# Cloud Computing Applications and Services

(Aplicações e Serviços de Computação em Nuvem)

## Distributed Applications

University of Minho

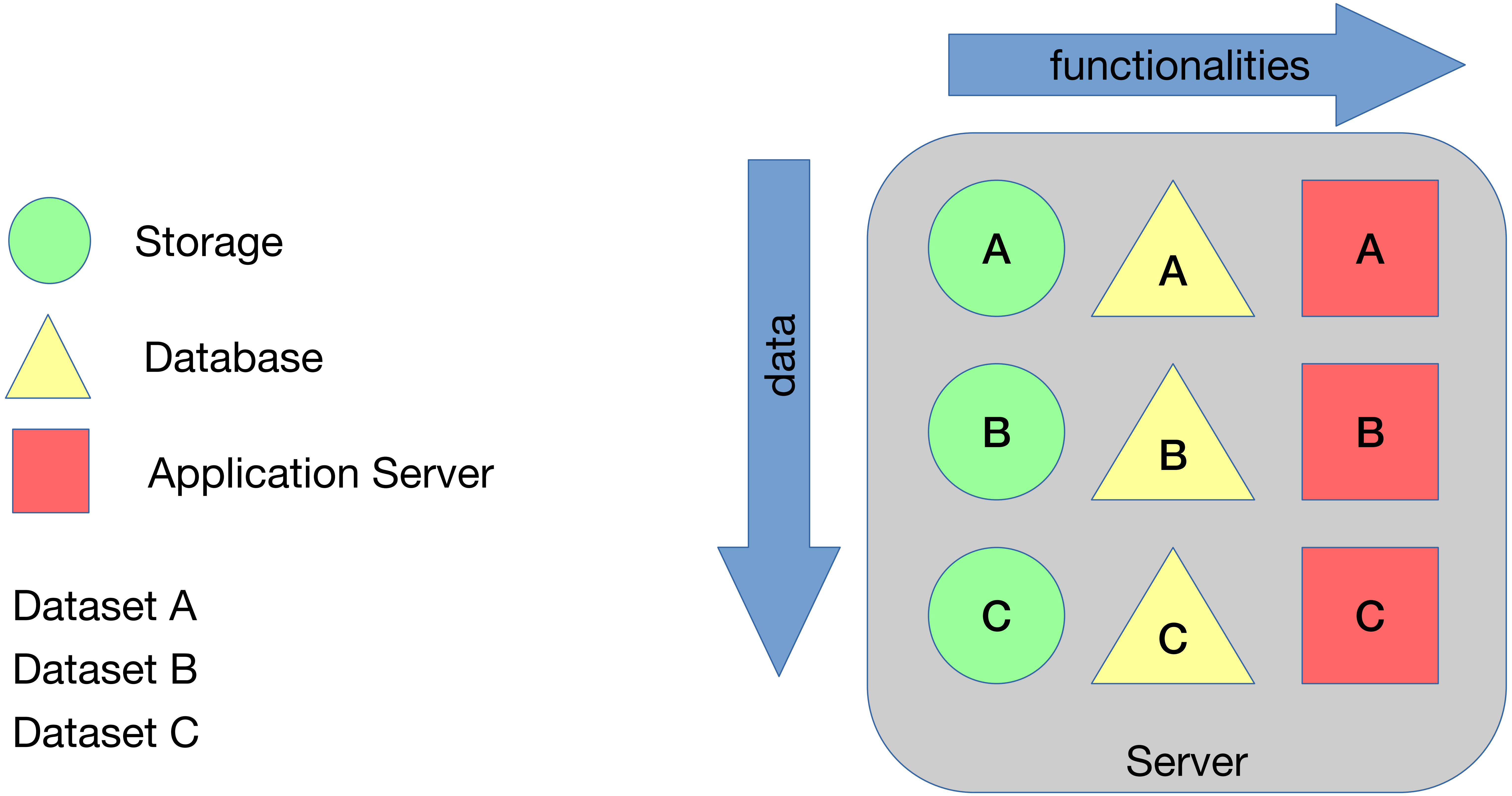
2024-2025

# Motivation and Goals

- Main concerns: **why distributed systems?**
  - **Modularity**, decoupling different components (concerns) (e.g., storage, database, application server, web server)
  - **Performance / Scalability** (more servers doing the work means more speed!)
  - **Availability / Dependability** (hardware and software often fail!)
- Main patterns, mechanisms and architectures: **how to distribute?**

# Monolithic system

- Multiple components (i.e., storage, database, application server) serving multiple targets (e.g., data from different clients) in the same server



# Challenges

- **Question:** Any ideas on how to solve these two challenges?
  - What if the server cannot handle the load imposed by clients?
  - What if the server fails?

# Challenges

- **Question:** Any ideas on how to solve these two challenges?
  - What if the server cannot handle the load imposed by clients?
  - What if the server fails?

- What if the server cannot handle the load imposed by clients?
  - **Scale-up:** increase server resources
  - **Scale-out:** increase number of servers

- What if the server fails?
  - **Redundancy:** have redundant servers

# Distributed system

- Main distribution patterns/mechanisms:
  - **Replication**
  - **Partitioning**
  - **Service-orientation**
- All of these address the **scale-out** of a service/application
- Replication also provides **redundancy**!
- Not mutually exclusive, can be combined

# Replication

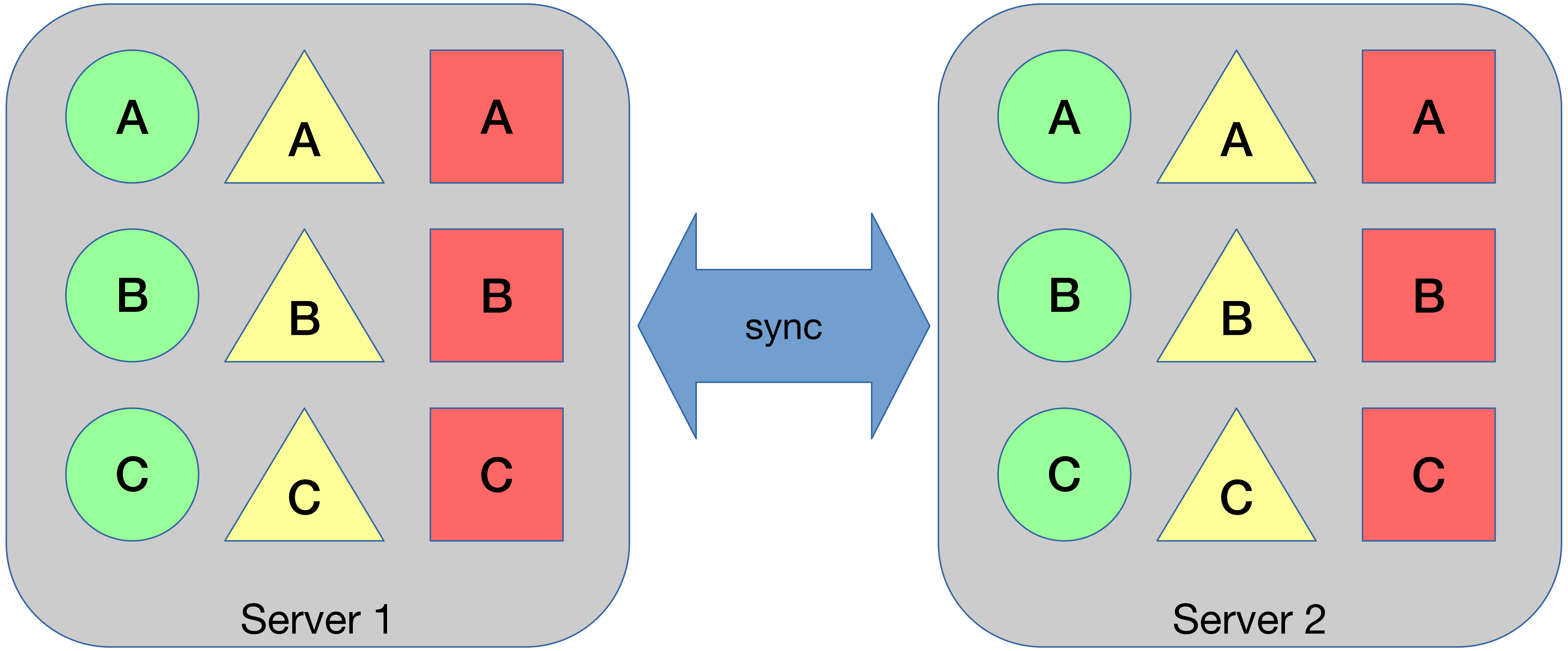
- Multiple copies of the same data and/or functionality
- Addresses **dependability** (i.e., if Server 1 fails, Server 2 does its work) and **scale-out** (i.e., user requests balanced across Servers 1 and 2)

