# Application Architectures

# Layered structure

Division of the work of an application into 3 general functions, which can evolve independently:

- Presentation:
  user input and commands, and display

- Business logic:
  business objects, rules, processing logic, processes

- Data:
  storage and logical access

Distribution onto « Tiers »

Distribution of the layers onto multiple machines ("tiers") communicating over a network
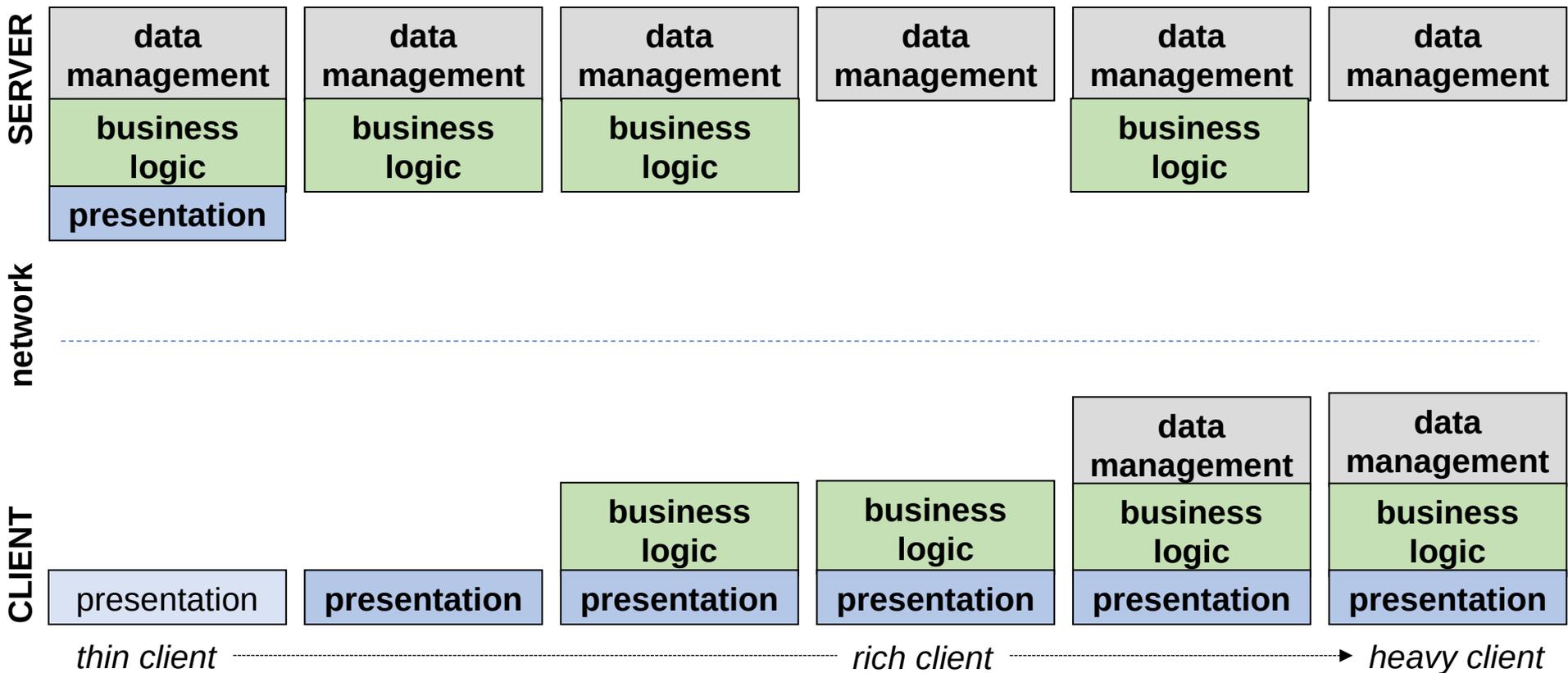
Monolithic and Single-tier Applications

Monolithic application

The 3 application layers are intimately interlaced in the same code base

```
import java.io.*;
public class ReadFromFile {
    public static void main(String[] args) throws Exception  {
        File file = new File("C:\\Users\\galtier\\Desktop\\test.txt");
        BufferedReader br = new BufferedReader(new FileReader(file));
        String st;
        while ((st = br.readLine()) != null)
            System.out.println(st.toUpperCase());
        encrypt(file, "mySecretKey");
    }
}
```

data management

presentation

business logic

Single-tier Application

Everything is local

# Modular Monolith

well-defined modules with strict boundaries, deployed as a single unit
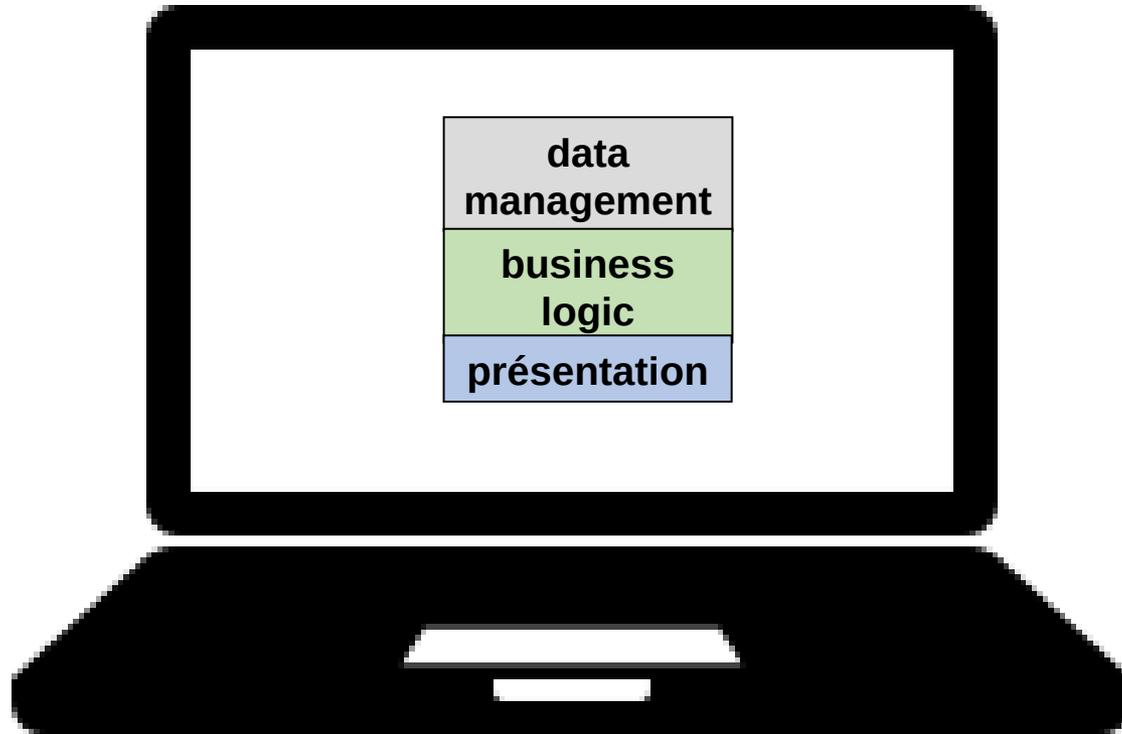
1st architectural style, but still relevant

- The area of "pre-network" PCs (late 70's – mid 80's)

- Still lots of stand-alone apps
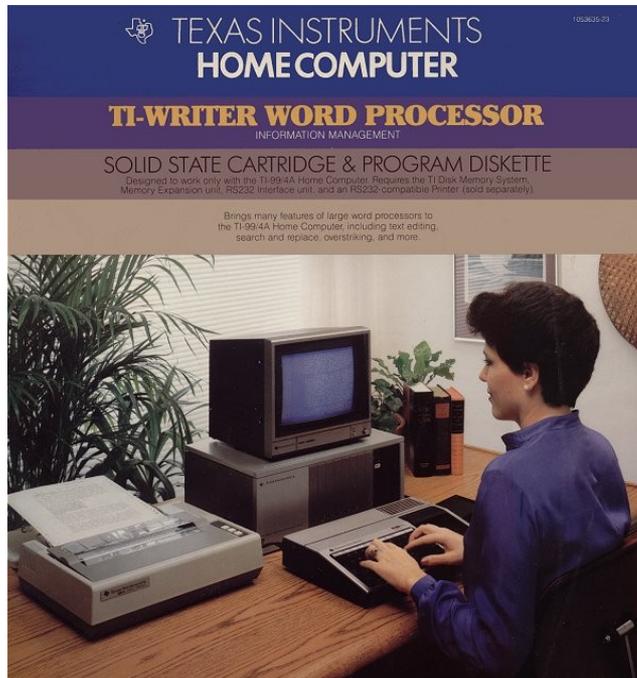
Advantages of single-tier

- Performance: 0 latency
- Safety by isolation
- Operate even in disconnected mode
- Simplicity (complexity reduced to the one of the code)

Disadvantages of monolithic applications

- Code is complex to learn, debug and evolve

- Even a minor upgrade requires a complete reinstallation of the entire application

