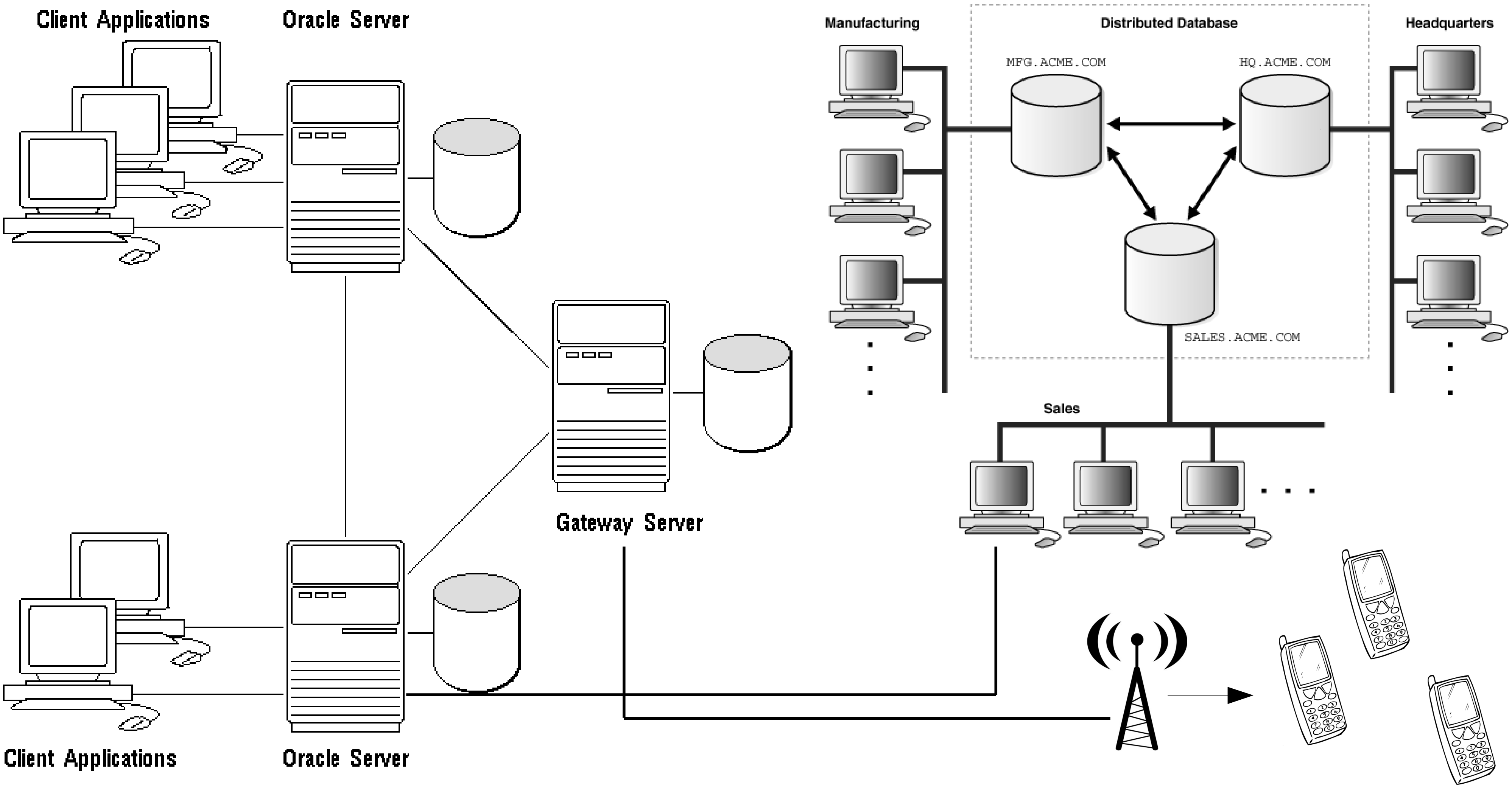
6

- Distributed Systems
  - Definition
  - Examples
- Distributed Applications
  - Definition
  - Motivations
  - Frameworks

Distributed Systems

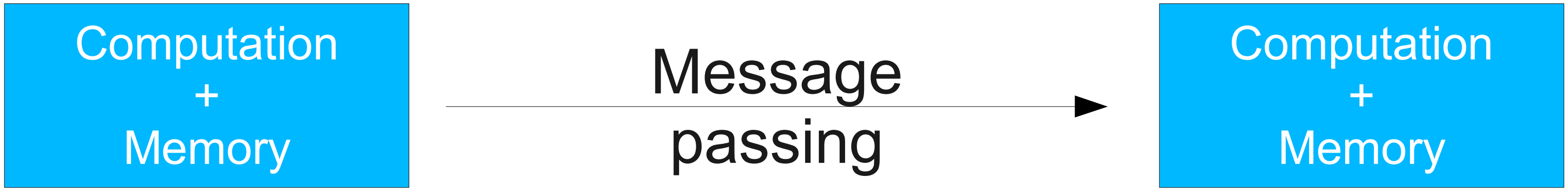
- A system including
  - Multiple (often >>2) **computation agents**
    - Each computation agent has
      - Computating power (CPU)
      - Local working memory (RAM)
      - Local permanent memory (storage), optional
  - An **interconnection structure** connecting those
    - Not necessarily a complete graph!
      - Routing more typical
    - Not necessarily able to transmit arbitrary data!
      - Although normally a general-purpose network technology is used, so arbitrary messages possible

Distributed Systems



Distributed Systems

- We can abstract it to:

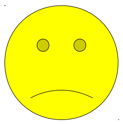


Distributed Systems

- Definition (W. Emmerich)
  - A distributed system consists of a **collection** of **autonomous** computers, connected through a **network** and **distribution middleware**, which enables computers to **coordinate their activities** and to **share the resources of the system**, so that **users** perceive the system as a single, integrated computing facility.
- Too strict for our purposes
    - No need for a middleware
    - No need for resource sharing
    - No need to trick the user's perception

Distributed vs. Centralized

- **Centralized** system
  - Single component
  - Parts not autonomous
  - Resources available locally
  - Monolithic software architecture
  - Single point of control
  - Single point of failure
- **Distributed** system
  - Multiple components
  - Independent parts
  - Resources may not be available
  - Necessarily modular architecture
  - Multiple PoC
  - Multiple PoF



Distributed vs. Concurrent

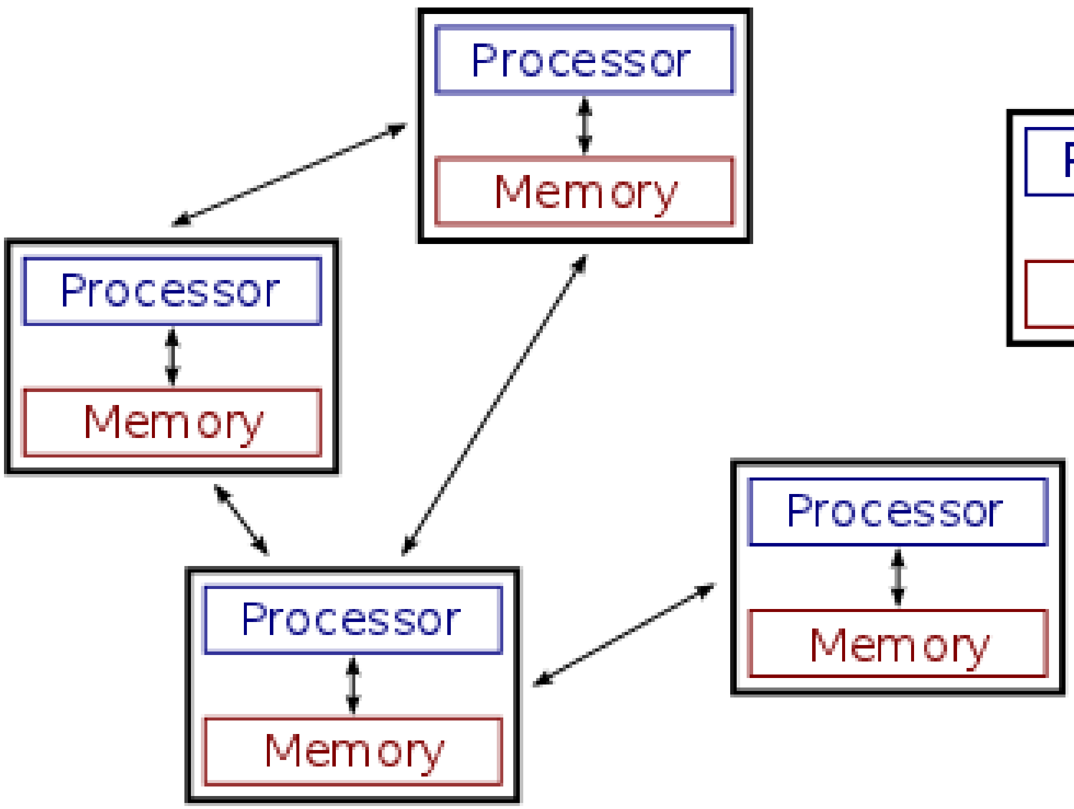
- **Concurrent** means that multiple things happen at once
  - e.g., single computer with multi-tasking OS
- **Distributed** systems are **concurrent** systems
  - By virtue of having multiple computation agents
- **Concurrent** systems are not necessarily **distributed**
  - It might well be that there is no connecting infrastructure