🟢 Storage

△ Database

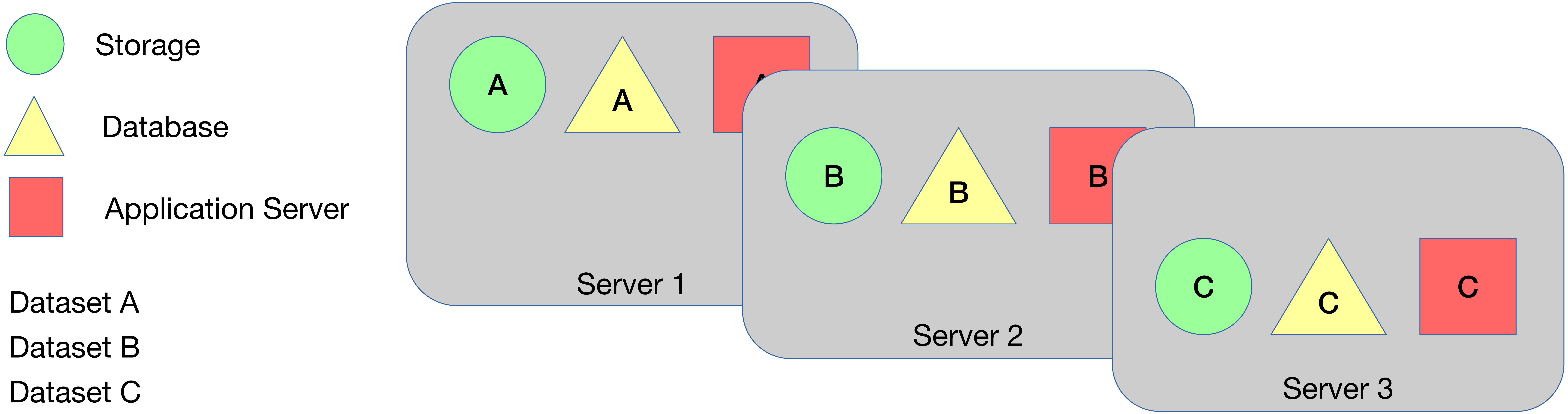
🟥 Application Server

Dataset A
Dataset B
Dataset C



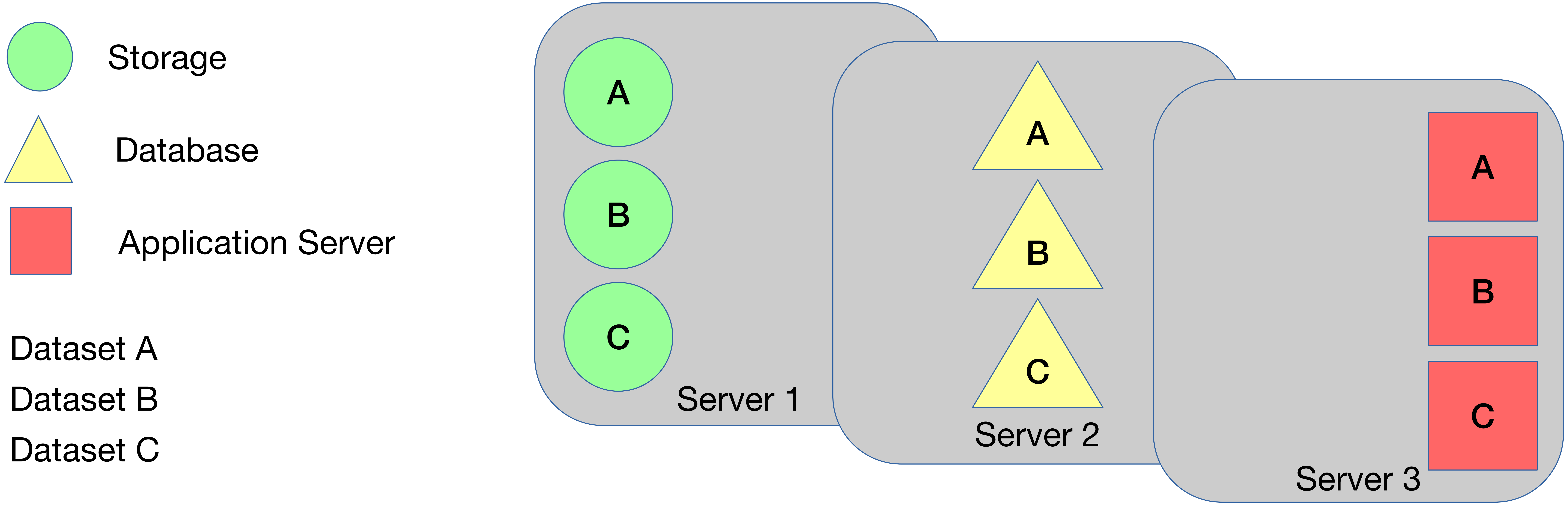
# Partitioning

- A server is split horizontally (***Sharding***)
  - Again, it can be applied to computation (functionality) and/or data
- Addresses **scale-out** (i.e., Server 1 handles data/computation for client A, Server 2 for client B, and so on…)

