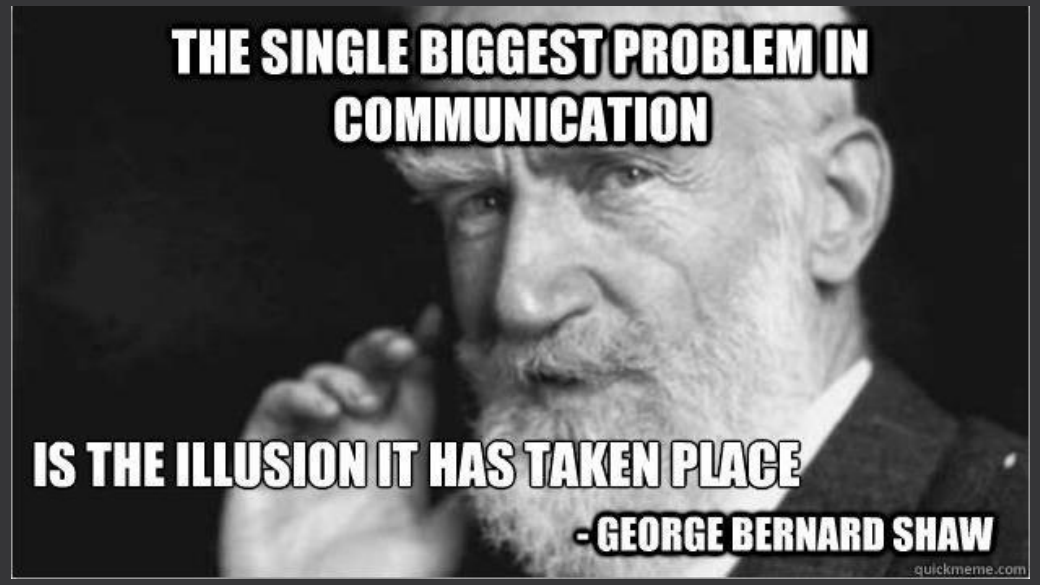
Communication Patterns in Microservices

Discussing the API architectures, technologies, and techniques.

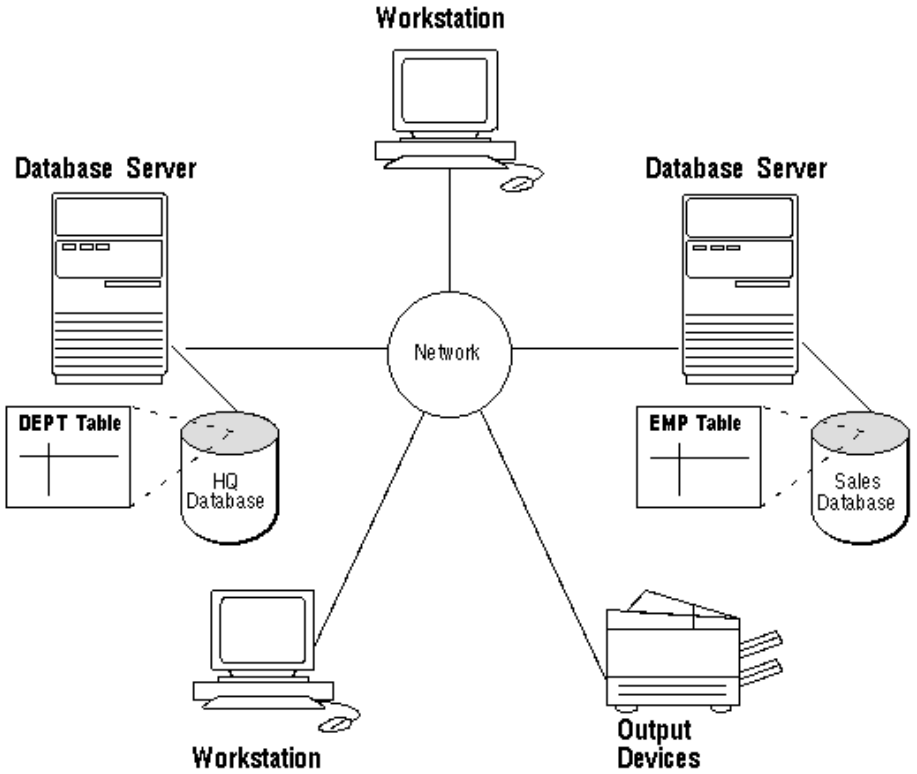
Authors:

- Mehdi Eidi
- Reyhaneh Mehdi Gholizadeh
- Mehrdad Heshmat



Introduction

- What is a distributed system?
- Why to care?

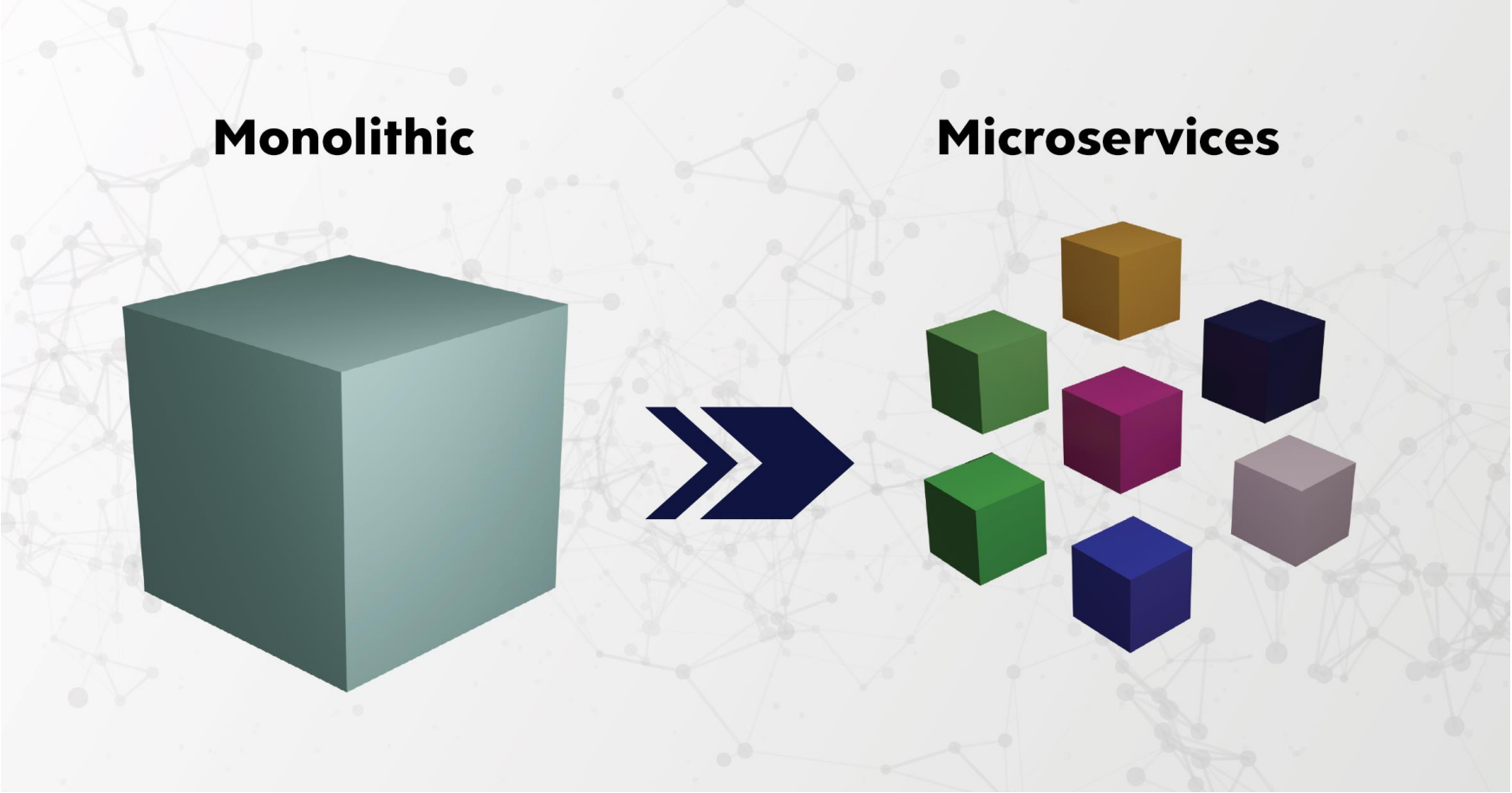


Why is distributed systems hard?