Distributed vs. Parallel

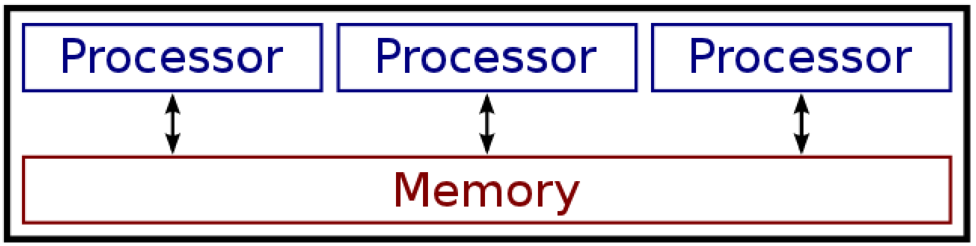
- **Parallel** means that we have multiple computing units, but a single (shared) memory
- **Parallel** systems may be configured to work as **distributed** systems
  - By using message-passing paradigm, etc. MPI
- **Parallel** systems are not necessarilty **distributed** systems per se
- Proper **distributed** systems are rarely **parallel** systems

Distributed vs. Parallel

- Distributed



- Parallel



Source: Wikipedia

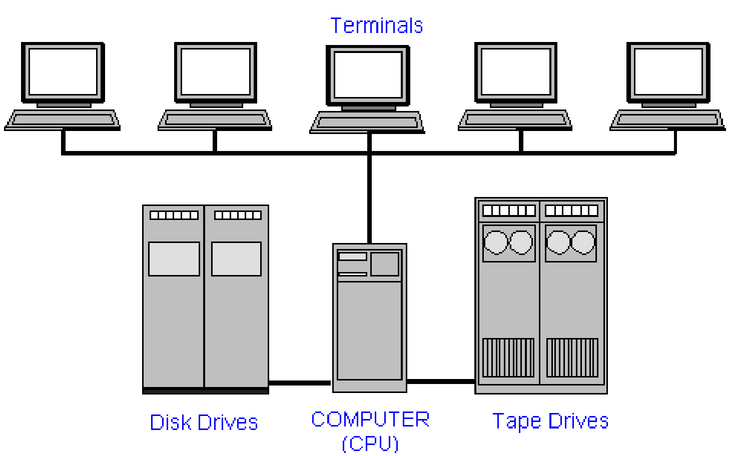
Distributed vs. Remote

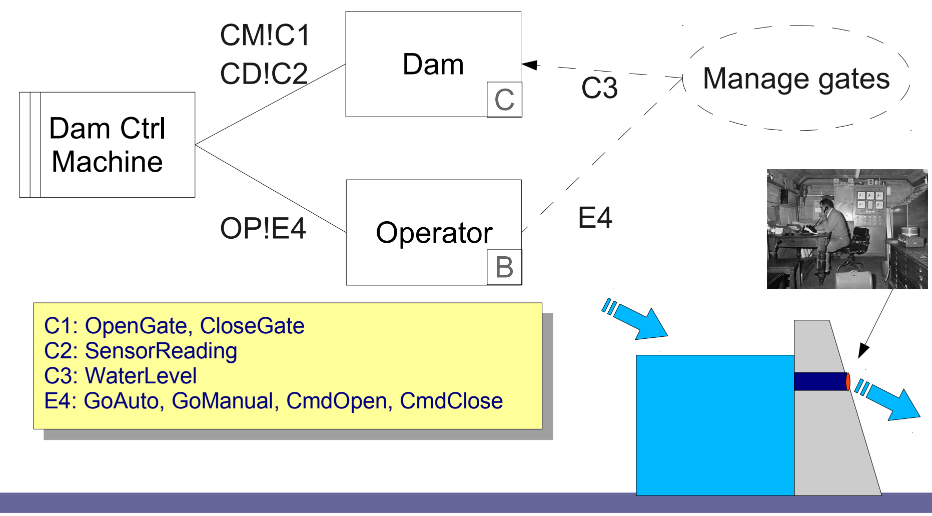
- Distributed systems are often (but not necessarily) spread over several locations
- But each node has
  - Computation capabilities
  - Memory

- Remote systems always have components spread over several locations
- But nodes can have
  - No computation capabilities
  - No Memory

Distributed vs. Remote

- Examples of Remote-but-not-distributed
  - An old-style mainframe connected via serial lines to “dumb” terminals (teletypes)
  - A control system whose electromechanical sensors and actuators are far away, connected through electrical wires, pressure tubes, etc.





Why distributed systems?

- They occur naturally
  - Any computer network is potentially a distributed system
- They are economically sensible
  - Distribute load among many cheap components
- They are more robust
  - Well-designed distributed systems are resilient to failures
- The problem at hand can be distributed
  - e.g., monitoring of water temperature across the oceans

A Word of Warning

- "Distributed" does not imply "Networking"
- Often, each node in a distributed system is a full computer
  - At times, a general-purpose computer fitted with a specific program
  - At times, a custom-built hardware or a device that can communicate through the network
- But it is not necessary, example:
  - In **SmallTalk** there are no programs, just systems
  - Each object has a private state, communicates via message passing (not method calls!) with other objects

Designing distributed systems

- Designing a distributed system is a matter of
  - **Plan**
    - Identifying the (expected) system functions
  - **Strategy**
    - Partitioning the functions among components
  - **Tactics**
    - Ensuring that components behave and communicate as expected
  - **Evaluation**
    - Verify that the full system delivers what is expected
    - Verify that the expected levels of performance, robustness, maintainability etc. are obtained

Designing ... ems

Requirements

- Designing a di... tributed sys...
  - **Plan**
    - Identifying the (exp... ...d) syste...
  - **Strategy**
    - Partitioning the fu... ...onents
  - **Tactics**
    - Ensuring that com... ...d communicate as expected
  - **Evaluation**
    - Verify that the full s...
    - Verify that the expected ... maintainability etc. are obtained

Design
Performance modeling

Development
APIs
Protocols
Testing

V&V (more testing)
Addressing frames concerns
Performance measuring

Designing ... ems

- Designing a di... ...ted syst...
  - **Plan**
    - ...d) syste...
    - Partitioning the fu... ...s
  - **Tactics**
    - Ensuring that con... expected
  - **Evaluation**
    - ...is of performance, robustness, maintainability etc. are obtained

Requirements

Design
Performance modeling

Problem frames (problem patterns)

Development
APIs
Protocols
Testing

Architectural patterns
Design patterns

V&V (more testing)
Addressing frames concerns
Performance measuring

Network programming

Architecture

- The **architecture** of a distributed system is a description of
  - Which **components** make up the system
    - Identity: how are components named/identified?
    - Functions: which services does a component offer?
    - Properties: features and values
  - Which **connectors** are available between them
    - Identity: how are connectors named/identified?
    - Properties: features and values

