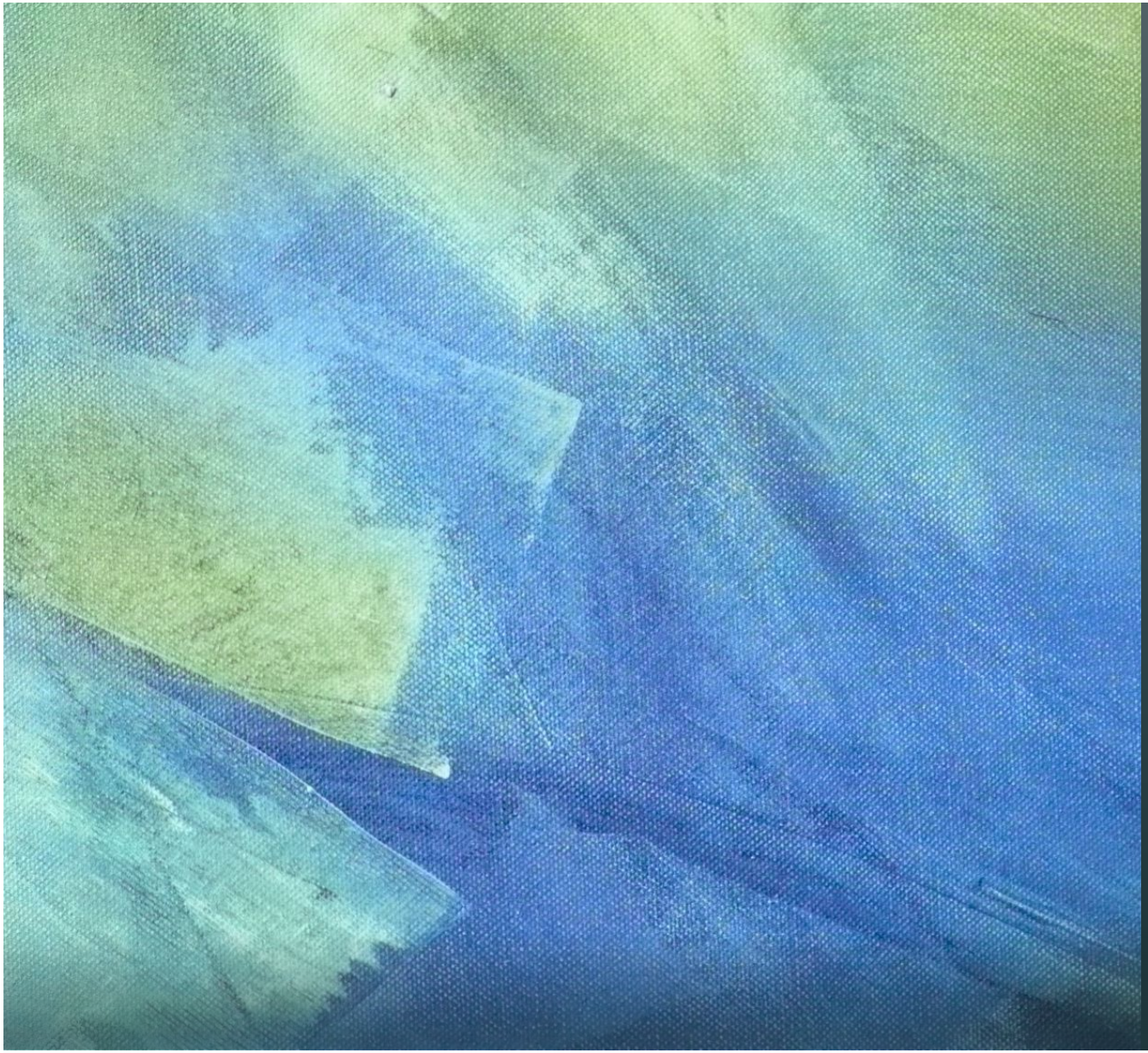


Interprocess communication in a microservice architecture

Wiwat Vatanawood

Duangdao Wichadakul

Nuengwong Tuaycharoen

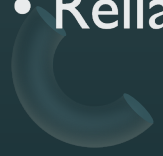
Pittipol Kantavat

This chapter covers

- Applying the communication patterns:
  - Remote procedure invocation
  - Circuit breaker
  - Client-side discovery
  - Self registration
  - Server-side discovery
  - Third party registration
  - Asynchronous messaging
  - Message broker

This chapter covers (cont.)

