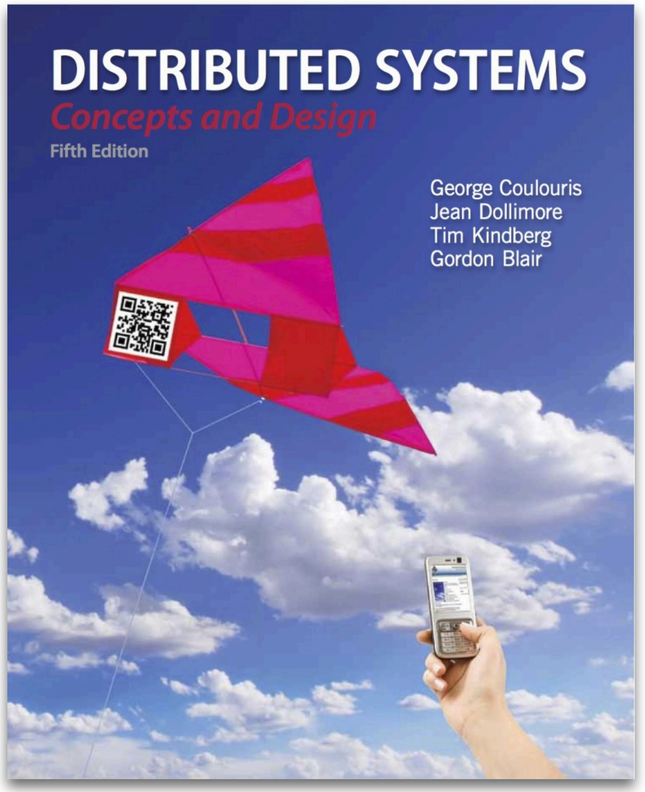
Distributed Systems 2023-2024: Introduction Part 2 - System models

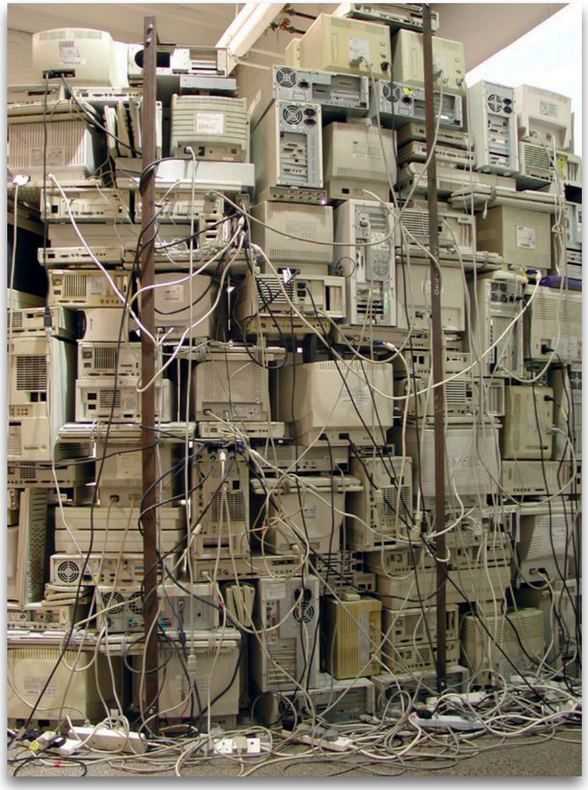
Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
September 2023

Learning resources

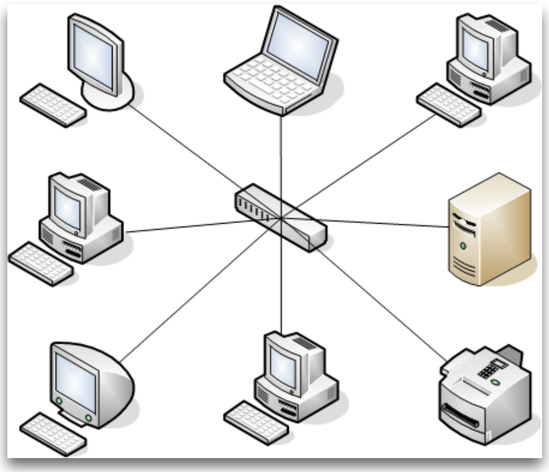
- Textbook, chapter 2
  - Introduction
  - Physical models
  - Architectural models
  - (Fundamental models) (see later lecture on coordination)
  - Summary



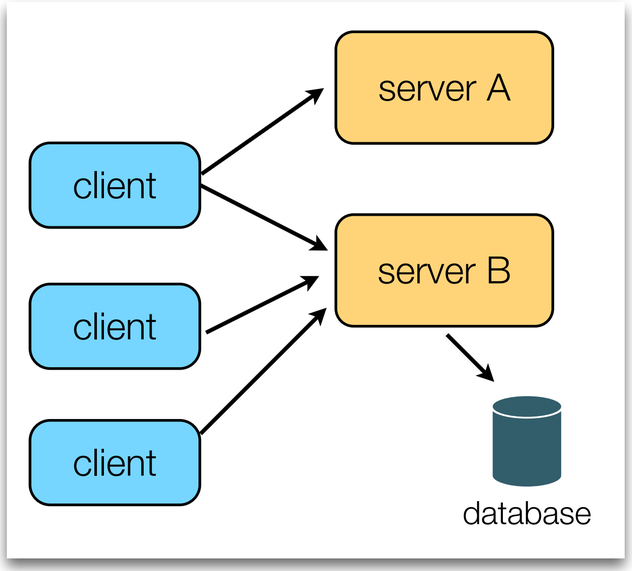
Models of distributed systems



abstract



abstract



Real-world systems

Messy and complex

Physical models

Consider the kinds of devices and how they're interconnected, without regard for specific technologies