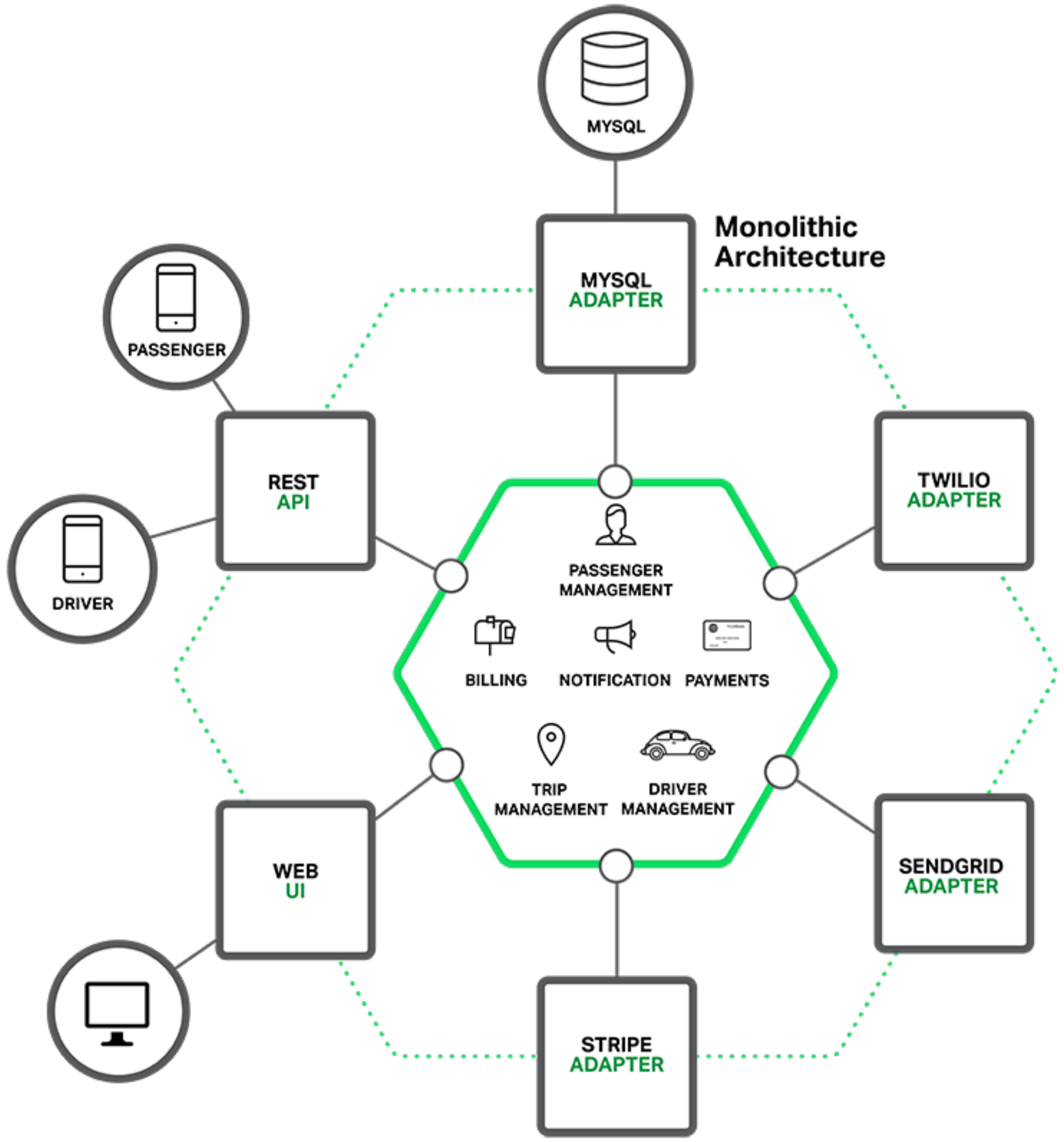


# Service-Oriented Architecture (SOA)

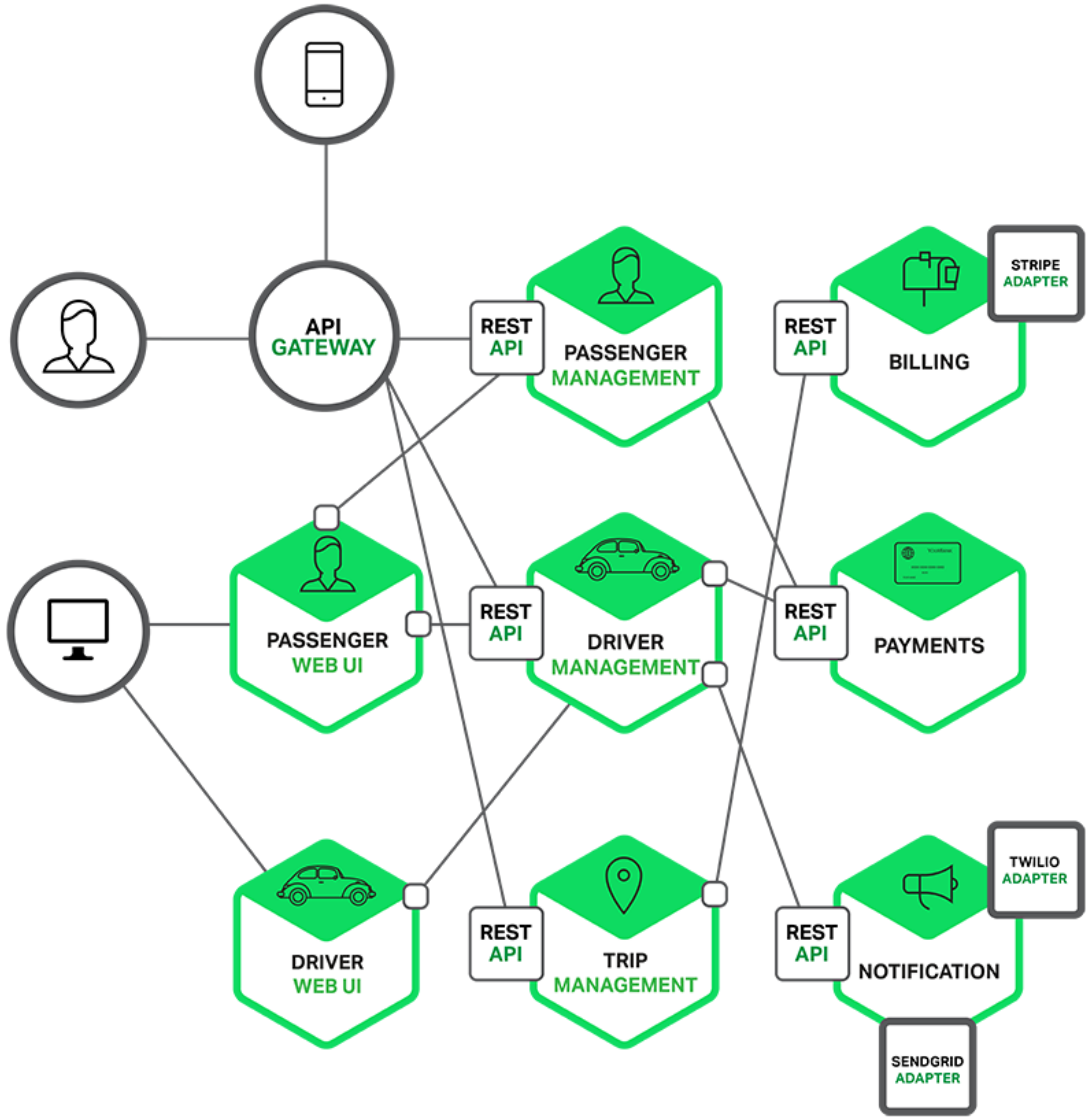
- A server is split vertically (e.g., Microservices)
- Addresses **scale-out** and **modularity** (i.e., Server 1 handles storage services, Server 2 handles database services and Server 3 handles application services)



