modifying the way the system is constructed or operates (e.g., increase the **number** of deployed machines in the system)

- Scaling up can be applied only to a limited extent
- All world-size scaling DSs employ scaling out

Main Scaling Out Techniques

- Partitioning and Work Distribution
  - Split the work to be done into small pieces and assign them to different processes (increasing their number when needed). Examples: the Web, the DNS
- Replication
  - Replicate processes and data. Examples: caching (data), web server clustering (processes)
- Communication Latencies Hiding/Limitation
  - use asynchronous communication (instead of synchronous)
  - Process data where they are (rather than moving them)

Strategies for Openness

- Use standard (or documented/public) protocols and APIs
- Leave Implementations totally free (apart from adherence to agreed protocols and APIs)

  => neutral interfaces

  => **interoperability** of different implementations

  relies on using the same protocols

  => **portability** of implementations

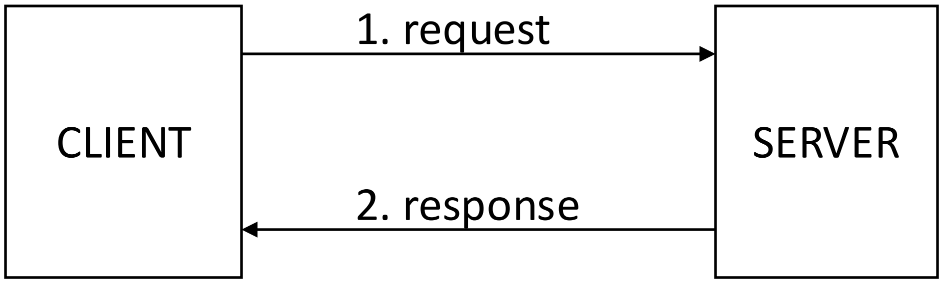
  relies on using the same APIs
- Composability and extensibility
  - Make systems composed of several simple and easily replaceable elements

Classification of DS Architectures

- Broad Classification
  - Client-server (centralized organization)
  - Peer-to-Peer (fully decentralized organization)
  - Hybrid

The Client-Server (C/S) Interaction Model

- Currently this is the most used interaction model
- Each interaction occurs between two processes: one plays the **client role** and the other one plays the **server role**.
- Each interaction is based on a message exchange: the client sends a **request** and the server sends back a **response**

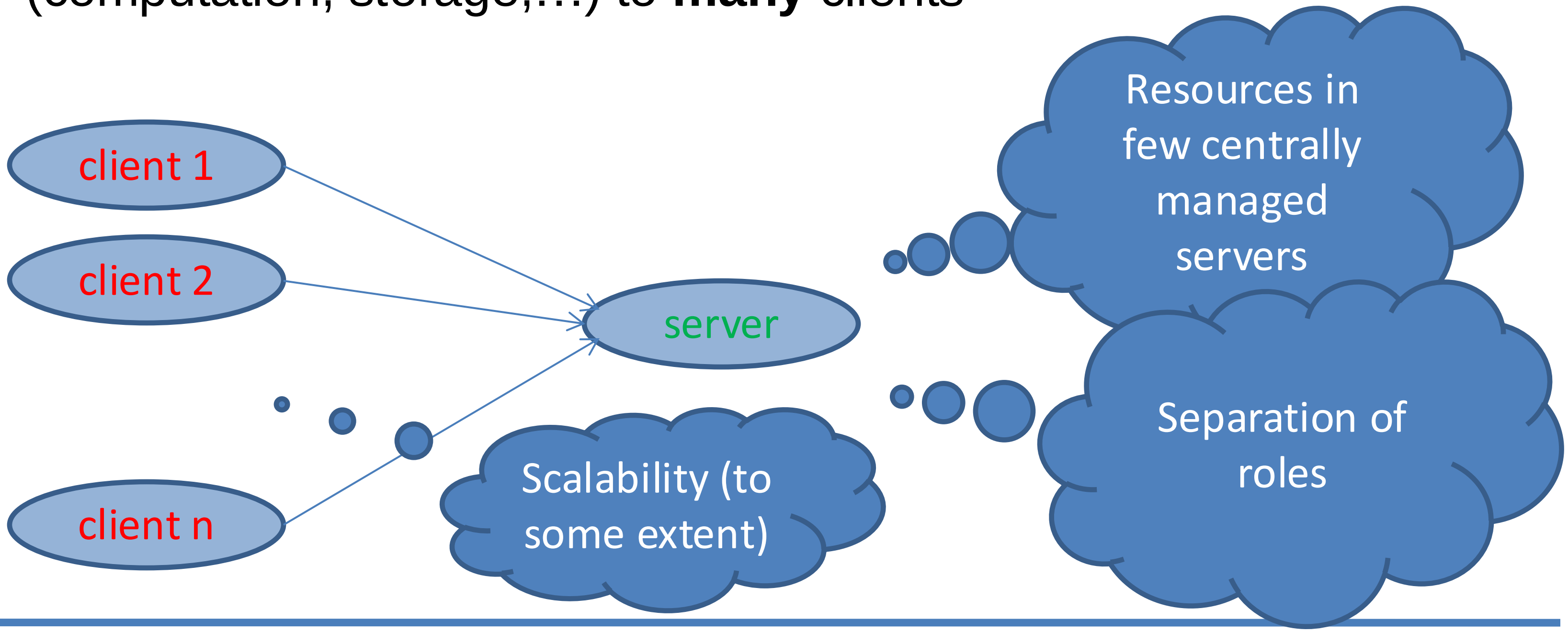


The Client-Server (C/S) Interaction Model

- Note that the role (client or server) refers to how the process plays a single interaction
- A single process can play the client role in some interactions and the server role in other interactions