- A failure in one "layer" renders the application completely unusable

- Inability to leverage heterogeneous technologies

- Not cloud-ready

NOT WANTED
up or down
★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★

THE MONOLITH
FOR · EXPENSIVE TO SCALE · DIFFICULT TO
MANTAIN · REWARDS · MORE TIME WITH
FAMILY · GOOD SLEEP NIGHTS
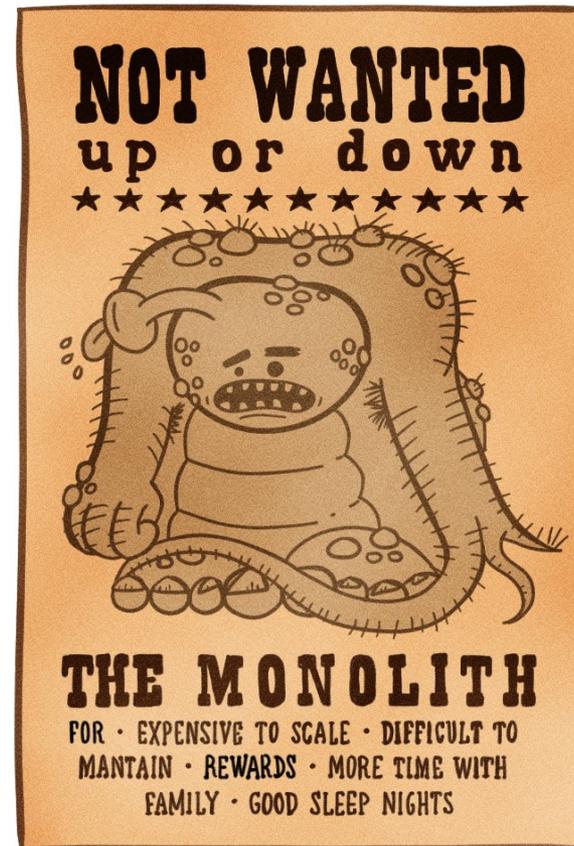
Daniel Stori
thanks to michael tharrington

https://dzone.com/articles/not-wanted-comic

Disadvantages of single-tier applications

"I'm a stand-alone PC but I'm lonely and want to be part of a popular network."
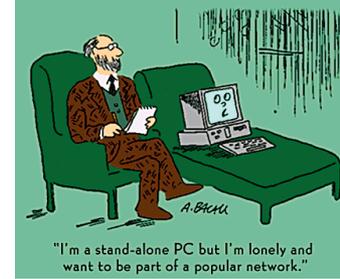
- Performances: depend on the capabilities of the host
- Shared resources impossible, requires duplicates (waste of resources)
- No fault tolerance
- Nomadism is difficult:
    - Access limited to physically logged-in users
    - More difficult (if not impossible) to continue a task from a different workstation
- Deployment is difficult:
    - Requires actions on each terminal
    - To be reinstalled if the underlying system needs to be reinstalled
- From the publisher's point of view:
    - No fix possible without user action
    - Application vulnerable to reverse engineering

Mainframe Architectures

Principle
"host" Architecture
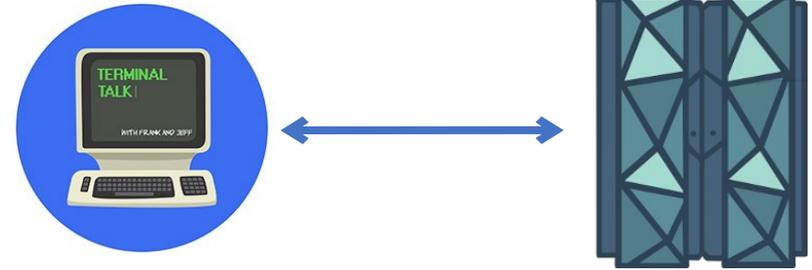
- Supercomputer :
  - ensures the data persistence, processing, and presentation
  - proprietary hardware and OS (IBM)

- passive clients :

  thin client visualization application
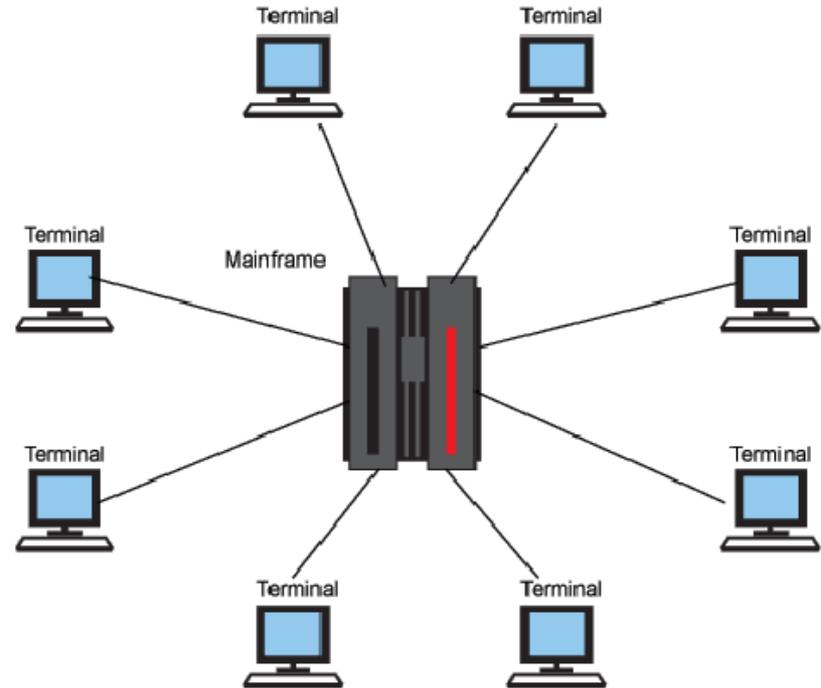
Advantages

- Performances: handle a very large number of simultaneous queries on very large databases

- Consistency, stability and long-term support

- Security

- Reliability (IBM Z customers: 99.9999% uptime)

  Robustness: https://www.ibmmainframeforum.com/mainframe-videos/topic10889.html

Performances

- Ability to process a very large number of simultaneous queries on very large databases

Batch or real time operation:

- Batch back-office

Batch job

Input data → Application program — Processes data to perform a particular task → Output data

- Transactional

Online (interactive) transaction

Query / Reply ← Application program — Accesses shared data on behalf of an online user →
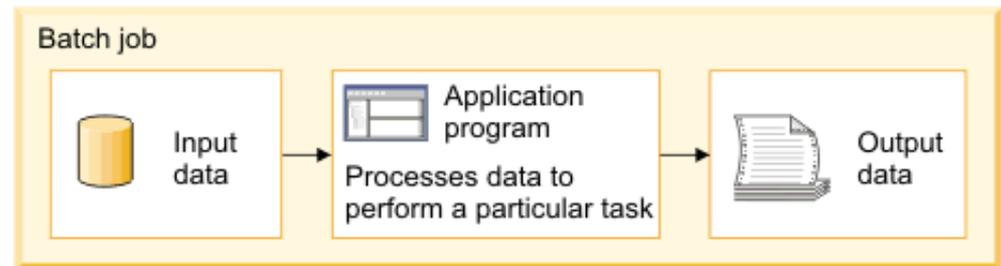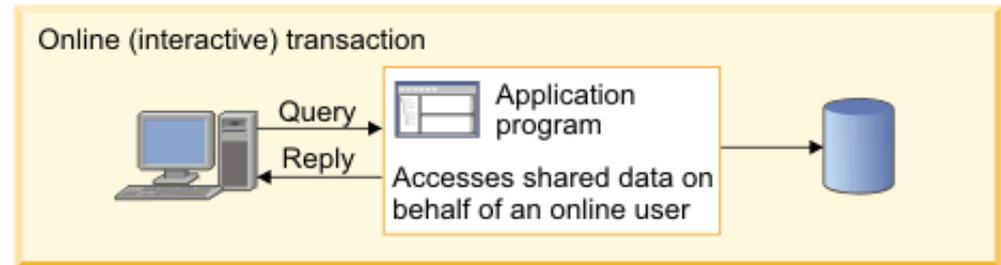
- Used in banks, insurance companies, airlines...