# Monolithic to Microservices



*Monolithic*



*Microservices*

**Source:** *https://www.nginx.com/blog/introduction-to-microservices/*

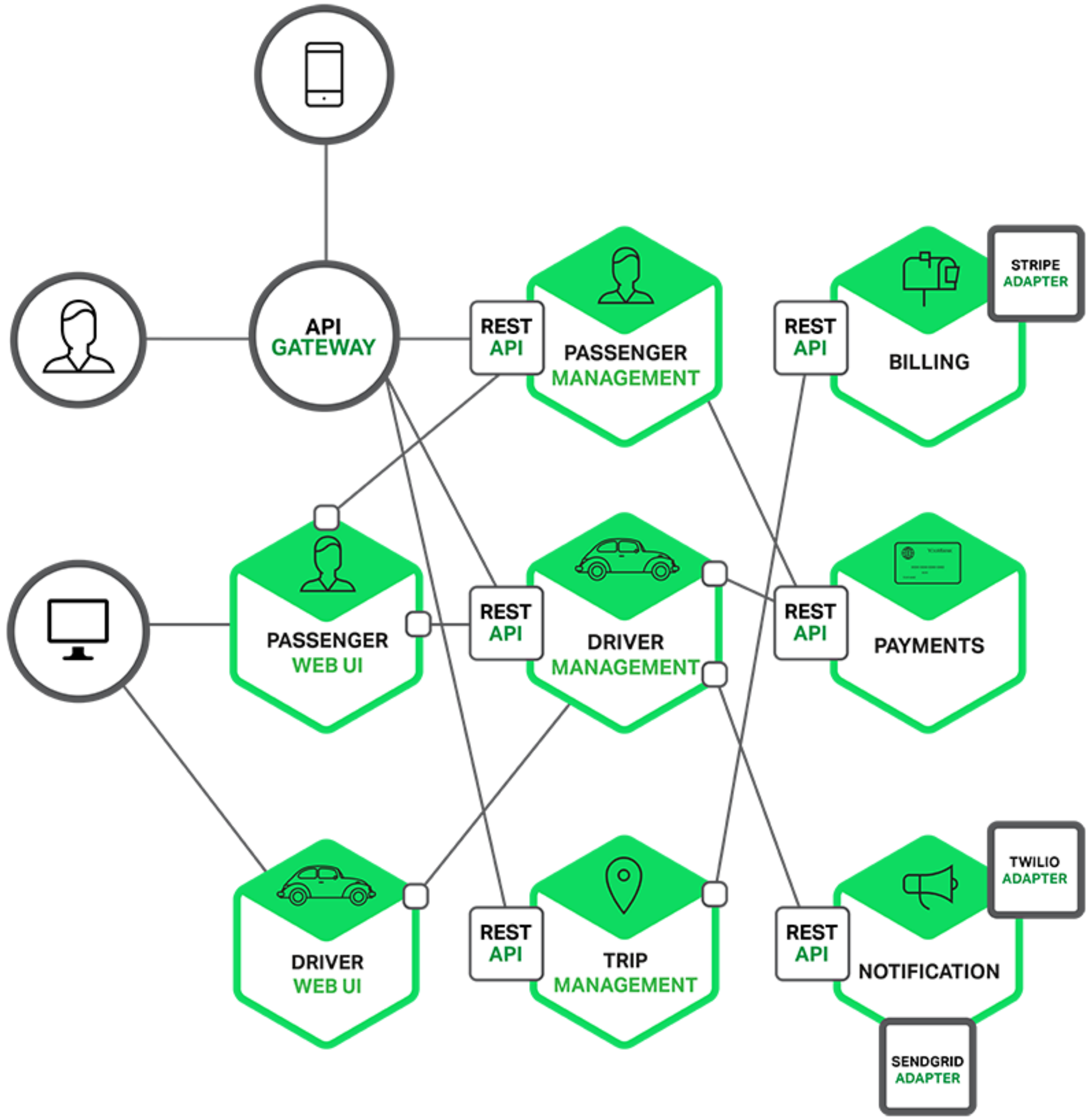
# Microservices

- ◉Each service implements specific **functionality** and can **scale independently**

- ◉Decomposition may be troublesome:
  - ‣ how micro is micro? Should I further decompose my functionalities into smaller services?

- ◉Consistency
  - ‣ How can one ensure data is consistent across all services?

- ◉Complex deployment and testing



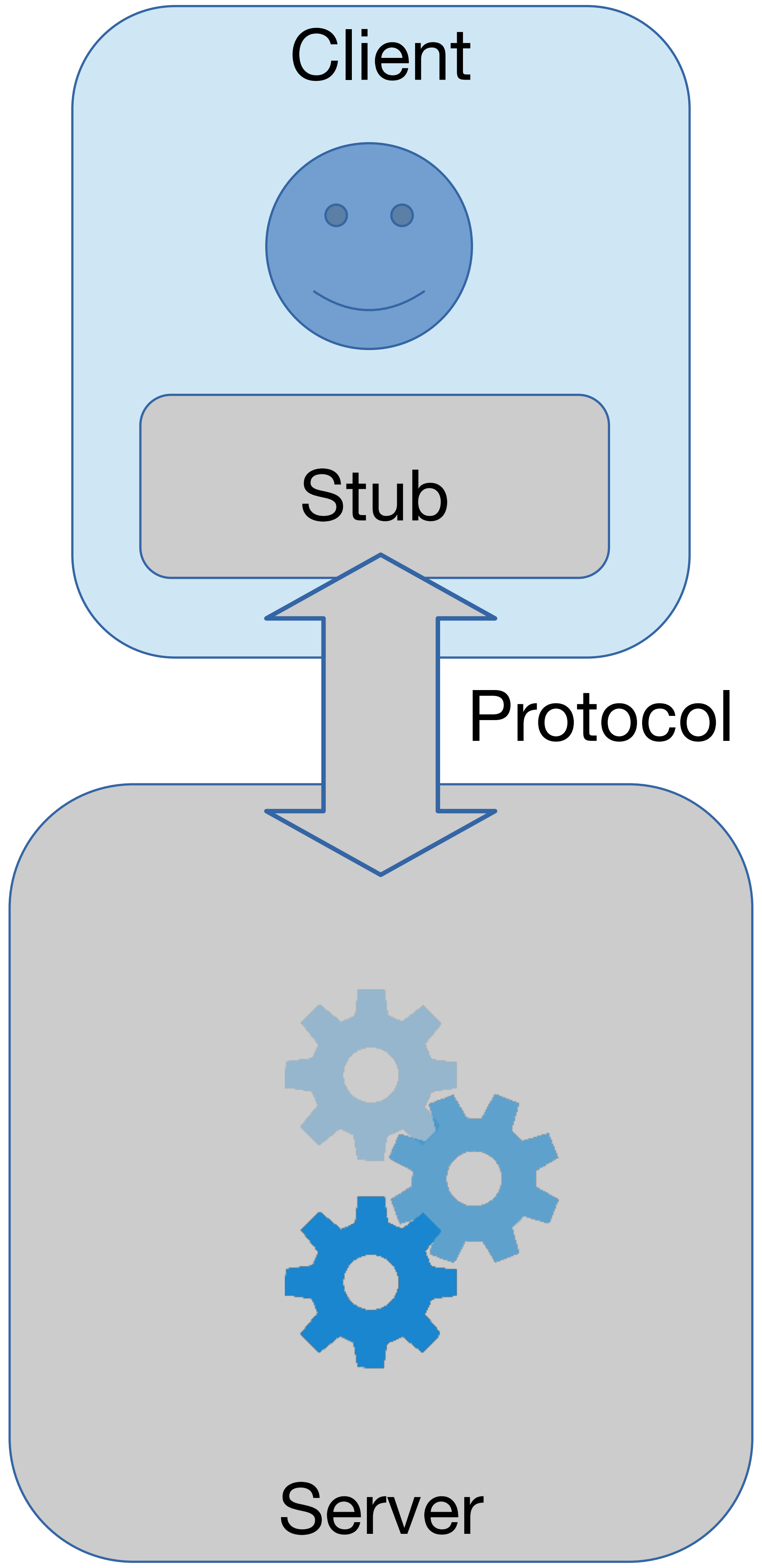
*Microservices*

# Distributed architectures

- We talked about distribution mechanisms and patterns, what about the **architectures** that enable these?