Architectural models

Consider the computational and communication tasks performed by the system's computational elements

Physical models

- Baseline model (chapter 1): interconnected nodes, hardware and software modules located at networked computers; these computers communicate and coordinate by message passing.

- Refinements can be based on historical evolution of computing and networking devices

- See also the examples in part 1 / chapter 1

Physical models

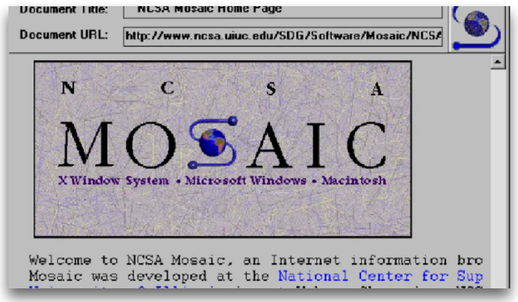
‘80s-‘90s

Local area networks
Workstations
Early internet (dial-up)

Dial-up modem

‘90s-‘00s

Wireless networks
Desktops, laptops
Broadband and WWW



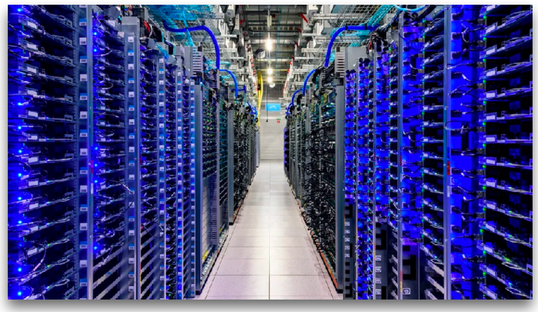
NSCA Mosaic, the first browser (1993)

Cable modem

‘00s-‘10s

Bluetooth, 3G networks
Smartphones, IoT devices
Datacenters and Cloud

iPhone launch in 2007



Large-scale datacenters

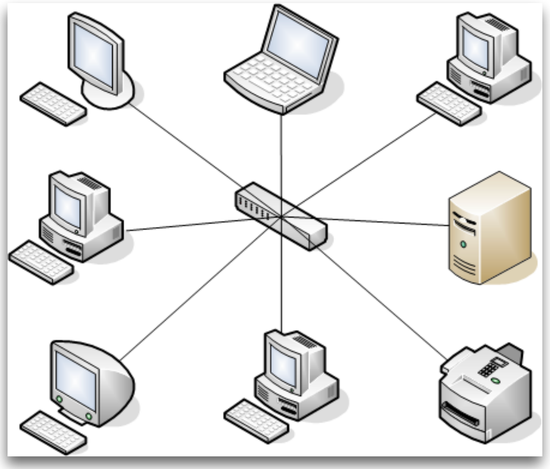
2020

Bluetooth low energy, 5G networks
Smartphones as fast as servers
Near-ubiquitous network coverage
Satellite computers, Ultra Wide Band