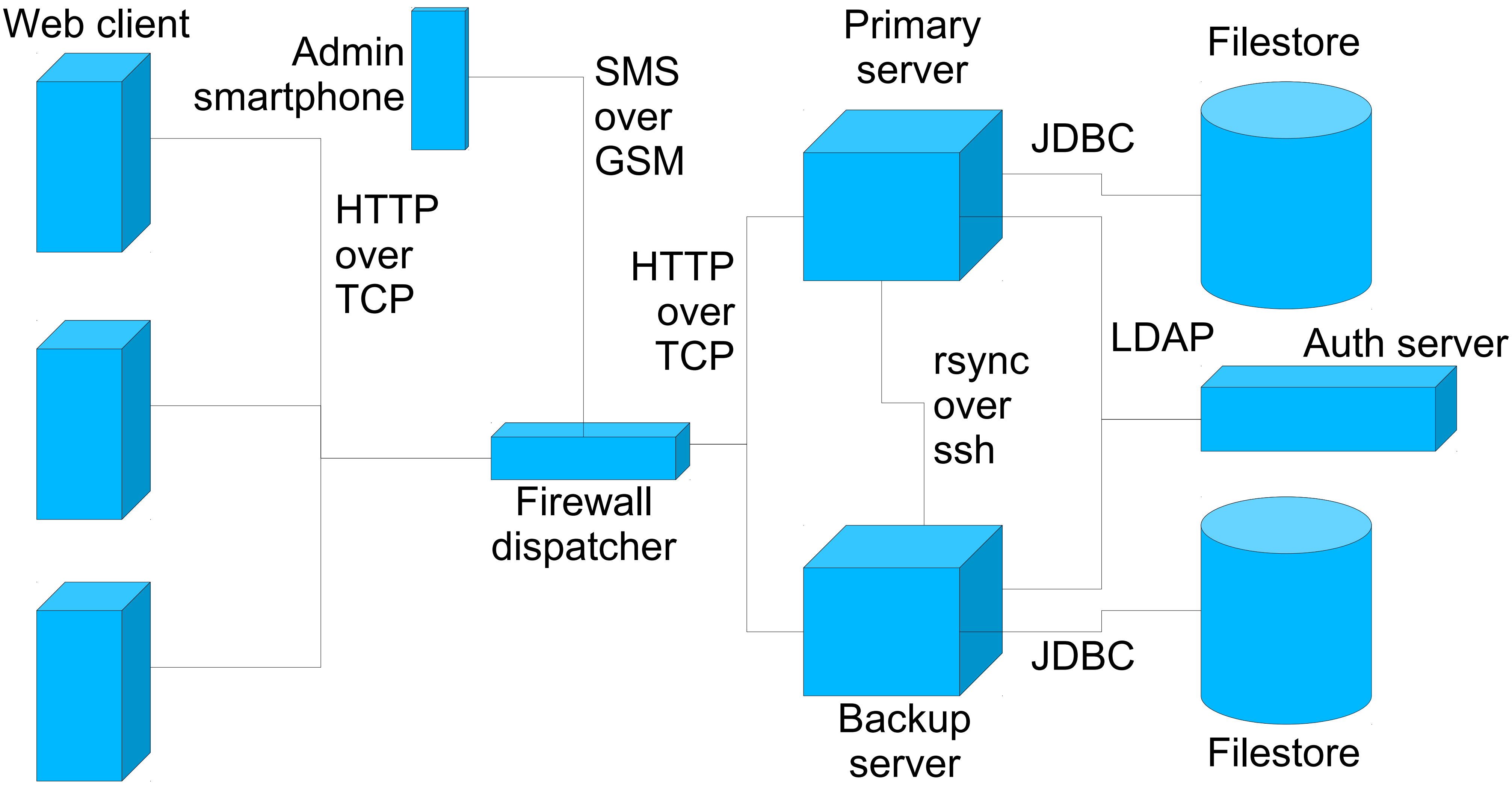
Components

- Typically, a full programmable computer, customized with some specific software
  - Same concept as that of *machine* in the Problem Frames approach
  - Hint: Use PF to analyze the problem that each single component is supposed to solve!
- At times, specialized hardware
  - In all cases: independent computation and communication capabilities

Connectors

- Typically, some kind of networking infrastructure, provided with
  - Media layer standards: electrical, radio, optic, …
  - Communication protocols: TCP/IP, web services, CORBA, SMS, Bluetooth, USB, …
- Could be something less typical
  - Example: in object-oriented programming, objects are components, and method invocations are connectors
    - Call semantics provides connector specification

Example



Web client
Admin
smartphone
SMS
over
GSM
HTTP
over
TCP
HTTP
over
TCP
Firewall
dispatcher
Primary
server
JDBC
Filestore
LDAP
Auth server
rsync
over
ssh
Backup
server
JDBC
Filestore

Architecture

- We can talk about a **concrete architecture**
  - e.g.: the particular way a number of machines are set up and connected here at Polo Fibonacci to provide: shared homes, authentication service, print server, etc.
- We can talk about a **class of architectures**
  - e.g.: an installation with a router, file server, print server, email exchange (all running copies of the same software) replicated at several sites
- We can talk about an **architectural style**
  - Ignoring specific details and adjustments

A few common architectures

- **Monolythic**
  - A single machine, with some device
    - Device: no computation power **or** no local memory
- Not a distributed architecture
- Increasingly rare
  - And anyhow, not in the scope of interest

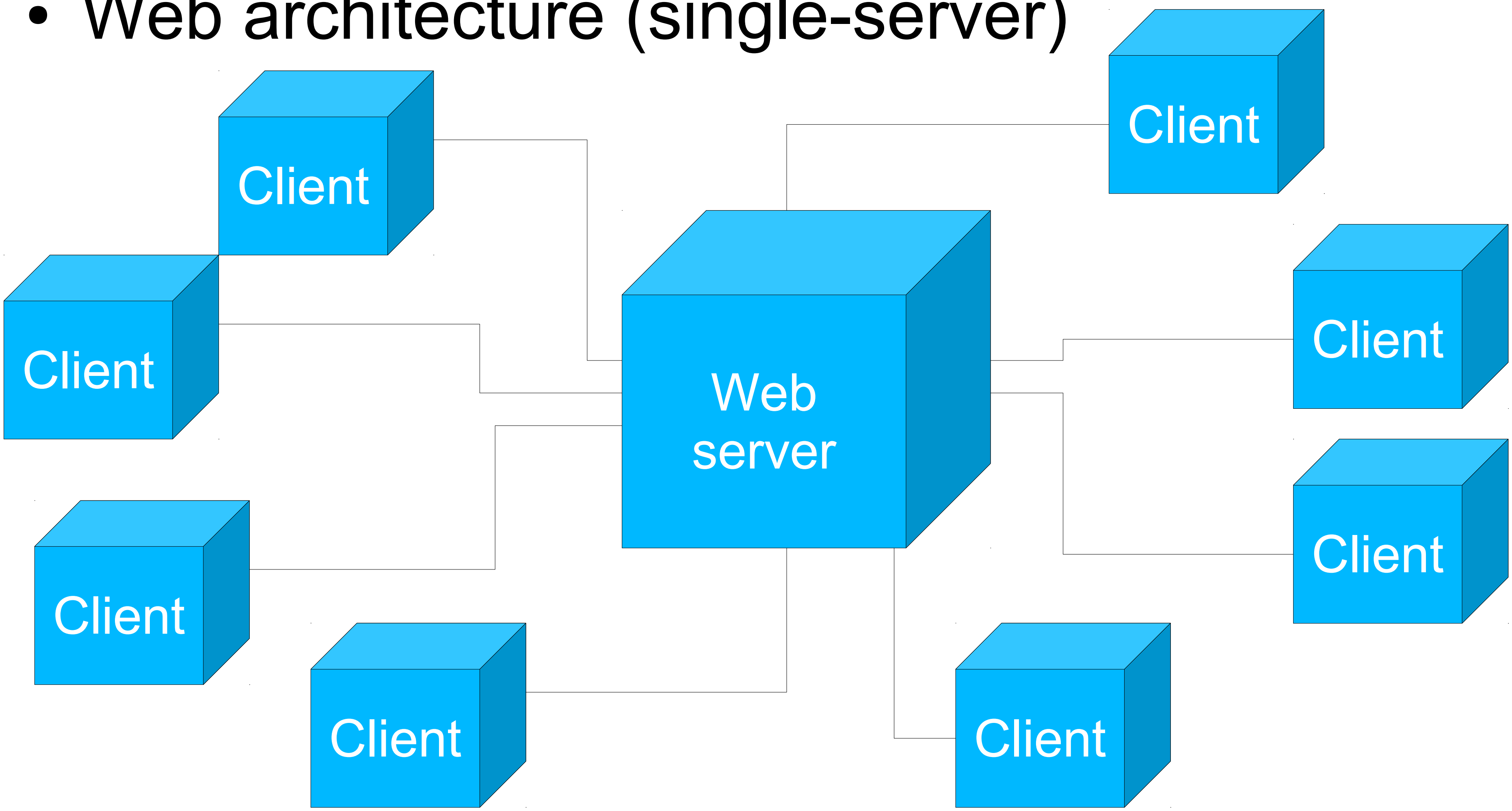
Booooring...