- The importance of interprocess communication in a microservice architecture
- Defining and evolving APIs
- The various interprocess communication options and their trade-offs
- The benefits of services that communicate using asynchronous messaging
- Reliably sending messages as part of a database transaction



Agenda

- Overview of interprocess communication in a microservice architecture
- Communicating using the Synchronous Remote procedure invocation pattern
- Communicating using the Asynchronous messaging pattern

Overview of interprocess communication

- There are lots of different IPC technologies
  - Synchronous request/response-based communication mechanisms, such as HTTPbased REST or gRPC
  - Asynchronous, message-based communication mechanisms such as Advanced Message Queuing Protocol (AMQP) or STOMP
- Also, a variety of different messages formats
  - Human-readable, text-based formats such as JSON or XML
  - Binary format such as Avro or Protocol Buffers

Interaction styles

- The first dimension :-
  - One-to-one—Each client request is processed by exactly one service.
  - One-to-many—Each request is processed by multiple services.
- The second dimension :-
  - Synchronous—The client expects a timely response from the service and might even block while it waits.
  - Asynchronous—The client doesn't block, and the response, if any, isn't necessarily sent immediately.

Interaction styles in two dimension

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

One-to-one interactions

- Request/response
  - A service client makes a request to a service and waits for a response.
  - The client expects the response and might event block while waiting
- Asynchronous request/response
  - A service client sends a request to a service (replies asynchronously).
  - The client doesn't block while waiting
    (because the service might not send the response for a long time).
- One-way notifications
  - A service client sends a request to a service, but no reply is expected or sent.

One-to-many interactions

- Publish/subscribe
  - A client publishes a notification message
  - Consumed by zero or more interested services.
- Publish/async responses
  - A client publishes a request message
  - Waits for a certain amount of time for responses from interested services.

Defining APIs in a microservice architecture

- A well-designed interface exposes useful functionality while hiding the implementation
- Enables the implementation to change without impacting clients.
- It's important to precisely define a service's API using *Interface Definition Language* (IDL)

Message formats

- *Messages* usually contain data.
- Two main categories of message formats: text and binary
  - Text-based formats such as JSON and XML
  - Binary formats such as Protocol Buffers and Avro

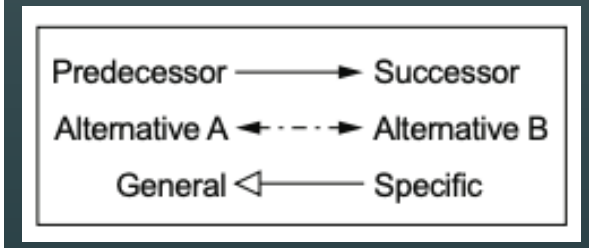
Agenda

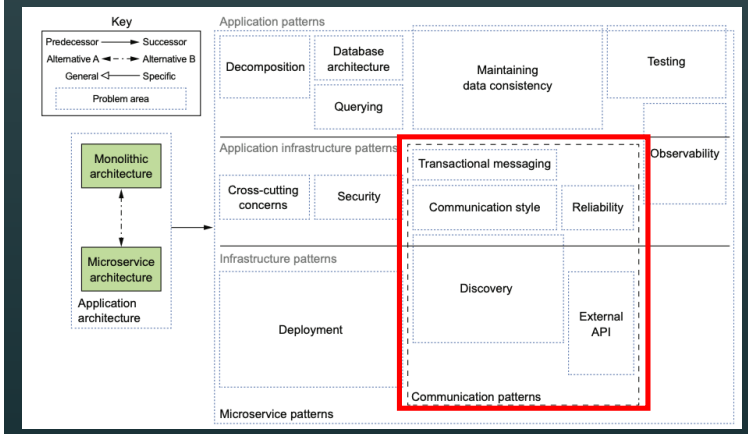
- Overview of interprocess communication in a microservice architecture
- Communicating using the Synchronous Remote procedure invocation pattern
- Communicating using the Asynchronous messaging pattern