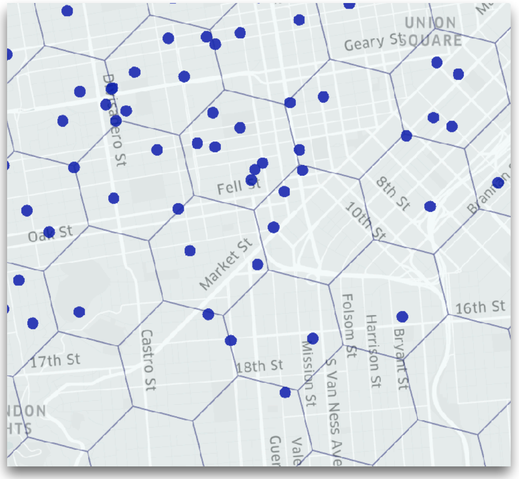
Ubiquitous video
streaming services

Starlink satellite network

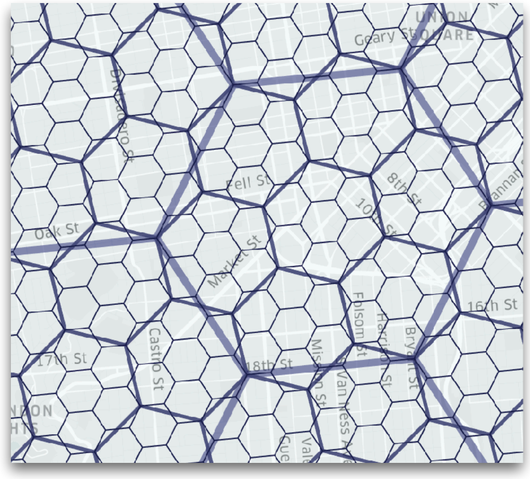
Physical models



‘90s-era Local Area Network (LAN) environment

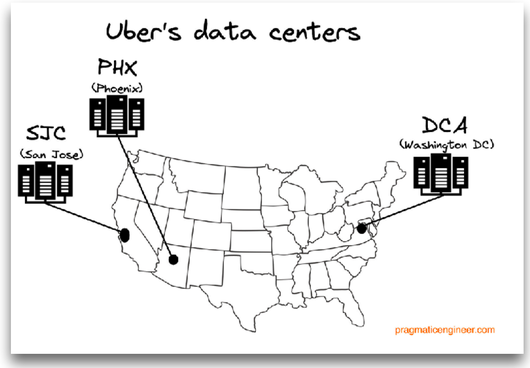


Mobile clients (riders and drivers)



Geo-spatial hierarchical index (H3)

Modern-day global “Webscale” environment
Example: Uber ride hailing service



Multiple cloud datacenters

Modern distributed systems are “systems of systems”

- Physical architectures are “systems of systems”, similar to the concept of “networks of networks”
- Examples:
  - Environmental management systems for flood prediction
  - Activity and cargo tracking in a large harbor
  - The vast compute & storage infrastructure behind all “Webscale” services (Google, Facebook, Apple, Microsoft, Amazon, …)

Architectural models

- Key Questions
  - 1. What are the computational **entities** in the system?
  - 2. How do they **communicate** with each other? (communication paradigm)
  - 3. What are their **roles** and responsibilities?
  - 4. What is their **placement**? (how are the entities mapped onto the physical model)?

1. Computational entities

- Traditional view (80's, 90's) - context: distributed systems was about programming a *system*; entities are
  - **Nodes** (often physical server machines, typically with fixed IP address)
  - **Processes**
- Evolution from the 90's onwards: .... About programming an *application* as well, entities are
  - **Objects** (e.g. Java RMI)
  - **Components** (e.g. Microsoft DCOM, Enterprise Javabeans, OSGi bundles)
- Evolution from the 00's onwards: .... About programming and *composition* and *virtualization*
  - **(Web) Services**, RESTful API services (see later)
  - **Containers** (e.g. Docker) and compositions of containers (e.g. Kubernetes Pods) (see Capita Selecta course)

Clarification: objects, components, services

- Objects (classes) versus Components:
  - Components specify not only their provided interfaces, but also the assumptions they make in terms of other components (interfaces) that must be present for a component to fulfil its function.
  - Dependencies are made explicit => more complete contract => enables third-party composition
- Components versus Web services:
  - Components often tied to a specific technology platform (e.g. Java & JVM, .NET framework)
  - Web services are intrinsically integrated into widely supported Web standards (HTTP, URL, XML, JSON). This improves decoupling and facilitates openness, heterogeneity, transparency

2. Communication Paradigms

- This will be the main subject of the next courses! (Textbook chapters 4, 5, 6)
- In brief, three major paradigms:
  - 1. **Inter-process communication** (reading/writing a byte stream shared between two processes)
    - Usually based on OS network programming APIs (e.g. UNIX sockets)
  - 2. **Remote invocation** (basic request/response interaction between two processes)
    - Remote procedure call (in procedural languages) or remote method invocation (in object-oriented languages)
  - 3. **Indirect Communication** (communication with a known or unknown group of processes)
    - Often by registering “Observers” or “Listeners” or “Callbacks” on an event source

(1) entities and (2) communication paradigms, overview (Fig 2.2)

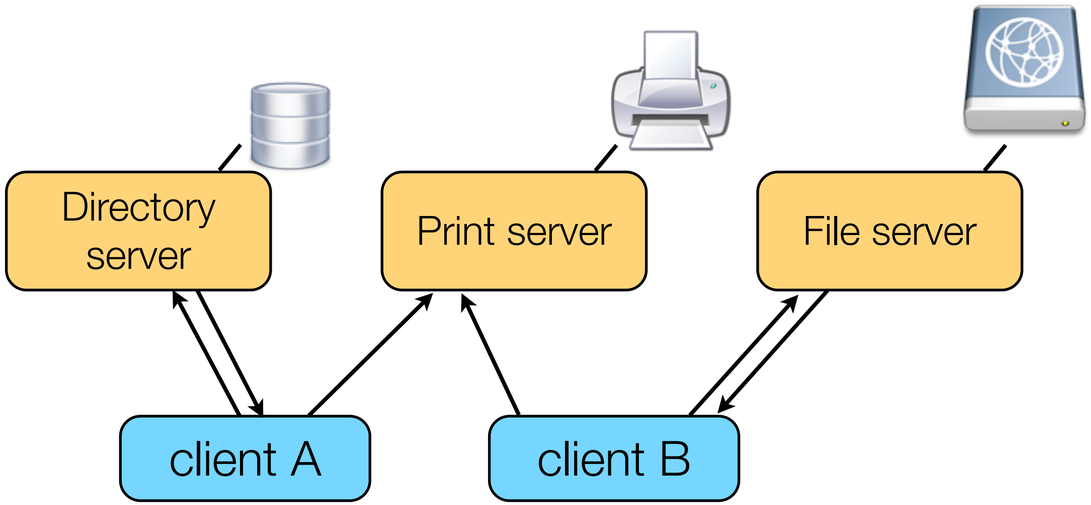
Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