A few common architectures

- **Client / server**
  - A generic term for any architecture where
    - Components are divided into **one server** and **one or more client**
    - The server is typically more powerful (faster, more memory, more storage, priviledged data, etc.) than the clients
    - The server offers **services** to the clients
    - Interactions are **initiated by clients**
    - The connectors form a **star** around the server
  - Found literally *everywhere!*

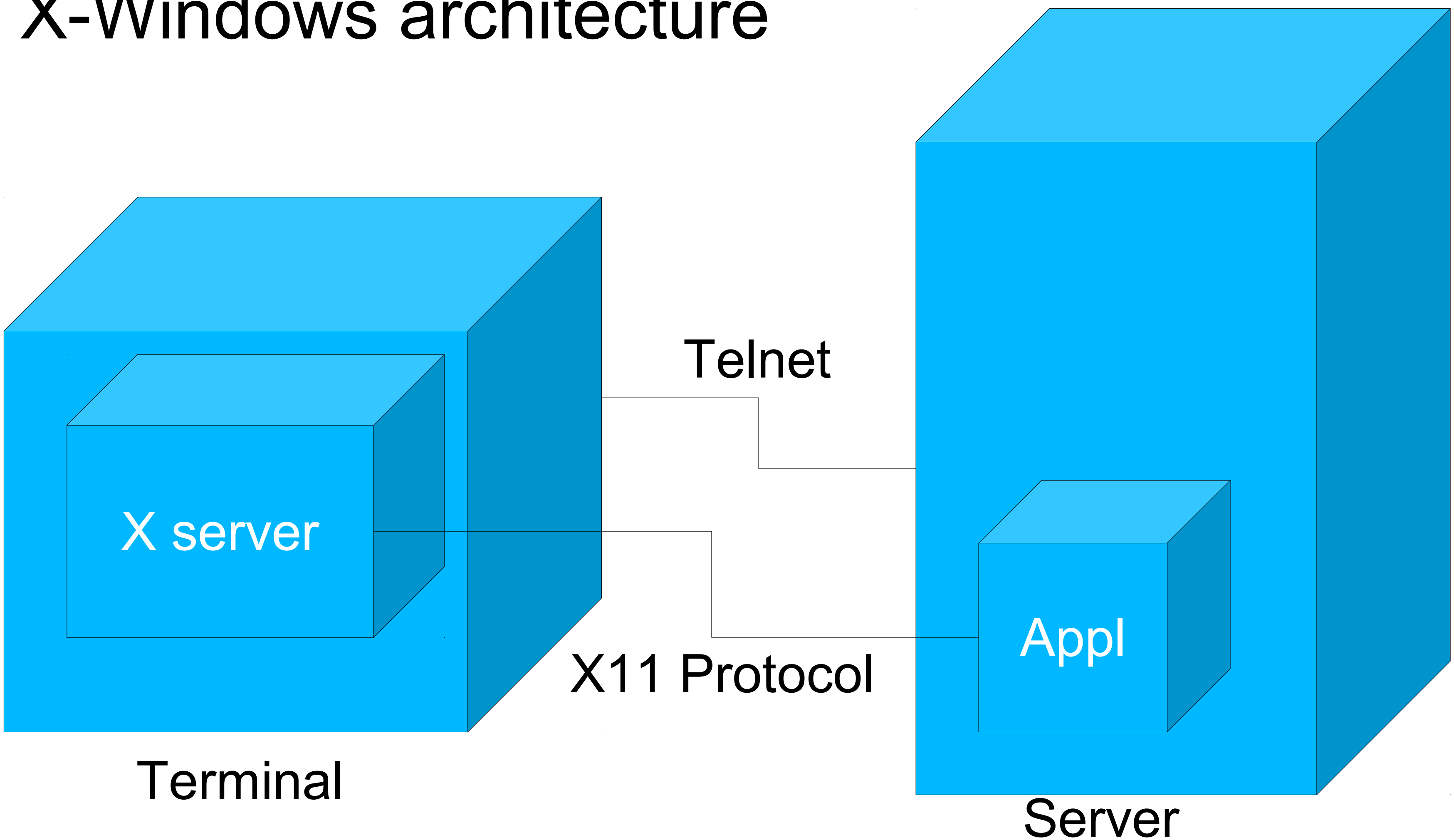
Example

- Web architecture (single-server)



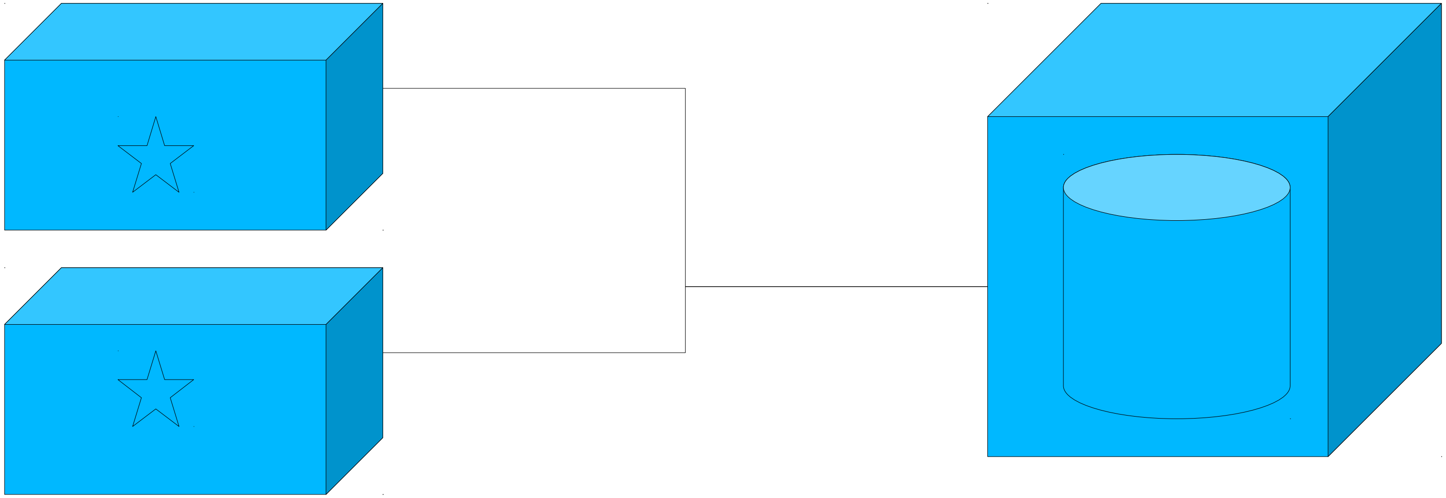
A less obvious example

- X-Windows architecture



A few common architectures

- **Two-tier architecture**
- Is essentially a client-server architecture, where
  - The server holds the data (storage)
  - The client performs the computation

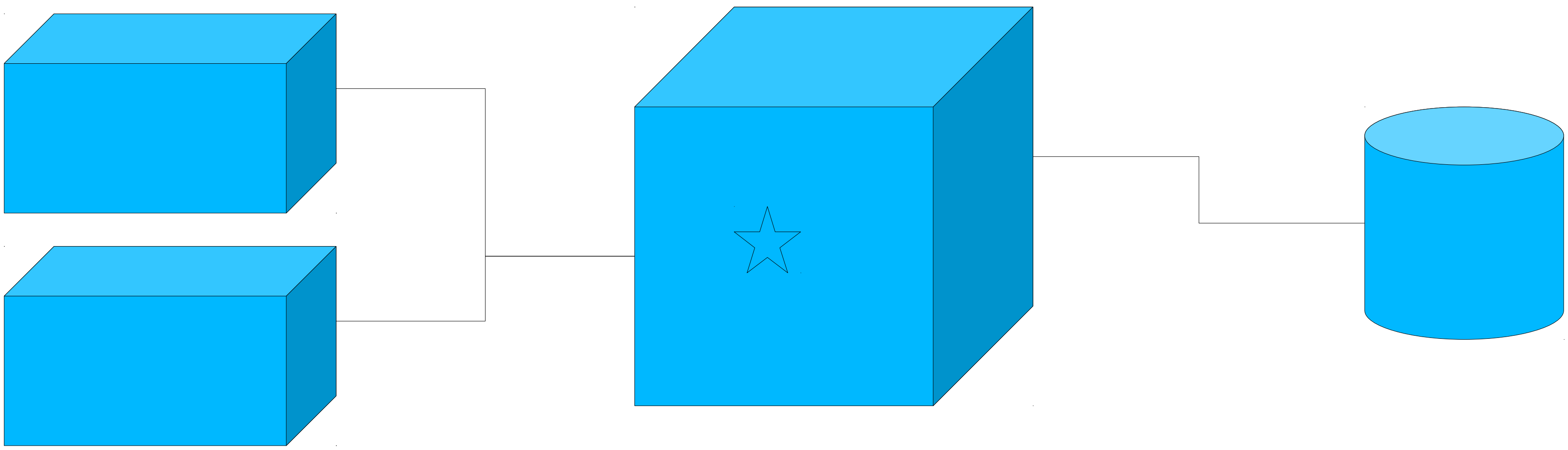


Two-tier architectures

- Advantages
  - Easy to implement for simple applications
  - Distributes computational load
- Disadvantages
  - Scalability may be an issue
  - Hard to update all clients when the computation changes
  - Clients must know the exact structure of the data
  - Might create heavy load on network with many clients and lots of data travelling around