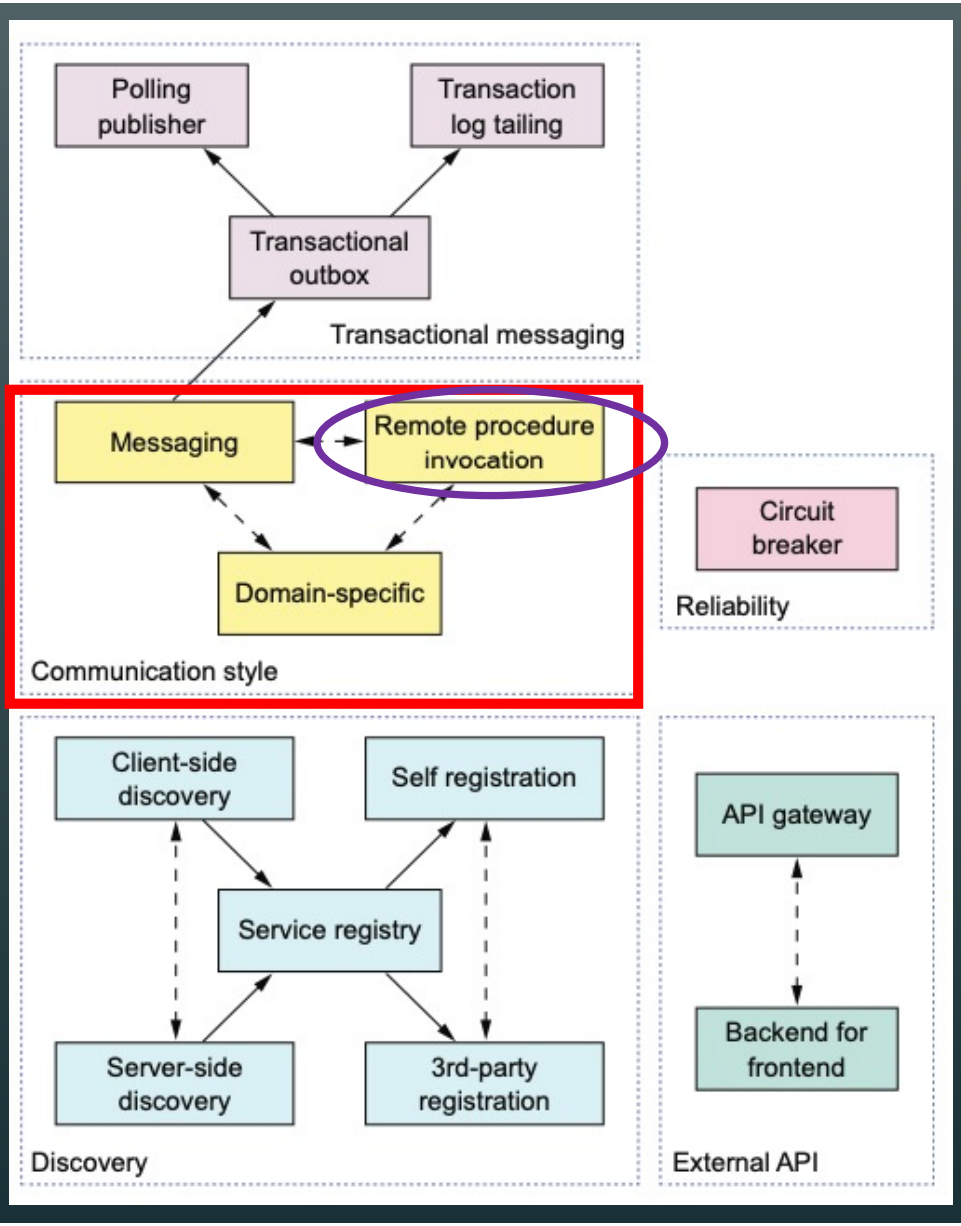
Communicating using the Synchronous Remote procedure invocation pattern