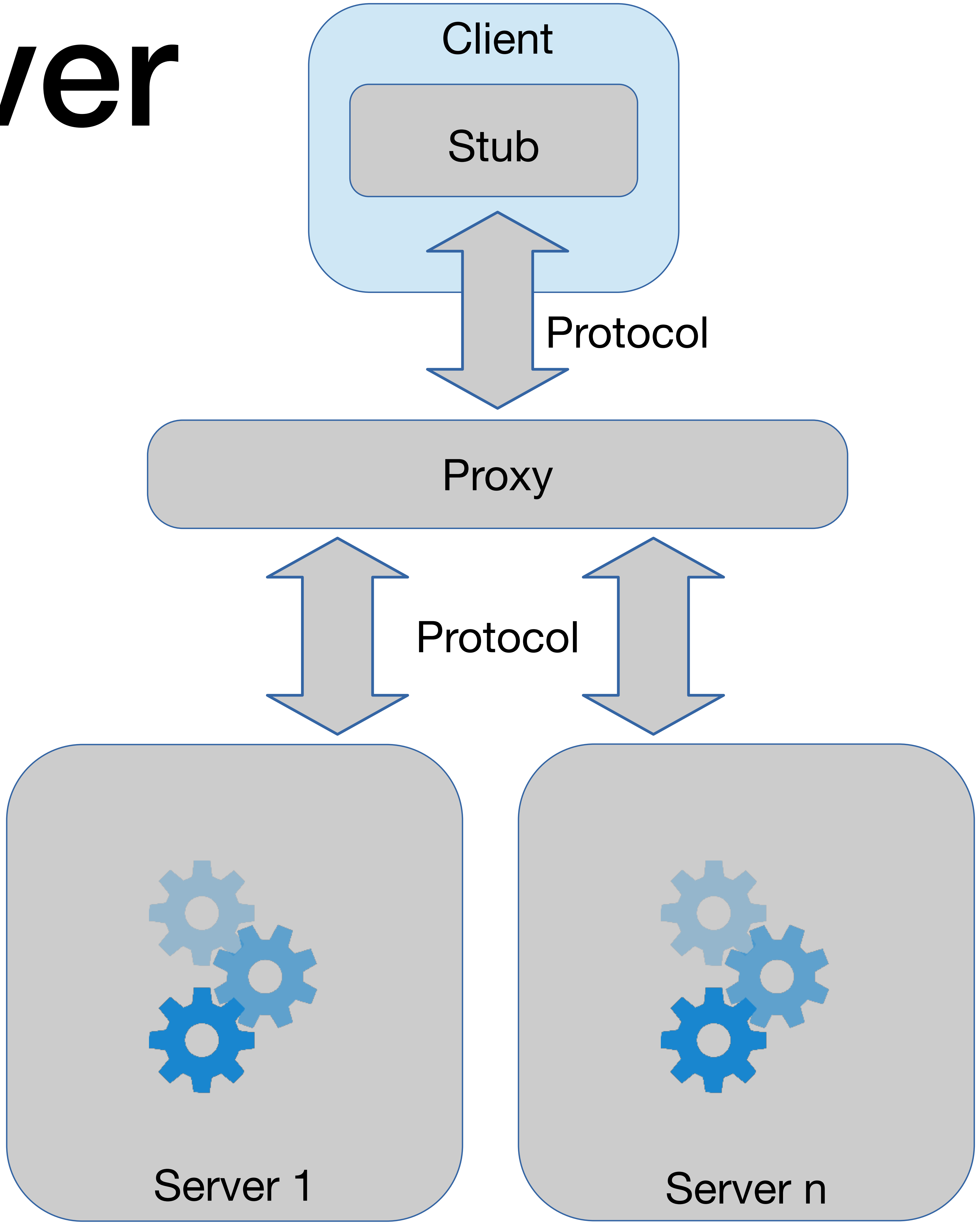
# Client-server

- **Functionality** and **data** are **in the server**
- A **stub** runs embedded in the client
  - provides an API to interact with the server
  - abstracts the details of the protocol
  - It is part of the server software package
- <u>*Example:*</u> the Web
  - “protocol” is HTTP
- If there is only one server…
  - One does not grant scalability or dependability!



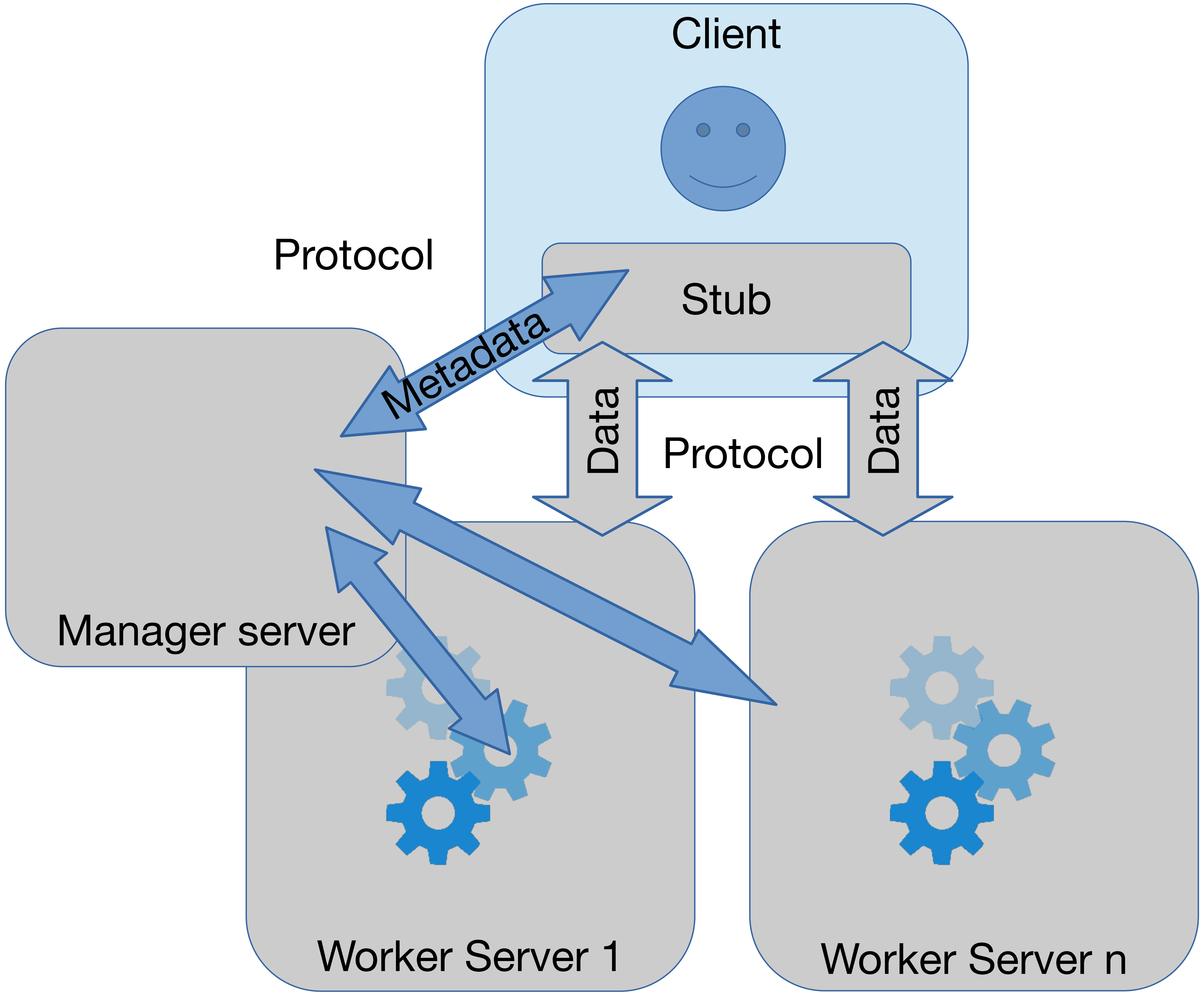
# Proxy-server

- The **proxy** abstracts the interaction with multiple servers
  - Good for implementing transparent **replication and/or sharding** of data and functionality!
- A **proxy** is different from a **stub**!
  - **Proxy:** makes the underlying servers transparent to clients, sitting below the protocol to communicate with clients
  - **Stub:** makes the protocol transparent to clients
  - They can be combined!
- The proxy may be a **scalability** and **availability bottleneck**!
  - Single point of contention and failure!
- *Example*: MongoDB