A few common architectures

- **Three-tier architecture**
  - Three separate functions
    - User interface
    - Application logic (computation)
    - Storage (data)



Three-tier architecture

- Advantages
  - Easy to update application logic and data structure
  - Limited data transfer load
  - More scalable than two-tier
- Disadvantages
  - Mildly more complex
  - User interaction is limited by presentation layer
  - Computational load on server can be severe

Example

- **Web applications** are often three-tier architectures
- The **presentation** component is a web browser running on a client machine
  - e.g.: Google Chrome + Javascript
- The **application server** hosts the application
  - e.g.: Apache Tomcat + Java + Servlet/EJB
- The **storage** component hosts a DBMS
  - e.g.: MySQL + a RAID or NAS

A few common architectures

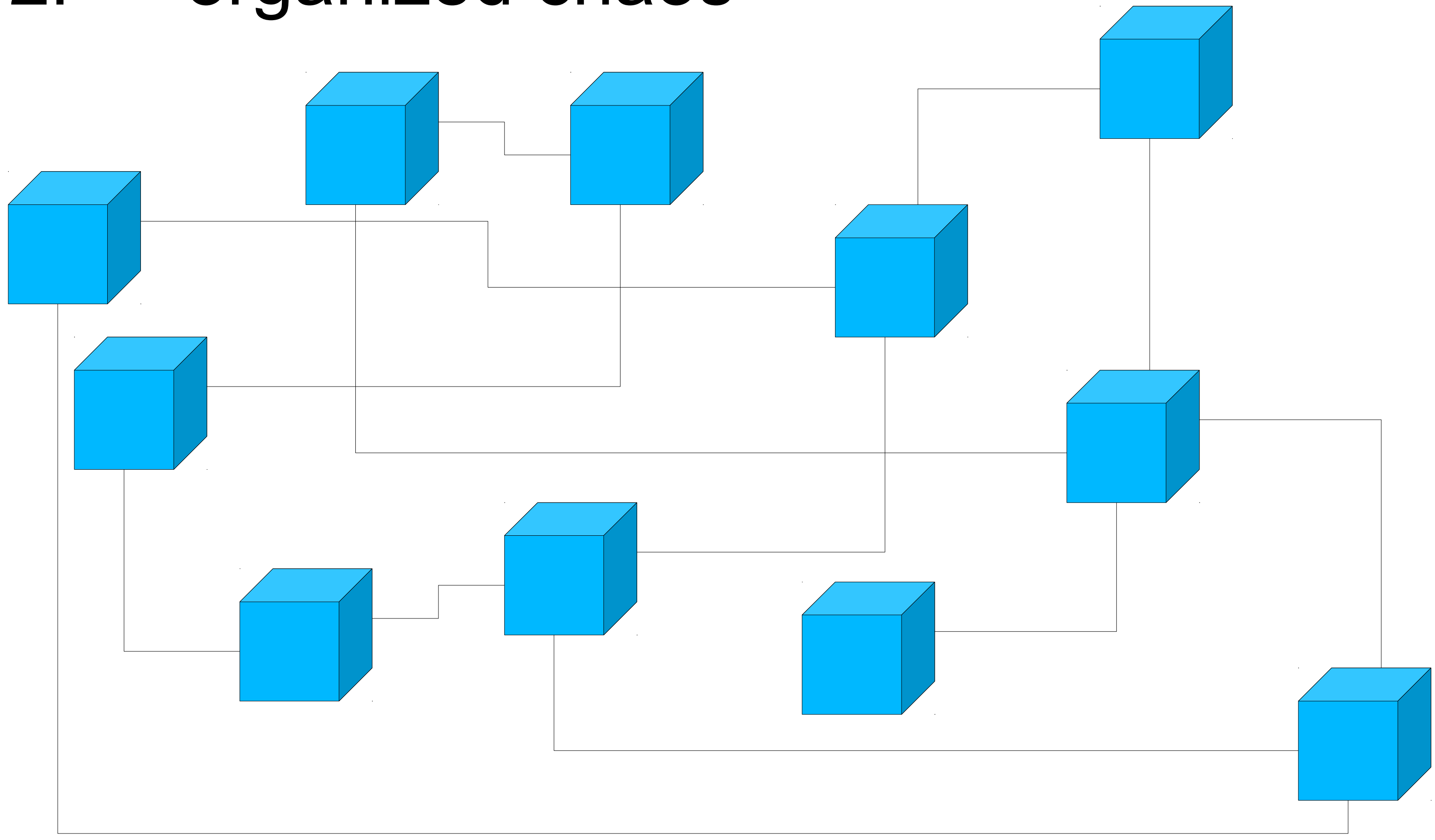
- **Peer to peer** (P2P) **architecture**
- Each component is a the same time a client and a server (a **peer**)
- Each node offers and consumes the same services
- There can be several layers of peers
  - Nodes, super-nodes, hyper-mega-ultra-super-...
  - Hosting files, hosting indexes, hosting catalogues of nodes hosing indexes, etc.

Example

- Most file-sharing software is based on P2P
  - **eMule**, **Kazaa**, **BitTorrent**, etc.
- But also less visible P2P
  - **Skype** is based on a dynamically reconfigured, multi-layer P2P architecture
  - **Ordinary nodes** run the Skype application
  - **Supernodes** run the Skype application, are not firewalled, have good computational power and network connection
  - **Login server** (centralized) authenticates users

