L05 System Design III

Security, Reliability, and Availability

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

A three-part series: System design in our course

L03: Design I

- **Modularity**
  - How to design modular systems?
- **Data management**
  - How to manage your data?

L04: Design II

- **Performance**
  - How to design performant systems?
- **Concurrency (Scale-up)**
  - How to scale-up systems?
- **Scalability (Scale-out)**
  - How to scale-out systems?

L05: Design III

- **Security**
  - How to secure your systems?
- **Fault tolerance**
  - How to make systems **reliable** & **available**?

System implementation

Today's learning goals

- **Part I:** Security
  - Security engineering
  - Software security in the cloud
- **Part II:** Reliability and availability
  - Single-node reliable systems and associated issues
  - Replication as the general recipe for fault-tolerance
  - Fault-tolerance for stateful services
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Outline

- **Part I: Security**
  - **Security engineering**
    - Threat model and security (CIA) properties
    - Security design principles: Compartilization, least privileges, and isolation
    - A general recipe to build secure systems
    - Access control: ACLs and capabilities
  - Software security in the cloud
- **Part II:** Reliability and availability
- **Part III:** Pattern implementation

Software security 101

- "**Computer security** studies how systems behave in the presence of an adversary"
  - **Actively tries to cause the system to misbehave*
- A computer system **executes computation according to a specification**
  - A deviation from the specification leads to computer failure or security vulnerabilities
- **Bugs Vs. Vulnerabilities:**
  - **Bug:** A flaw in a computer system that results in an unexpected outcome or failure
  - **Security vulnerability:** A vulnerability is a bug which can be exploited by an attacker, to perform unauthorized actions within a computer system

Security engineering

- **Security policies**
  - **Threat model:**
    - What are we trying to protect?
    - What are our assumptions (deployment, attackers capabilities, etc.)?
  - **Security properties:** What are the properties we are trying to enforce?
- **Security design principles**
  - Least privilege
  - Comparatilization
  - Isolation via privilege mediation (a reference monitor)

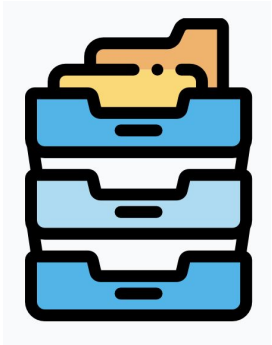
Threat model

- A **threat model** is used to explicitly list all threats that jeopardize the security of a system
- A threat model defines the **environment of the system** and **the capabilities of an attacker**
  - Define system assumptions: **Trusted and untrusted parts**
  - Enumerating and prioritizing **all potential threats**
  - **Risk management a**nd trade-offs
- **Typical questions:**
  - What are the **high value-assets in a system**?
  - Which **components of a system are most vulnerable**?
  - What are the **most relevant threats or attack surfaces**?

Security CIA properties

- **Confidentiality**
  - Confidentiality of a service **limits access of information to privileged entities**
  - Confidentiality requires **authentication and access rights** according to a policy
- **Integrity**
  - Integrity of a service **limits the modification of information to privileged entities** (or an attacker cannot modify protected data)
  - Integrity property also requires **authentication and access rights** according to a policy
- **Availability**
  - Availability of a service guarantees that the **service remains accessible** (or, availability prohibits an attacker from hindering computation
  - The availability property guarantees that **legitimate uses of the service remain possible**

Security design principles

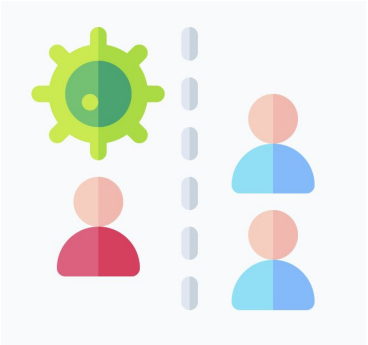


#1:

Compartilization

#2:

Principle of least privileges

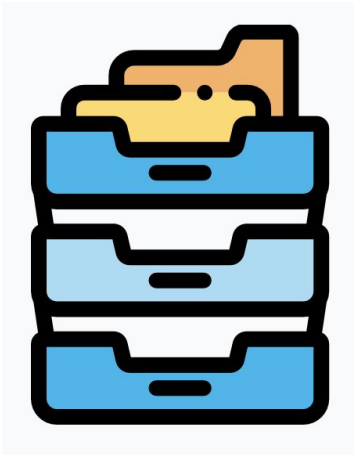


#3:

Isolation via privilege mediation

#1: Compartilization

- The idea behind **compartmentalization is to break a complex system into small components** (“divide and conquer” from modularity – L03) that follow a well-defined communication protocol to request services from each other
  - Under this model, **faults/security vulnerabilities can be constrained to a given compartment/module**
  - After compromising a single compartment, an **attacker is restricted** to the protocol to request services from other compartments
  - To compromise a remote target compartment, **the attacker must compromise all compartments on the path** from the initially compromised compartment to the target compartment

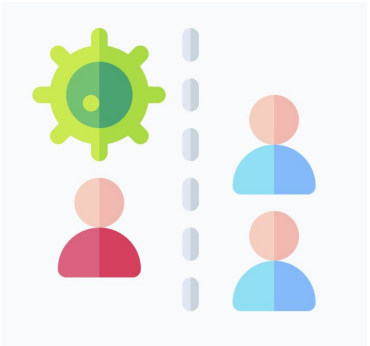


#2: Least privilege

- The principle of least privilege guarantees that **a component has the least amount of privileges needed to function**
  - In other words, each component should only be given the privilege it requires to perform its duty and no more
- **Different components need privileges (or permissions) to function**
  - If a **component follows least privilege then any privilege that is further removed from the component removes some functionality**

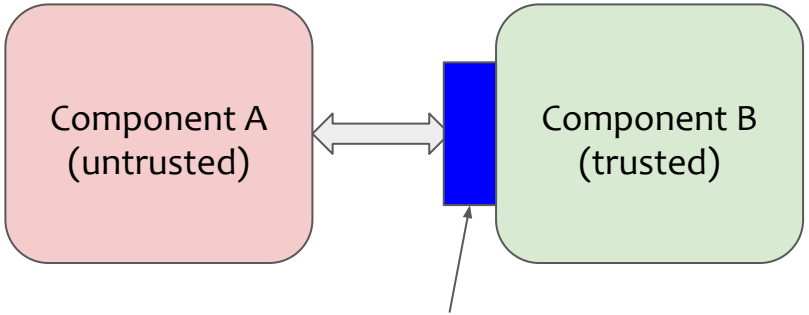
#3: Isolation

- Isolation separates **two components from each other and confines their interactions to a well-defined API**
- To enforce isolation between components, all of them require some form of **“security monitor”**
  - A **security monitor runs at higher privileges** than the isolated components and ensures that they adhere to the isolation
  - **Any violation to the isolation is stopped by the security monitor** and, e.g., results in the termination of the violating component



A high-level recipe for secure system design

- Break system into compartments
- Ensure each compartment is isolated
- Ensure each compartment runs with least privilege
- Treat compartment interface as the trust boundary



Always aim to reduce the **trusted computing base (TCB)** → **Lower TCB is better!**

Reference/Trusted monitor

- **Mediates requests** from applications
  - **Enforces confinement**
  - Implements a **specified protection policy**
- **Must always be invoked**
  - Must be **'impossible' to circumvent**
- **Tamperproof**
  - Reference **monitor cannot be killed,** or if killed then monitored process is killed too
  - **Runs in high-privileged mode** (part of the trusted computing base)
- Ideally, **small enough to be analyzed and validated**

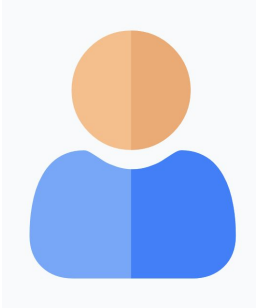
Authentication vs authorization

- **Authentication**
  - **Determining the identity of a user**
  - May also involve verification of IP address, machine, time, etc…
  - Determine whether a user is allowed to access the system at all
  - Used for audit, not just authorization
- **Authorization**
  - Assuming identity of user is known, **determine whether some specific action is allowed**
  - Access control allows authorization

Access control: Terminology

- **Protected entities:** "objects/resources" **O**
  - **Objects can be any resource**
    - Files
    - System resources (servers, printers, etc.)
    - Memory

- **Active objects:** "subjects/principals" **S** (i.e., users/processes)
  - **Typical each subject associated with some principal/owner**
    - In identity-based access control, security policies ultimately refer to the human users who are the principals



Subject/Principal

Object/Resources

Two mechanisms for access control

Access control lists (ACLs)

Capabilities

ACL vs capabilities

- **ACLs**
  - ACL is like a guest list with the names
  - Only the guest with names on the list are allowed

- **Capabilities**
  - Capabilities are like a ticket
  - Anyone with a valid ticket can access the venue

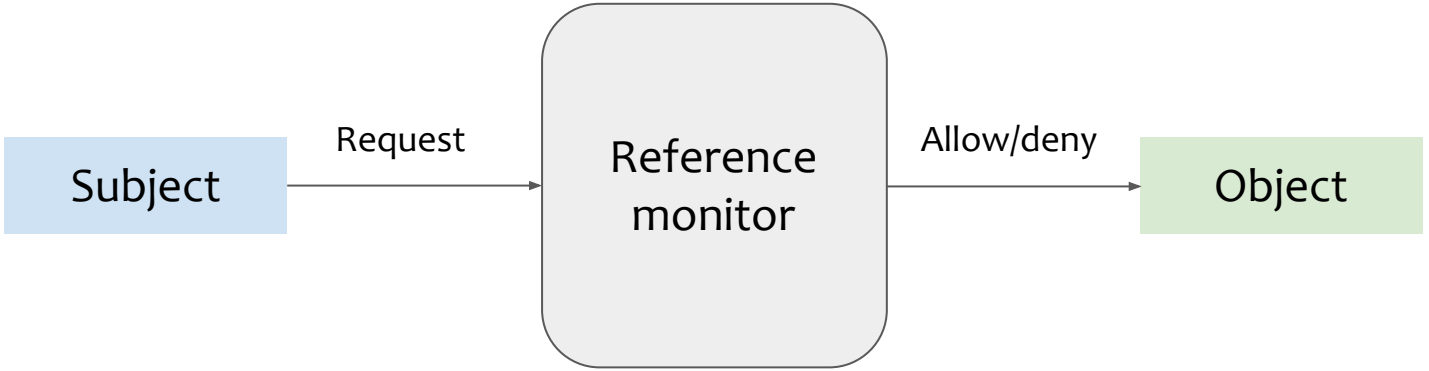
Two mechanisms for access control

Access control lists (ACLs)

Capabilities

Access control via “reference monitor”

- Determining who has access to specific files, or other system resources



Access control lists (ACLs)

- **An ACL** is a list of permissions associated with a system resource (object)
  - An ACL specifies **which users or system processes are granted access to objects**, as well as **what operations are allowed on given objects**

- **General hints:**
  - **Fine-grained access control is better;** e.g., control access to files not just directories
  - **Least privilege principle grants** minimum abilities necessary to complete task

Access control matrix

- Matrix indexed by all **subjects and objects**
  - Characterizes rights of each subject with respect to each object
- Formally: **set of objects O** and **subjects S**, set of **possible rights**
  - Matrix A with subjects labeling the rows and objects labeling the columns
  - The entry (s,o) contains the rights for s on o
  - Examples: read/write/execute/etc.