3. Roles and responsibilities

- Two main paradigms:
  - Client-server
  - Peer-to-peer

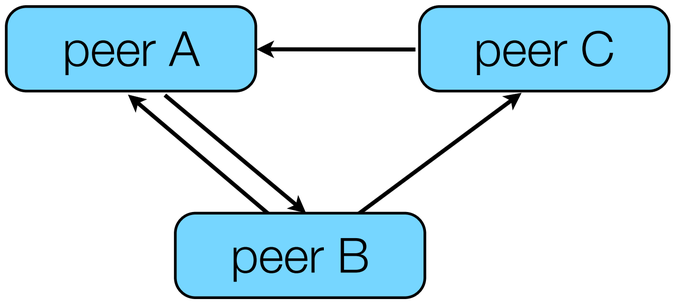
Client-server architecture

- Defines two roles for interacting entities:
  - **Client**: entity that is requesting a service (a task)
  - **Server**: entity that provides the service (performs the task)
- Client initiates the request, the server waits for requests and responds
- A server can be the client of another server!



Peer-to-peer (p2p) architecture

- Defines only one role for interacting entities:
  - **Peer**: acts as both a client and a server
- Each peer is assumed to have (roughly) equivalent capabilities or responsibilities (often runs the same program)
- Goal is **decentralize** service provisioning and management
  - No single point of failure
  - Better scaling and utilisation of resources (e.g. network bandwidth, cfr. Bittorrent file transfer)



4. Placement

- Where are the entities placed in terms of the actual physical, distributed infrastructure?
- Note: no universal solution – why?
- Some typical strategies:
  - **Map services to multiple servers** (example: on the Web each website is hosted on its own server)
  - **Caching** (example: a web browser caches previously downloaded resources to speed up page rendering on subsequent page visits)
  - **Mobile code** (example: JavaScript code downloaded from a web server and executed locally in the browser to make a webpage interactive and responsive)
  - **Mobile agents**: running code + data that can “travel” from one computer to another. Briefly popular in Java in late ‘90s because of the portability of its bytecode, but security risks outweighed the benefits.

Architectural Patterns

Architectural Patterns

- Architectural patterns build on top of the “basic” architectural elements discussed above
- Patterns are composite, **recurring structures** that have been shown to work well in specific cases
- Examples of common patterns:
  - Layering
  - Tiered architectures
  - Proxies

Layering

- The concepts of *platform* and *middleware*

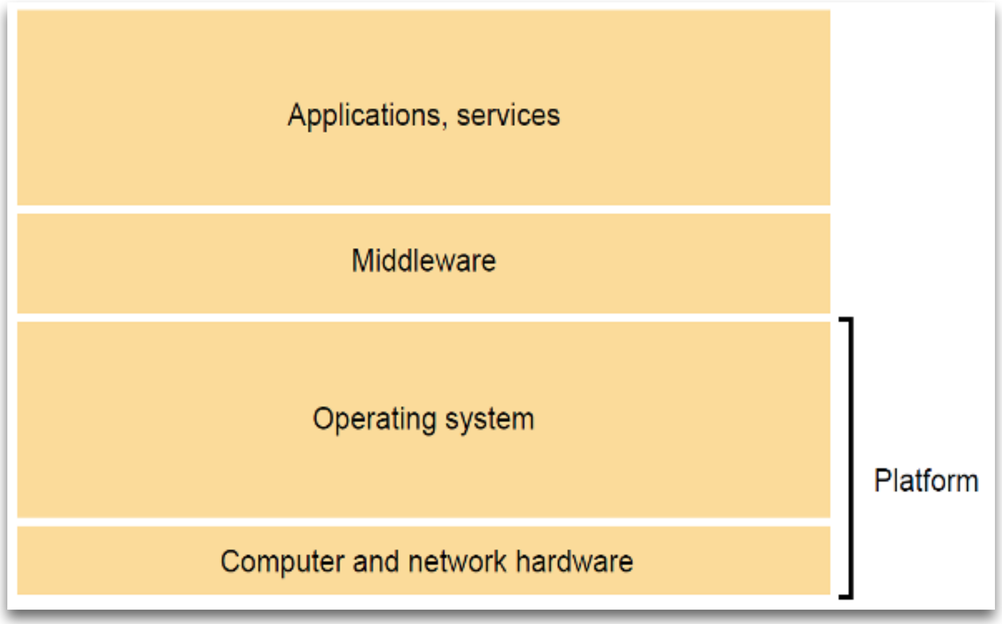
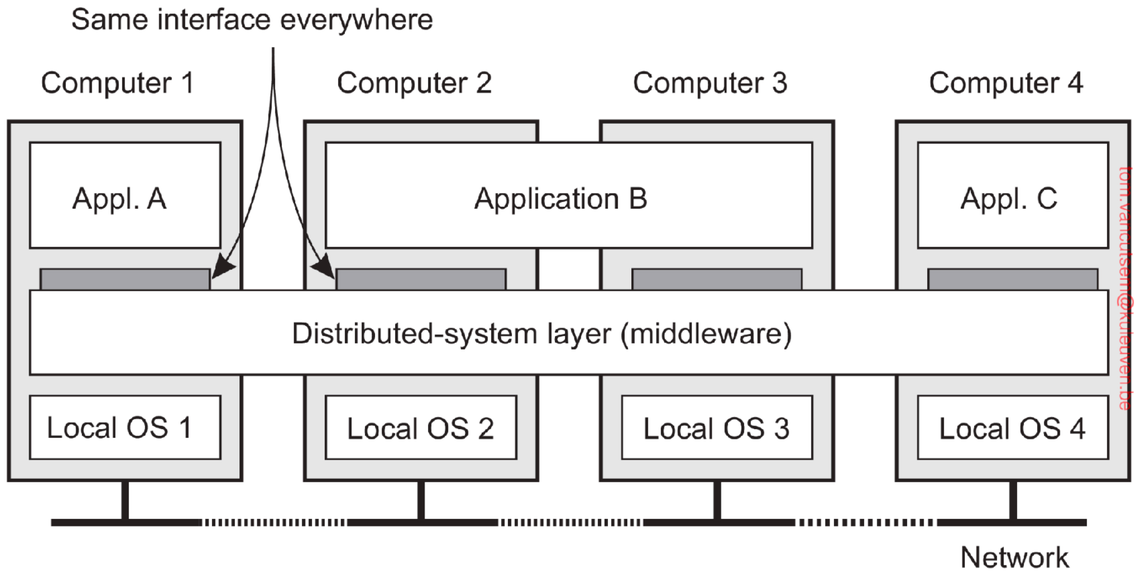