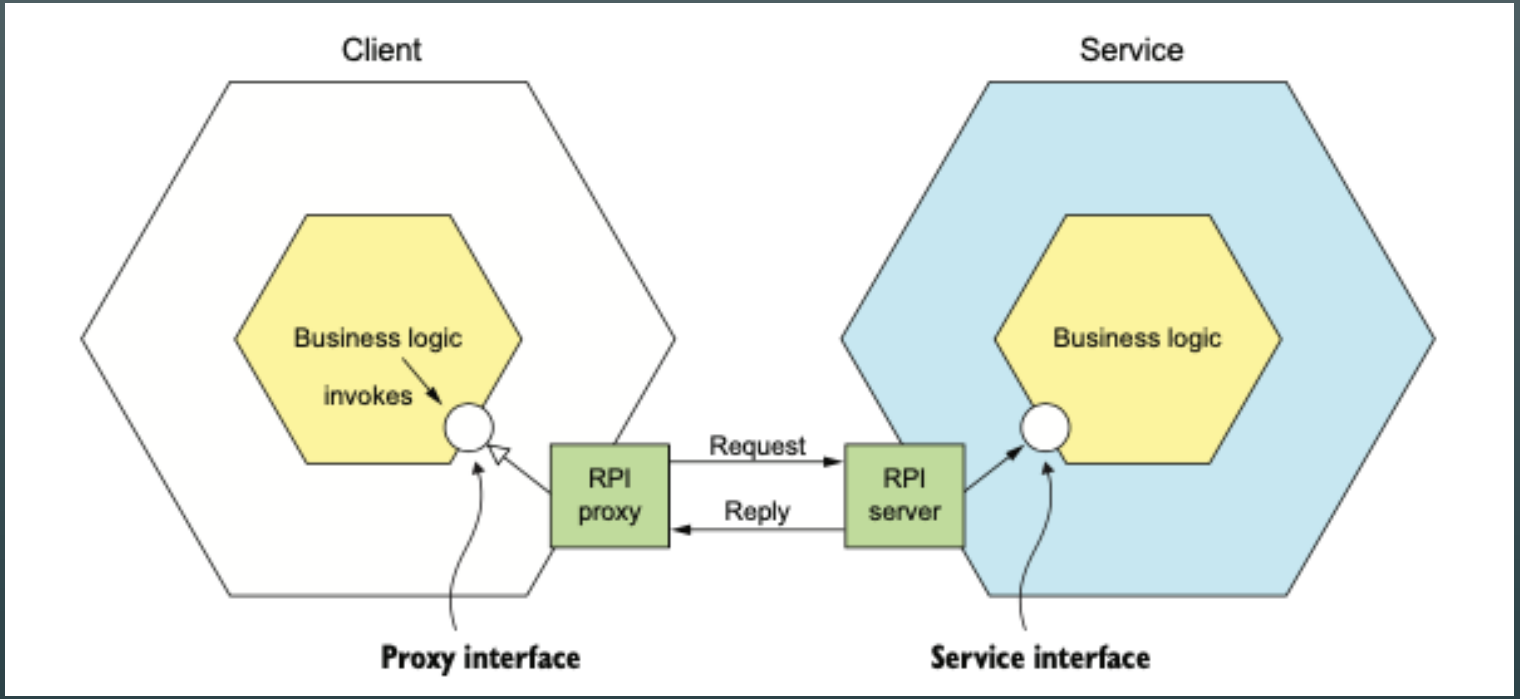
- When using a remote procedure invocation-based IPC mechanism,
  - A client sends a request to a service, and the service processes the request and sends back a response.
  - Some clients may block waiting for a response.
  - Some might have a reactive, non-blocking architecture.
- The client assumes that the response will arrive in a timely fashion.