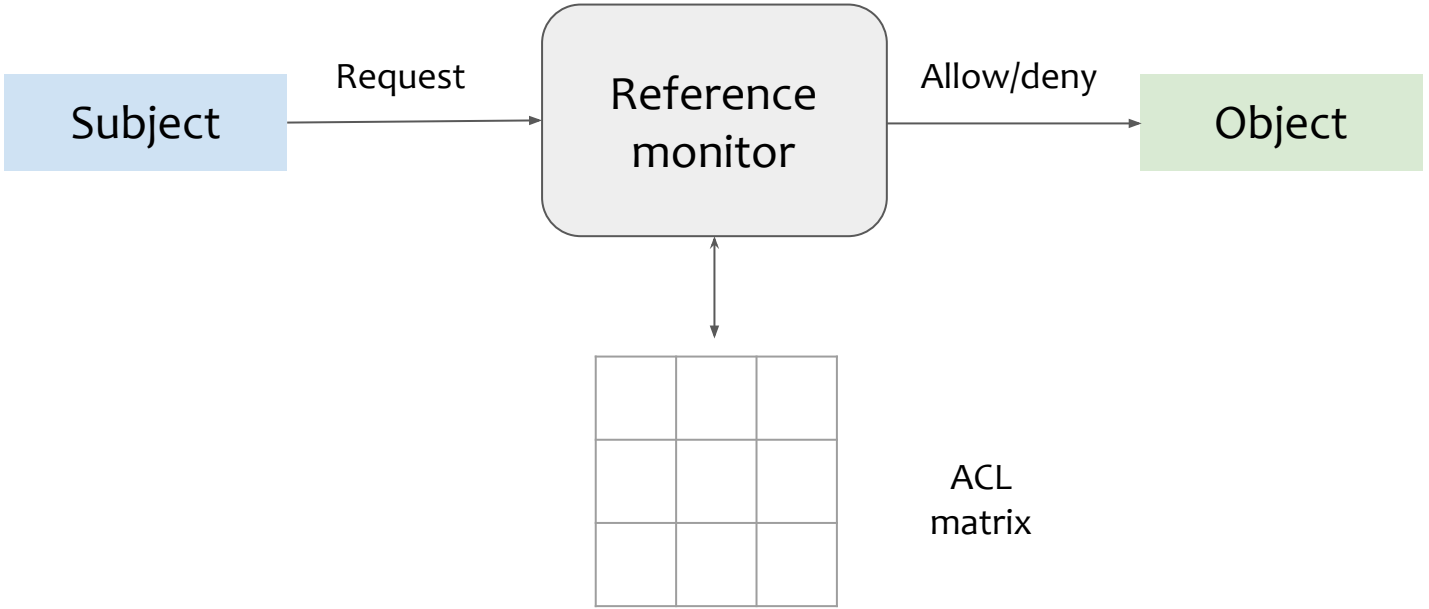
Objects

Subjects

	File 1	**…**	**File n**
User 1	{r,w}		{}
..			
User n	{}		{r, w, x}

Access control matrix

- **Access control matrix** provides a useful way to think about the access rights
  - It can also be a way to implement access control



Limitations of ACLs

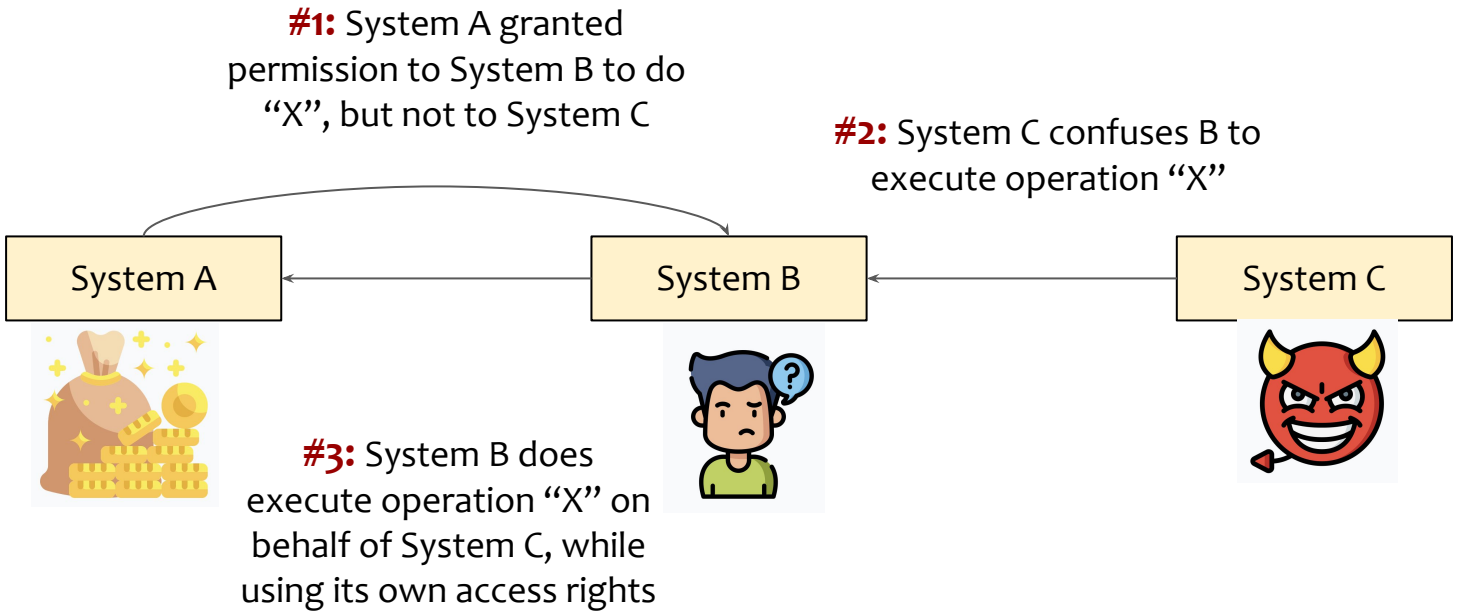
- **Number of subjects/objects is very large**
  - Most entries blank/default
- **One central matrix modified** every time subjects/objects are created/deleted or rights are modified
  - "Small' change can result in "many" changes to the access control matrix
  - E.g., making a file publicly readable
- ACLs are vulnerable to the **"Confused deputy problem"**
  - A confused deputy is a legitimate, more privileged computer program that is tricked by another program into misusing its authority on the system
  - It is a specific type of **privilege escalation**

Confused deputy problem:

- https://en.wikipedia.org/wiki/Confused_deputy_problem
- https://dimosr.github.io/confused-deputy/

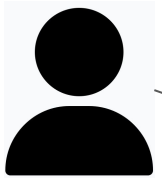
Confused deputy problem

- The **confused deputy problem** is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action



Confused deputy problem: The original example

Normal user: Instructs compiler to write debug info to a file



Malicious user: Instructs compiler to write the debug info to an important file (e.g., billing info)

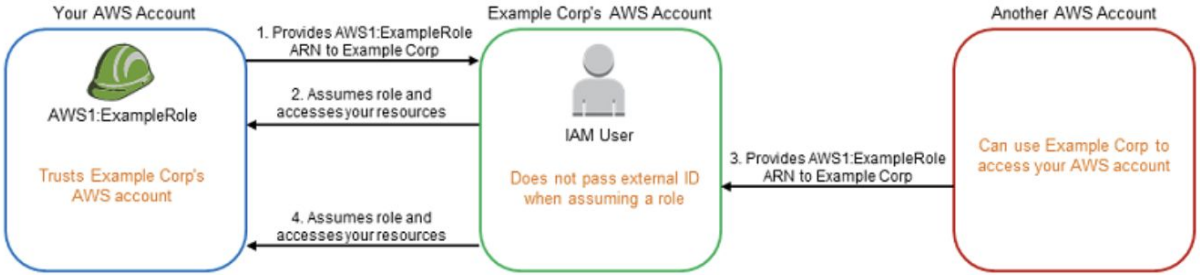
Writes the compiler output to a given file

Compiler running on a time-sharing service across users

Compiler has access to **the filesystem**

An example of confused deputy problem in AWS

- A **third-party service (Example Corp)** linked to **your AWS account** can be used by **another "malicious" AWS account** to access your resource!



How Amazon AWS prevents the confused deputy problem?
https://docs.aws.amazon.com/IAM/latest/UserGuide/confused-deputy.html

Two mechanisms for access control

Access control lists (ACLs)

Capabilities

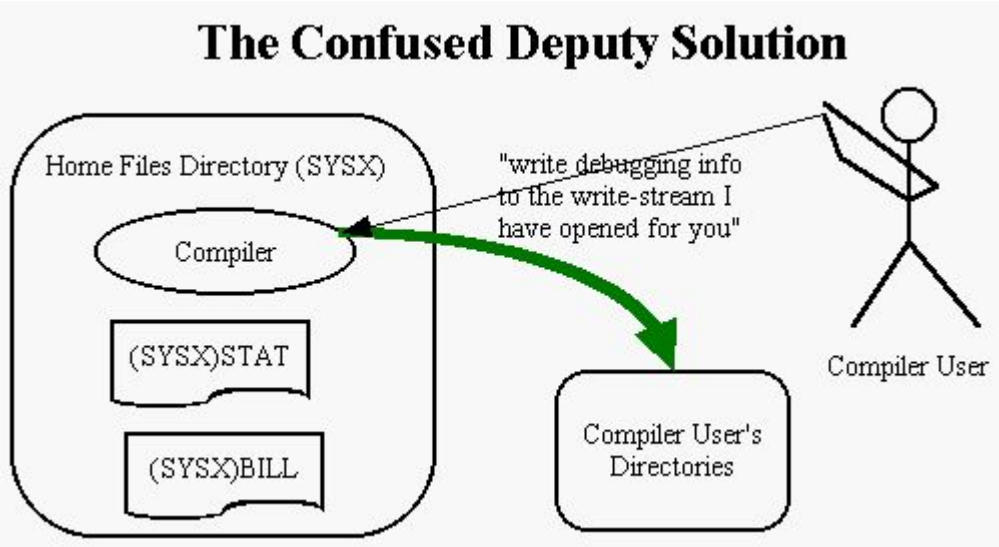
Capabilities

- **A capability** is a unique, unforgeable token that gives a process permission to access an entity or object in system
- Types of operations on capabilities
  - **Delegate:** Delegation refers to the process of transferring or granting certain privileges or access rights from one entity to another
  - **Revoke:** Revoke refers to the act of canceling or removing previously granted access rights or privileges
  - **Grant:** Granting access involves providing or giving permission to a user or entity to perform certain actions or access specific resources in a computer system

Capability systems protect against the confused deputy problem!
The Confused Deputy (or why capabilities might have been invented):
https://www.cis.upenn.edu/~sga001/classes/cis331f19/resources/hardy88confused.pdf

Capabilities can solve the confused deputy problem!

- **Users employ capabilities to grant/revoke/delegate** access to the resources
  - **Important:** The compiler doesn't have access to the file system, it receives capability to write to a file (provided by the compiler user)



The confused deputy solution
http://www.skyhunter.com/marcs/capabilityIntro/sysxfix.html

More advantages of capabilities

- Capabilities allow **a convenient view of the access rights**
  - When a subject holds a capability for an object, it knows it has access to that object
  -
- Capabilities allow **finer-grained treatment of subjects**
  - E.g., at the process level rather than the user-level

- Capabilities are better at **enforcing "principle of least privilege"**
  - Provide access to minimal resources, to the minimal set of subjects

Summary

Access control lists (ACLs)

- ACL associated with each object
- Upon request, check user/group against the ACL
- Relies on authentication of users

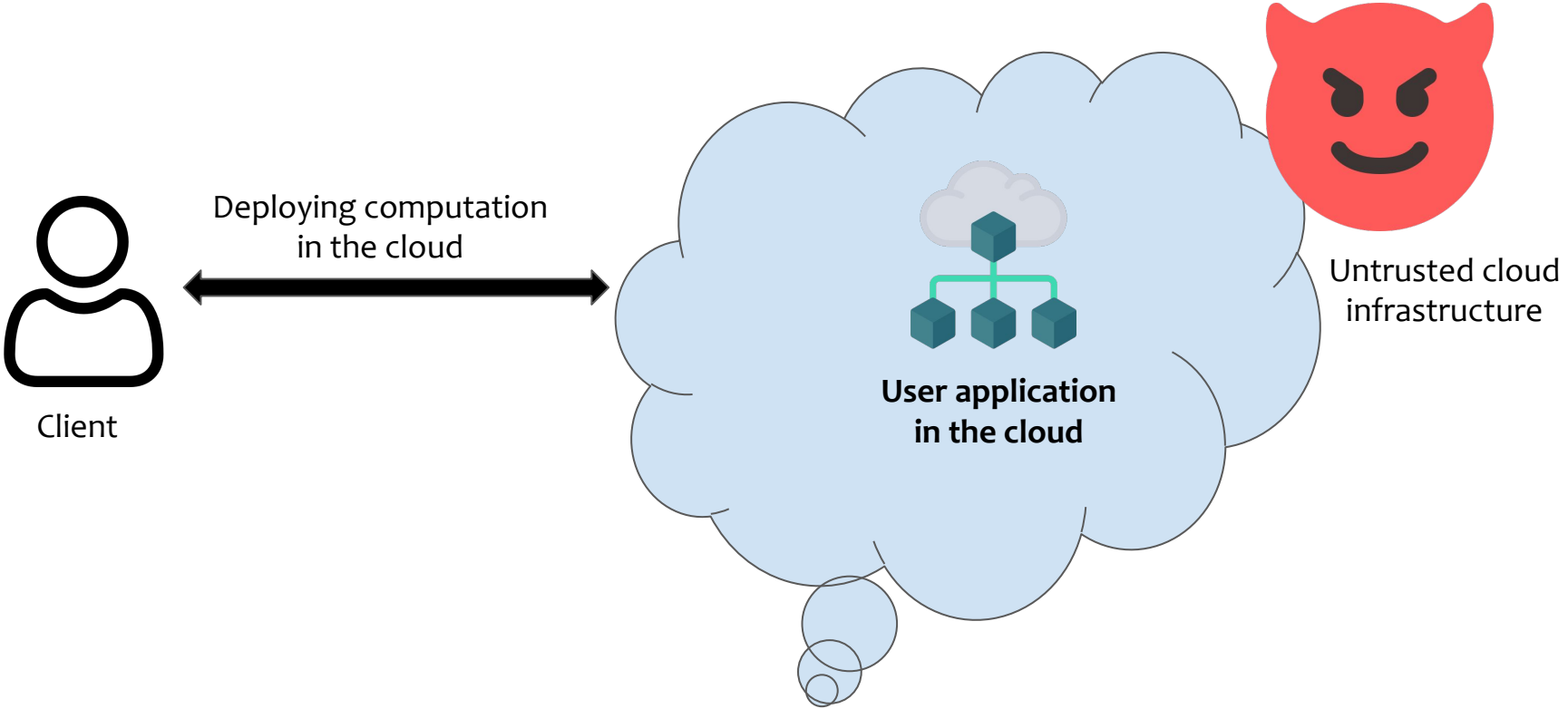
Capabilities

- Can be passed from one user/process to another
- Upon request, check validity of capability
- No need to know the identity of the user/process making the request