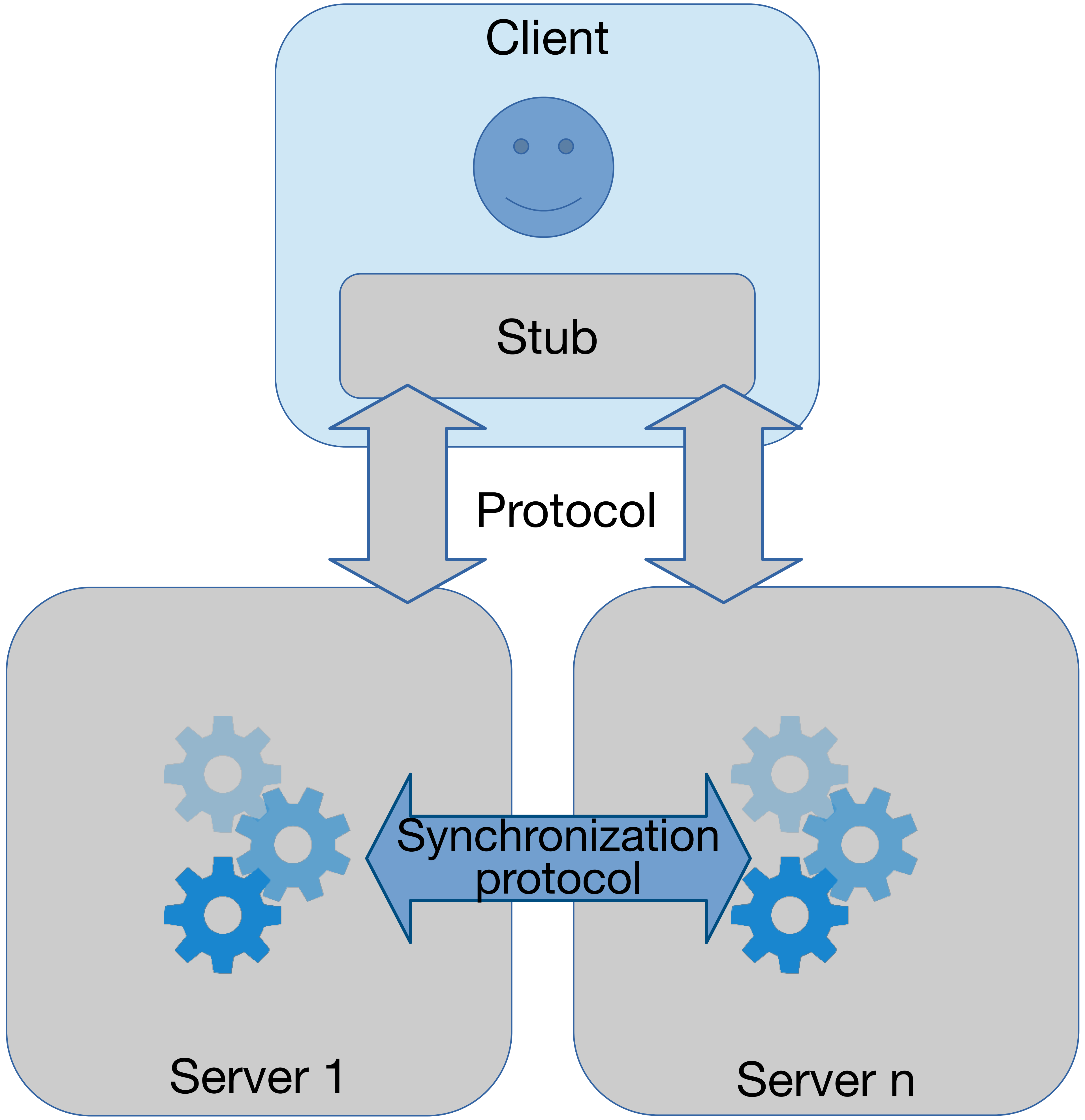
# Manager-worker

- Functionality is split between a **manager** and **worker** servers
- Common architecture, for example, for storage systems such as HDFS, Lustre, …
  - **Manager:** handles the location of files (metadata)
  - **Workers:** handle the content of files (data)
  - Data may be **replicated/sharded** across several workers
- Further addresses **scale-out** i.e., by splitting functionality!
- Manager may still be a single point of contention and failure!



# Server group

- ◉All servers can serve/process client requests
  - ‣ The failure of one server does does not compromise any **data/functionality** of the service!
- ◉A **synchronization protocol** is used to ensure the replication/consistency of servers' state
  - ‣ Requires **coordination** across servers
- ◉**No single point of failures**!
- ◉Coordination across server makes it **harder to scale-out**!
- ◉*Example*: Zookeeper