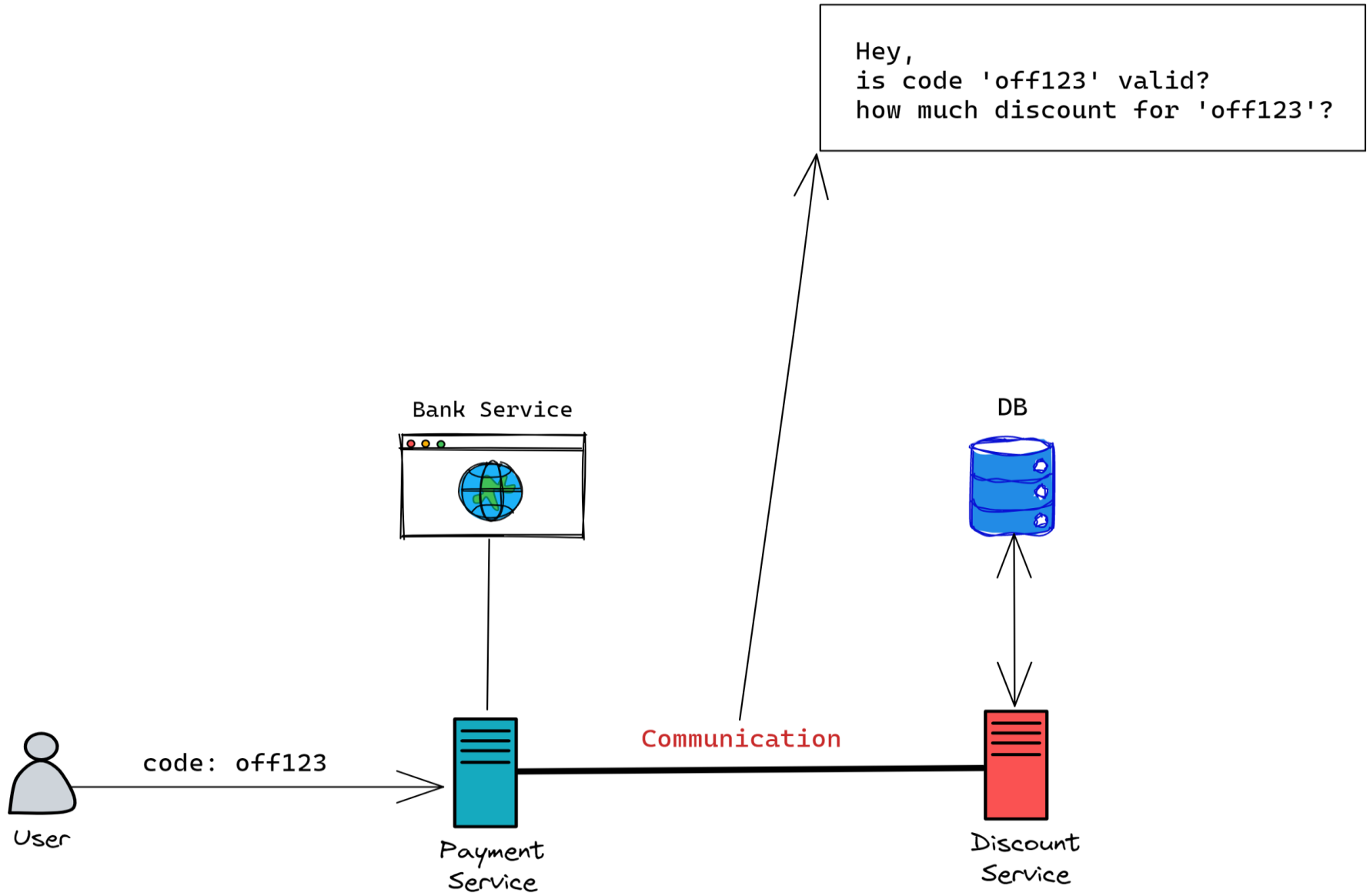
Some of distributed systems challenges:

- Communication
- Scalability
- Fault Tolerance
- …

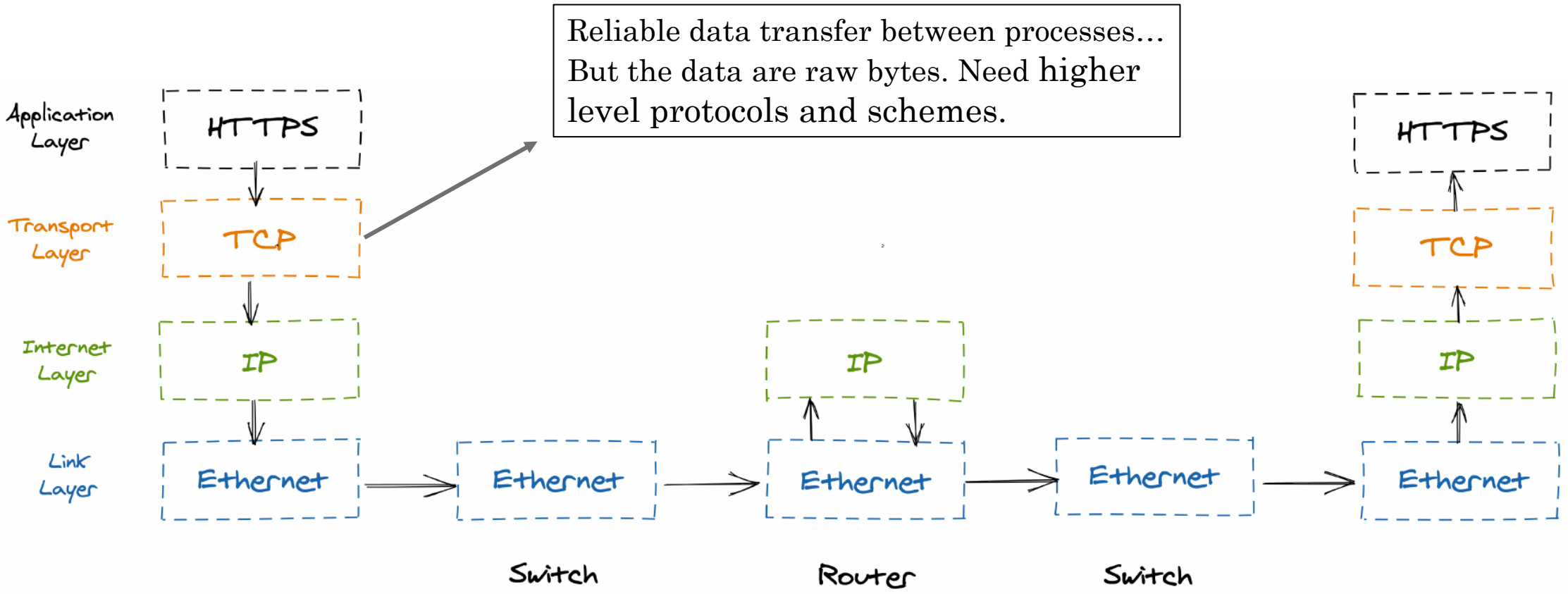
Microservices are connected with one another and external systems through inter-service communication techniques. This rises the importance of building robust communication among microservices.