Outline

- **Part I: Security**
  - ~~Security engineering~~
  - **Software security in the cloud**
    - Security challenges in the cloud
    - Secure system stack
      - Compute
      - Network
      - Storage
    - Authentication, key management, and attestation
- **Part II:** Reliability and availability
- **Part III:** Pattern implementation

A secure cloud architecture



Challenges of system security in the cloud

- **Outsourced infrastructure**
  - Third-party infrastructure
- **Multitenancy**
  - Computing infrastructure is shared across multiple tenants
- **Identity management/authorization**
  - Require establishing trust
- **Compliance (e.g., GDPR)**
  - Data and code might be handled in a different administrative jurisdiction
- **Misconfiguration and software bugs**
  - Large trusted computing base (infrastructure) operated/developed by multiple third-parties

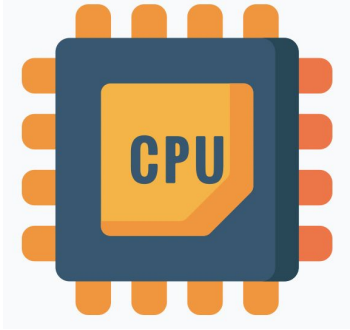
Design of secure systems in the cloud

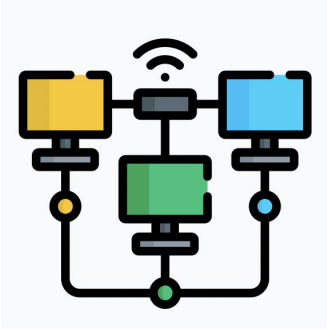
- **Secure systems stack**
  - Computing
  - Network
  - Storage
- **Authentication, key management, and attestation**
  - Identity and access management (IAM)
  - Key management service (KMS)
  - Remote attestation

Design of secure systems in the cloud

- **Secure systems stack**
  - Computing
  - Network
  - Storage
- **Authentication, key management, and attestation**
  - Identity and access management (IAM)
  - Key management service (KMS)
  - Remote attestation

Part A: Secure systems stack in the cloud





Compute
(Data in use)

Protection the code and data, while computing on the CPU, accessing over the RAM

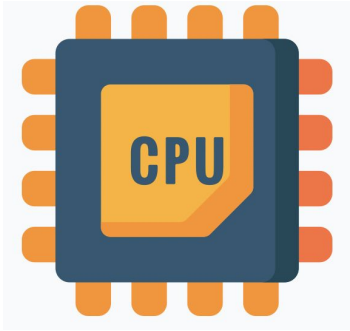
Network
(Data in motion)

Protecting the data being transmitted over the untrusted network connection

Storage
(Data at rest)

Protecting the data storage on untrusted persistent storage, e.g., disk/SSDs

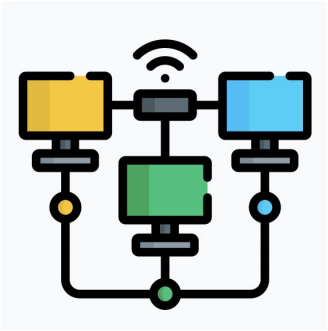
Part A: Secure systems stack in the cloud



Compute

(Data in use)

Protection the code and data, while computing on the CPU, accessing over the RAM



Network

(Data in motion)

Protecting the data being transmitted over the untrusted network connection

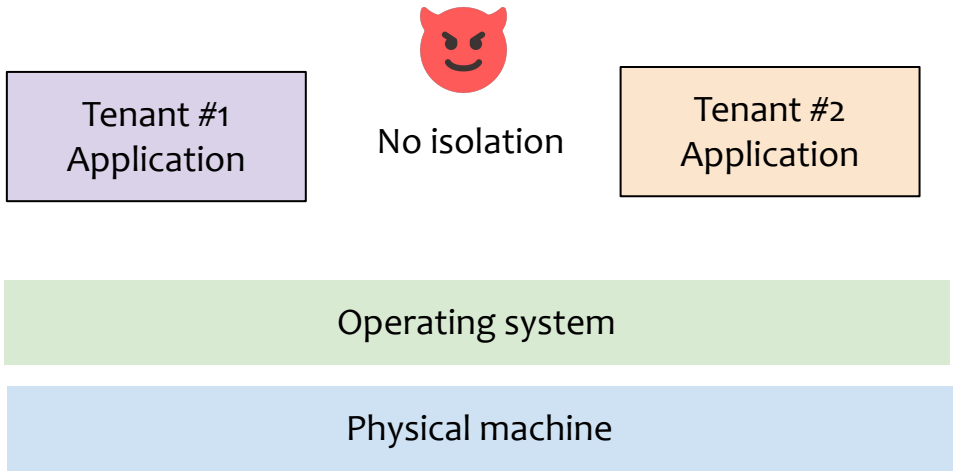
Storage

(Data at rest)

Protecting the data storage on untrusted persistent storage, e.g., disk/SSDs

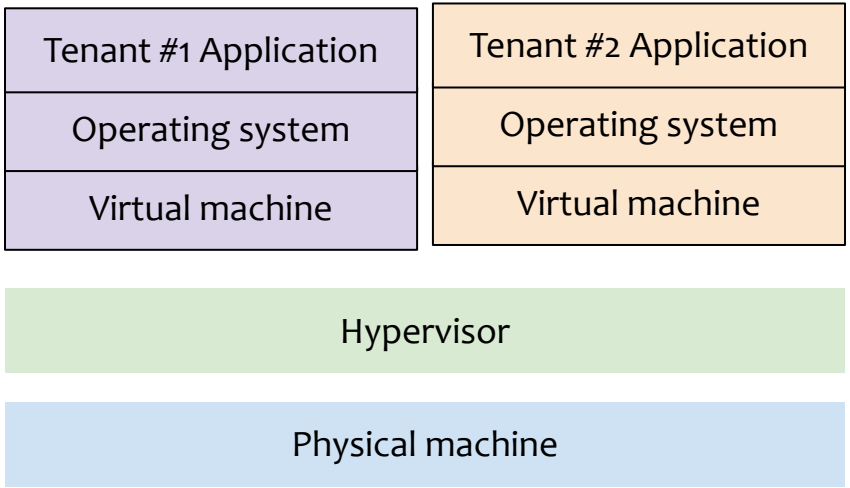
Cloud security for compute

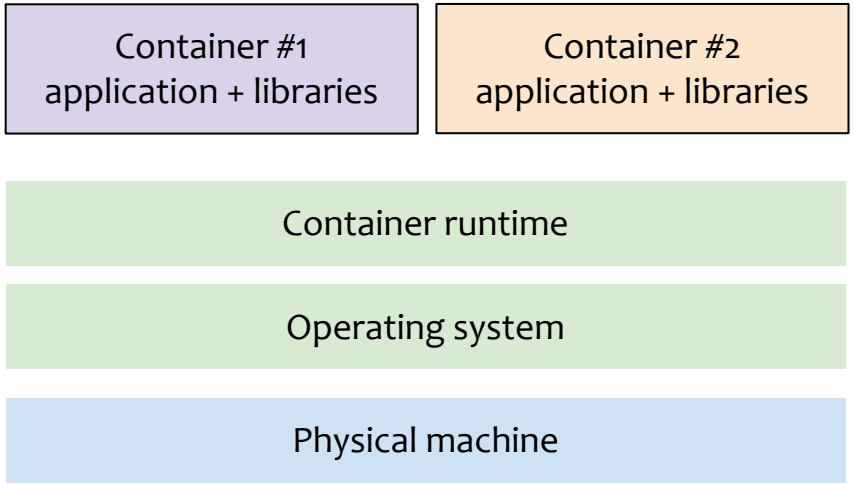
- In the cloud environments, a physical machine is shared among multiple tenants
- The cloud providers use **virtualization** to isolate multiple tenants
  - Virtual machines
  - Containers



https://aws.amazon.com/what-is/virtualization/

Virtualization: Virtual machines vs Containers





Virtual machines:

- Stronger isolation w/ the Hypervisor
- Lower performance

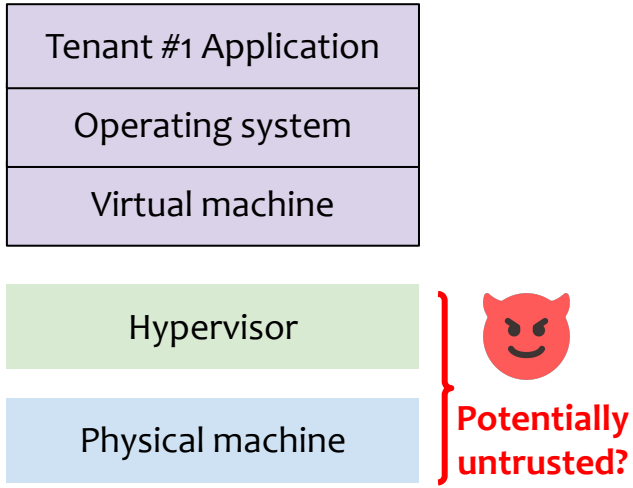
Container:

- Weaker isolation w/ the Operating Systems
- Better performance

https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

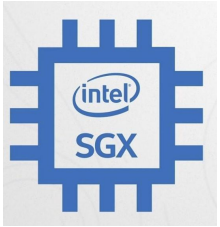
Confidential computing: Advanced threats in the cloud

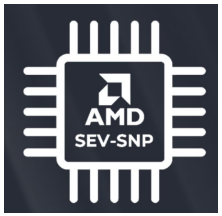
- **Virtualization** *only* isolates/protects from co-located tenants on the same host/machine
  - However, virtualization **does not protect against a "potentially untrusted" cloud provider**
- **Modern cloud computing requires stronger security properties!**
  - What if the cloud provider turns rogue?
  - What if there is a malicious administrator?
  - What if the cloud provider needs to serve a governmental subpoena?
  - What if sensitive use-cases (Banks/medical) require stronger security compliance?



Confidential computing: Protecting computation!

- **Confidential computing** is a cloud computing technology that isolates sensitive data in a protected CPU "enclave" during processing
  - **Even the cloud providers is out of the trusted computing base (Hypervisor)**
  -
- **Hardware assisted trusted computing**
  - New hardware extensions that encrypt/decrypt data during computing





arm
TRUSTZONE

The Arm Confidential Compute Architecture
Arm CCA

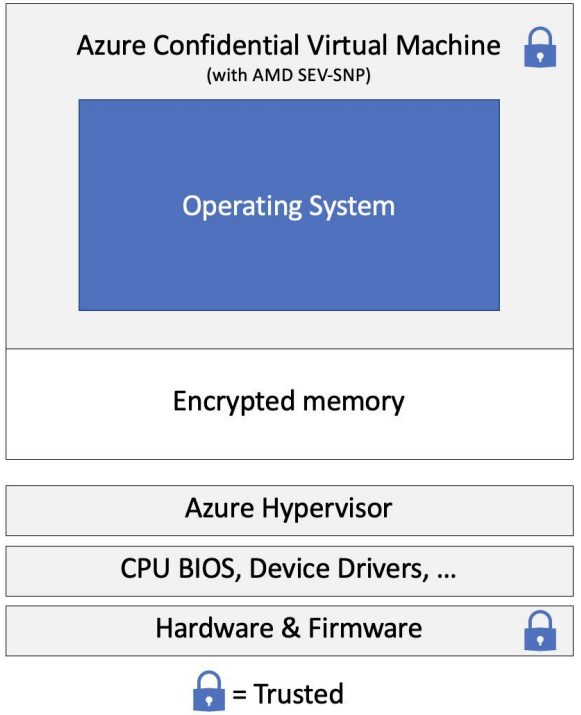
RISC-V Keystone

Intel® Trust Domain Extensions

Confidential computing: https://blogs.nvidia.com/blog/2023/03/01/what-is-confidential-computing/