Transactions

- *Program accessing and/or modifying persistent data*
- A good transaction is
  - **A**tomic
  - **C**onsistent
  - **I**solated
  - **D**urable
- Transactional monitor ("TP monitor")
  Schedules transactions executed in parallel
  - Multiplexing of requests on system resources
  - Transaction management (respect of ACID properties)
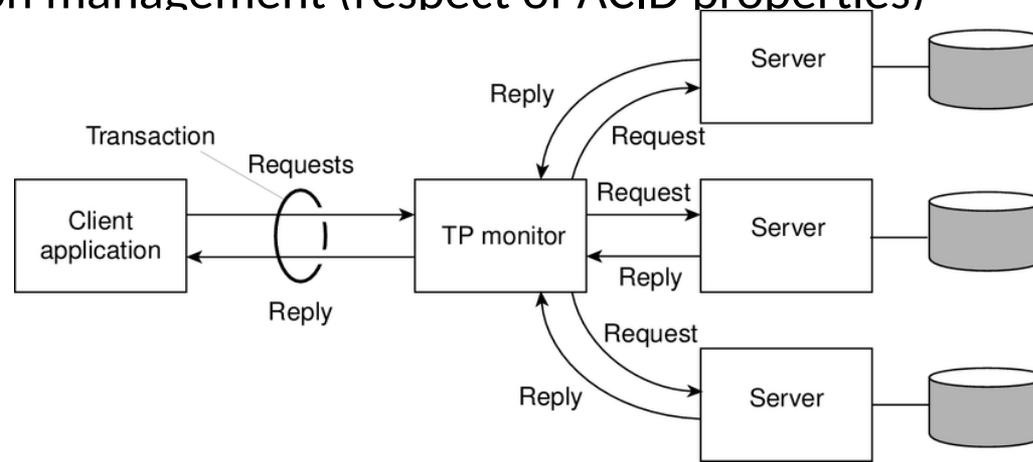
Extensively used

- 71% of the Fortune 500, 96 of the top 100 banks use mainframes

- process 30 billion business transactions per day, 87% of credit card transactions

- 250 billion lines of COBOL code, and 5 billion new lines each year

- Growth Outlook:
  - demand for HPC
  - increase in the number of banking transactions
  - development of blockchain

Obstacles to growth

- Proprietary solutions

- Huge investment
  - but no more than a server farm

(https://planetmainframe.com/2021/09/the-ibm-mainframe-the-most-powerful-and-cost-effective-computing-platform-for-business/

- Shortage of skilled mainframe staff
  - but Cobol is easy to learn

- Real alternatives + migration experience
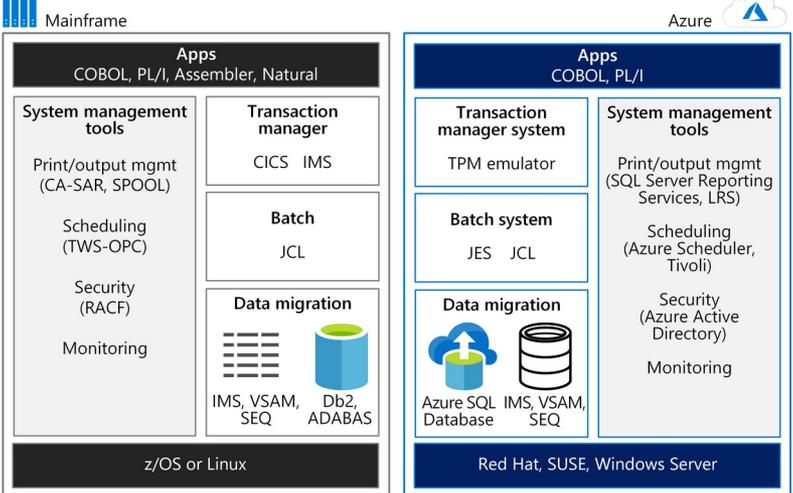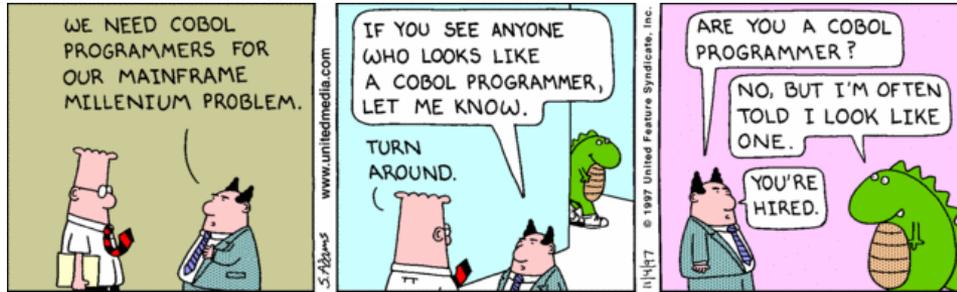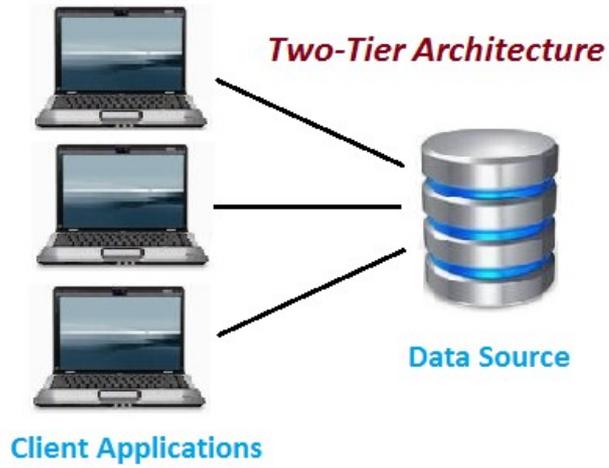
2-tier Architecture

The origin: "1.5-tier" Architecture

- Development of LANs

workstations / **heavy clients**

| presentation |
| business logic |
| data management |

local network

data

file server
(shared data storage,
but data management service reduced to tree-like organization of files)
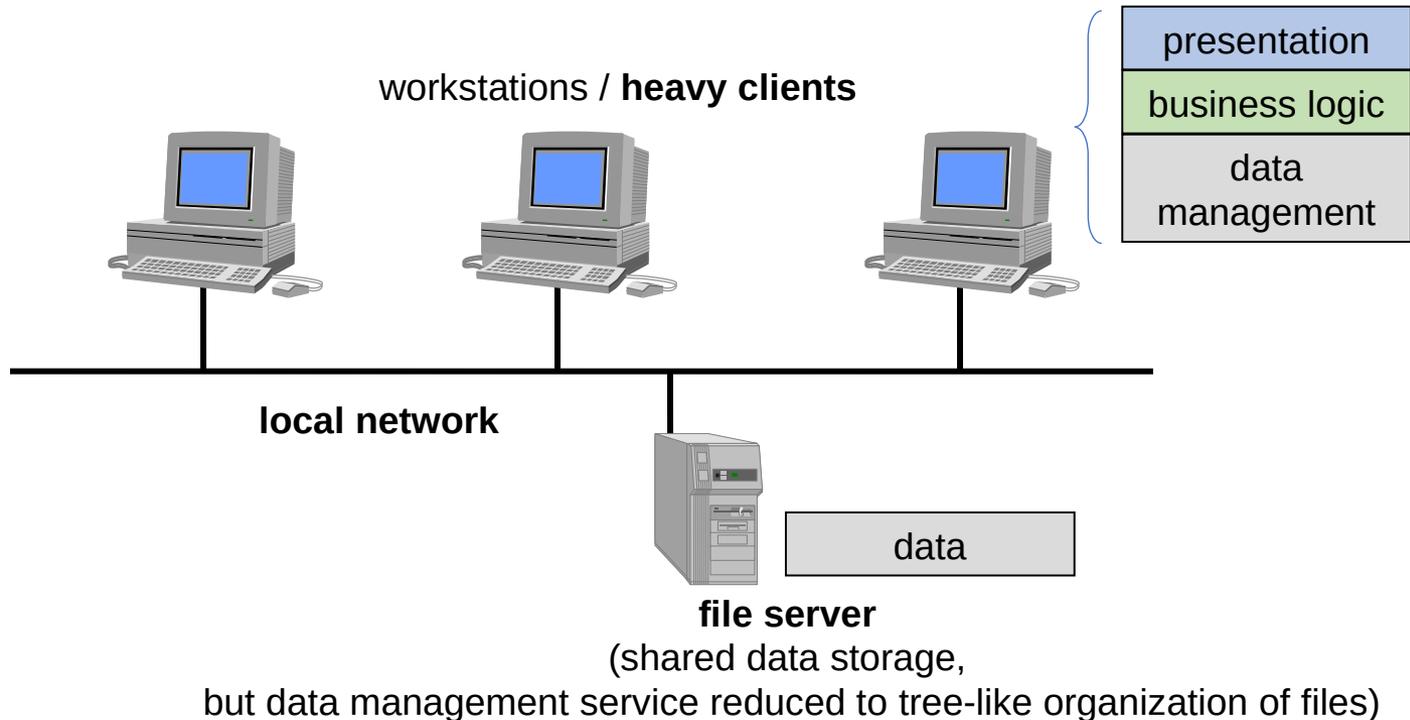
- Advantages: information sharing:
  - better communication
  - requires less resources

2-tier Architecture
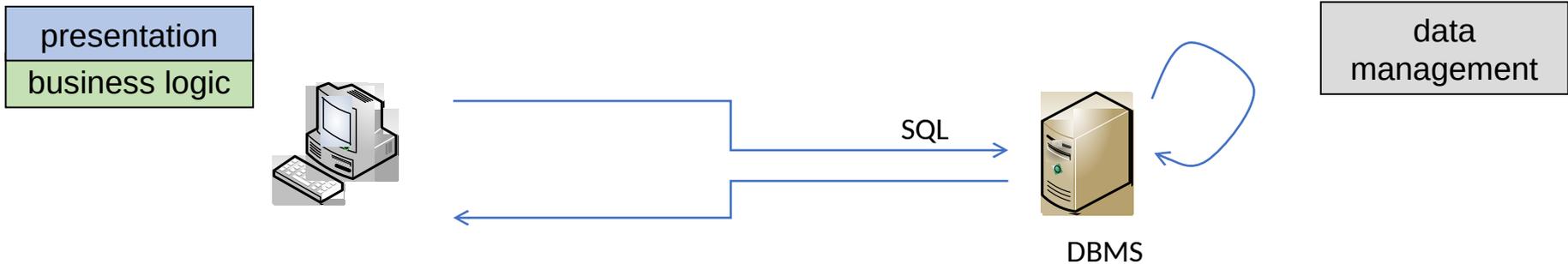
- Central database server
  - Manages physical I/O and provides logical data manipulation
  - Integrity control
  - Secure, optimized, transactional access
- Data handling is decoupled from its representation on disk, closer to the application logic

