Software Architecture

Jianyong Wang(王建勇)

Department of Computer Science and Technology
Tsinghua University, Beijing, China

Outline

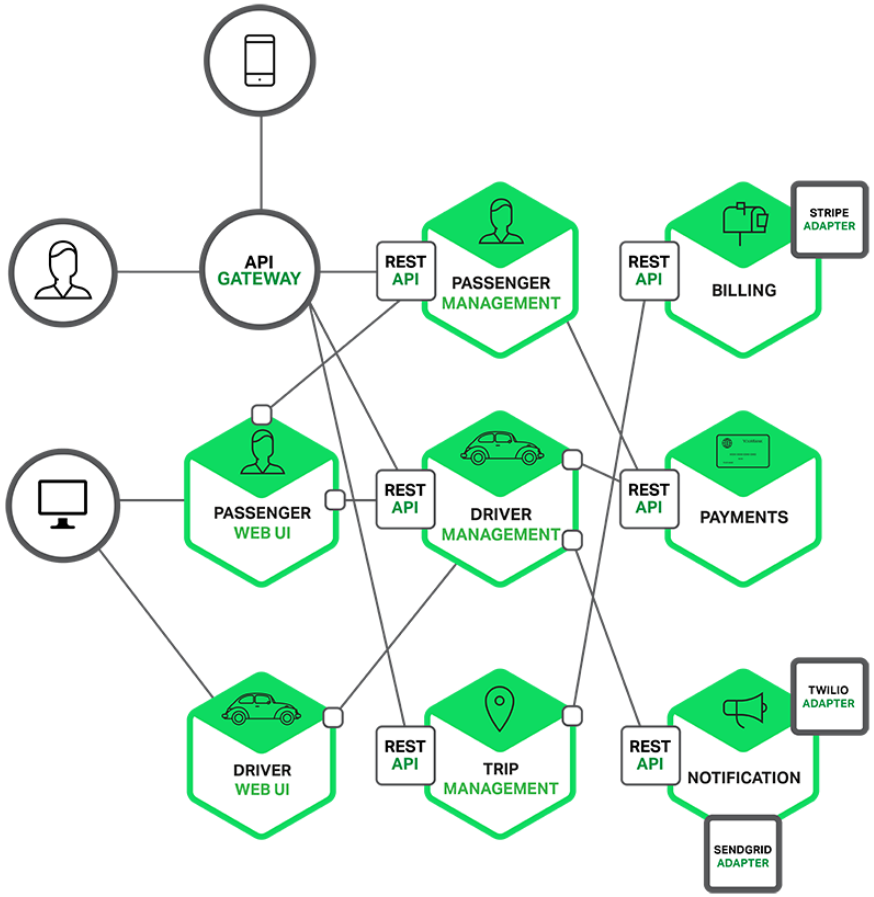
- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

Architecture and Quality Attributes

Software Architecture Definition

"The fundamental organization of a system, embodied in its *components*, their *relationships* to each other and the *environment*, and the *principles* governing its design and evolution."

ANSI/IEEE Std 1471-2000,
Recommended Practice for Architectural Description of Software-Intensive Systems



Why is Software Architecture Important?

- Architecture acts as the skeleton of a system
  - There is no single right architecture, but there are more or less suitable skeletons for the job.
  - E.g. it is hard to imagine a distributed VOIP network like Skype using anything other than a peer-to-peer architecture.
- Architecture influences quality attributes (e.g., performance, availability, modifiability, etc)
  - Beyond supporting required functionality, a system's architecture enables or inhibits qualities.
    - E.g. a system that was designed to support a hundred users may be impossible to scale up to a hundred thousand without an architectural change.

[George Fairbanks, 2010]

Performance

Performance

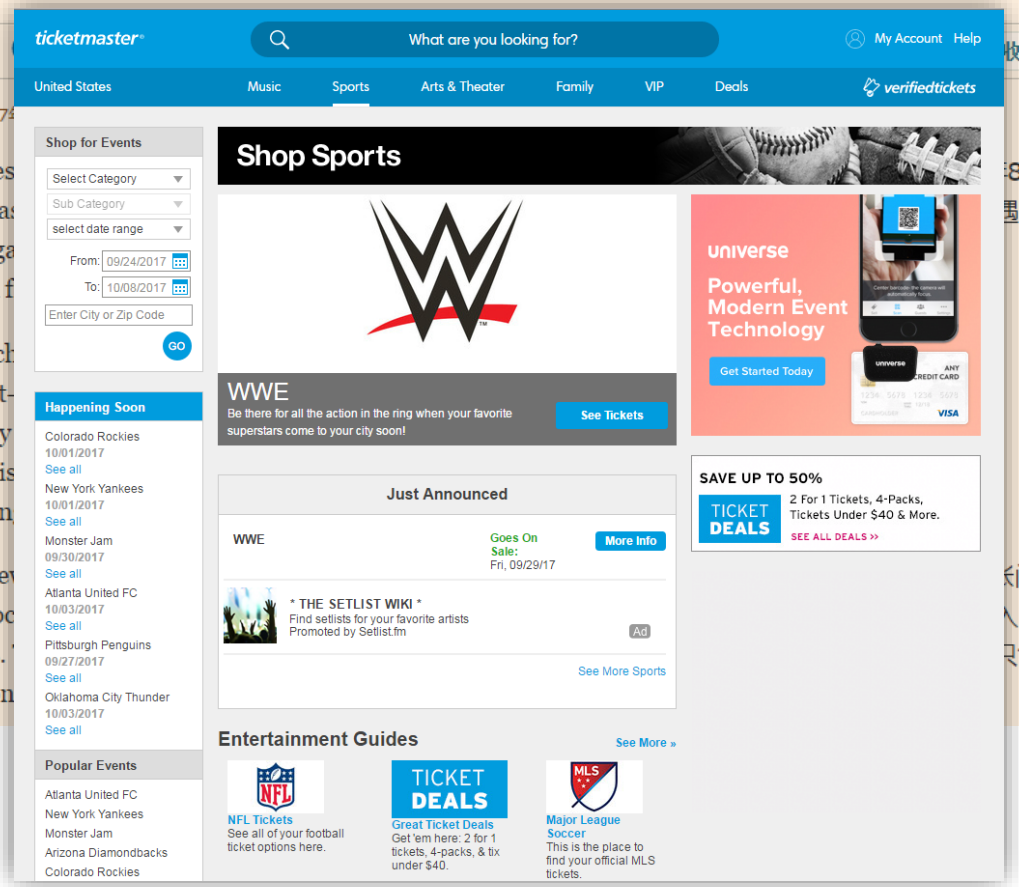
- Performance is about time and the software system's ability to meet timing requirements.
- It is a pervasive quality of software systems: everything affects it, from the software itself to all underlying layers, such as operating system, middleware, hardware, communication networks, etc.
- For much of the history of software engineering, performance has been the driving factor in system architecture.

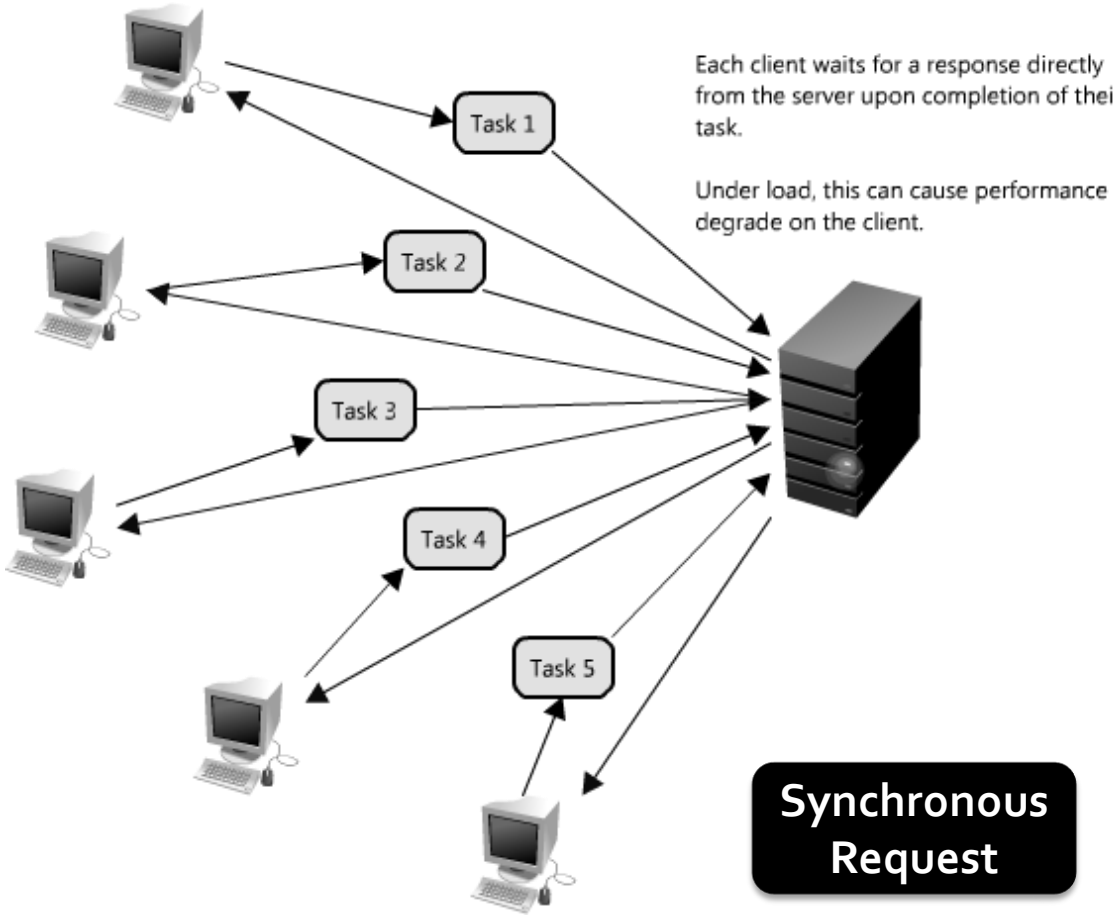
Example Performance Design

DEMAND CRASHES BEIJING'S OLYMPIC TICKETING SYSTEM

北京奥运订票系统出现故障

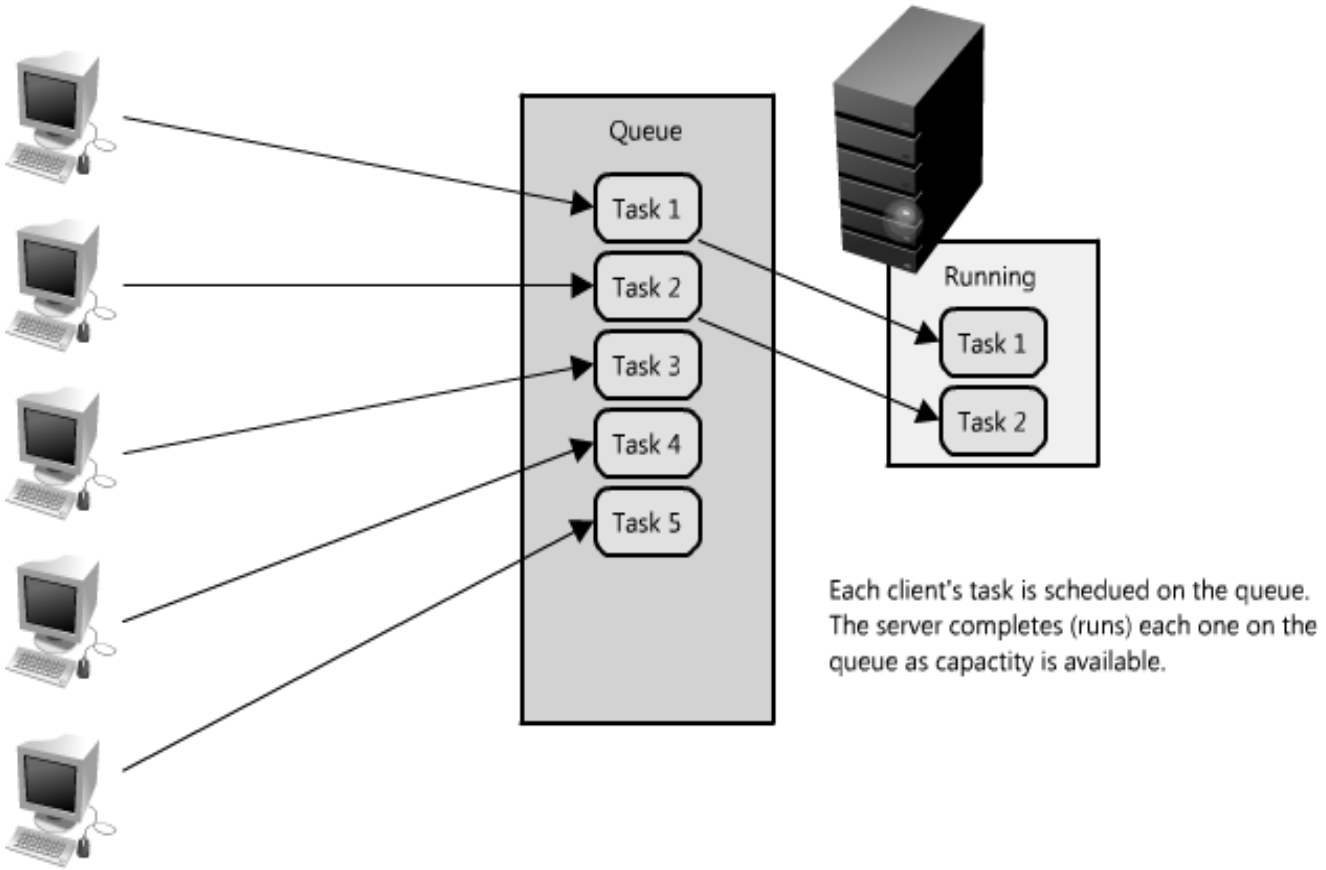
北京奥组委称，原因在于网上订票浏览量过大





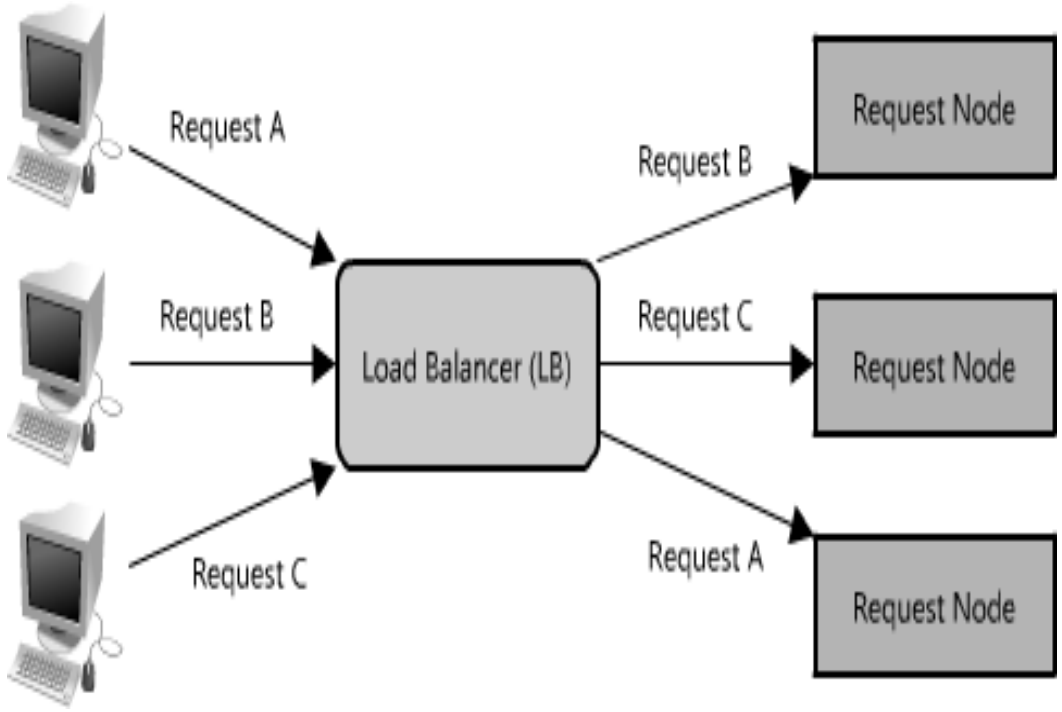
http://www.aosabook.org/en/distsys.html

Queues



Queues are fundamental in managing distributed communication between different parts of any large-scale distributed system, and there are lots of ways to implement them. There are quite a few open source queues like RabbitMQ, ActiveMQ, BeanstalkD, but some also use services like Zookeeper, or even data stores like Redis.

Load Balancer



Load balancers are a principal part of any architecture, which handle a lot of simultaneous connections and route those connections to one of the request nodes, allowing the system to scale to service more requests by just adding nodes.

Availability

Failure is INEVITABLE

- When designing a high-availability or safety-critical system, failure is not an option, it is almost INEVITABLE.
- What will make your system safe and available is planning for the occurrence of failure or (more likely) failures, and handling them with aplomb.

Availability

- Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval.

$$\frac{MTBF}{MTBF + MTTR}$$

MTBF: mean time between failures
MTTR: mean time to repair

Example System Availability Requirements

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8.0 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

"AWS will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage of at least 99.95% during the Service Year. In the event Amazon Ec2 does not meet the Annual Uptime Percentage commitment, you will be eligible to receive a Service Credit as described below..... "

Dependable Computing

Fault Prevention	• How to prevent the occurrence or introduction of faults
Fault Tolerance	• How to deliver the correct service in the presence of faults
Fault Removal	• How to reduce the number or severity of faults
Fault Forecasting	• How to estimate the current number, the future incidence and likely consequence of faults

Fault-Tolerant Architecture

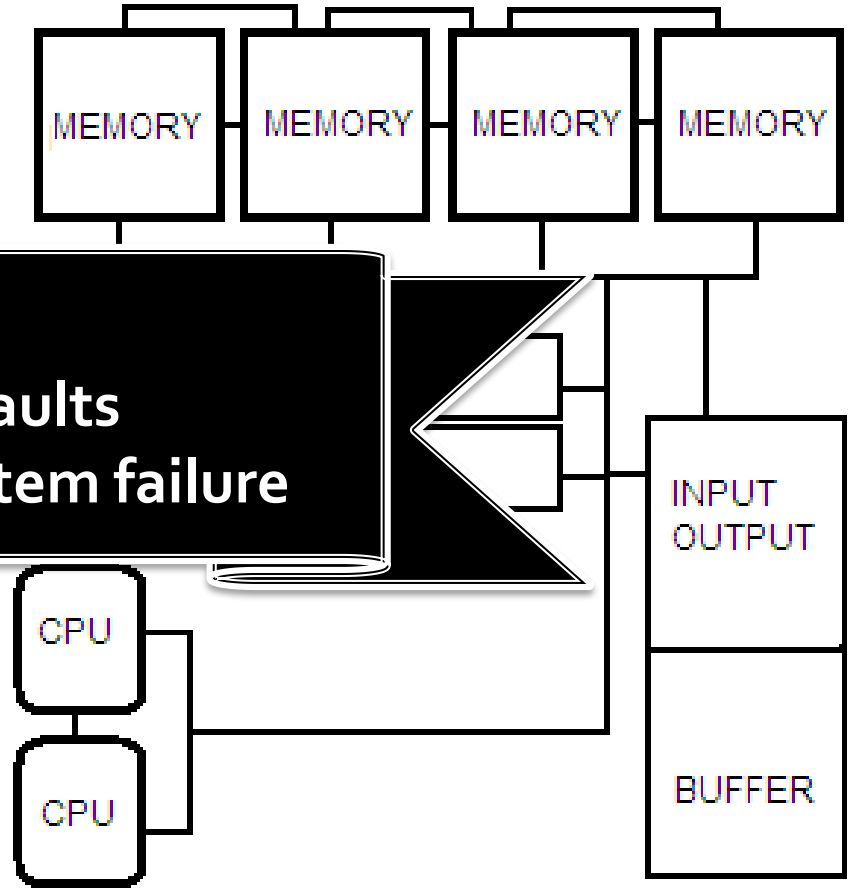
Fault-tolerance / graceful degradation

- The property that enables a system to continue operating properly in the event of the failure.
  - Masking re
    and Shannon,195

The Goal:
Ensure that system faults DO NOT result in system failure

Fault-tolerant computer systems

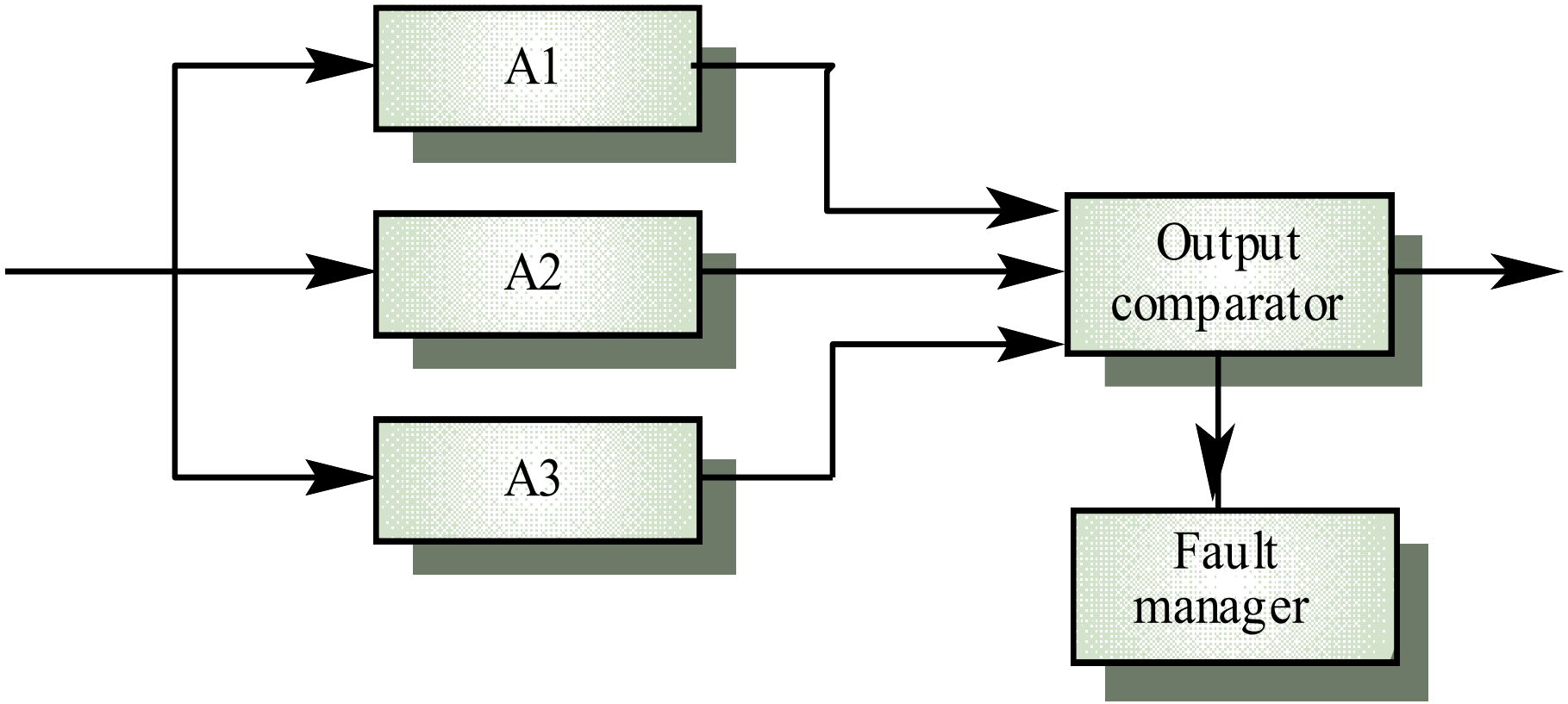
- systems designed around the concepts of fault tolerance.
  - Integrating masking with practical techniques of error detection, fault diagnosis, and recovery (A. Avizienis, 1967)
  - N-version programming (A. Avizienis, 1977)



Hardware Fault Tolerance Architecture

- Depends on triple-modular redundancy (TMR)
- There are three replicated identical components which receive the same input and whose outputs are compared
- If one output is different, it is ignored and component failure is assumed
- Based on the assumption: most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure

Hardware Reliability with TMR



Software Fault Tolerant Architecture

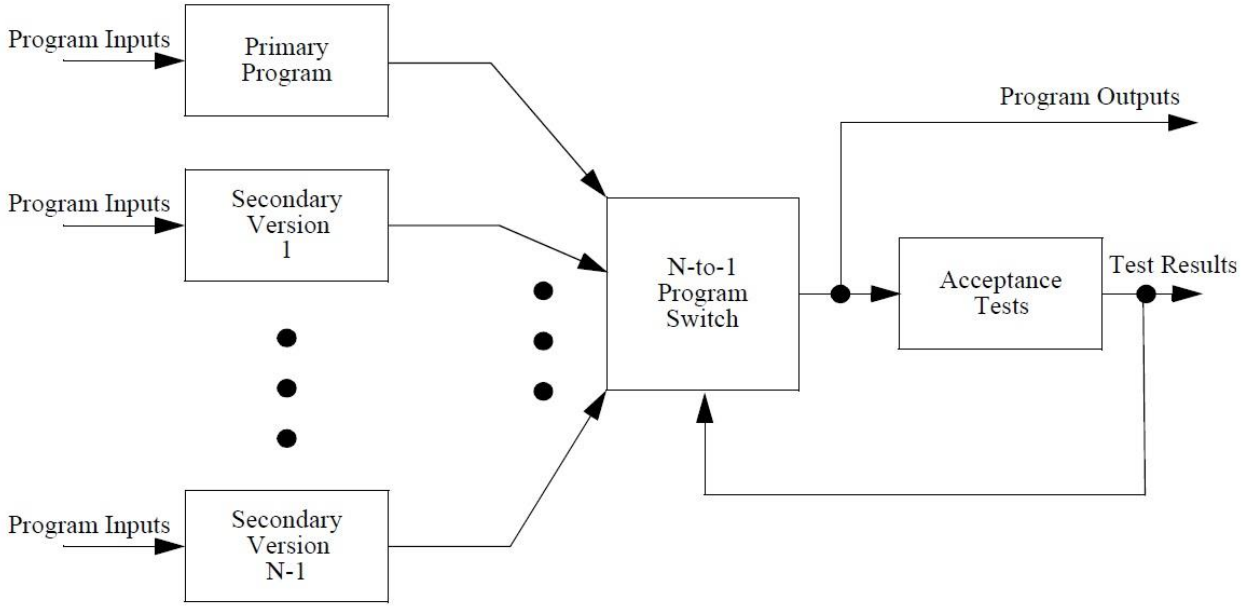
- The success of TMR at providing fault tolerance is based on two fundamental assumptions
  - The hardware components do not include common design faults
  - Components fail randomly and there is a low probability of simultaneous component failure
- Neither of these assumptions are true for software
  - It isn't possible simply to replicate the same component as they would have common design faults
  - Simultaneous component failure is therefore virtually inevitable
- Software systems must therefore be diverse

Design Diversity

- Different versions of the system are designed and implemented in different ways. They therefore ought to have different failure modes.
- Different approaches to design (e.g. object-oriented and function oriented)
  - Implementation in different programming languages
  - Use of different tools and development environments
  - Use of different algorithms in the implementation

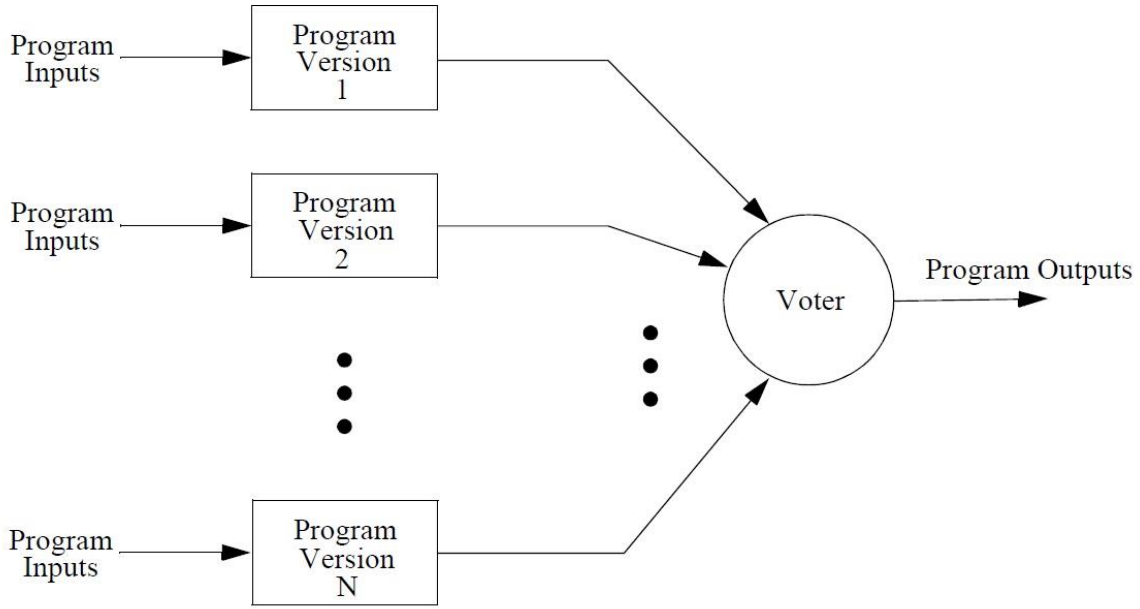
Software Analogies to TMR

- Recovery blocks
  - A number of explicitly different versions of the same specification are provided
    - One version is designated as the primary version, and the remaining *N*-1 versions are designated as spares, or secondary versions
  - The primary version of the software is always used unless it fails to pass the acceptance tests
    - Otherwise, the first secondary version is tried. This process continues until one version passes the acceptance tests, the system fails because none of the versions can pass the tests



Software Analogies to TMR

- N-version programming
  - The same specification is implemented in a number of different versions by different teams. All versions compute simultaneously and the majority output is selected using a voting system.
  - This is the most commonly used approach e.g. in Airbus 320.



N-Version Programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.

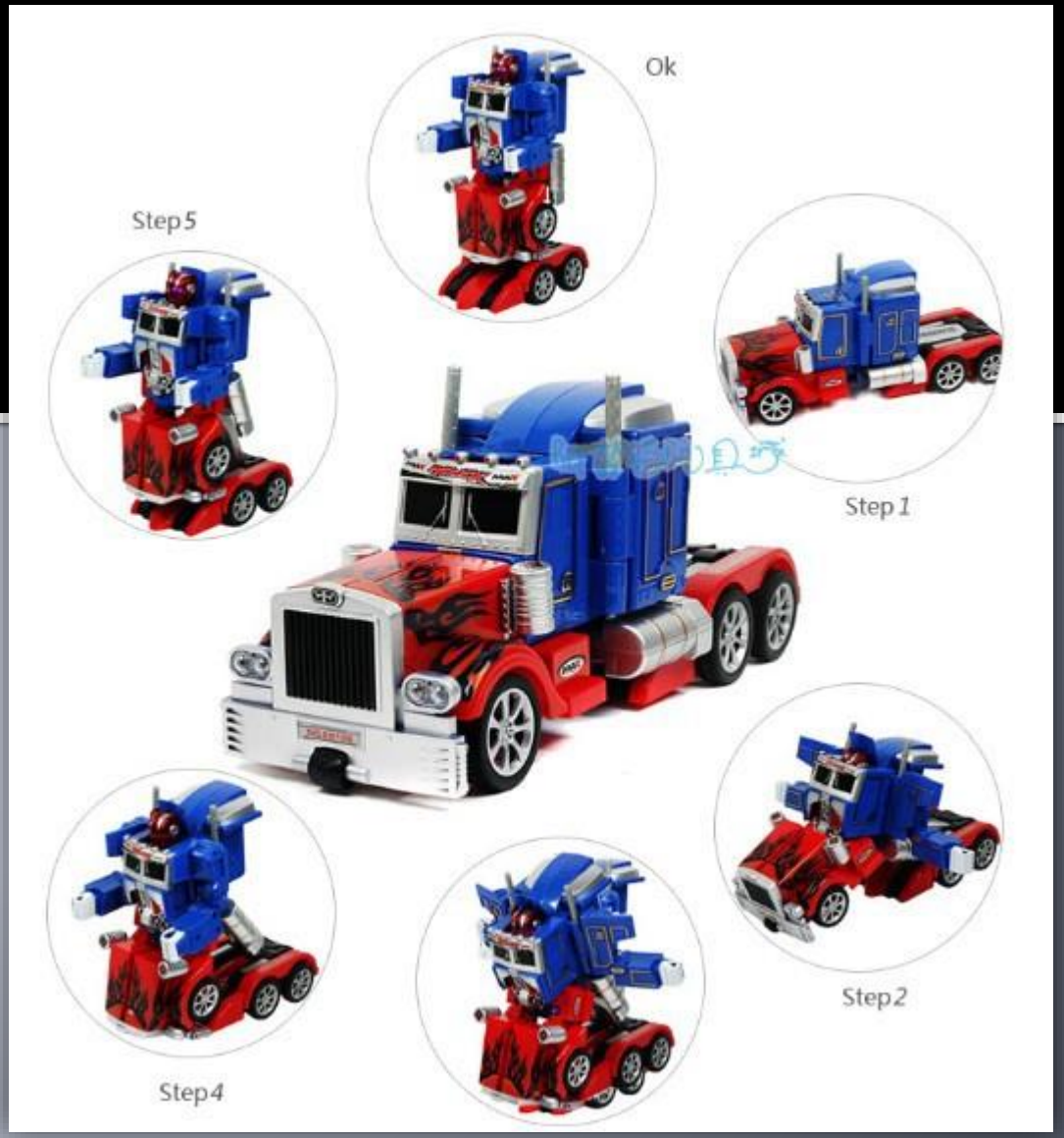
- There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Problems With Design Diversity

- Teams are not culturally diverse so they tend to tackle problems in the same way
- Characteristic errors
  - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place.
  - Specification errors
    - The *N* versions of a program are still developed from a common specification
    - If there is an error in the specification then this is reflected in all implementations
    - This can be addressed to some extent by using multiple specification representations

Modifiability

Modifiability is about change, and our interest in it centers on the cost and risk of making changes.



Change is INEVITABLE

- If change is the only constant in the universe, then software change is not only constant but ubiquitous.
- Changes happen
  - to add new features, to change or even retire old ones.
  - to fix defects, tighten security, or improve performance.
  - to enhance the user's experience.
  - to embrace new technology, new platforms, new protocols, new standards.
  - to make system work together, even if they were never designed to do so.

The CelsiusTech Story

- CelsiusTech Systems, as Swedish defense contractor supplying shipboard command and control systems to navigate around the world
- The monumental crisis in 1985
  - Compelled to build two systems much larger than anything the company had ever attempted
  - Had trouble meeting schedules and budgets before
- The motivation to an innovative solution
  - One solution that would satisfy the (very different) requirements of both customers
  - The solution will also satisfy customers that come after the two.
  - A new "architecture" with the capability across a wide (but planned) range of products.

The CelsiusTech Story

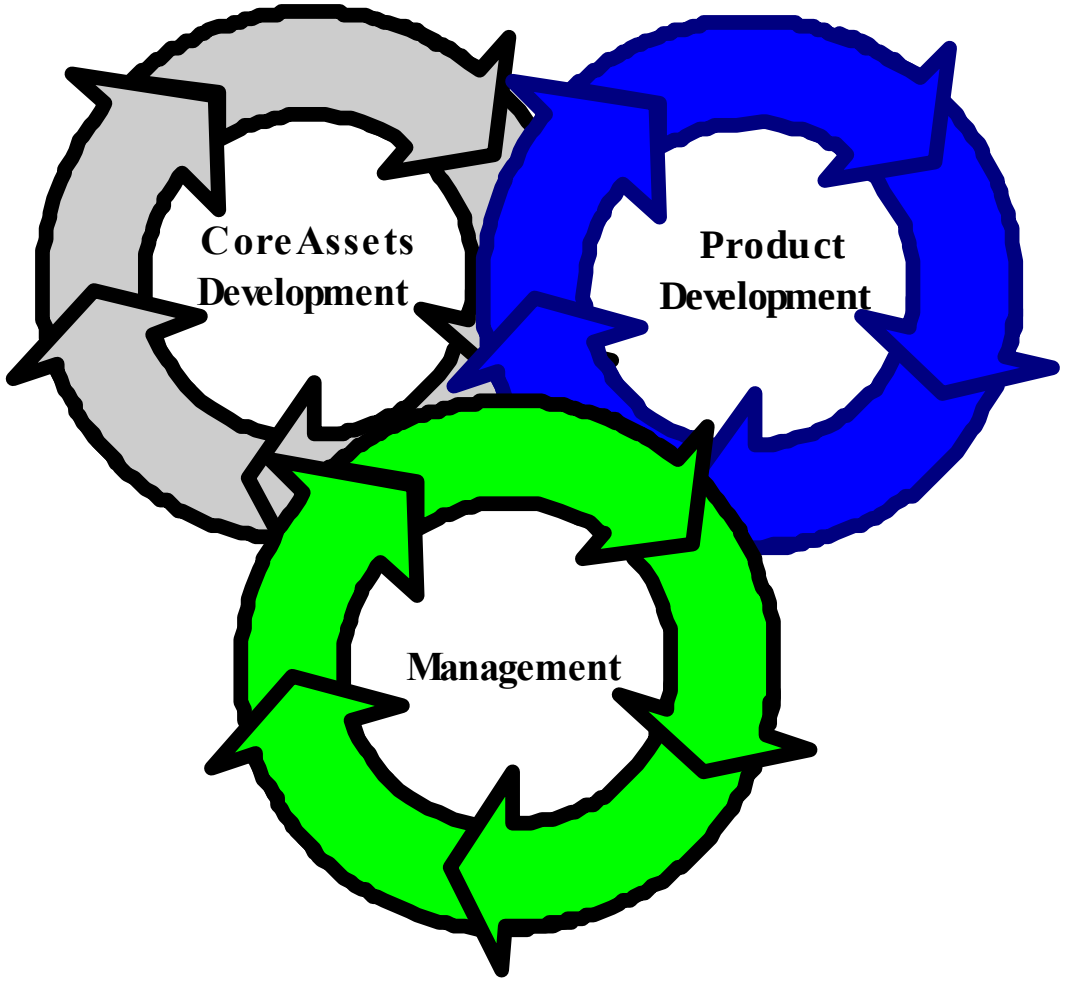
- The result of the new strategy called "Product line"
  - Not only did it allow the company to deliver both pilot systems, but on subsequent projects (more than 50 of them)
    - Took years off delivery schedules
    - Allowed a smaller staff to produce more systems
    - Software reuse levels into 90 percent
  - Examples
    - Integration testing of one of these enormous (1.5 million lines of Ada) real-time, safety-critical, highly distributed systems can be routinely handled by one or two people at most
    - Porting one of these system to a whole new computing environment and operating system takes less than a month.

Product Line

"A software product line (or an Application Family) is a set of software-intensive systems *sharing a common, managed set of features* that satisfy the specific needs of a particular market segment or mission and that are developed from *a common set of core assets* in a *prescribed way*."

- A related set of applications that has a common, domain-specific architecture
- Common core asset base is reused each time a new application is produced
- Each application is an instantiation of the core assets, specialized in some way

Three Essential Activities



- Feedback Loop
  - **Products are built from core assets**
  - **Assets are extracted, evolved, and generalized**
  - Visionary management to invest resources, coordinate new production and assets availability
- Iteration
  - **Core assets accumulation**
  - Products creation
  - Coordination

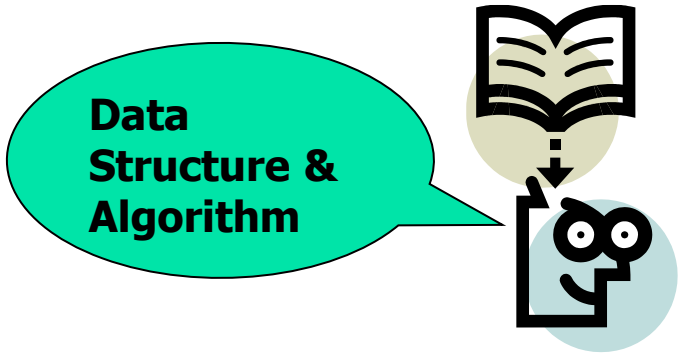
Outline

- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

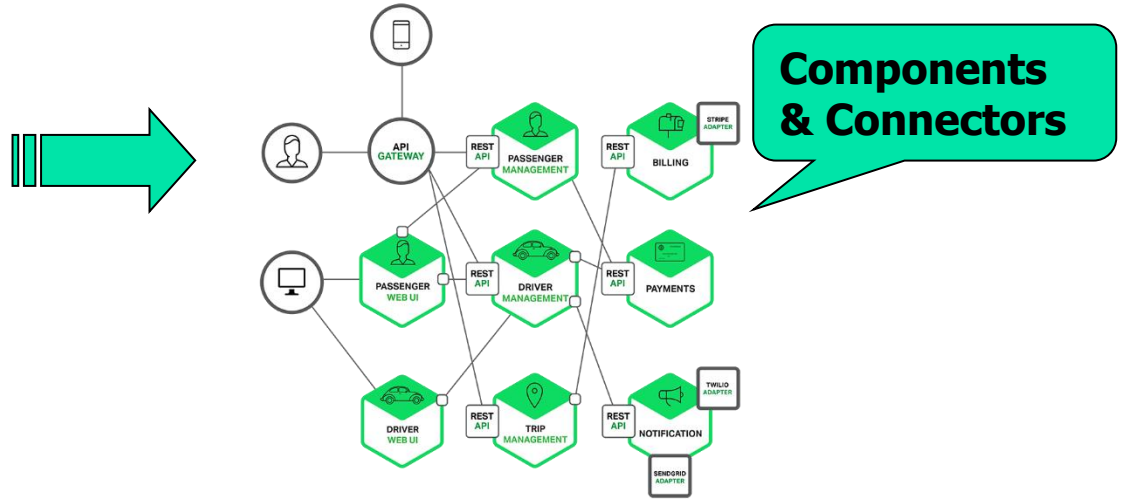
Architecture Design Principles

Software Architecture Design

Programming in the Small



Programming in the large



Software architecture shows the system's structure and hides the implementation details, **focusing on how the system components interact with one another.**

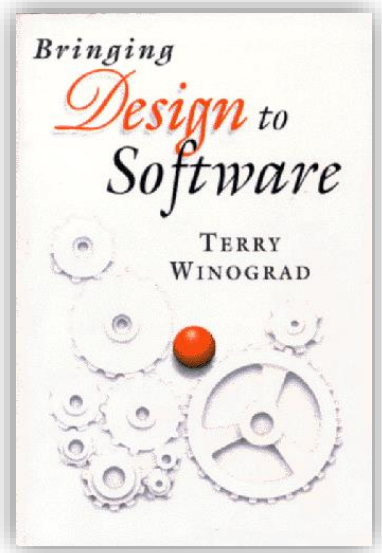
Software architecture pattern: how subsystems are connected together to meet the application's functional and non-functional requirements

Goals of architecture design

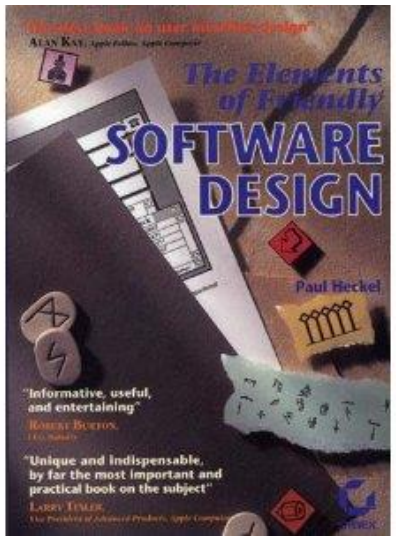
- High Performance
- High Availability
- High Scalability
  - Capacity
  - Change
- High Heterogeneity
- High Security
- Low Cost

Design Principles

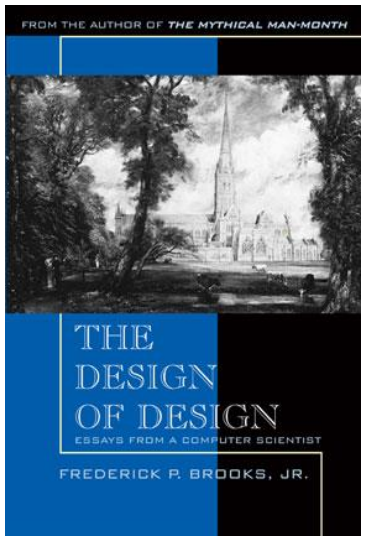
1. KISS = Keep It Simple Stupid
2. Modularity
3. Separation of Concern
4. Design for CHANGE
5. Design for REUSE



[1996]

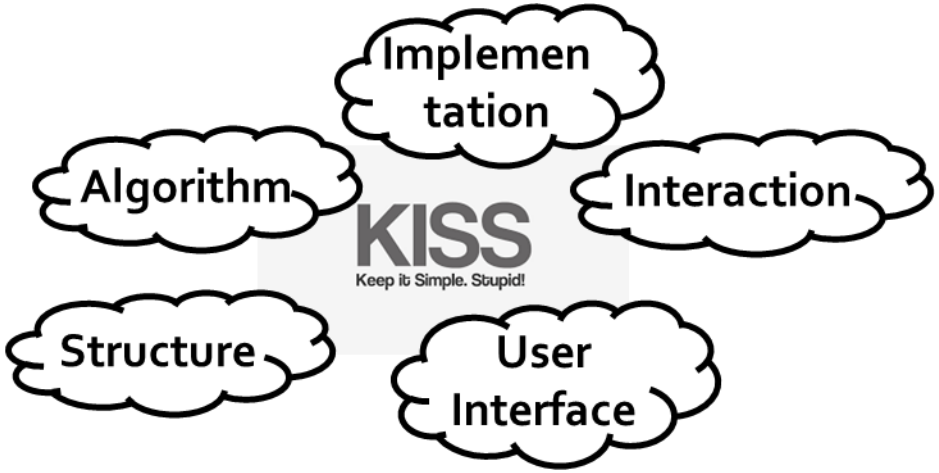


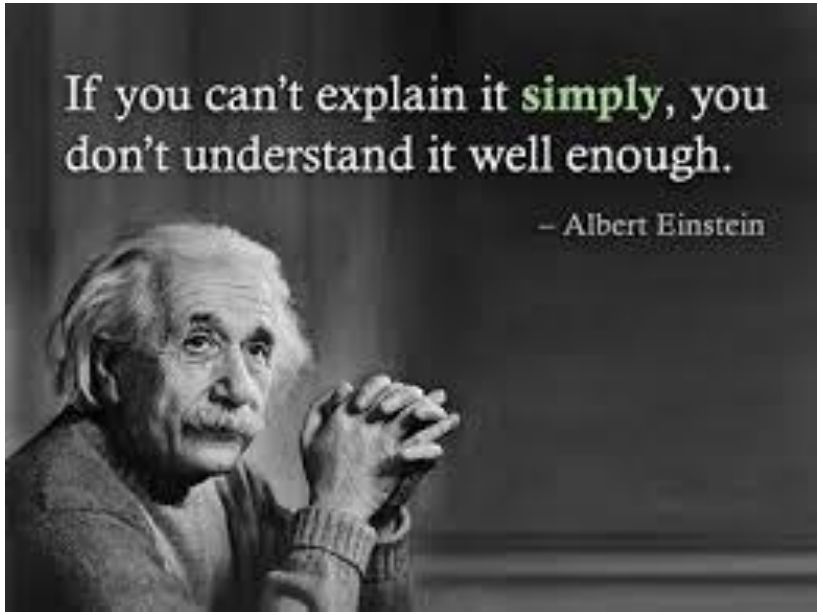
[1994]



[2010]

KISS – Keep It Simple Stupid





Simple is Beauty

"There are two ways of constructing a software design: One way is to make it **so simple** that there are obviously no deficiencies, and the other way is to make it **so complicated** that there are no obvious deficiencies. **The first method is far more difficult**. "

— C. A. R. Hoare

The 1980 ACM Turing Award Lecture

Delivered at ACM '80, Nashville, Tennessee, October 27, 1980

C.A.R. Hoare

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960 to 1968 he held a number of positions with Elliot Brothers, Ltd., England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science *Honoris Causa* by the University of Southern California in 1979.