The five groups of communication patterns









- The client's business logic invokes an interface implemented by an RPI proxy adapter class
- The RPI proxy class makes a request to the service.
- The RPI server adapter class handles the request by invoking the service's business logic.

Representational State Transfer - REST

- Today, it's fashionable to develop APIs in the RESTful style
- Roy Fielding, the creator of REST, defines REST as follows:

> *REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.*

https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Using REST

- A key concept in REST is a *resource*, which typically represents a single business object
  - Customer or Product, or a collection of business objects
- REST uses the HTTP verbs (method) for manipulating resources, which are referenced using a URL
- For example, a GET request returns the representation of a resource, (often in the form of an XML document or JSON object)

Specifying REST APIs

- As mentioned earlier, you must define your APIs using an interface definition language (IDL).
- Unlike older communication protocols like CORBA and SOAP, REST did not originally have an IDL
- The most popular REST IDL is the Open API Specification (www.openapis.org), which evolved from the Swagger open-source project

Benefits of using REST

- It's simple and familiar.
- You can test an HTTP API from within a browser using, for example, the Postman plugin, or from the command line using curl (assuming JSON or some other text format is used).
- It directly supports request/response style communication.
- HTTP is firewall friendly.
- It doesn't require an intermediate broker, which simplifies the system's architecture.

Drawbacks of using REST

- It only supports the request/response style of communication.
- Reduced availability.
  - Client and service communicate directly without an intermediary to buffer messages.
  - They must both be running for the duration of the exchange.
- Clients must know the locations (URLs) of the service instances(s)
  - Using *service discovery* mechanism
- Fetching multiple resources in a single request is challenging.

Google Protocol RPC - gRPC

- A framework for writing cross-language clients and servers
  - (See details at - https://en.wikipedia.org/wiki/Remote_procedure_call)
- gRPC APIs use a Protocol Buffers-based IDL
  - Google's language-neutral mechanism for serializing structured data
  - Supporting both simple request/response RPC and streaming RPC
  - Binary messages in the Protocol Buffers format using HTTP/2
  - (See details at - https://www.grpc.io)

An excerpt of the gRPC API for the Order Service

```
service OrderService {
    rpc createOrder(CreateOrderRequest) returns (CreateOrderReply) {}
    rpc cancelOrder(CancelOrderRequest) returns (CancelOrderReply) {}
    rpc reviseOrder(ReviseOrderRequest) returns (ReviseOrderReply) {}
    ...
}
message CreateOrderRequest {
    int64 restaurantId = 1;
    int64 consumerId = 2;
    repeated LineItem lineItems = 3;
    ...
}
message LineItem {
    string menuItemId = 1;
    int32 quantity = 2;
}
message CreateOrderReply {
    int64 orderId = 1;
}
...
```