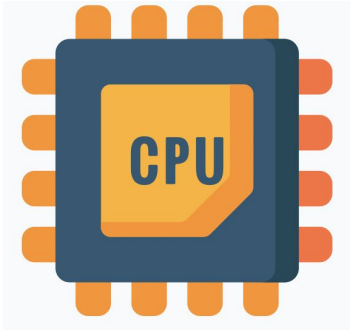
Confidential computing in the cloud

- **Hardware-assisted "secure enclaves"**
  - Keeps the data encrypted in DRAM
  - Special memory encryption engine de/en-crypts data in the cache line
  - Protects the cache lines
- **Confidential VMs:** Full VM encryption technology
  - Isolates the untrusted cloud provider
  - Do not require trusting the underlying cloud infrastructure or hypervisor
- **Commercial offered by cloud providers**
  - Google, Microsoft, Alibaba



Google Cloud Azure

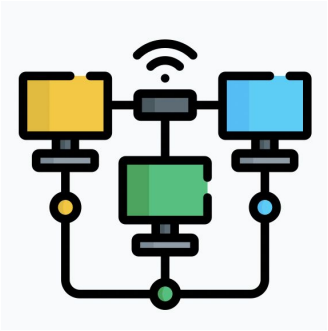
Part A: Secure systems stack in the cloud



Compute

(Data in use)

Protection the code and data, while computing on the CPU, accessing over the RAM



Network

(Data in motion)

Protecting the data being transmitted over the untrusted network connection

Storage

(Data at rest)

Protecting the data storage on untrusted persistent storage, e.g., disk/SSDs

Cloud network security

- **Cloud network security** refers to the security measures—technology, policies, controls, and processes—used to protect public, private, and hybrid cloud networks
- Three primary aspects in network security:
  a. **Services authenticity:** How to establish secure endpoints or to ensure the authenticity of a service?
  b. **Secure communication:** How to ensure confidentiality, integrity, ***and freshness*** of network messages?
     - **Freshness:** Replay attack (re-sending stale messages)
  c. **Services availability:** How to prevent Denial of Service attacks (DoS)?

https://cloud.google.com/learn/what-is-cloud-network-security

#a: Service authenticity

- How does a client authenticate the identity of a server?
  - **Certificate-based authentication**
  -
- Certification-based authentication involves:
  - **Server's certificate**
  - **Certificate authority (CA)**

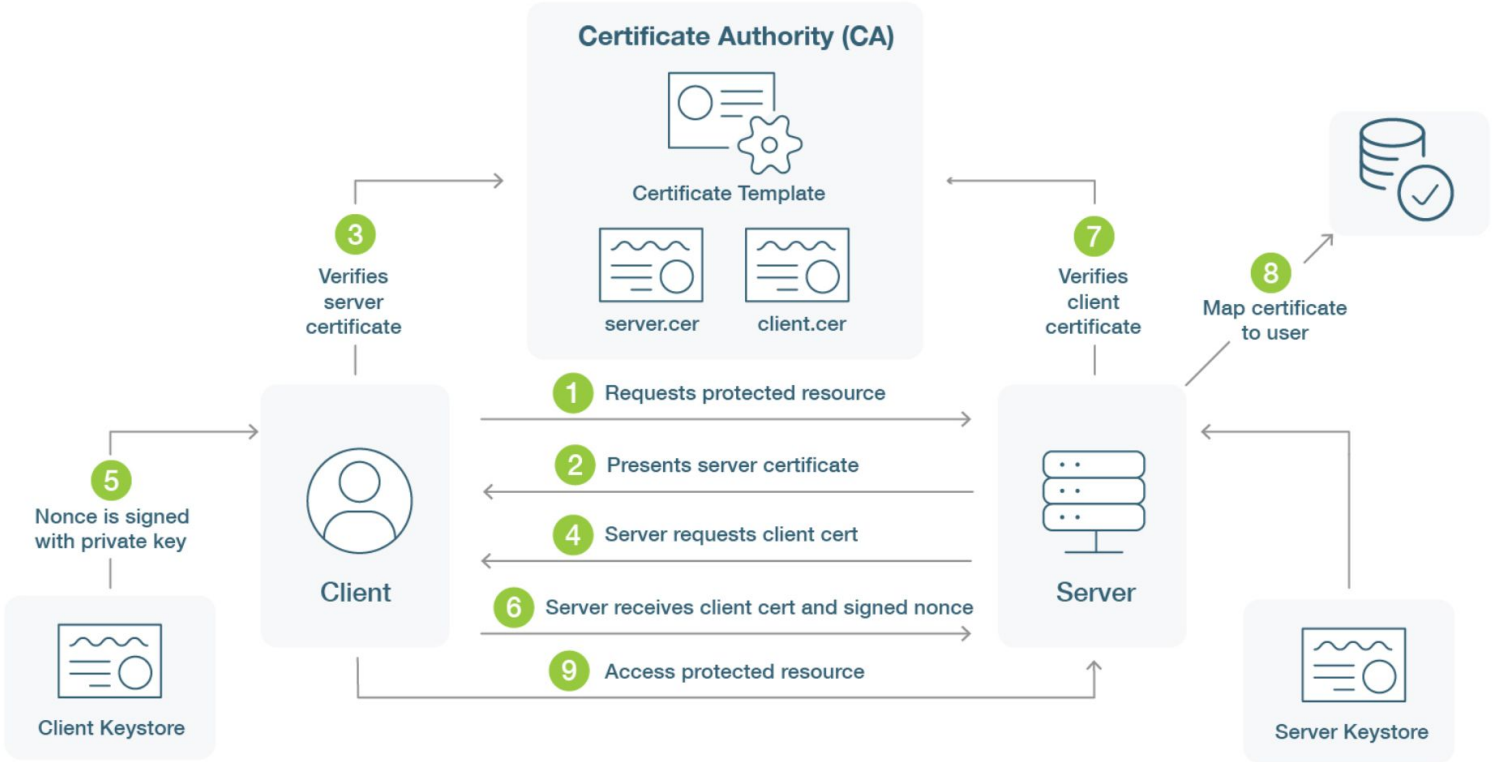
- **Authentication process:** The client verifies the server certificate with the certificate authority that issued it. This confirms that the server is who it says it is, and that the client is interacting with the actual owner of the domain

Key idea: Certificate-based authentication

- **A digital certificate** is an electronic document that is used to identify an individual, a server, a company, or some other entity, and to associate that identity with a public key
  - Like a driver's license, a passport, a student ID, a library card, or other commonly used personal IDs, a certificate provides generally recognized proof of a person's identity

- **Certificate authorities (CAs)** are entities that validate identities and issue certificates
  - Clients and servers use certificates issued by the CA to determine the other certificates that can be trusted

Certificate-based Authentication



https://www.yubico.com/resources/glossary/what-is-certificate-based-authentication/

#b: Secure communication

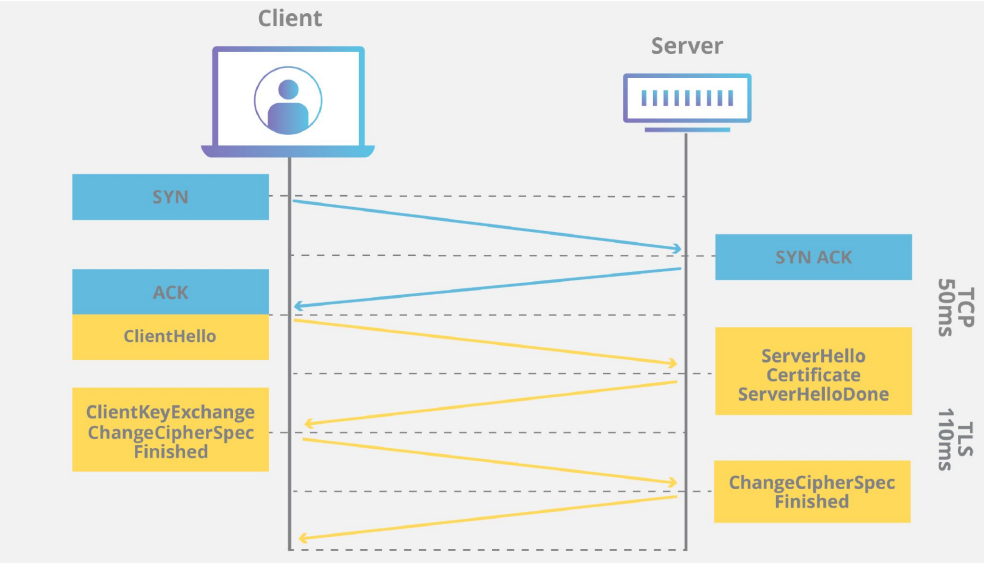
- **Transport layer security (TLS)**, formerly known as SSL, is the most common method of **securing communication of data "in motion"** (over the network)
  - Encrypts traffic over the network
- **TLS should be combined with "Authentication"**
  - TLS is **an encryption and authentication protocol** designed to secure Internet communications

How TLS works?

A TLS handshake is the process that kicks off a communication session between two parties

- **Exchange messages** to acknowledge each other (version of TLS to use)
- **Verify each other** (service authenticity)
- Establish the **cryptographic algorithms they will use**
- **Agree on session keys** for encrypted communication

TLS handshakes are a foundational part of how HTTPS works



https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/

#c: Service availability: DOS attacks

- **A Denial of Service (DoS) attack** is a malicious attempt to affect the availability of a targeted system, such as a website or application, to legitimate end users
  - Typically, attackers generate large volumes of packets or requests ultimately overwhelming the target system
  - **Distributed DOS (DDOS):** In case of a Distributed Denial of Service (DDoS) attack, and the attacker uses multiple compromised or controlled sources to generate the attack

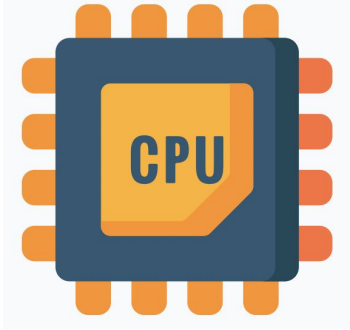
DOS protection: https://aws.amazon.com/shield/ddos-attack-protection/

How to thwart DOS attacks?

- Minimize the attack surface with a **trusted gateway**
  - Profiles the incoming requests (IP sources, credentials, etc.) and employs ACLs/security policies to prune invalid/malicious requests

CloudFlare's approach for DOS protection:
https://support.cloudflare.com/hc/en-us/sections/360007347692

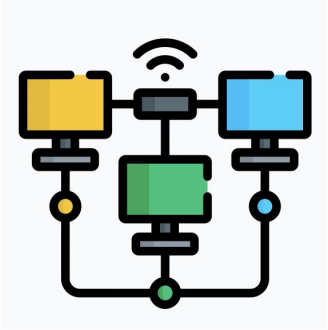
Part A: Secure systems stack in the cloud



Compute

(Data in use)

Protection the code and data, while computing on the CPU, accessing over the RAM



Network

(Data in motion)

Protecting the data being transmitted over the untrusted network connection

Storage

(Data at rest)

Protecting the data storage on untrusted persistent storage, e.g., disk/SSDs

Cloud security for storage

- **Stateful applications rely on storage systems** for data management
- **Storage security** primarily deals
  - **Confidentiality:** Encryption of data
  - **Integrity:** Compute MAC (hash signature) and validate any tampering of the data
  - **Freshness:** Relies on a "trusted counter" from the hardware to provide monotonically increasing state of storage system (or prevent rollback attacks)
- **Cloud providers** provide mechanisms for encrypting data in the cloud, while maintaining MAC in a secure vault for integrity:
  - For example, Key Management Services (KM)S provides APIs for encrypting the data https://cloud.google.com/kms/docs/encrypting-application-data

Design of secure systems in the cloud

- **Secure systems stack**
  - Computing
  - Network
  - Storage
- **Authentication, key management, and attestation**
  - Identity and access management (IAM)
  - Key management service (KMS)
  - Remote attestation

Identity and access management (IAM)

- **Identity and access management (IAM)**
  - IAM authorizes who can take action on specific resources in the cloud
- **A lot of powerful features**
  - Single access control interface
  - Fine-grained control
  - Automated access control recommendations
  - Built-in audit trail
  - Web, programmatic, and command-line access

Google IAM https://cloud.google.com/iam
Amazon IAM https://aws.amazon.com/iam/

Key management service (KMS)

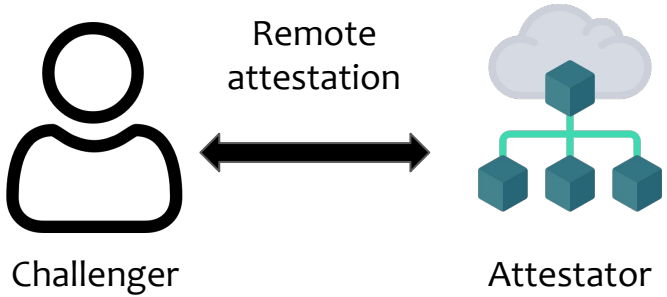
- **Key management service (KMS):**
  - **Cloud KMS is a cloud-hosted key management service** that lets you manage cryptographic keys for your cloud services
  - They also **provide support for encryption, decryption, signing, and verification** using a variety of key types and sources
  - These **KMS securely manage keys using HSM (Hardware Security Modules)**
  - **KMS offers REST APIs** to manage keys, and also encrypt and decrypt data using specific keys