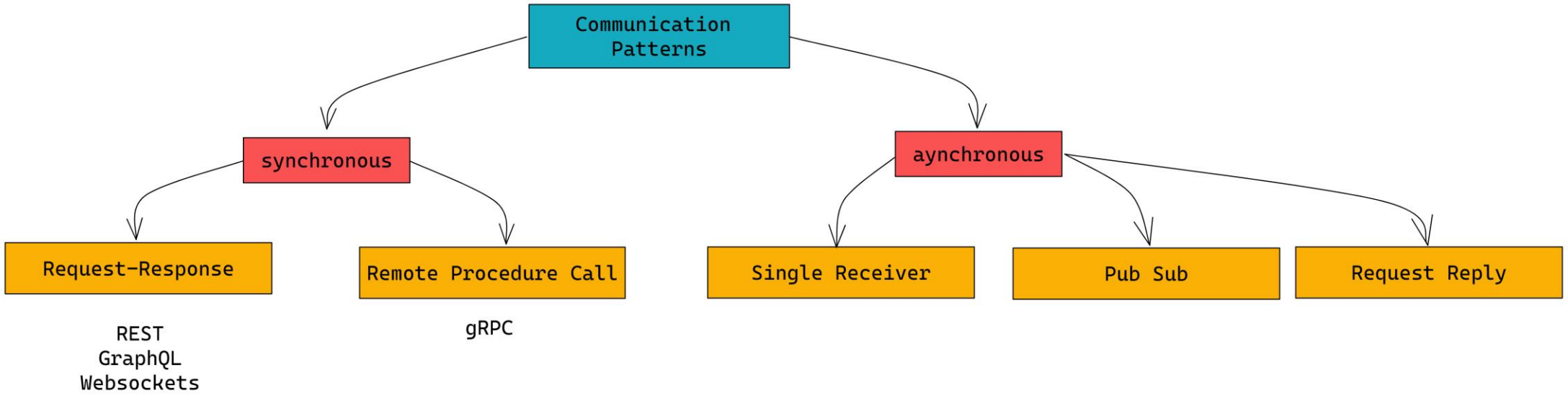


Communication Example

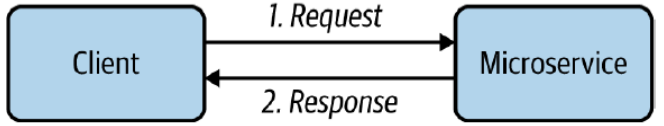


Network, the underlying infrastructure for communication

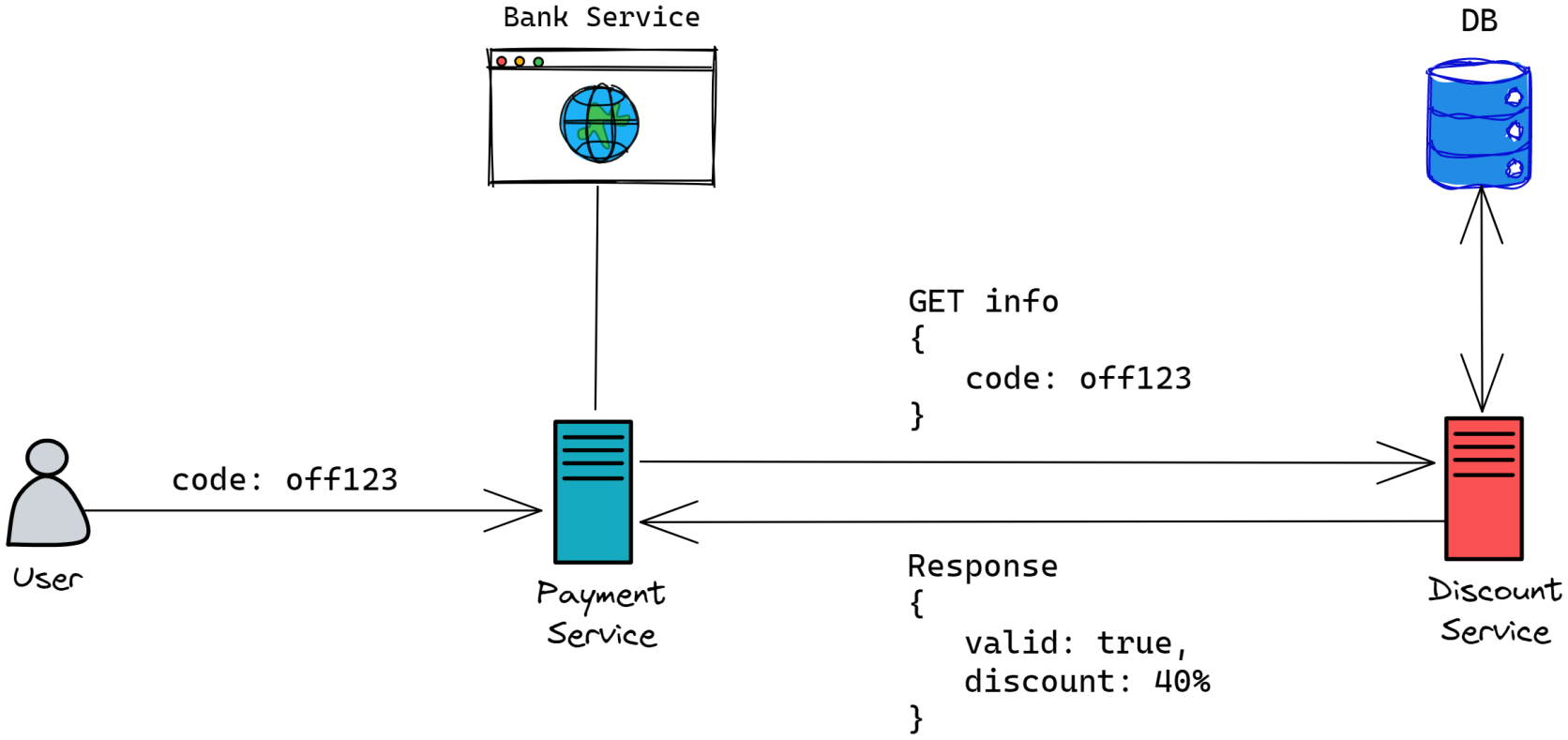



Communication Patterns
synchronous
aynchronous
Request-Response
Remote Procedure Call
Single Receiver
Pub Sub
Request Reply
REST
GraphQL
Websockets
gRPC