Benefits of using gRPC

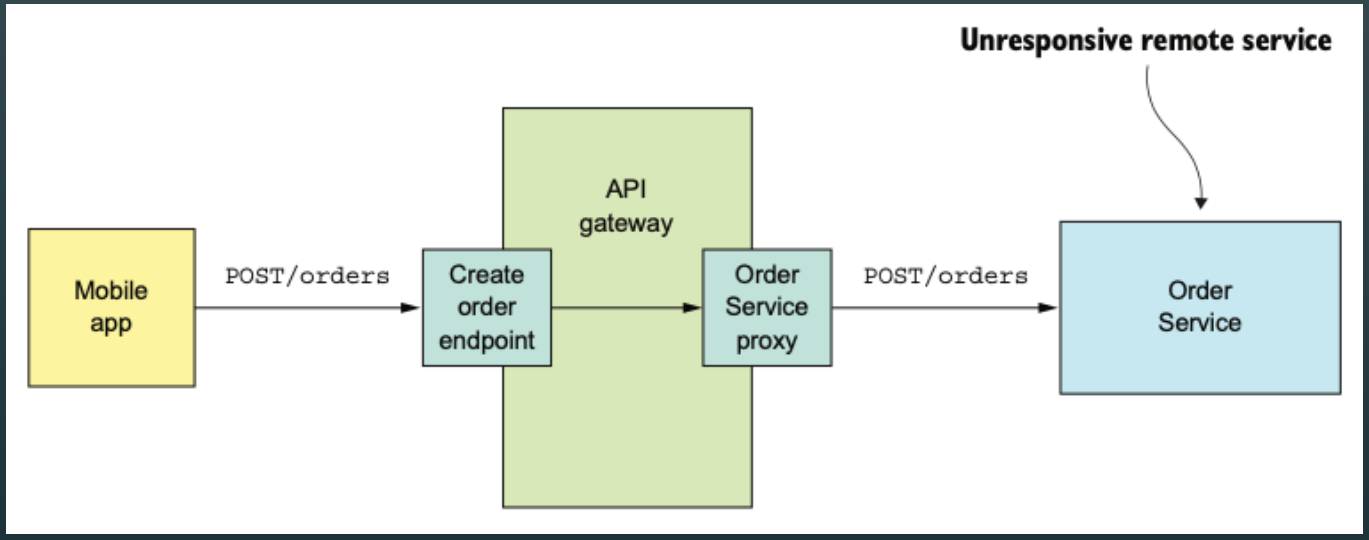
- It's straightforward to design an API with a rich set of update operations.
- It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- Bidirectional streaming enables both RPI and messaging styles of communication.
- It enables interoperability between clients and services written in a wide range of languages.

Drawbacks of using gRPC

- It takes more work for JavaScript clients to consume gRPC-based API comparison to REST/JSON-based APIs.
- Older firewalls might not support HTTP/2.

Handling partial failure using the Circuit breaker pattern (1)

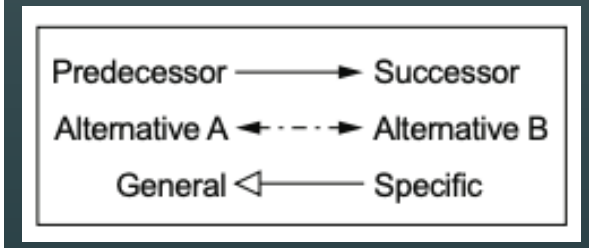
- Using a synchronous request → a risk of partial failure
- An API gateway must protect itself from unresponsive services