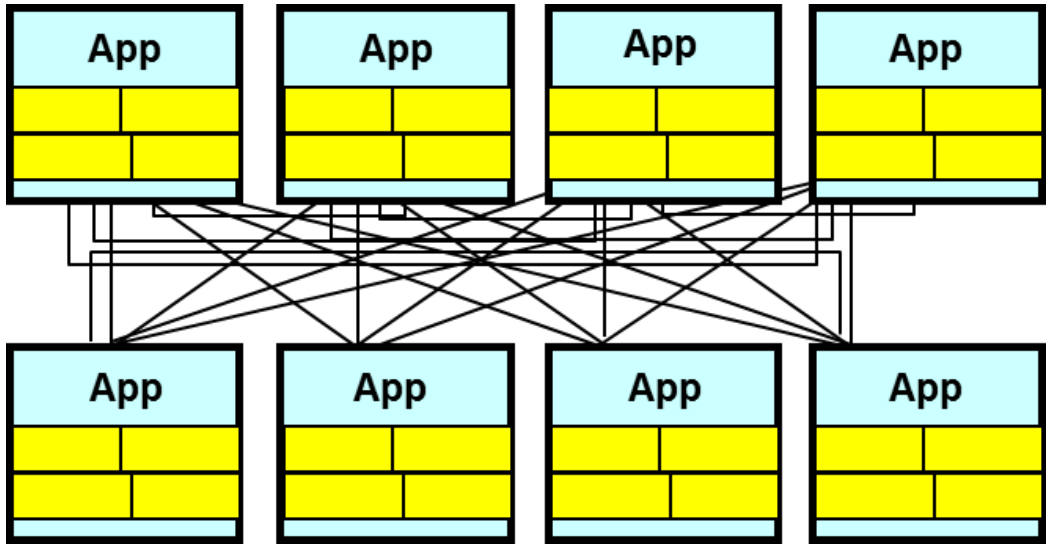
The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A. M. Turing, an English mathematician who made many important contributions to the computing sciences.

The Emperor's Old Clothes

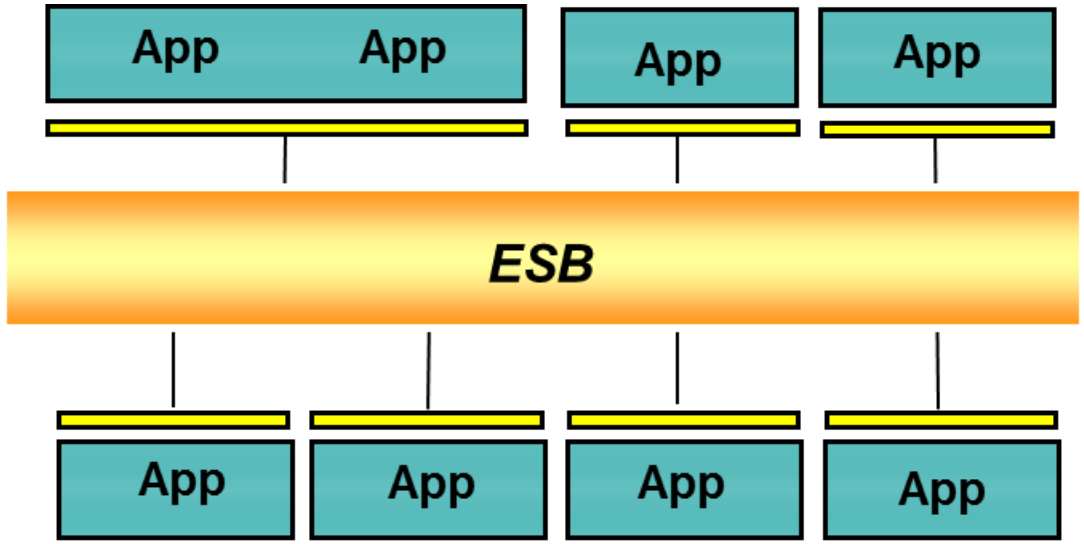
Charles Antony Richard Hoare
Oxford University, England

Simple Structure

- Simplify interface
- Decouple module dependencies

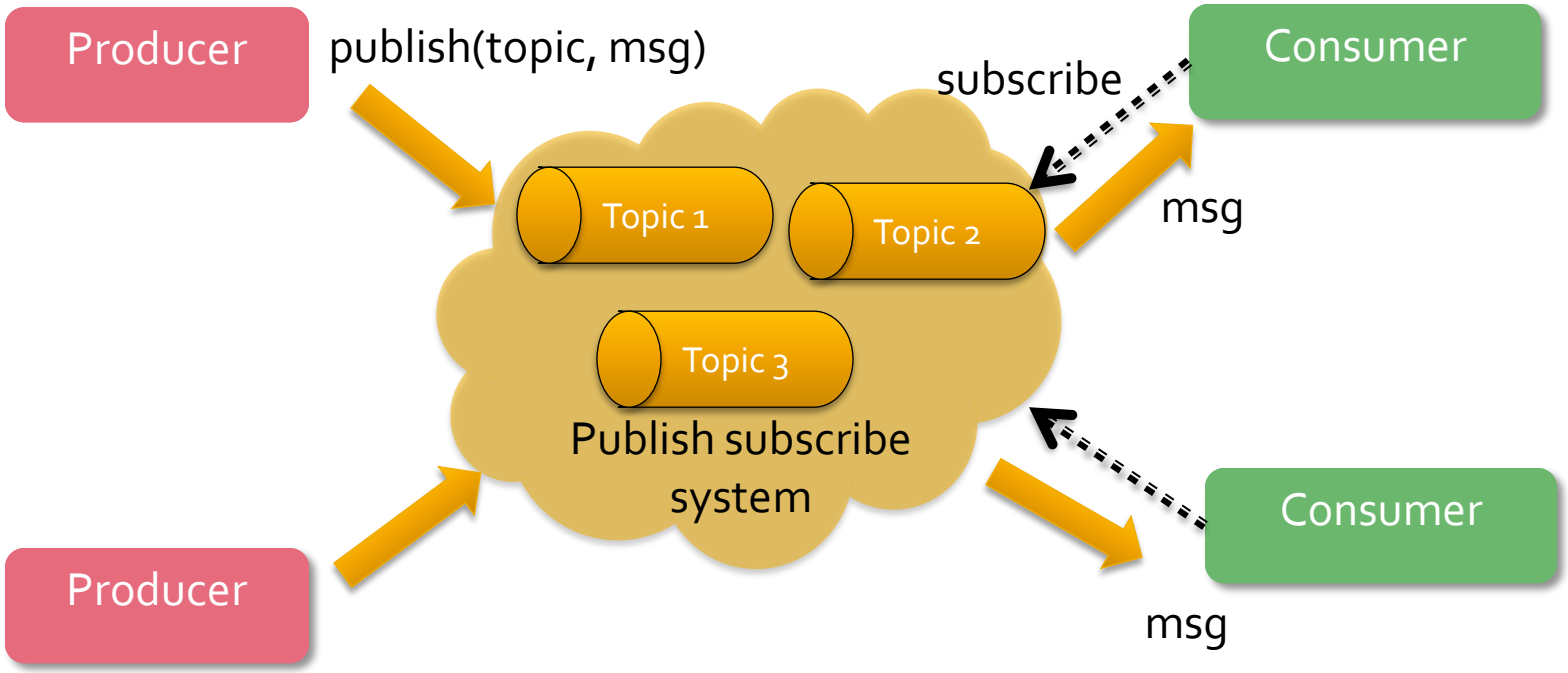


System Integration – Direct Interfacing

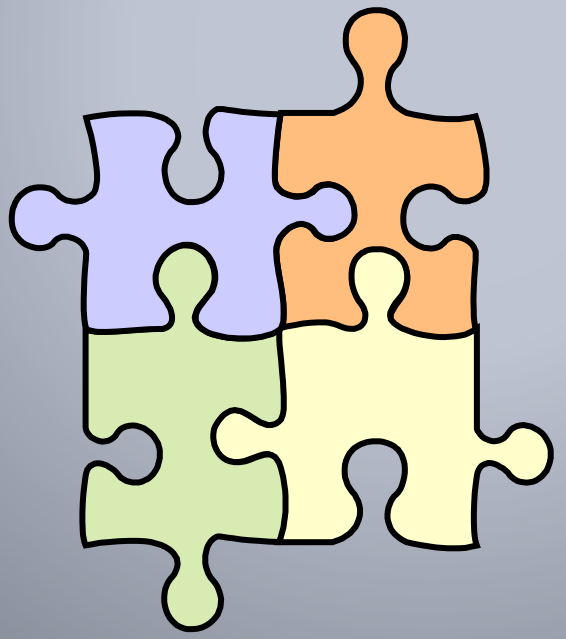


System Integration – Middleware Based

Simple Structure



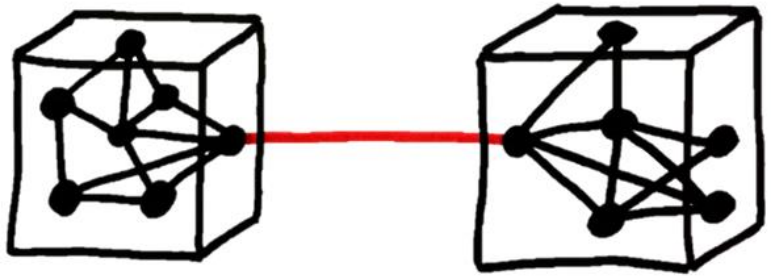
Modularity



Modularization

- Dividing complex system into simpler components

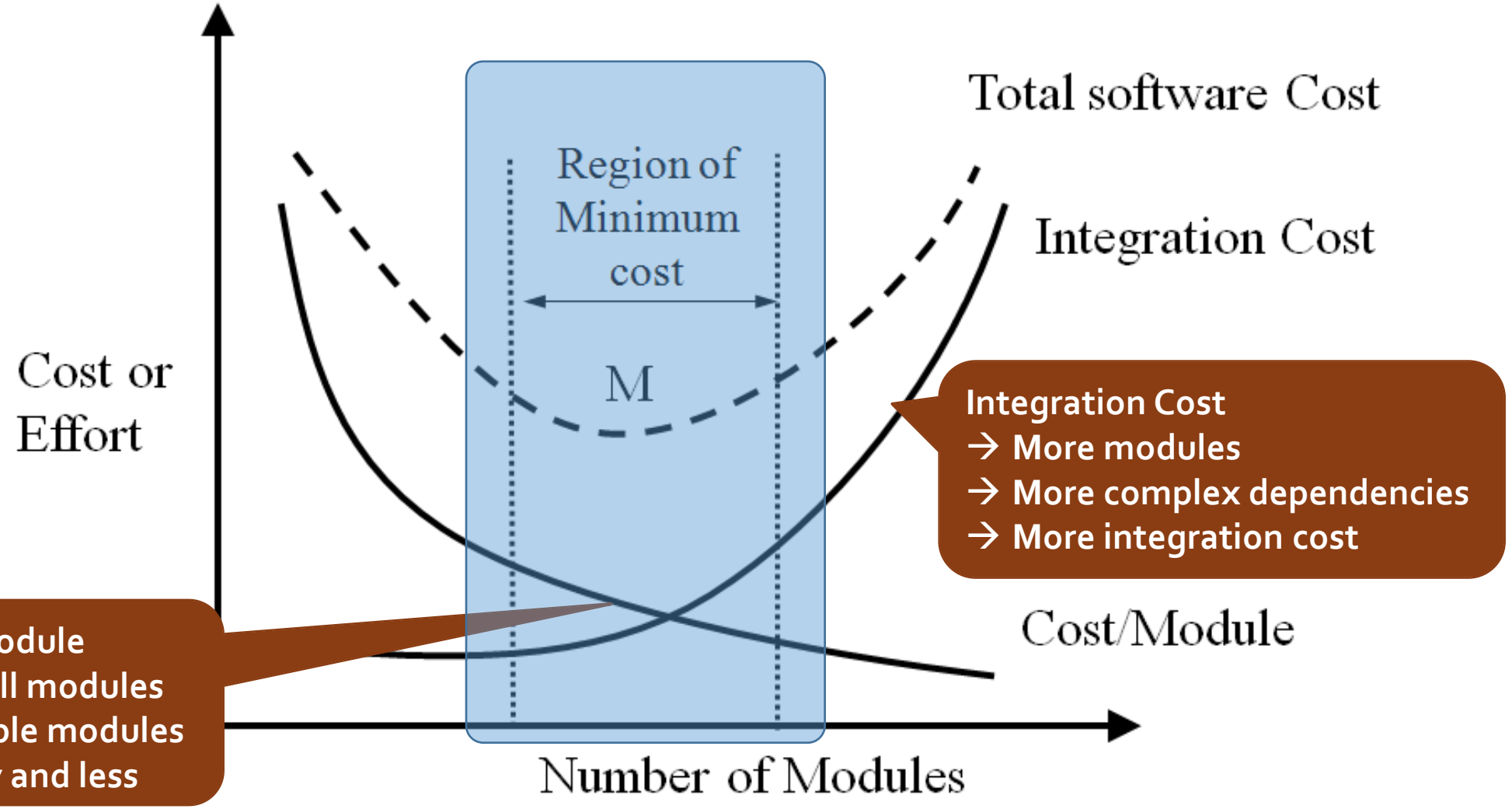
Strong Cohesion
Weak Coupling



[Stevens, Myers, Constantine 1974]

- Expected benefits
  - Development can be more easily planned
  - Software increments can be defined and delivered
  - Changes can be more easily accommodated
  - Testing and debugging can be conducted more efficiently
  - Long-term maintenance can be conducted without serious side effects

Modularization



Separation of Concerns

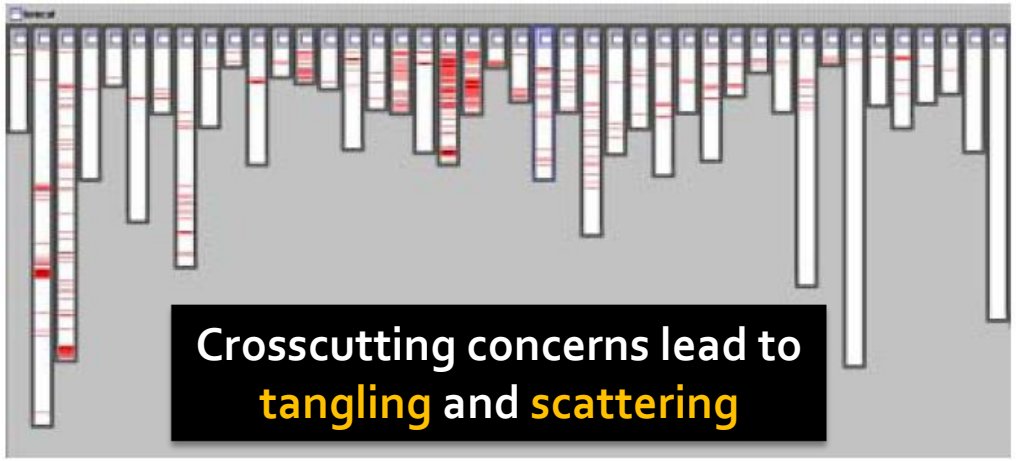
Divide and Conquer

- Many decisions are strongly related and interdependent
- Concentrate on different aspects of a problem separately, e.g. software qualities, development process, operating constraints
- Separation may be based on time, qualities, views, size, level of details etc.

Crosscutting Concerns

- Concerns whose implementation cuts across a number of program components.

- This results in problems when changes to the concern have to be made – the code to be changed is not localized but is in different places across the system.



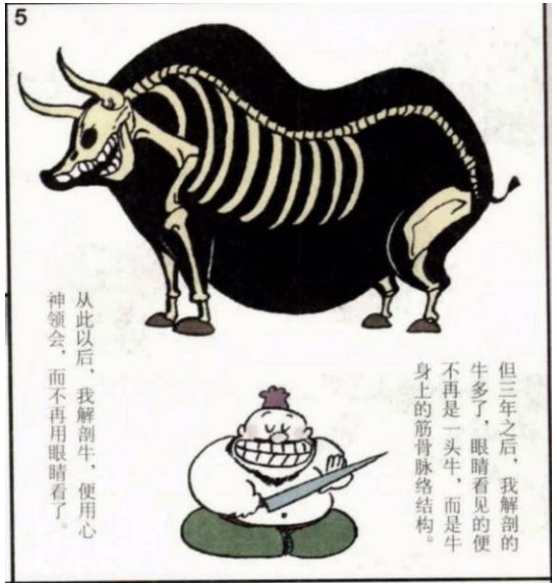
logging in org.apache.tomcat

- Red shows lines of code that handle logging
- Not in just one place
- Not even in a small number of places

Software should be organized so that each program element does one thing and one thing only.

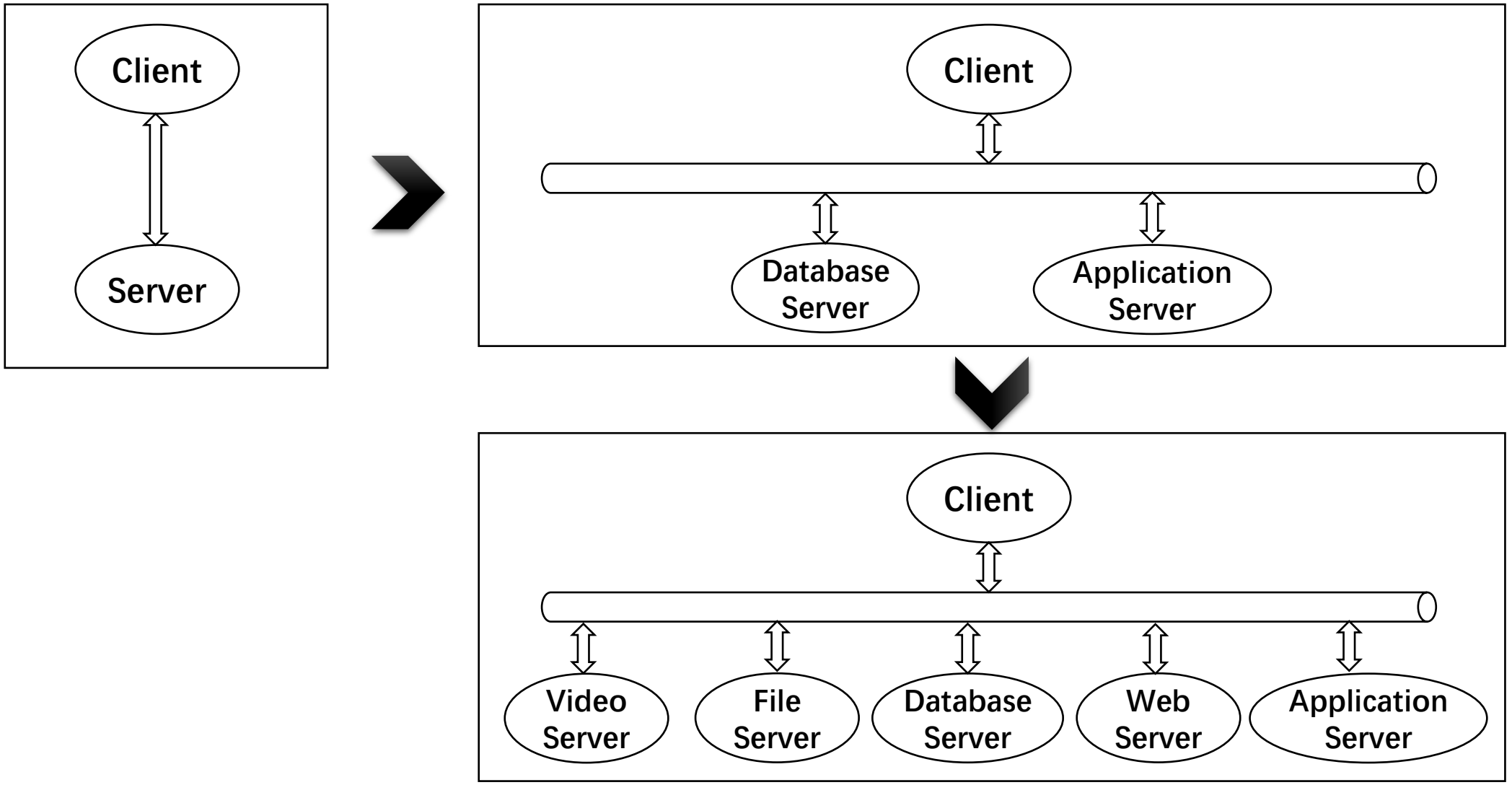
Separation of Concerns

- Systematic thinking to solve complex problems
  - Decomposition
    - to decompose complex problem to sub-problems
    - Small, simple, controllable
  - Synthesis
    - To compose solutions of sub-problems into an integrated solution



目无全牛
官止神行
批郤[xì]导窾[kuǎn]
切中肯綮[qìng]
游刃有余
踌躇满志

Example – Architecture Design



Design for Change

The First Law of Software Engineering ……

- Change is inevitable and frequent due to
  - bugs-elimination
  - changing user requirements
  - technology evolution
- Change in the software product as well as the development process
- Strategy is to consider the likely changes as part of the system design
  - Isolate likely changes in specific portions of software such that changes will be restricted to small portions

"When we were first taught how to program, we were given a specific program and told to write one program to do that job…. Today, the software designer should be aware that he is ***not designing a single program but a family of programs***…"

-- Designing software for ease of extension and contraction [David Parnas, ICSE'78]

Modifiability

- Modifiability "relates to the cost of change and refers to the ease with which a software system can accommodate changes. " [Northrop 2004]

What can change?

What is the likelihood of the change?

When is the change made and who makes it?

What is the cost of the change?

Reuse

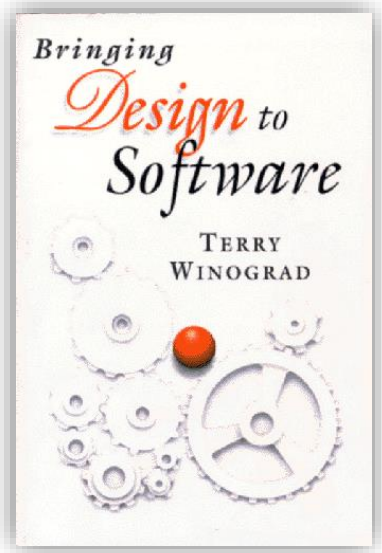
Design with and for reuse

- To avoid "reinventing the wheel"
  - DRY (Don't Repeat Yourself) and avoid WET (Write Every Time)
- Software is expensive. Reuse of software assets is essential to increase ROI (return on investment)
  - Lower production and maintenance cost
  - Faster delivery
  - Increased quality
- Software reusable assets
  - Req., design, code, test cases
  - Problem solution, design solution
  - Tools
  - Process
  - Experiences and best practices
- Adopt mature open source component

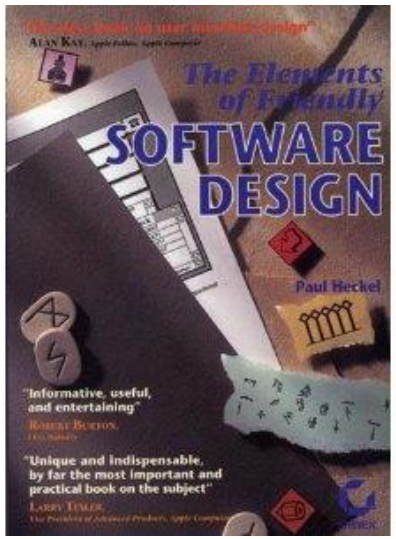
- **Handcraft → Assembly line**
- **1913-14, Ford Motor Company, mass production**
  - **Interchangeability**
  - **Reinventing workflow and job descriptions**
- **Benefits:**
  - **Reduced unit-cost – Affordable Ford Model T**
  - **Consistent quality**
  - **Improved productivity**

Architectural Design Principles

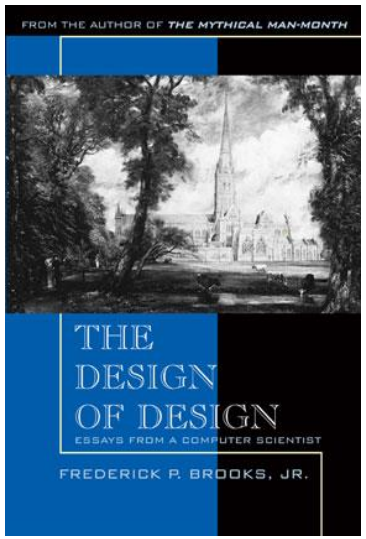
1. KISS = Keep It Simple Stupid
2. Modularity
3. Separation of Concerns
4. Design for CHANGE
5. Design for REUSE



[1996]



[1994]



[2010]

Outline

- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

Architecture Styles

Common Architecture Styles

- Indep. components
  - Event-based
- Data Flow
  - Pipes and Filters
- Call / Return
  - Layered

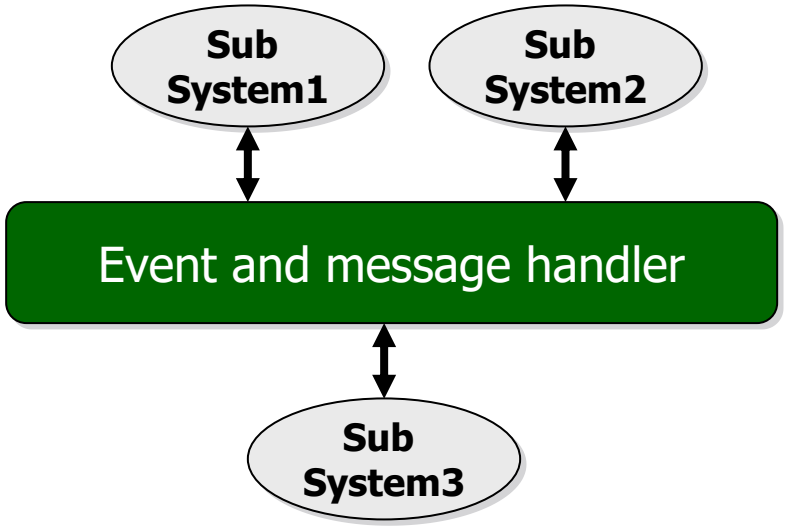
Event-Based: Definition

"This architectural style is characterized by the style of communication between components: Rather than invoking a procedure directly or sending a message, a component announces, or broadcasts, one or more events."

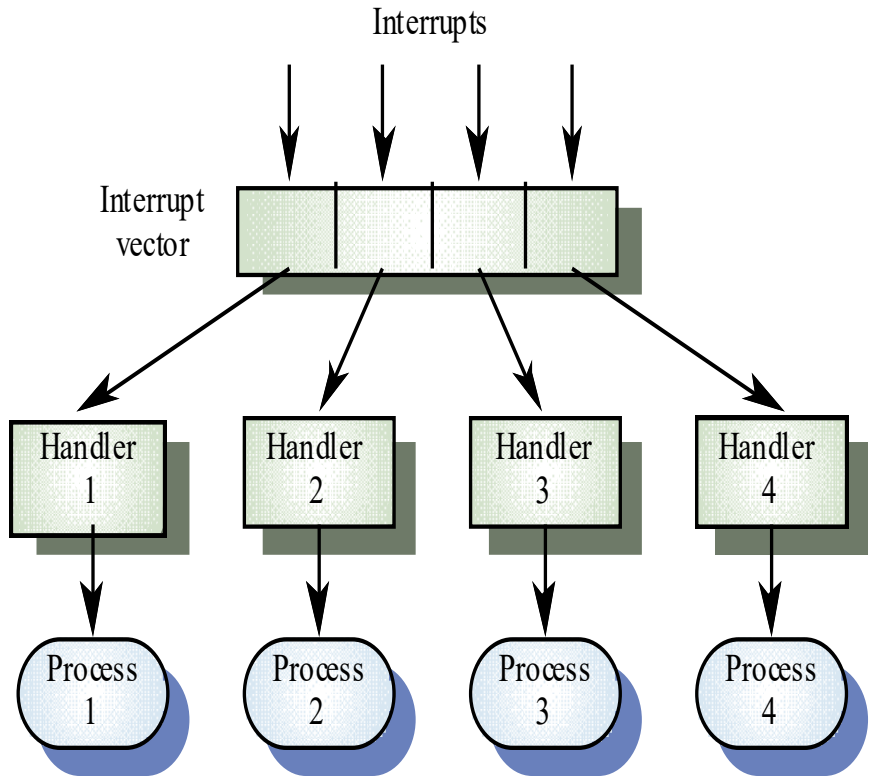
- Explicit invocation
- Implicit invocation

  - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
  - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.

Implicit Invocation

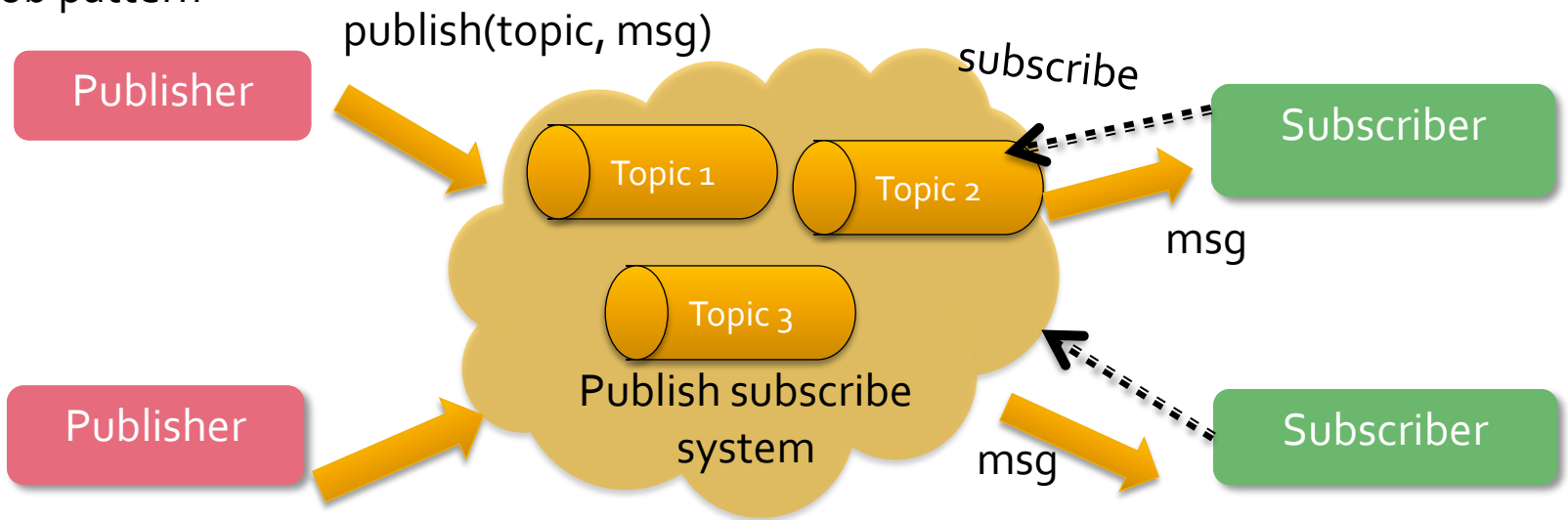


Selective Broadcasting



Interrupt Driven

Example: pub/sub pattern



Implicit Invocation: Elements

- Components: object or processes
  - Interfaces define event announcement
  - Interfaces define event processing
- Connectors: binding between event announcer and processor
  - Components register interest in an event by associating a procedure with the event
  - Components interact by "announcing" events
  - Upon receiving an event, the associated procedures are implicitly invoked
  - Order of invocation is non-deterministic

Implicit Invocation: Advantages

- Problem decomposition
  - Computation and coordination are separated
- System maintenance
  - Add and replace components with minimum affect on others
- System reuse
  - Plug in new components by registering it for events
- Performance
  - Invocations can be parallelized

Implicit Invocation: Disadvantages

- Problem decomposition
  - Loss of control
    - The responding components
    - The order of invocation
    - This is why it is also called "Independent components" style
  - Unpredictable interactions, hard to ensure the correctness
  - Hard to exchange data, shared repository may be required
- System maintenance and reuse
  - Requires a centralized "yellow pages" of who knows what: events, registrations, dispatch policies
- Performance
  - Data sharing imply some performance penalty

Pipes and Filters: Definition

"The pipes and filters architectural pattern (style) provides a structure for systems that *process a stream of data*. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems."

Pattern-Oriented Software Architecture, Frank Buschmann, et al, 1996

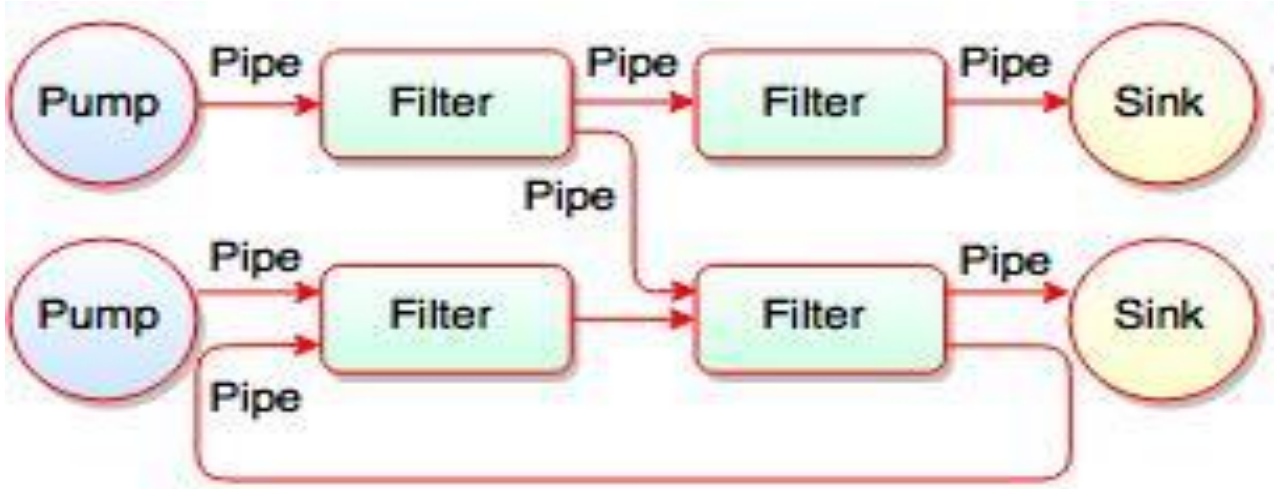


Fig. Pipes and Filters

Pipes and Filters: Elements

- Components: Filters
  - Read streams of data on input
  - Produce streams of data on output
  - Local incremental transformation to input stream
- Connectors: Pipes
  - Conduit for streams, e.g. first-in-first-out buffer
  - Transmit outputs from one filter to input of another

Example: Traditional Compiler

Linux Shell : cat "app.log" | grep "^error" | cut -f 2

Pipes and Filters: Advantages

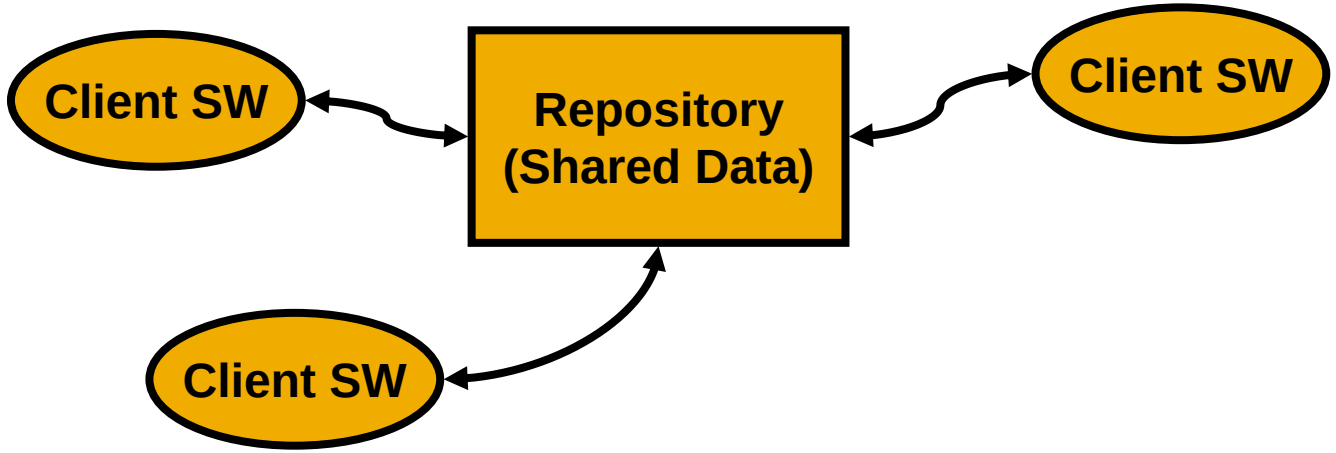
- Problem decomposition
  - Filter composition
  - Hierarchical decomposition
- System maintenance
  - Filters can be added or replaced, ease of extension and modification
- System reuse
  - Reorganization of filters : agreement on the interface data format
  - Black box approach
- Performance
  - Enable parallel processing / concurrent execution of filters

Pipes and Filters: Disadvantages

- Problem decomposition
  - Degenerated to 'batch processing'
  - Hard to design incremental filters
  - Sharing global data is expensive or limiting
  - Interactive applications are difficult
- Performance
  - Parsing and unparsing overhead
  - Possible deadlock with finite buffers

Repository: Definition

- A central data store holds the large amount of data to be shared among sub-systems



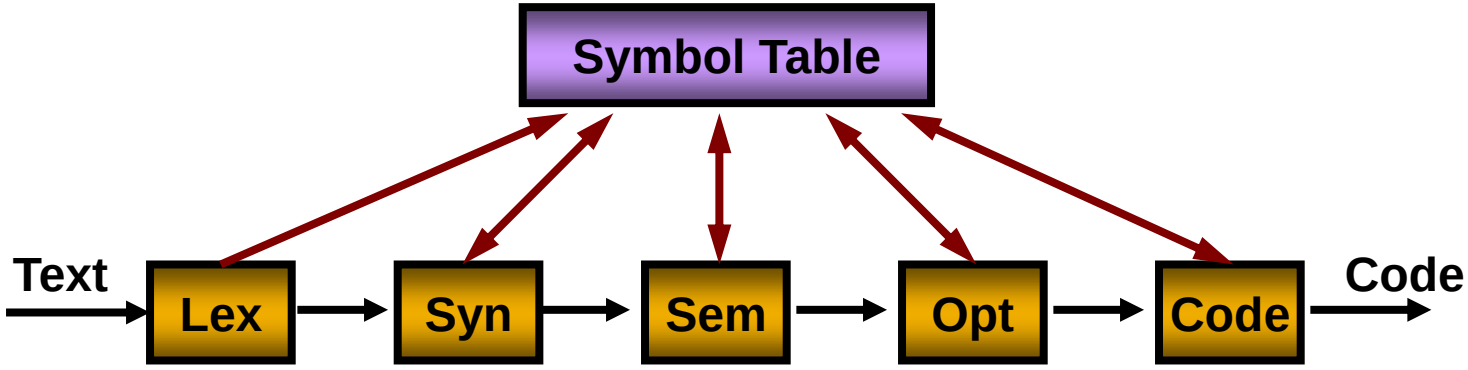
Repository: Elements

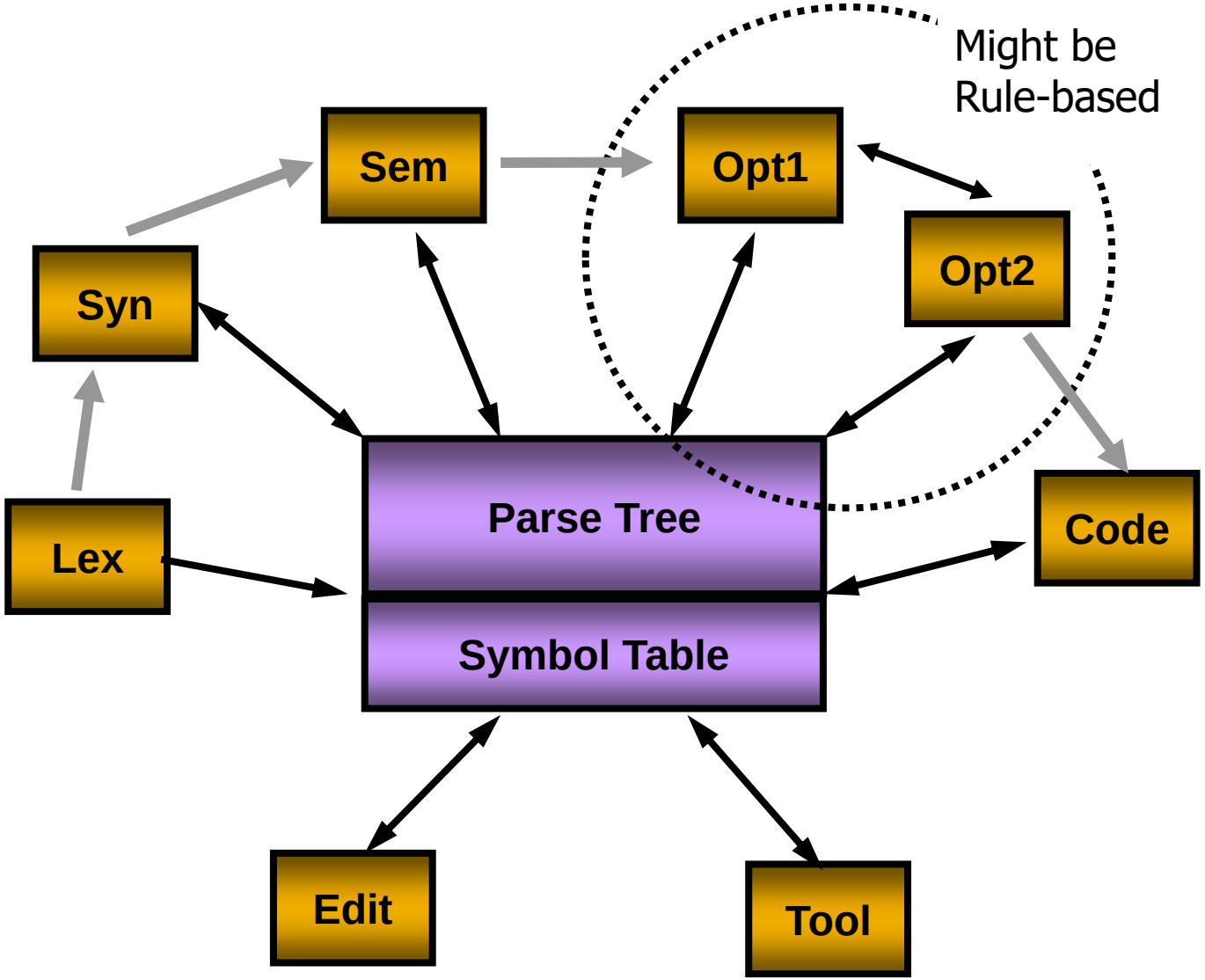
- Components
  - Central data store represents system state
  - A collection of independent components operate on the central data store
- Connectors
  - Transactional Database model (Passive)
    - Incoming stream of transactions trigger the processes to act on the data
  - Blackboard model (Active)
    - The state of the data store triggers the selecting processes to execute

Repository: Example – Compiler

Review: Pipe and Filter model

Updated: Shared Symbol Table





Canonical Compiler, Revisited

Repository: Advantage

- Efficient way to share large amounts of data. No need to transmit data explicitly from one sub-system to another.
- Sub-systems need not be concerned with how data is produced and used by other sub-systems
- Centralised management e.g. backup, security, access control and recovery, etc.
- Sharing model is published as the repository schema. Straight-forward integration of new tools given that they are compatible with the agreed data model.