Sync Patterns, Request-Response

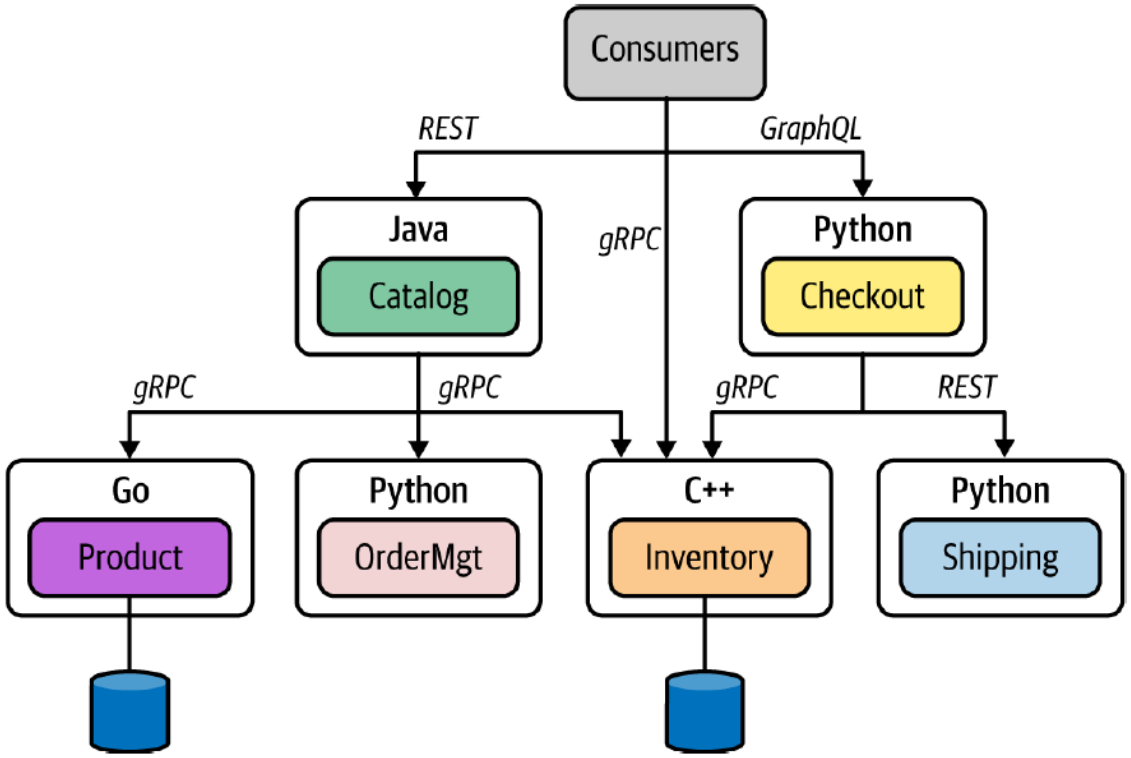


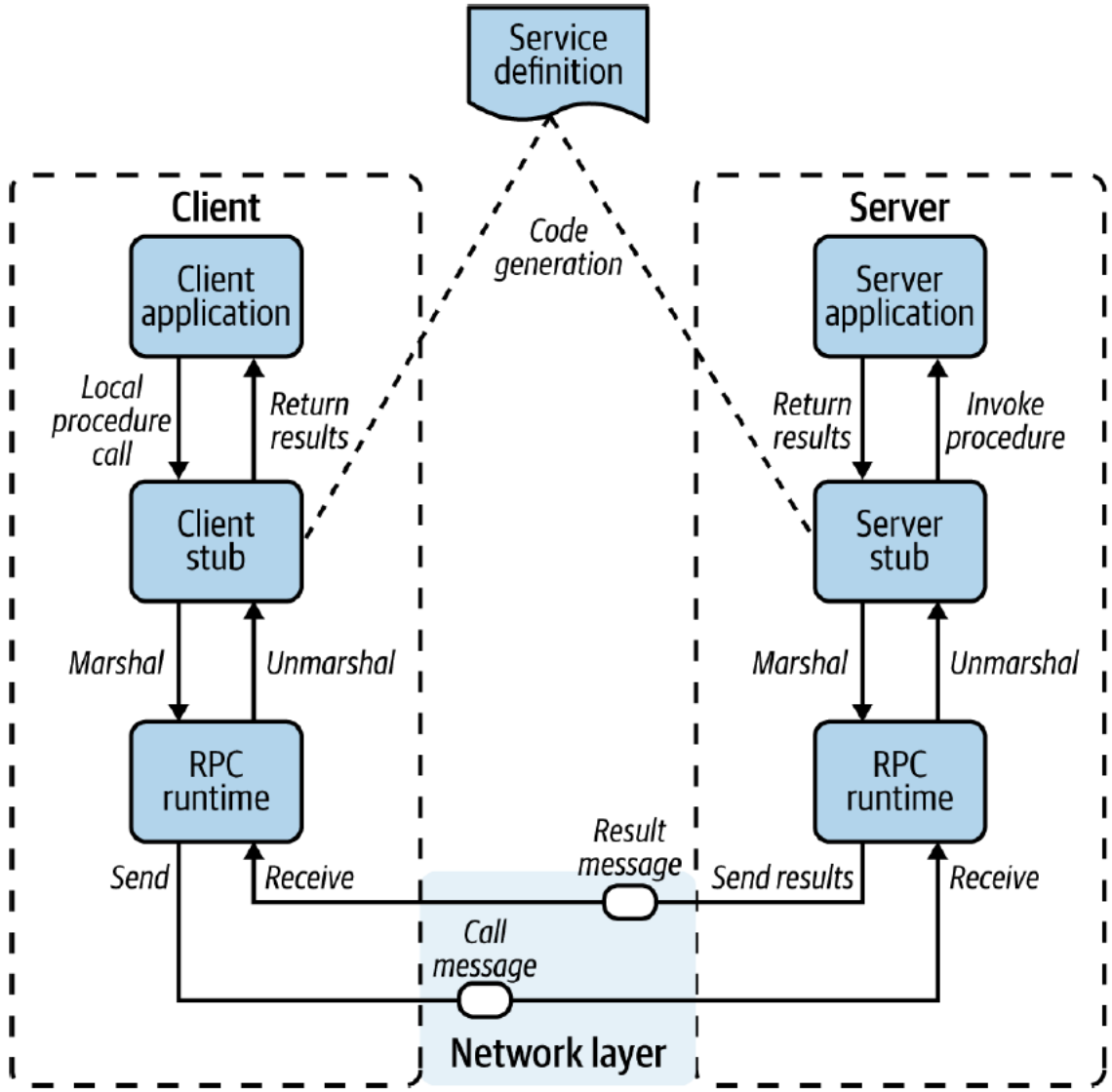
- Most common
- Communication channel kept open until time-out
- Also known as query-based interaction
- Suitable for external facing services
- Agnostic
- HTTP, RESTful
- Be careful of dependency chain and coupling