2-tier Architecture limits

- identical problems to single-tier:

    Not tolerant to client or server failures, updates require user's action…

- excessive use of stored procedures:
    - breaks the principle of single responsibility
    - complex to maintain
    - adherence with the physical model

- performance :

    Server and access network = bottlenecks

Thank you, 2-tier Architecture

- Microcomputing (previously confined to office automation) has taken on a growing role in IS

- The DBMS offer has grown, SQL has become widespread

- Has triggered the evolution towards more flexible architectural proposals

- Still relevant for simple applications

Relational model (Codd)

Invention of SQL

Launch of IBM PC

Lotus 1-2-3 killer app for PC

SQL becomes ANSI std

Microsoft and Sybase co-develop SQL Server

Oracle 7 introduces stored procedure

| Data stored in proprietary DB on mainframes | Flat files are shared across networks and managed manually | Flat files are shared across networks and managed manually |

1970   1973   1976   1979   1982   1985   1988   1991   1994   1996   1999

Command and Query Responsibility Segregation (CQRS)

- Separation of reads and writes.
- Often combined with Event Sourcing.
- Improves scalability and performance.

3-tier to 5-tier Architectures

3-tier

Client/Presentation Layer

Server/Application Layer

Database/ Data Layer

presentation

business logic

data management
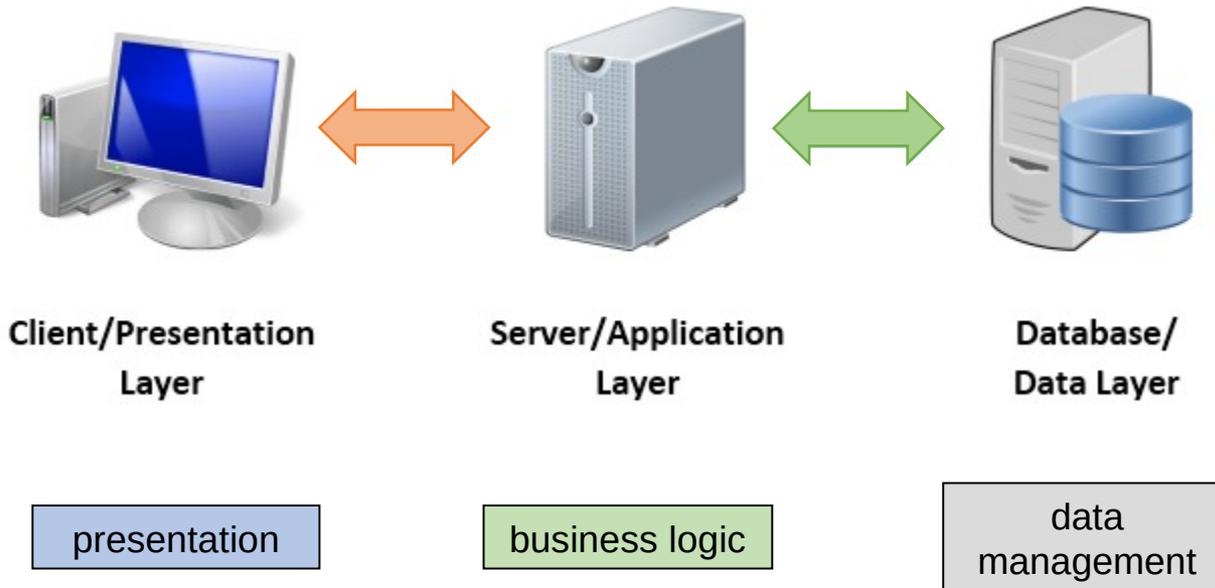
Example:
Classical Web Architecture


Users

Software Technologies

Presentation Tier

Model Interface — Forms & Scripts — Graph Tools — Results Analysis

Client-side Programming:
Ajax, Java Applets, Adobe Flex, JSP/ASP/PHP, Java Script

XML/HTML

Logic Tier

Analysis
Simulation
Parameter Estimation
Sensitivity Analysis

Data Handling
Storage
Retrieval
Conversion

Central Server

Visualization
Graph Layout
Interactivity

Server-side Programming:
Servlets, JSP/PHP/ASP, CGI, Python, Perl

Webservers:
Apache Tomcat
Apache HTTP

XML/SQL

Data Tier

Database, Data Warehouse and External Data Sources

Databases:
MySQL, Oracle

Data Formats:
XML, SBML, OWL, SQL

4-tier, 5-tier

Client Layer	Presentation Layer	Business Layer	Integration Layer	Data Layer
Browser	Presentation Logic Processing	Application Logic Processing	Data Access	DS-CBID Database
Tablet	Session State Management	Business Model	Messaging	Delivery System
Web Service			Service Integration	

presentation | business logic | data management

Perspectives for multi-tier architecture

- Corrects some of the problems of 2-tier architecture
  - Maintainability, evolvability, deployment
- Very popular model for non-intensive systems
- But to be completed to meet the challenges of reliability, performance, and scalability

SOA & Micro-services Architecture

Siloed Architecture
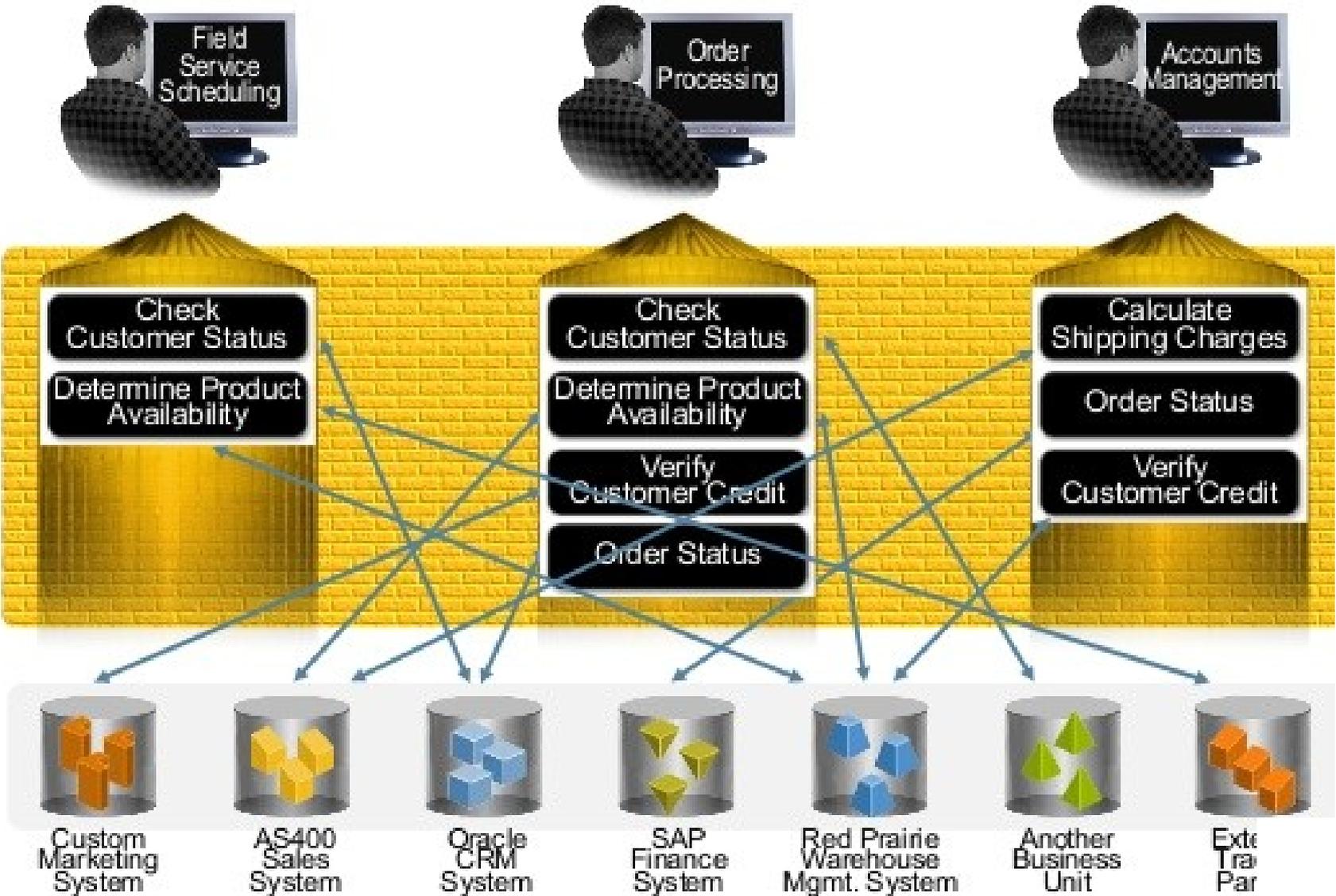
Problems with siloed architecture

- Early 2000s: growing need for interoperability in enterprise systems.

- Problem: isolated business applications, hard to integrate.