Key Management Service

Amazon AWS KMS: https://aws.amazon.com/kms/faqs/
Google KMS: https://cloud.google.com/security-key-management

Remote attestation

- **Remote attestation** is a method by which a client (e.g., a cloud user) ensures the trustworthiness of the hardware and software configuration of a remote server (e.g., a server in the cloud)
- Remote attestation is based on
  - **Hardware support for attestation**
  - **A trusted boot mechanism**
  - **A remote attestation protocol:** a security protocol in which a trusted party (using a trusted hardware) assures the integrity of the remote machine



Remote attestation in the cloud:
https://patterns.arcitura.com/cloud-computing-patterns/mechanisms/attestation_service
Bootstrapping Trust in Commodity Computers:
https://www.andrew.cmu.edu/user/bparno/papers/bootstrapping-sok.pdf

References

- Software Security: Principles, Policies, and Protection (SS3P, by Mathias Payer)
  - https://nebelwelt.net/SS3P/
  - **Chapter 2:** Software and System Security Principles
- CSE 127: Intro to Computer Security (UCSD)
  - https://cseweb.ucsd.edu/classes/wi22/cse127-a/

Outline

- ~~**Part I:** Security~~
- **Part II: Reliability and availability**
  - Terminology: System failures, fault types and sources, and properties and metrics
  - Single-node reliable systems and associated issues
  - Replication as the general recipe for fault-tolerance
  - Fault-tolerance for stateful services
- **Part III:** Pattern implementation

Faults

- **Faults** are deviation from the expected behavior
- **Faults happen** due to a variety of factors:
  - Hardware failure
  - Software bugs
  - Operator errors/mis-configurations
  - Network errors/outages
- **Faults are the norm, not an exception! –** At scale, fault happen very regularly
  - Let's assume, the failure probability of a hardware component (e.g., disk, memory) is one failure per year
  - If you build your system with 10,000 such components, what's the failure rate?
    - Failure rate of the system = (1/365) * 10,000 = 27.4 faults per day!

We need to design fault-tolerant systems!

Faults

- **Two types**
  - **Transient faults** → E.g., bit flips
  - **Permanent faults**

- **A fault can lead to**
  - **Fail-stop (Crash faults)** → Process crashes
  - **Byzantine faults** → Arbitrary behavior, e.g., data corruption, security attacks
    - Can be exploited by an attacker leading to security vulnerabilities

Fault-tolerant systems

- **A fault tolerant system** is **reliable** and **available** in the presence of component or subsystem failures

  - **Reliable:** The service works as per the specs/protocols

  - **Available:** The service is available when required

Fault tolerance metrics (Recap from L02)

Reliability metrics: common failure metrics that get measured and tracked for any system

- **Mean time between failures (MTBF)**
- **Mean time to failure (MTTF):**
- **Mean time to repair (MTTR):**

Availability metrics: We measure availability through the percentage of time that a service is guaranteed to be available for use in a year. For e.g., usually referred to as "9s"

- **99.99% (four nines)**
- **99.999% (five nines)**

Roadmap: How to make systems fault-tolerant?

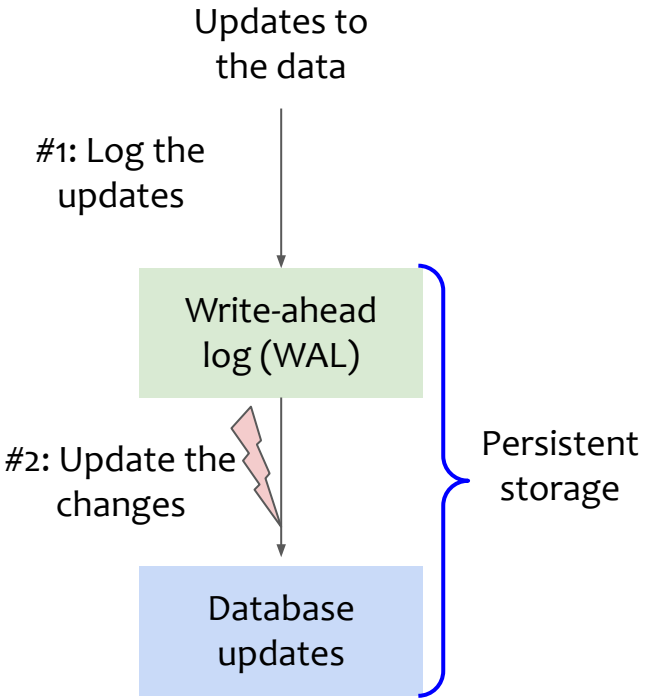
1. Single-node fault-tolerant systems
   - Write-ahead logging for system reliability
   - Issues with a single-node fault tolerance approach
2. Replication as the general recipe
   - Replication for stateless services
   - Issues with replication for stateful services
3. Replication for stateful services
   - Primary-backup replication
   - State machine replication

#1: Single-node fault tolerant systems

- **Example:** A banking service, small enough to "fit" on a single machine
- **Task:** Reliable transaction processing to fulfill the **ACID properties**
- **Approach:**
  - **Write-ahead logging:** First record the changes in a log, and then apply the changes to the database
  - **Recovery:** In case of system failure during the database update, undo/redo the changes based on the log

Write-ahead logging

- **Write-ahead logging** is a standard way to ensure data integrity and reliability
  - **Any changes made on the database are first logged in an append-only file** called write-ahead Log or Commit Log
  - Thereafter, **the actual blocks having the data (row, document) on the persistent storage are updated**
  - **System recovery** relies on **"replaying the logs"** for uncommitted operations after a crash on reboot
- **Types of logs**
  - **Undo log** allows a database to undo a transaction
  - **Redo log** allows to redo a transaction



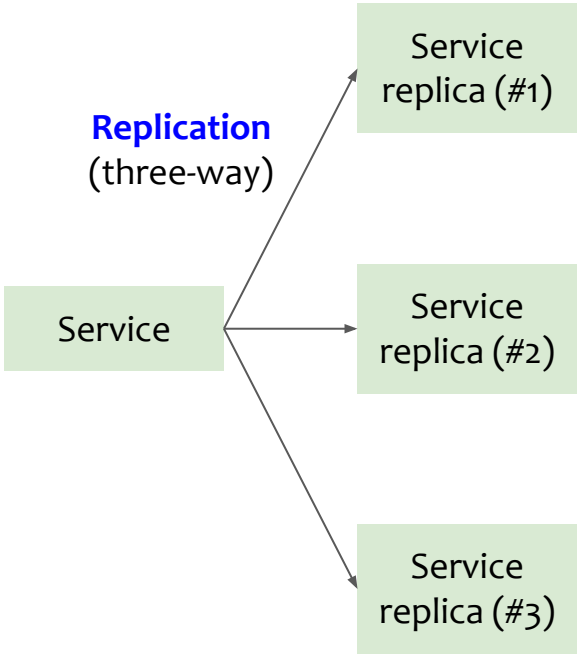
https://blog.cykerway.com/posts/2018/11/18/database-undo-log-and-redo-log.html

Issues with the logging-only approach

- Write-ahead logging is not sufficient for fault-tolerance:
  - **Availability:** The systems is not available during the recovery process
    - The system needs to reboot and replay the logs after the failure
  - **Reliability:** The systems is not completely reliable,
    - For example, the hardware/disk of the single machine can fail permanently

#2: The general recipe – Replication

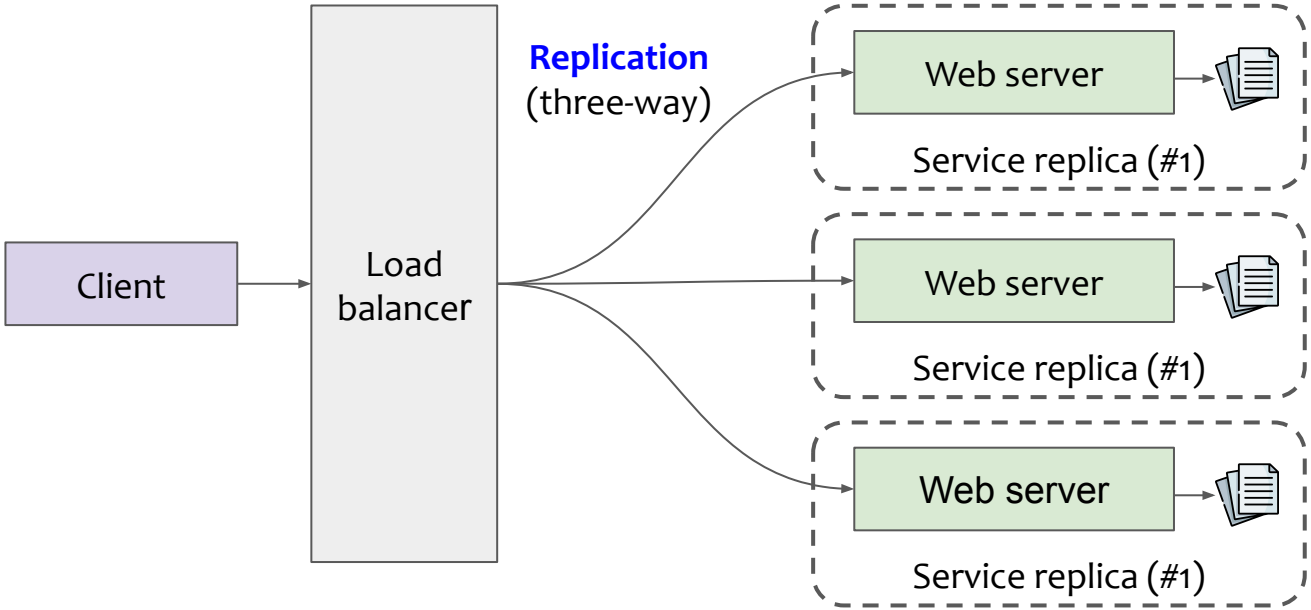
- **Replication** is the primary basis for a fault-tolerant system design
  - **Keeping the application state across multiple machines** on several different nodes, potentially different locations/data centers
  - **Replication provides redundancy:** if some nodes are unavailable, the application can still function from the remaining nodes
  - **A side advantage:** Replication can also help **improve performance** (primarily for read-only workloads)



Replicated system: Stateless services

Stateless application don't maintain a state, e.g., serving web requests for static files

Any machine can handle any request => load balancer forwards requests to any server



Each machine has a copy of the files that it is serving. As long as the files don't change, it's easy to add more web server machines.

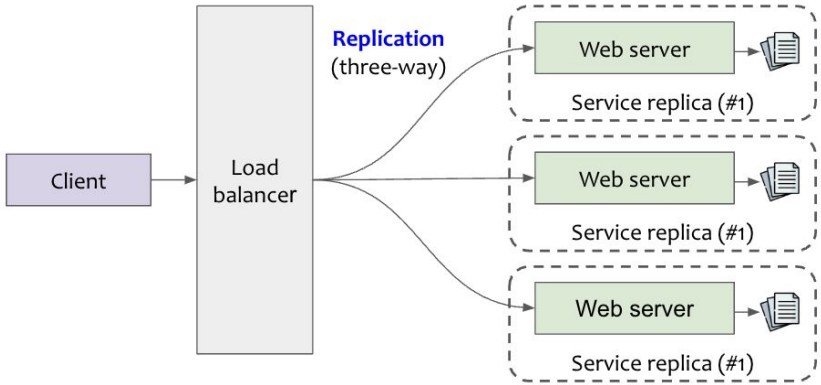
Use e.g. Kubernetes for managing servers.

kubernetes

https://ably.com/blog/engineering-dependability-and-fault-tolerance-in-a-distributed-system

Advantages of replicating stateless services

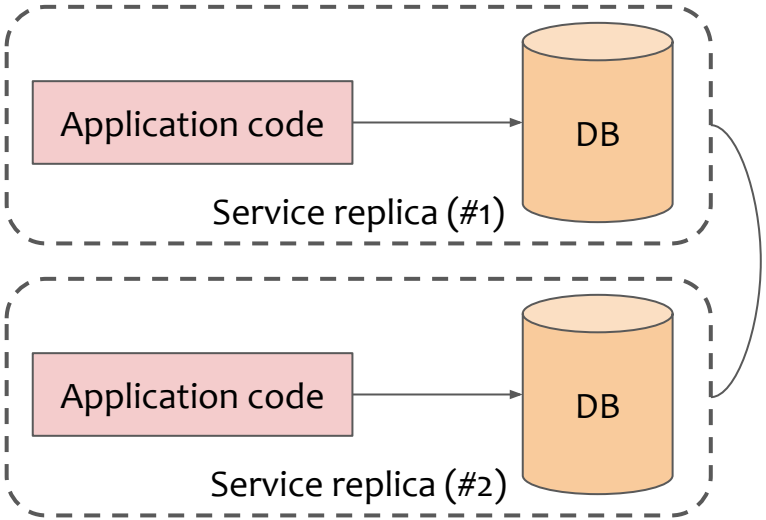
- **Fault tolerance**
  - High availability
  - High reliability
- **Performance**
  - Read request can be handled by any replica independently
- **Scalability**
  - The system scales linearly with addition of more replicas



Issues with replicating stateful services

A stateful service

employing a database for storing its state/data



Challenge:

How to keep the state consistent across replicas?