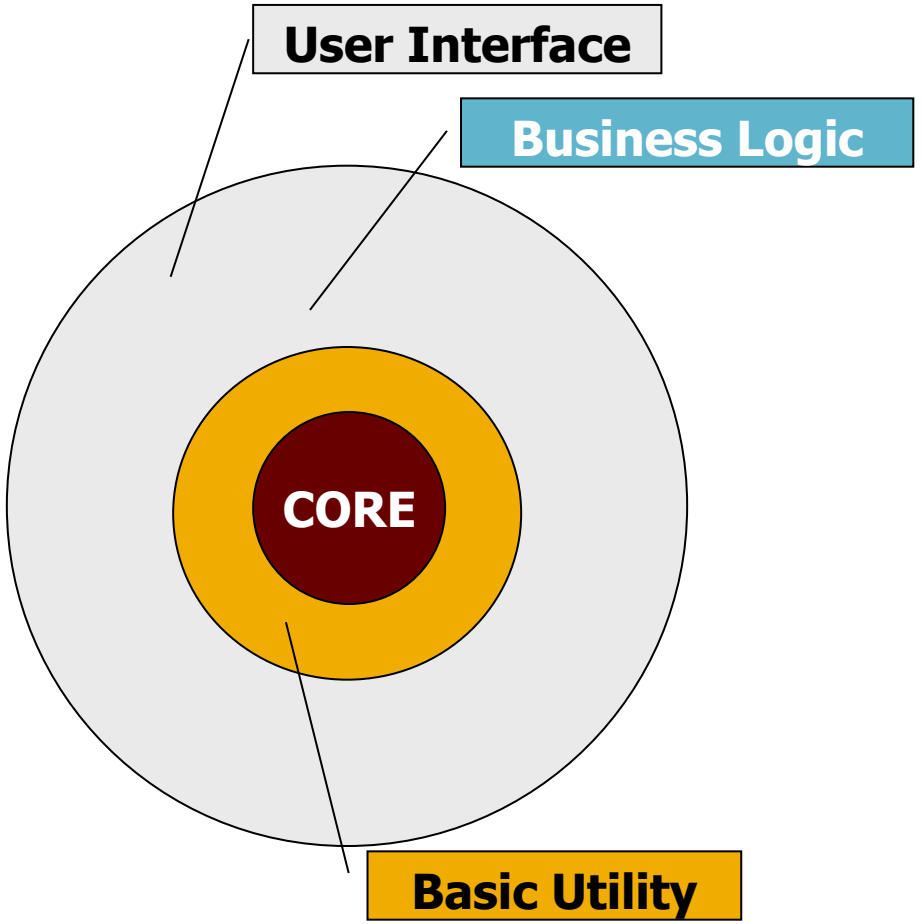
Repository: Disadvantage

- Sub-systems must agree on a repository data model
  - Performance may be adversely affected by this comprise
  - Difficult to integrate new sub-systems if their data models do not fit the agreed schema
- Data evolution is difficult and expensive
- Difficult to distribute efficiently, data redundancy and inconsistency problems
- Forces the same policy on all sub-systems

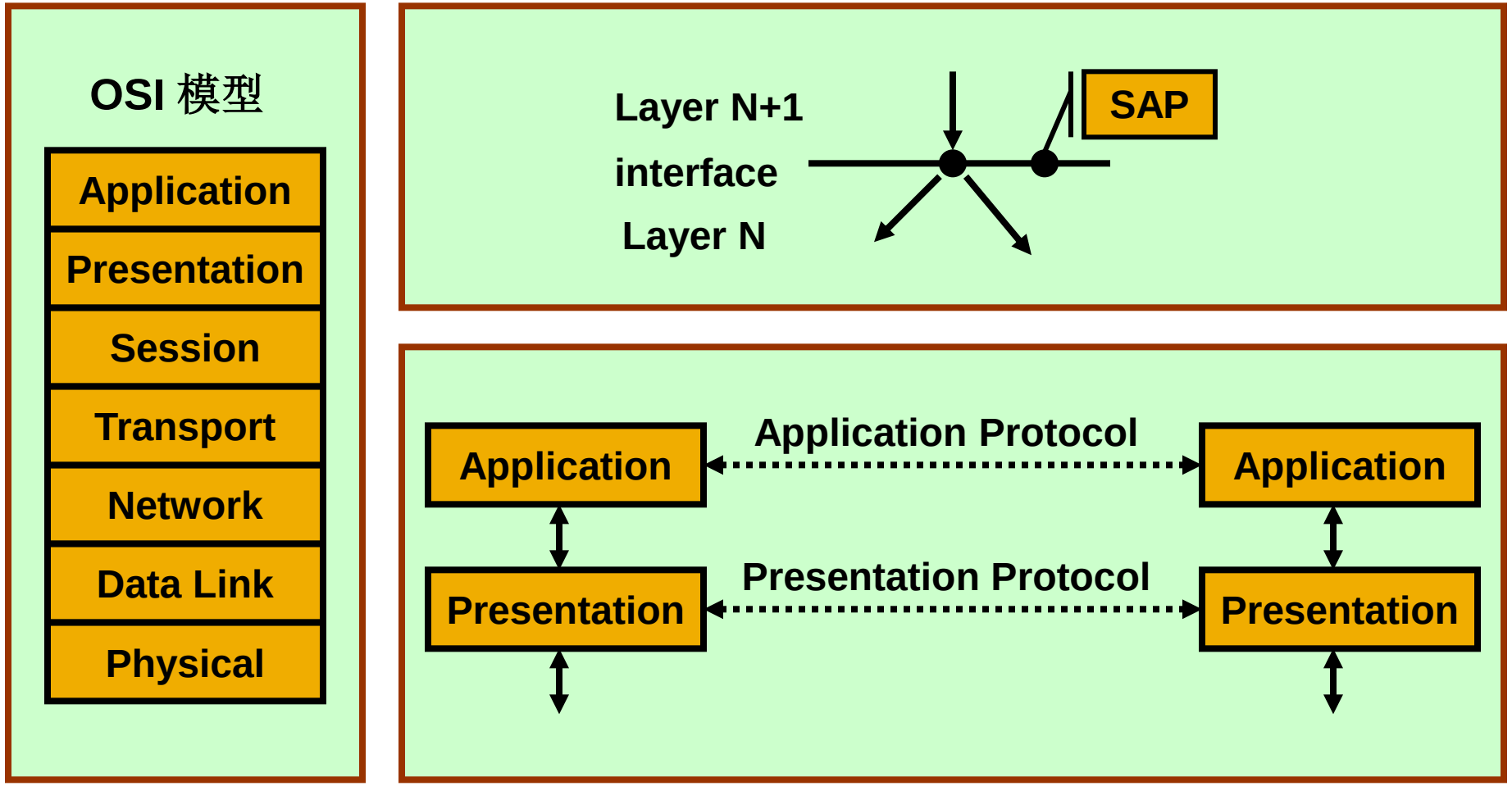
Layered: Definition

"Layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below.

Layers may be hidden to all except adjacent layers or layers may be partially hidden."



Example:


OSI 模型
Application
Presentation
Session
Transport
Network
Data Link
Physical
Layer N+1
interface
Layer N
SAP
Application
Application Protocol
Application
Presentation
Presentation Protocol
Presentation

Layered: Elements

- Components: Layers
  - Group of subtasks which implement a “virtual machine” at certain layer in the hierarchy
- Connectors: Protocol / Interface
  - Each layer hides lower layer and provides services to higher layer
  - Each layer serves distinct functions

Layered: Advantage

- Problem decomposition
  - Increasing levels of abstraction, partition complex problems
- System maintenance
  - A layer only interacts with the layer above and the one below. Limited change effect
- System reuse
  - Each layer supports a set of standard interfaces
  - Different implementations of the same interfaces can be interchanged

Layered: Disadvantage

- Layered abstraction is difficult
  - Not all systems are easily structured in a layered fashion
- Communication down through layers and back up may cause performance penalty
- Opaque layers prevent user control of deep functionality
- Layer bridging

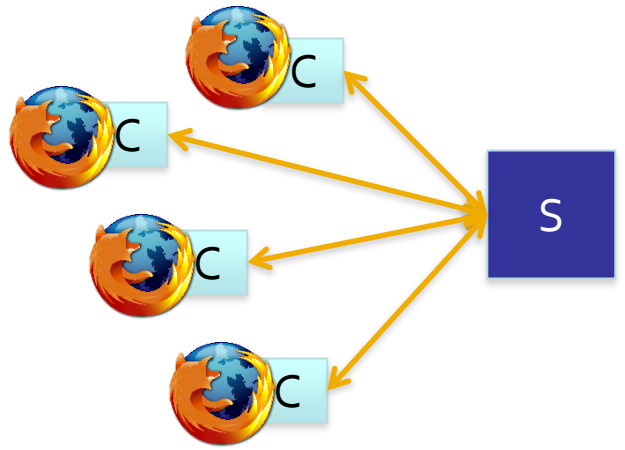
Outline

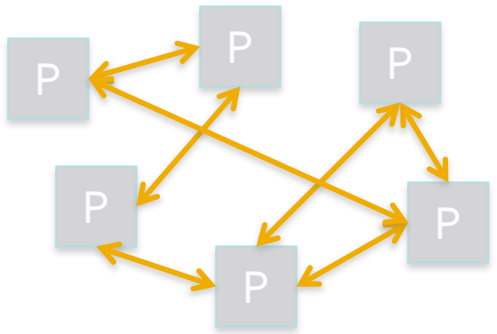
- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- **Distributed Architecture Styles**
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

Distributed Architecture Styles

Distributed patterns to know:

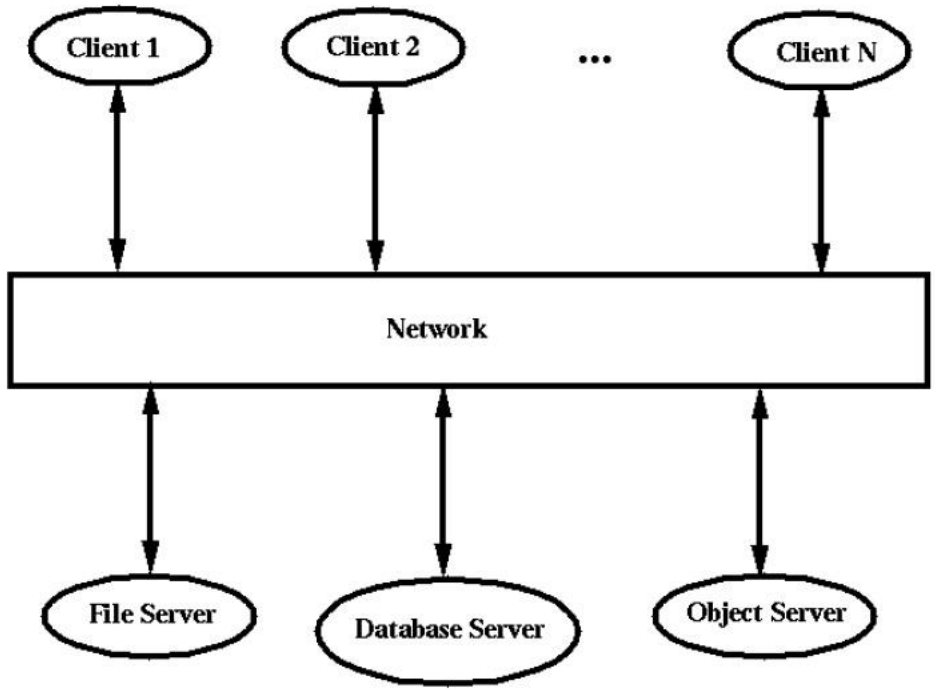
- Architectural patterns in internet based services:
  - Client-Server
  - Peer-to-Peer
- In client-server architectural pattern, a client makes requests and the server responds to many client requests. Client-Server captures the reusable behavior of specialized servers and clients without specifying how servers and clients should be implemented.
- Peer-to-Peer design would cause all entities to act as clients and servers. While flexible, it makes it difficult to implement software to do either job well.





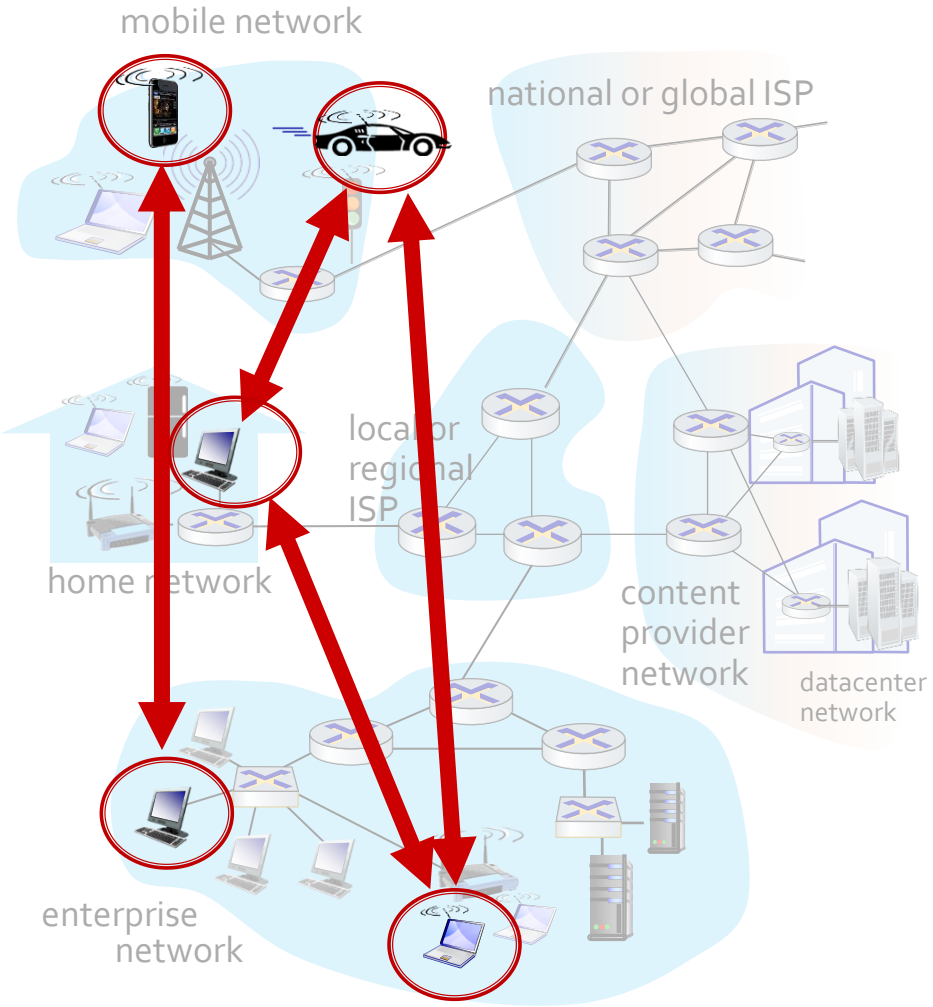
Client-Server Style

- Suitable for applications that involve distributed data and processing across a range of components.
- Components:
  - Servers: Stand-alone components that provide specific services such as printing, data management, etc.
  - **Clients:** Components that call on the services provided by servers.
- Connectors:
  - The network, which allows clients to access remote servers
- Examples: File servers, database servers



Peer-to-peer (P2P) Style

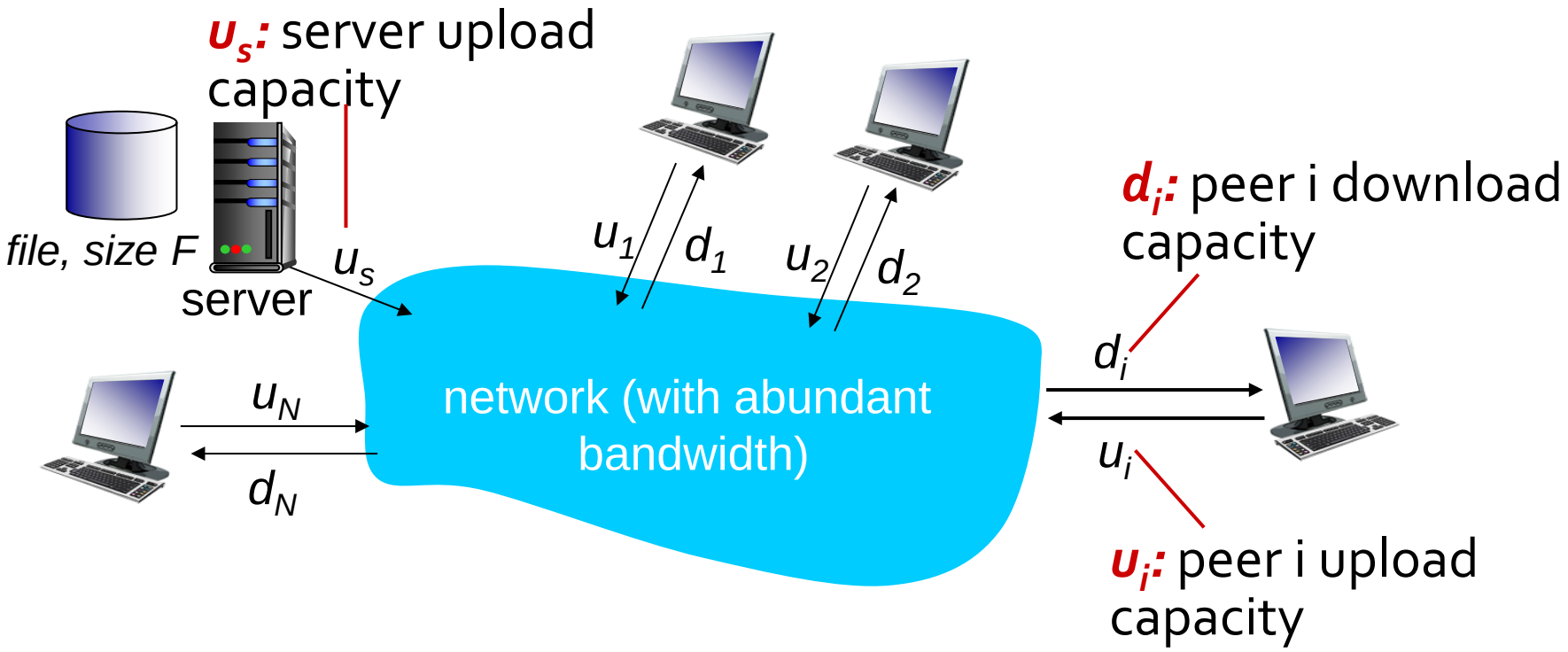
- *No* always-on server
- Arbitrary end systems directly communicate
- Peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, and new service demands
- Peers are intermittently connected and change IP addresses
  - complex management
- Examples: P2P file sharing (BitTorrent)), VoIP (Skype), streaming (KanKan



File distribution: client-server vs P2P

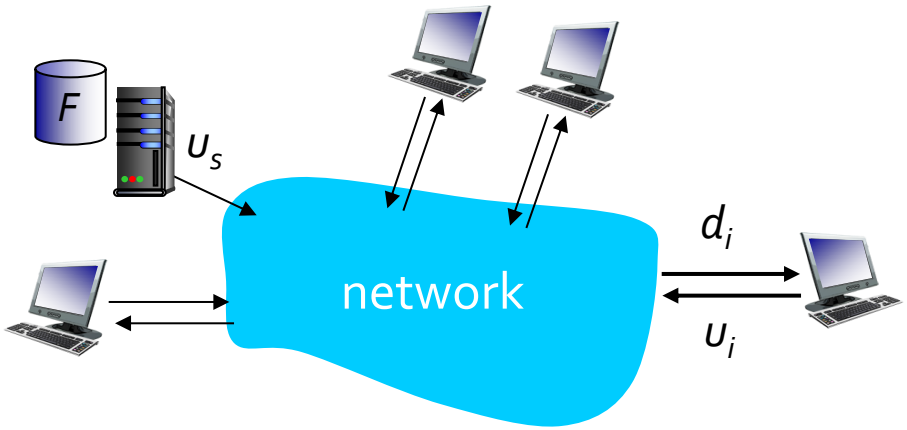
Q: how much time to distribute file (size *F*) from one server to *N peers*?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies:
  - time to send one copy: F/u_s
  - time to send *N* copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} =slowest client download rate
  - maximum client download time: F/d_{min}



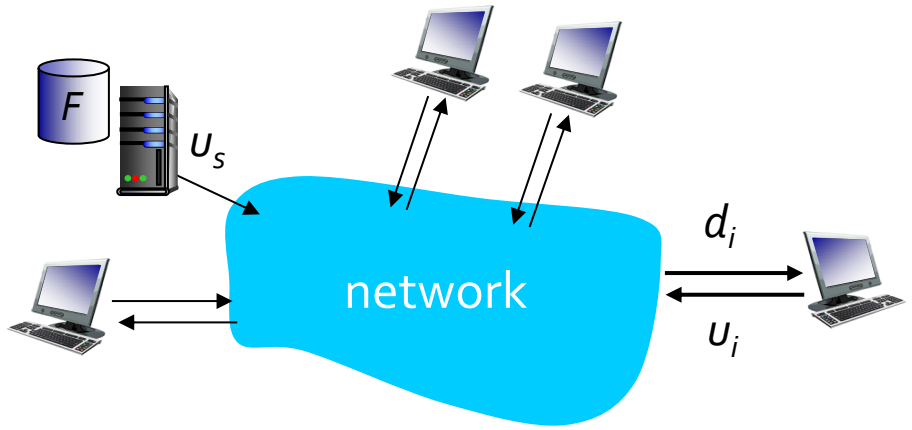
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy:
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - maximum client download time: F/d_{min}
- *clients:* as aggregate must download NF bits
  - max upload rate (limiting max download rate) is $u_s + \Sigma u_i$

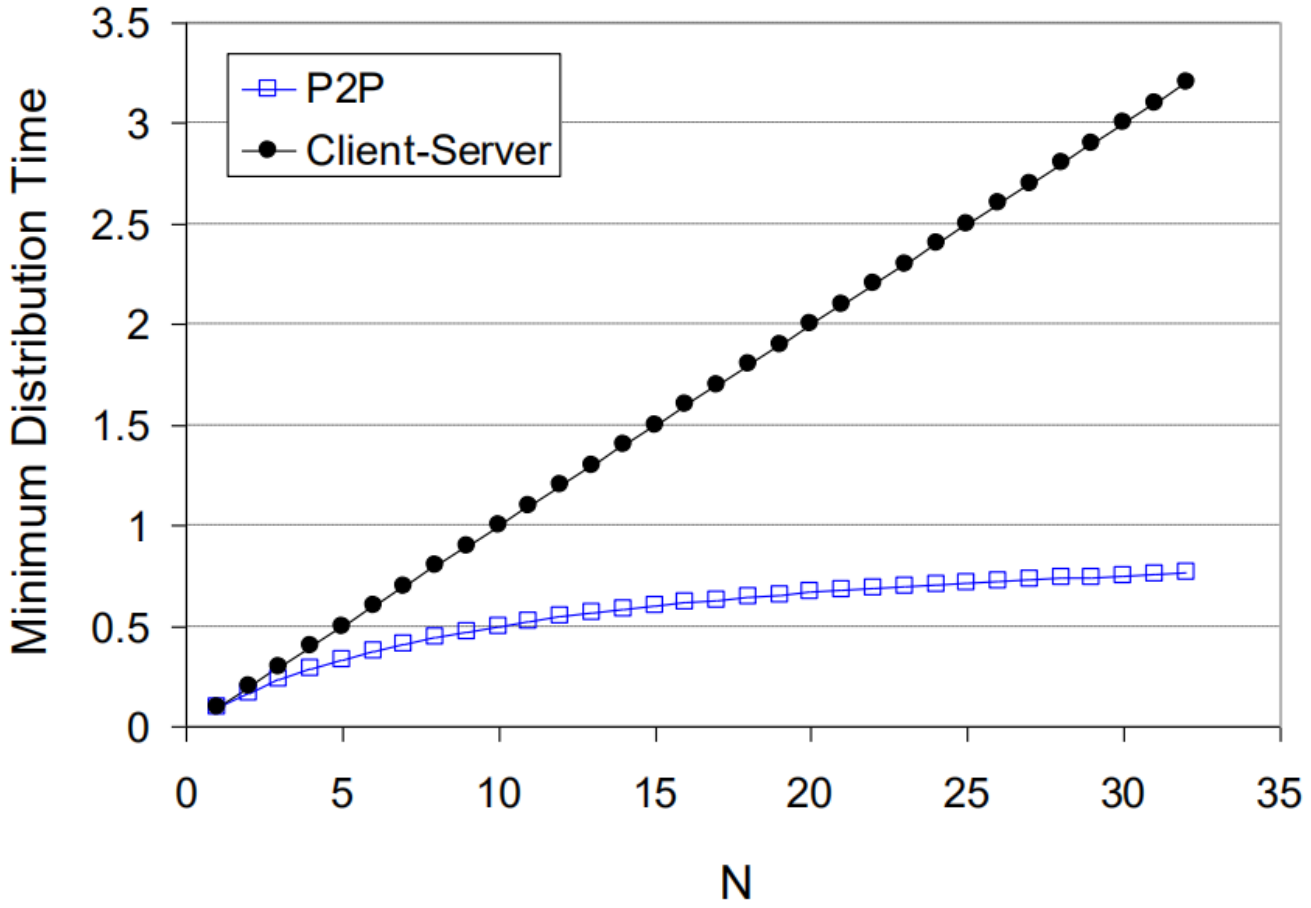


time to distribute F to N clients using P2P approach

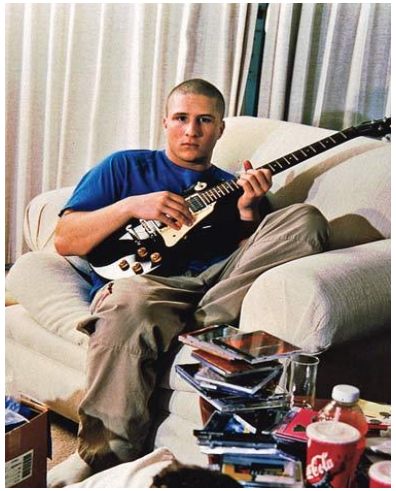
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in N …
… but so does this, as each peer brings service capacity

File distribution time: Cient-Server vs. P2P

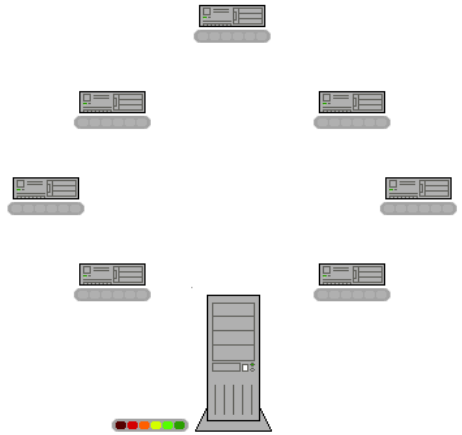


Napster
a pioneering P2P
file sharing system



- Jan. 1999, Shawn Fanning, 18, creates the Napster application and services while a freshman at Northeastern University. It allows music enthusiasts to download copies of songs that were otherwise difficult to obtain.
- May 1999, Napster Inc. is founded
- 2000, RIAA sued Napster on grounds of contributory and vicarious copyright infringement under the US DMCA.
- July 2001, Napster closed down all its services.

BitTorrent
P2P content sharing



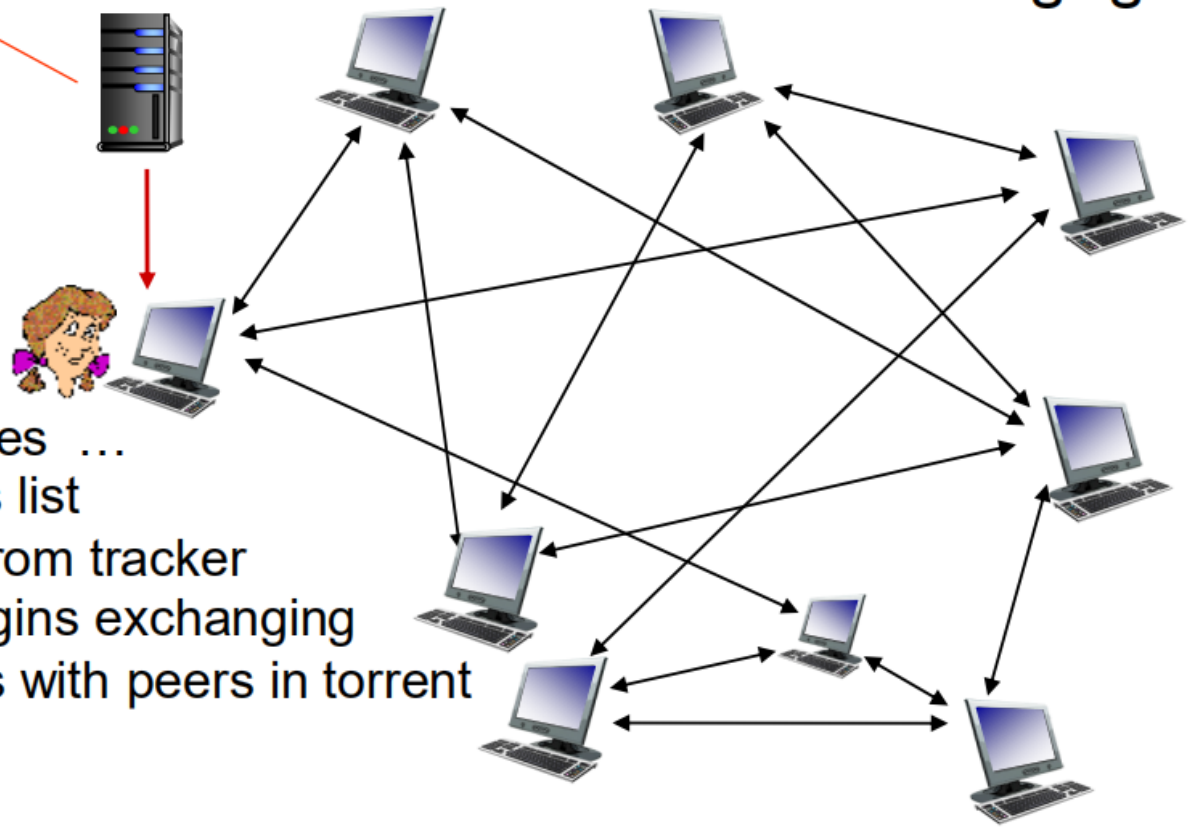
BitTorrent is one of the most common protocols for transferring large files, such as digital video files containing TV shows or video clips or digital audio files containing songs.

- Apr. 2001, Bram Cohen, a former University at Buffalo student, designed the protocol.
- 2013, BitTorrent has 15-17 million concurrent users at any time.
- The protocol itself is perfectly legal, problems stem from using the protocol to traffic copyright infringing networks. More than 200,000 people in US have been sued for filesharing on BitTorrent since 2010.

BitTorrent: P2P content sharing

tracker: tracks peers participating in torrent

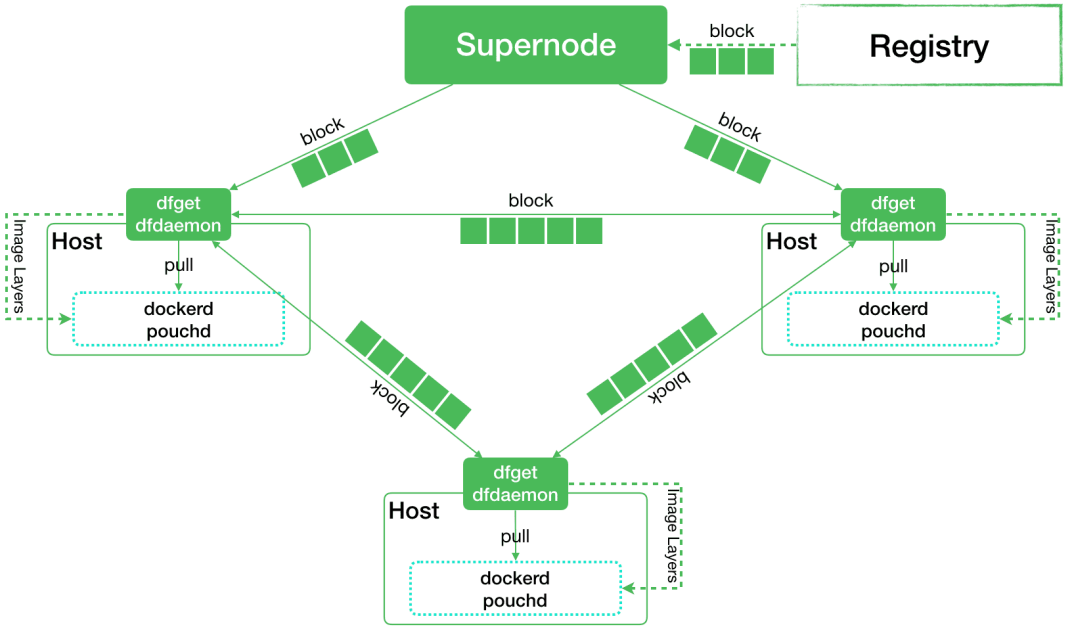
torrent: group of peers exchanging chunks of a file

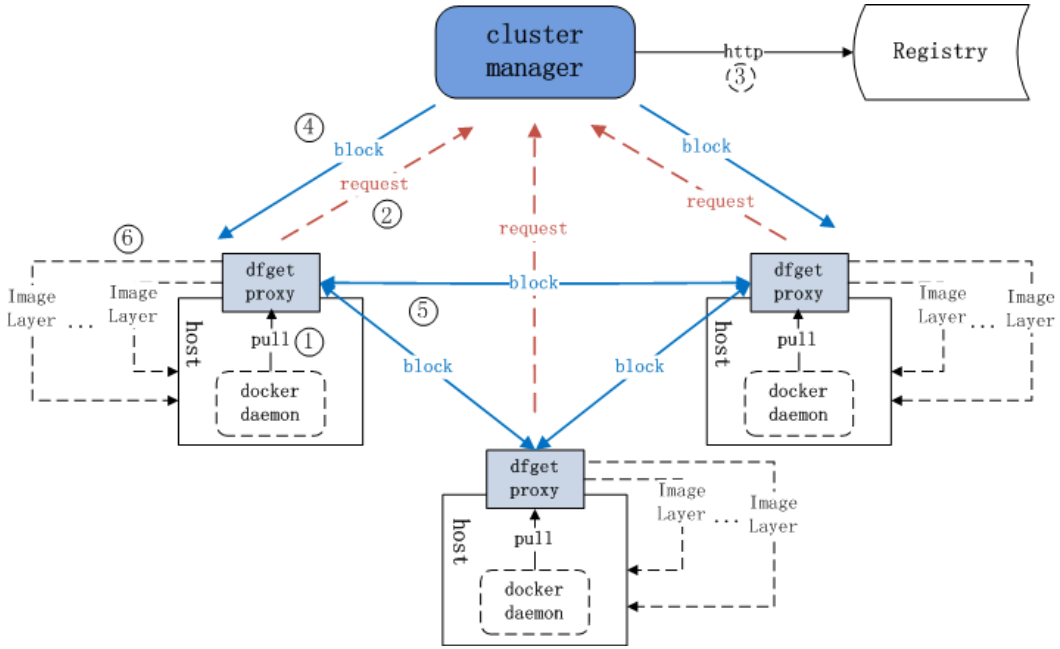


Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

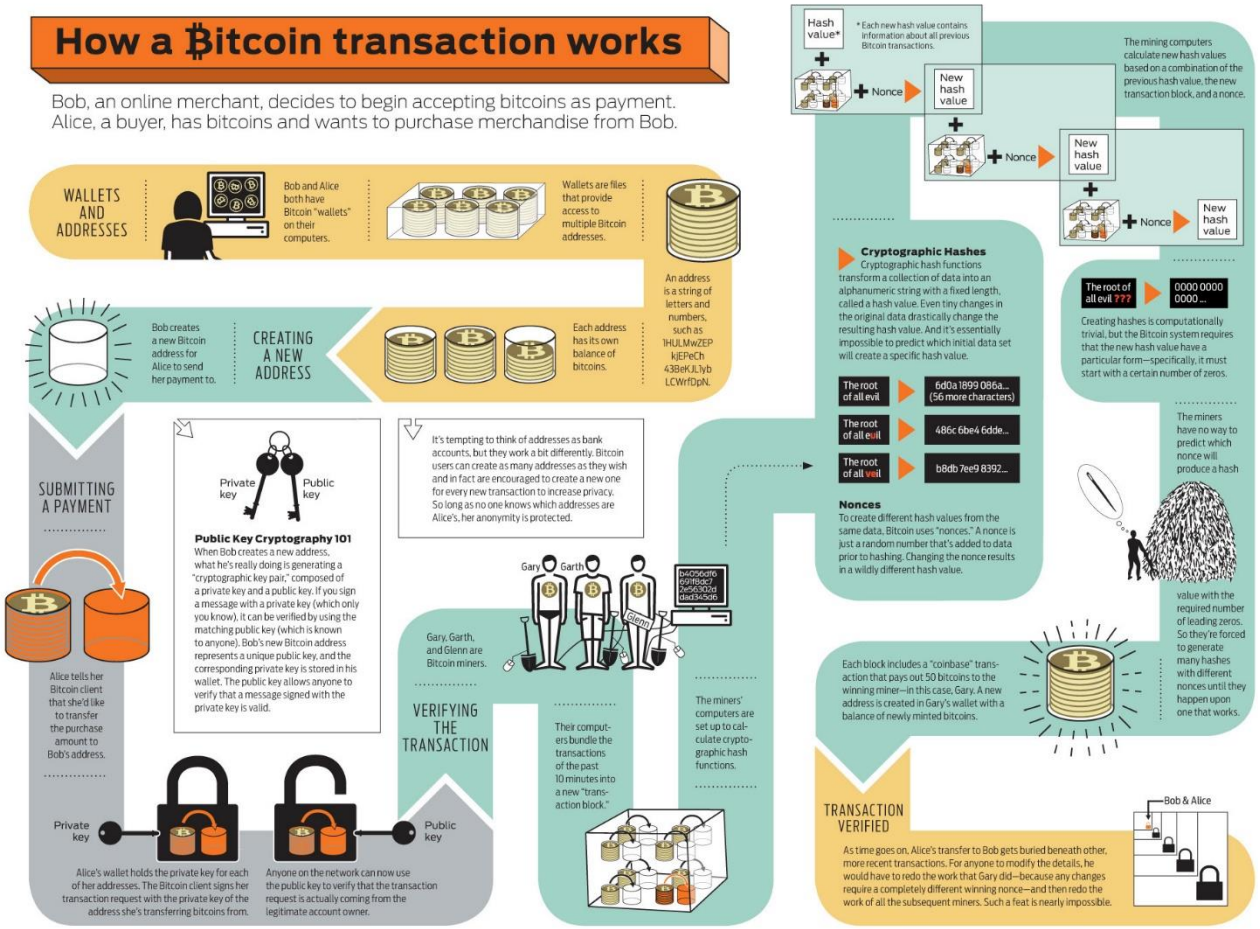
DragonFly: P2P file distribution within a large enterprise

Pulling Images by Dragonfly





Who Is Using Dragonfly?



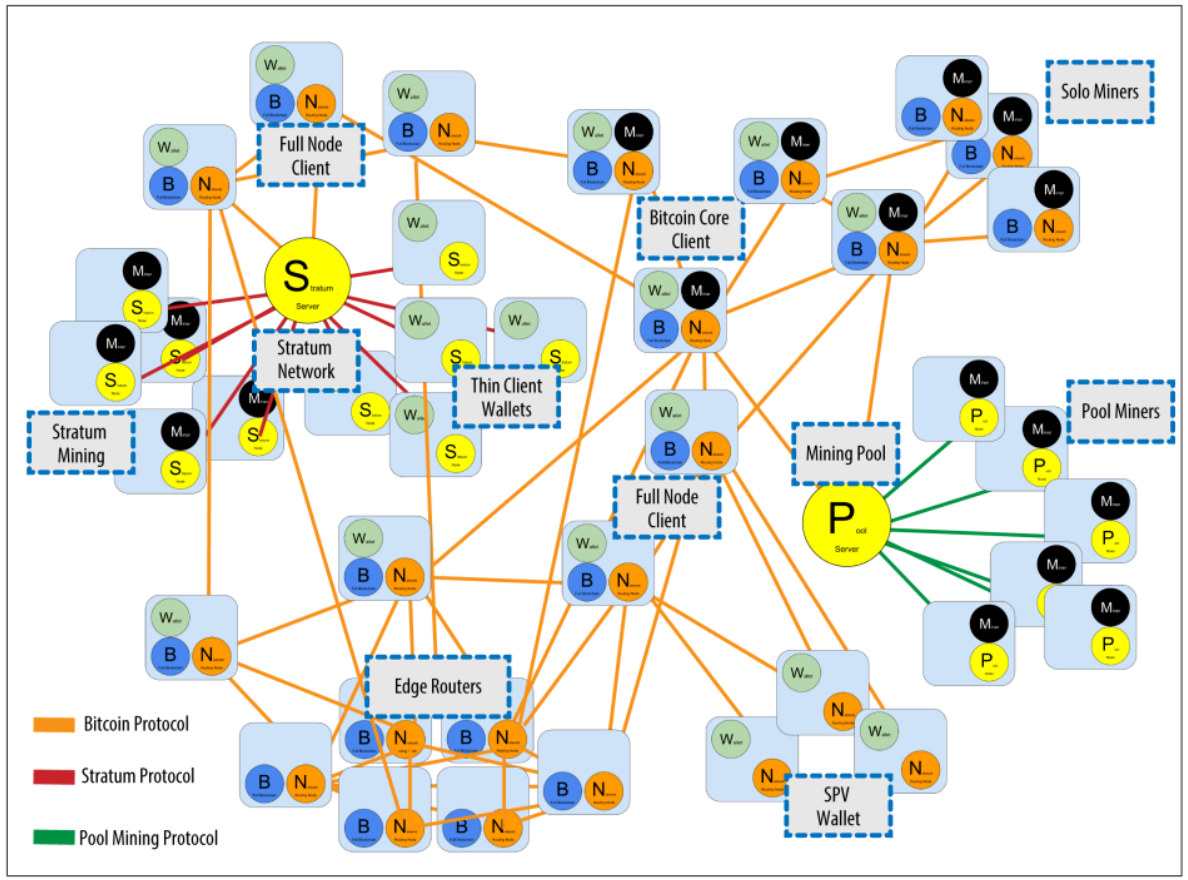


Figure 6-3. The extended bitcoin network showing various node types, gateways, and protocols

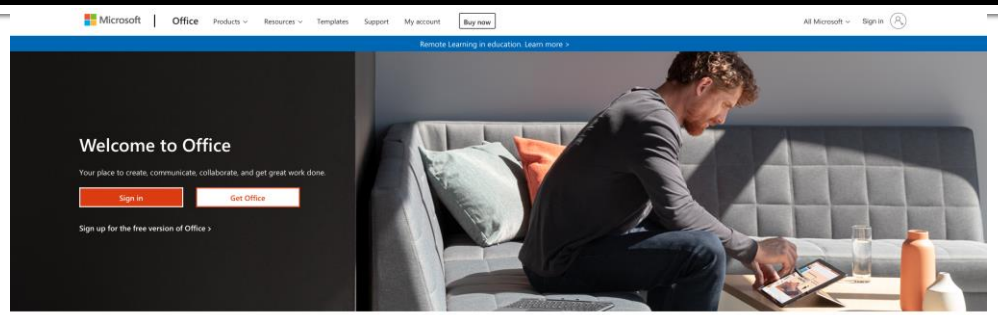
Bitcoin is a worldwide cryptocurrency and digital payment system. The system is peer-to-peer, and transactions take place between users directly, without an intermediary. These transactions are verified by network nodes and recorded in a public distributed ledger called a blockchain.

Software as a Service (SaaS) and Cloud Computing

What do they have in common?

"shrink-wrapped software"
➔ SaaS (Software as a Service)

1. Data stored safely in cloud
   - Lose device ≠ lose data
   - Groups can easily share/collaborate



2. 1 copy of SW, single HW/OS environment
   - no compatibility hassles for developers, users
   - beta test new features on 1% of users?
   - upgrade == deploy to cloud

3. Continuous customer feedback possible, respond by rolling out new changes quickly
   - Forever beta testing
   - ideal match for Agile?

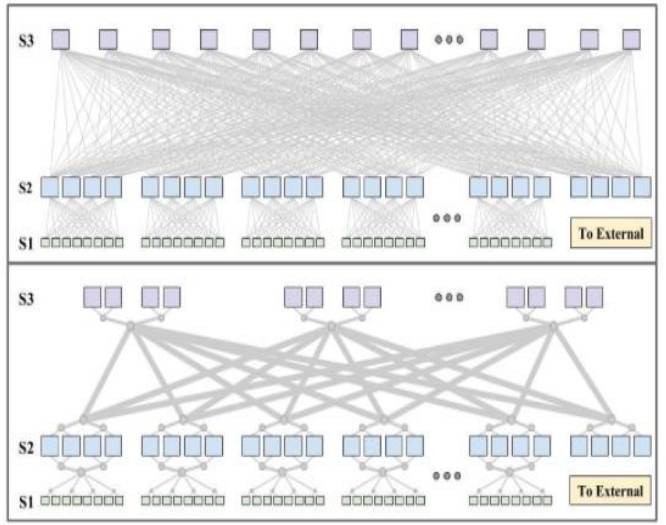
Ideal hardware infrastructure for SaaS?