- Waste of resources

- Complex maintenance

- Lack of data sharing and consistency

- Complexity of IAM (Identity and Access Management)

- Difficult to scale up

- ...

Microservices Architecture
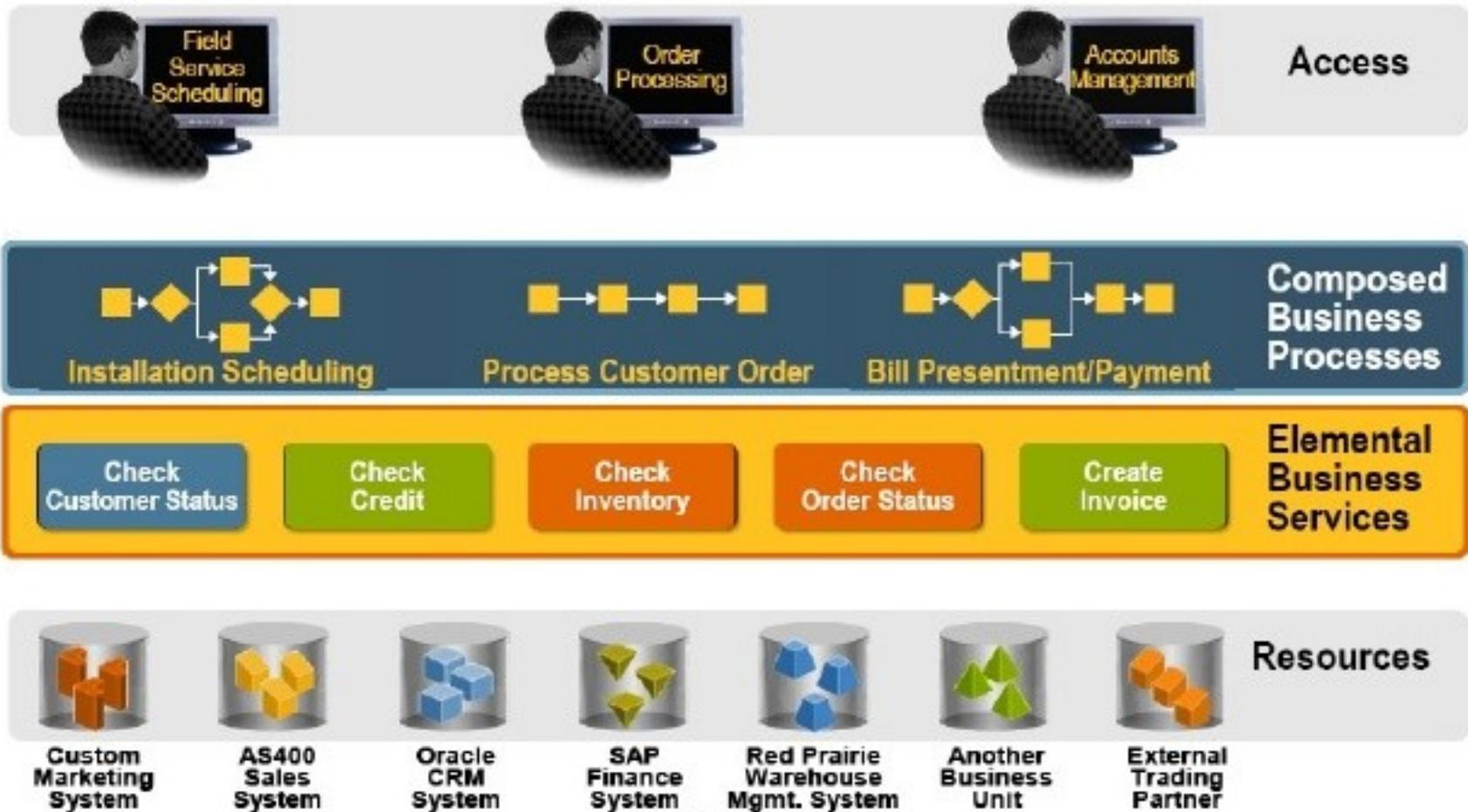
(Micro)Service Concept

- Black box performing 1 specific task (business or technical function)

- Can be used via an API (= contract between the customer and the supplier)

- Can call on other services

- Designed to be duplicated → *stateless:*
  - *No application state*
  - *Or client-specific state provided in the request*
  - *Or state on external storage shared with other services*

Advantages of the microservice architecture

- Reuse

- Scaling and fault tolerance thanks to easy duplication

- Fault isolation

- Independent development and deployment

- Ability to use the most appropriate technology for each module

- Small development teams
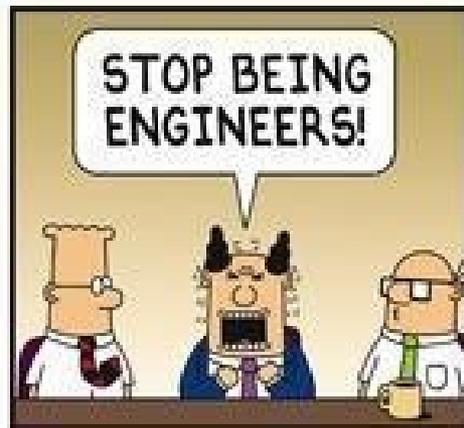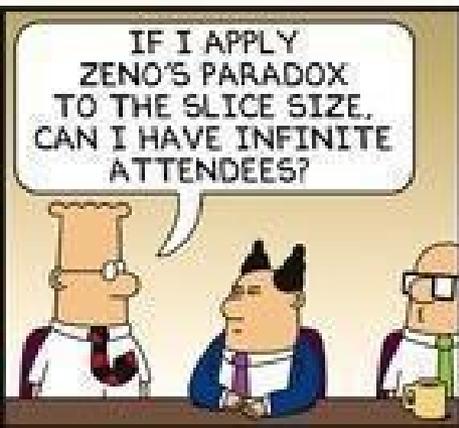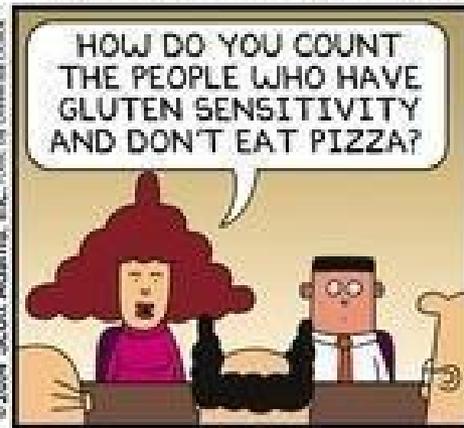
Event-Driven Architecture (EDA)

- Components communicate via events

  Kafka, RabbitMQ...

- Commonly used in **asynchronous** microservices
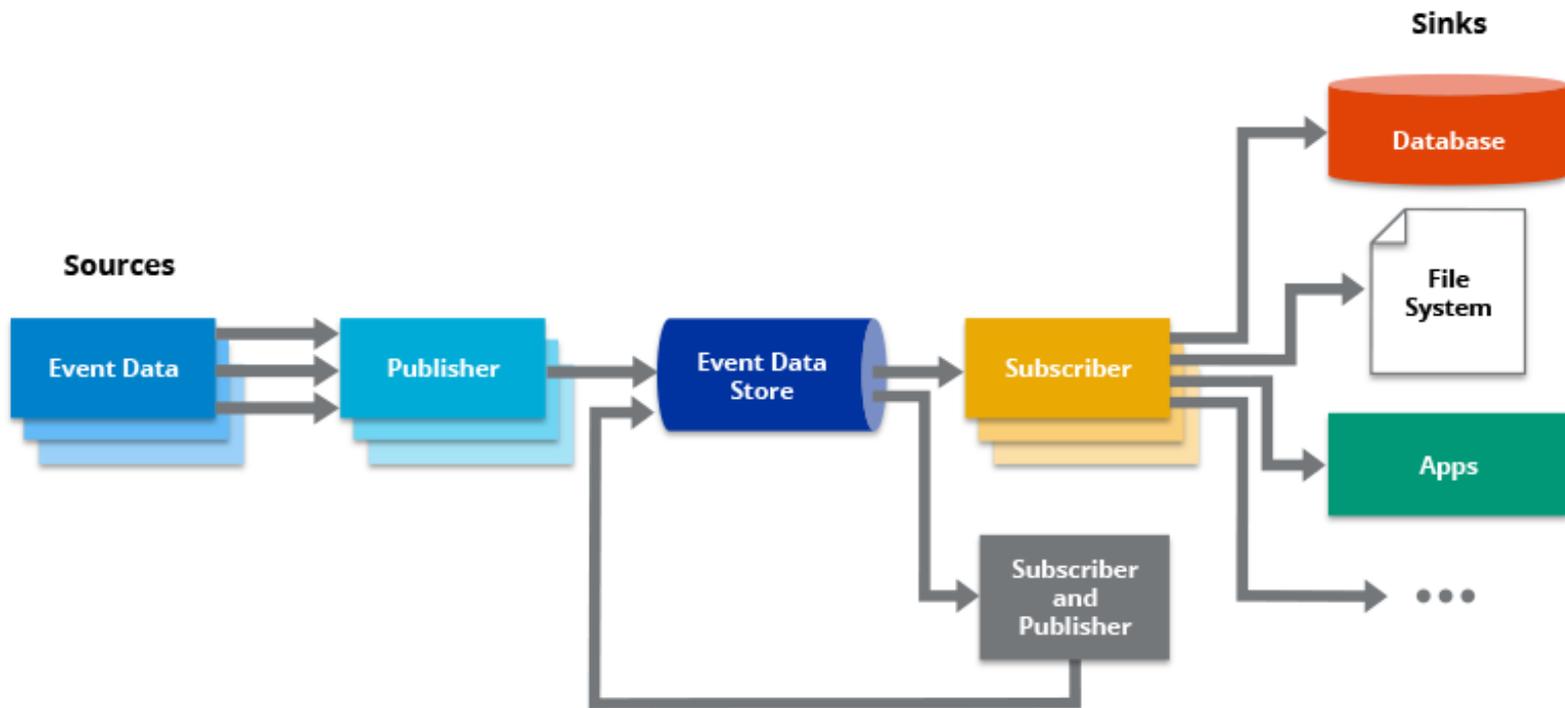
Actor Model