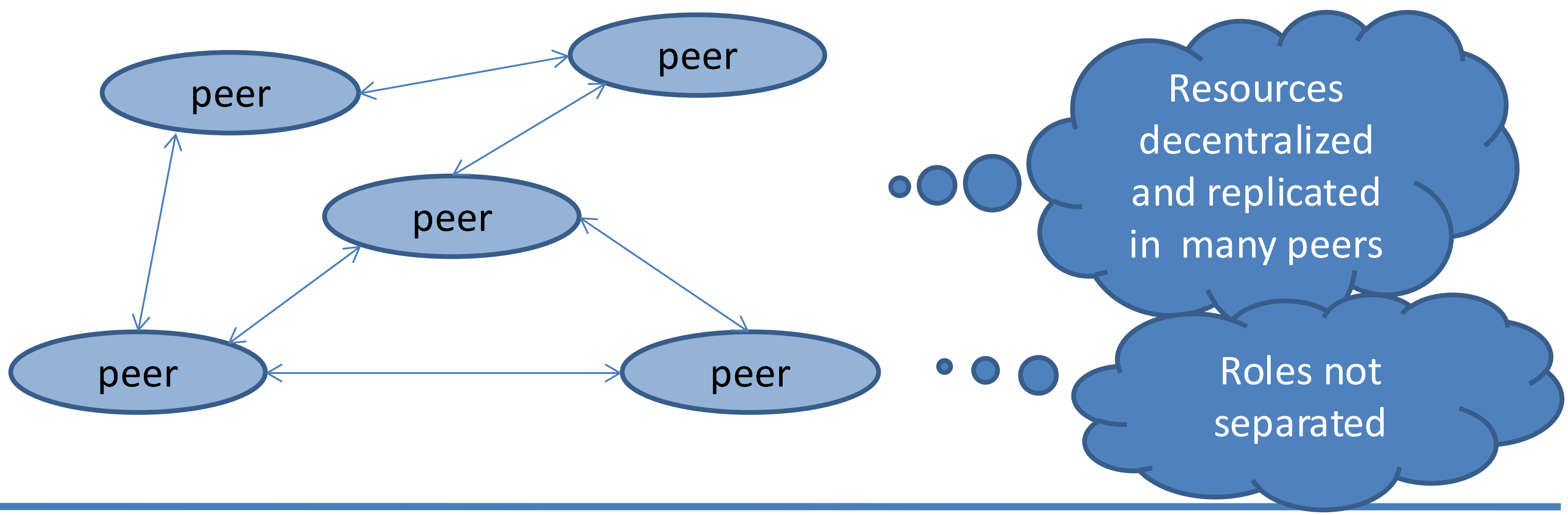
Client-Server (C/S) Centralized Architectures

- A client-server architecture is one where processes are divided into two classes:
  - client processes that **only** play the client role
  - server processes that play the server role and offer some service (computation, storage,…) to **many** clients



Peer-to-peer (P2P) Decentralized Architectures

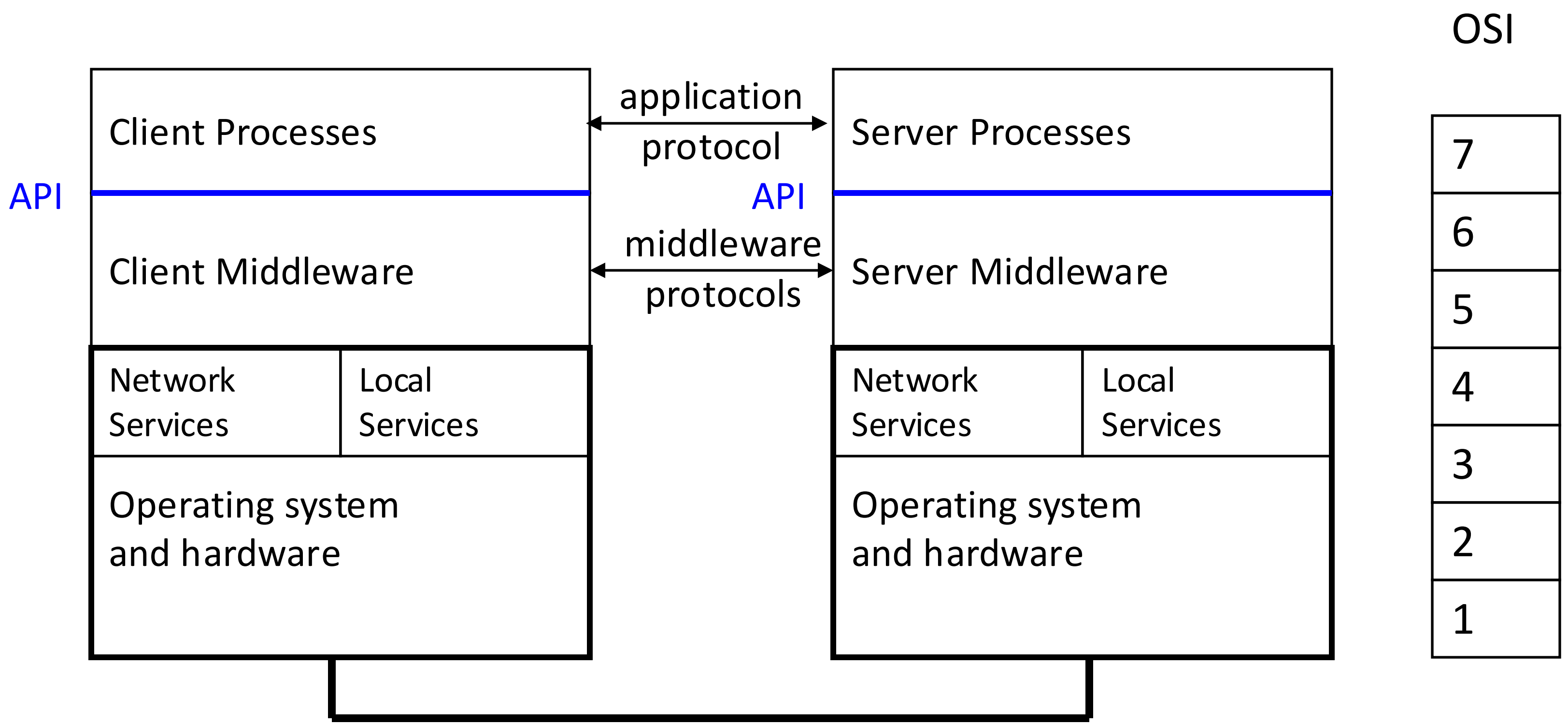
- A Peer-to-Peer architecture is one where all processes are **identical peers** that can play both the client and the server roles
- Each peer can start a C/S interaction with any other peer it knows, at any time



Discussion: C/S versus P2P

- in C/S, server processes may become bottlenecks: they must run on powerful, reliable hosts
  - server processes may become congested with requests
  - server processes can be single points of failure
- in P2P, less control is possible
  - information is distributed and often replicated
  - security is more difficult to provide
- P2P can provide cheap reliability and performance
- C/S is a simple and well-understood organization (separation of roles helps in making things simple)