- SaaS's 3 demands on infrastructure
  1. Communication
  2. Scalability & elasticity
  3. Dependability ("24 x 7")

Services on Clusters

- (Mostly) commodity computers connected by (mostly) commodity Ethernet switches
  1. More scalable than conventional servers
  2. Much cheaper than conventional servers
     - 20X for equivalent vs. largest servers
  3. Dependability via extensive redundancy
  4. Few operators for 1000+ servers
     - Careful selection of identical HW/SW
     - Virtual Machine Monitors and containerization (Docker, Kubernetes) simplify operations

Warehouse Scale Computers



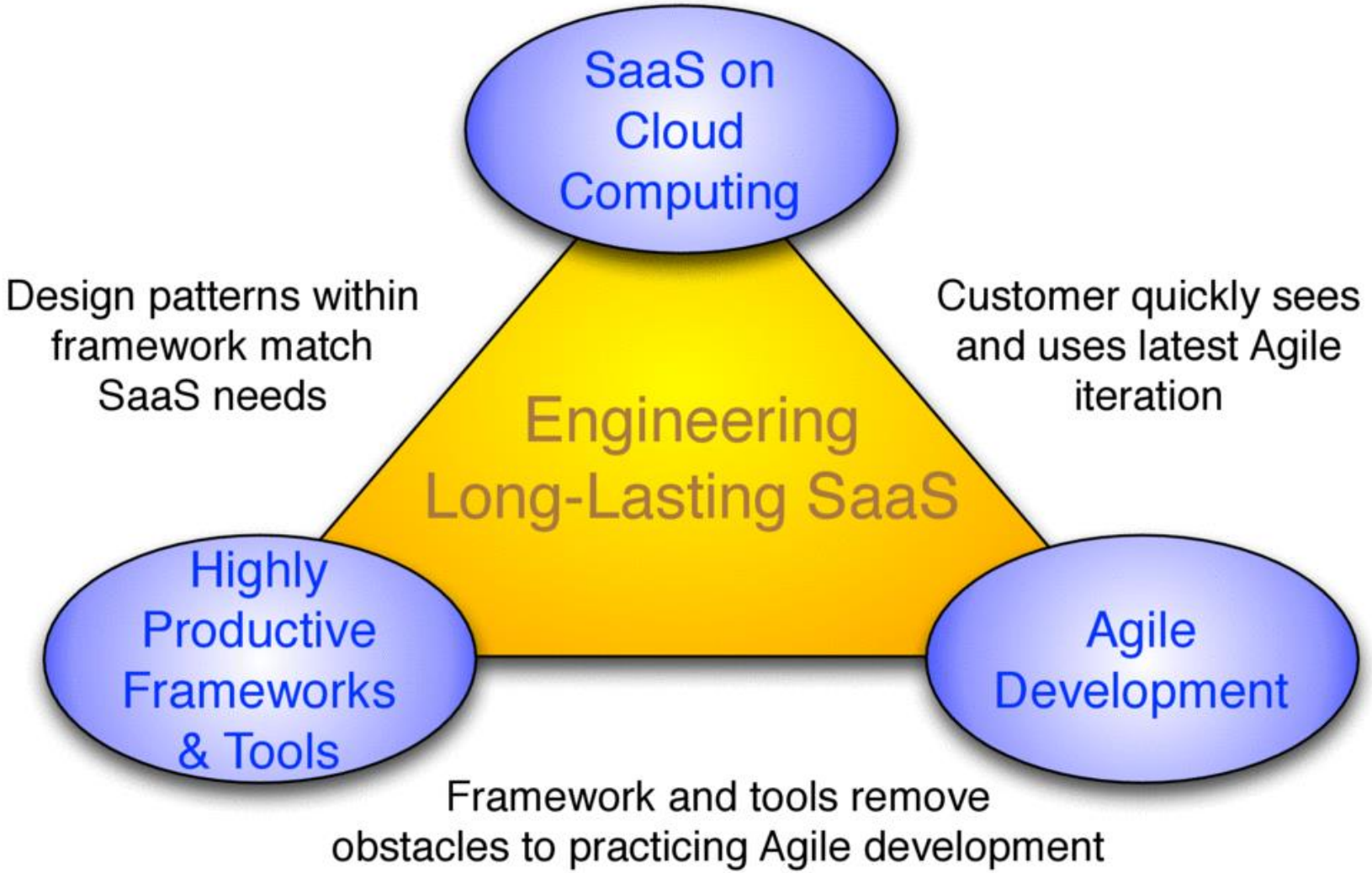
- Clusters grew from 1000➔ 100,000 servers based on customer demand for SaaS
- Economies of scale pushed down cost of largest datacenter by factors 3X to 8X
  - Purchase, house, operate 100K v. 1K computers
- But…traditional datacenters only 10%-20% utilized
- Idea: Earn $ offering pay-as-you-go usage at less than customer's costs for as many computers as customer needs

2007: Utility Computing arrives

- Buy computing, storage, communication *by the hour* (from about $0.02/server/hour) as a service

  1000 computers x 1 hour = 1 computer x 1000 hours
- *Public/utility cloud computing:* Amazon EC2, Google Cloud, Microsoft Azure
  - Infrastructure managed by dedicated experts can lower Total Cost of Ownership (TCO)
  - Scale up & down instantly (elasticity)
- Success stories: FarmVille on AWS
  - Zynga FarmVille: 1M users in 4 days, 10M in 2 months, 75M in 9 months (Prior biggest game had ~5M users)
  - Netflix: owns ~none of its streaming infrastructure

SaaS matches Agile, and levels field for smaller players



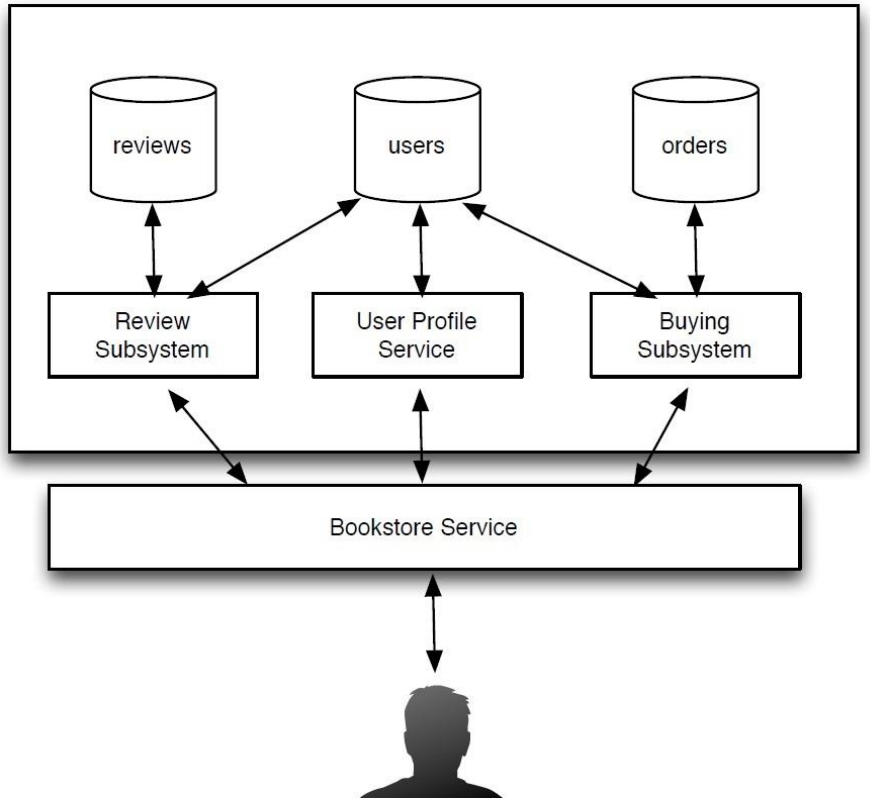
Caveat

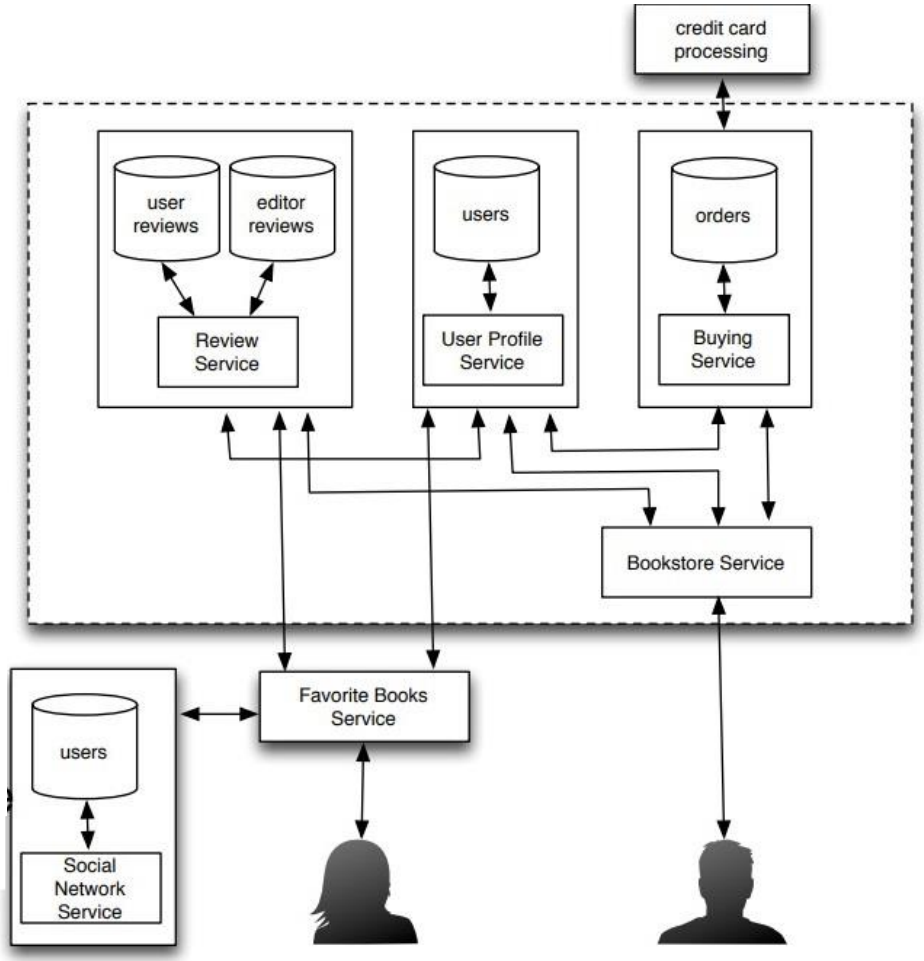
Inexpensive warehouse-size clusters give you the *potential* of unlimited scalability…

but you have to design scalability into the software too.

SOA: Service Oriented Architecture

- Set of independent composable services that can be combined as black boxes



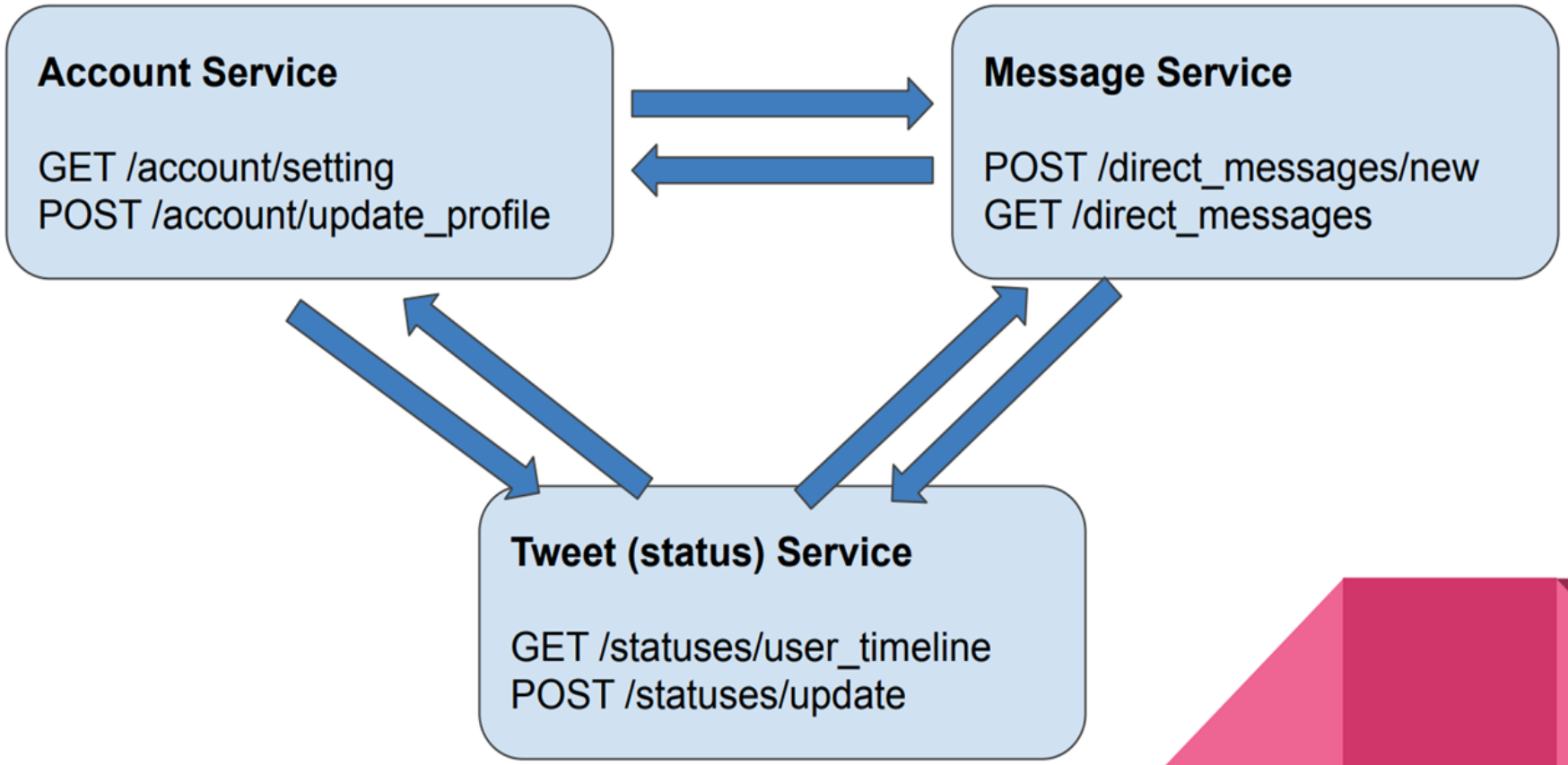


Service-Oriented Architecture (SOA)

- "Service-oriented architecture is a **client/server** design approach in which an application consists of software services and software service consumers (also known as clients or service requesters). SOA differs from the more general client/server model in its definitive emphasis **on loose coupling** between software **components**, and in its use of separately standing **interfaces**."
  - Gartner. 1996, Service Oriented Architecture

- "Essentially, SOA is a software architecture **that starts with an interface definition** and builds the **entire application topology** as a topology of interfaces, interface implementation and interface calls. SOA would be better-named 'interface-oriented' architecture."
  - Gartner. 2003, Service-Oriented Architecture Scenario

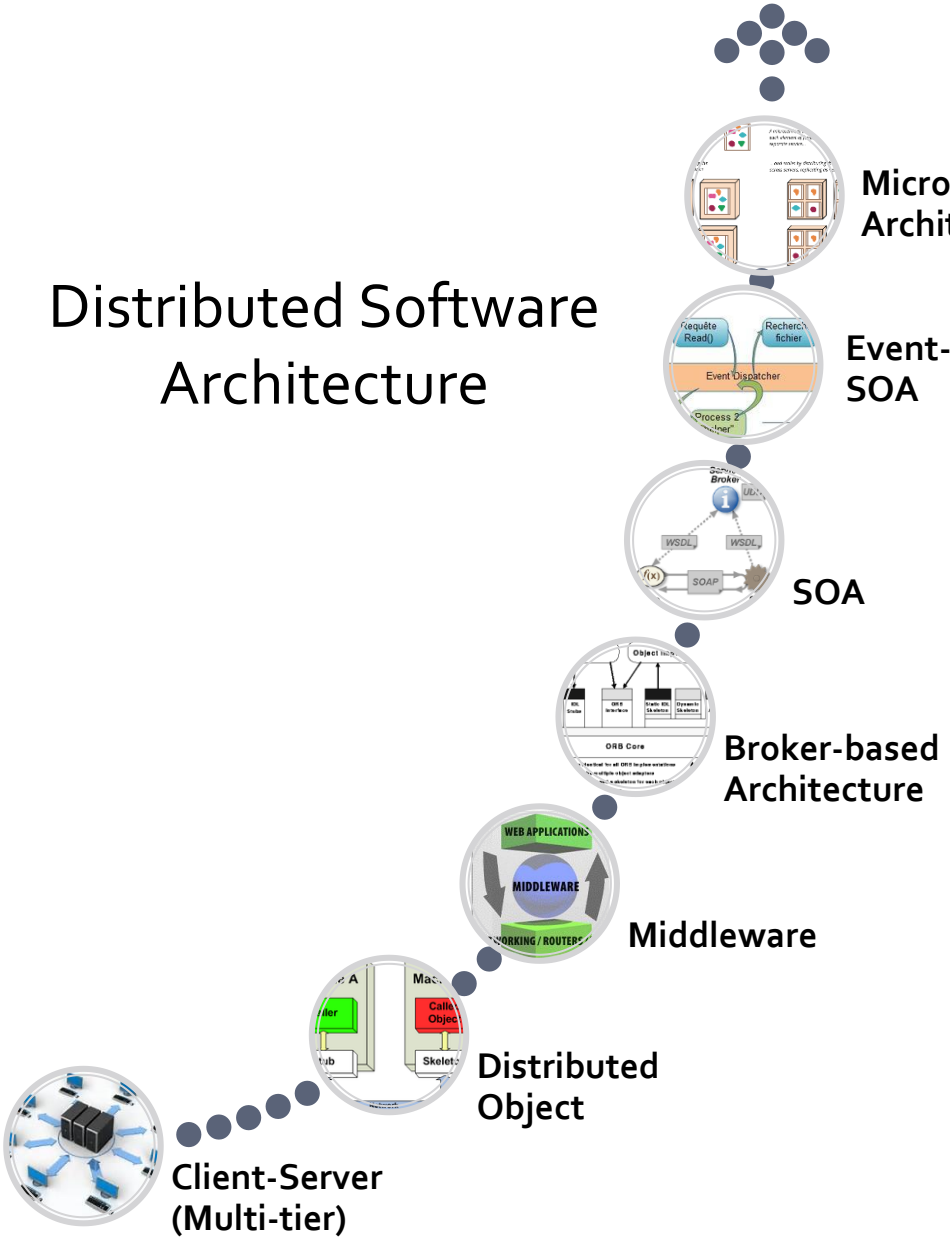
Twitter in SOA example



SOA Benefit & Cost Analysis

Reusable: can combine existing services where each service is implemented with their own language/framework (implementation hidden behind API)	Performance: for each service, need to dig into the software stack of a network interface due to layers of APIs, which can cause **performance penalty: State is split across services**
Easy testing: each service does only one thing, testing that one thing in isolation is easier Allows developers to use best tool for job for each service	**Dependability:** Partial Failure is possible in SOA since services can fail independently from others, so it is not just a measure of "app is working" vs "app isn't working". In cases like this testing can be harder (integration and system tests, validation tests)
Agile Friendly: works best with small-medium teams, since SOA allows small teams to develop services and then combining them.	Development Work: each service has to build their **own interface** instead of a single monster interface for the entire application, REST simplifies this each service essentially being end-to-end services that can stand on their own
Same team is responsible for developing, testing and operating the service so **customer-feedback cycles can be done quickly and efficiently**	Dev/Ops: developers need to know about operations to be able to manage functionality and performance. If "**you build you run**" then you need to know ops deeper

Distributed Software Architecture



Component + **Connector**

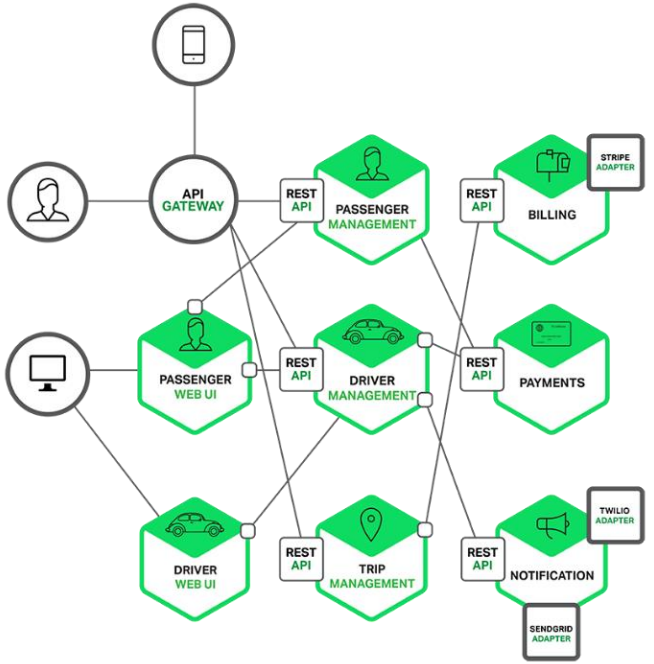
- **Component patterns**
  - Distributed process
  - Distributed object
  - **Service**
  - **Microservice**
- **Connector patterns**
  - **Remote Procedure Call （RPC)**
  - **REST**
  - Stub/Skeleton of Distributed object
  - Middleware
  - Broker-based
  - Messaging
  - Event-driven

The ultimate goal: to deliver better software faster.

Micro-Service Architecture

- The Micro-service architectural style is an approach to developing a single application **as a suite of small services**, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery.

Industrial Microservice System:
WeChat system: 3,000 services, over 20,000 nodes Netflix system: 500+ microservices, about two billion API requests every day

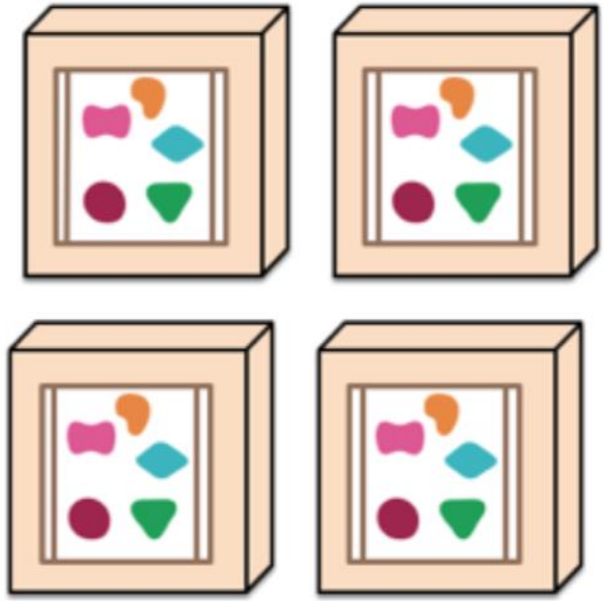


https://www.martinfowler.com/articles/microservices.html
http://2012.33degree.org/pdf/JamesLewisMicroServices.pdf

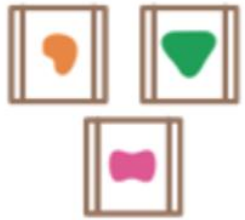
From Monolithic Application to Microservices

A monolithic application puts all its functionality into a single process...

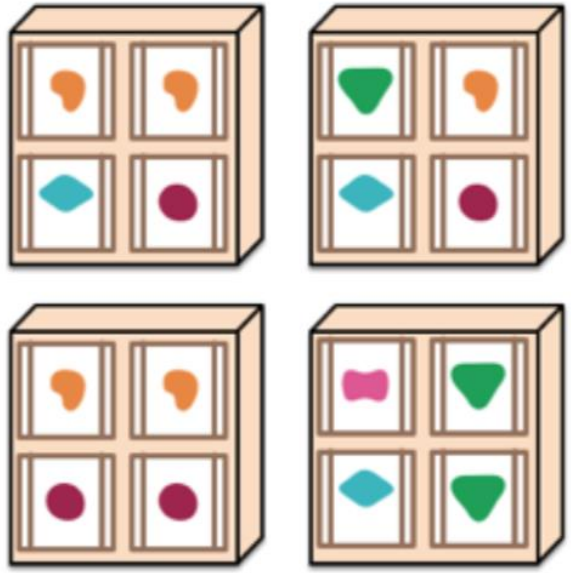
... and scales by replicating the monolith on multiple servers



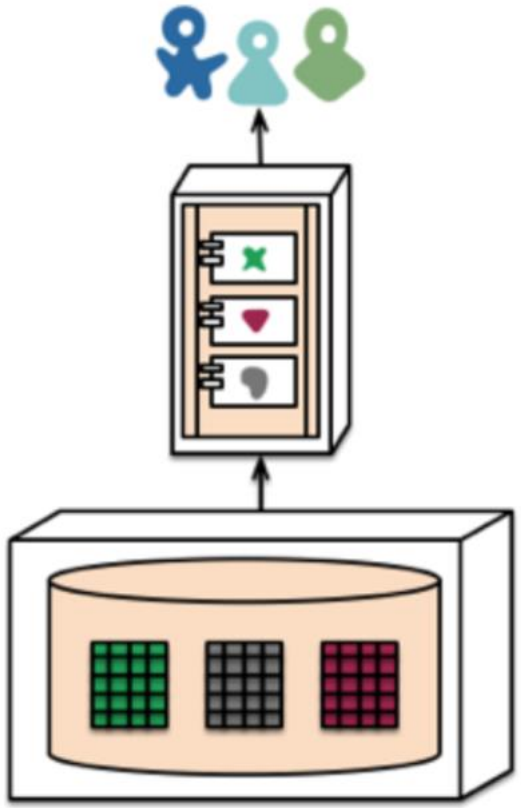
A microservices architecture puts each element of functionality into a separate service...



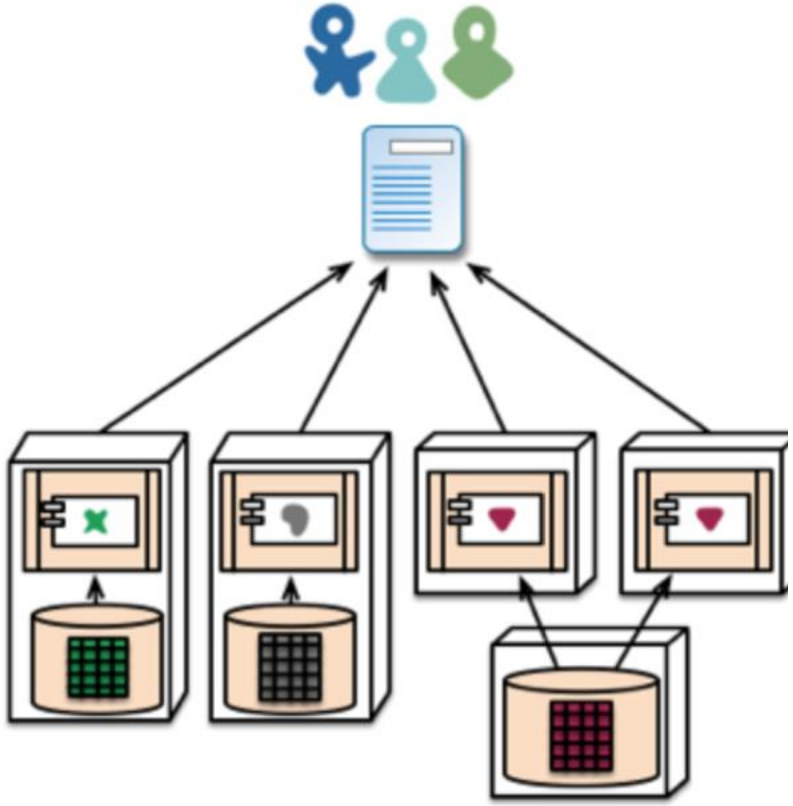
... and scales by distributing these services across servers, replicating as needed.



Database Deployment



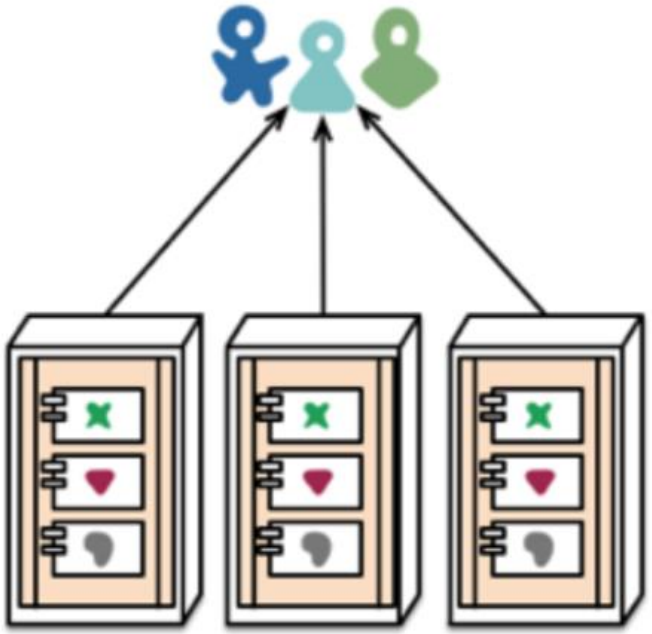
monolith - single database



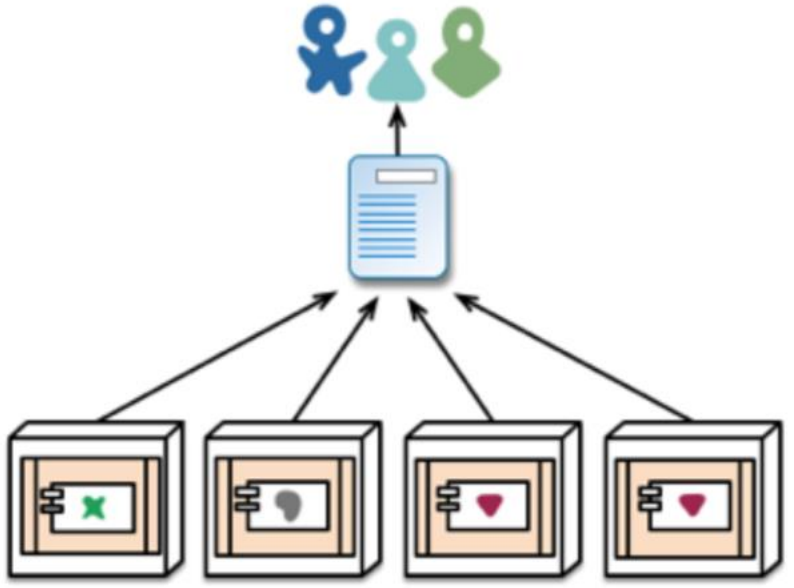
microservices - application databases

Credit: James Lewis and Martin Fowler, Microservices

Module Deployment



monolith - multiple modules in the same process



microservices - modules running in different processes

Credit: James Lewis and Martin Fowler, Microservices

Micro-Service Architecture: suites of independently deployable services

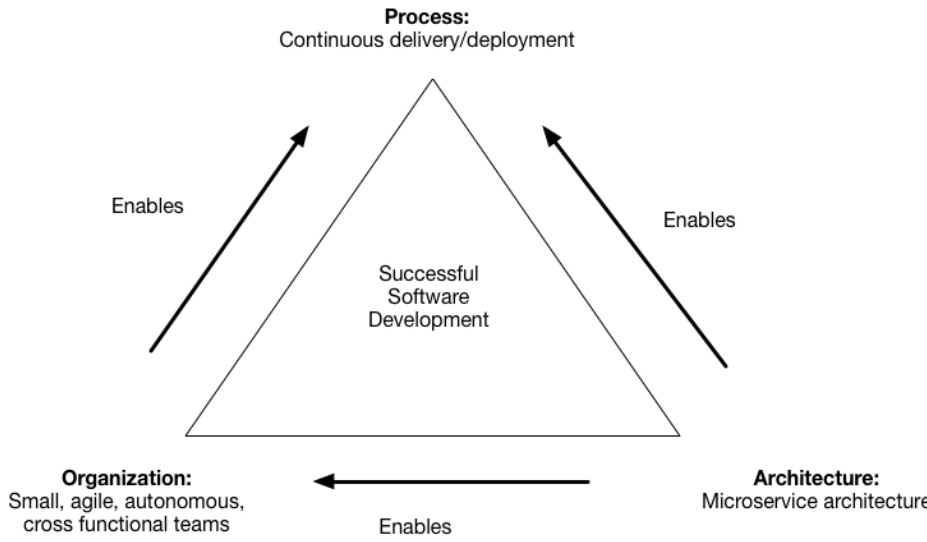
- A means to an end: enabling continuous delivery/deployment.
- Characteristics (J. Lewis and M. Fowler)
  - Using services as building blocks (components) through Published Interfaces.
  - Organized around business capabilities.
  - Development team takes full responsibility for the software in production.
  - Smart endpoints and dumb pipes
  - Decentralized control of languages

James Lewis

Martin Fowler

Micro-Service Architecture

- Decentralized database
  - Each service manage its own database, either different instances of the same database technology, or entirely different database system – an approach called **Polyglot Persistence**.
- Design for failure
  - Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operation and business relevant metrics.
- Evolutionary Design
  - See service decomposition as a further tool to enable application developers to control changes in their application without slowing down change.
  - Micro-services can have independent replacement and upgradeability.



Four generations of microservice architecture:

(a) Container orchestration.

(b) Service discovery and fault tolerance.

(c) Sidecar and service mesh.

(d) Serverless architecture.

Credit: Jamshidi et al., Microservices-– The Journey So Far and Challenges Ahead

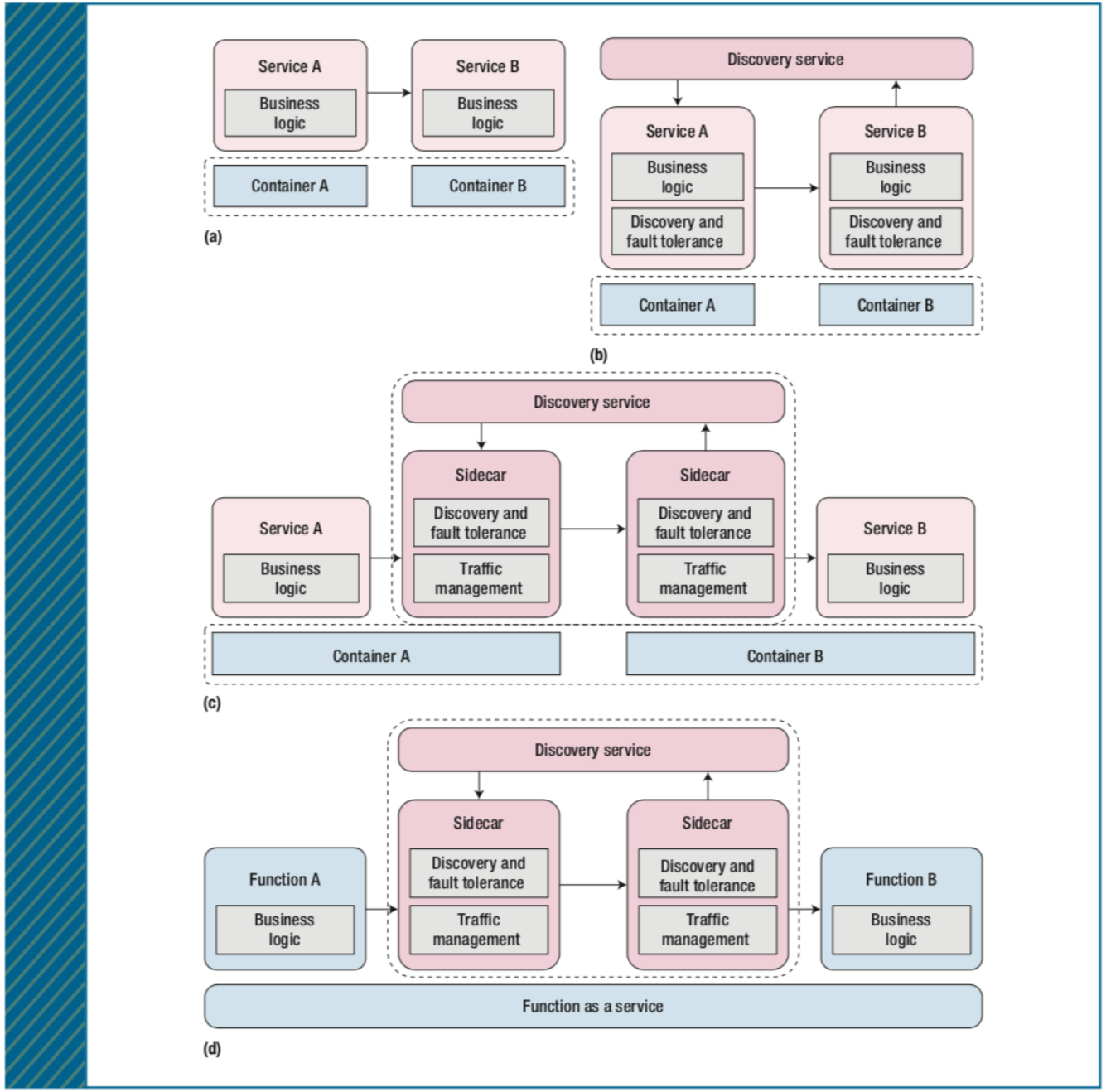
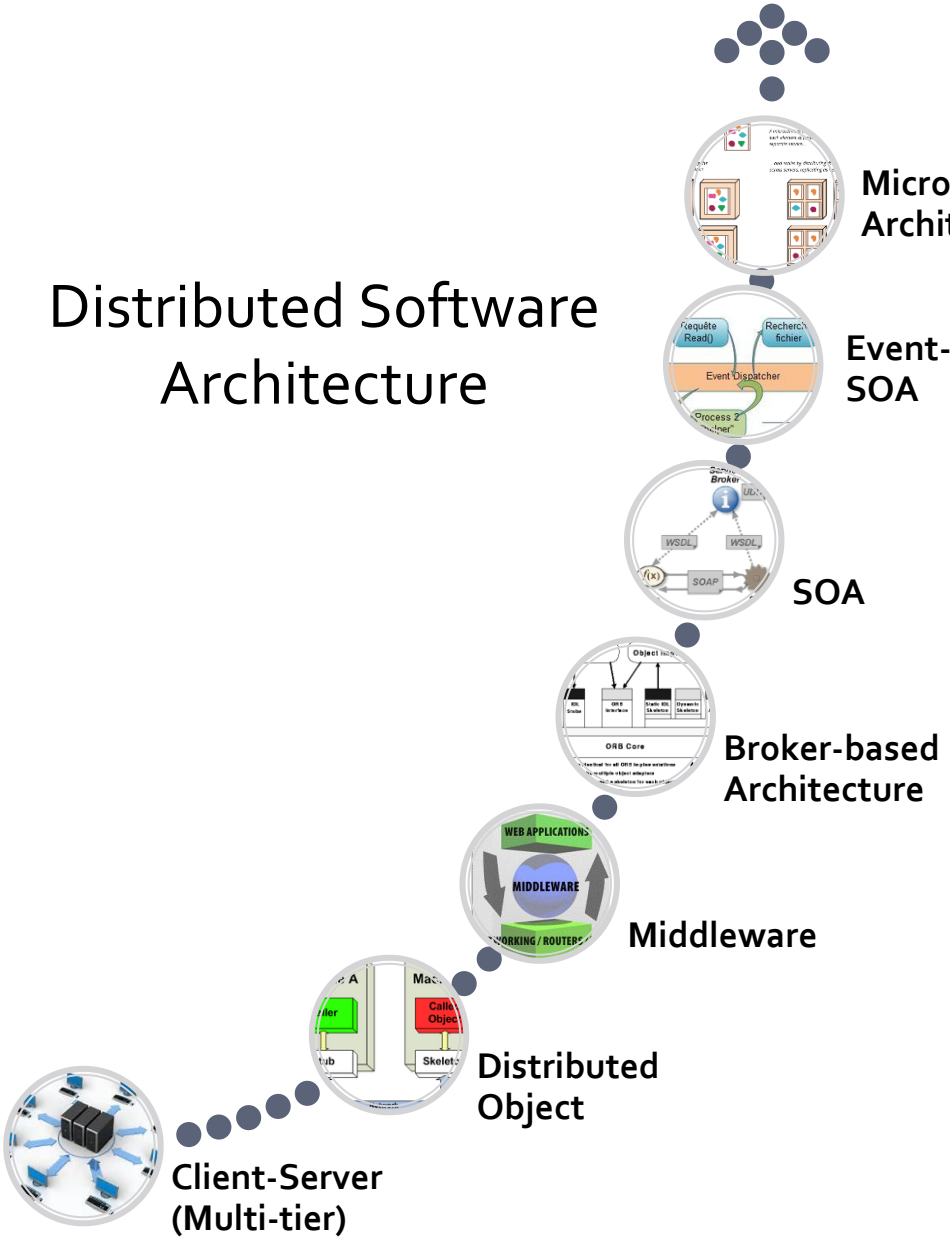


FIGURE 2. Four generations of microservice architecture. (a) Container orchestration. (b) Service discovery and fault tolerance. (c) Sidecar and service mesh. (d) Serverless architecture.

Distributed Software Architecture



Component + **Connector**

- **Component patterns**
  - Distributed process
  - Distributed object
  - **Service**
  - **Microservice**
- **Connector patterns**
  - **Remote Procedure Call （RPC)**
  - **REST**
  - Stub/Skeleton of Distributed object
  - Middleware
  - Broker-based
  - Messaging
  - Event-driven

The ultimate goal: to deliver better software faster.

REST:
Everything is a Resource

Which API framework wins?

- SOAP? WSDL? UDDI? XML-RPC? CORBA?
- Internet evolves by developer consensus…
  - easy for developers to understand/use
  - easy to express common use cases, possible to do rare ones
  - no burdensome license required
  - promotes code design matched to Web stds (e.g., SOA)
- And the winner is…REST

REST (Representational State Transfer)—Roy Fielding, 2000

- Route != *page* or *action*, but *resource & operation*
  1. What is primary *resource* affected?
  2. What operation is to be done, and what are possible results and side effects?
  3. What other data needed for operation, and how specified?
- A resource may have multiple possible *representations*
- Responses may include hyperlinks to additional resources
- A RESTful service or API is one whose operations follow these guidelines

Web apps as RESTful services

- A RESTful Web app is a service whose resources and operation results can have multiple representations, the most common of which are:
  - JSON (usually; rarely XML) for programmatic access
  - HTML for human interface

- Thus, the key design question for such an app is: *What are the resources, what are the relationships among them, and what operations are allowed?*

Basic Operations: CRUDI

Action	Meaning	Usual HTTP method
Create	Create new resource instance	POST
Read	Retrieve resource instance	GET
Update	Modify existing resource instance in place	PATCH or PUT
Delete	Destroy resource instance	DELETE
Index or List	Enumerate (filtered?) collection of resources	GET

- e.g. `GET /movies`

```
{"movies":
[
 {"id": 253, "title": "Hidden Figures"},
 {"id": 254, "title": "Raw"},
 {"id": 255, "title": "The Big Sick"},
   ....
 {"id": 500, "title": "Interstellar"}
 ]
}
```

Examples: *Possible* routes for manipulating Movie resources

```
R: GET /movies/253
```

Argument identifying the *primary resource* often appears as part of URI path

Primary resource in this case is movie catalog itself; additional params customize the operation

```
I: GET /movies?name=hidden+figures
```

```
C: POST /movies
```

What about updating?

- Loose convention: POST creates a resource, PUT or PATCH updates, DELETE deletes it

- Response may be headers with no body, or may include JSON object representing newly-created or newly-updated resource

Operations other than CRUDI?

- What about "add movie to my watch list"?

- *Tip:* When it seems awkward to express an operation in terms of resources, ask whether there is another resource type waiting to be defined, and what *relationship* it has to existing resource(s).

- *Sometimes this will result in a simpler design: new resource, but only CRUDI ops on it.*

Constraints to REST

- **Stateless**
  - The calls can be made independently of one another, and each call contains all of the data necessary to complete itself successfully.
    - E.g. API key, access token, user ID, etc.
  - A REST API should not rely on data being stored on the server or sessions to determine what to do with a call, but rather solely rely on the data that is provided in that call itself.
  - It helps increase the API's reliability by having all of the data necessary to make the call, instead of relying on a series of calls which may result in partial fails.
  - In order to reduce memory requirements and keep your application as scalable as possible, a RESTful API requires that any state is stored on the client – not on the server.

Constraints to REST

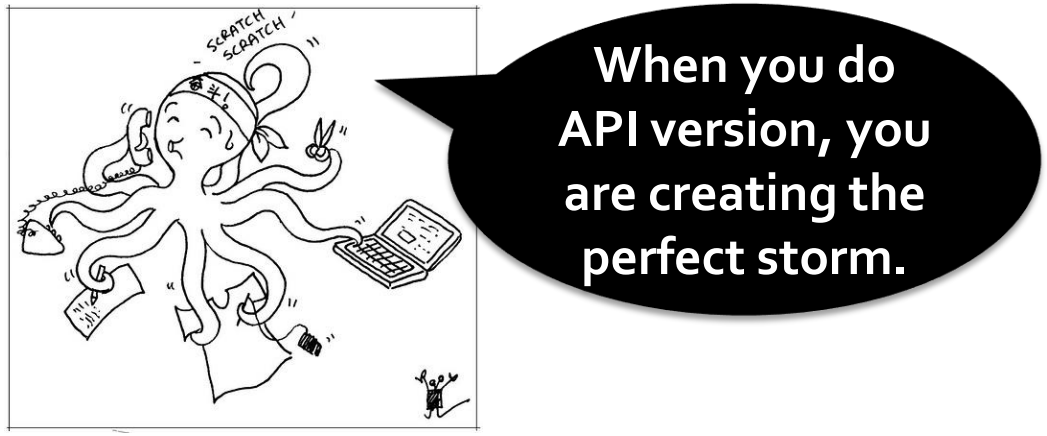
- **Cache**
  - Because stateless API can increase request overhead by handling large loads of incoming and outbound calls, a REST API should be designed to encourage the storage of cacheable data.
  - This helps not only greatly reduce the number of interactions with API, but also provide API users with the tools necessary to provide the fastest and most efficient apps possible.
  - Caching is done on the client side.

Constraints to REST

- **Uniform Interface**
  - The interface should provide an unchanging, standardized means of communicating between the client and the server
    - E.g. HTTP with URI, CRUD and JSON
  - Allows independent evolution of the application without having the application's services, or models and actions, tightly coupled to the API layers itself.
  - API should be decoupled from both technology stack and service layer so that as you make changes to your application's technology, the way the API interacts with users is not impacted.