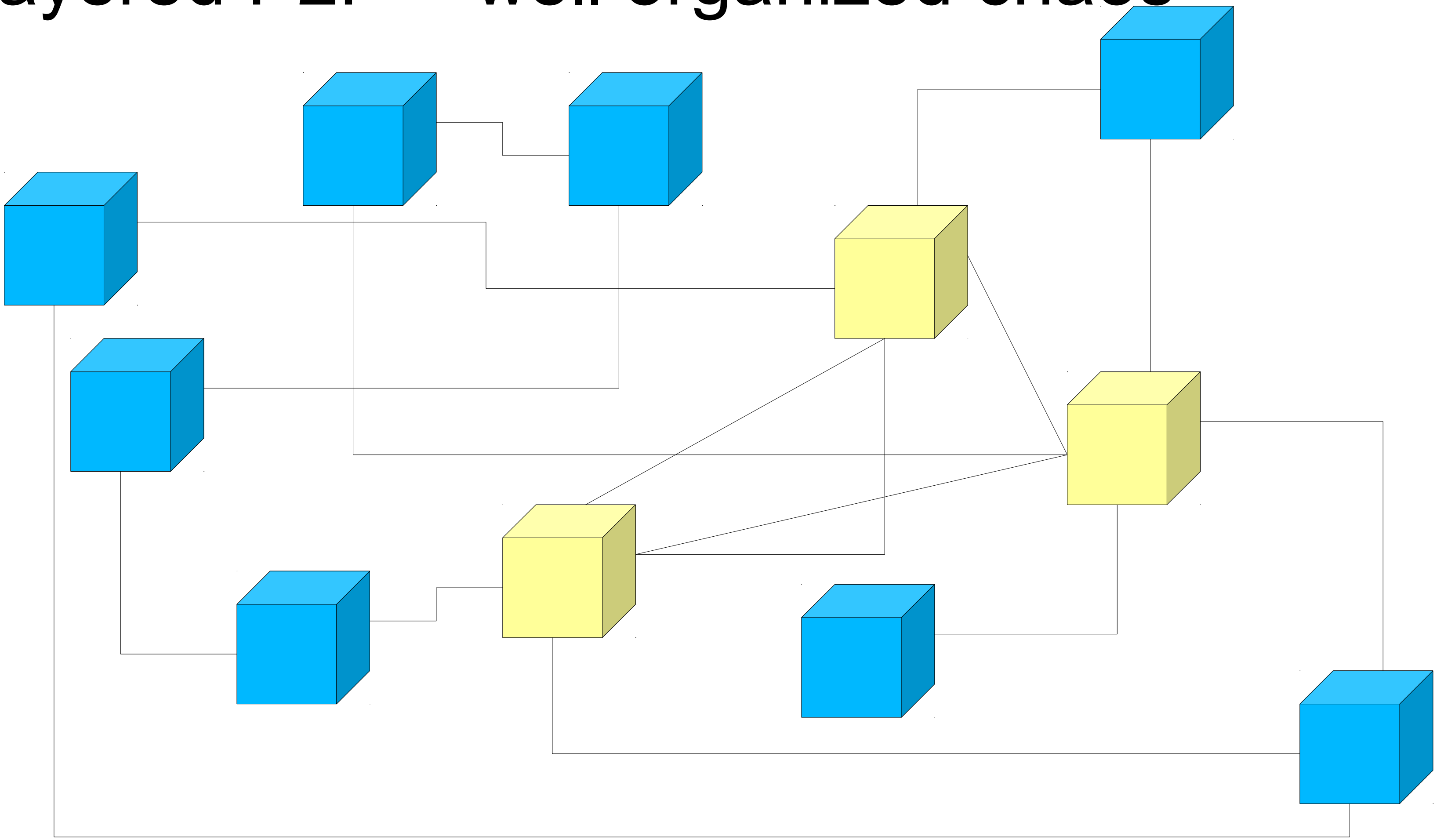
Example

- P2P = organized chaos



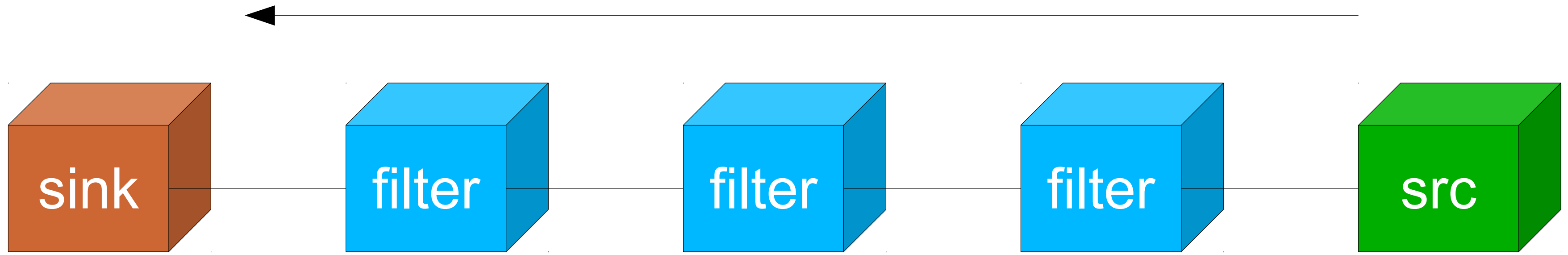
Example

- Layered P2P = well organized chaos



A few common architectures

- **Pipe & Filter** architecture
  - Components perform different functions, but have common interfaces
  - Components that produce data: **sources / wells**
  - Components that process data: **filters**
  - Components that consume data: **sinks**

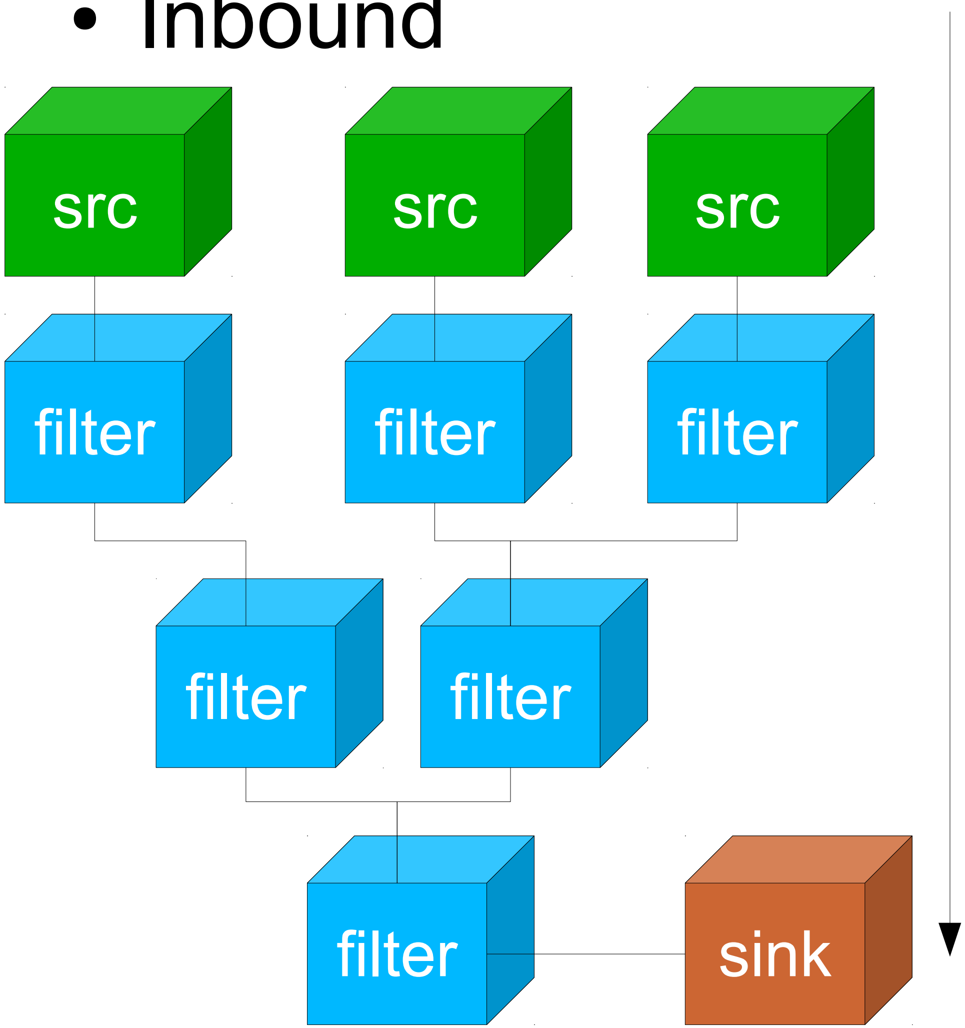


A few common architectures

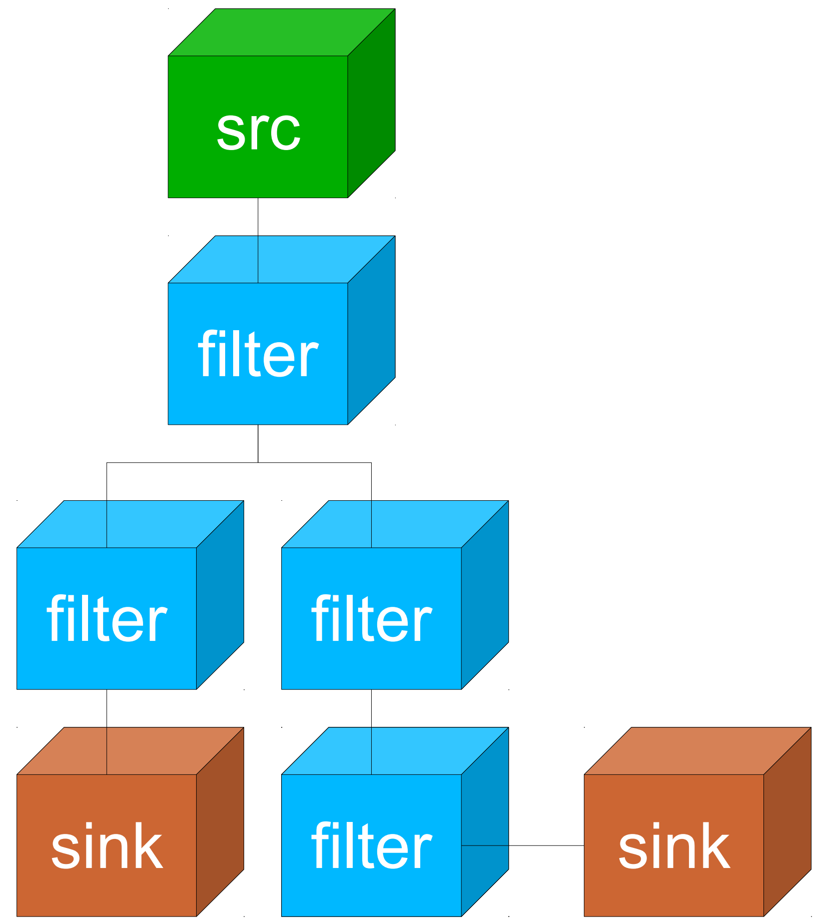
- Straight pipe & sink are not common
  - Too many machines for single task
  - Can be useful to distribute load for a CPU-intensive task
  - E.g.: re-encoding large video libraries
- Variations exist where **inbound** or **outbound** trees are used
- Inbound: collecting and processing data from multiple sources
- Outbound: distributing data to many clients