Anatomy of C/S Software



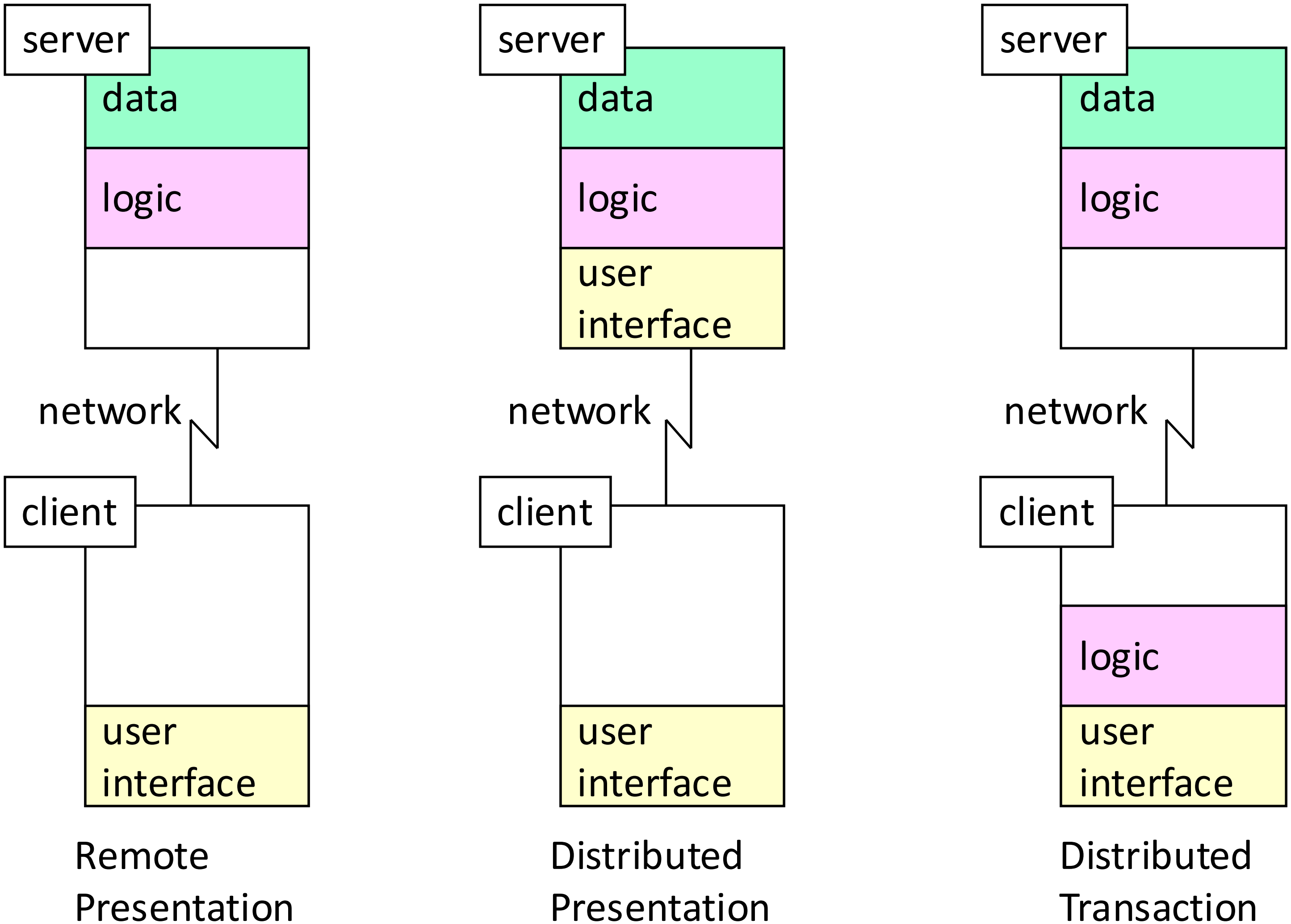
C/S Vertical Distribution: Application Tiers and Physical Tiers

- Each application can be **logically** divided into three main parts (logical tiers):
  - user interface tier
  - processing tier
  - data tier
- In a C/S system the logical tiers of an application are mapped onto different processes running on possibly different hosts (physical tiers)
- Different mappings are possible between logical and physical tiers

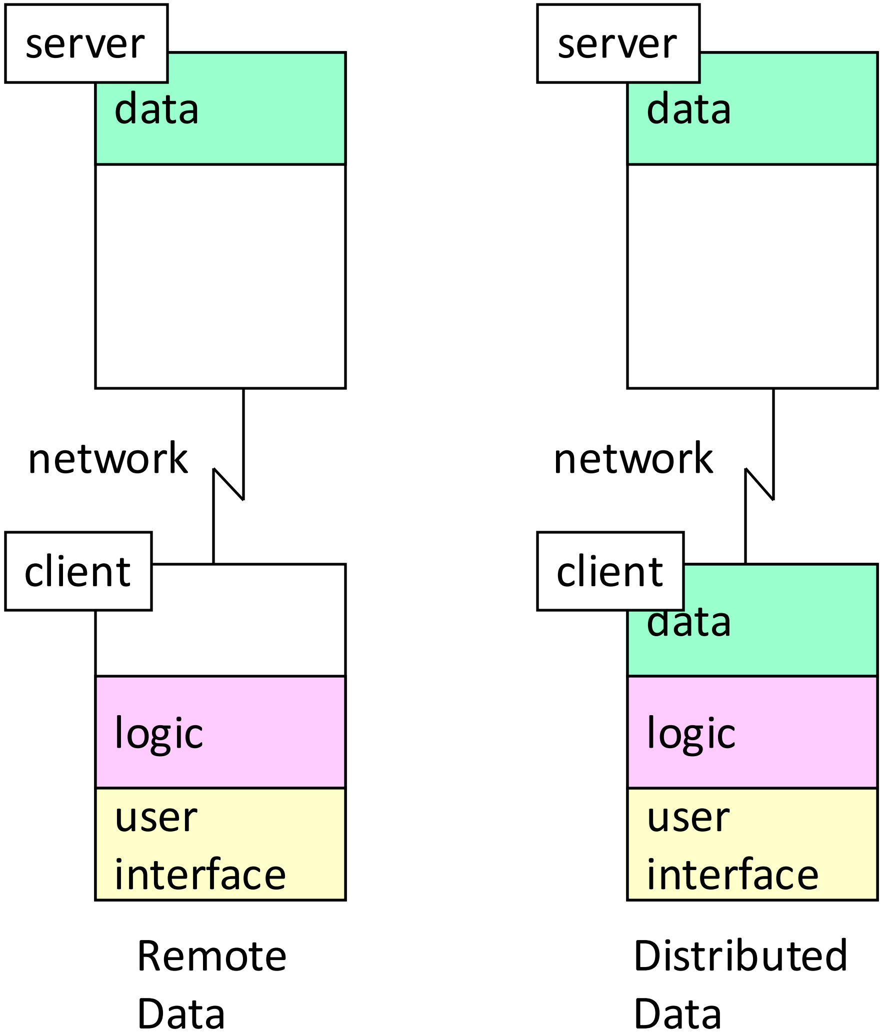
C/S 2-tier Architectures

- Only 2 physical tiers: one client tier (also known as front end) and one server tier (also known as back end)
- This is the classical architecture, typical of the first generation of C/S systems.
- Different configurations are possible, according to how logical tiers are mapped onto physical tiers.