Figure 2.7 Software and hardware service layers in distributed systems



(Image credit: Maarten van Steen & Andrew Tanenbaum,
"Distributed Systems", 4th edition)

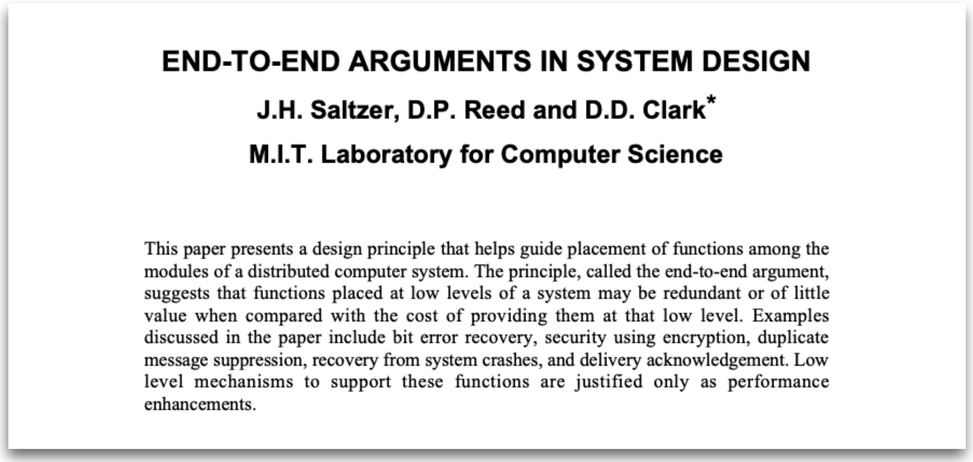
Layering: middleware layer

- Purpose:
  - **Mask** platform **heterogeneity**
  - Provide a **convenient programming interface**
    - Often raising the level of abstraction (example: remote method invocation versus reading/writing bytes on a socket)
  - Provide **basic infrastructural services** (example: name resolution, persistent storage)
- Examples of middleware:
  - Remote method invocation service (Java RMI, gRPC, …)
  - Group communication services
  - Event notification services (a.k.a. “message brokers” or “event bus”)
  - Distributed file systems (offering e.g. replicated file storage)
  - Distributed job processing (e.g. Google MapReduce, Apache Spark)

Layering: middleware layer

- Key limitation / trade-off: the “**end-to-end argument**” in systems design
- Basic insight: some aspects can only be decided at the application level
- Example: an e-mail protocol must acknowledge successful receipt of an e-mail by the server, even when the e-mail is sent to the server via a reliable network protocol such as TCP/IP.

“[Some communication-related functions] can be completely and correctly implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible.”