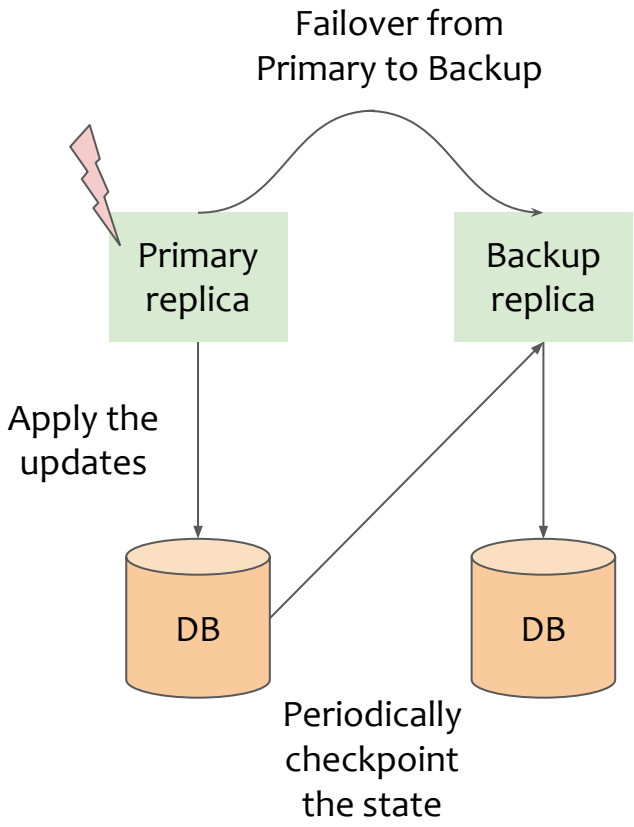
#3: Replication for stateful services

- Each replica stores the copy of the entire state (for fault-tolerance), and a state replication protocol ensures that the replicas are consistent
  - The protocol ensures that every write to the state/data is processed by every replicas, otherwise the replicas would no longer contain the same data
- Two prominent approaches
  - **Primary-backup replication**
    - Aka passive replication, or active-passive replication
    - Based on **replicating the state (or side-effects)**
  - **State machine replication**
    - Aka active replication
    - Based on **replicating the operations**

Primary-backup replication

Primary-backup replication involves having a primary replica (also called the leader) and one or more backup replicas (also called followers)

- **Single leader (active):** The primary replica processes all client requests and updates its state accordingly
- **Backup replicas (passive):** Backup replicas maintain a copy of the primary replica's state, which they periodically synchronize with the leader to ensure consistency
- **Failover mechanism:** If the primary replica fails, one of the backup replicas is elected as the new primary replica to continue serving client requests
-



Primary-backup replication

Advantage:

- **Simplicity:** Relatively straightforward to implement

Disadvantages:

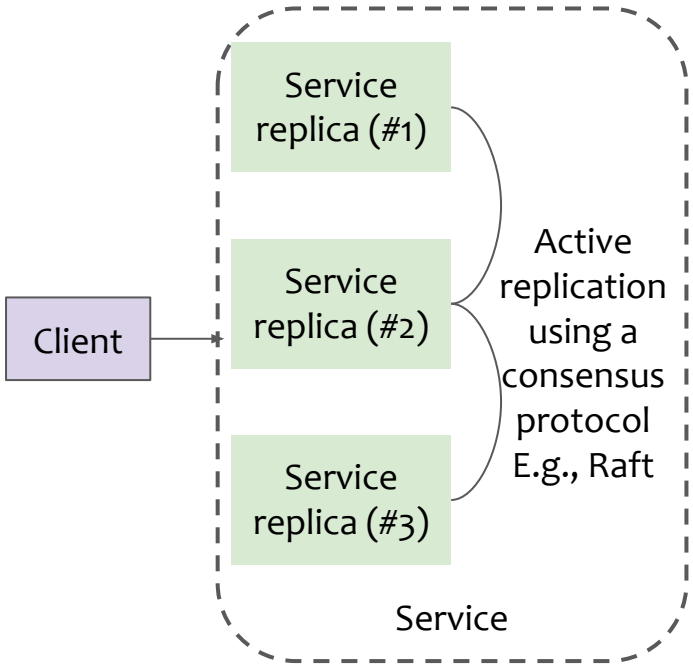
- **Limited scalability:** The system's throughput is limited by the capacity of the primary replica
- **Increased latency:** If the primary replica fails, failover introduces a delay until a new primary replica is elected and “pending” state is applied
- **State loss:** The “un-checkpointed” state can be lost!

State machine replication

State machine replication (SMR) is based on replicating operations instead of replicating the states. Each replica runs the same set of operations in the same order, ensuring that they all arrive at the same state.

Ingredients for **state machine replication**:

- Implement service as a **deterministic state machine**
- **Replicate:** Replicas coordinate using a consensus protocol, such as Paxos or Raft, to agree on the order of operations
- Provide **all replicas with the same input**



Implementing fault-tolerant services using the state machine approach: a tutorial https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf
RAFT protocol: https://raft.github.io/

State machine replication

Advantages:

- **Scalability:** State machine replication can be more scalable than primary-backup replication because multiple replicas can process client requests concurrently
- **Fault tolerance:** The system can tolerate the failure of some replicas as long as a majority remains operational with no/minimal fail-over time

Disadvantages:

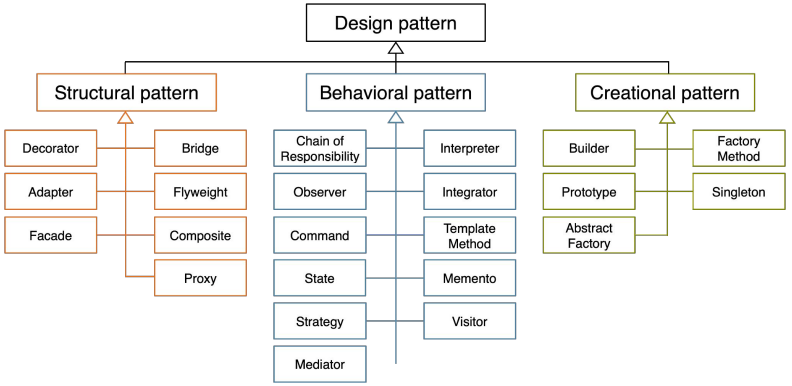
- **Complexity:** Implementing state machine replication requires more intricate coordination mechanisms and consensus protocols

Outline

- ~~**Part I:** Security~~
- ~~**Part II:** Reliability and availability~~
- **Part III: Pattern implementation**
  - Adapter pattern
  - Observer pattern
  - Strategy pattern

Pattern implementation

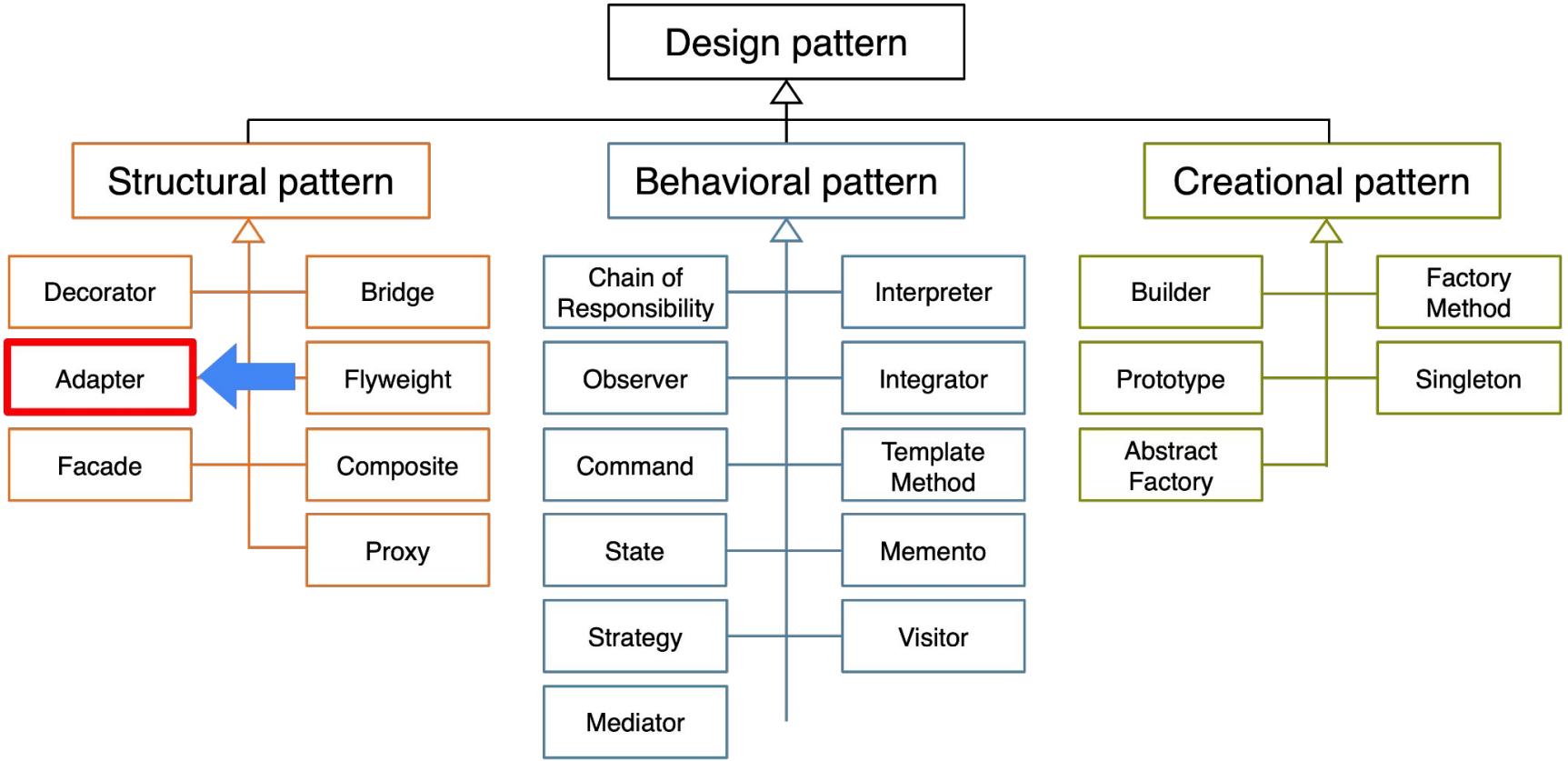
- **What are design patterns?**
  - Reusable solutions to common problems in software design
  - Best practices that can be adapted to specific situations
  - Improve code maintainability, flexibility, and extensibility
- **Design patterns taxonomy:**
  - **Structural:** Concerned with the composition of classes and objects
  - **Behavioural:** Define the ways objects interact and communicate with one another
  - **Creational:** Deal with the process of object creation



Outline

- ~~**Part I:** Security~~
- ~~**Part II:** Reliability and availability~~
- **Part III: Pattern implementation**
  - **Adapter pattern**
  - Observer pattern
  - Strategy pattern

Design patterns taxonomy



Adapter pattern

- **Problem**: An existing component offers functionality, but is incompatible with the new system being developed
- **Solution**: The adapter pattern connects incompatible components
  - Allows the reuse of existing components
  - Converts the interface of the existing component into another interface expected by the calling component
  - Useful in **interface engineering** projects and in **reengineering** projects
  - Often used to provide a new interface for a legacy system

→ Also called a **wrapper**

Example1: Accessing a power charger

- **Scenario:** Evgeny is using a phone that requires power
- **Problem:** Evgeny's phone battery is empty, he has access to a US Charger that offers 110 Volt charging
- **Challenge:** provide power to the US Charger in Germany