C/S 2-tier Architectures Classification (Gartner Group)

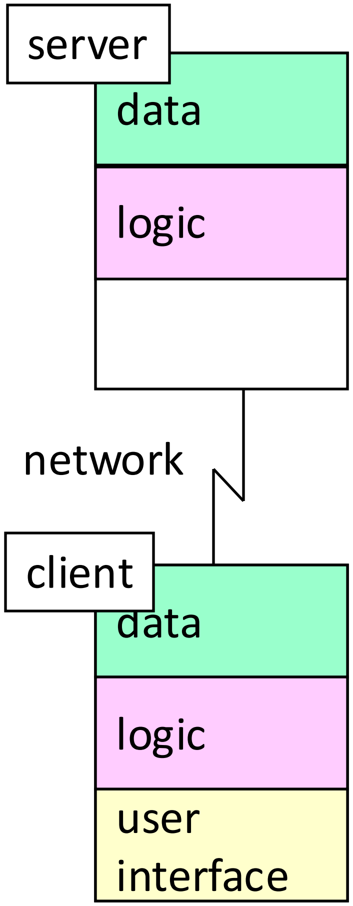


C/S 2-tier Architectures Classification (Gartner Group)



Other possible architectures exist

Example:

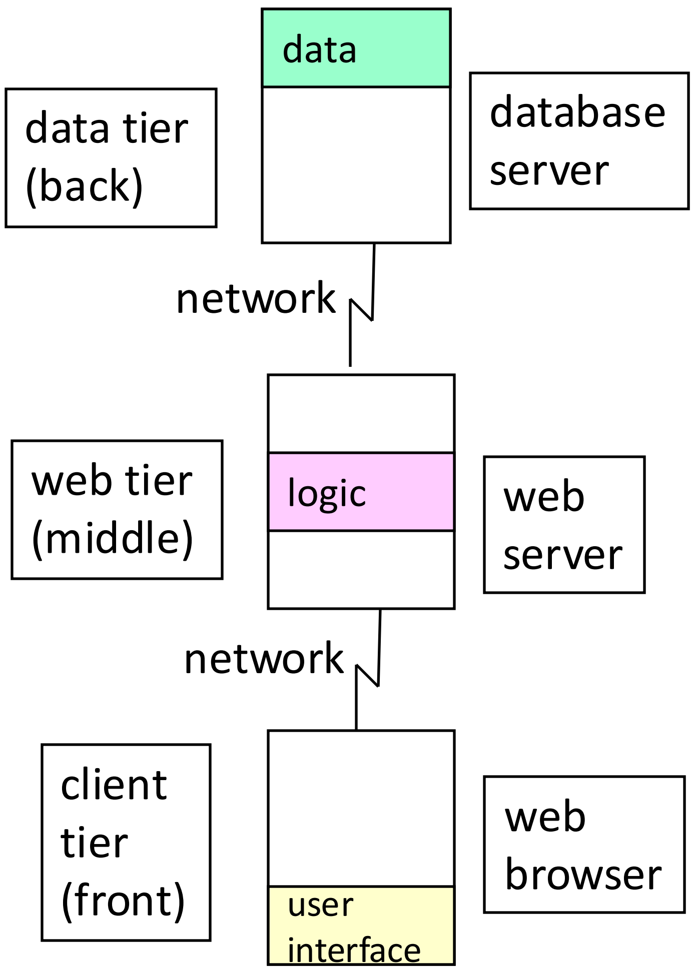


Computation offloading

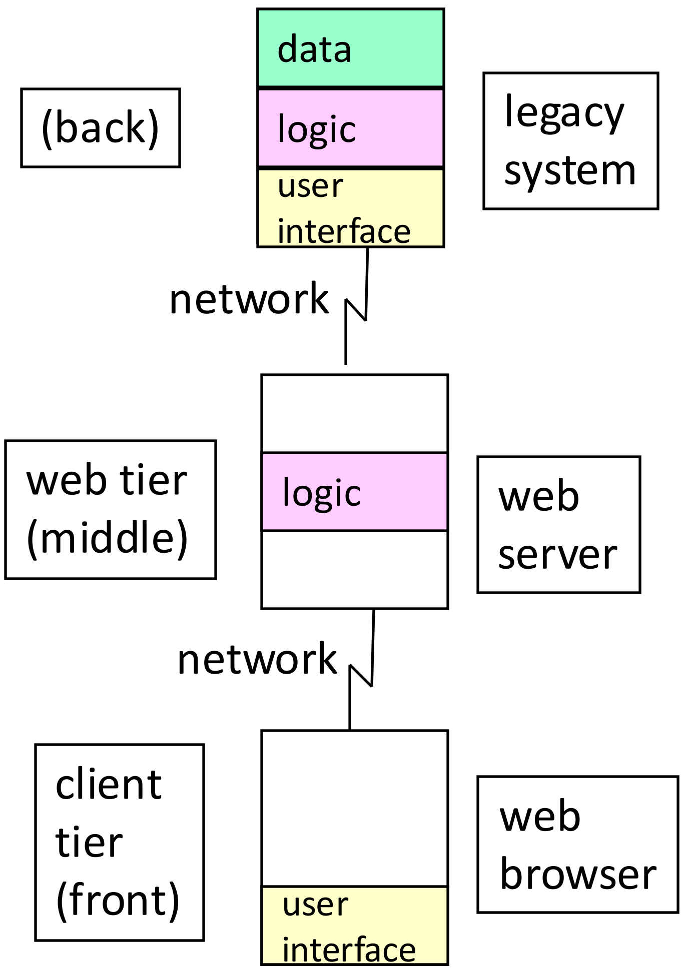
C/S 3-tier Architectures

- C/S 2-tier architectures have **scalability** and **flexibility** limitations
- These limitations can be overcome by introducing an intermediate physical tier (middle machine), with various possible functions:
  - workload balancing on different back ends
  - filtering (e.g. web application firewalls)
  - protocol conversion, access to legacy systems (gateways)
- Another possibility is to add tiers to
  - split/compose services
  - delegate (and decouple) specific application components (e.g. data)

C/S 3-tier Architectures: Examples



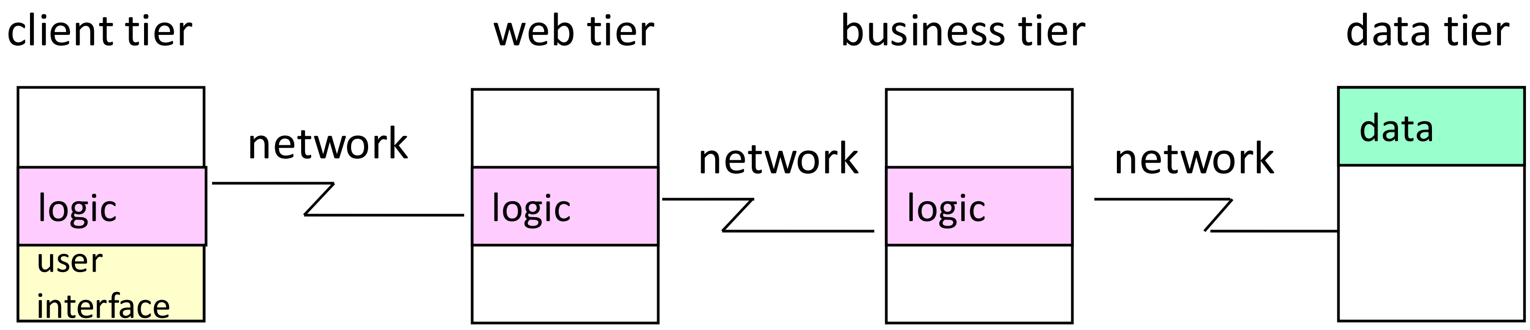
Web gateway



Web access to Legacy System

C/S multi-tier Architectures

- For even further flexibility and for complex systems, the number of physical tiers can be further increased
- Example: 4-tier architecture