END-TO-END ARGUMENTS IN SYSTEM DESIGN

J.H. Saltzer, D.P. Reed and D.D. Clark*

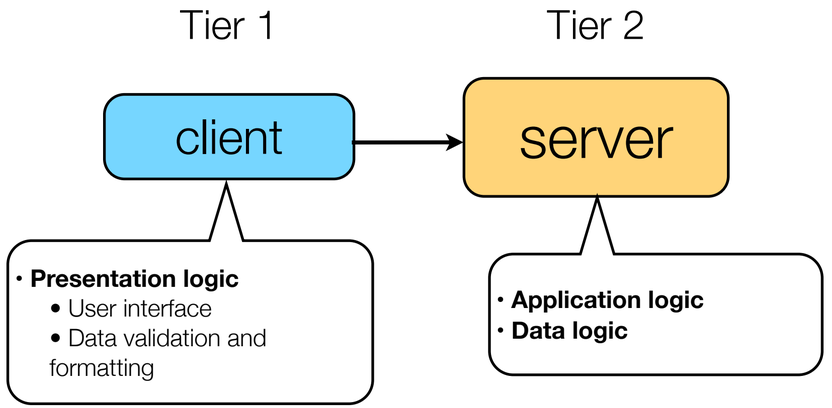
M.I.T. Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement. Low level mechanisms to support these functions are justified only as performance enhancements.

Seminal 1981 paper introducing the end-to-end argument

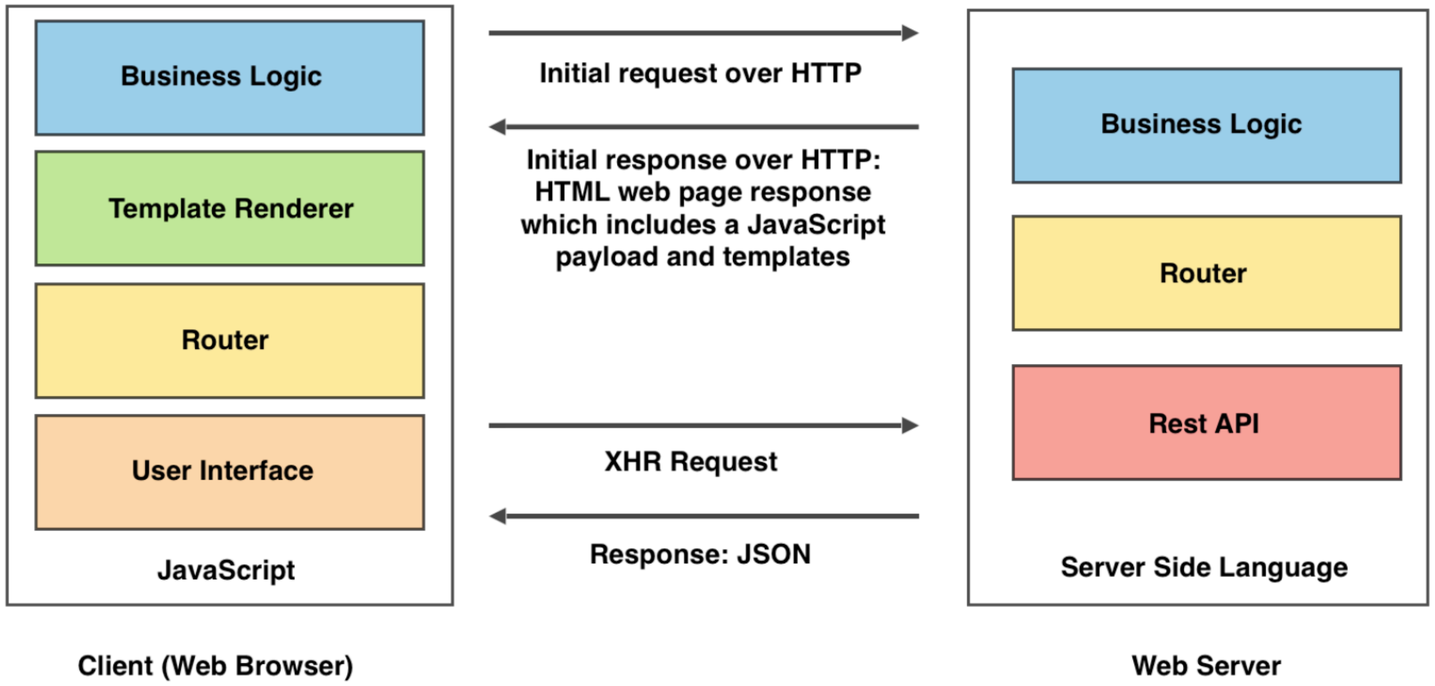
Two-tier architecture

- Split **presentation** logic (on the client) from **application** and **data** logic (on the server)



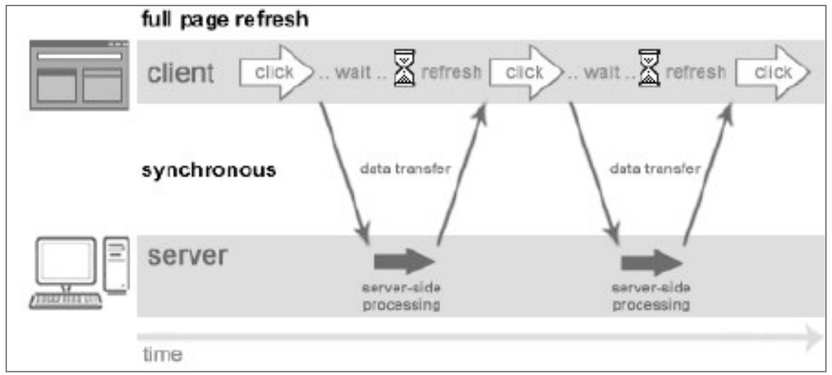
Two-tier architecture example: modern single-page web applications