Example: Accessing a power charger

- **Solution:** Provide the voltage adapter to deliver the correct voltage

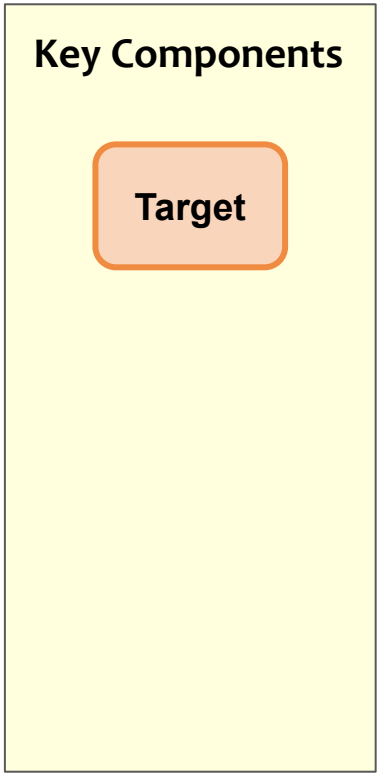
charge()

provide230()

provide110()

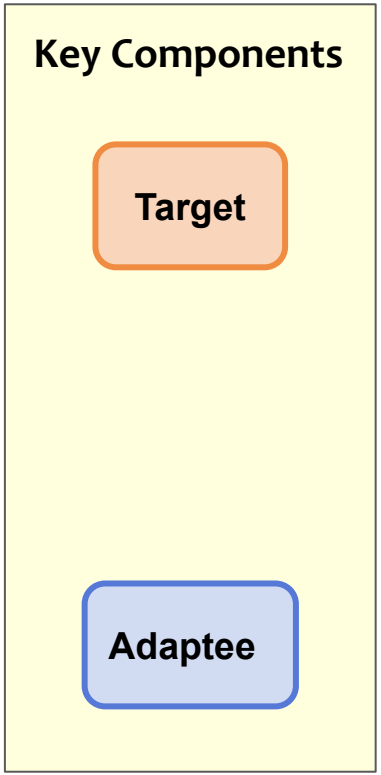
Adapter pattern: Key components

- **Target:** The desired interface that the client wants to use
  - Defines the methods that the client will call



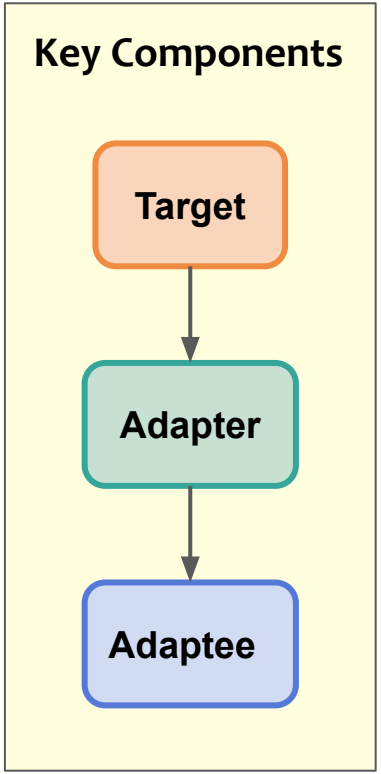
Adapter pattern: Key components

- **Target:** The desired interface that the client wants to use
  - Defines the methods that the client will call
- **Adaptee:** The existing interface that needs to be adapted
  - Provides the functionality that needs to be used by the client, but its interface is incompatible with the Target



Adapter pattern: Key components

- **Target:** The desired interface that the client wants to use
  - Defines the methods that the client will call
- **Adaptee:** The existing interface that needs to be adapted
  - Provides the functionality that needs to be used by the client, but its interface is incompatible with the Target
- **Adapter:** The class that converts the Adaptee's interface into the Target's interface
  - Implements the Target interface and holds a reference to the Adaptee, making it possible to call the Adaptee's methods using the Target's interface



Adapter Pattern

- **Benefits**
  - **Increased flexibility:** Adapter allows classes to work together even with incompatible interfaces
  - **Improved reusability:** Adaptee's functionalities can be reused without changing its existing code
  - **Easier maintenance:** Changes in the Adaptee do not affect the client code using the Target interface
- **Use Cases**
  - When you want to use an existing class with an incompatible interface
  - When you need to create a reusable class that cooperates with unrelated or unforeseen classes
  - When you need to provide a uniform interface to different APIs or libraries

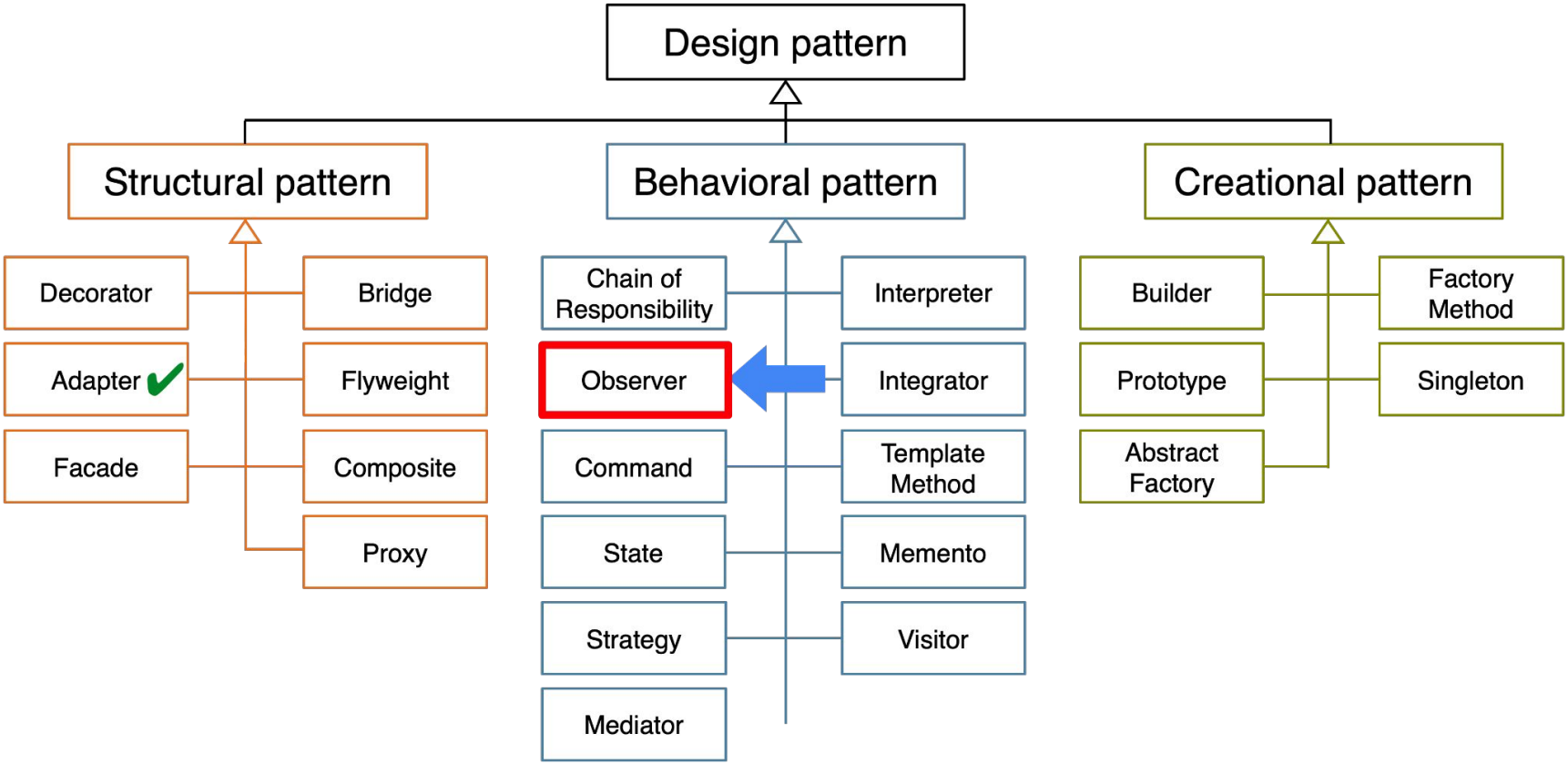
References for Adapter Pattern

- **Design Pattern Taxonomy:** Structural design pattern
- **Key Components:**
  - *Target:* The desired interface for the client
  - *Adaptee:* The existing, incompatible interface
  - *Adapter:* The class that converts the Adaptee's interface into the Target's interface
- **Purpose:** Interface conversion
- **Benefits:** Flexibility, reusability, and easier maintenance
- **References:**
  - https://refactoring.guru/design-patterns/adapter
  - https://sourcemaking.com/design_patterns/adapter
  - https://www.geeksforgeeks.org/adapter-pattern/
  - https://en.wikipedia.org/wiki/Adapter_pattern

Outline

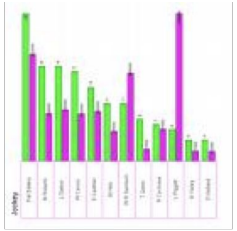
- ~~**Part I:** Security~~
- ~~**Part II:** Reliability and availability~~
- **Part III: Pattern implementation**
  - ~~Adapter pattern~~
  - **Observer pattern**
  - Strategy pattern

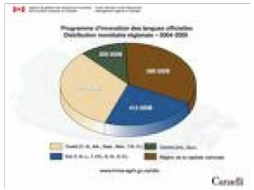
Design patterns taxonomy



Observer pattern

- **Problem:**
  - An object that changes its **state** often, e.g., a portfolio of stocks
  - Multiple views of the current state, e.g., histogram view, pie chart view, timeline view
- **Requirements**
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - The system design should be highly extensible
  - It should be possible to add new views - for example, an alarm - without having to recompile the observed object or existing views





Observer pattern

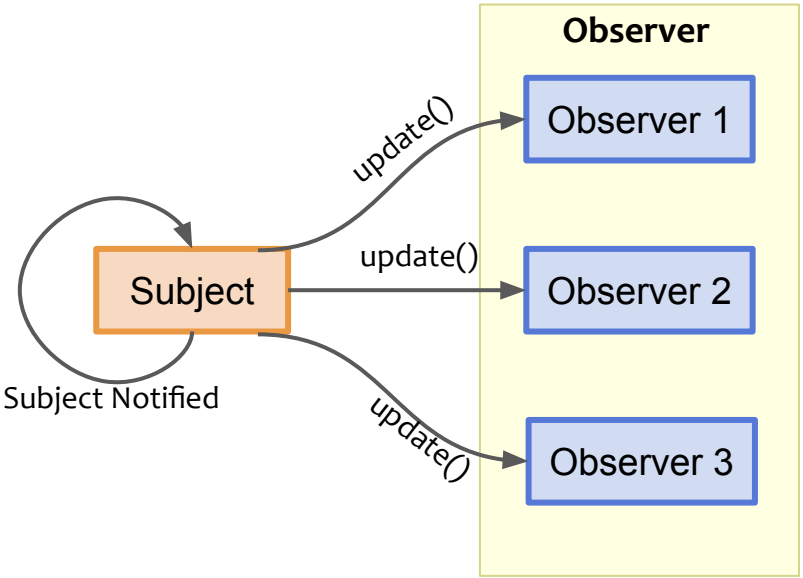
- **Solution:** model a one-to-many dependency between objects
  - Connect the state of an observed object - **subject** with many observing objects - the **observers**
  - When the state of the **subject** changes, all its dependents (**observers**) are automatically notified and updated.

→ Also called **Publish and Subscribe**

  - The subject acts as the publisher, while the observers are the subscribers who receive notifications.