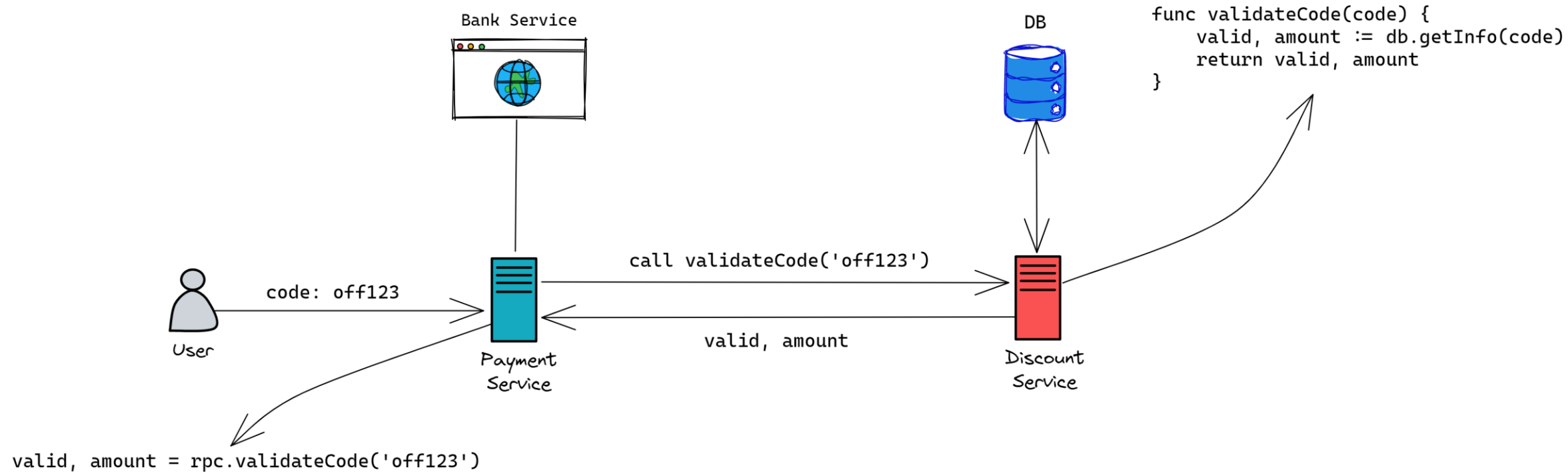
Sync Patterns, Remote Procedure Call

- Whats RPC?
- Suitable for inter-microservice communication
- IDL
- Old complicated CORBA, new gRPC
- gRPC on HTTP2
- Type Safe





Example

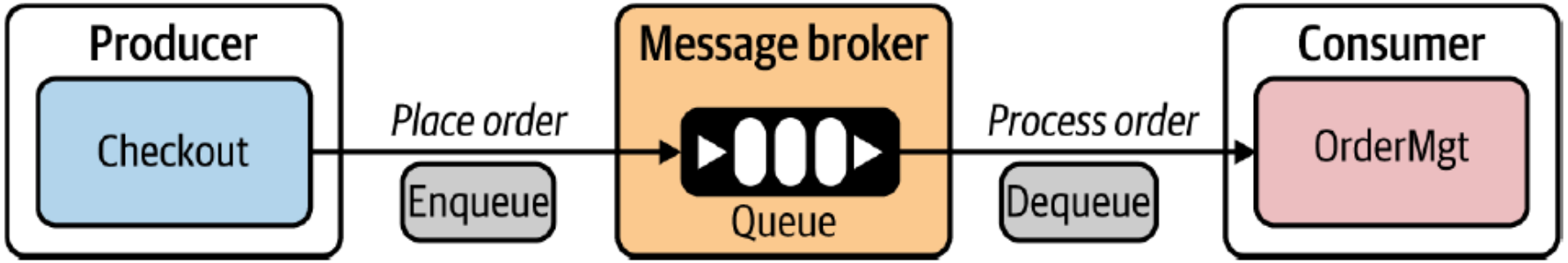

Bank Service
DB
func validateCode(code) {
valid, amount := db.getInfo(code)
return valid, amount
}
call validateCode('off123')
code: off123
valid, amount
User
Payment Service
Discount Service
valid, amount = rpc.validateCode('off123')

Sync Patterns, summary

Pattern	When to use	When not to use	Benefits
Request-Response	Services need real-time responses. Service contracts need to be flexible. To interoperate with many types of consumers. Services are exposed to external consumers.	Low-latency and high-throughput communication is required. Strict contract-first interactions are required.	The most interoperable and standard communication pattern for implementing services exposed to external as well as internal consumers.
Remote Procedure Calls	High-performance communication among services is critical. To enforce a strict contract-first approach for building services. Service business logic needs to be completely independent from the underlying wire protocol and its semantics.	Service interoperability with multiple application types such as web or mobile apps is required. You have to enable loose contracts and flexibility for consumers.	Suitable for efficient and type-safe service-to-service communication.

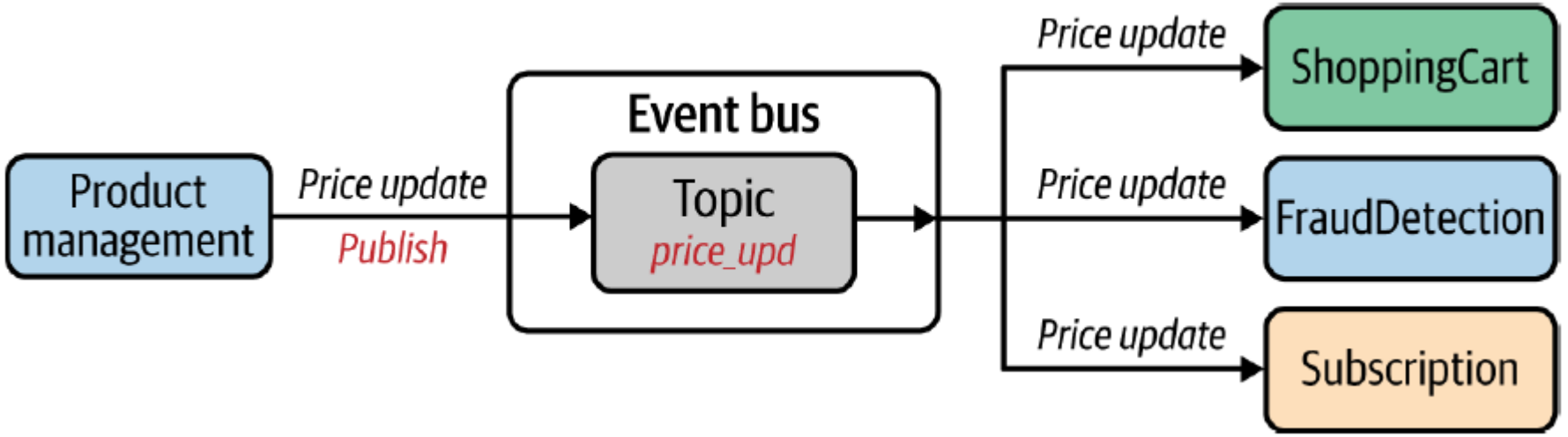
Async Patterns, Single Receiver

- One target service
- Message Broker & Queue
- Messages are commands
- Also known as point-to-point async messaging
- Reliable guaranteed, ordered delivery
- Can use ack
- Popular protocol: AMQP – Advanced message queuing protocol
- Language Agnostic
- RabbitMQ, Apache ActiveMQ, Azure Service Bus