- Each "actor" is an isolated entity that reacts to messages posted in its own queue
  - Akka, Erlang/Elixir...
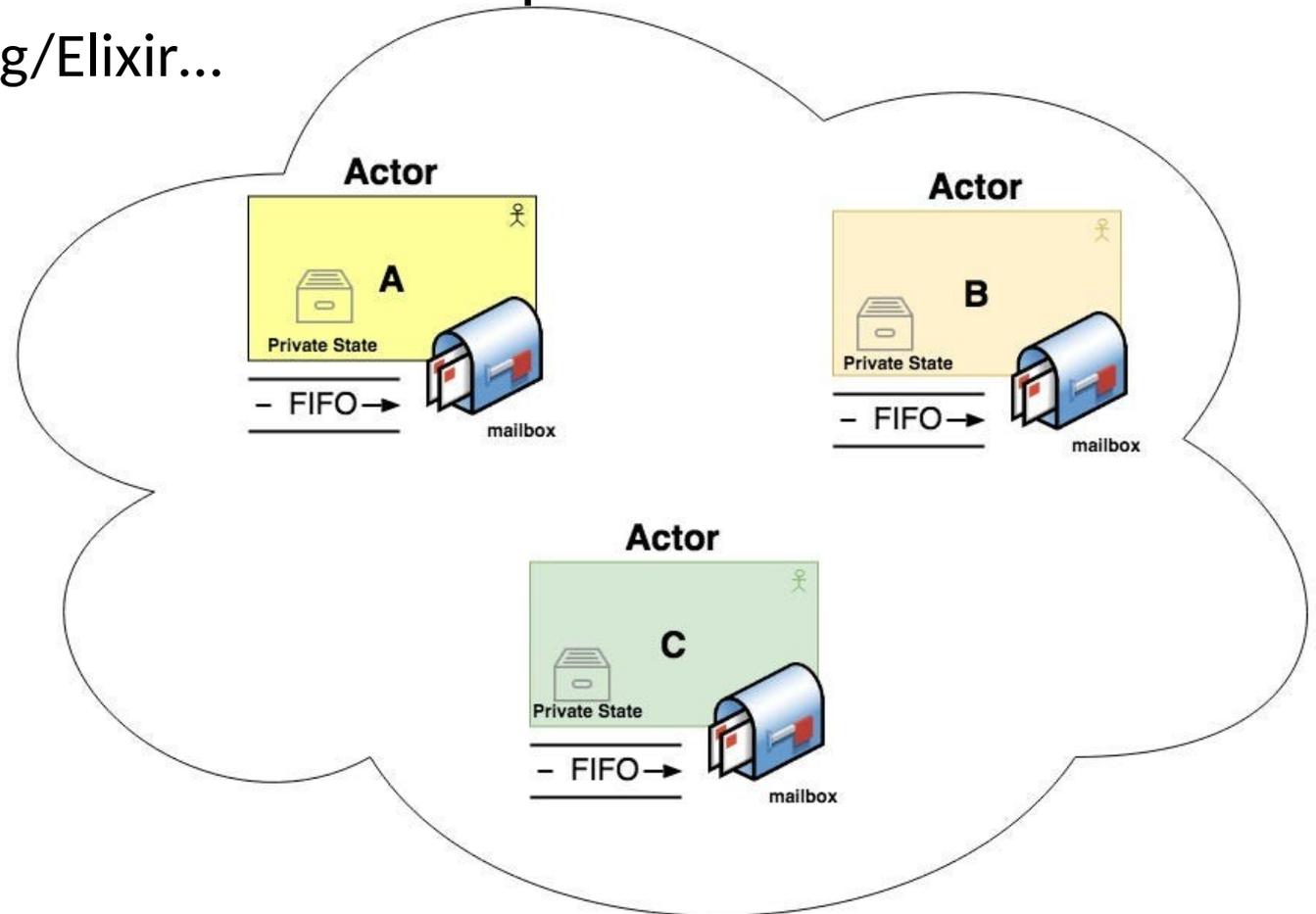
Limits of Microservice Architecture

- Operational Complexity
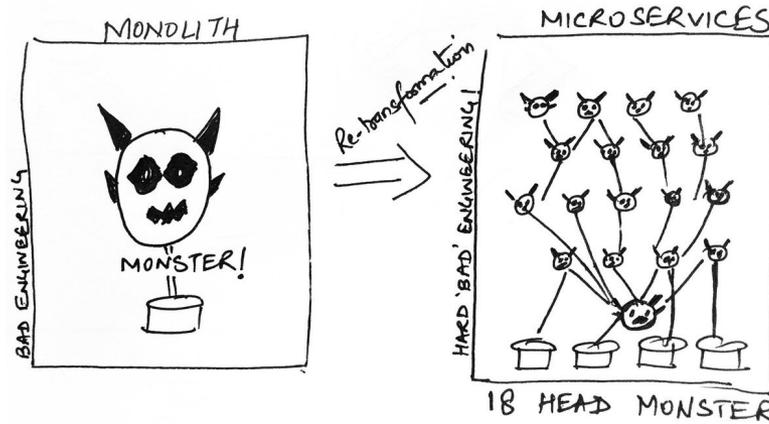
- Distributed System Challenges

    Network latency, service discovery, timeouts, debugging…
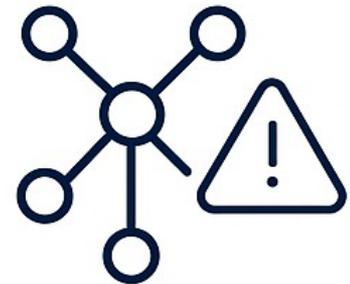
- Data Consistency Management

Limits of Microservice Architecture

- End-to-end Testing Difficulty

- Higher Resource Consumption
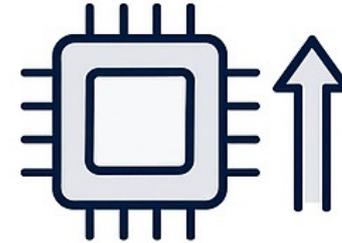
- Steeper Learning Curve
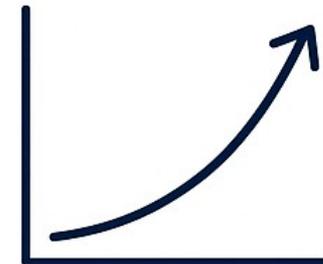
Middleware

Solutions to ease the connection between services:

- Locally:
  - Inter-process communication: system, MPI, Unix Domain Socket, etc

- Across the network:
  - Synchronous Remote Procedure Call
  - Asynchronous Messages

Remote Procedure Call Flow
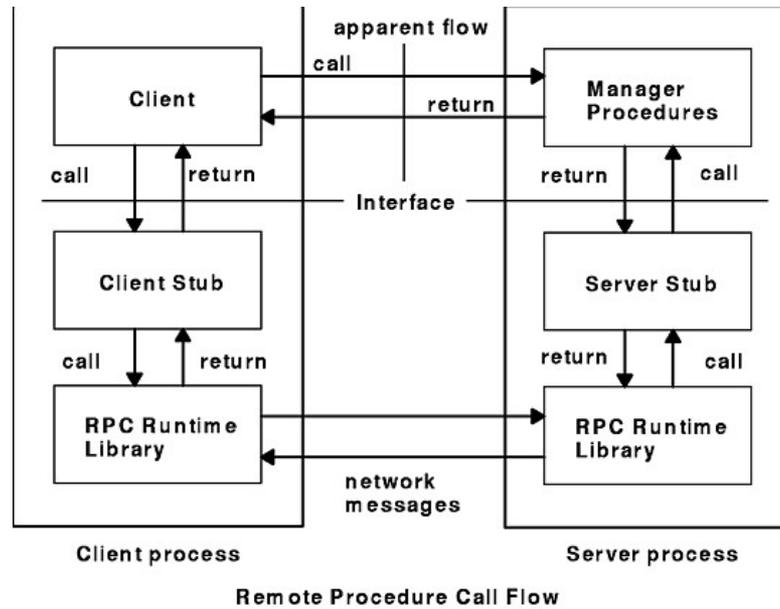
Remote Procedure Call (RPC) and
# Object Request Broker (ORB)