Observer pattern: Key components

- **Subject:** The object that has one-to-many dependencies with other objects (observers)
  - Maintains a list of observers and provides methods to add, remove, and notify observers
- **Observer:** An interface that defines the methods to be implemented by objects that need to be notified when the Subject's state changes
  - Specifies the methods to update the observer's state when notified by the Subject



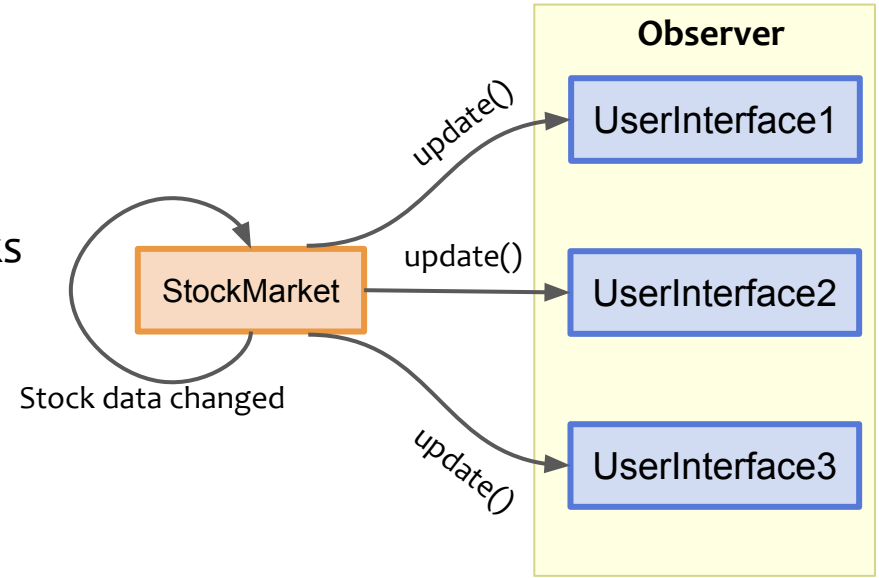
Variants of the observer pattern

3 variants for maintaining the consistency

1. **Pull Model:** Observer requests state from Subject (MVC: View requests updates from Model)
2. **Push Model:** Subject automatically sends updates to Observer (MVC: Model sends updates to View)
3. **Hybrid Model:** Observer can use either Pull or Push (MVC: Model provides both Pull and Push updates to View)

Example: A stock market application

- **Scenario:** A stock market application that displays stock prices for various stocks to multiple users
- **Problem:** When stock prices change, all users should receive updates for the stocks they are interested in
- **Challenge:** Update users in real-time without tightly coupling the stock data source to the individual user interfaces



Observer pattern

- **Benefits**
  - Maintain consistency across redundant observers
  - Optimize a batch of changes to maintain consistency
  - Promotes loose coupling between the subject and its observers
- **Use Cases**
  - When a change in one object requires updating other objects, and the number of objects is unknown or can change
  - When an object should be able to notify other objects without making assumptions about who those objects are or what they do

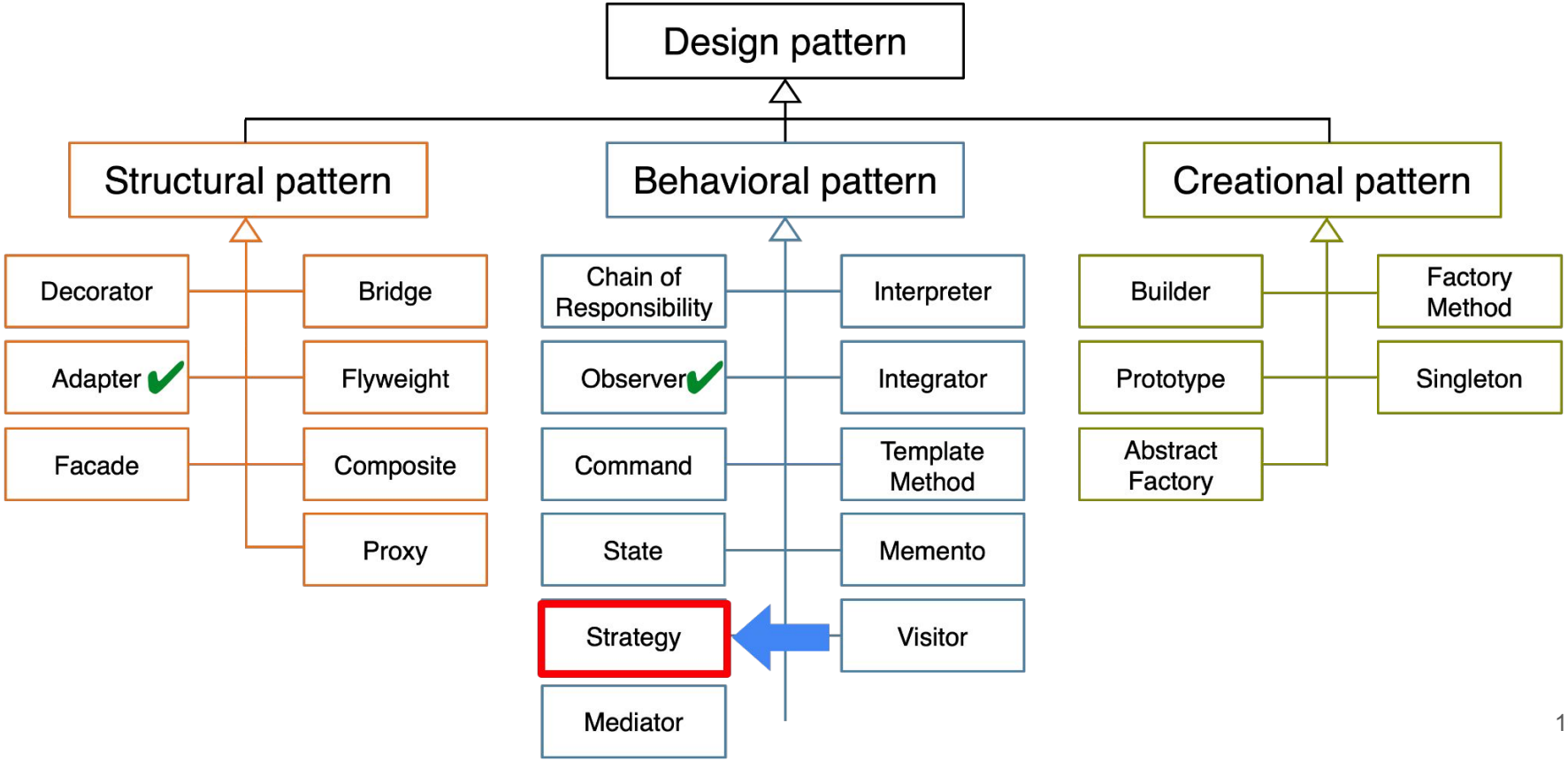
References for Observer Pattern

- **Design Pattern Taxonomy:** Behavioural design pattern
- **Key Components:** Subject, Observer
- **Purpose:** Manage one-to-many dependencies between objects
- **Benefits:** Loose coupling, dynamic observer management, simultaneous notifications
- **References:**
  - https://refactoring.guru/design-patterns/observer
  - https://sourcemaking.com/design_patterns/observer
  - https://www.javadesignpatterns.com/patterns/observer/
  - https://www.geeksforgeeks.org/observer-pattern-in-java/

Outline

- ~~**Part I:** Security~~
- ~~**Part II:** Reliability and availability~~
- **Part III: Pattern implementation**
  - ~~Adapter pattern~~
  - ~~Observer pattern~~
  - **Strategy pattern**

Design patterns taxonomy

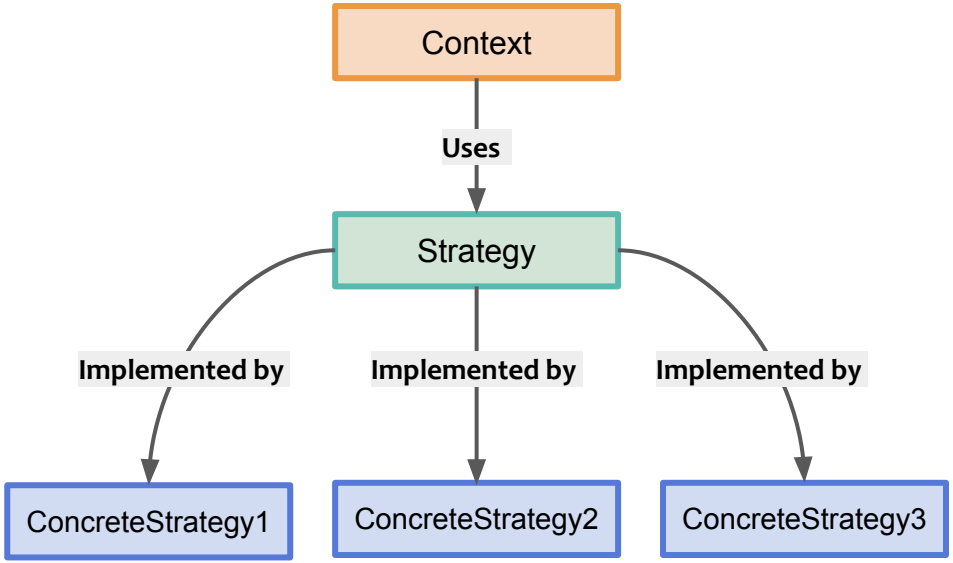


Strategy pattern

- **Problem:** Different algorithms exist for a specific task
- **Requirement:**
  - If we need a new algorithm, we want to add it without changing the rest of the application or the other algorithms
- **Solution:** the strategy pattern allows switching between different algorithms at run time based on the context and a policy

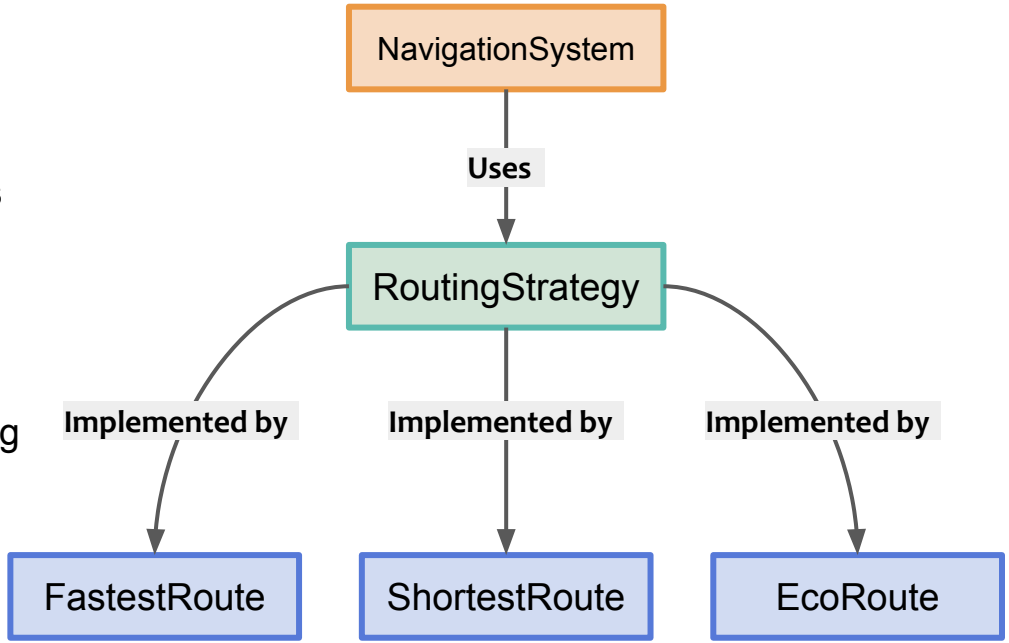
Strategy pattern: Key components

- **Context:** The class that uses a Strategy to perform an operation
  - Contains a reference to a Strategy object and delegates the execution of the algorithm to it
- **Strategy:** An interface that defines the methods to be implemented by different algorithms
  - Specifies the methods to execute the algorithm
- **Concrete Strategies:** Classes that implement the Strategy interface and define a specific algorithm
  - Provides a specific implementation of the algorithm



Example: Navigation routing options

- **Scenario:** A navigation application that provides different routing options based on user preferences
- **Problem:** Users may choose different types of routes: fastest, shortest, or most fuel-efficient
- **Challenge:** Implement a system that can easily switch between routing algorithms based on user preferences without modifying the core navigation code



Strategy pattern

- **Benefits**
  - Encourages separation of concerns and code reusability
  - Allows adding new algorithms without modifying the Context or existing algorithms
  - Supports selecting algorithms at runtime
- **Use-cases**
  - When different variations of an algorithm are required
  - When a class has a behaviour that is implemented by multiple algorithms and can be switched during runtime

References for strategy pattern

- **Design pattern taxonomy:** Behavioural design pattern
- **Key components:** Context, Strategy, Concrete Strategies
- **Purpose:** Encapsulate algorithms and make them interchangeable
- **Benefits:** Separation of concerns, code reusability, runtime algorithm selection
- **References:**
  - https://refactoring.guru/design-patterns/strategy
  - https://www.javadesignpatterns.com/patterns/strategy/
  - https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Clues for the use of design patterns

- **For Adapter Pattern:**
  - Need to convert the interface of a class into another interface, clients expect
  - Want to reuse an existing class, but its interface is incompatible with the rest of the system
  - Need to provide a unified interface to a set of classes with diverse interfaces
- **For Observer Pattern:**
  - Need to establish a one-to-many relationship between objects
  - Need to update a set of dependent objects automatically when an object's state changes
  - Want to decouple a subject from its observers, promoting loose coupling
- **For Strategy Pattern:**
  - Need to define a family of algorithms and make them interchangeable
  - Want to provide a way to select an algorithm at runtime
  - Need to encapsulate algorithms and avoid code duplication

References

- Design Patterns. Elements of Reusable Object-Oriented Software – Gamma, Helm, Johnson & Vlissides
- Pattern-Oriented Software Architecture, Volume 1, A System of Patterns - Buschmann, Meunier, Rohnert, Sommerlad, Stal
- Pattern-Oriented Analysis and Design - Composing Patterns to Design Software Systems - Yacoub & Ammar
- https://sourcemaking.com

Summary

- **Part I:** Security
  - Security engineering
  - Software security in the cloud
- **Part II:** Reliability and availability
  - Single-node reliable systems and associated issues
  - Replication as the general recipe for fault-tolerance
  - Fault-tolerance for stateful services
- **Part III:** Pattern implementation
  - Adapter pattern
  - Observer pattern
  - Strategy pattern