Versioning – A Necessary Evil

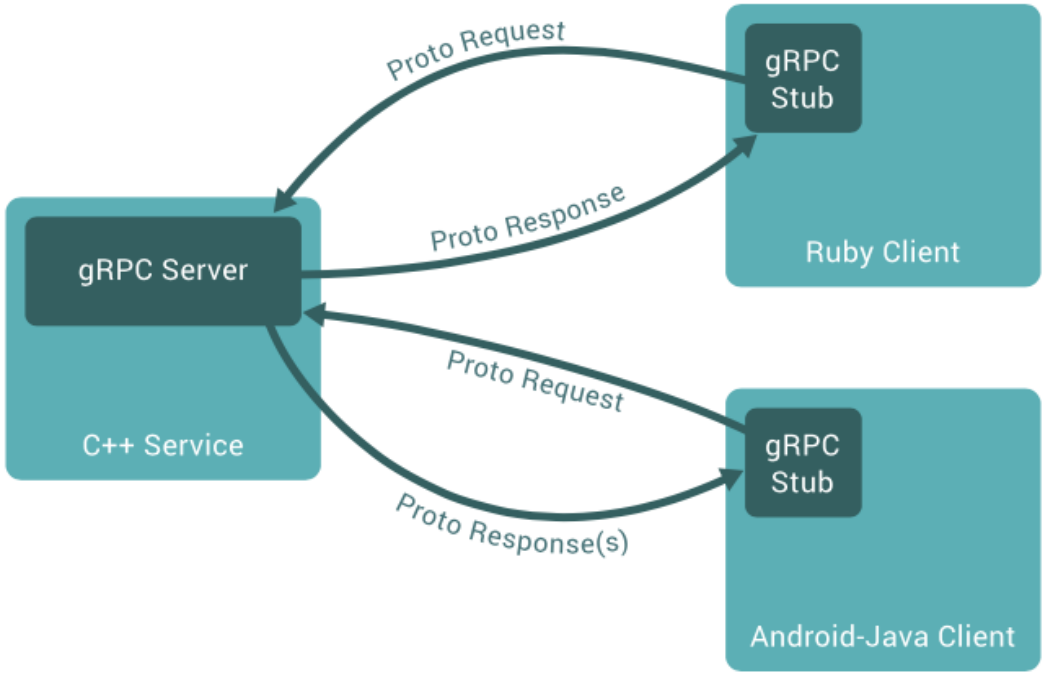
- Going into API design with a long-term focus, which means that versioning becomes a **last-resort** or doomsday option for when your API can no longer meet your needs.
- API Versioning is costly
  - Supporting Service
  - Maintenance cost
  - Usage confusion

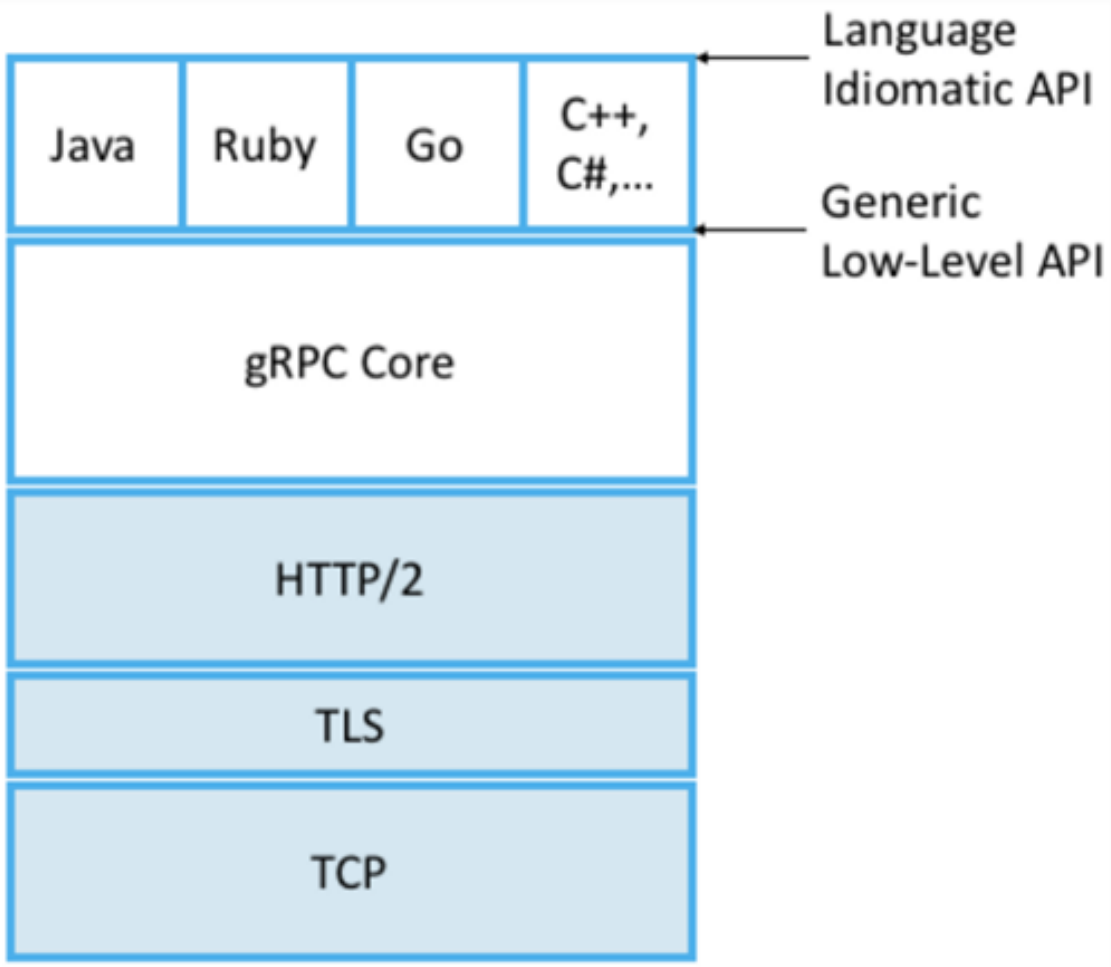


Summary: RESTful APIs (Representational State Transfer)

- *Everything is a resource.*
  - Basic ops: create, read, update, delete, list
  - Awkward to express desired operation?
    ➔ maybe new resource needs to be defined
- Resource can have different representations
  - Displayable HTML page for human user
  - JSON or XML data structure
  - hence *representational* in REST
- REST "won" because it's simple and matches engineering constraints of how Web evolved, e.g. stateless HTTP

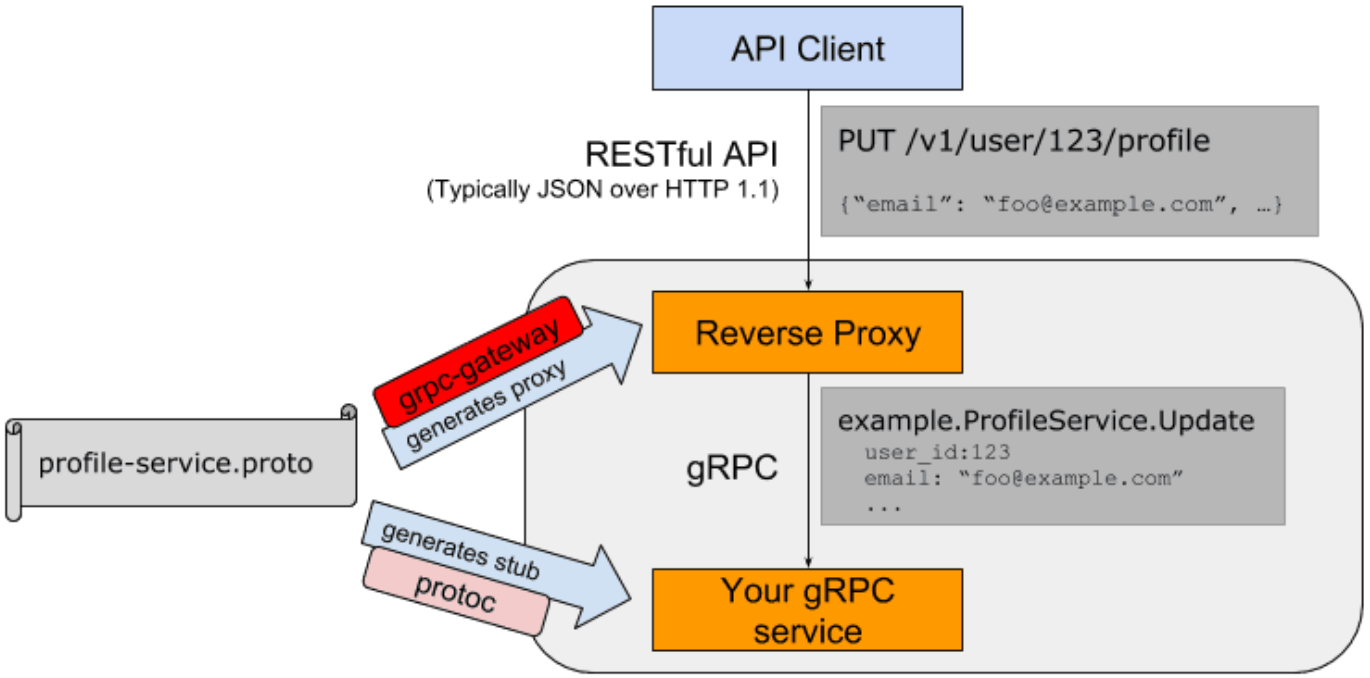
Remote Procedure Call (e.g., gRPC)





Tens of Billions of RPC calls per second at Google

RPC vs REST



RPC	REST
Library needed. Can return back any format, although usually tightly coupled to the RPC type (i.e. JSON-RPC)	No library support needed, typically used over HTTP Returns data without exposing methods Supports any content-type (XML and JSON used primarily)
Requires user to know procedure names Specific parameters and order Requires a separate URI/resource for each action/method Typically utilizes just GET/POST Requires extensive documentation Stateless More efficient	Single resource for multiple actions Typically uses explicit HTTP action Verbs (CRUDI) Documentation can be supplemented with hypermedia Stateless

Outline

- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

Architecture Design Patterns for Quality Attributes

Architecture Patterns for High Performance

常见的性能指标

- 响应时间

操作	响应时间
打开一个网页	几秒
在数据库中查询一条记录（有索引）	十几毫秒
机械磁盘一次寻址定位	4毫秒
从机械磁盘顺序读取1MB数据	2毫秒
从SSD磁盘顺序读取1MB数据	0.3毫秒
从远程分布式缓存Redis读取一个数据	0.5毫秒
从内存中读取1MB数据	0.25毫秒
网络传输2KB数据	1微秒

常见的性能指标

- 并发数
  - 系统能同时处理请求的数量
  - 注册用户数 >> 在线用户数 >> 并发用户数
- 吞吐量
  - 单位时间内系统处理的请求数量，体现系统的整体处理能力。
  - 请求数/秒，页面数/秒，访问人数/天，处理的业务数/小时
  - TPS（每秒事务数），HPS（每秒HTTP请求数），QPS（每秒查询数）
- 性能计数器
  - 描述服务器或操作系统性能的数据指标
  - 系统负载（正在被CPU执行和等待被执行的进程数目总和），对象与线程数，内存使用，CPU使用，磁盘与网络I/O

不同视角的性能指标

不同视角下的网站性能

用户视角

用户在浏览器上直观感受到的网站响应速度。

-- 用户计算机和网站服务器通信的时间

-- 网站服务器处理的时间

-- 用户计算机浏览器构造请求解析响应数据的时间

优化手段：前端架构、HTML式样、浏览器端的并发和异步、浏览器缓存策略、CDN服务、反向代理

开发人员视角

应用程序本身及其相关子系统的性能。

-- 响应延迟

-- 系统吞吐量

-- 并发处理能力

-- 系统稳定性

优化手段：缓存、集群、异步消息队列、代码优化

维护人员视角

基础设施性能和资源利用率。

-- 网络运行商的带宽能力

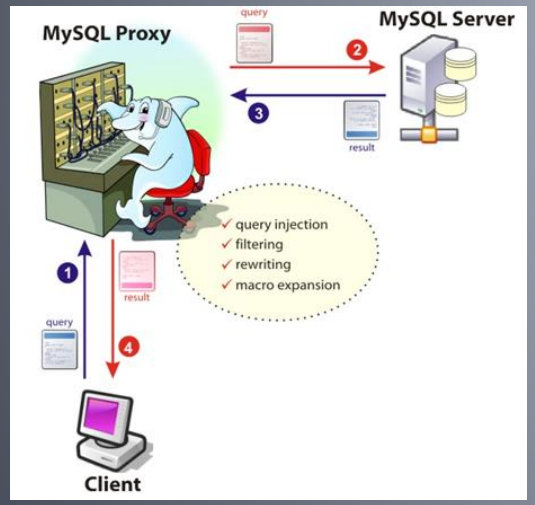
-- 服务器硬件的配置

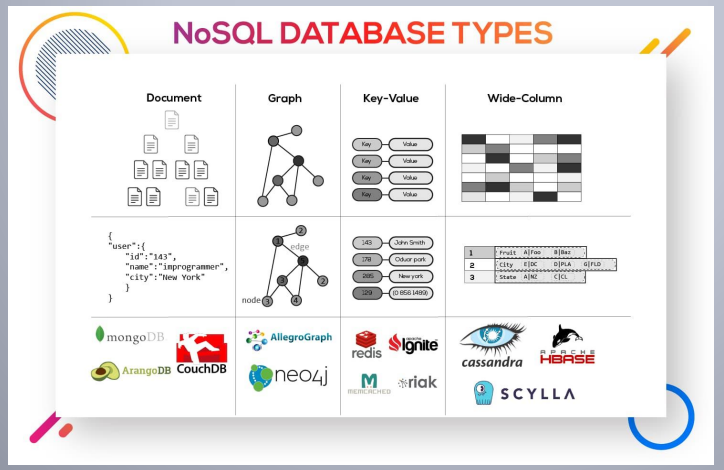
-- 数据中心网络架构

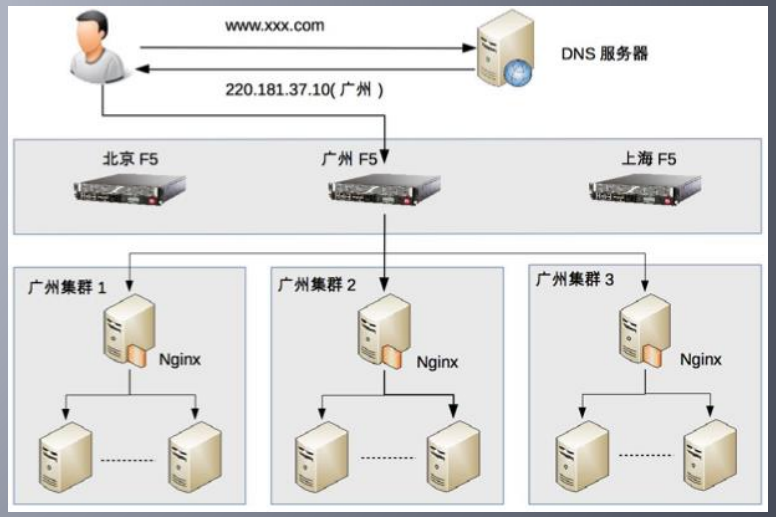
-- 服务器和网络带宽的资源利用率

优化手段：骨干网、定制服务器、虚拟化技术

High Performance Patterns

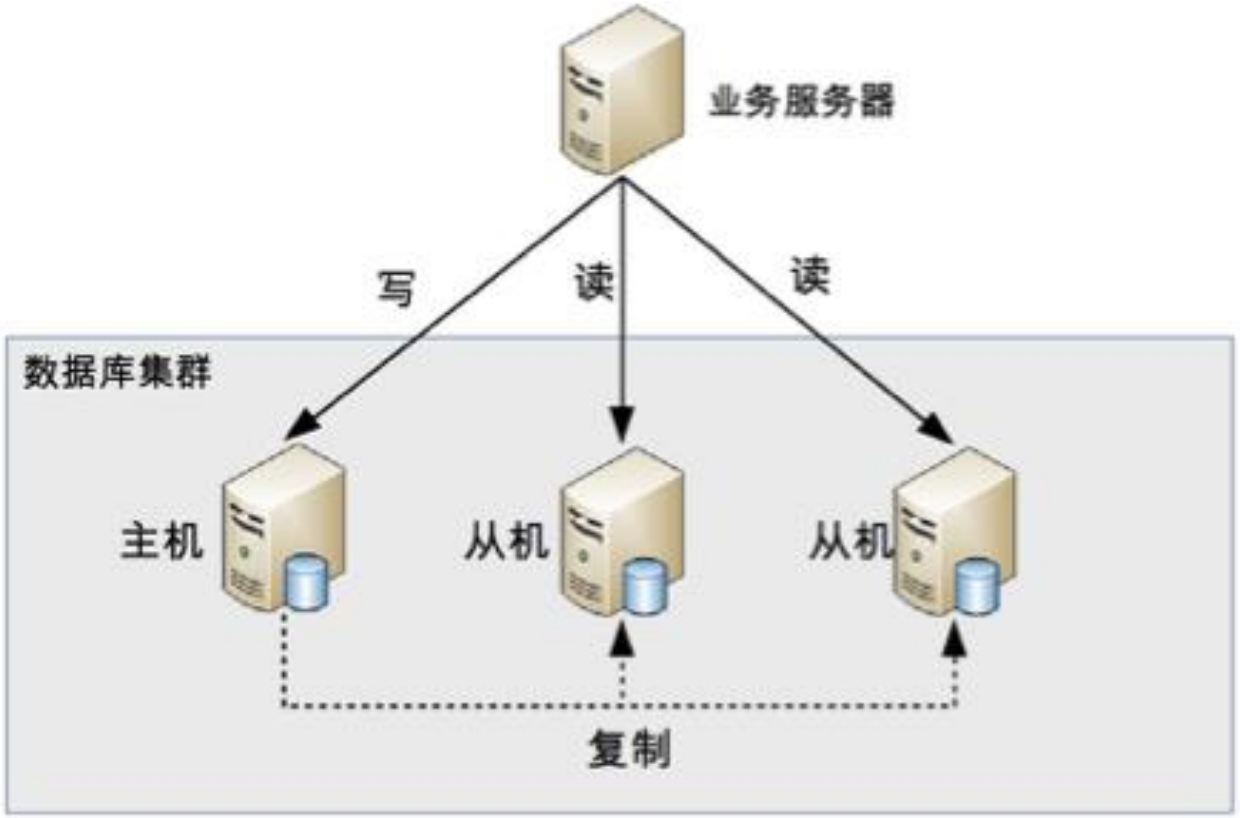






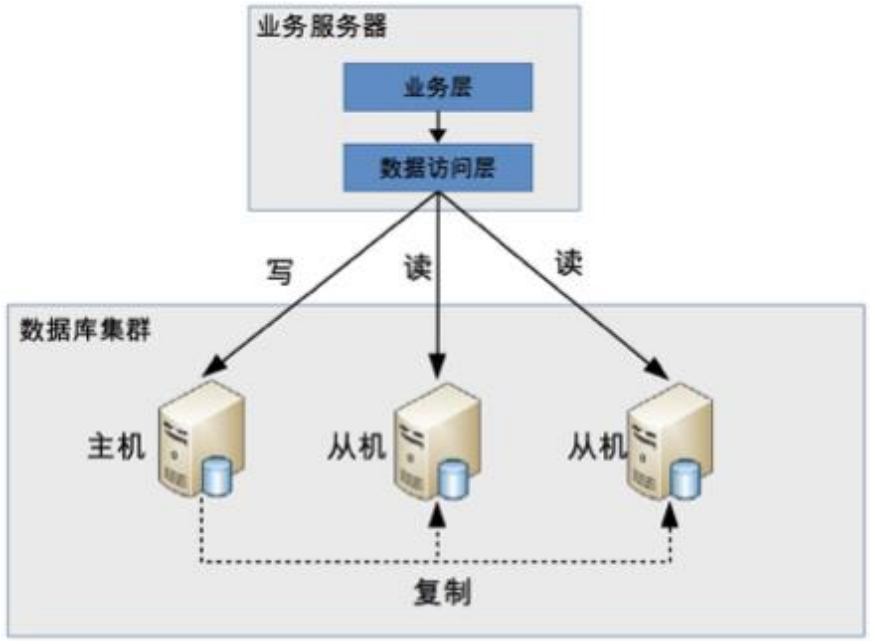
高性能关系数据库集群：读写分离

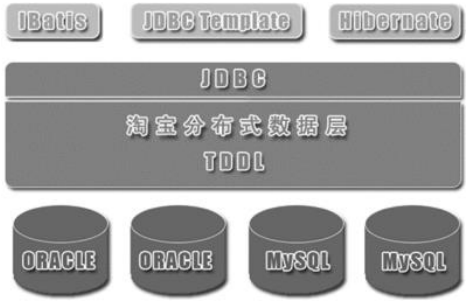
- 将数据库读写操作分散到不同的节点上



读写分离的分配机制

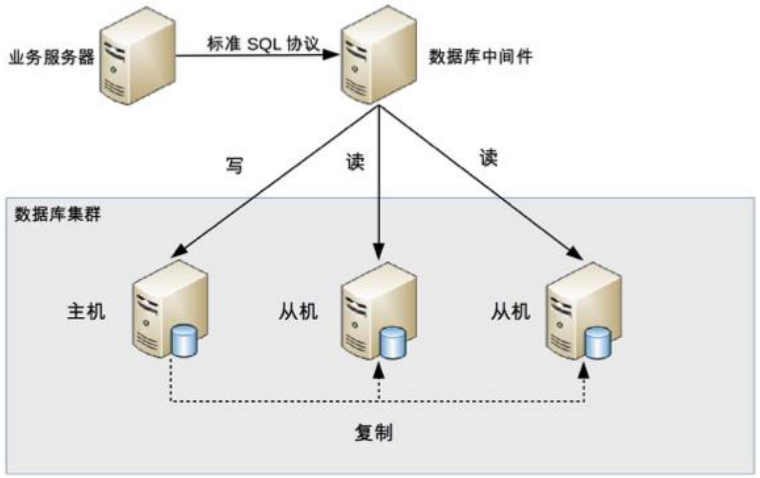
程序代码封装

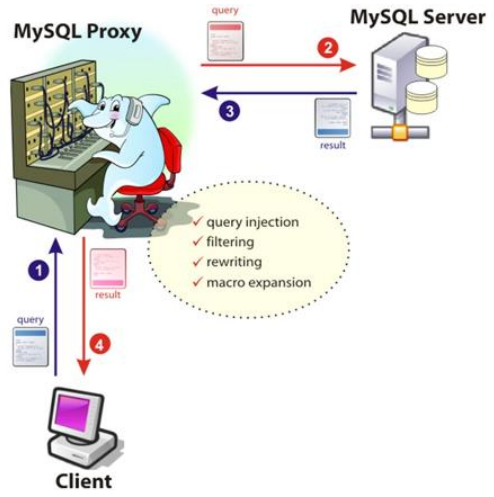




淘宝的 TDDL
(Taobao Distributed Data Layer)

中间件封装

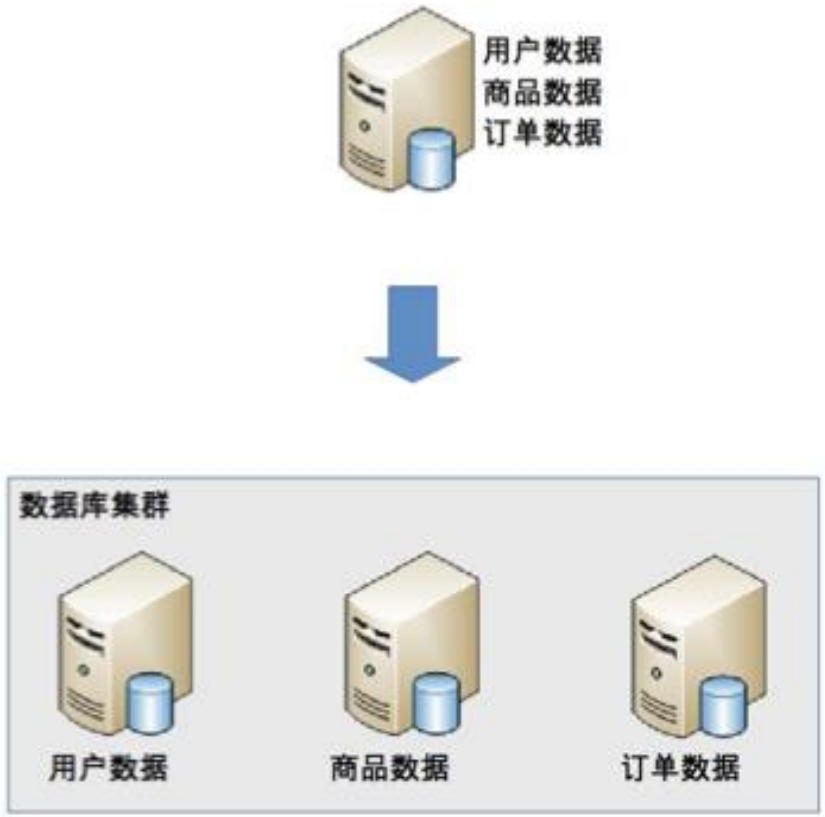




奇虎 360 公司数据库中间件 Atlas

高性能关系数据库集群：业务分库

- 按照业务模块将数据分散到不同的数据库服务器



一般来说，单台数据库服务器能够支撑10万用户量量级的业务

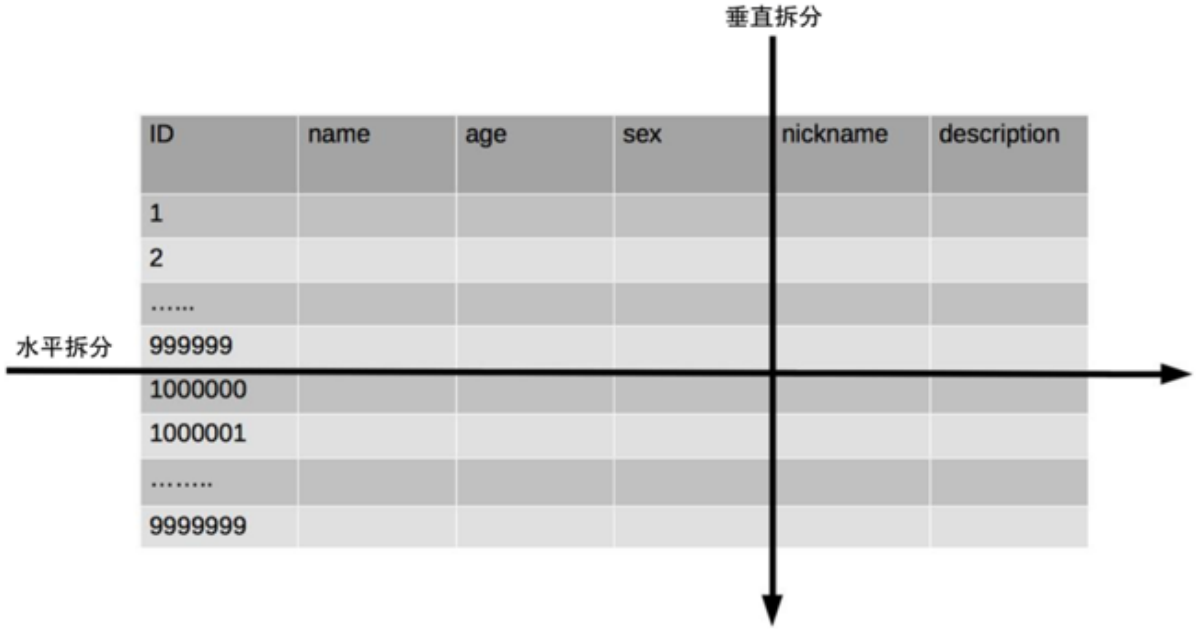
分库能够支撑百万甚至千万用户规模的业务

缺点：

操作复杂
代码工作量变大
成本变高

高性能关系数据库集群：分表

同一业务的单表数据也可能达到单台数据库服务器的处理瓶颈， 如淘宝几亿用户的数据



- 垂直分表适合将表中某些不常用且占了大量空间的列拆分出去
  - 查询次数可能变多
- 水平分表适合表行数特别大的表（千万级）
  - 范围路由、hash路由、配置路由

高性能NoSQL

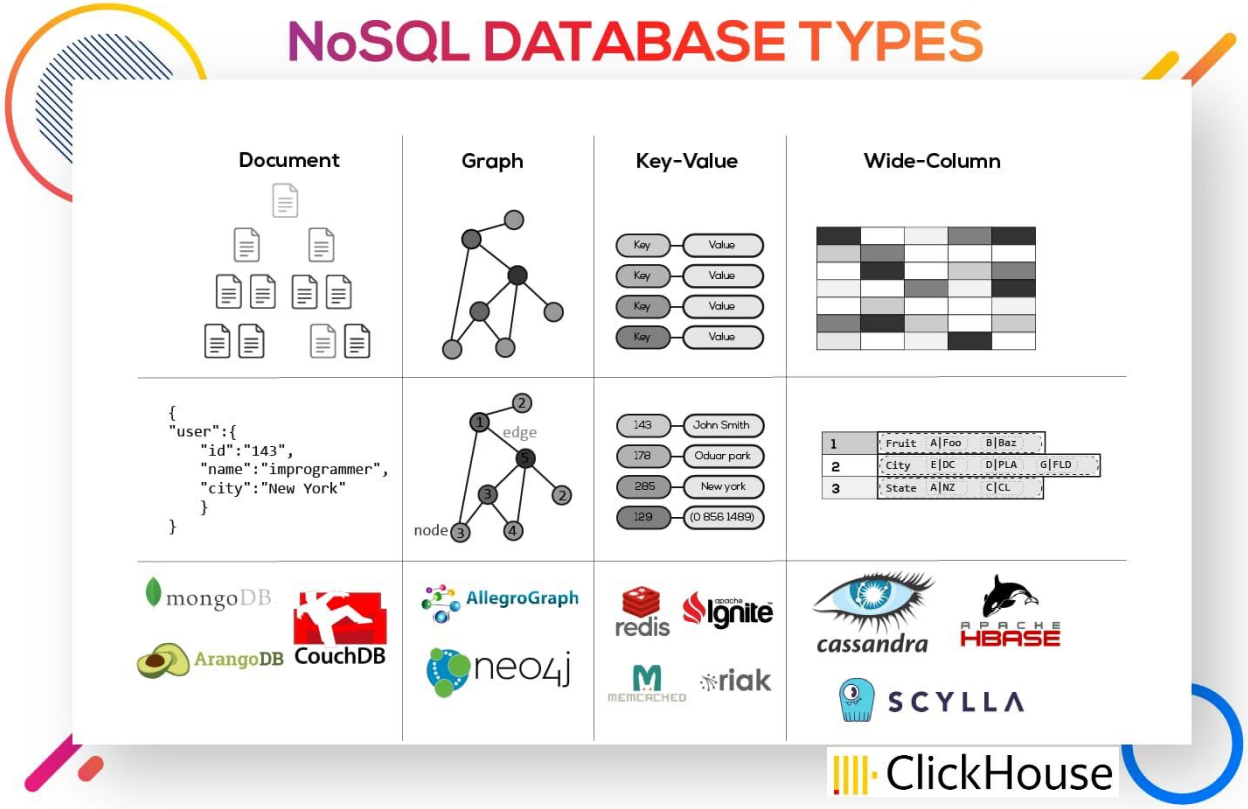
K-V 存储：解决关系数据库无法存储数据结构的问题，以 Redis 为代表。

文档数据库：解决关系数据库强 schema 约束的问题，以 MongoDB 为代表。

列式数据库：解决关系数据库大数据场景下的 I/O 问题，以 HBase 、ClickHouse为代表。

全文搜索引擎：解决关系数据库的全文搜索性能问题，以 Elastic Search 为代表。

图数据库： 解决图数据的性能问题， 以Neo4j为代表



将 NoSQL 作为 SQL 的一个有力补充，NoSQL != No SQL，而是 **NoSQL = Not Only SQL。**

NewSQL

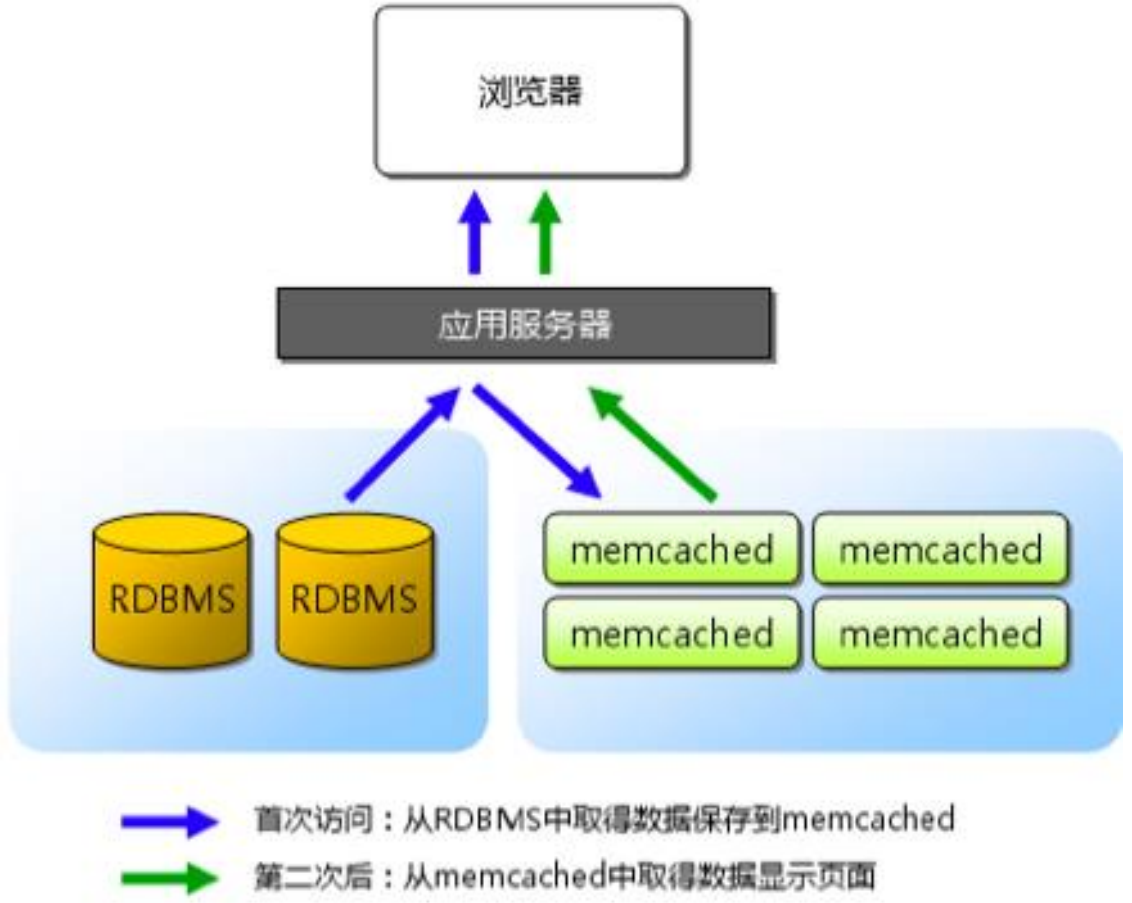
NewSQL is a class of **relational database** management systems that seek to provide the **scalability of NoSQL** systems for online transaction processing (**OLTP**) workloads while maintaining the **ACID guarantees** of a traditional database system.

Many enterprise systems that handle high-profile data (e.g., financial and order processing systems) are too large for conventional relational databases, but have transactional and consistency requirements that are not practical for NoSQL systems.

	Old SQL	noSql	newSql
关系模型	Yes	No	Yes
SQL语句	Yes	No	Yes
ACID	Yes	No	Yes
水平扩展	No	Yes	Yes
大数据	No	Yes	Yes
无结构化	No	Yes	No

高性能缓存架构

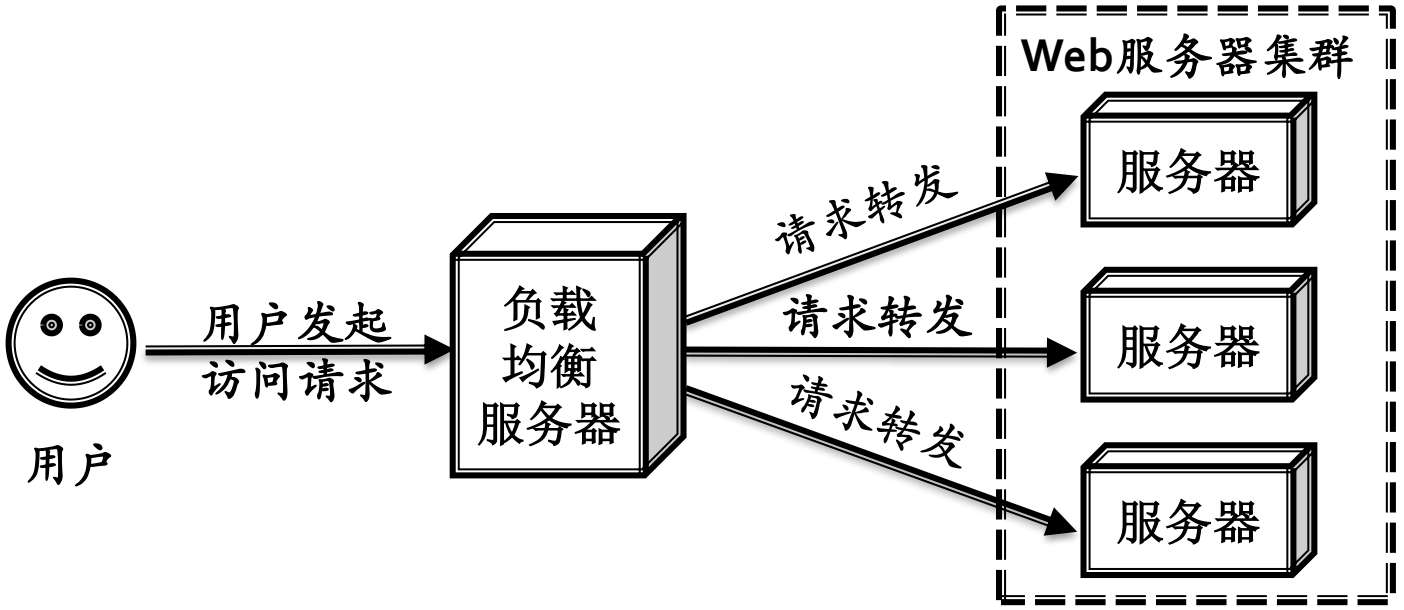
- 需要经过复杂运算后得出的数据，存储系统无能为力
- 读多写少的数据，存储系统有心无力
- 缓存系统需要解决：
  - 缓存穿透
  - 缓存雪崩
  - 缓存热点



单台 Memcache 服务器简单的 key-value 查询能够达到 TPS 50000 以上

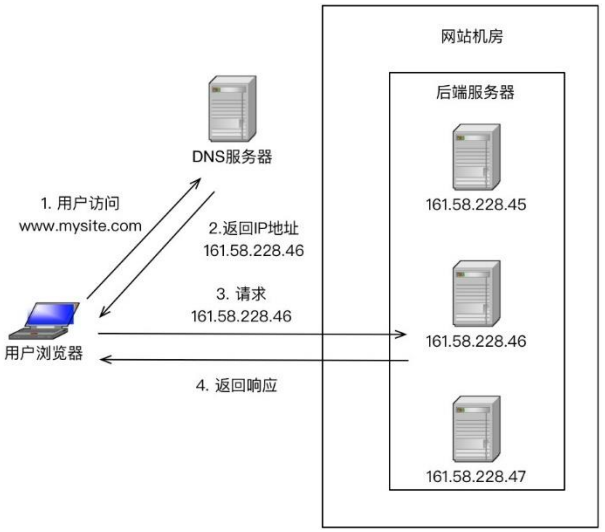
使用集群

- 在网站高并发访问的场景下，使用负载均衡技术为一个应用构建一个由多台服务器组成的服务器集群，将并发访问请求分发到多台服务器上处理，避免单一服务器因负载压力过大而响应缓慢，使用户请求具有更好的响应延迟特性。

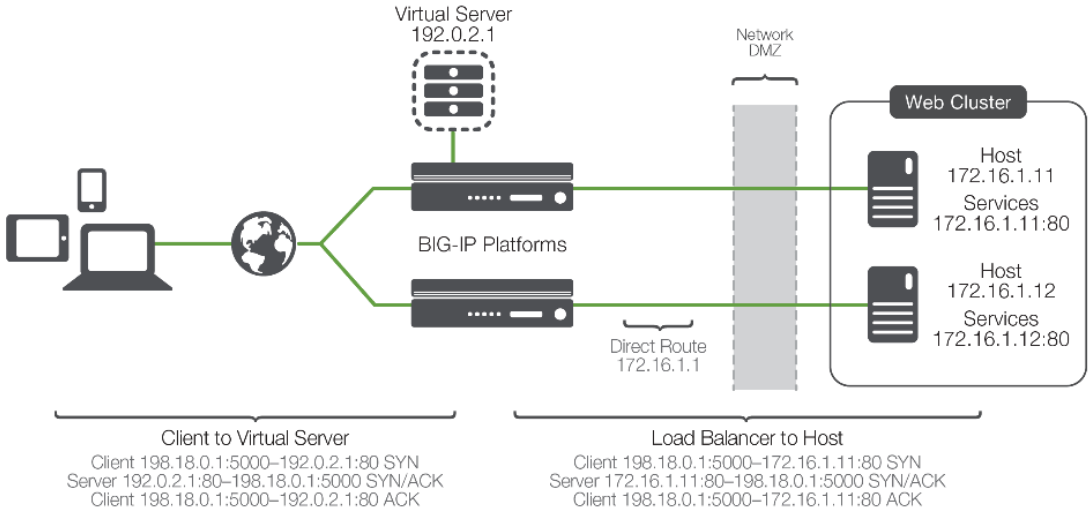


负载均衡技术分类

DNS 负载均衡



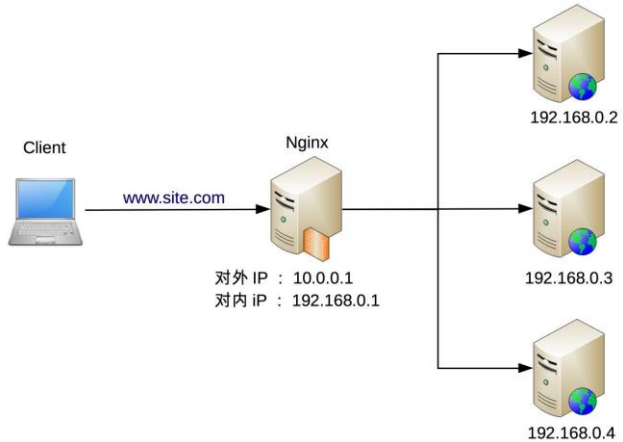
硬件负载均衡



LVS 四层

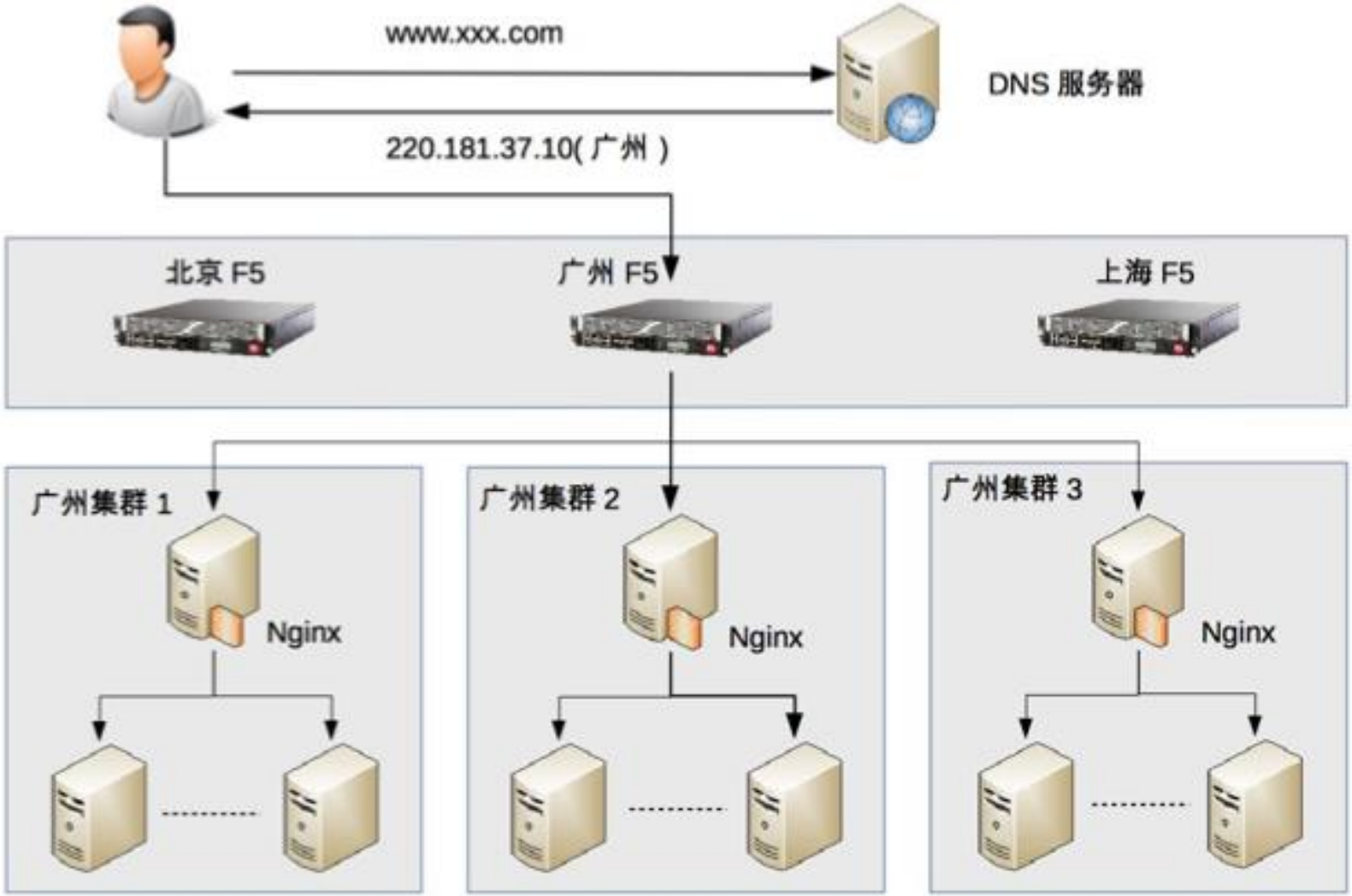
F5：硬件四层

软件负载均衡



Nginx: 应用层

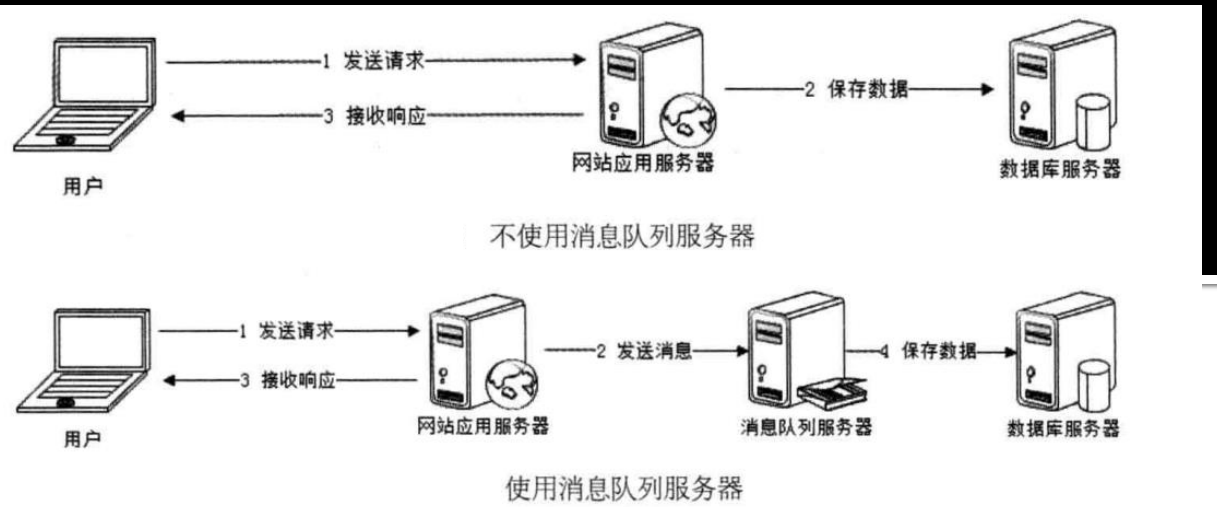
负载均衡典型架构：三种技术互补



高性能负载均衡：算法

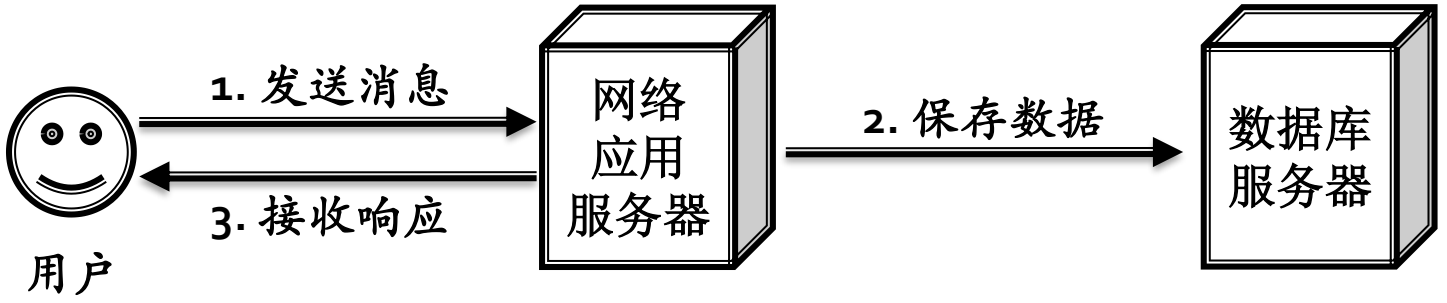
- 轮询
  - 服务器不宕机、服务器与负载均衡器不断连
- 加权轮询
  - 服务器性能有差异
- 负载最低优先
  - 连接数、HTTP请求数、CPU负载、I/O负载
  - 需要采样
- 性能最优优先
  - 处理速度
  - 需要采样
- Hash
  - 源地址
  - ID

应用服务器性能优化

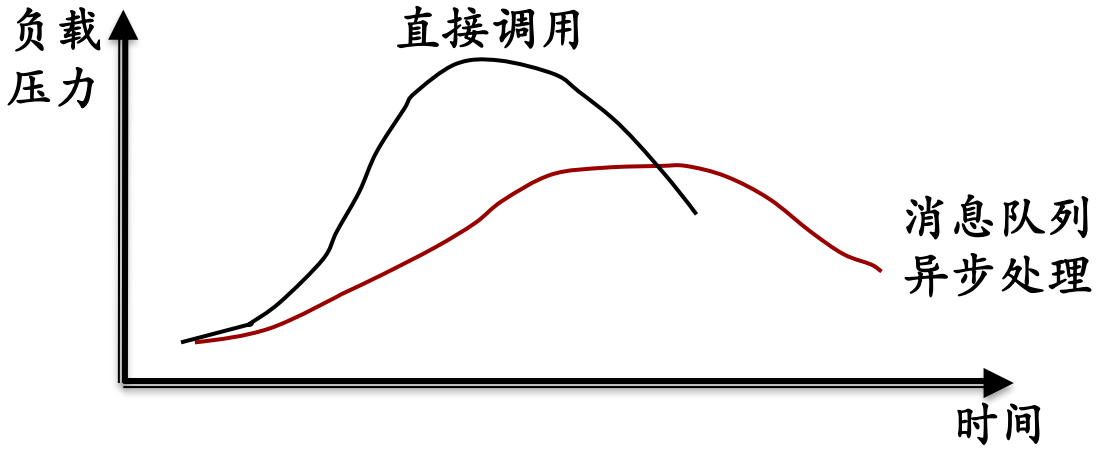


- 异步操作
  - 使用消息队列将调用异步
    - 不使用消息队列，用户的请求数据直接写入数据库，在高并发的情况下，会对数据库造成巨大的压力，响应延迟加剧。
    - 使用消息队列后，用户请求的数据发送给消息队列后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。
  - 消息队列具有很好的削峰作用
    - 通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。
    - 电子商务网站促销活动中，合理使用消息队列，可有效抵御促销活动刚开始大量涌入的订单对系统造成的冲击。

不使用
消息队列
服务器



使用
消息队列
服务器



任何可以晚点做到事情
都应该晚点再做。

Architecture Patterns for High Availability

Failure is INEVITABLE

- When designing a high-availability or safety-critical system, failure is not an option, it is almost INEVITABLE.