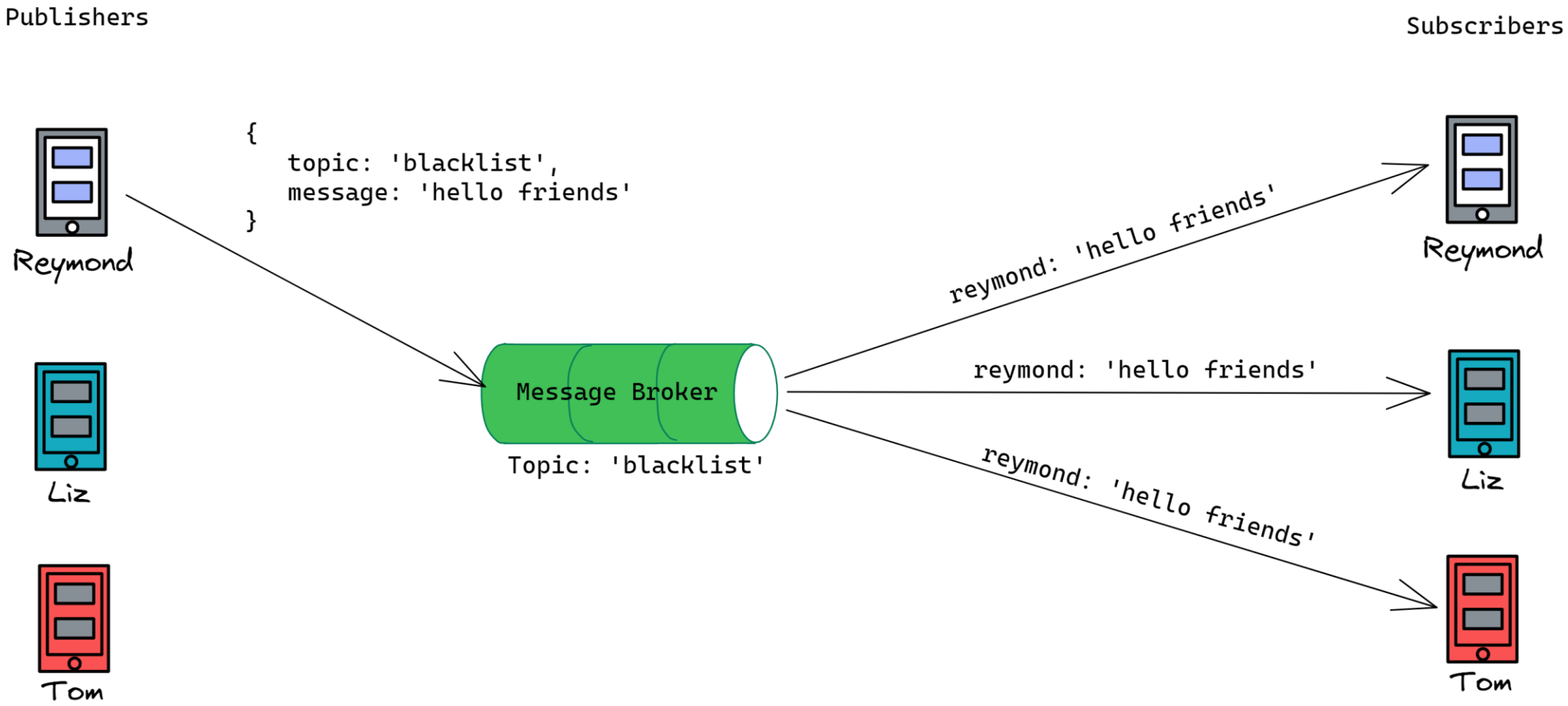
Async Patterns, Pub-Sub

- Multiple receivers
- e.g. notify multiple services of an event occurrence (event driven)
- Publishers & Subscribers
- Topic based
- Normally the broker sends messages to only the online subscribers
- Some brokers support durable subscription for guaranteed delivery
- Apache Kafka, NATS, Amazon SNS, Azure Event Grid, RabbitMQ, …



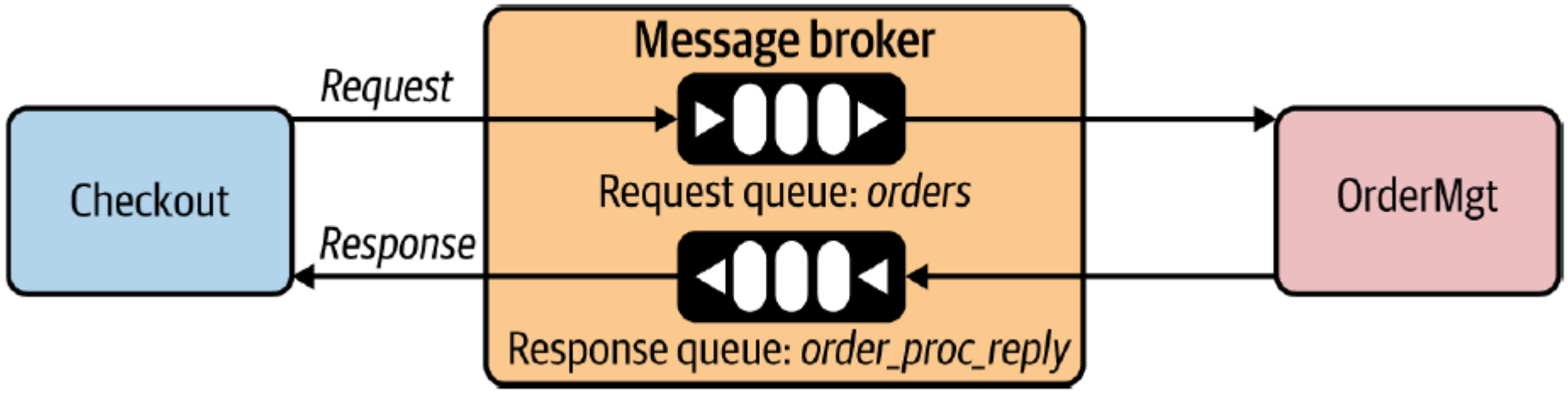
Pub-Sub Example

Suppose that Reymond, Liz, and Tom create a group in a chat app called ‘blacklist’



Async Patterns, Async Request Reply

- Producer expects reply
- Separate channel for reply
- Response queue (callback queue)
- Messages have meta-data about reply location, id, …
- RabbitMQ, ActiveMQ, Azure Service Bus