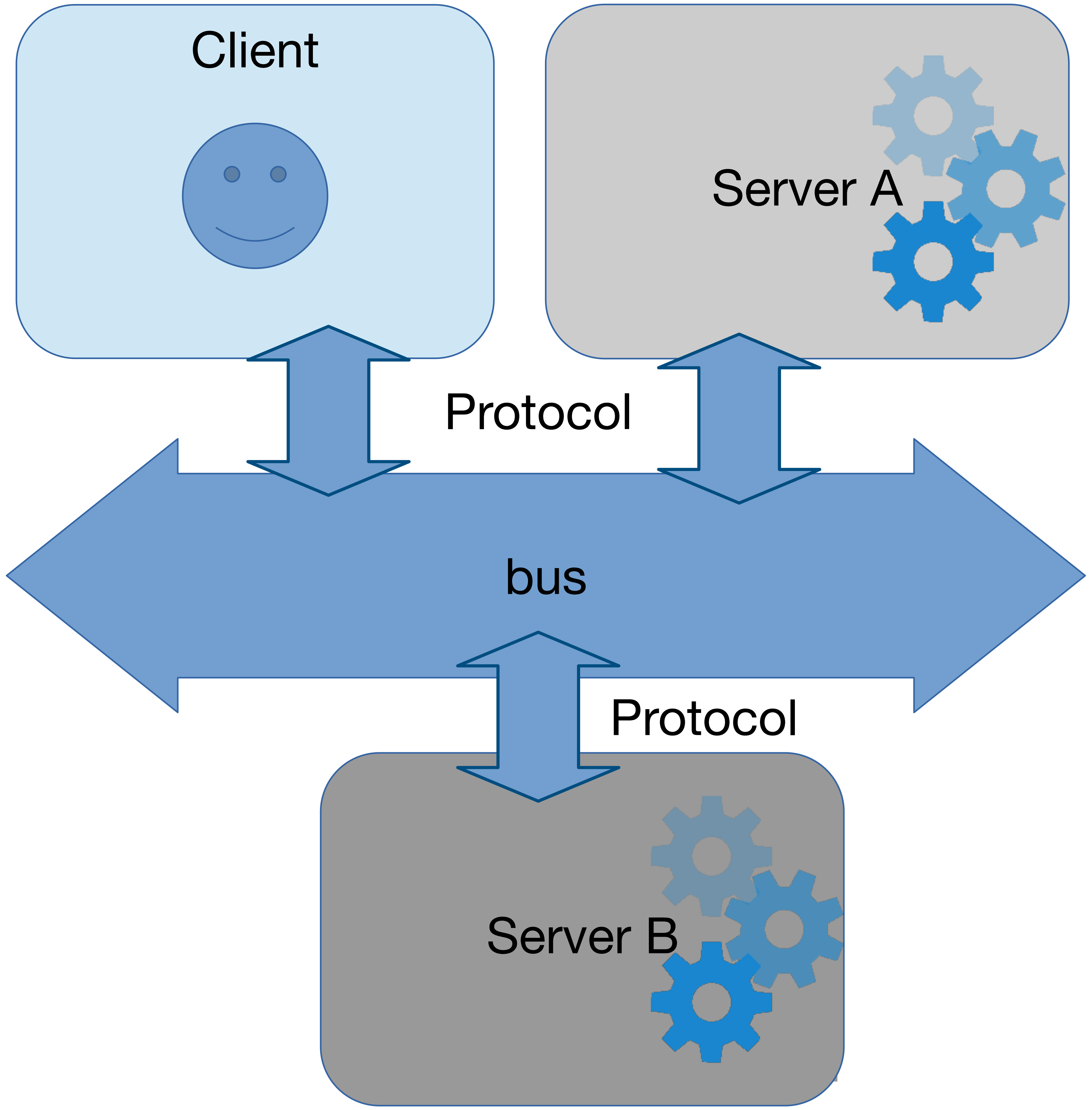


# Bus

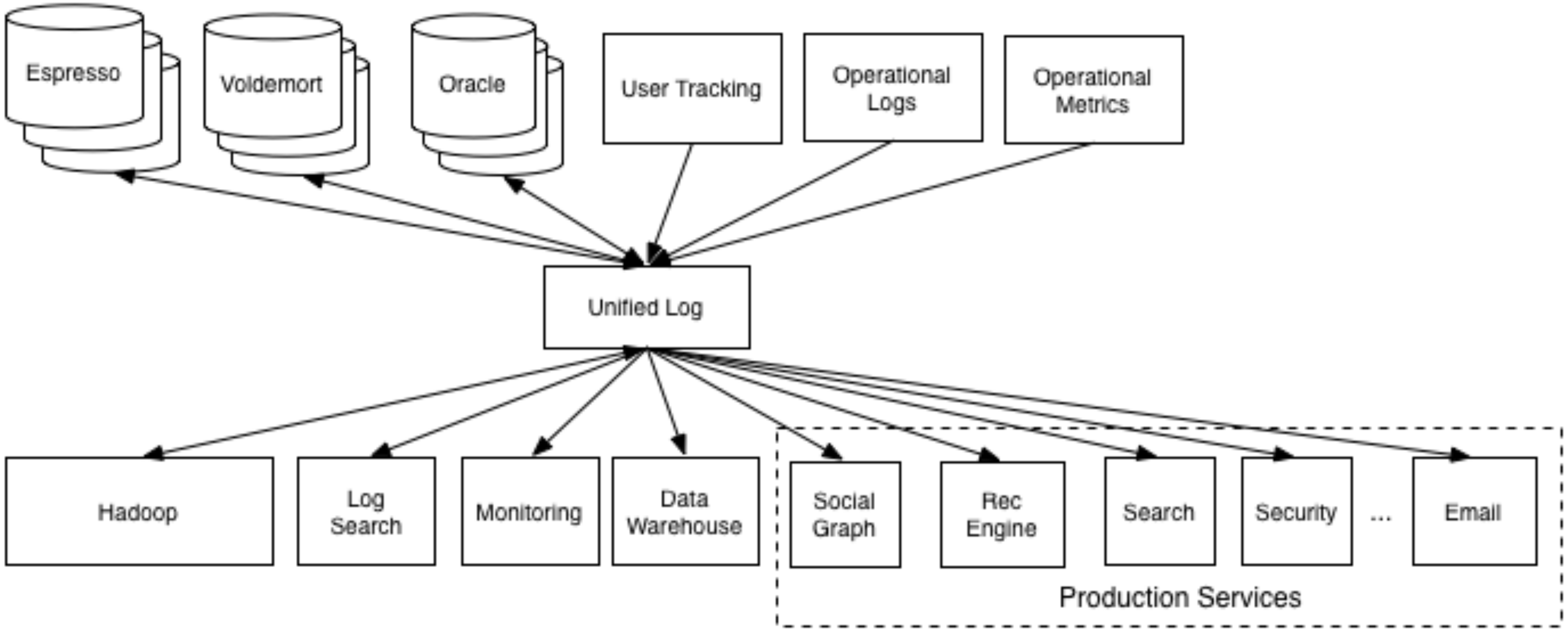
- ◉A **bus** routes messages
  - Participants **publish and consume** messages from the bus by using a given protocol
  - Decouples producers from consumers

- ◉Increased **flexibility**!
  - Clients (users/servers) can **both consume** a set of messages and **produce** another set
  - Easier to add new clients to the **bus**

- ◉*Example*: Kafka



# Apache Kafka



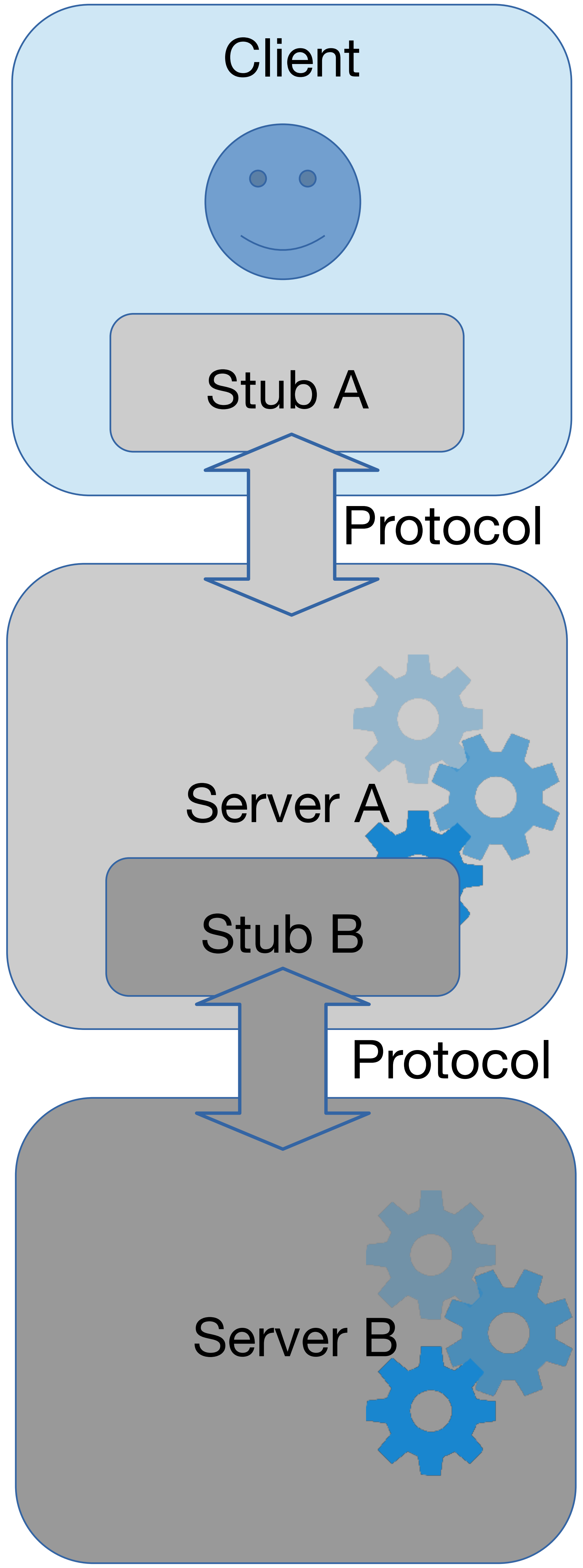
LinkedIn

# Multi-tier

- Each server acts as a client of the next tier
  - Nested client-server pattern

- Allows independent deployment and scaling of different functionality

- *Example*: Swap
  - “protocol A” == Web (e.g.)
  - “Stub B” == Database Driver!
  - “protocol B” uses SQL