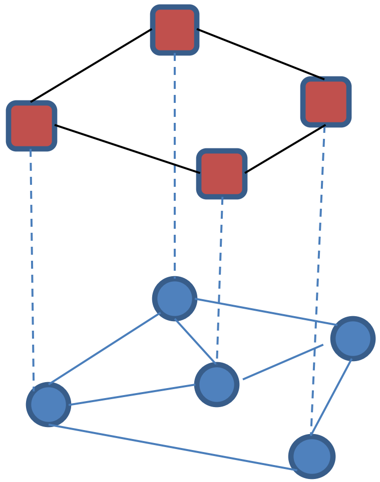
P2P Horizontal Distribution

- In P2P systems, all peers perform the same tasks, but on their own share of the complete data set
- Sometimes the data set is partitioned, sometimes data are replicated in order to increase reliability and performance

P2P Architectures: Overlay Network

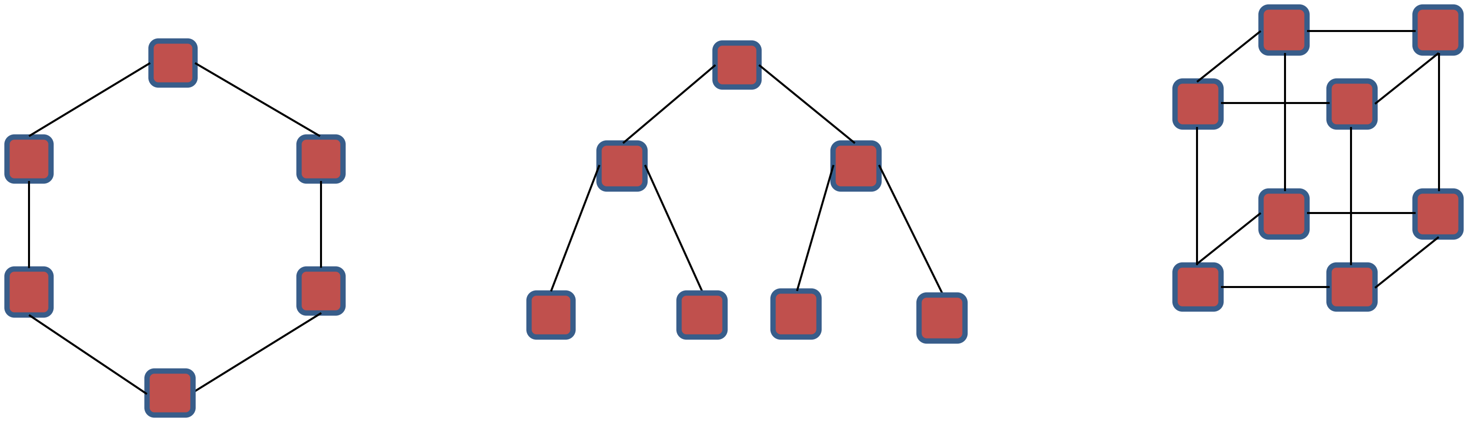
- A P2P system is built by its peers which create an overlay network:
  - Each peer can communicate only with some other peers through virtual channels
  - One key aspect of each P2P architecture is how the overlay network is created and managed

- Two main types of overlay networks are used:
  - Structured
  - Unstructured



Structured P2P Systems

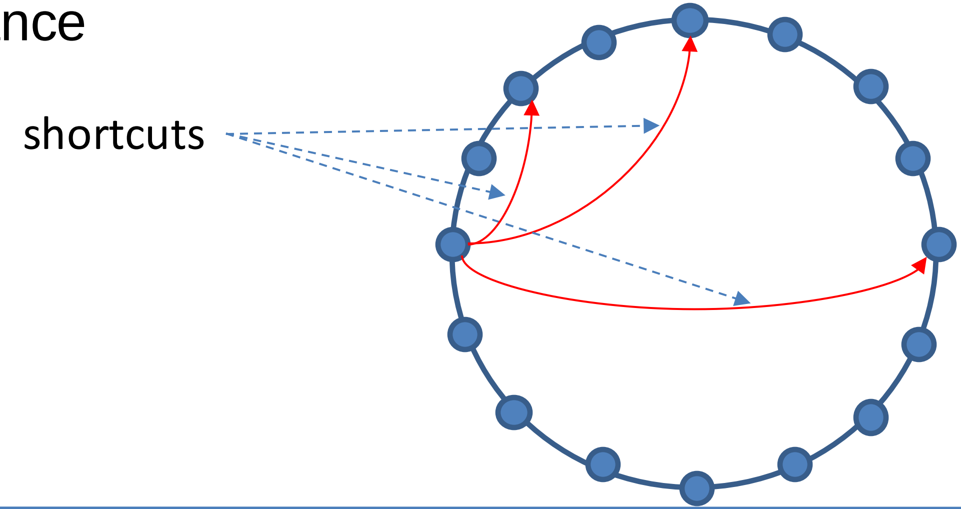
- The overlay is built with a deterministic, regular topology (e.g. ring, tree, or grid)



- Topology-specific routing algorithms are used to reach specific peers

Example: Chord

- One of the practical p2p organizations for implementing a distributed hash table
  - Structured ring topology
  - A distributed algorithm is used to reach the peer responsible for a hash key in O(log(N)) time
  - Open to dynamic joining and leaving
  - Fault tolerance

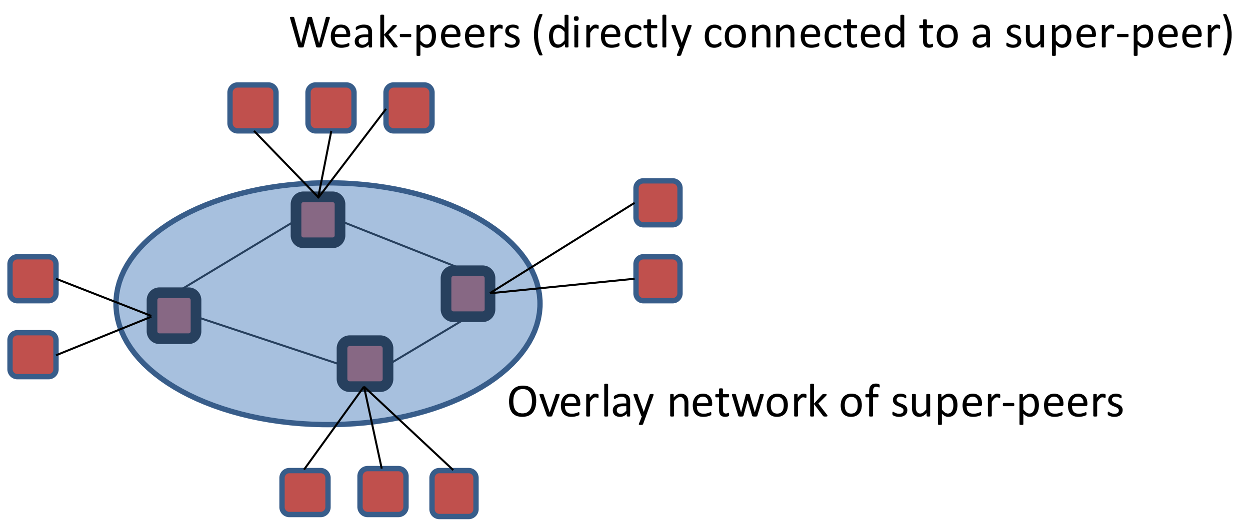


Unstructured P2P Systems

- Each peer connects to some other peers in a more or less random way (e.g. discovery of nearby peers)
- Different strategies can be used to reach the other peers
  - Flooding
  - Random walks
  - Policy-based search