Example

- Inbound

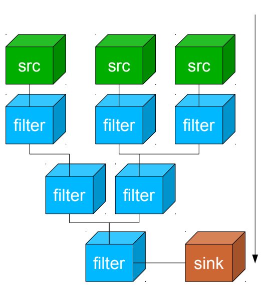


- Outbound

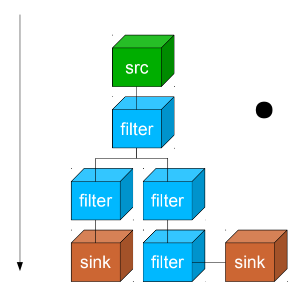


Example

- Inbound
  - Collecting processed data from a number of sensors distributed geographically
    - Ocean monitoring
  - Taking decisions based on a number of different variables
    - Volcano alert
    - Stock market analysis

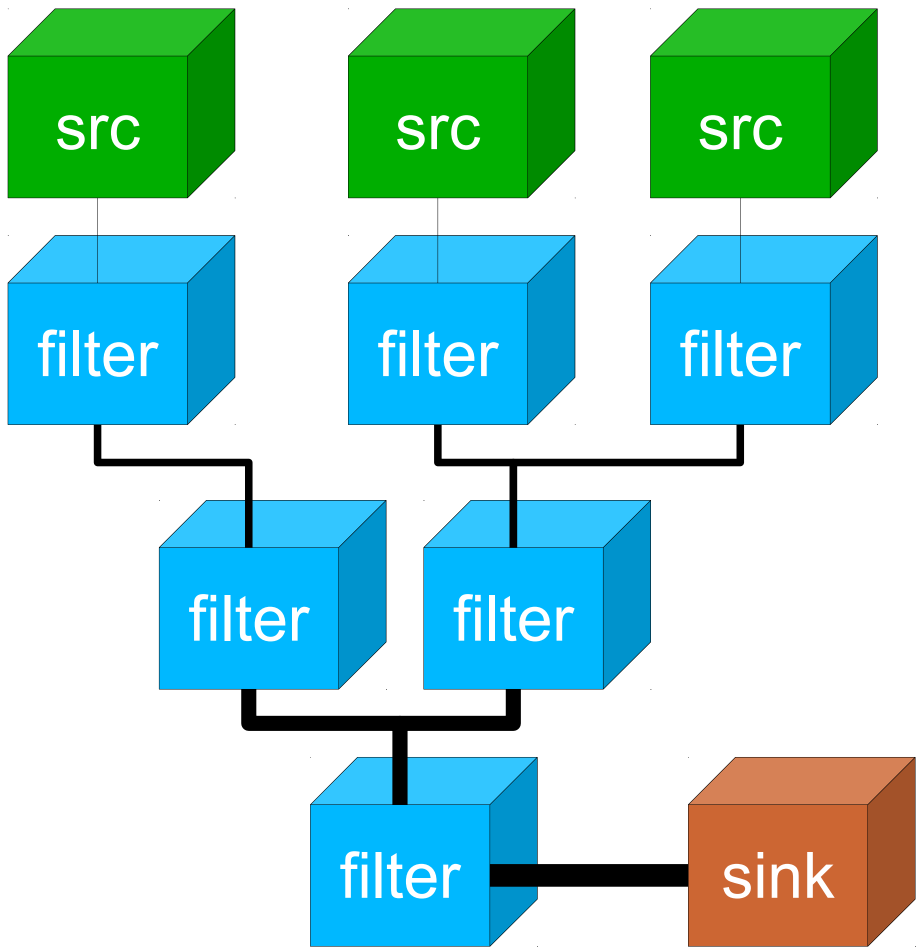


- Outbound
  - Serving processed data to many clients
  - Each client can perform further customized analysis
    - Providing data from a LHC experiment to various teams
    - Providing intelligence “signals” to CIA and FBI



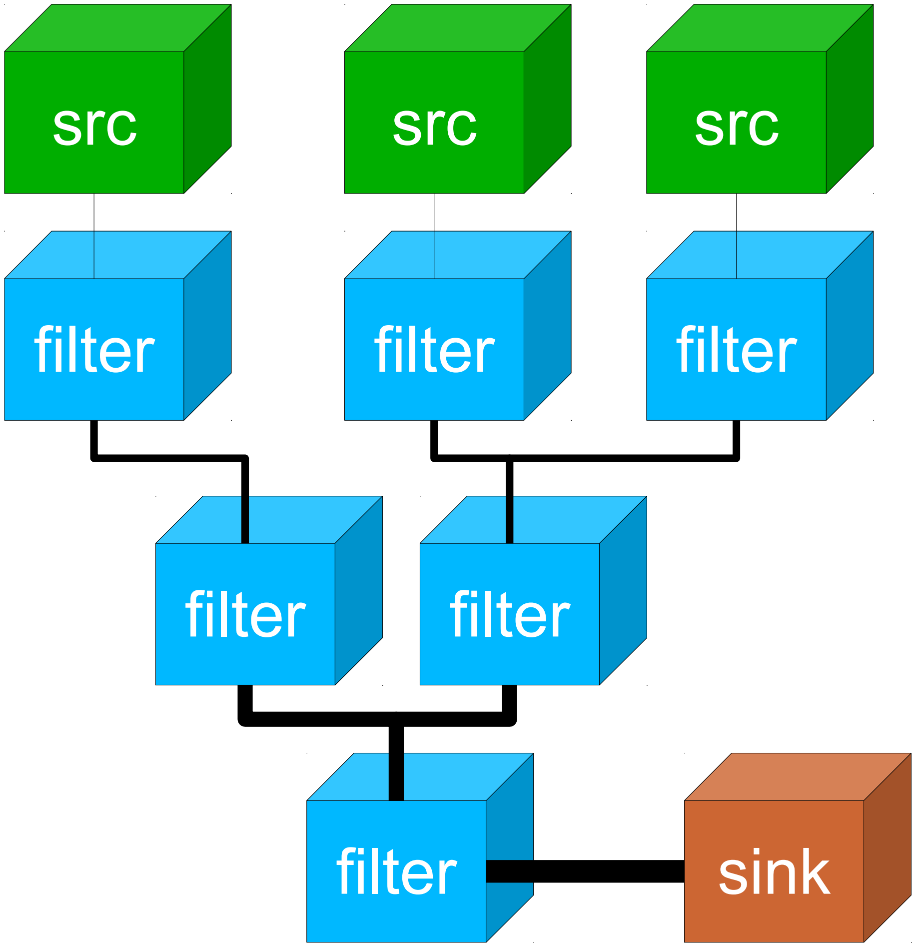
A few common architectures

- **Fat tree architecture**
  - Similar to a tree-based pipe and filter
  - But, connectors closer to the **root** have higher capacity than connectors at the nodes
- Inbound & outbound versions as expected