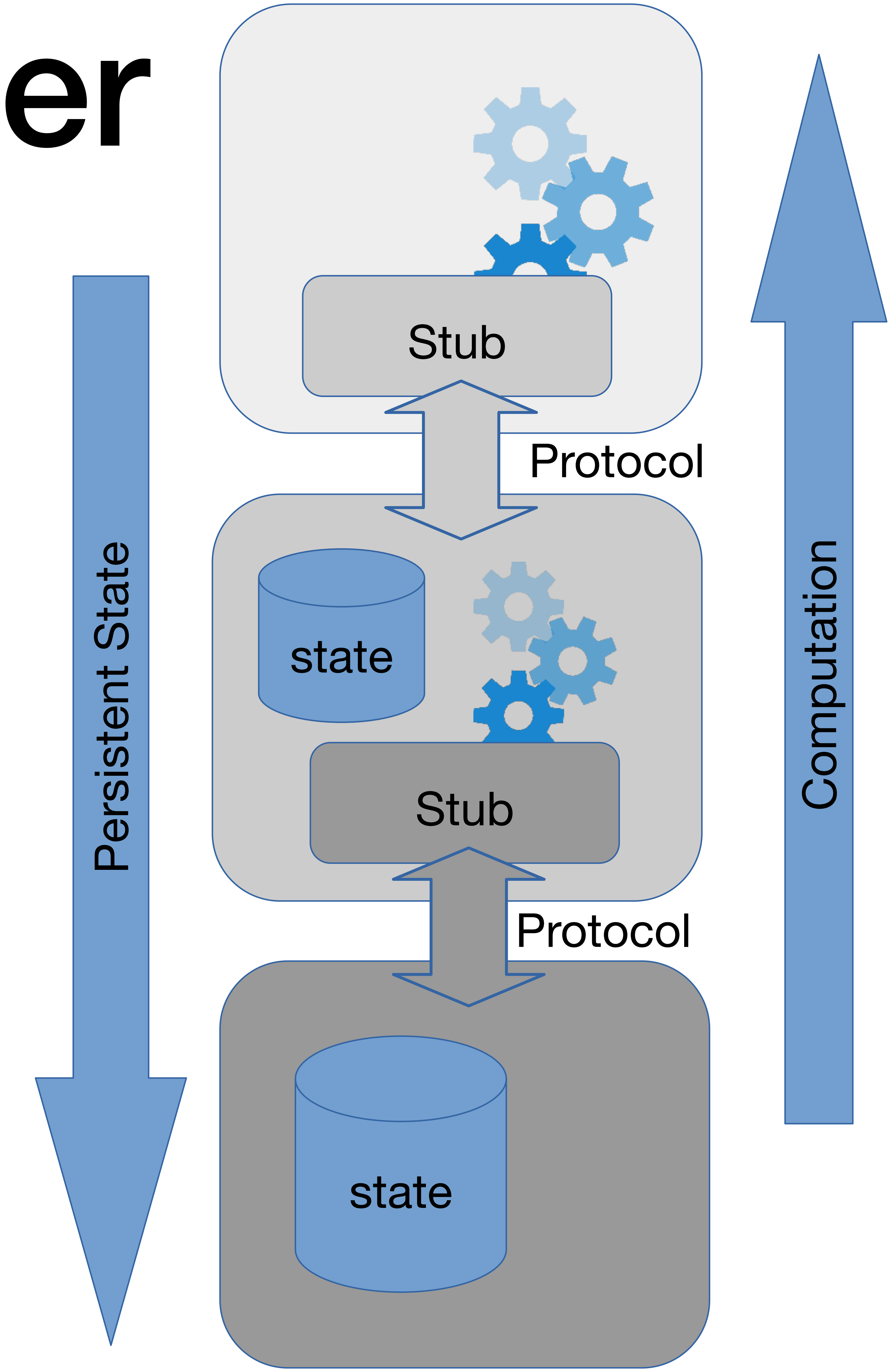


# State in multi-tier

- Typically (not always the case…)
  - No state in upper tiers: Web server
  - Transient / cached state in middle tiers: Application Server
  - Persistent state at lower tiers: Database

- **Question:** Which tiers does one need to consider to ensure availability?

- Computation is typically easier to replicate and shard than persistent state
  - **Question:** Based on what we discussed so far, any idea on why?

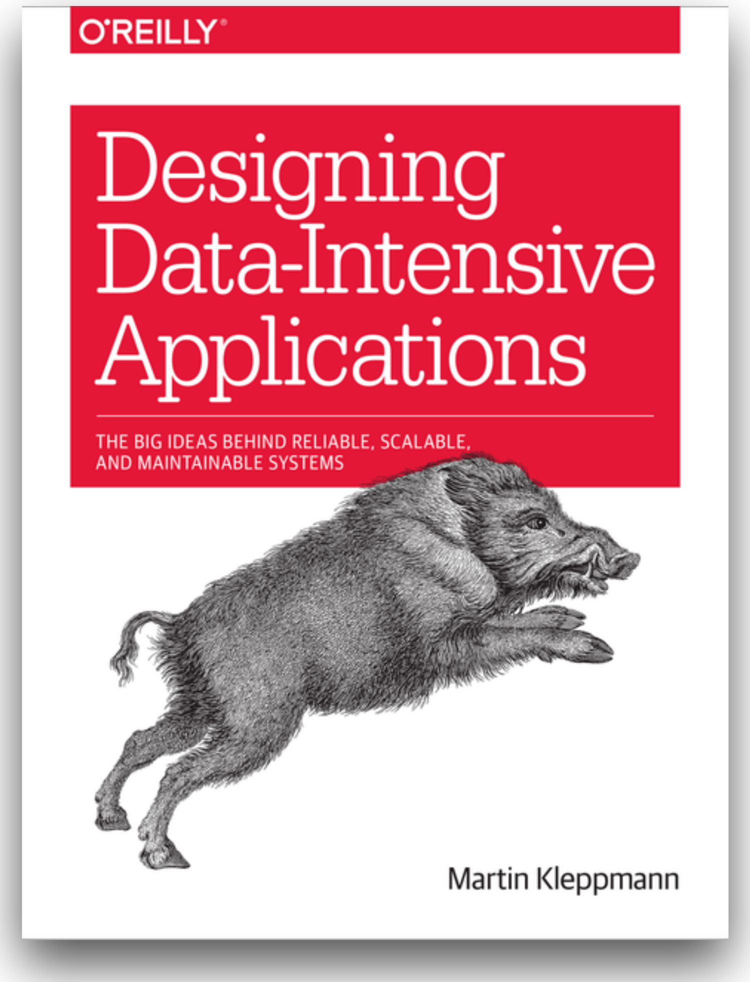


# Final Remarks

- Distribution **patterns**, **mechanisms** and **architectures** can be combined to meet different performance, scale, and dependability requirements
- Examples:
  - Data can be sharded across several servers for scalability, while data shards can be replicated for dependability purposes
  - Proxy and manager server components may follow a server-group architecture to avoid being single point of failures and/or contention
  - Each layer of a multi-tier application may follow a specific architecture and implement its own distribution mechanisms

# Further Reading

- M. Kleppmann. ***Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems***. O’Reilly, 2017
- C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. 2015. ***Holistic configuration management at Facebook***. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). ACM, New York, NY, USA, 328-343.


O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann

# Questions?