- **Client** = webpage with embedded Javascript code
- **Server** = web server that serves the static web content *and* offers an API that the client can call to fetch additional server state, or update server state

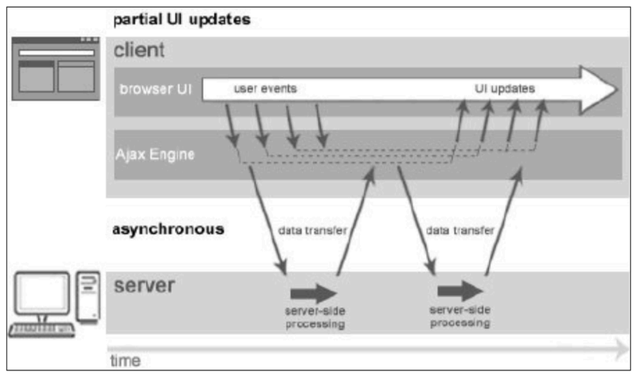


Aside: the role of AJAX in the evolution of rich web applications

- AJAX = **A**synchronous **Ja**vaScript and **X**ML
- It refers to websites that use JavaScript to make dynamic HTTP requests back to the server
- Websites before AJAX:
  - HTML pages with embedded forms
  - Every change required a full reload of the page
  - Terrible user experience



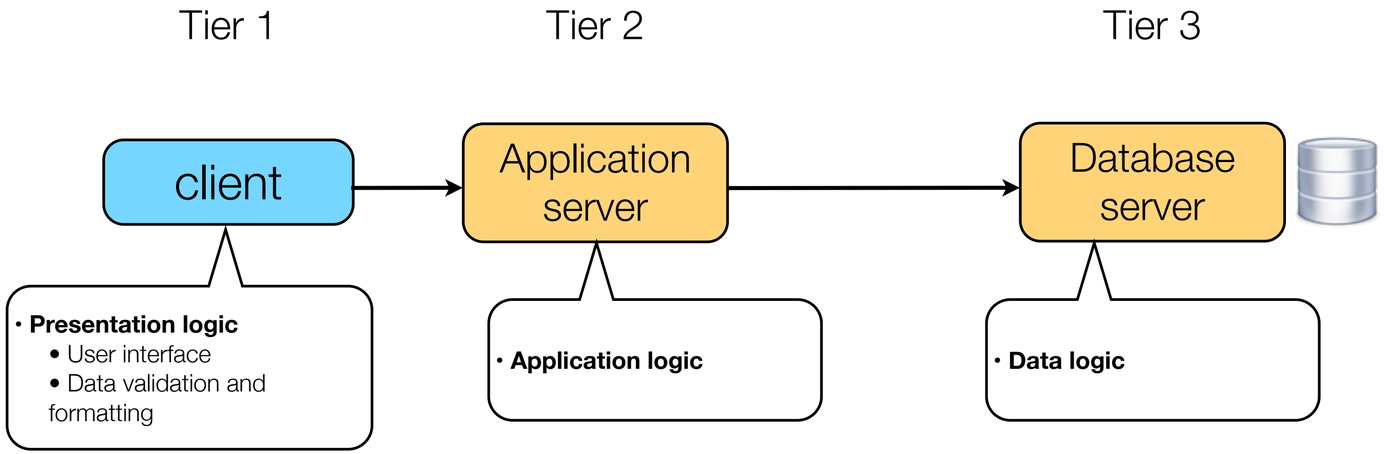
- Websites that use AJAX:
  - JavaScript code asynchronously fetches data from server
  - On reply, the script updates the page without reloading
  - This enabled the rich, interactive user experience that we now take for granted (e.g. Gmail, Google Docs, Google Maps, …)