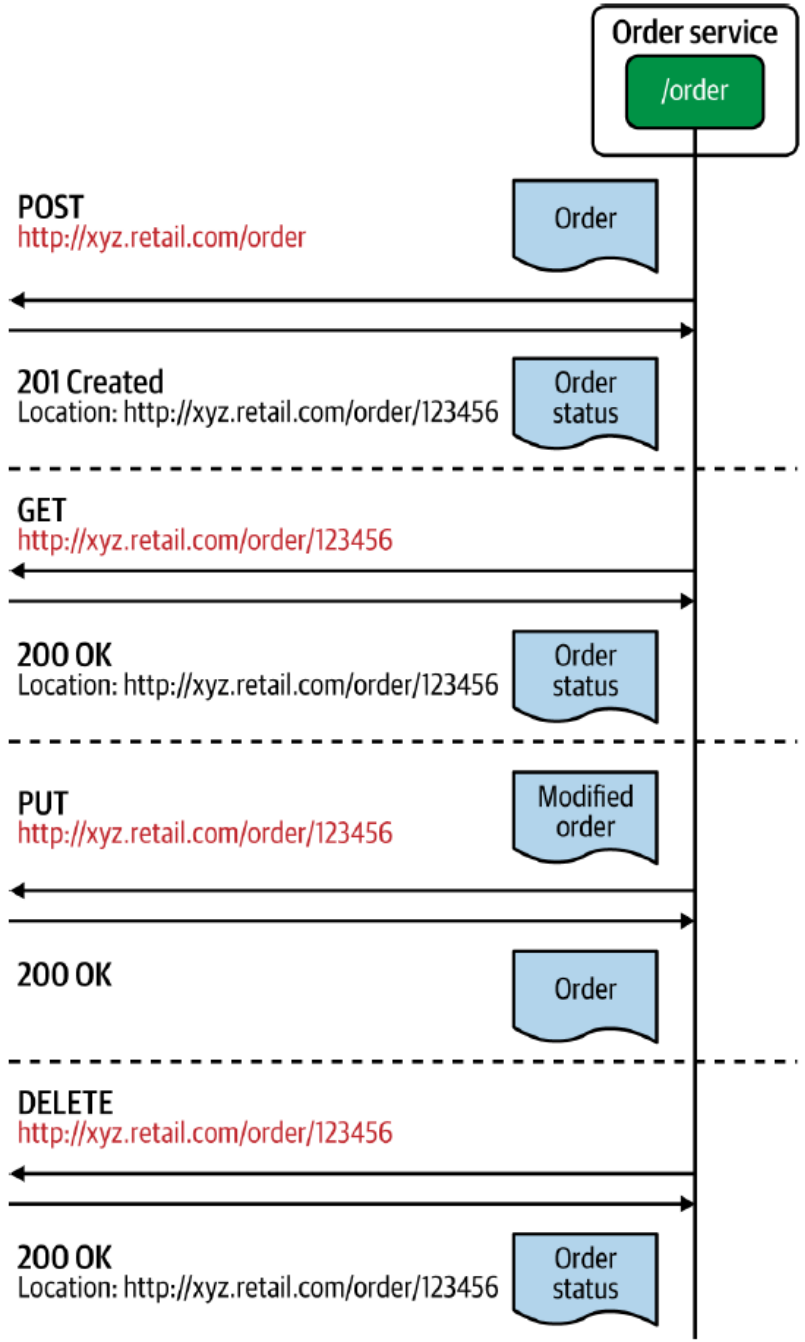


Async Patterns, Summary

Pattern	When to use	When not to use
Single-Receiver	One microservice sends an asynchronous command to another microservice. For ordered message delivery. For guaranteed message delivery.	Efficient data transfer is required without delivery semantics such as at-least-once.
Multiple-Receiver	More than one consumer is interested in the same message/event.	Usually not suitable when you need guaranteed message delivery.
Asynchronous Request-Reply	For asynchronous messaging scenarios in which correlation is required between a request and a reply.	Shouldn't be used as a reliable messaging alternative to synchronous request-response patterns.

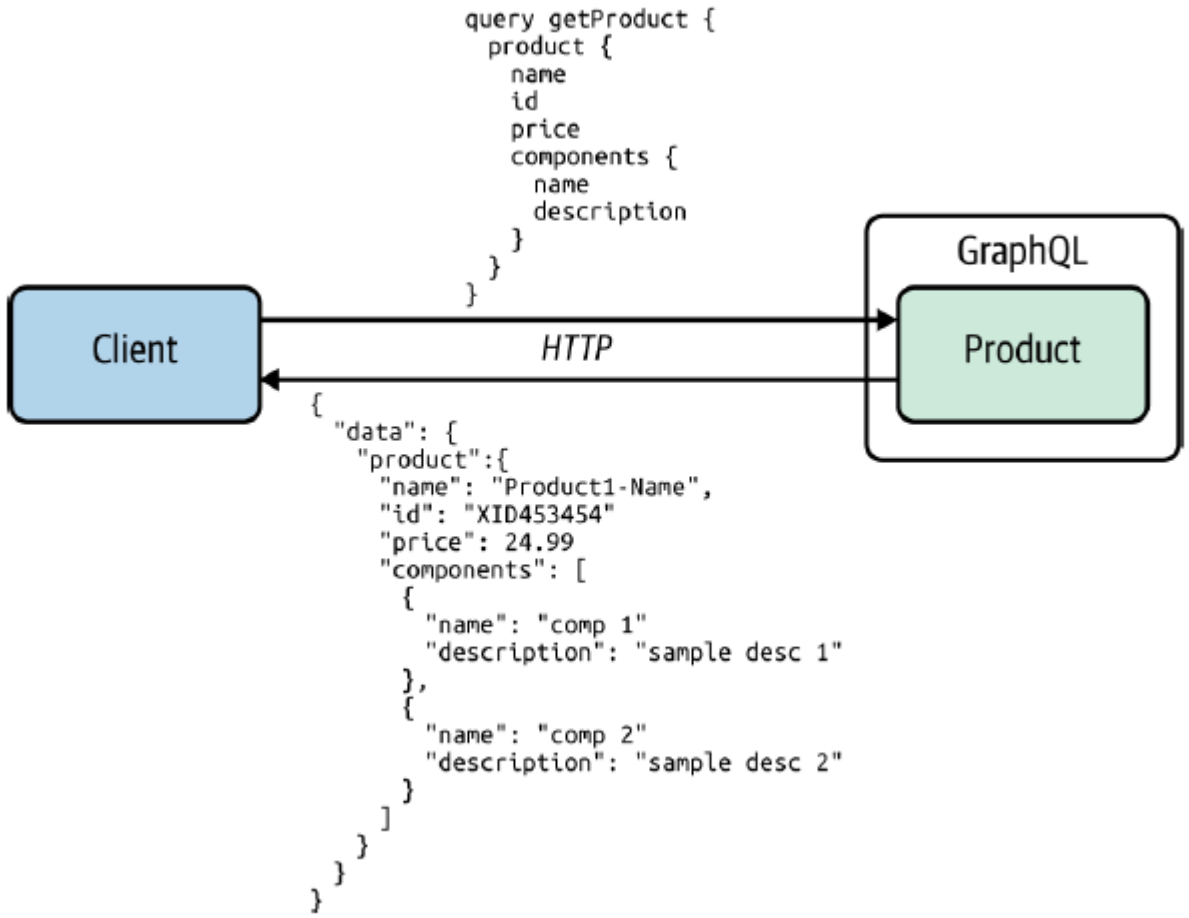
RESTful Services

- Request-Response Pattern
- REST architectural style
- URI for resource identifier
- Representation of the resources
- Protocol agnostic (most popular: HTTP)
- HTTP operations GET, POST, PUT, …
- Text-based data formats: JSON, XML, …
- Language agnostic