- What will make your system safe and available is planning for the occurrence of failure or (more likely) failures, and handling them with aplomb.

CAP Theorem in Distributed Data Store

CAP Theorem: In a distributed system (a collection of interconnected nodes that share data.), you can **only have two out of the following three guarantees** across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed.

Eric Brewer is a retired Berkeley Professor and VP of infrastructure at Google

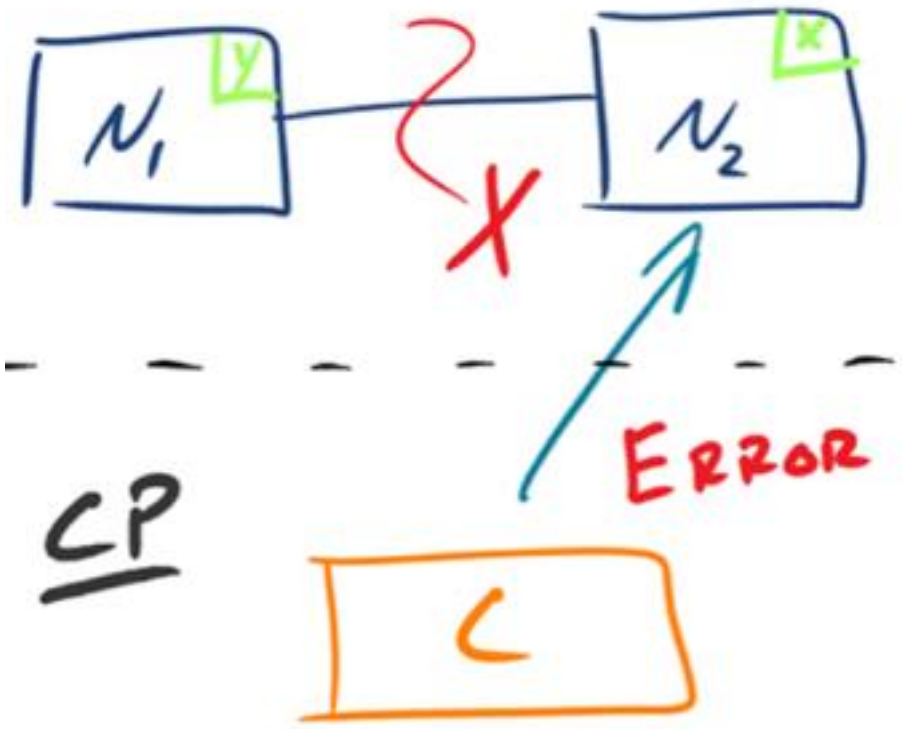
- ***C****onsistency*: Every read receives the most recent write or an error

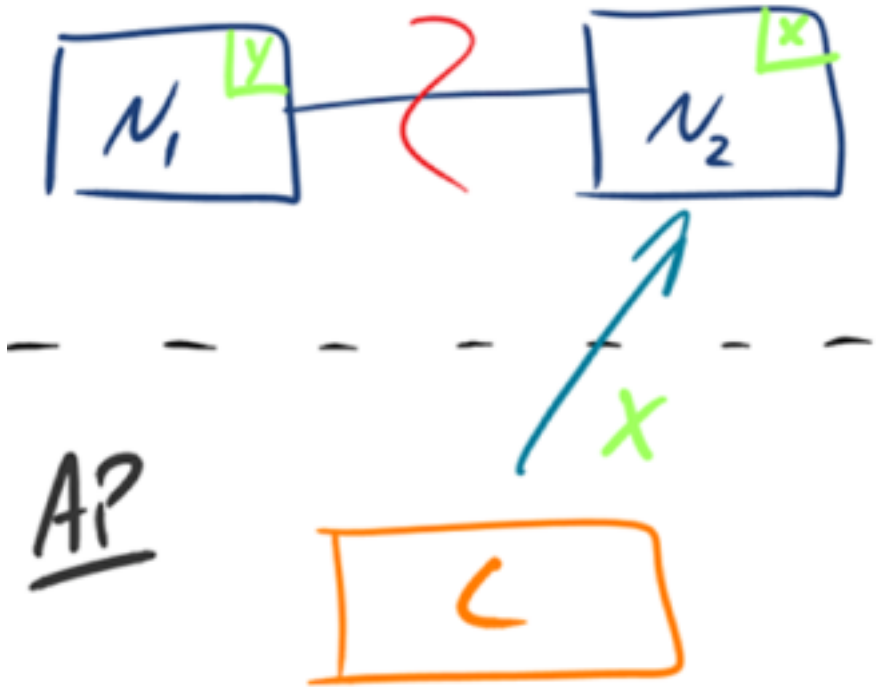
- ***A****vailability*: Every request receives a (non-error) reasonable response, without the guarantee that it contains the most recent write

- ***P****artition tolerance*: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

(Pick CP or AP)

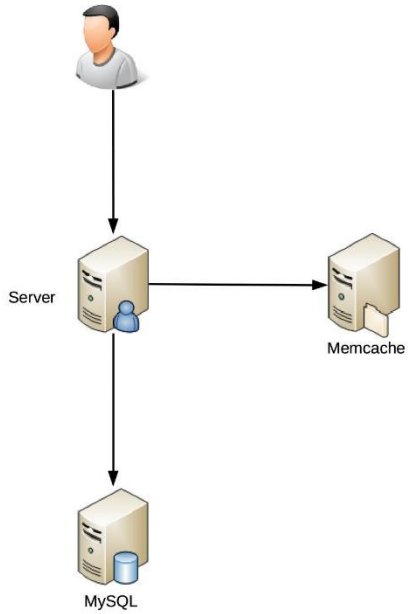
CAP Applications



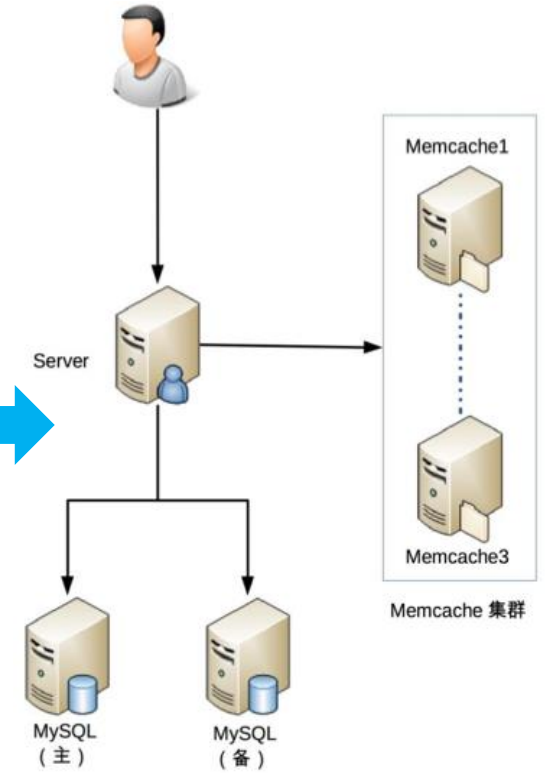


Failure mode and effects analysis)(故障模式与影响分析)

给出初始的架构设计图 → 假设架构中某个部件发生故障 → 分析此故障对系统功能造成的影响 → 根据分析结果，判断架构是否需要进行优化



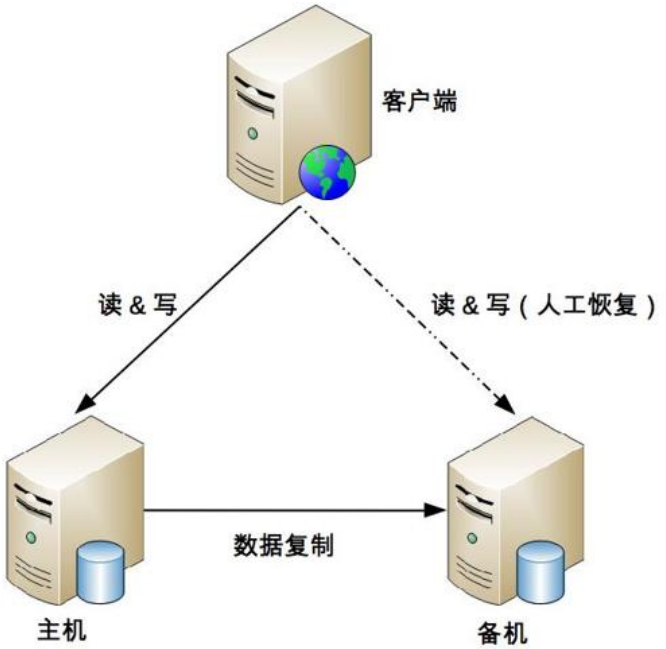
功能点	故障模式	故障影响	严重程度	故障原因	故障概率	风险程度	已有措施	规避措施	解决措施	后续规划
登录	MySQL无法访问	当MC中无缓存时，用户无法登录，预计有60%的用户	高	MySQL服务器断电	中	中	无	无	无	增加备份MySQL
登录	MySQL无法访问	同上	高	Server到MySQL的网络连接中断	中	中	无	无	无	MySQL双网卡连接
登录	MySQL响应时间超过5秒	60%的用户登录时间超过5秒	高	慢查询导致MySQL运行缓慢	高	高	慢查询检测	重启MySQL	无	不需要
登录	MC无法访问	所有用户都到MySQL查询信息，MySQL压力会增大，响应会变慢	低，虽然慢一些，但用户还是能够登录	MC服务器断电	中	低	无	无	无	MC集群
注册	MySQL无法访问	用户无法注册	低，因为新注册的用户每天大约只有100个	MySQL服务器断电	中	低	无	无	无	无，因为即使增加备份机器，也无法作为主机写入
注册	MC无法访问	无影响，用户注册流程不操作MC	无	MC服务器断电	中	低	无	无	无	不需要



高可用存储-双机: 如何复制，如何应对延迟，如何应对复制中断?

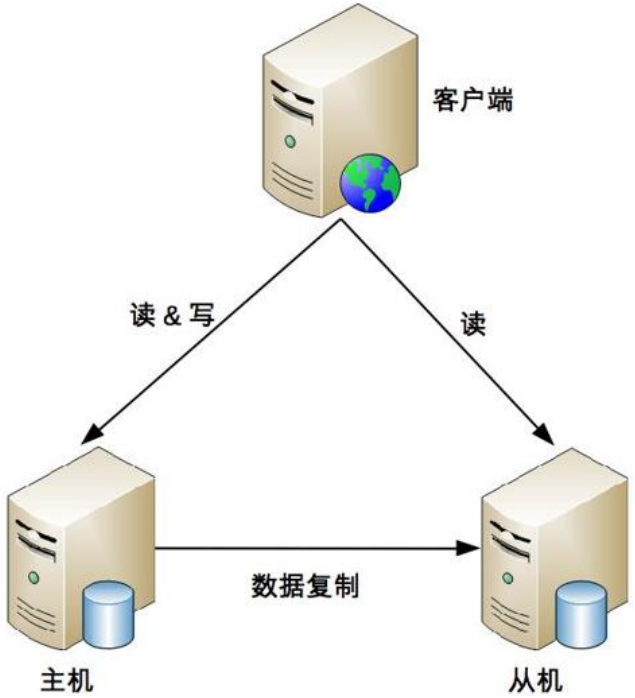
主备复制：数据变更频率低

应用场景：学生管理系统、员工管理系统、假期管理系统等



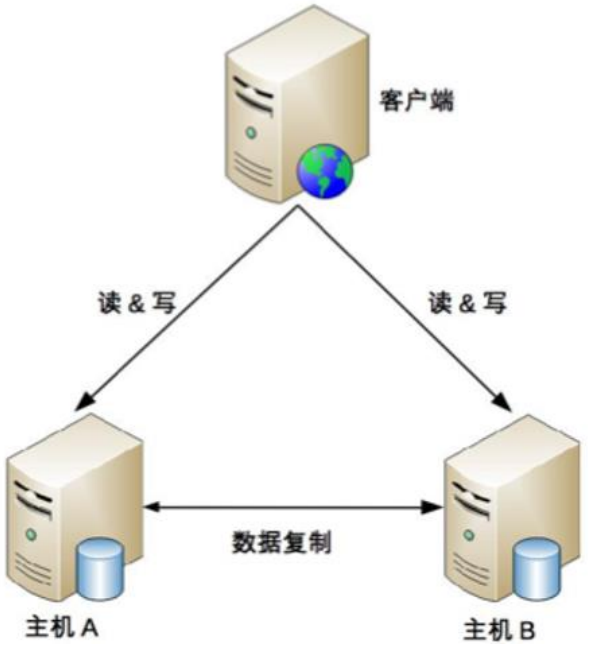
主从复制: 适合写少读多 （1：10，1：100） 的业务

应用场景：论坛、新闻网站等



主主复制：适合于那些临时性、可丢失、可覆盖的数据场景

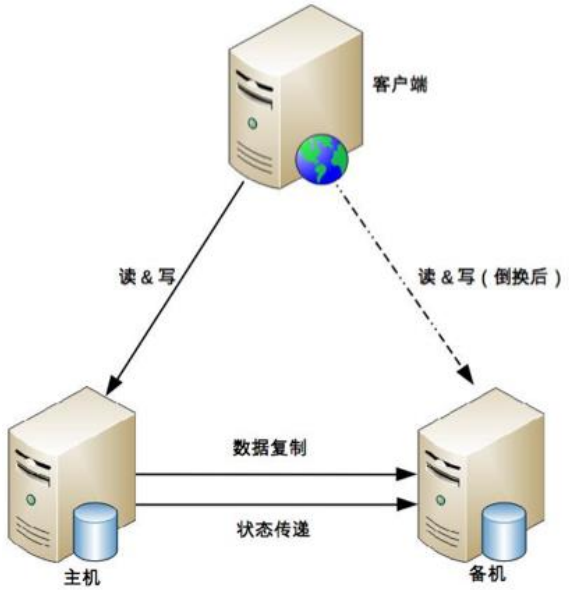
应用场景： 如论坛的草稿数据（可以丢失）



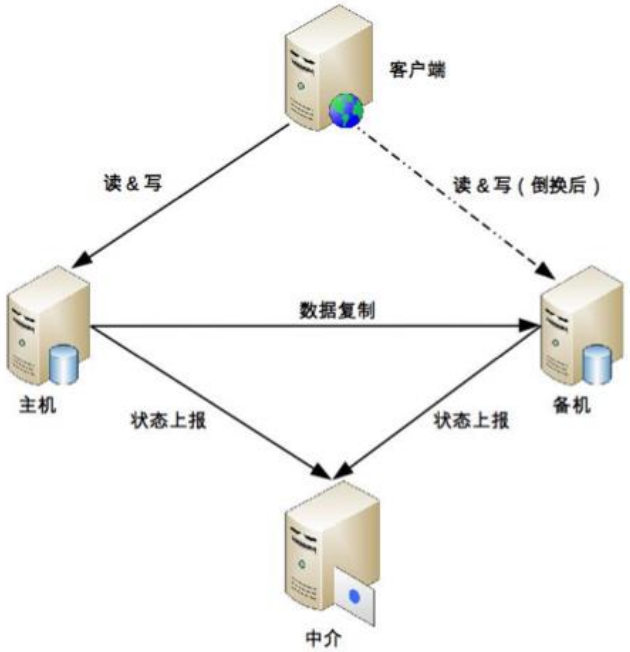
主备切换:

状态判断：状态传递渠道、状态监测 切换决策：切换时机、切换策略、自动程度

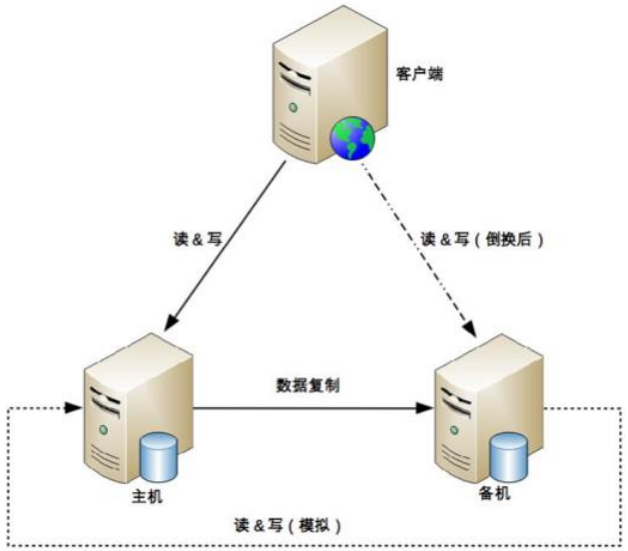
互联式



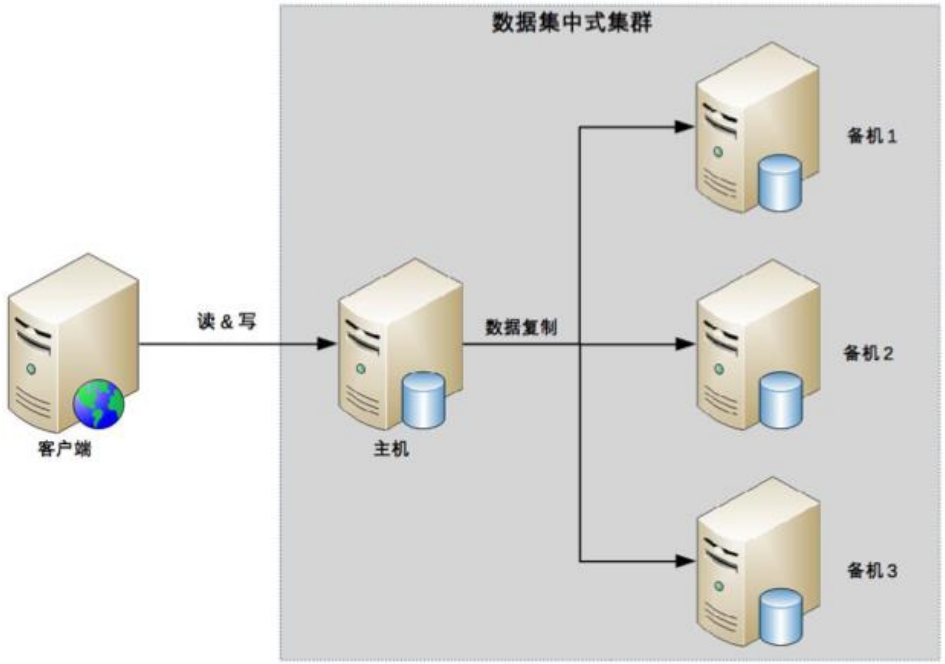
中介式



模拟式



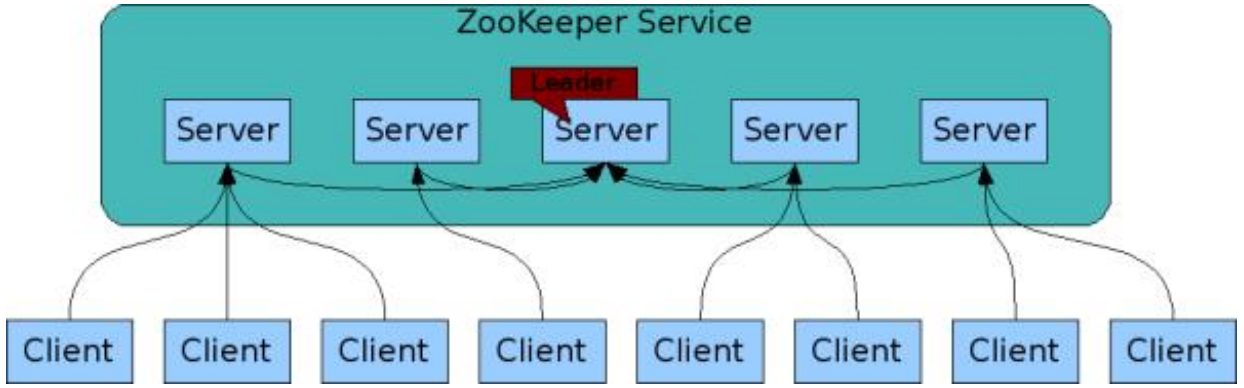
高可用存储架构：数据集中式集群



目前开源的数据集中集群以 ZooKeeper 为典型：

- 处理备机对主机状态判断不一致的问题
- 主机故障后，如何决定新的主机

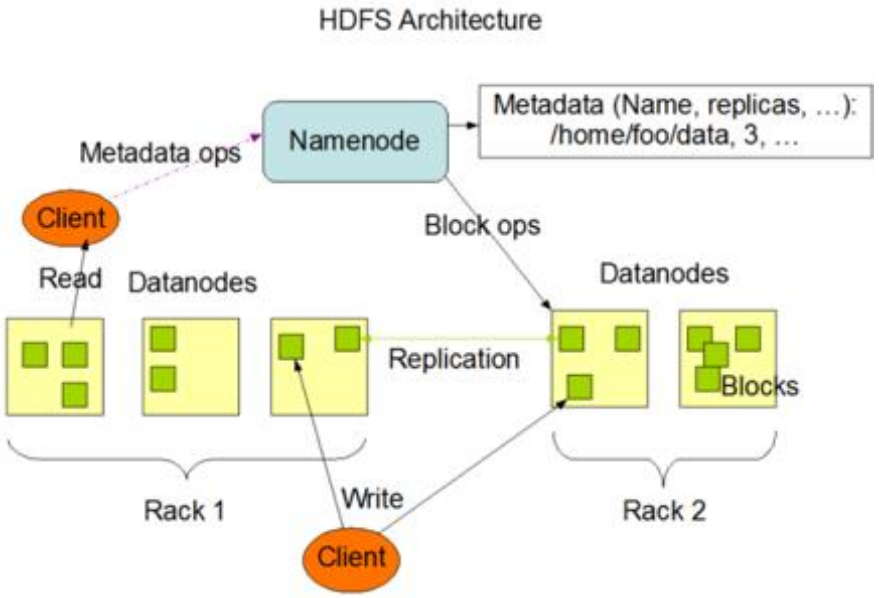
ZooKeeper 集群，一般推荐 5 台机器左右，数据量是单台服务器就能够支撑



高可用存储架构：数据分散集群

可伸缩性良好，适合业务数据量巨大、集群机器（上百台甚至上千台服务器）数量庞大的业务场景。

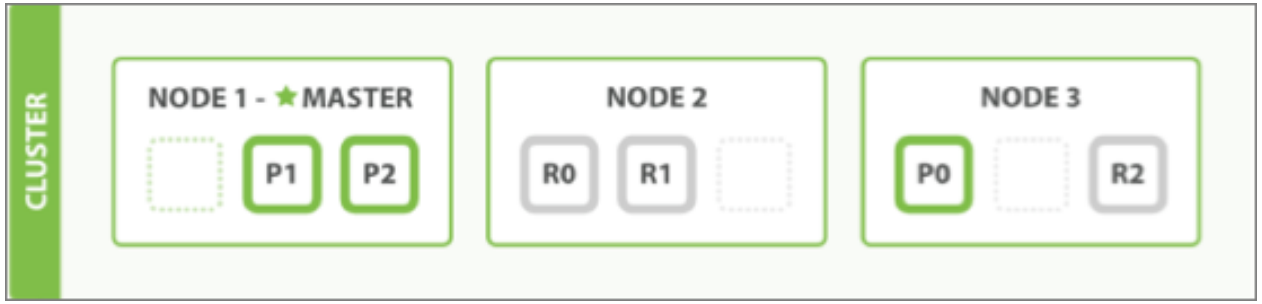
独立的服务器： 如Hadoop 的Namenode就是负责数据分区的分配



非独立服务器

Elastic search 集群通过选举一台服务器来做主节点，负责数据分区的分配

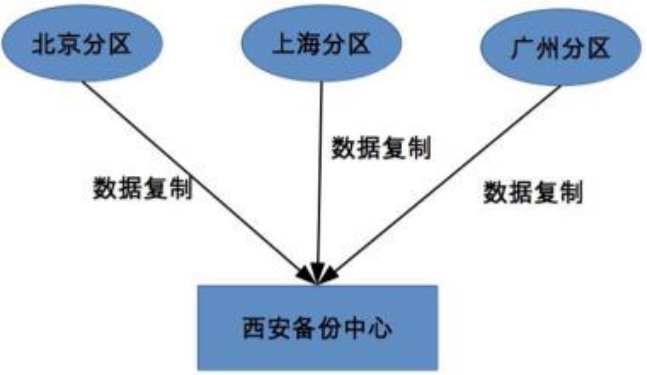
In Elastic Search, the master node is responsible for **lightweight cluster-wide actions** such as creating or deleting an index, tracking which nodes are part of the cluster, and deciding which shards to allocate to which nodes. node.



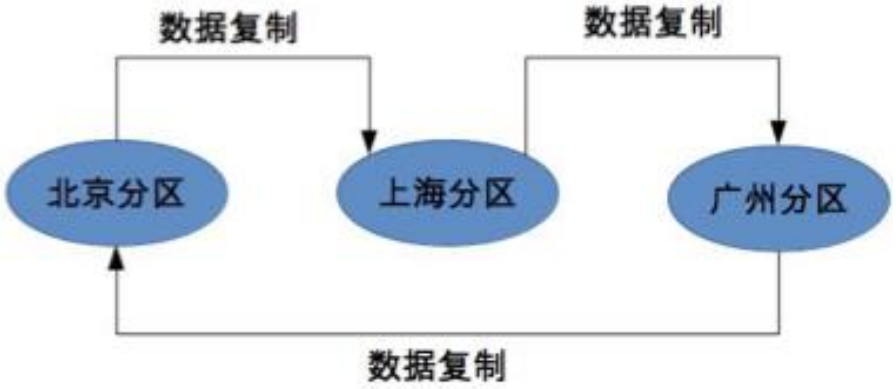
高可用存储：数据分区

数据分区指将数据按照一定的规则进行分区，不同分区分布在不同的地理位置上，每个分区存储一部分数据，通过这种方式来规避地理级别的故障所造成的巨大影响

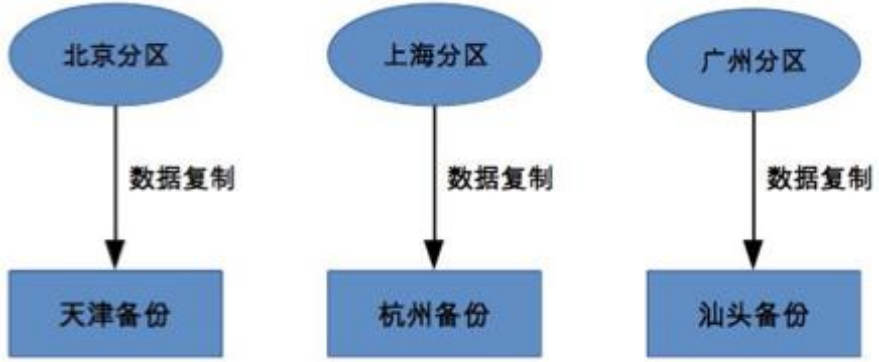
集中式



互备式

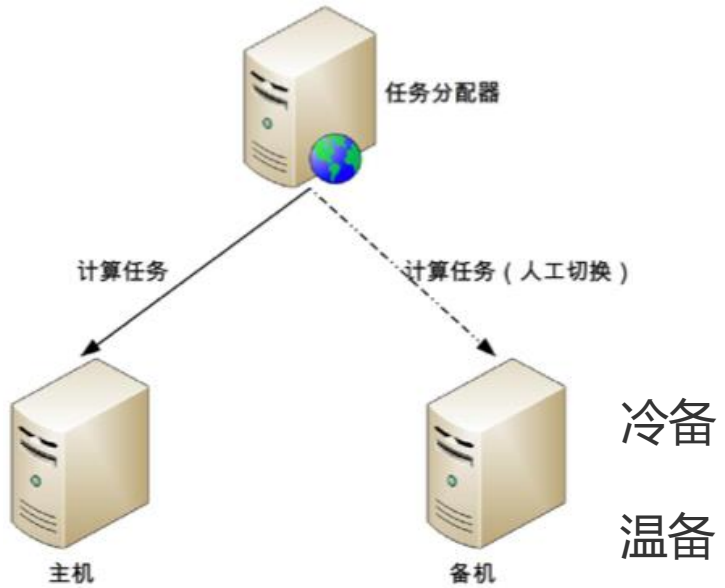


独立式



计算高可用架构：主备、主从

主备

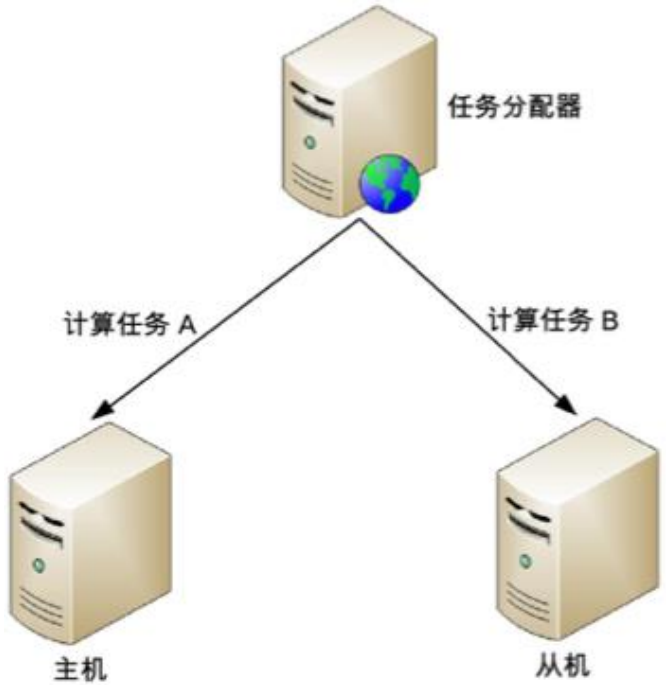


冷备

温备

适用场景：内部管理系统、后台管理系统这类使用人数不多、使用频率不高的业务，不太适合在线的业务

主从

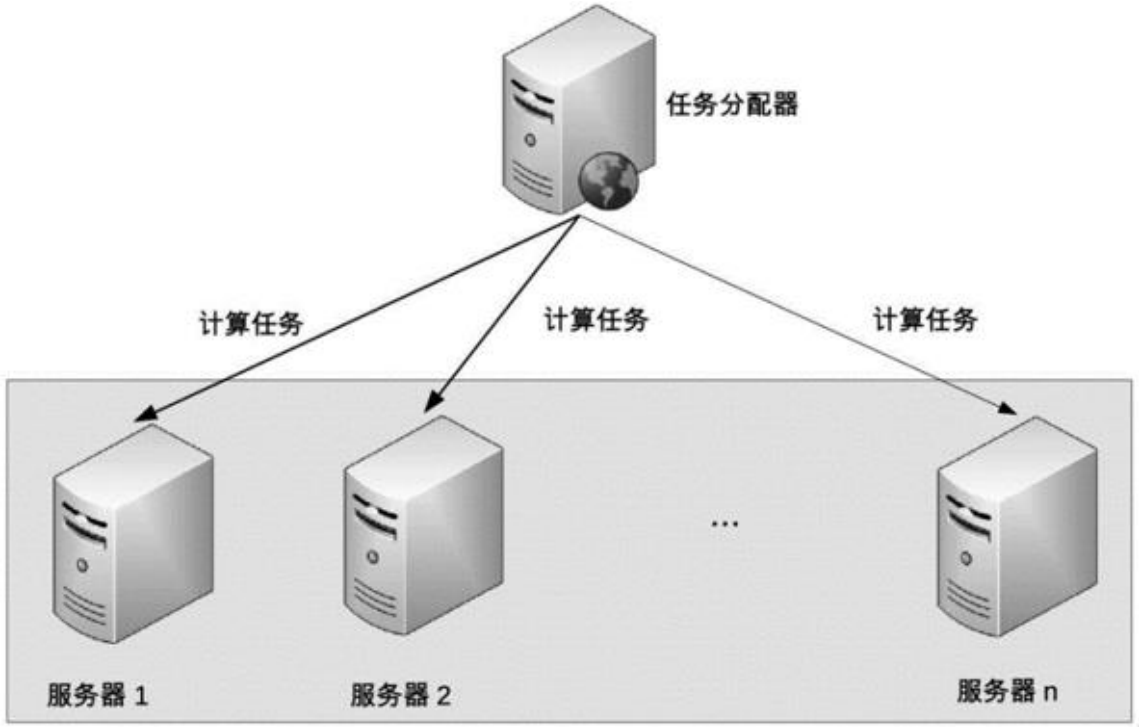


仍然是手工切换

计算高可用集群

对称集群：负载均衡集群

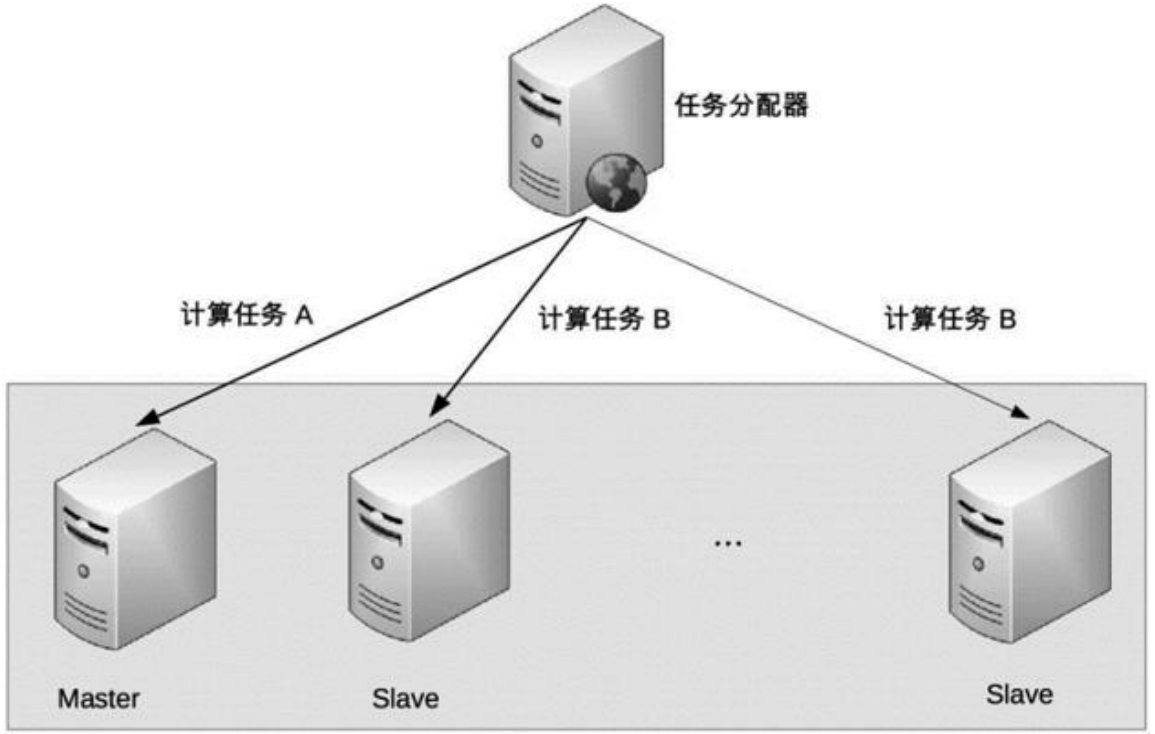
任务分配策略：轮询、随机...
服务器状态：宕机、断网...
执行状态：任务卡死、执行时间过长...



非对称集群

不同角色的服务器承担不同的职责

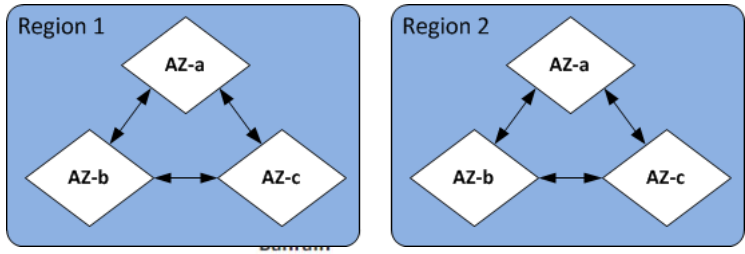
- 任务分配策略：将任务划分为不同类型并分配给不同角色
- 角色分配策略：leader选举（ZAB、Raft等）



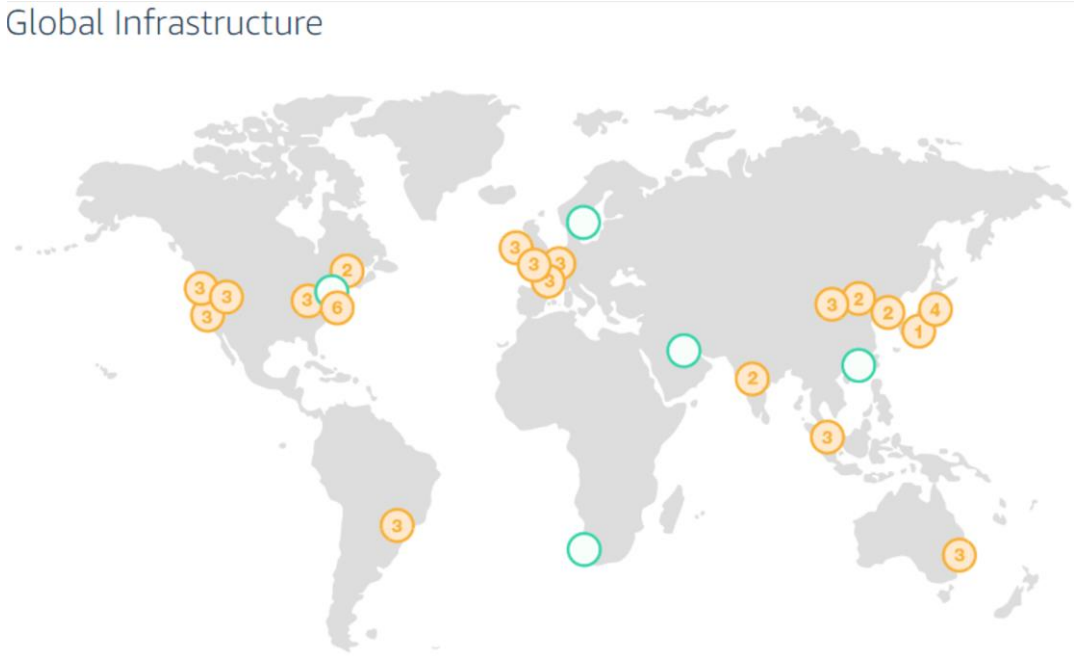
业务高可用的保障：异地多活架构

正常情况下，用户无论访问哪一个地点的业务系统，都能够得到正确的业务服务。某个地方业务异常的时候，用户访问其他地方正常的业务系统，能够得到正确的业务服务。　　需要处理：业务分级、数据分类、数据同步、异常处理

同城异区　　跨城异地　　跨国异地



适用范围举例：支付宝、微信、滴滴



Region & Number of Availability Zones

US East
N. Virginia (6),
Ohio (3)

US West
N. California (3),
Oregon (3)

Asia Pacific
Mumbai (2),
Seoul (2),
Singapore (3),
Sydney (3),
Tokyo (4),
Osaka-Local (1)[1]

Canada
Central (2)

China
Beijing (2),
Ningxia (3)

Europe
Frankfurt (3),
Ireland (3),
London (3),
Paris (3)

South America
São Paulo (3)

GovCloud (US-West) (3)

AWS Regions are large and widely dispersed into separate geographic locations.
Availability Zones are distinct locations within an AWS Region that are engineered to be isolated from failures in other Availability Zones. They provide inexpensive, low-latency network connectivity to other Availability Zones in the same AWS Region.