Three-tier architecture

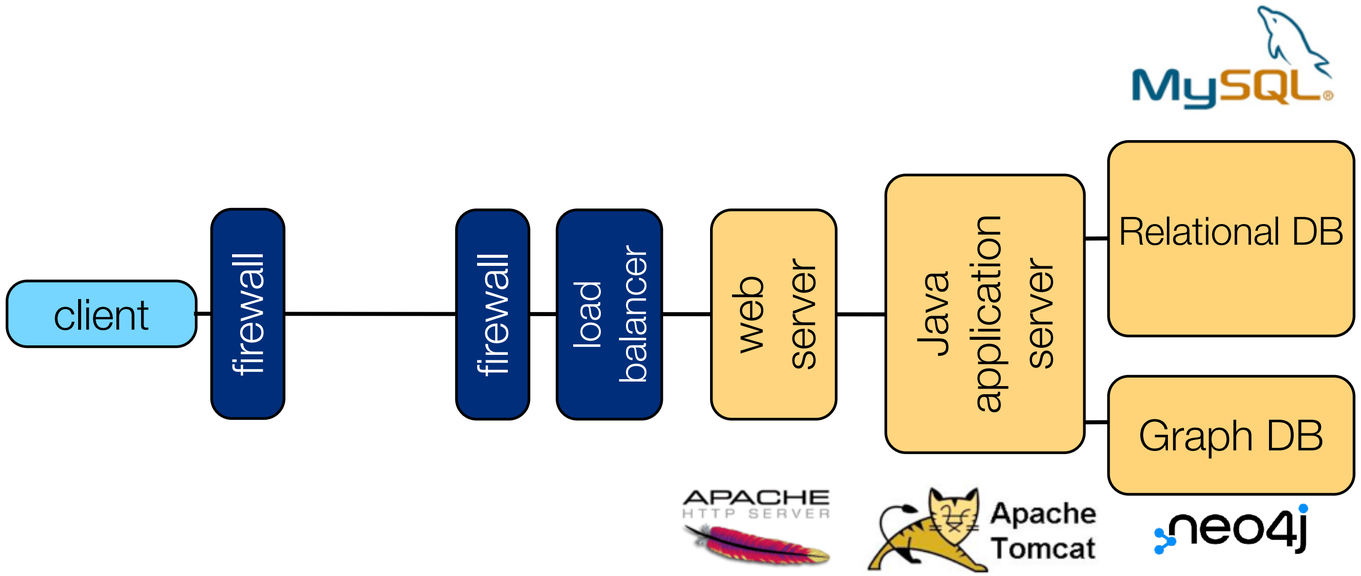
- The **data logic** is further separated into a separate system independent of the **application logic**

- Example: a web server that stores its data in a database



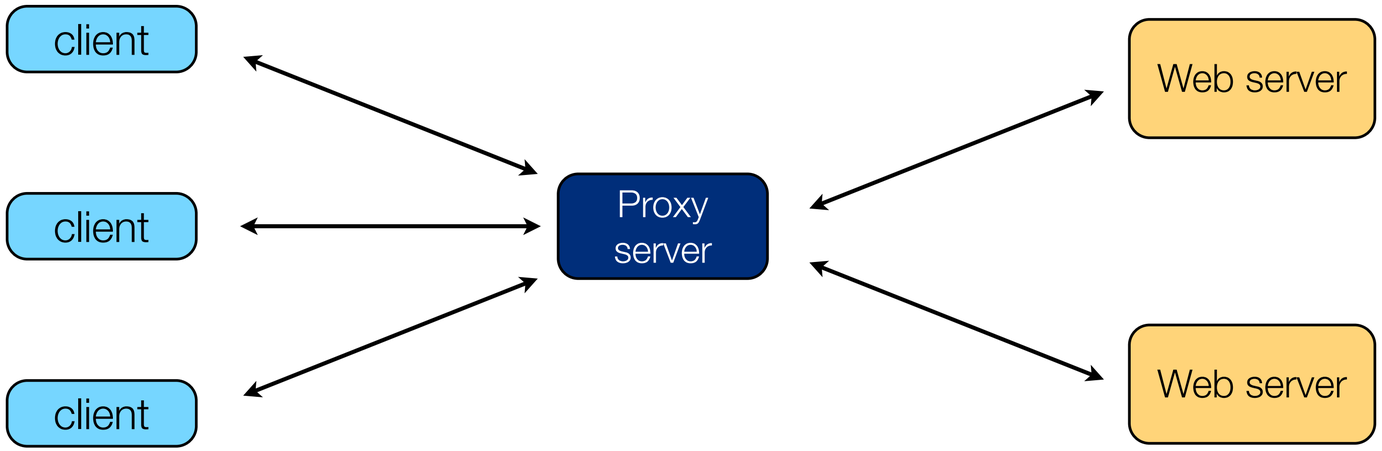
Multi-tier (or N-tier) architecture

- Real-world distributed application architectures are often multi-tier



Proxies

- A proxy server can:
  - **Cache** server responses
  - **Distribute load** across servers (= load balancer)
  - **Hide** the existence of many servers
  - **Filter** requests (= firewall)
  - Add **authentication** or security (e.g. TLS encryption)



More patterns (textbook section 2.3.2)

- Thin clients
- Brokers
- The Proxy pattern
  - Not to be confused with proxy servers!
  - In remote method invocation systems, a proxy object is a local object that offers the same interface as a remote object, forwarding requests from the local object to the remote object
  - Will become clearer when we discuss the details of Java RMI (see later lecture)

Architectural Patterns (recap)

- Architectural patterns build on top of the “basic” architectural elements discussed above and provide composite, recurring structures that have been shown to work well in specific cases
- Examples of common patterns:
  - Layering: platforms and middleware
  - Tiered architectures: two-tier, three-tier, multi-tier
  - Proxy servers

Summary

- Architectural models: describing distributed systems at a high-level
- Many examples of commonly know paradigms, architectural elements and interaction types have been introduced briefly; these must be revisited at the end of the course!
- The role of middleware has been clarified; this will continue.

Questions?