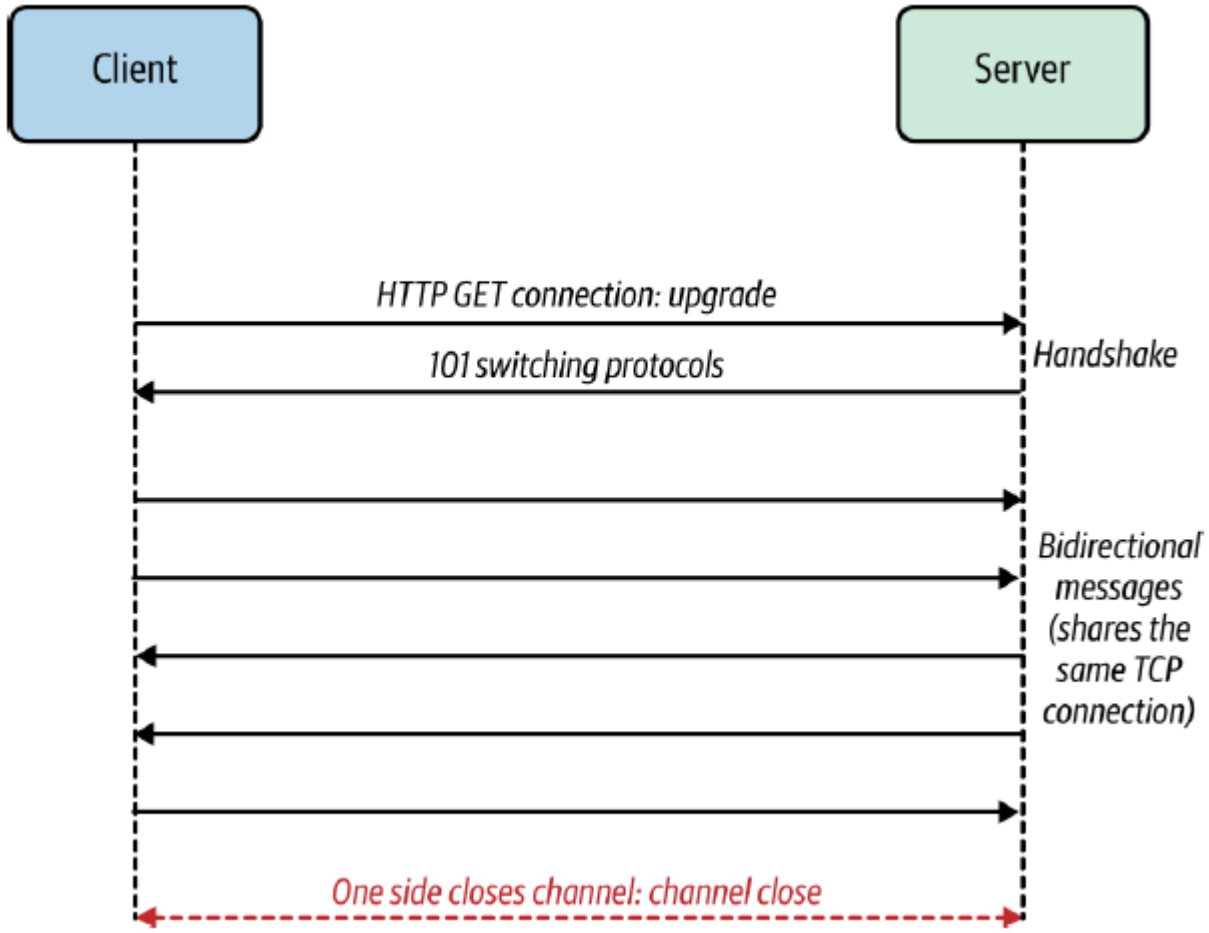
GraphQL

- Request-Response Pattern
- Based on querying
- Client has control over the response (unlike REST)
- Over HTTP under the hood
- Suitable for external-facing services
- Clients don't have to get redundant data



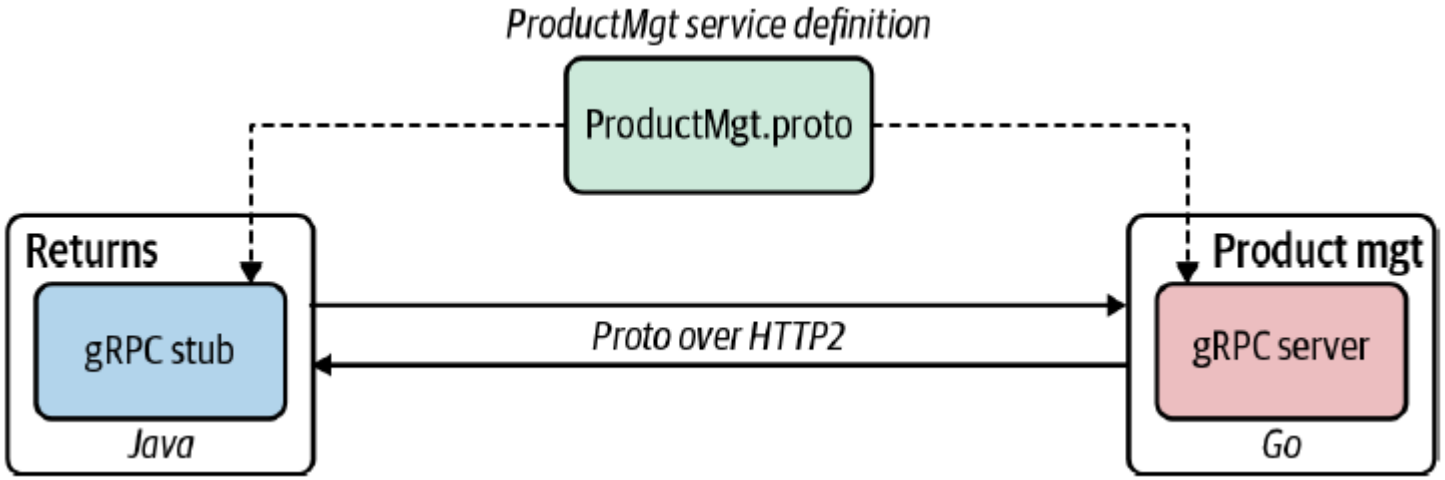
WebSocket

- Request-Response Pattern
- TCP over the Web
- Full duplex communication
- Async messaging
- Single TCP Connection
- Uses HTTP for initial handshaking
- Real-time messaging



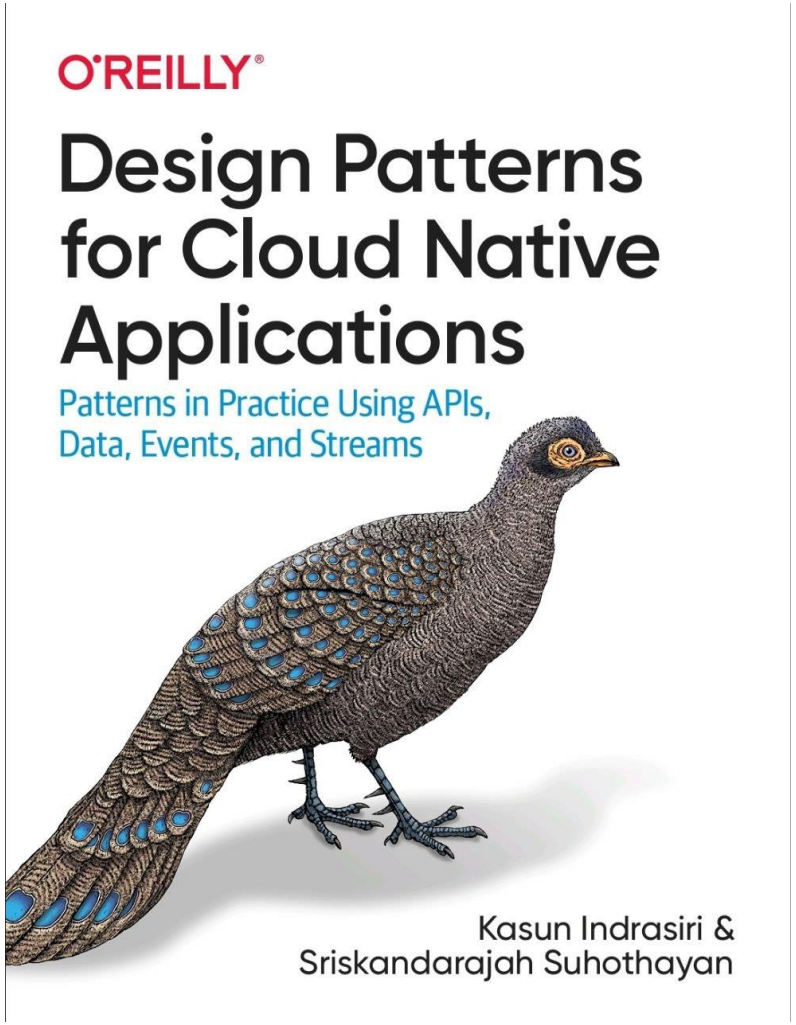
gRPC

- RPC Framework
- Binary data serialization: protobuf
- On top of HTTP2
- Can build polyglot apps
- Suitable for service-to-service communication



Main Resource

- Design Patterns for Cloud Native Applications by Indrasiri



Thanks for listening