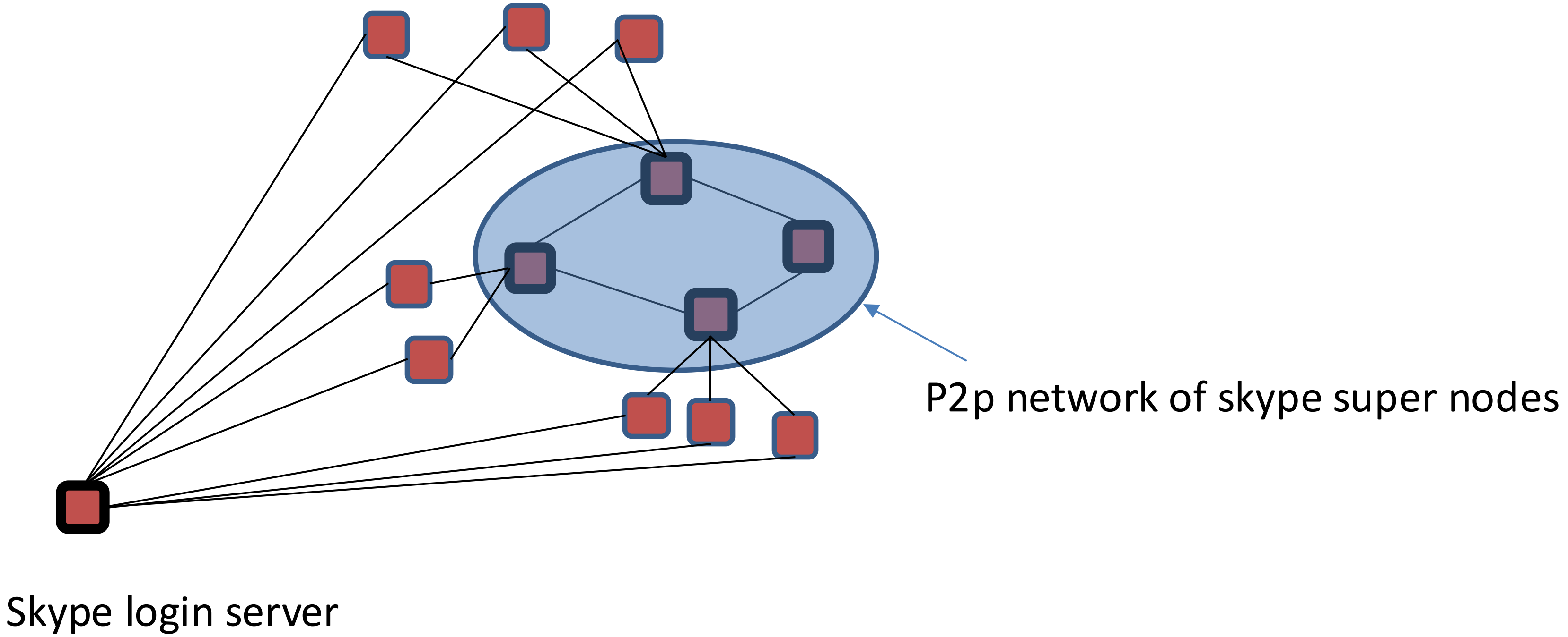
Hierarchical and Hybrid P2P Systems

- Hierarchical: Peers are not all the same:
  - Normal (weak) peers
  - Super-peers