RPC

- [asynchronous] loose coupling between client and server

client   *server*

```
instruction i
      ↓
r = fac(x, y)
      ↓
instruction k
```

application layer

```
fac(int a, int b) {
   temp = a
   for i from 1 to b
      temp = temp * a
   return temp
}
```

proxy: local representative (on the client) of the server

déballe le résultat	emballe le nom de la méthode et ses paramètres
réceptionne	envoie

middleware layer

reconstitue l'appel avec le nom de la méthode et ses paramètres	emballe le résultat
reçoit la requête	envoie

proxy: local representative (on the server) of the client

OS and hardware layer

receive ← send → **network** → receive ← send

- The proxies handle:
  - network calls
  - format transformations between the client and server

(some) RPC implementations and frameworks

- Rise:
  - 80's: Sun RPC (as part of NFS protocol): simple, limited to Unix systems
 - 90's: DCE RPC (Open Software Foundation): platform-independent, rich set of functionalities (transactions, encryption…), more complex to use
- Fall:
  - 94: RPC is "fundamentally flawed": communication latency, partial failures and concurrency issues…
  - Message passing alternatives
- Rise, again: more features, more supported formats/transports…
  - 98: XML-RPC: data are XML-formatted and exchanged over HTTP -> SOAP
  - 2005: JSON-RPC, lightweight
  - 2007: Apache Thrift (init. Facebook): support for multiple serialization format (including binary), support for multiple transport protocols, complete stack for creating clients and servers
  - 2009: Avro (Apache Hadoop)
  - 2016: gRPC (Google, open source): messages serialized using Protocol Buffers (binary), transported by HTTP/2, multiple features
  - 2021: Cap'n Proto (now developed by Cloudflare): performances!

Object Request Broker

- Object oriented RPC: method calls on remote objects

- Most popular technologies:
  - CORBA (Common Object Request Broker Architecture) (1991)
    - OO-RPC for heterogeneous objects
    - but also a set of services

High-level horizontal frameworks

Vertical frameworks

System functions

Application Objects	CORBA Domains	CORBA Facilities	CORBA Services
specifics	health / finance / e-commerce	cryptography / QoS management / IHM	transaction / naming / events

ORB
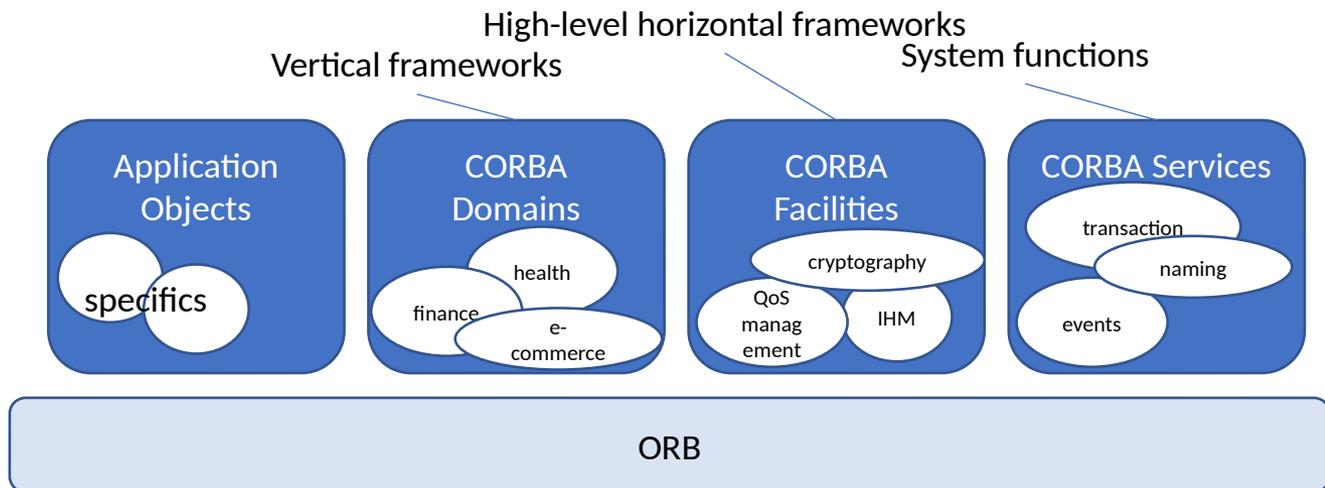
- DCOM (Distributed Component Object Model) (1995), .Net Remoting
  - Microsoft-equivalent to CORBA
- Java RMI (Remote Method Invocation) (1998)
 - for Java objects

CORBA perspectives

- Limitations:
  - local calls are treated the same as remote calls → inefficient
  - complex standard
  - difficult to have different versions of a service coexisting
  - fewer and fewer experts
- Why hasn't it disappeared?
  - still important legacy
  - one of the few candidates (with DDS) when there are strong real time constraints

    Alcatel-Lucent network management system, communications between military planes and ESA satellites, air control systems, Siemens electrical power plant management system…

Service call

- 1srt generation Web Services:
  - Requests and responses transported by SOAP messages, usually on top of HTTP
  - 4 patterns supported by WSDL:
    - Request - response
    - One way request
    - Notification
    - Request - response
  - WS-*: myriad of specifications to complete the messaging service

- Web service in a REST architecture:
  - URI-addressed resources
  - Requests and responses typically carried over HTTP, exploiting the semantics of HTTP methods
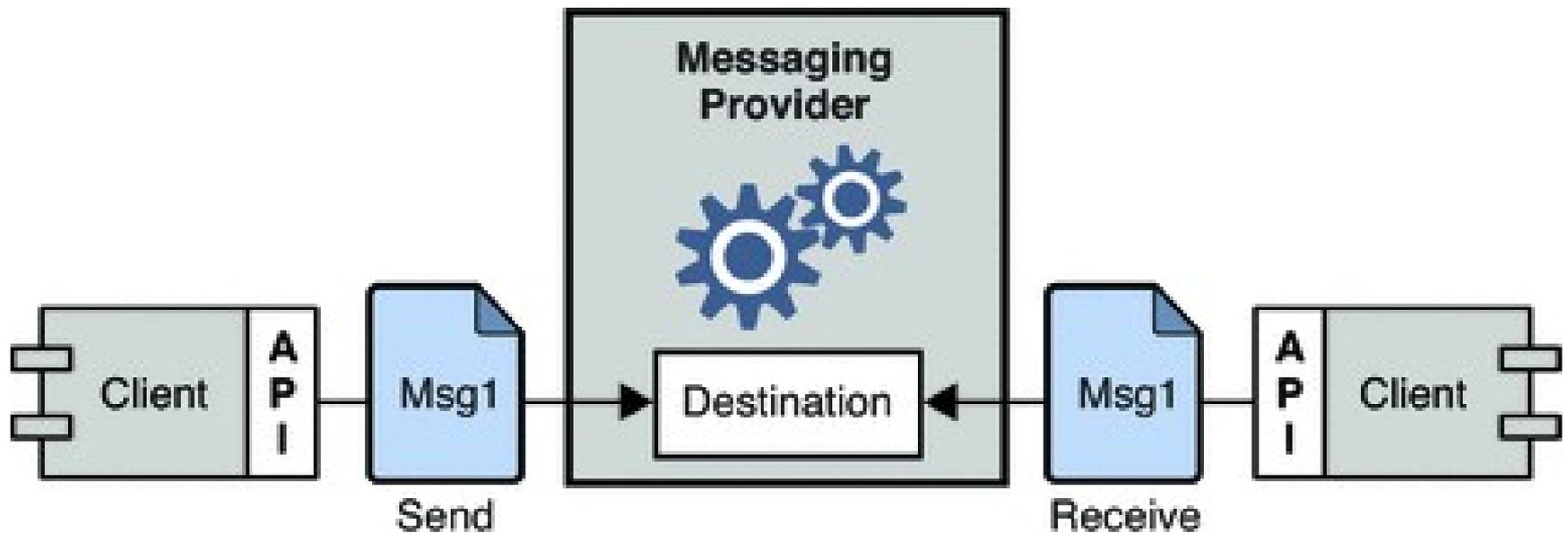
Message Oriented Middleware

Message Oriented Middleware

- Structure allowing one or more sources to transmit messages asynchronously to one or more destinations
  - No need to be connected simultaneously
  - Not need to know the source / the destination

Message based communications

Application A		Application B
Messaging API		Messaging API
Messaging client	Messaging system	Messaging client