Architecture Patterns for High Scalability

High Scalability

- The ability to accommodate any growth in the future, be it expected or not (e.g. the Internet)
  - More users
  - Respond faster (performance vs. scalability)
  - Continue delivering the expected Quality of Service under high load
- Openness
  - New components can be integrated with existing components
- Heterogeneity
  - Heterogeneity of hardware platform, operating systems, networks, programming languages can be easily supported

可扩展的基本思想

- 可扩展性：通过修改和扩展，不断地让软件系统具备更多的功能和特性，满足新的需求或者顺应技术发展的趋势
- 可扩展性架构设计，背后的基本思想都可以总结为一个字：拆！
- 拆：就是将原本大一统的系统拆分成多个规模小的部分，扩展时只修改其中一部分即可，无须整个系统到处都改，通过这种方式来减少改动范围，降低改动风险。
  - 面向流程拆分：将整个业务流程拆分为几个阶段，每个阶段作为一部分。
  - 面向服务拆分：将系统提供的服务拆分，每个服务作为一部分。
  - 面向功能拆分：将系统提供的功能拆分，每个功能作为一部分。
- 理解这三种思路的关键就在于如何理解“流程”“服务”“功能”三者的联系和区别。从范围上来看，从大到小依次为：流程 > 服务 > 功能

面向流程拆分

展示层：负责用户页面设计，不同业务有不同的页面。例如，登录页面、注册页面、信息管理页面、安全设置页面等。

业务层：负责具体业务逻辑的处理。例如，登录、注册、信息管理、修改密码等业务。

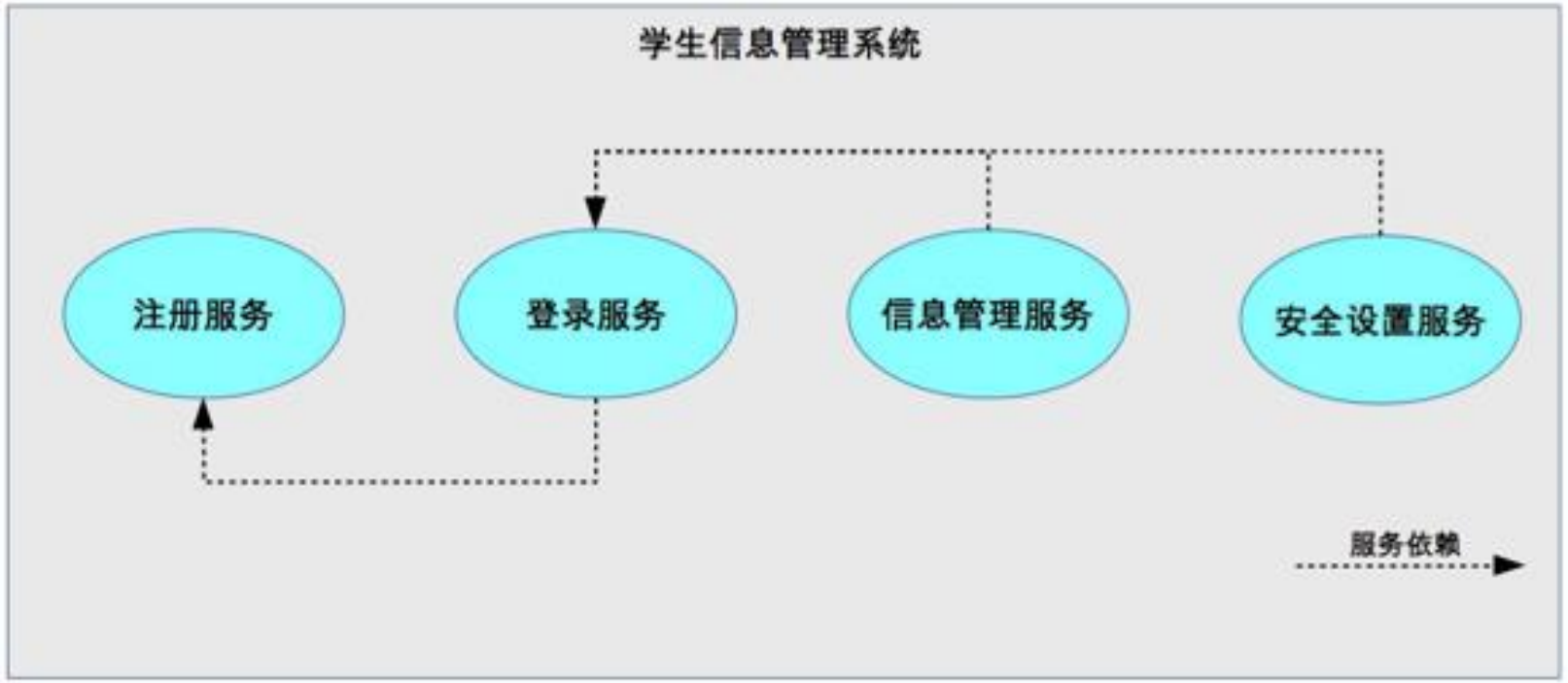
数据层：负责完成数据访问。例如，增删改查数据库中的数据、记录事件到日志文件等。

存储层：负责数据的存储。例如，关系型数据库 MySQL、缓存系统 Memcache 等。

扩展时大部分情况只需要修改某一层，少部分情况可能修改关联的两层，不会出现所有层都同时要修改。

面向服务拆分

面向服务拆分将系统拆分为注册、登录、信息管理、安全设置等服务



对某个服务扩展，或者要增加新的服务时，只需要扩展相关服务即可，无须修改所有的服务。

面向功能拆分

每个服务都可以拆分为更多细粒度的功能：

注册服务：提供多种方式进行注册，包括手机号注册、身份证注册、学生邮箱注册三个功能。

登录服务：包括手机号登录、身份证登录、邮箱登录三个功能。

信息管理服务：包括基本信息管理、课程信息管理、成绩信息管理等功能。

安全设置服务：包括修改密码、安全手机、找回密码等功能。

对某个功能扩展，或者要增加新的功能时，只需要扩展相关功能即可，无须修改所有的功能。

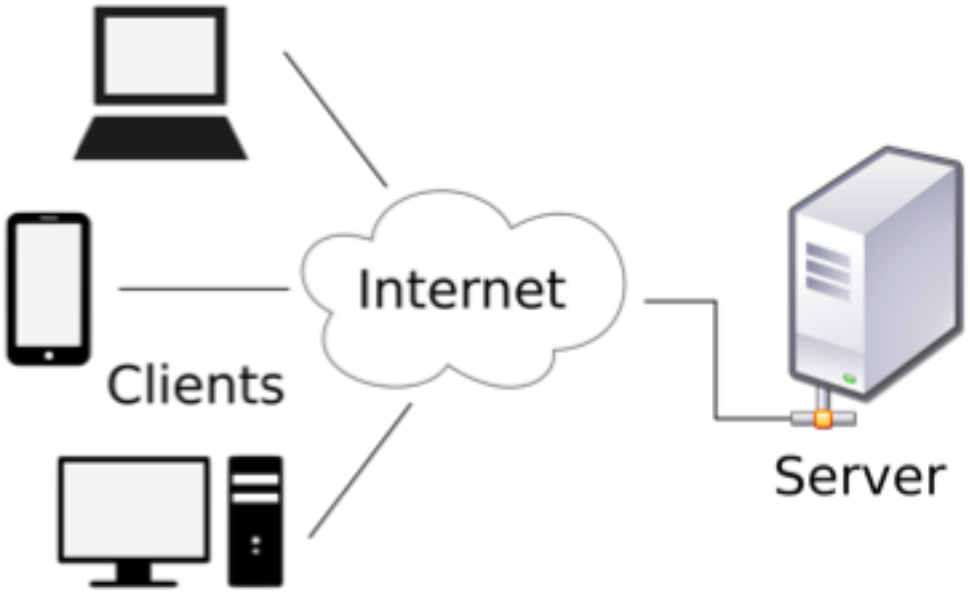
不同拆分方式对应的可扩展系统架构

- 不同拆分方式对应的可扩展系统架构
  - 面向流程拆分：分层架构。
  - 面向服务拆分：SOA、微服务
  - 面向功能拆分：微内核或插件式架构

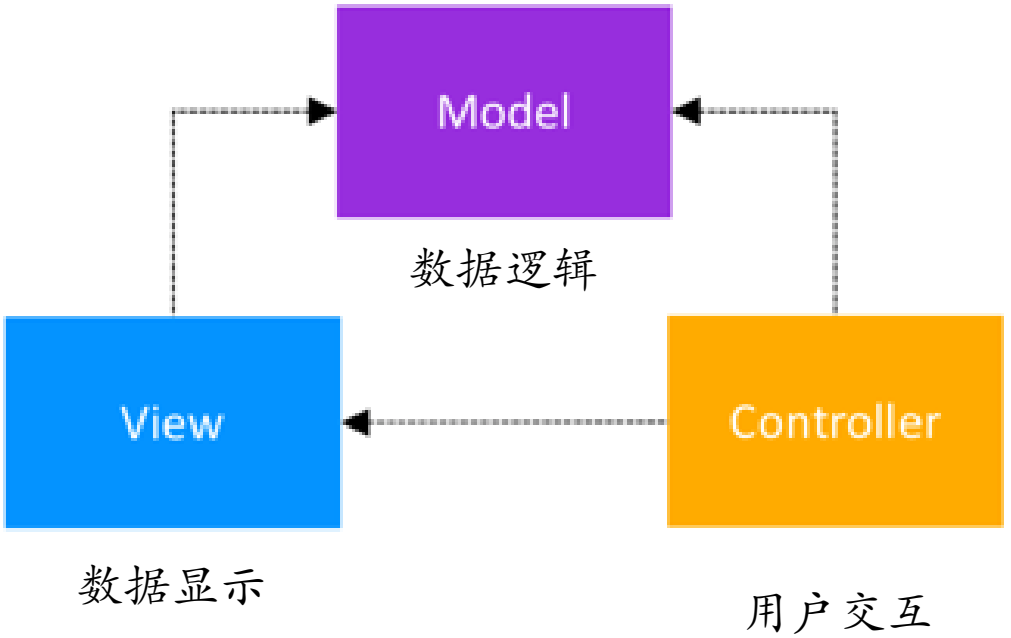
- 可以组合使用，以学生管理系统为例：
  - 整体系统采用面向服务拆分中的“微服务”架构，拆分为“注册服务”“登录服务”“信息管理服务”“安全服务”，每个服务是一个独立运行的子系统
  - 其中的“注册服务”子系统本身又是采用面向流程拆分的分层架构。
  - “登录服务”子系统采用的是面向功能拆分的“微内核”架构

面向流程拆分-分层架构： CS 、 MVC

CS: 划分的对象是**整个业务系统**，划分的维度是用户交互，即将和用户交互的部分独立为一层，支撑用户交互的后台作为另外一层

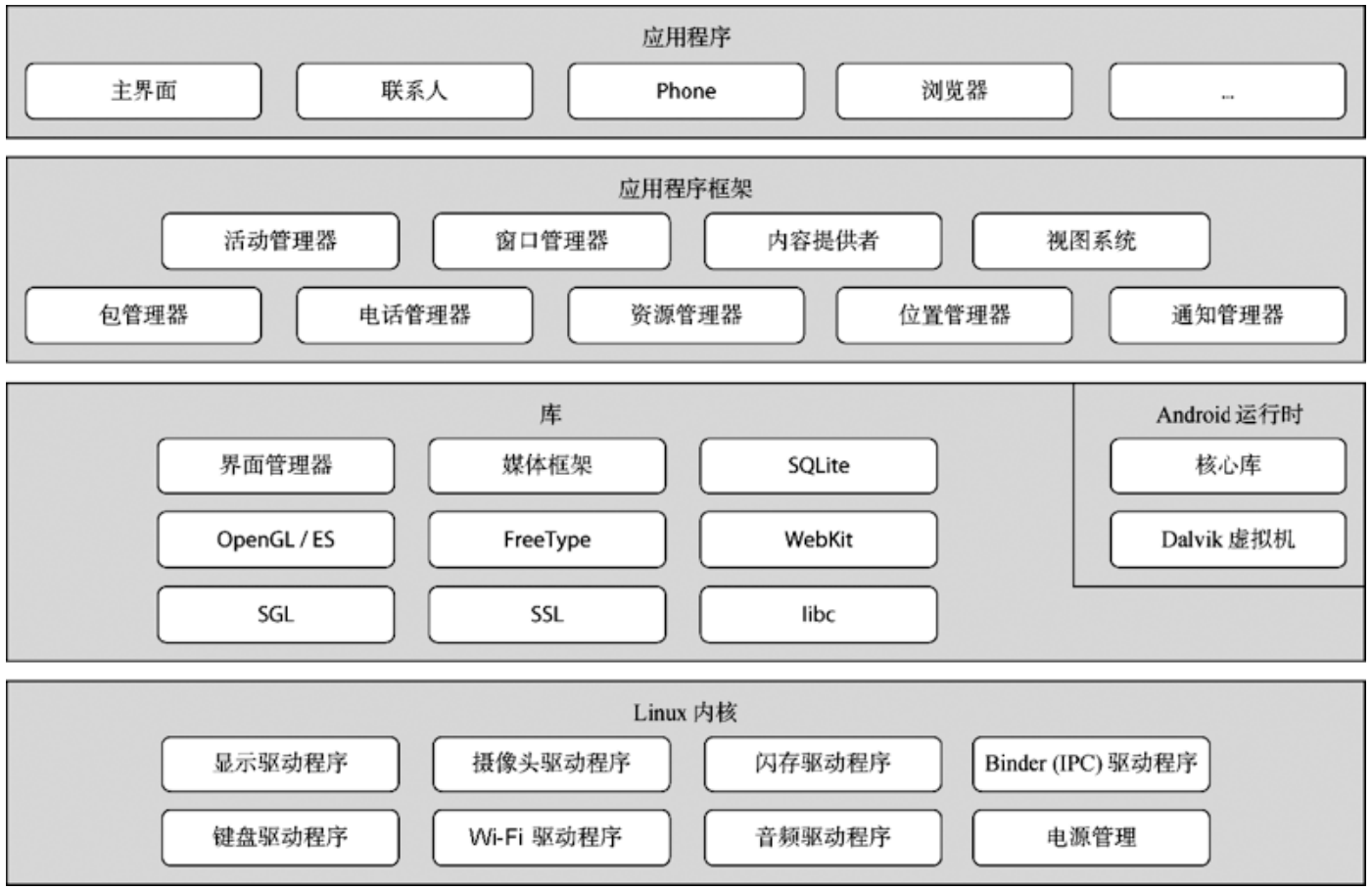


MVC: 划分的对象是**单个业务子系统**，划分的维度是职责，将不同的职责划分到独立层，但各层的依赖关系比较灵活



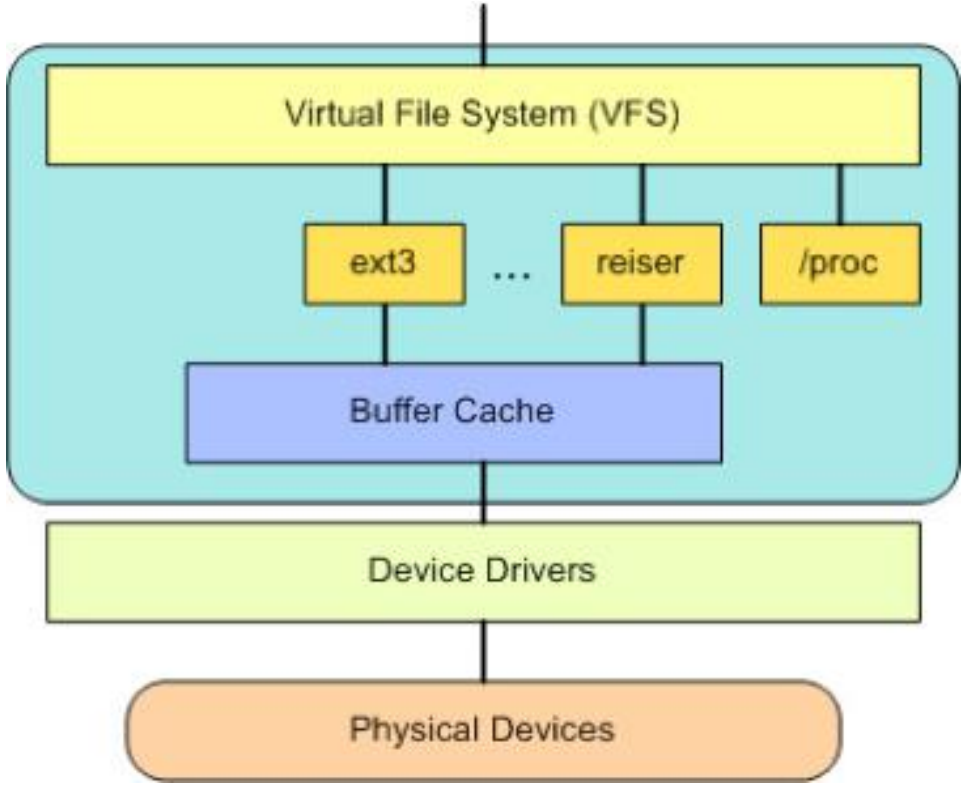
面向流程拆分-分层架构：
逻辑分层架构（按职责自顶向下依赖）

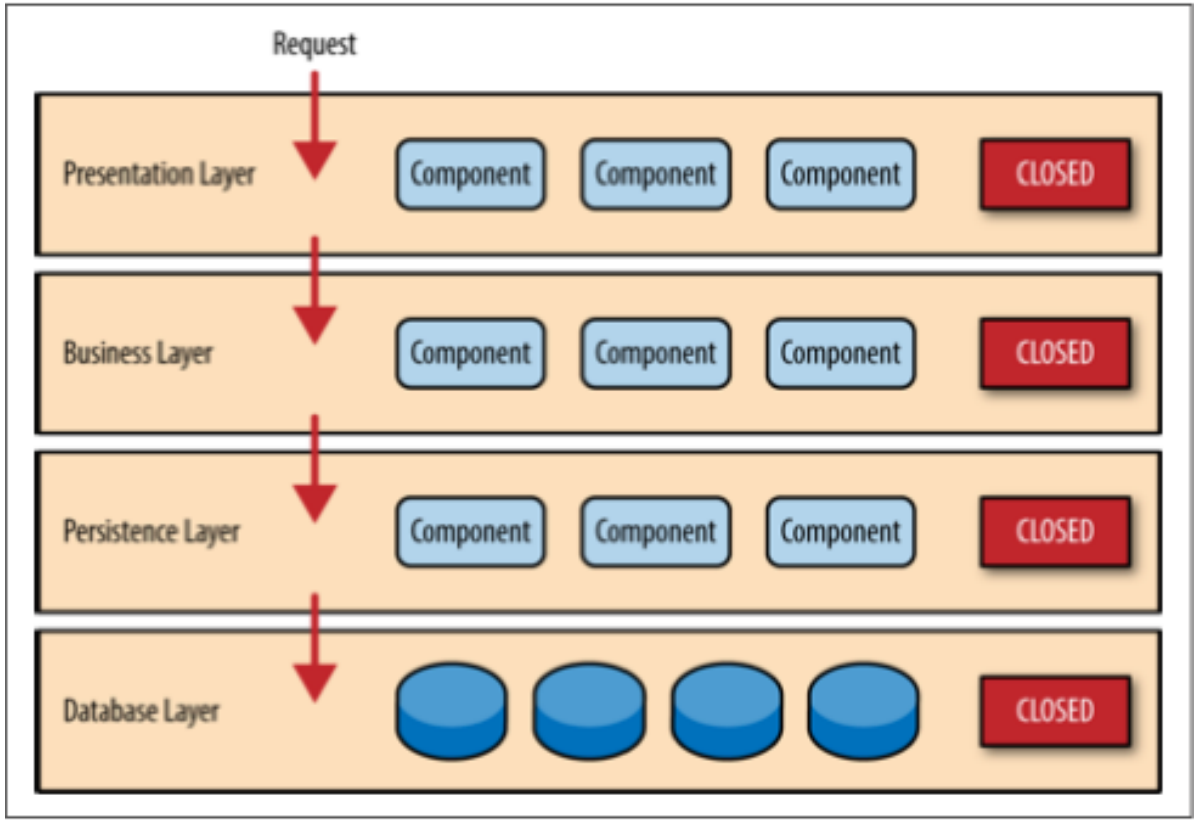
Android 分层架构



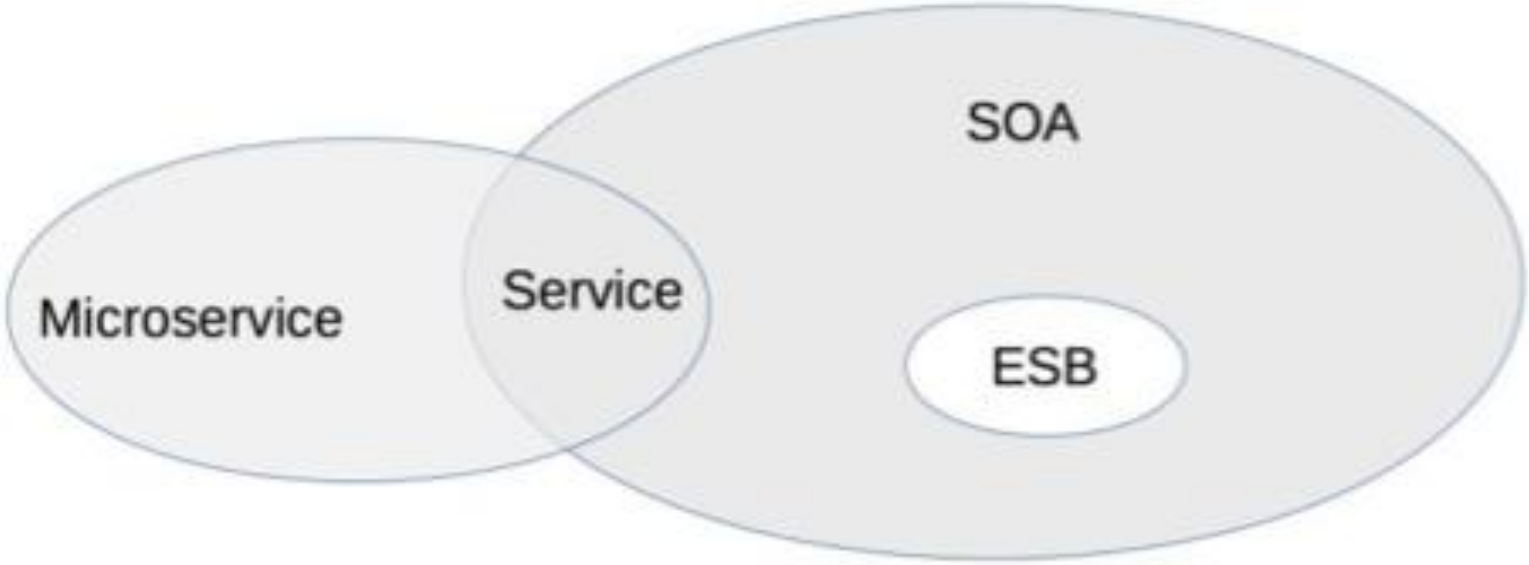
面向流程拆分-分层架构特点

1. 保证各层之间的差异足够清晰，边界足够明显，让人看到架构图后就能看懂整个架构
2. 隔离关注点（separation of concerns），即每个层中的组件只会处理本层的逻辑， 从而能够较好地支撑系统扩展
3. 层层传递，相邻层依赖，一般不跨层跳跃， 从而降低整体系统复杂度





面向服务拆分： SOA 与微服务对比

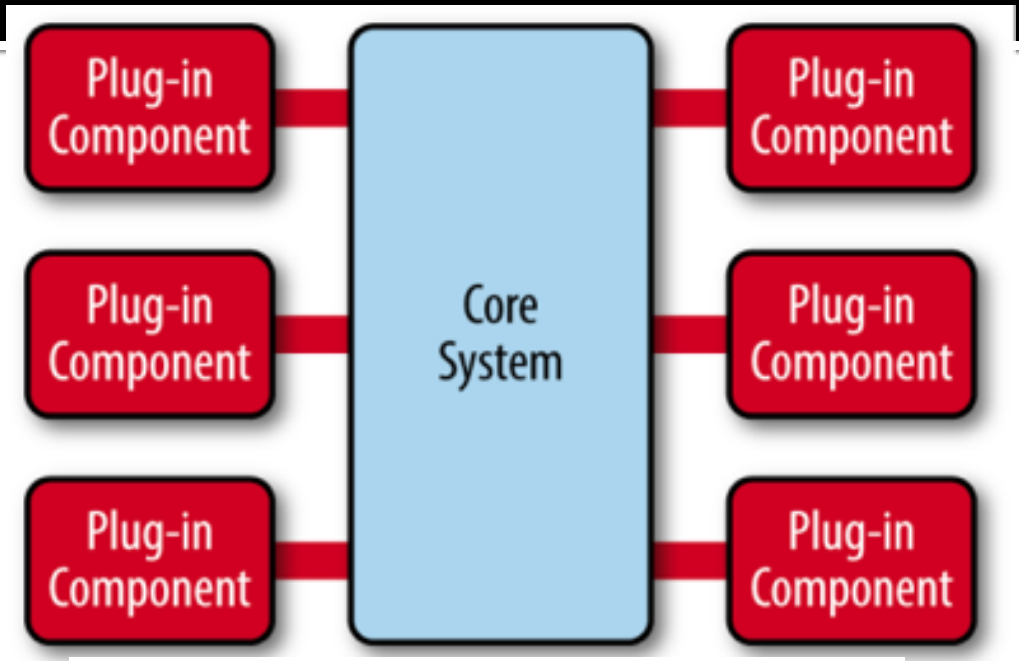


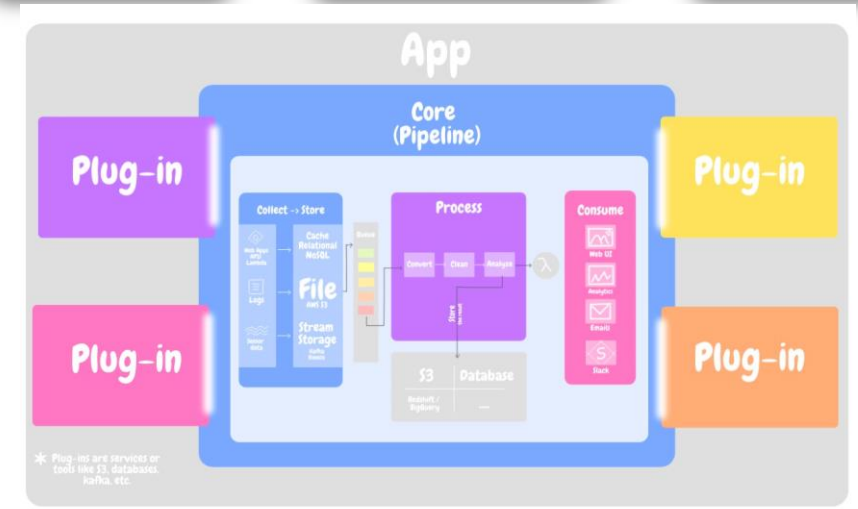
对比维度	SOA	微服务
服务粒度	粗	细
服务通信	重量级，ESB	轻量级，例如，HTTP RESTful
服务交付	慢	快
应用场景	企业级	互联网

面向功能拆分-微内核架构（插件式架构）

微内核架构（Microkernel Architecture），也被称为插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于下载安装的的应用。

- Browser
- Eclipse， VSCode 等IDE 软件
- UNIX 这类操作系统
- 淘宝 App 客户端软件
- 保险公司的保险核算逻辑系统，不同的保险品种可以将逻辑封装成插件





Outline

- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

（一）Large-Scale Ecommerce website Architecture

大型网站软件系统特点

- 高并发，大流量
- 高可用
- 海量数据
- 用户分布广泛，网络情况复杂
- 安全环境恶劣
- 需求快速变更，发布频繁
- 渐进式发展

大型网站架构演化发展历程

初始阶段

应用服务和数据服务分离

使用缓存改善网站性能

使用应用服务器集群改善网站的并发处理能力

数据库读写分离

使用反向代理和CDN加速网站响应

使用分布式文件系统和分布式数据库系统

使用NoSQL和搜索引擎

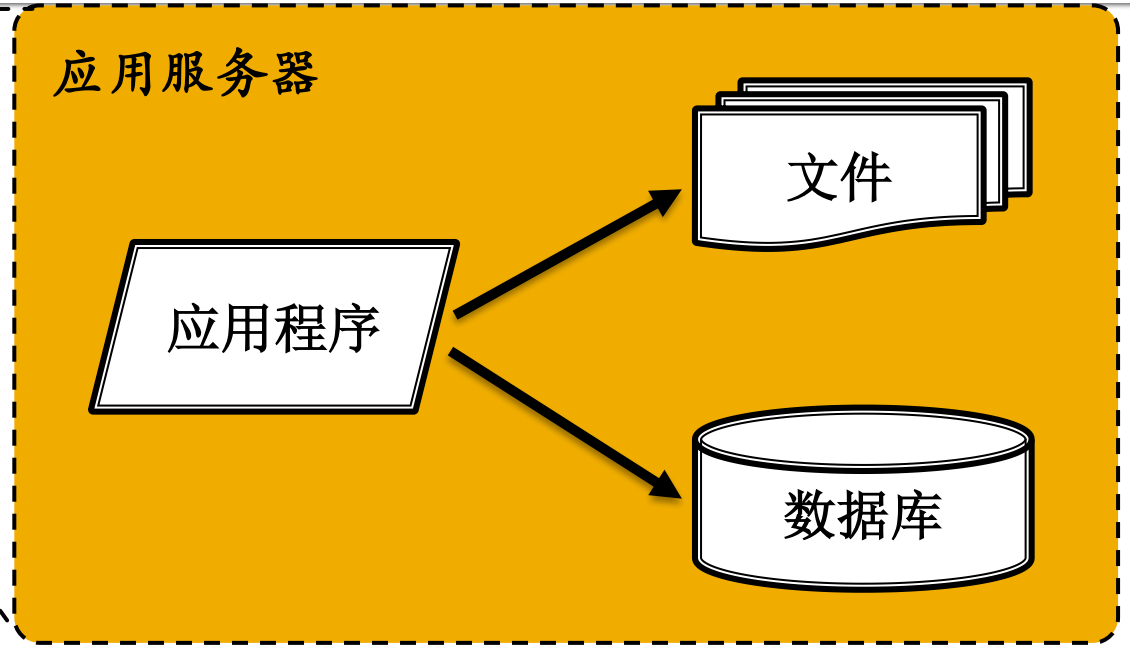
业务拆分

分布式服务

- 大型网站架构演化的价值观
  - 大型网站架构技术的核心价值是随网站所需灵活应对
  - 驱动大型网站技术发展的主要力量是网站的业务发展
- 网站架构设计误区
  - 一味追随大公司的解决方案
  - 为了技术而技术
  - 企图用技术解决所有的问题

大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务

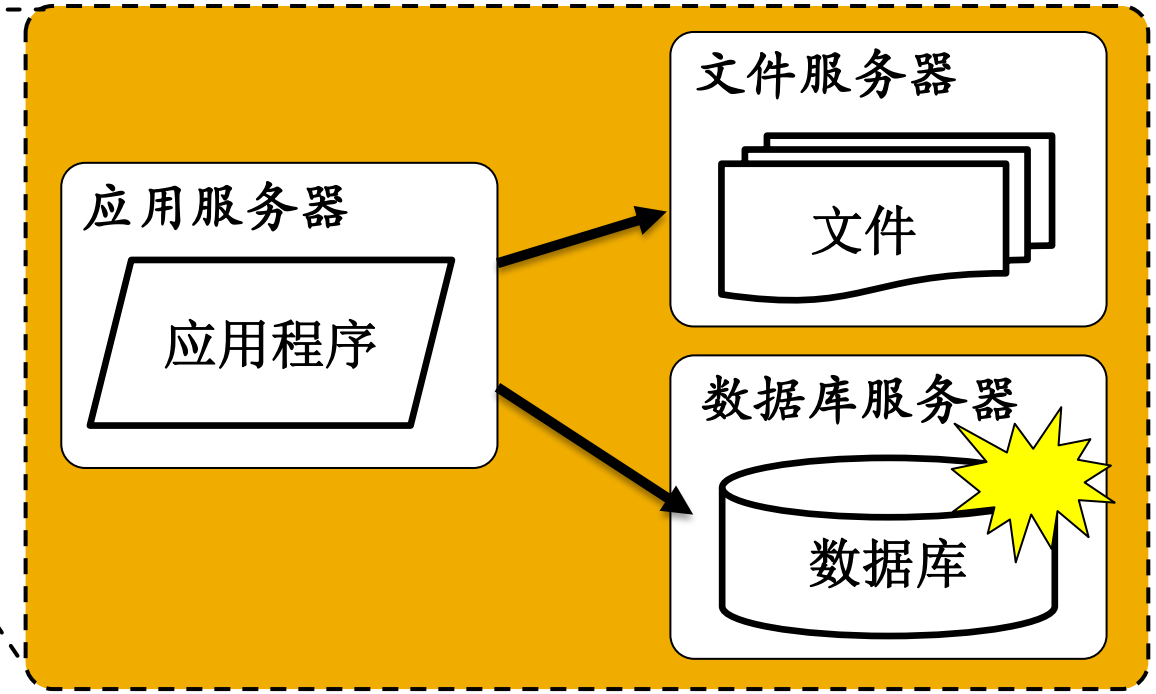


应用程序、数据库、文件等所有的资源都在一台服务器上。

典型的配置如：
服务器操作系统采用linux，应用程序采用PHP开发，部署在Apache上，数据库使用MySQL，汇集各种免费开源软件及一台廉价服务器。

大型网站架构演化发展历程

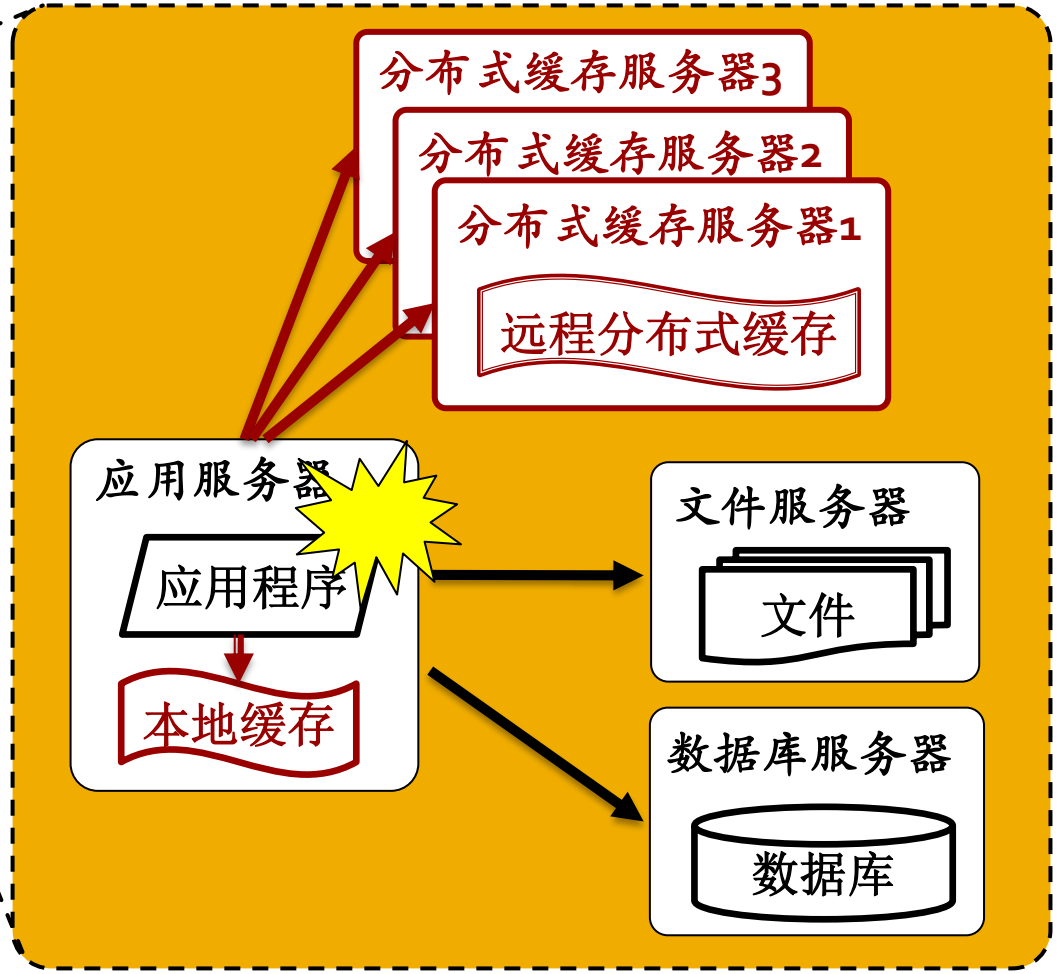
- 初始阶段
- **应用服务和数据服务分离**
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务



不同特性的服务器承担不同的服务角色，网站的并发处理能力和数据存储空间都得到了很大改善。

大型网站架构演化发展历程

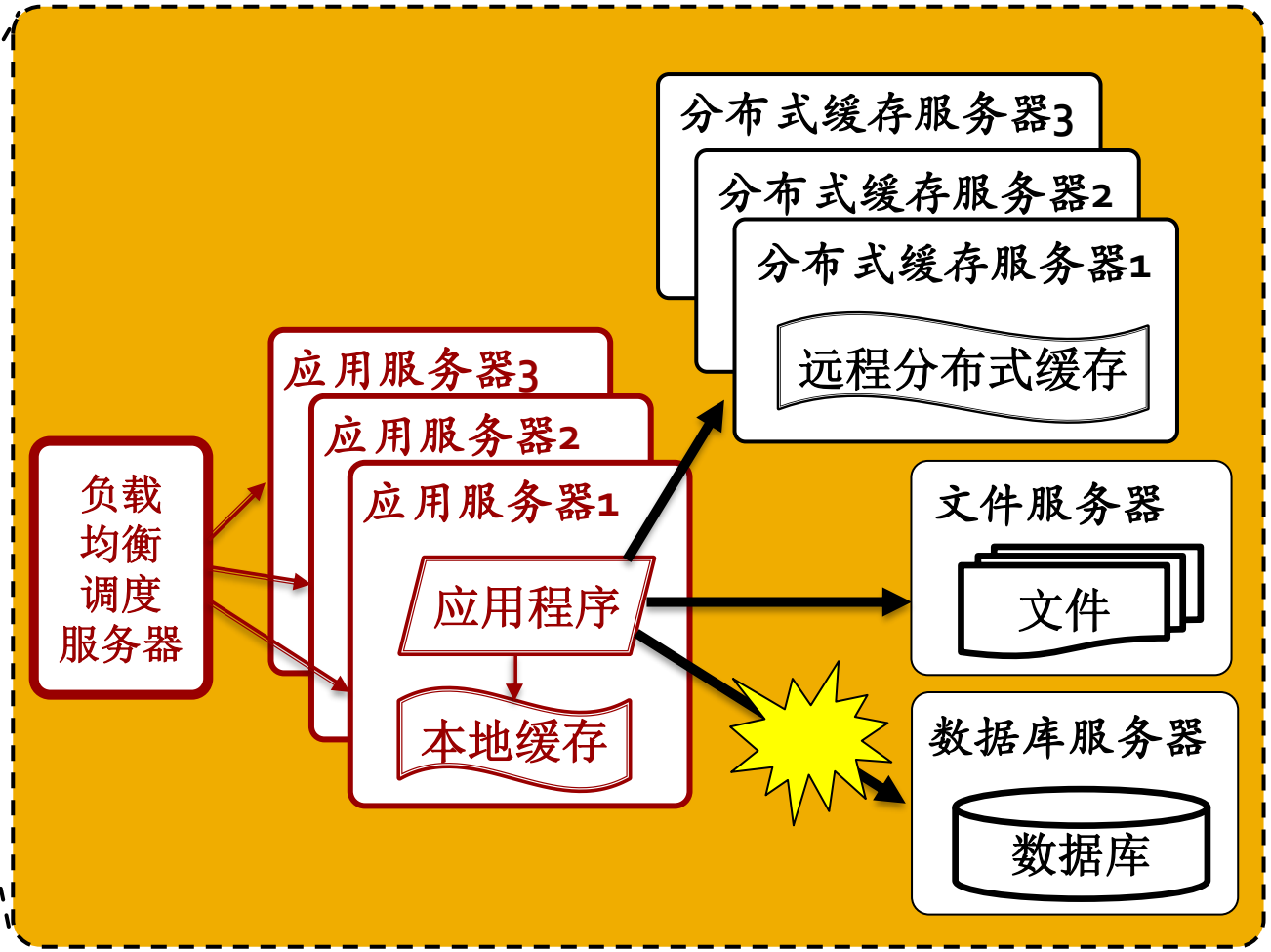
- 初始阶段
- 应用服务和数据服务分离
- **使用缓存改善网站性能**
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务



使用缓存之后，数据访问压力得到有效缓解，但是单一应用服务器能够处理的请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈。

大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- **使用应用服务器集群改善网站的并发处理能力**
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务

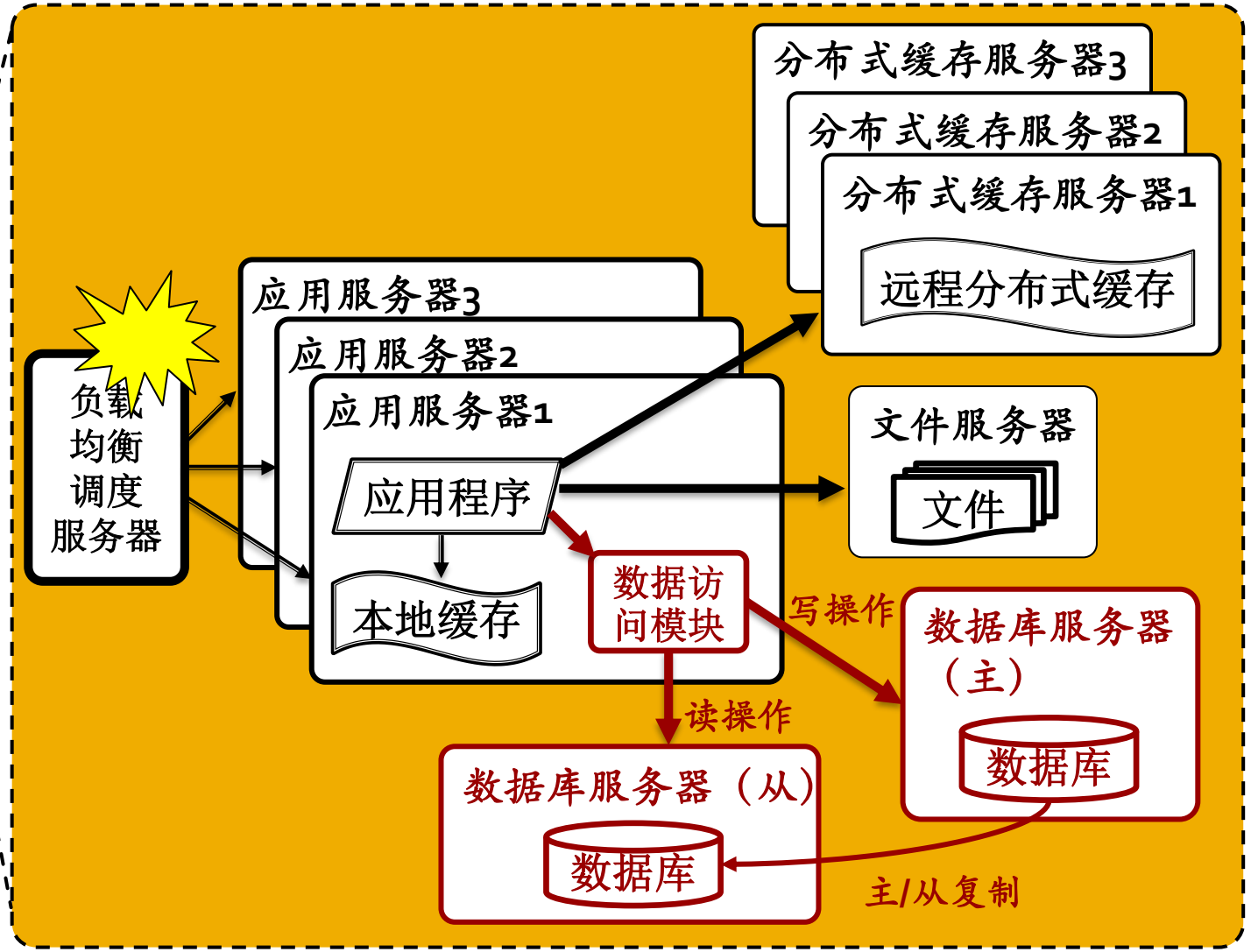


通过负载均衡调度服务器，可将来自用户浏览器的访问请求分发到应用服务器集群中的任何一台服务器上，服务器的负载压力不再成为整个网站的瓶颈。

部分数据库读操作（缓存访问不命中、缓存过期）和全部的写操作需要访问数据库。当负载压力过高时，数据库成为瓶颈。

大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- **数据库读写分离**
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务



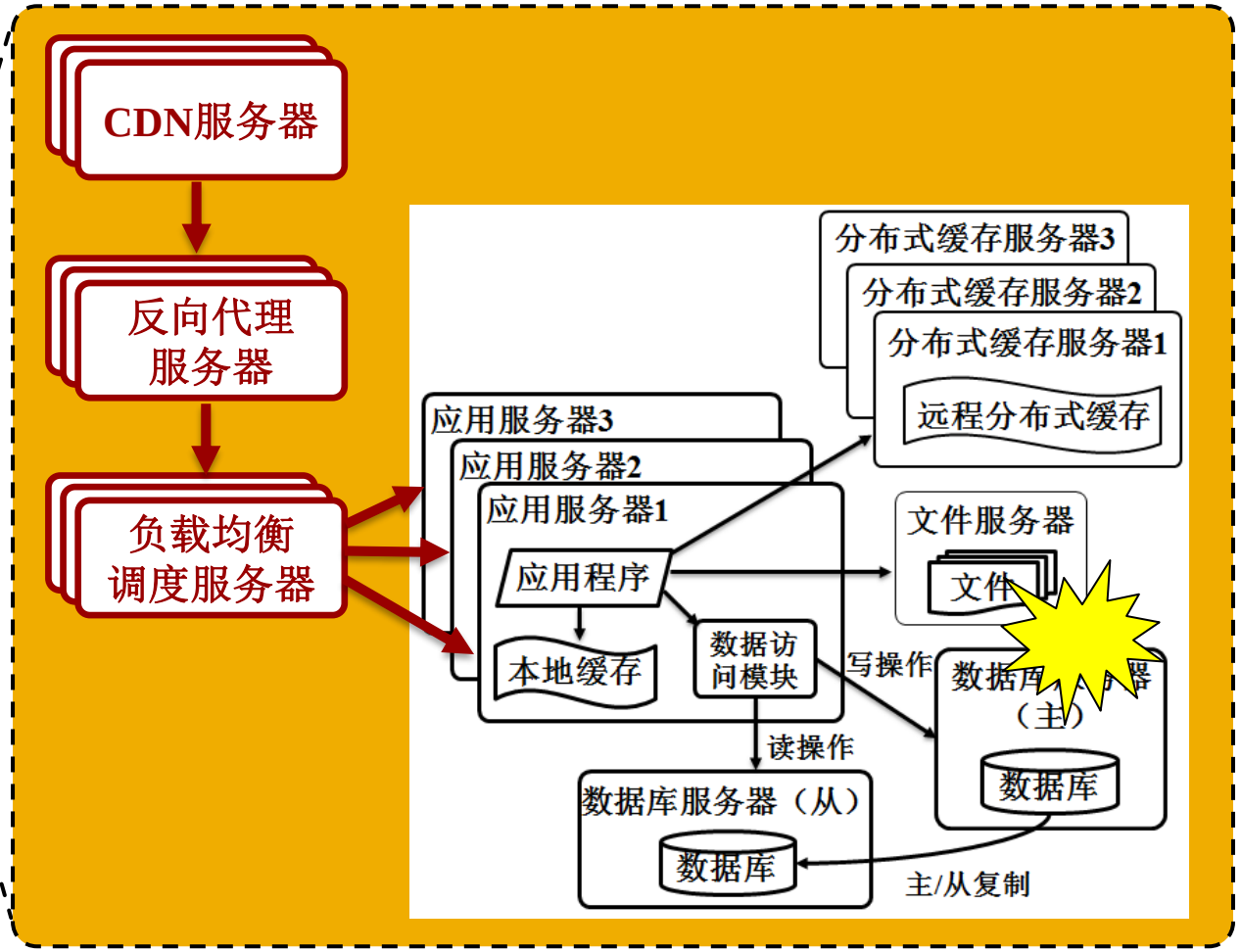
应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制机制将数据更新同步到从数据库。

应用服务器在读数据的时候，直接通过从数据库获得数据。

大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- **使用反向代理和CDN加速网站响应**
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务