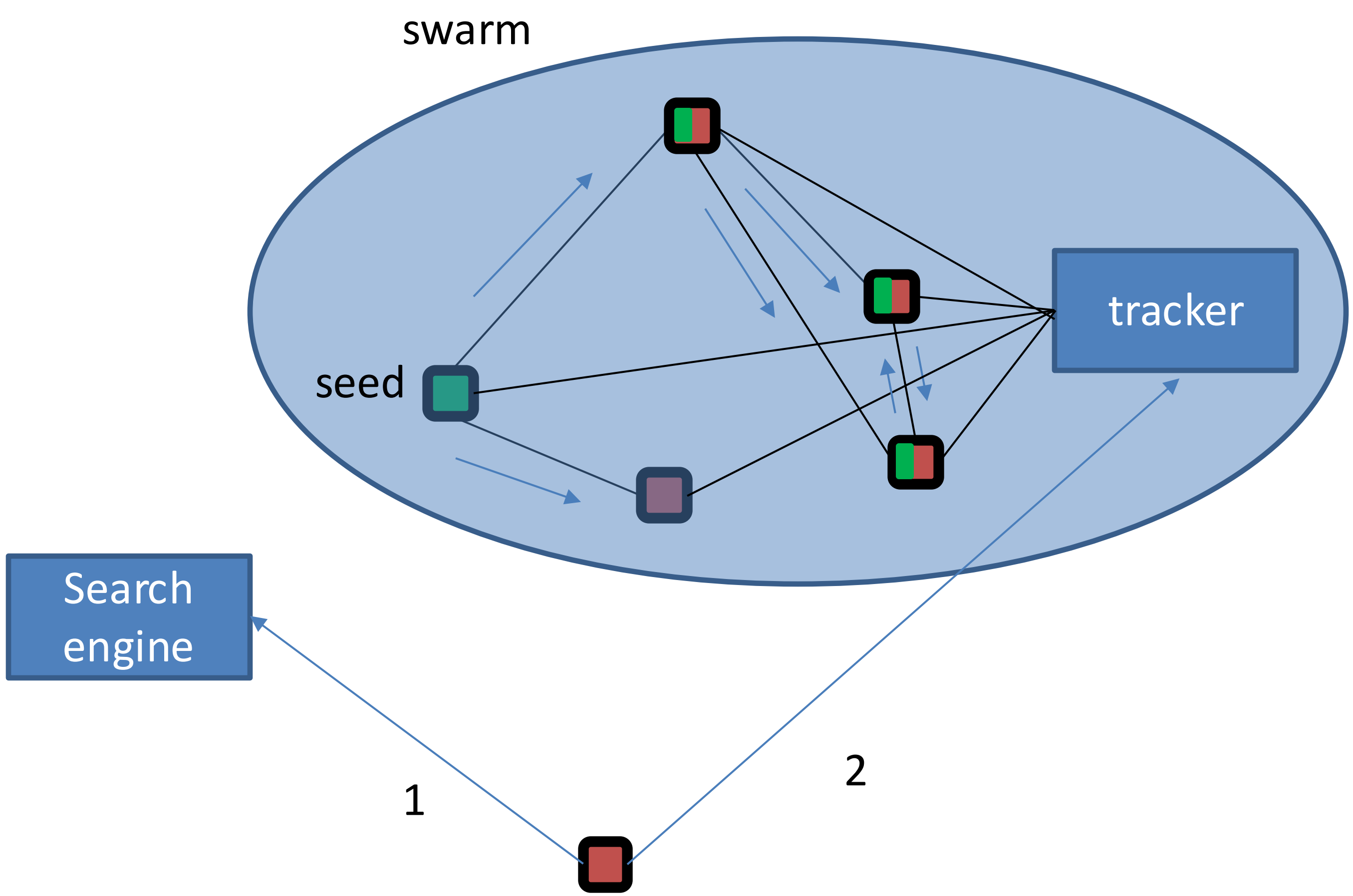


- Hybrid: combine C/S and P2P architectures together

Example: Skype



Other Example: BitTorrent



Abstractions for DSs

- Remote Procedure Calls (RPC)
- Message (or Event) Passing
  - Message-Oriented Middleware (MOM)
- Shared Persistent Data
  - Shared Databases, Tuple Spaces, Blockchains

Different forms of coordination (coupling):

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct (RPC)	Mailbox-based (MOM)
Referentially decoupled	Event-based (MOM)	Shared data

Higher-Level Abstractions

- The abstractions discussed so far are implemented directly by some middleware/protocol/API:
  - Examples: gRPC, Java Messaging System (JMS), JavaSpaces
- In some cases, Higher-level Abstractions are also used:
  - Distributed Object Systems
  - Distributed Services and Web Services
  - Distributed Resources
  - Publish-Subscribe Systems
  - Message Queues

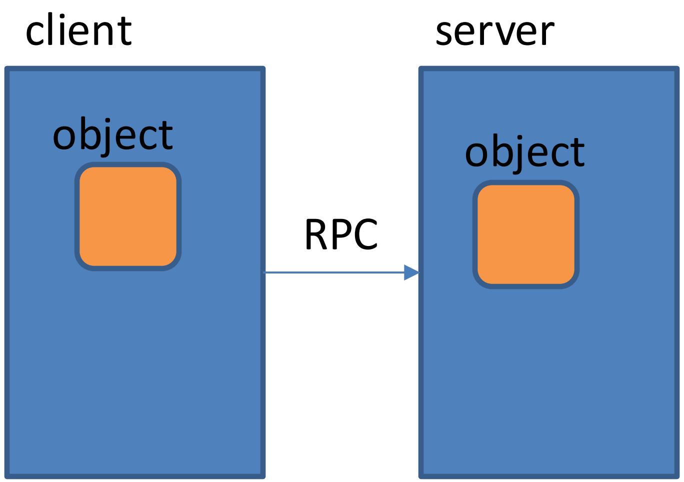
RPC-based

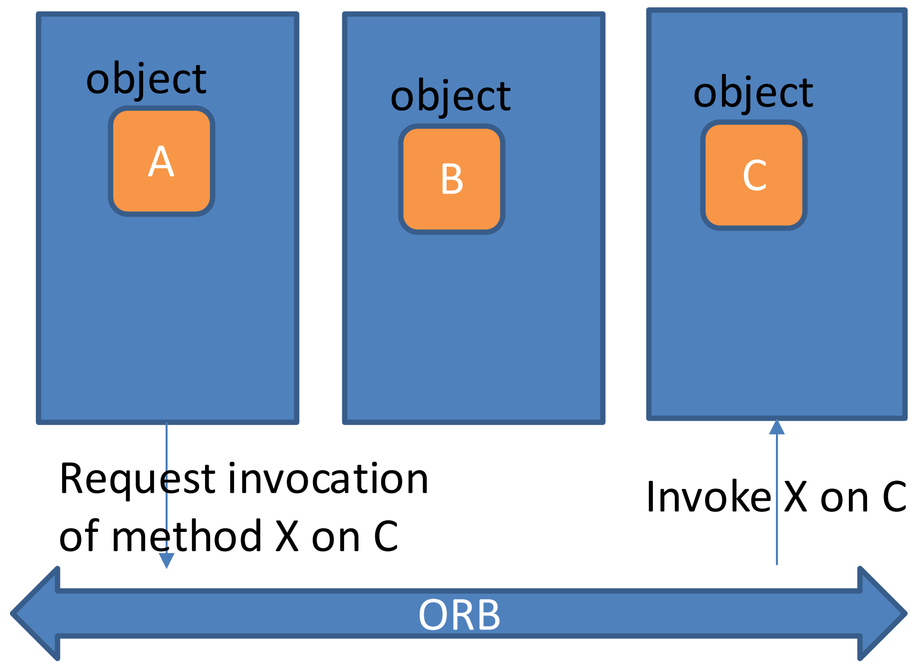
MOM-based

Distributed Object Systems

- Main idea: transfer the object model into a distributed environment
  - The objects of a single application program can “live” in **multiple hosts** connected by a network, but each object can be used as though it were local
  - Remote Method Invocations (RPC)
  - Examples:
    - Java RMI
    - OMG CORBA

Direct vs Broker Architectures





- Example: RMI

Example: CORBA

Distributed Services

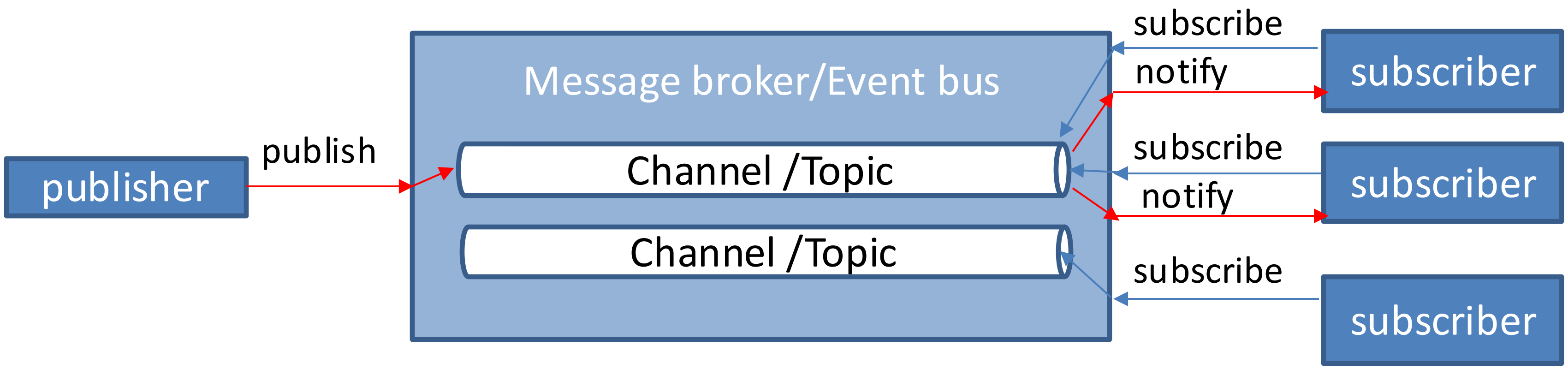
- Abstraction similar to distributed objects
- Main differences:
  - Services have **larger grain** than objects
  - Services are **autonomous** and **long-living** entities
  - Services may be made available for use by different clients from **different organizations**
  - Services enable **service composition**
- Example: SOA, web services