Message

Message

Message oriented middleware
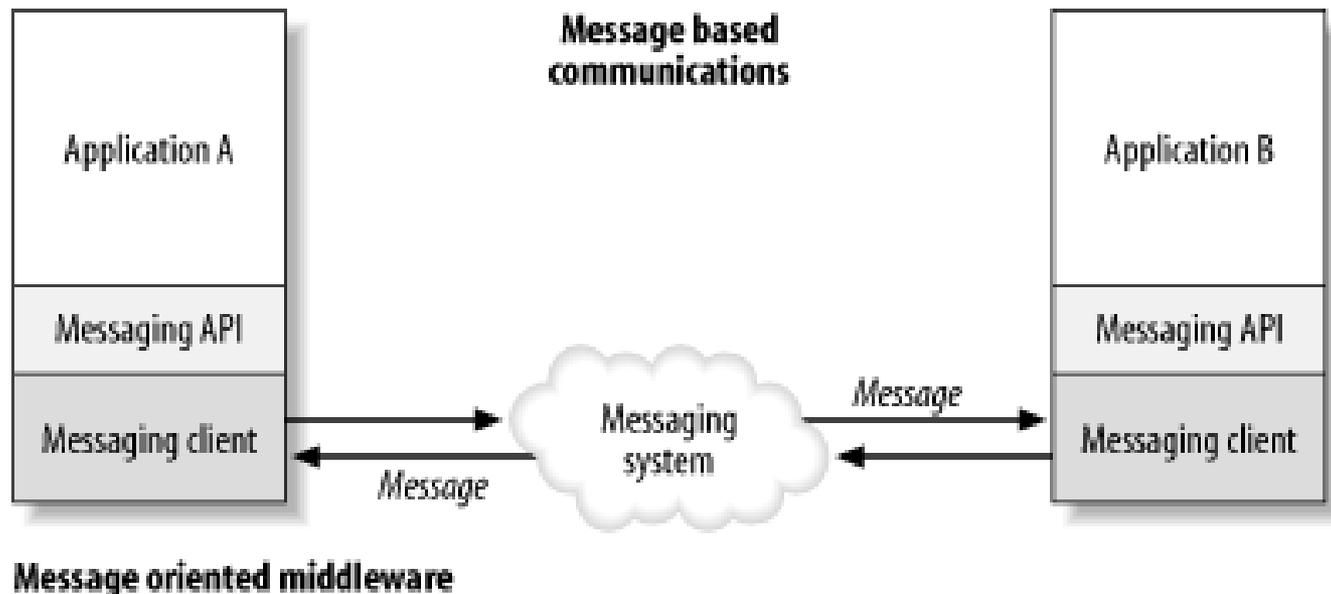
Optional Features

- Strict FIFO (, guaranteed delivery of messages in the right order) or hierarchical organization of messages, priority levels
- Point-to-point: a message read by a destination is no longer available for the others, or Publish-Subscribe : all subscribers to the queue receive a copy of each message (guaranteed delivery: at least once or exactly once)
- message filtering
- encryption/decryption functions, compression/decompression, format transformation
- message retention for offline consumers
- message expiration or validity date
- persistence (on physical media)
- reliability (Ack from MOM to sender and Ack from receiver to MOM)
- transactions
- …

Evolution of MOMs

- 95-2010: Earlier versions
  - 1994: IBM MQSeries (now IBM MQ): pioneer commercial MOM
  - 1994: TIBCO Rendezvous: high performance
  - 1996: Microsoft MSMQ, part of Microsoft Windows Server platform
  - 1998: Oracle MQ, now open source
  - 1999: FioranoMQ: HP for trading and finance
  - 2004: Apache ActiveMQ (open-source, java-based)
  - 2007: RabbitMQ (open-source, Erlang-based)
- 2010: Additional features:
  - 2011: Kafka: HA, replicate…
- 2010's: Integration with cloud technologies:
  - 2011: Amazon Simple QS
  - 2015: Google Cloud Pub/Sub
  - 2018: IBM Event Stream (based on Kafka), easily integrates with IBM cloud services
  - 2018: Azure Service Bus
 - 2019: CloudAMQP (based on RabbitMQ): automatic scaling

Middleware - MOM vs RPC

|	RPC	MOM
Métaphore	Appel téléphonique	Bureau de poste
Nature de l'appel	Synchrone (bloquant) : le client attend la réponse	Asynchrone (non bloquant) : le client peut continuer
Séquençage Client/Serveur	Ordre strict : le serveur doit être disponible avant l'appel	Pas d'ordre fixe : messages stockés en file d'attente
Communication	Directe entre le client et le serveur	Via une file de messages intermédiaires
Équilibrage de charge	Intégré au framework (gRPC) ou nécessite un outil externe	Automatique via la distribution dans les queues
Tolérance aux pannes	Faible : une panne bloque le client ou requiert un retry manuel	Forte : les messages restent dans la queue
Filtrage des messages	Non pris en charge	Facile à mettre en place
Performance	Rapide mais bloquant	Moins rapide (à cause du passage par une queue intermédiaire)
Gestion des transactions	Complexe : nécessite un protocole comme 2PC	Plus simple
Fonctionnement aynchrone	Nécessite une gestion plus ou moins complexe de threads	Natif

Overview
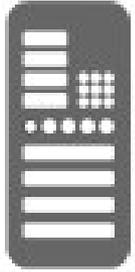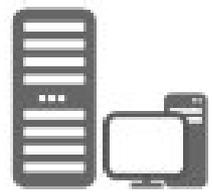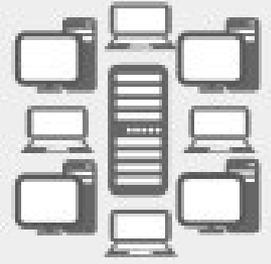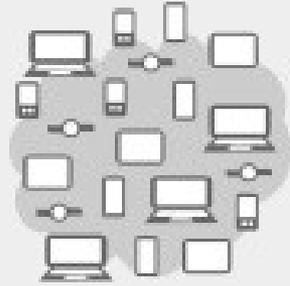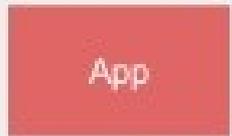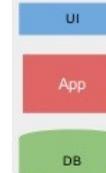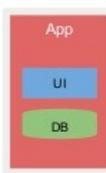
Aspect	Monolith	2-tier	3-tier	Microservices
Definition	Single codebase integrating all responsibilities	Client handles logic/UI, server handles data	Separation into presentation, business logic, and data	Independent services each handling a specific responsibility
Coupling	Strong	Moderate	Loose (per layer)	Very loose (per service)
Scalability	Hard (global redeployment)	Server is bottleneck	Layer-wise scalability possible	Individual services scale independently
Complexity	Simple initially, grows with code size	Medium (network + data layer)	Higher (requires coordination)	High (orchestration, observability...)
Deployment	One-step deployment	Centralized (DB server + client update)	Deploy per layer	Deploy per service
Maintenance	Hard when the code grows	Easier DB maintenance	Moderate	Easier (dedicated team per service)
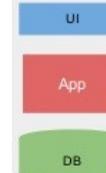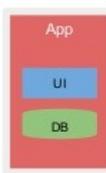
Aspect	Monolith	2-tier	3-tier	Microservices
Maintenance	Hard when the code grows	Easier DB maintenance	Moderate	Easier (dedicated team per service)
Technologies	Often a single stack	Heterogeneous between client and server	Each layer may use optimal stack	Freedom to choose the best tech per service
Fault Tolerance	One crash → total failure	Server crash = whole app down	Better fault isolation	Very good (service isolation + redundancy)
Cloud Compatibility	Poor (stateful, tight coupling)	Moderate	Good	Excellent (cloud-native)
Security	Basic (local)	Better control at server	Stronger control possible	High granularity in security policies per service
Mobility / Remote Access	Poor (local)	Somewhat limited	Good with web-based UI	Excellent (API-based, device-agnostic)
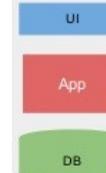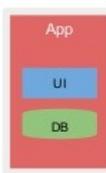
Aspect	Monolith	2-tier	3-tier	Microservices
Resource Efficiency	Efficient locally, but hard to scale	Better use of centralized DB	Moderate (centralized logic/data)	May be heavy (multiple containers)
Data Management	Local storage	Central DB	Central DB with shared logic	Decentralized or shared through APIs
Testing	Complex due to tight coupling	Easy unit tests, hard integration tests	Easier per layer	Unit tests easy, integration tests harder
Learning Curve	Low	Moderate	Higher	Steep (DevOps, distributed systems)
Initial Cost	Low	Low (except DB server)	Higher (infra + roles)	High (orchestration tools, CI/CD)
Use Case	Small tools, desktop apps	Database management systems	Enterprise-grade apps, CMS	Large-scale systems (Netflix, Amazon)

Conclusion

- Separation of Concerns enables better modularity, maintainability, and evolution of systems
- Application architectures evolved from monolithic and single-tier setups to multi-tier, SOA, and microservice-based systems
- Each architecture presents trade-offs in terms of performance, complexity, scalability, and fault tolerance

→ There is no one-size-fits-all architecture — the best choice depends on context, constraints, and future goals.

- Looking Ahead:
  - Emerging trends: Serverless, event-driven systems, function-as-a-service, edge computing
  - Ongoing challenge: balancing agility, cost, and resilience in an increasingly distributed world

Transition challenges

1. Service Boundaries Are Hard to Define

   Where do you split? By function? By domain? Wrong choices lead to tight coupling again.

2. Inter-Service Communication Replaces Function Calls

   Simple local function calls become remote API calls (with all the failure modes that implies).

3. Incremental Migration Is Tricky
   - Strangling the monolith gradually is complex: both architectures must coexist for a while.
   - You need backward compatibility, adapters, routing layers…

4. Team Reorganization
   - You may need to align teams to services (Conway's Law).
   - Autonomy requires product ownership, not just feature delivery.

5. Tooling Maturity Required
   - Microservices rely heavily on infrastructure: logging, tracing, service mesh, secrets management…
   - Without solid tools, you'll suffer from visibility gaps.

6. Increased Deployment Frequency

   Great in theory! But your CI/CD, monitoring, alerting, and rollback mechanisms need to scale with it.