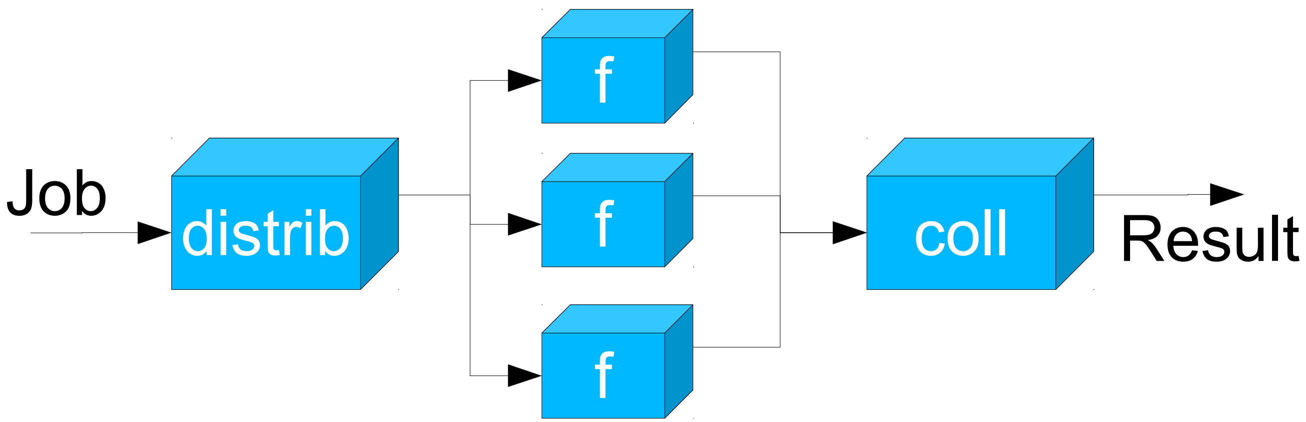
A few common architectures

- **Fat tree architecture**
  - Used when the amount of data is large (i.e., filters are not very selective)
  - Good scalability
  - Scarce re-usability (cannot re-configure dinamically)



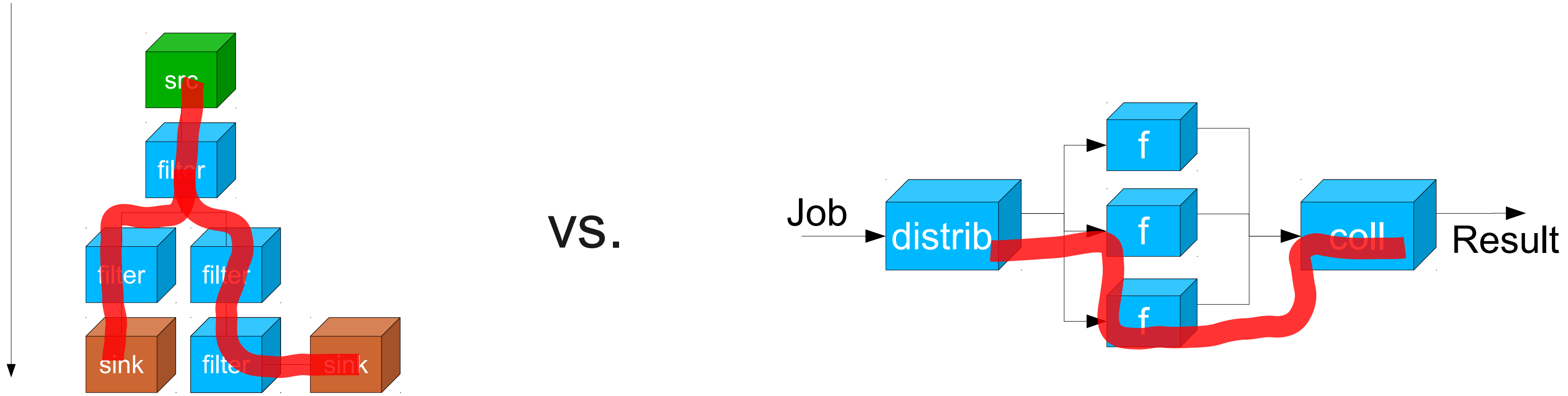
A few common architectures

- **Farm architecture**
  - A node acts as **distributor**
  - Any number of identical **functional nodes** perform the computation
    - Load sharing, resilience to faults
  - A node acts as **collector**



Pipe & Filter vs. Farm

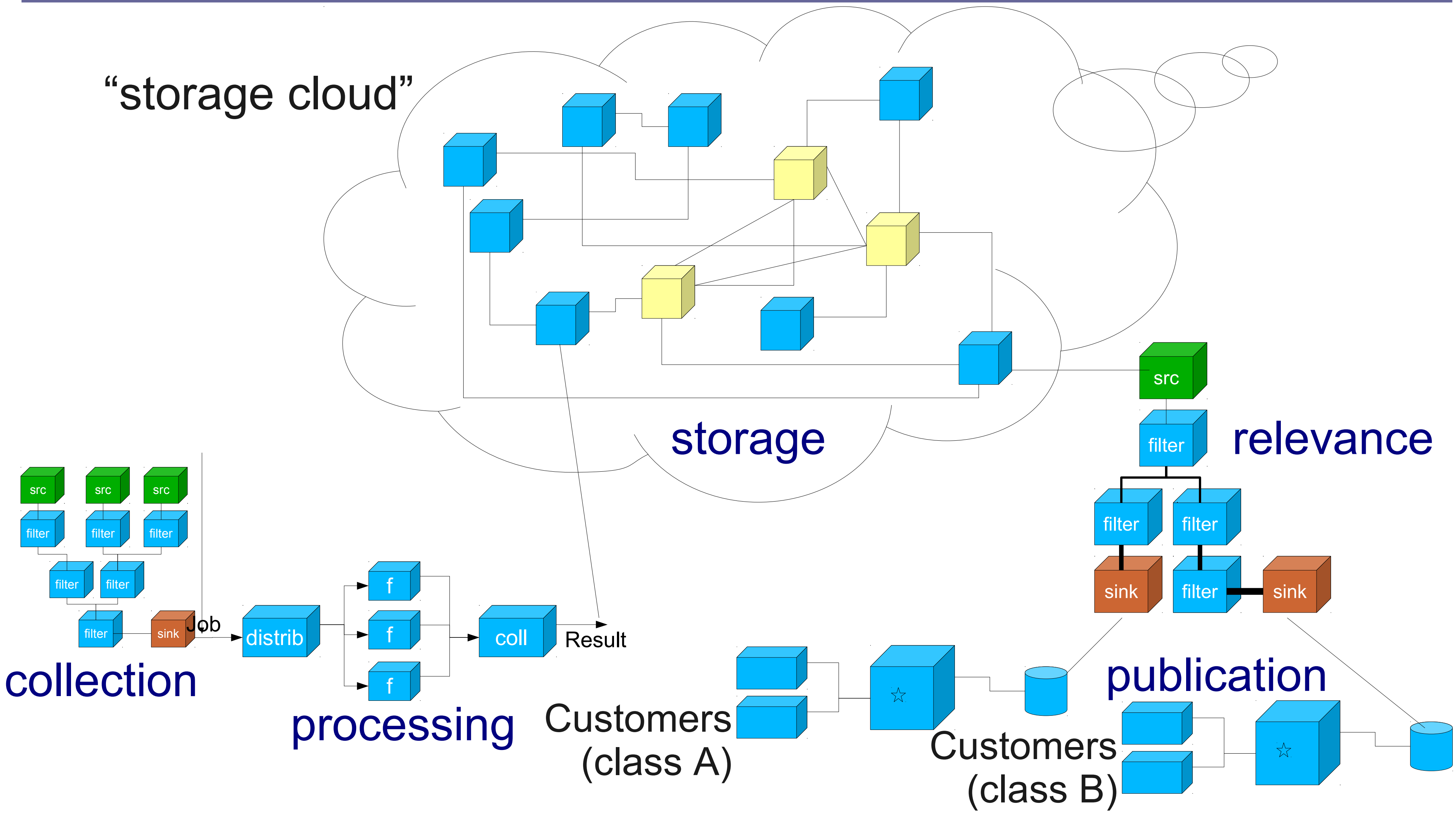
- In tree-based P&F, each message is **replicated** to all connected nodes
  - Multiple processing for same data
- In Farm, each message is sent to **just one** of the connected nodes
  - Same processing for multiple data



Combinations

- Most often, complex architectures are built out of a combination of the previous styles
  - Together with ad-hoc, application-specific solutions
- Example: Distributing financial analyses
  - An inbound P&F to get market data from all over the world
  - A farm to process each piece of data
  - An “inner” P2P storage layer (for robustness)
  - An outbound fat tree to distribute analyses to customers
  - A three-tier “outer” layer to present analyses to users (graphics and layout)

Combinations



Final recommendation

- The wider the repertoire of proven techniques a designe knows, the easier to envision a solution
- Most problem require **wise combination** of known techniques
- Some problem require **creativity** – Imagine!
- **Always** keep in mind:
  - Properties of components
  - Properties of connectors
  - Constraints & Concerns