Distributed Resources

- Abstraction similar to distributed services, introduced with REST and HTTP
  - Resource: data entity with state on which fixed operations (CRUD) are possible

Publish-Subscribe

- C/S Paradigm that achieves referential decoupling
  - Processes do not know each other
  - Clients can only perform two elementary operations:
    - Publish: generate message/ event
    - Subscribe: express interest for a class of messages/events
  - Servers (message broker/event bus)
    - take care of notifying each published message to all up and running subscribers



Publish-Subscribe

- The Message Broker/Event Bus can be
  - Centralized
    - Example: MQTT
  - Decentralized
    - Example: Data Distribution Service (DDS)
  - Cloud-based
    - Example: Amazon SNS (part of AWS)

Message Queues

- Distributed implementation of a queue
- Clients can queue and dequeue messages
- Messages in a queue remain stored until someone dequeues them (asynchronous interaction)

Examples

- Java/Jakarta Message Service (JMS)
  - Java API for MOM (supports both publish-subscribe and MQ)
- Advanced Message Queueing Protocol (AMQP)
  - open standard protocol for MOM that can be used to implement various models (publish-subscribe, MQ)
  - Examples of implementations: RabbitMQ, StormMQ, Apache Qpid
- Apache Kafka
  - distributed event streaming platform that supports the publish-subscribe paradigm
- and many more: IBM MQ, Oracle AQ,...

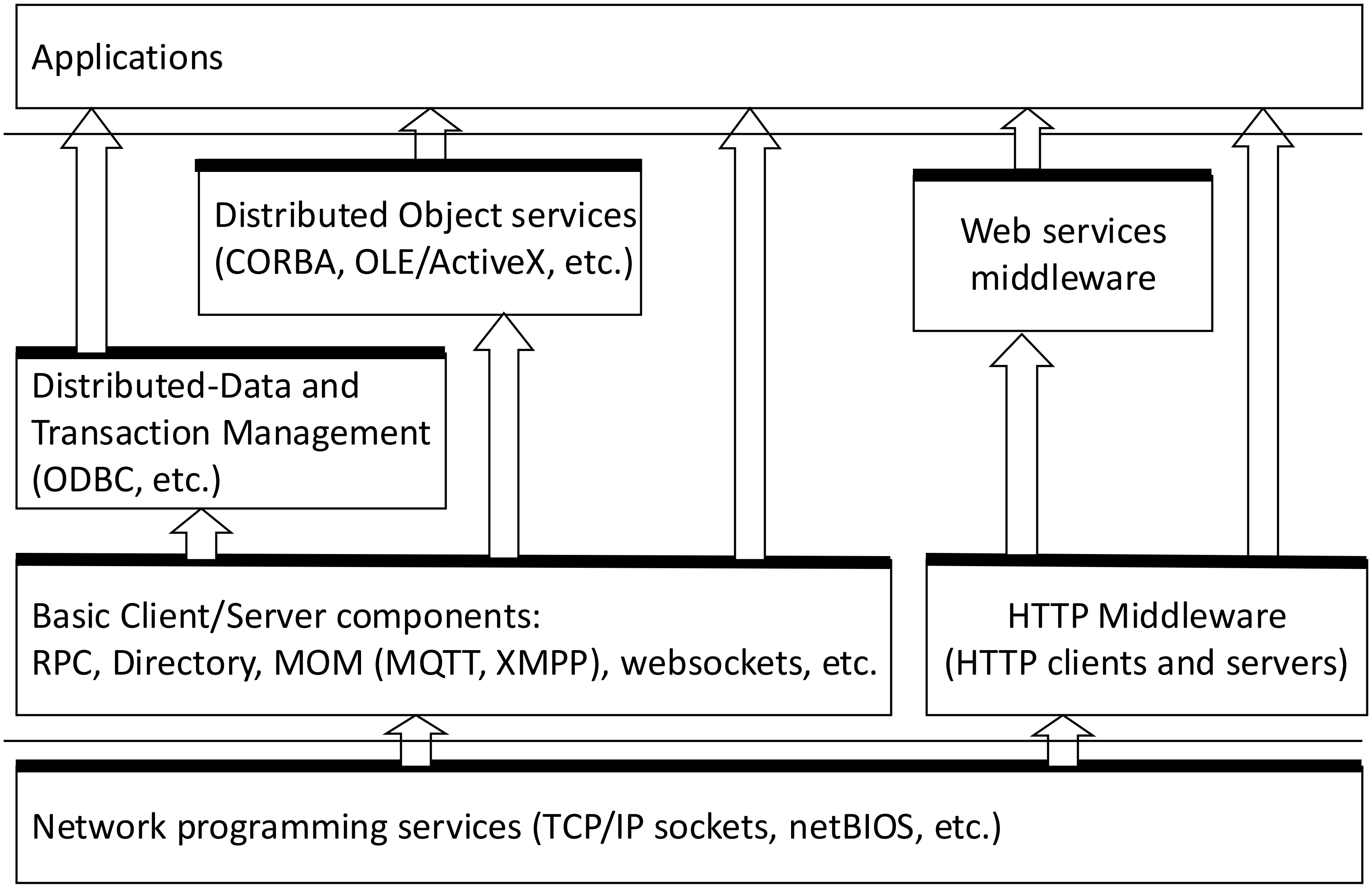
Lower-Level Communication Abstractions

- Sockets
  - Generic low-level communication abstractions
  - Offer elementary communication primitives
  - higher-level protocols must be user-defined on top
  - Maximum Flexibility: can be used to implement user-defined abstractions and protocols
- Examples:
  - TCP/IP Sockets
    - de-facto standard for accessing layer 4-3-2 protocols
  - WebSockets
    - Interface for managing TCP-based full-duplex channels
    - Based on a standard HTTP-compatible protocol

Middleware Architecture: Components

- Middleware can be organized as a set of **components**, each one providing some services accessed by an **API**
- Services offered by one component can be used by other components, in order to build more complex services (layering)
- The most **basic services** used by all middleware components are those providing typical network and transport layer communication facilities.

Middleware Components: Examples



Middleware Architecture: Frameworks

- Different kinds of middleware components can be combined into **frameworks** that
  - provide a multiplicity of services (generally all the services needed by typical distributed applications)
  - may include their own development tools
- Framework examples:
  - Oracle Java EE / Eclipse Jakarta EE
  - Microsoft .NET
  - Java Spring
  - Amazon AWS