CDN和反向代理的基本原理都是缓存。加快用户访问速度，减轻后端服务器的负载压力。

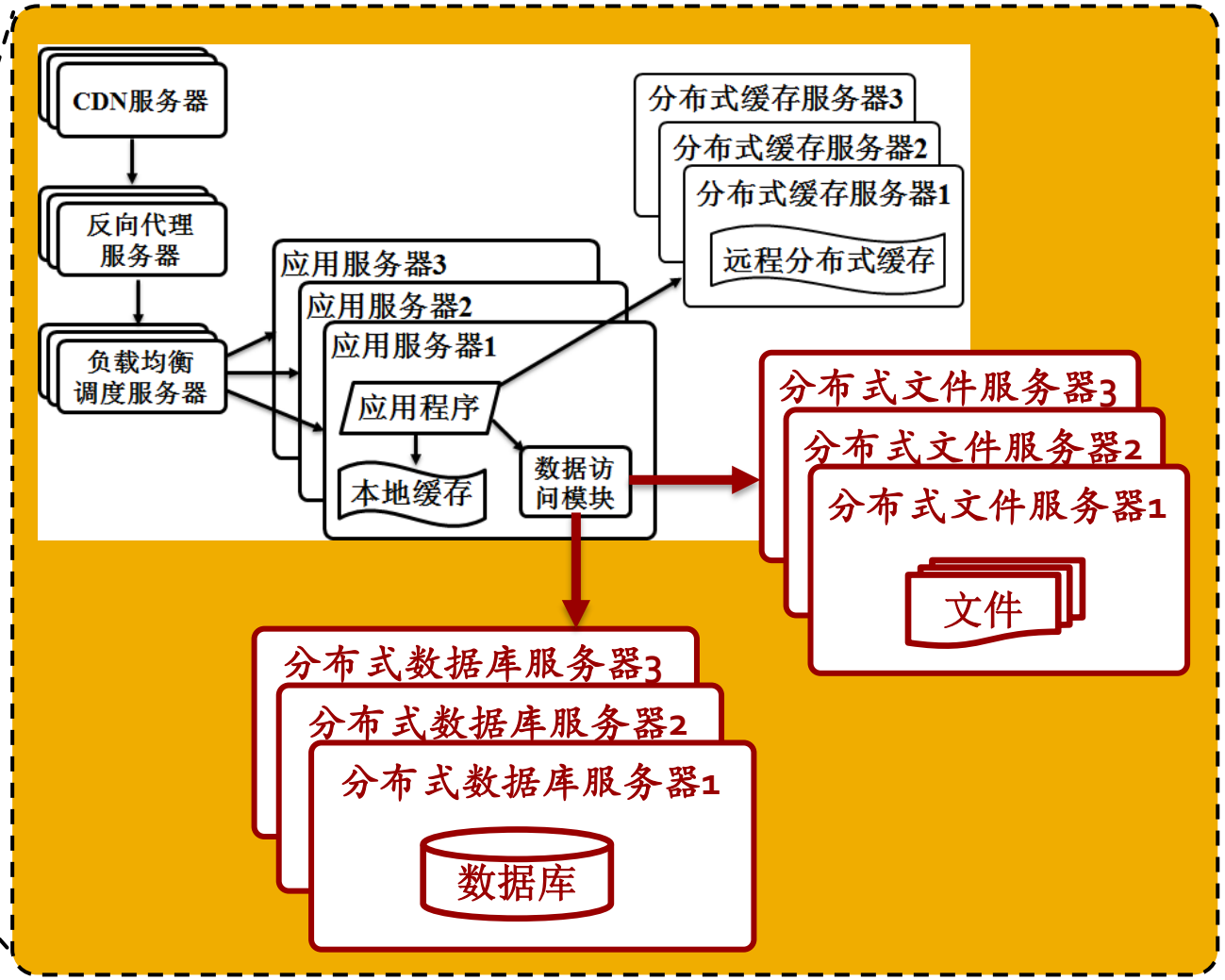


CDN部署在网络提供商的机房，使用户在请求网络服务时，可以从距离自己最近的网络提供商机房获取数据。

反向代理部署在网站的中心机房，当用户请求达到中心机房后，反向代理服务器中缓存着用户请求的资源，就直接返回给用户。

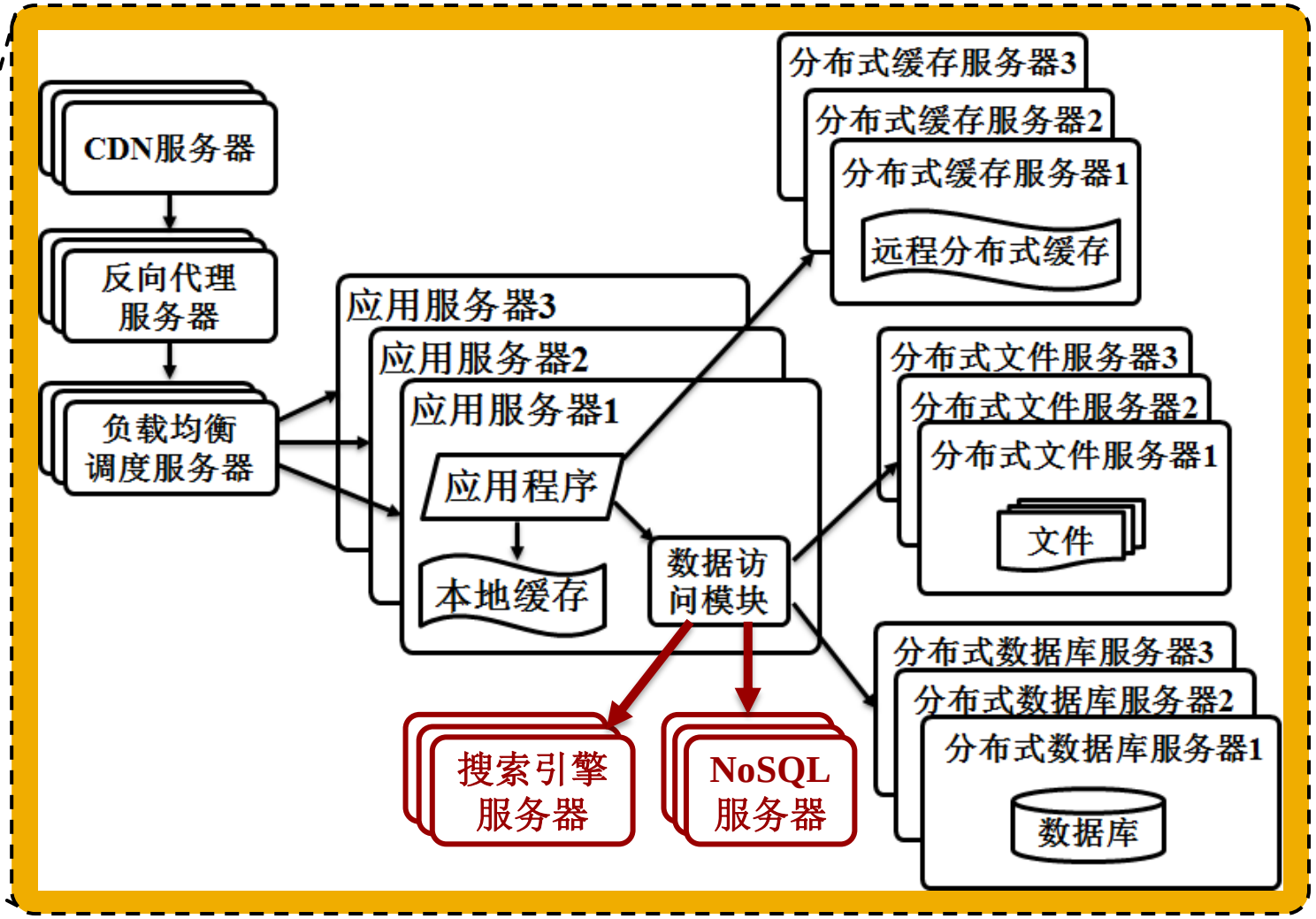
大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- **使用分布式文件系统和分布式数据库系统**
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务



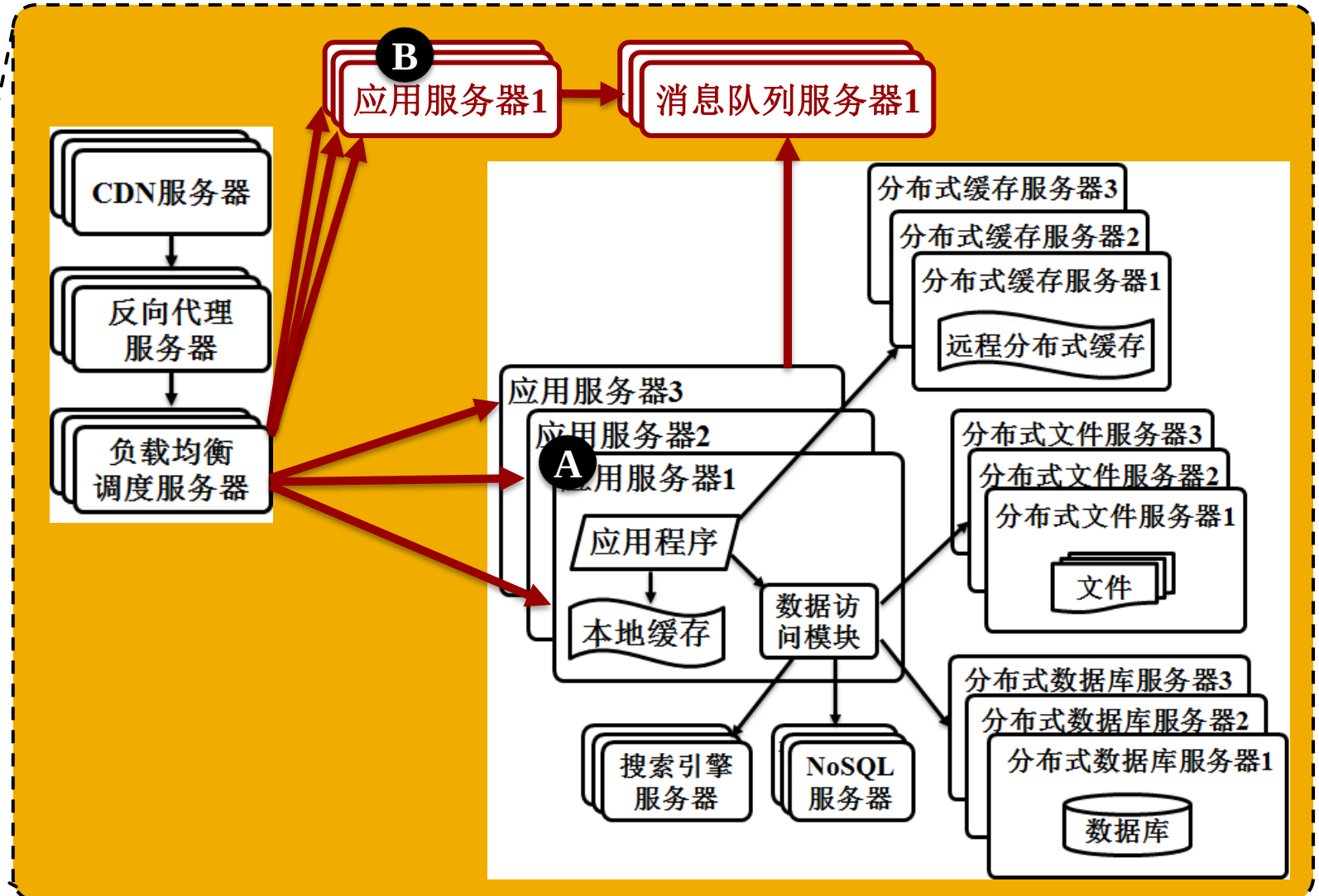
大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- **使用NoSQL和搜索引擎**
- 业务拆分
- 分布式服务



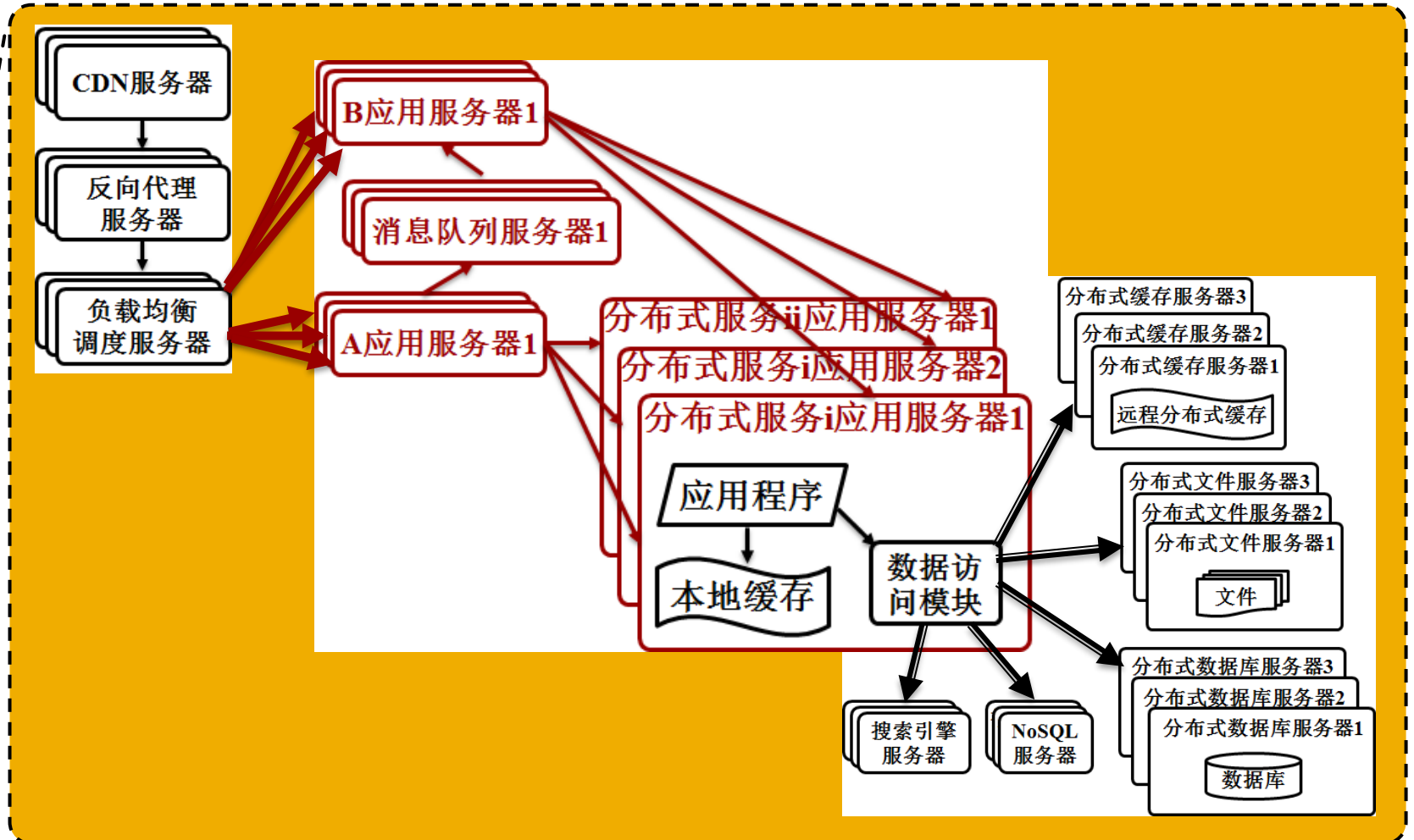
大型网站架构演化发展历程

- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- **业务拆分**
- 分布式服务



大型网站架构演化发展历程

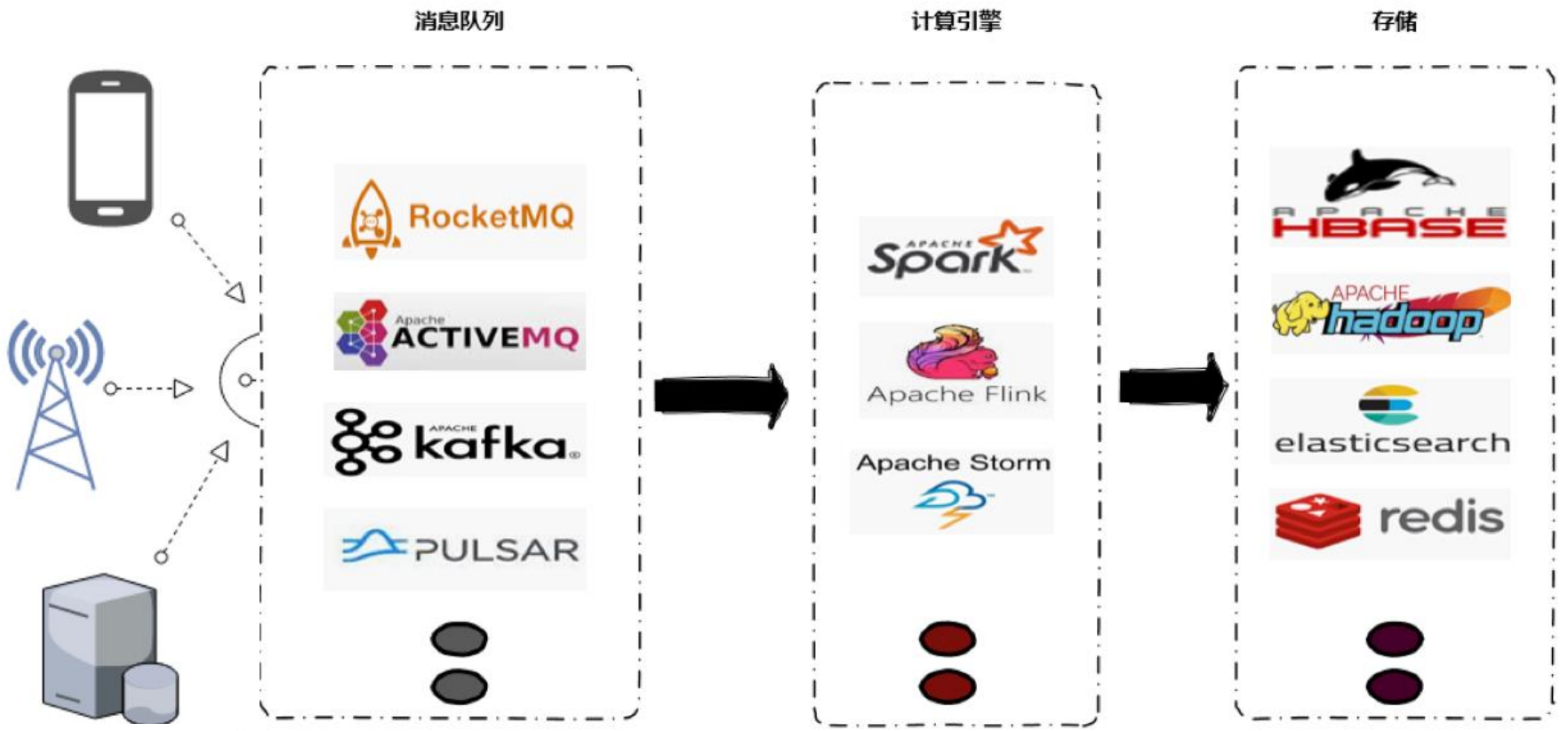
- 初始阶段
- 应用服务和数据服务分离
- 使用缓存改善网站性能
- 使用应用服务器集群改善网站的并发处理能力
- 数据库读写分离
- 使用反向代理和CDN加速网站响应
- 使用分布式文件系统和分布式数据库系统
- 使用NoSQL和搜索引擎
- 业务拆分
- 分布式服务



（二） NetEase CloudMusic Real-time Data Warehouse

网易云音乐实时数仓架构设计及备选方案

目标：将实时产生的日志采集后进行常见的ETL、全局聚合以及Window 聚合等实时计算，之后存储。



目前流平台通用的架构一般来说包括消息队列、计算引擎和存储三部分，通用架构如图所示：

- 客户端或者 web 的 log 日志会被采集到消息队列；
- 计算引擎实时计算消息队列的数据；
- 实时计算结果以 Append 或者 Update 的形式存放到实时存储系统中去。

为什么选 Kafka?

- 高吞吐，低延迟：每秒几十万 QPS 且毫秒级延迟；
- 高并发：支持数千客户端同时读写；
- 容错性：支持数据备份，允许节点丢失；
- 可扩展性：支持热扩展，不会影响当前线上业务。

为什么选择Apache Flink?

- 高吞吐，低延迟，高性能；
- 高度灵活的流式窗口；
- 状态计算的 Exactly-once 语义；
- 轻量级的容错机制；
- 支持 EventTime 及乱序事件；
- 流批统一引擎。

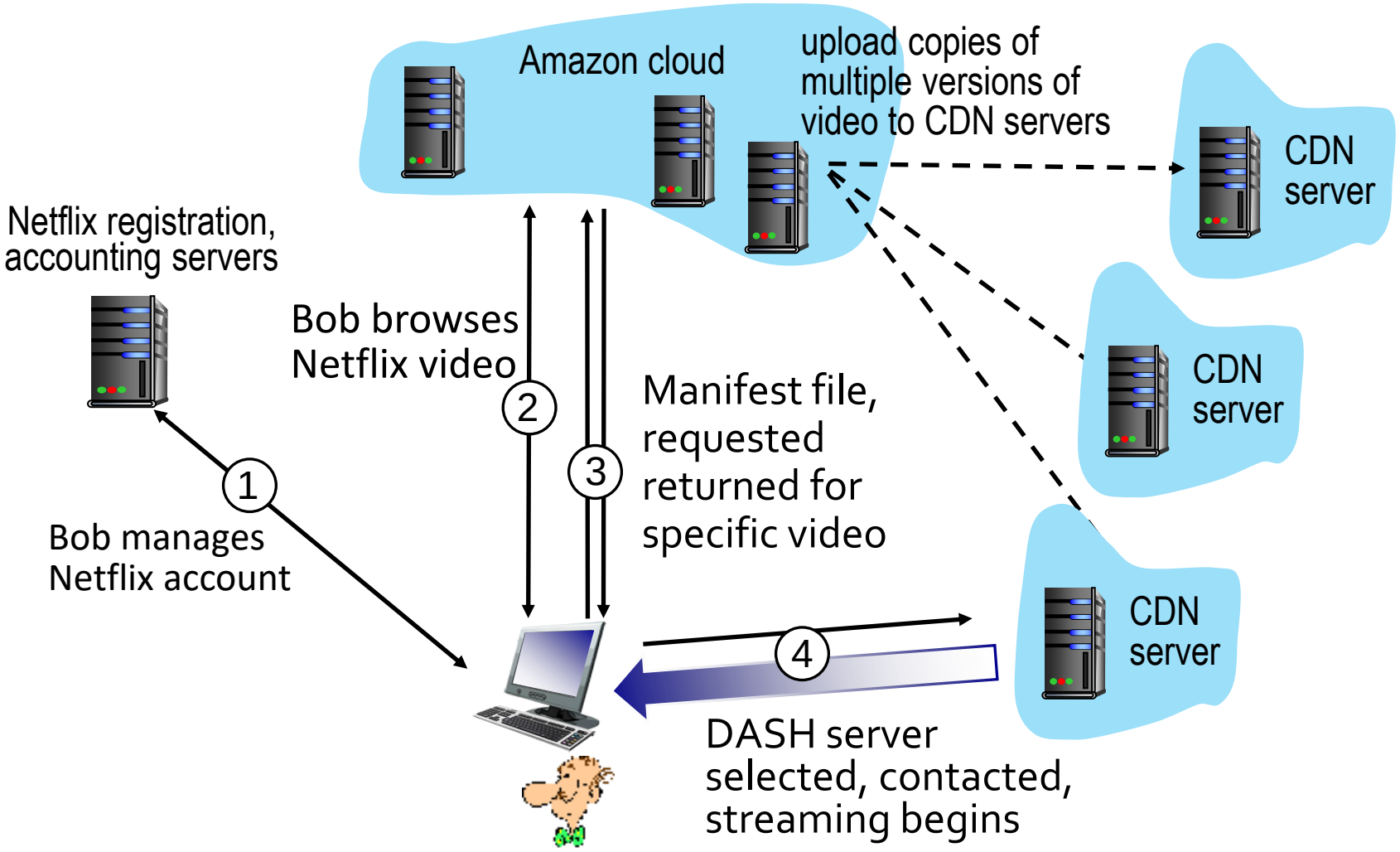
（三） Video Streaming Architecture

Video Streaming and CDNs: context

- stream video traffic: major consumer of Internet bandwidth
  - Netflix, YouTube, Amazon Prime Video: 80% of residential ISP traffic (2020)
- *challenge:* scale - how to reach ~1B users?

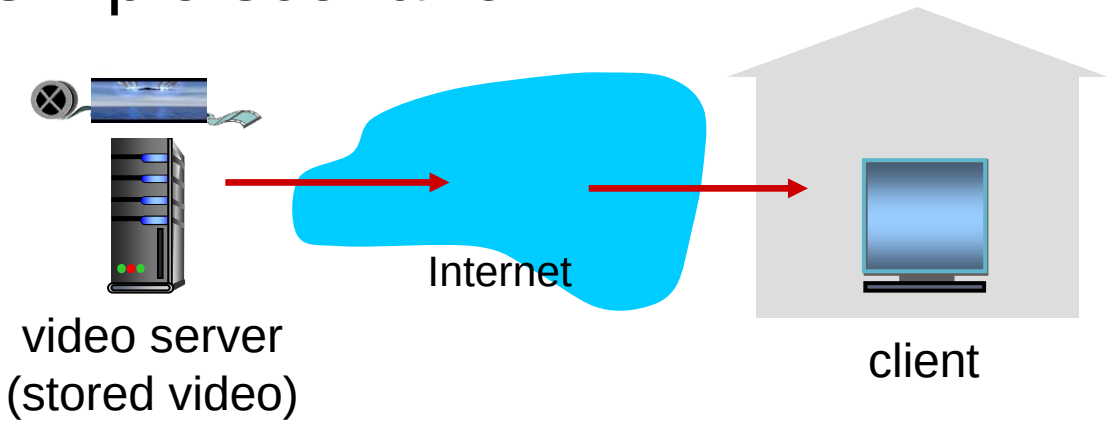
- *challenge:* heterogeneity
  - different users have different capabilities (e.g., wired versus mobile; bandwidth rich versus bandwidth poor)
- *solution:* distributed, application-level infrastructure

Case study: Netflix



Streaming stored video

simple scenario:

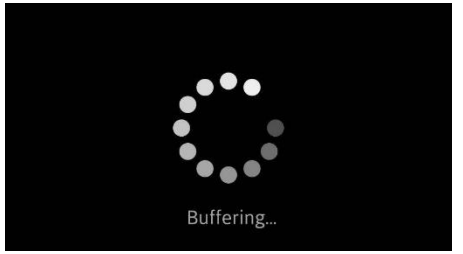


Main challenges:

- server-to-client bandwidth will *vary* over time, with changing network congestion levels (in house, access network, network core, video server)
- packet loss, delay due to congestion will delay playout, or result in poor video quality

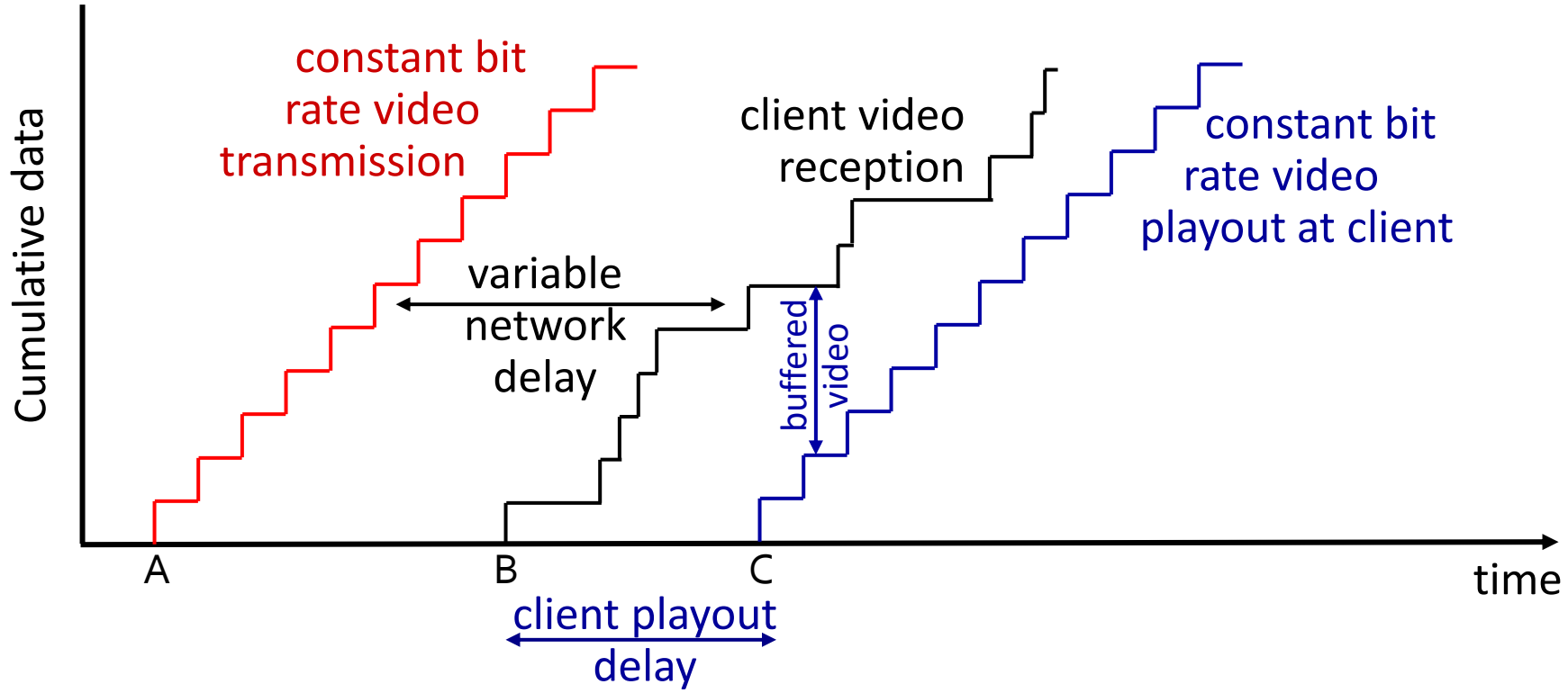
Streaming stored video: challenges

- continuous playout constraint: during client video playout, playout timing must match original timing
  - … but network delays are variable (jitter), so will need client-side buffer to match continuous playout constraint



- other challenges:
  - client interactivity: pause, fast-forward, rewind, jump through video
  - video packets may be lost, retransmitted

Streaming stored video: playout buffering

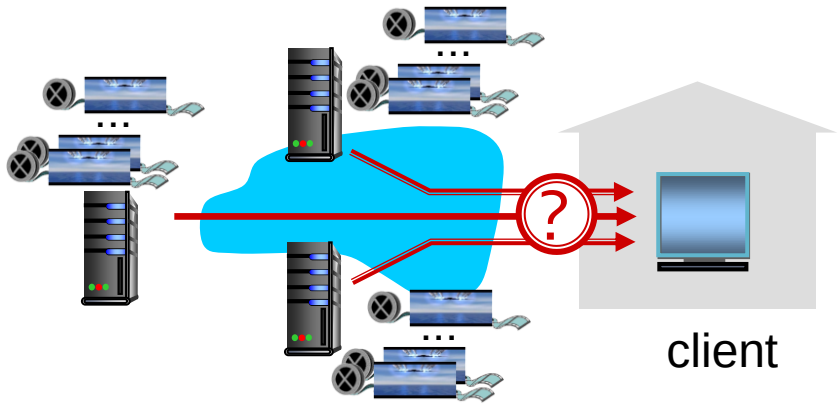


- *client-side buffering and playout delay:* compensate for network-added delay, delay jitter

Streaming multimedia: DASH

server:

- divides video file into multiple chunks
- each chunk encoded at multiple different rates
- different rate encodings stored in different files
- files replicated in various CDN nodes
- *manifest file:* provides URLs for different chunks



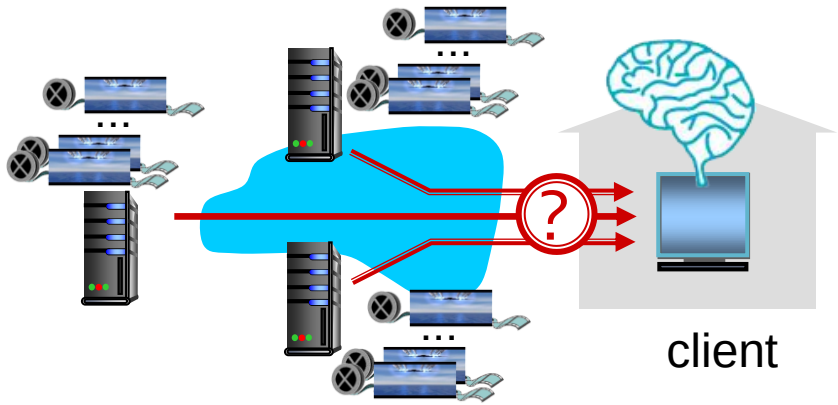
*D*ynamic, *A*daptive *S*treaming over *H*TTP

client:

- periodically estimates server-to-client bandwidth
- consulting manifest, requests one chunk at a time
  - chooses maximum coding rate sustainable given current bandwidth
  - can choose different coding rates at different points in time (depending on available bandwidth at time), and from different servers

Streaming multimedia: DASH

- *"intelligence"* at client: client determines
  - *when* to request chunk (so that buffer starvation, or overflow does not occur)
  - *what encoding rate* to request (higher quality when more bandwidth available)
  - *where* to request chunk (can request from URL server that is "close" to client or has high available bandwidth)



Streaming video = encoding + DASH + playout buffering

Content distribution networks (CDNs)

challenge: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *option 1:* single, large "mega-server"
  - single point of failure
  - point of network congestion
  - long (and possibly congested) path to distant clients

….quite simply: this solution *doesn't scale*

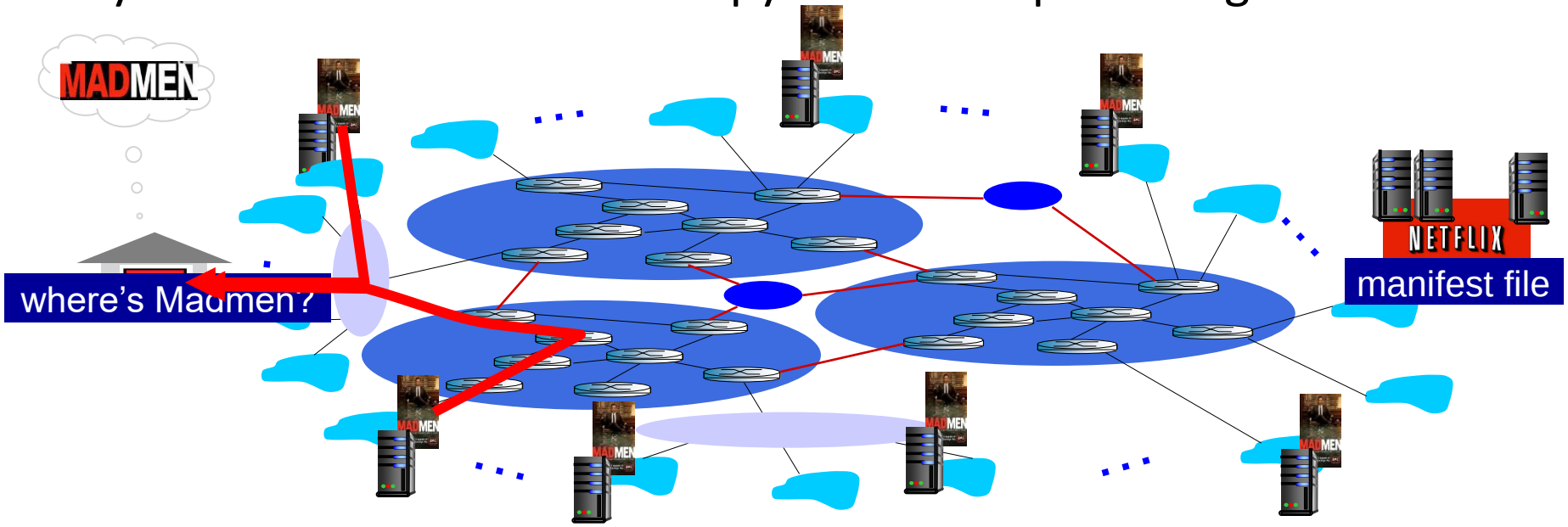
Content distribution networks (CDNs)

challenge: how to stream content (selected from millions of videos) to hundreds of thousands of *simultaneous* users?

- *option 2:* store/serve multiple copies of videos at multiple geographically distributed sites *(CDN)*
  - *enter deep:* push CDN servers deep into many access networks
    - close to users
    - Akamai: 240,000 servers deployed in > 120 countries (2015)
  - *bring home:* smaller number (10's) of larger clusters in POPs near access nets
    - used by Limelight

Content distribution networks (CDNs)

- CDN: stores copies of content (e.g. MADMEN) at CDN nodes
- subscriber requests content, service provider returns manifest
  - using manifest, client retrieves content at highest supportable rate
  - may choose different rate or copy if network path congested



Content distribution networks (CDNs)

Internet host-host communication as a service

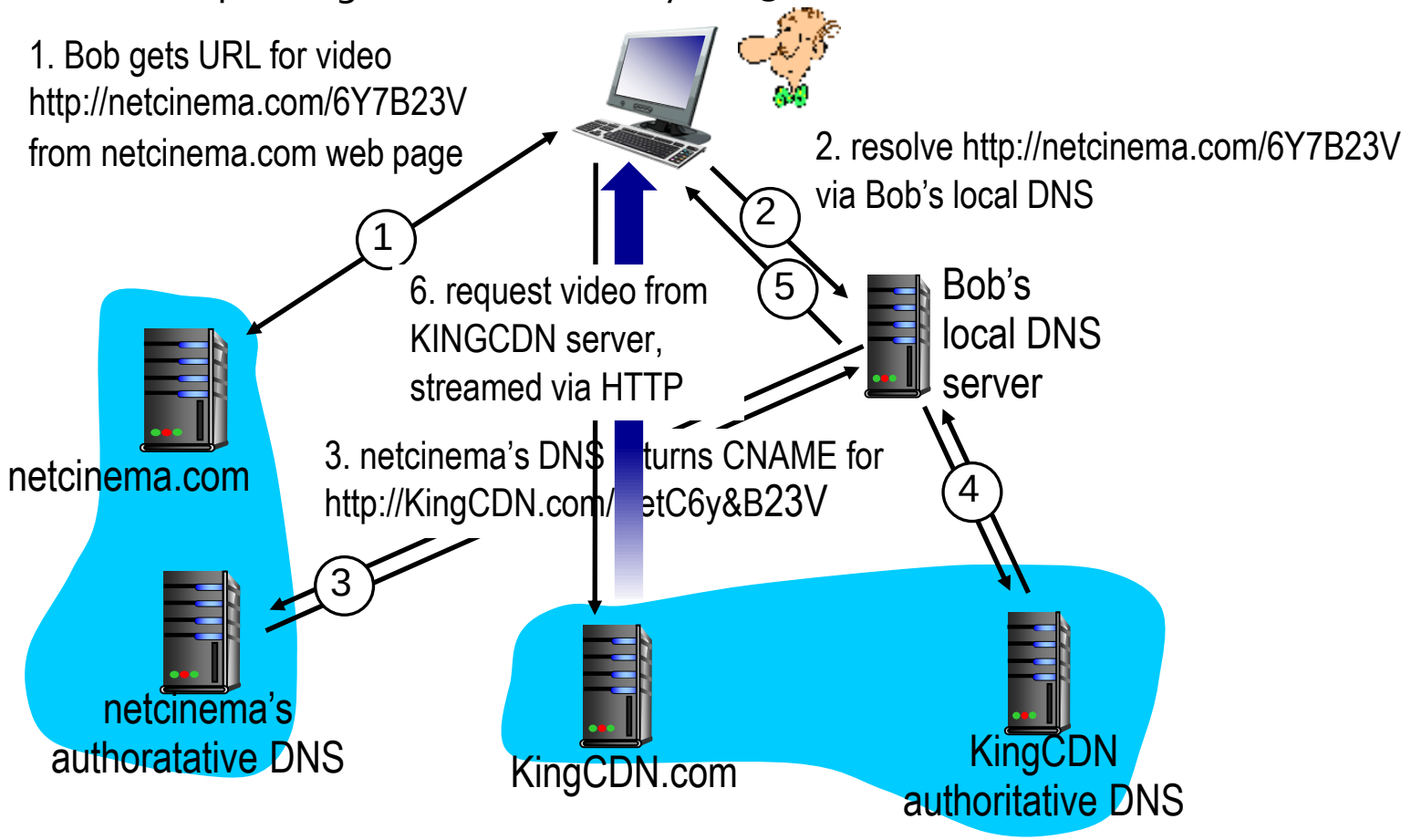
OTT challenges: coping with a congested Internet from the "edge"

- what content to place in which CDN node?
- from which CDN node to retrieve content? At which rate?

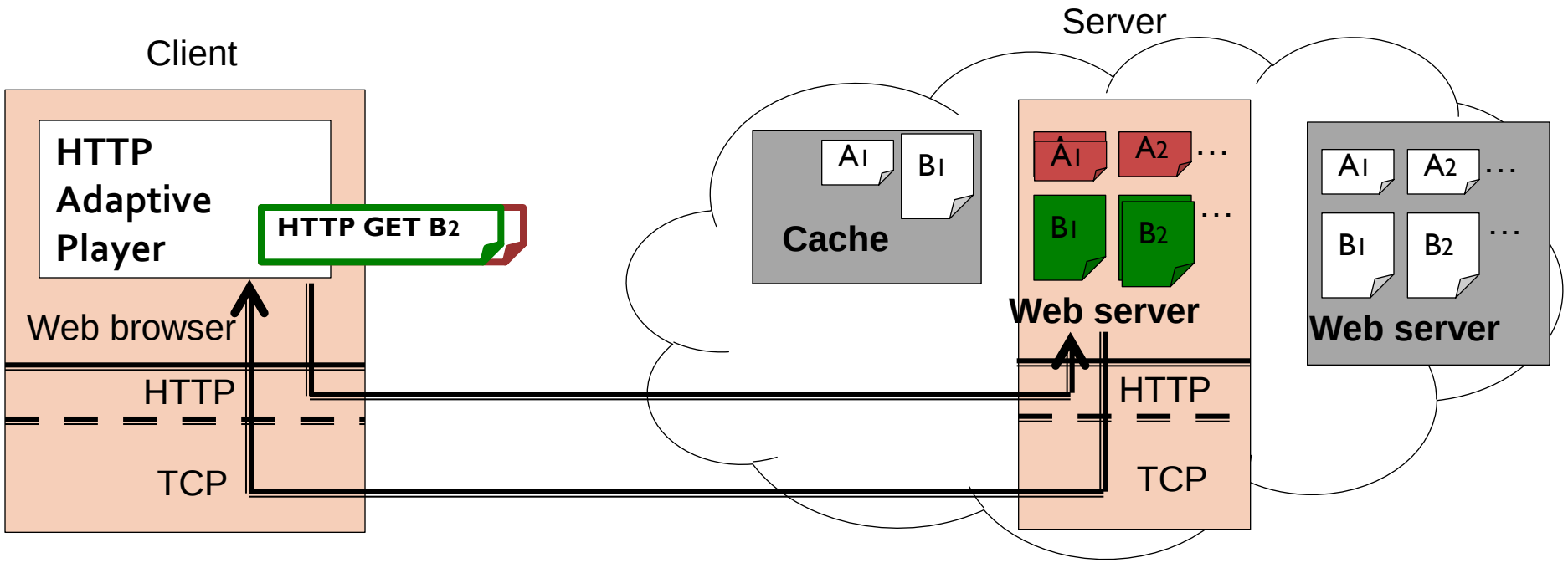
CDN content access: a closer look

Bob (client) requests video http://netcinema.com/6Y7B23V

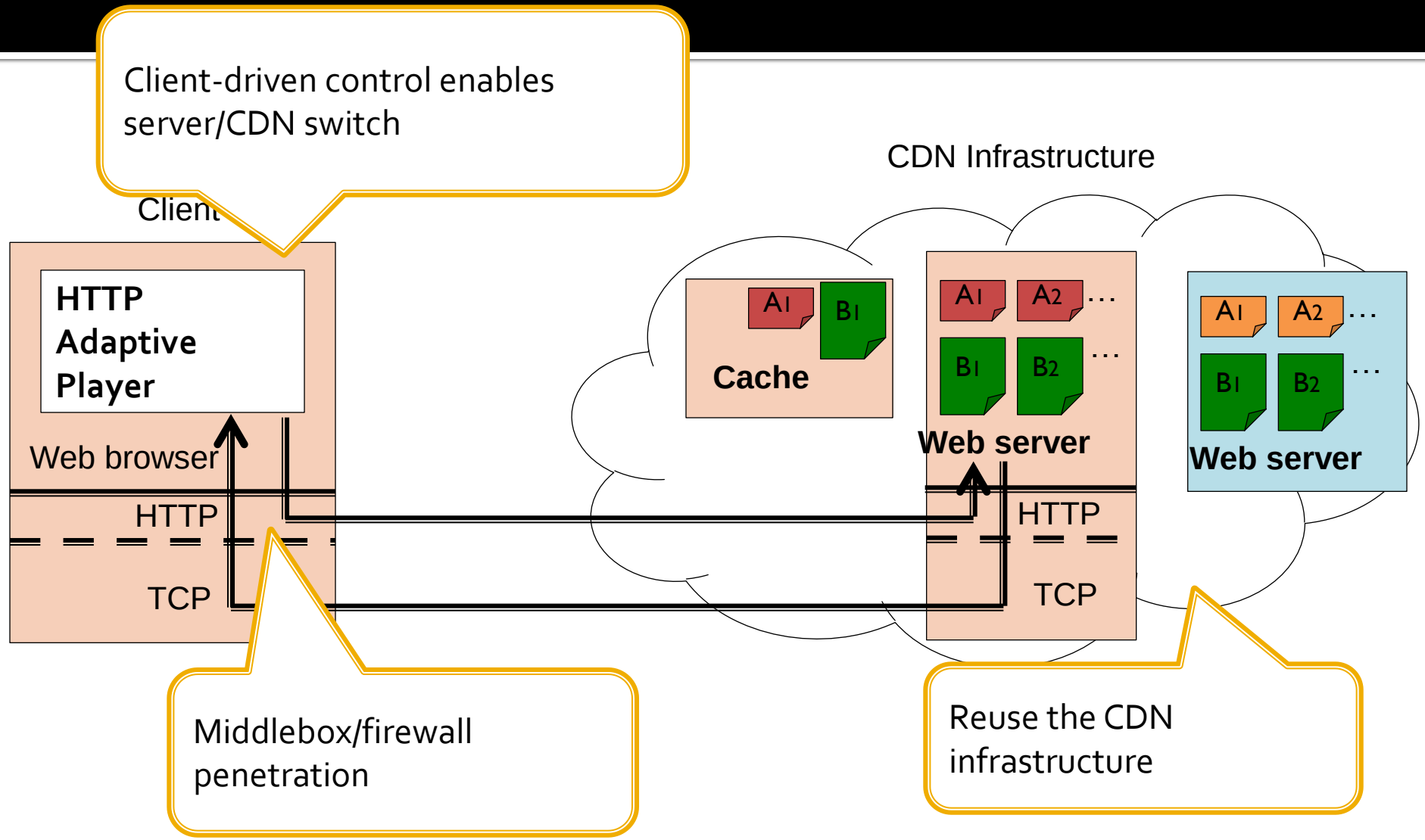
- video stored in CDN at http://KingCDN.com/NetC6y&B23V



HTTP Chunking Protocol



Reasons for Wide Adoption



Outline

- Architecture and Quality Attributes
- Architecture Design Principles
- Architecture Styles
- Distributed Architecture Styles
- Architecture Design Patterns for Quality Attributes
- Large-Scale Architecture Examples

Thank you!