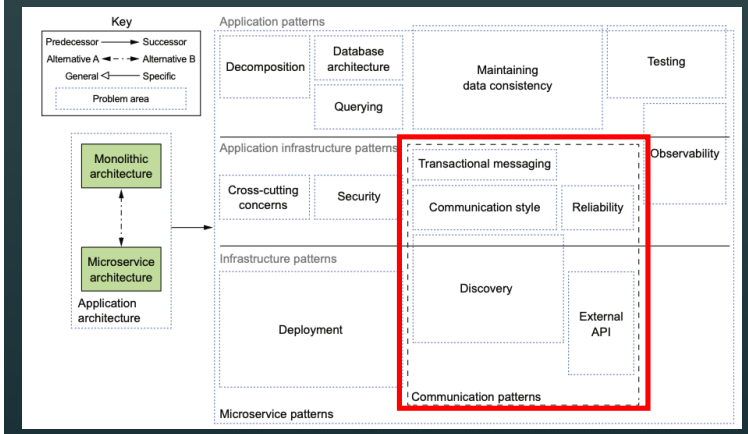


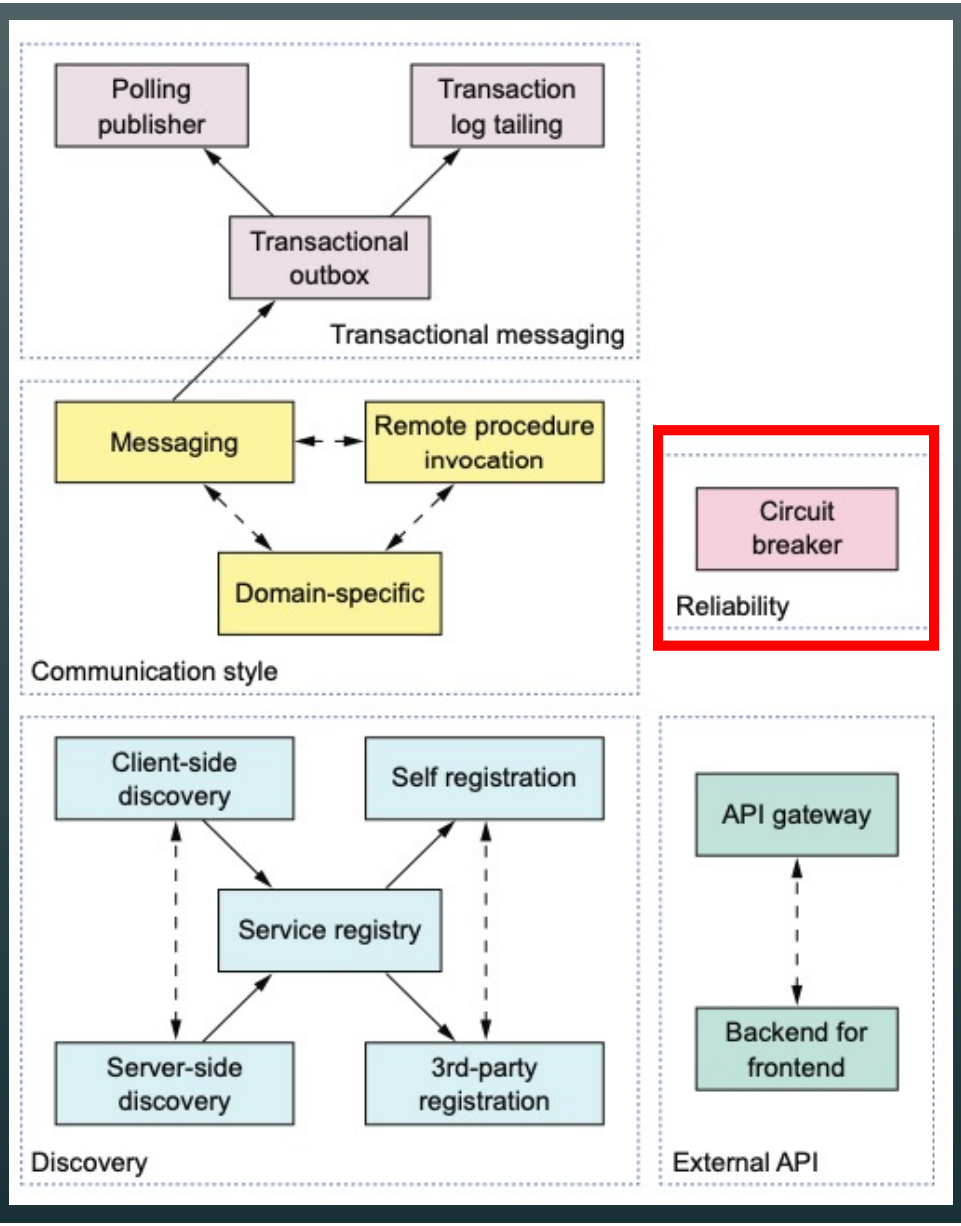
Handling partial failure using the Circuit breaker pattern (2)

- It's essential to design your services to prevent partial failures
- There are two parts to the solution:
  1. Design RPI proxies to handle unresponsive remote services.
  2. Decide how to recover from a failed remote service.

The five groups of communication patterns







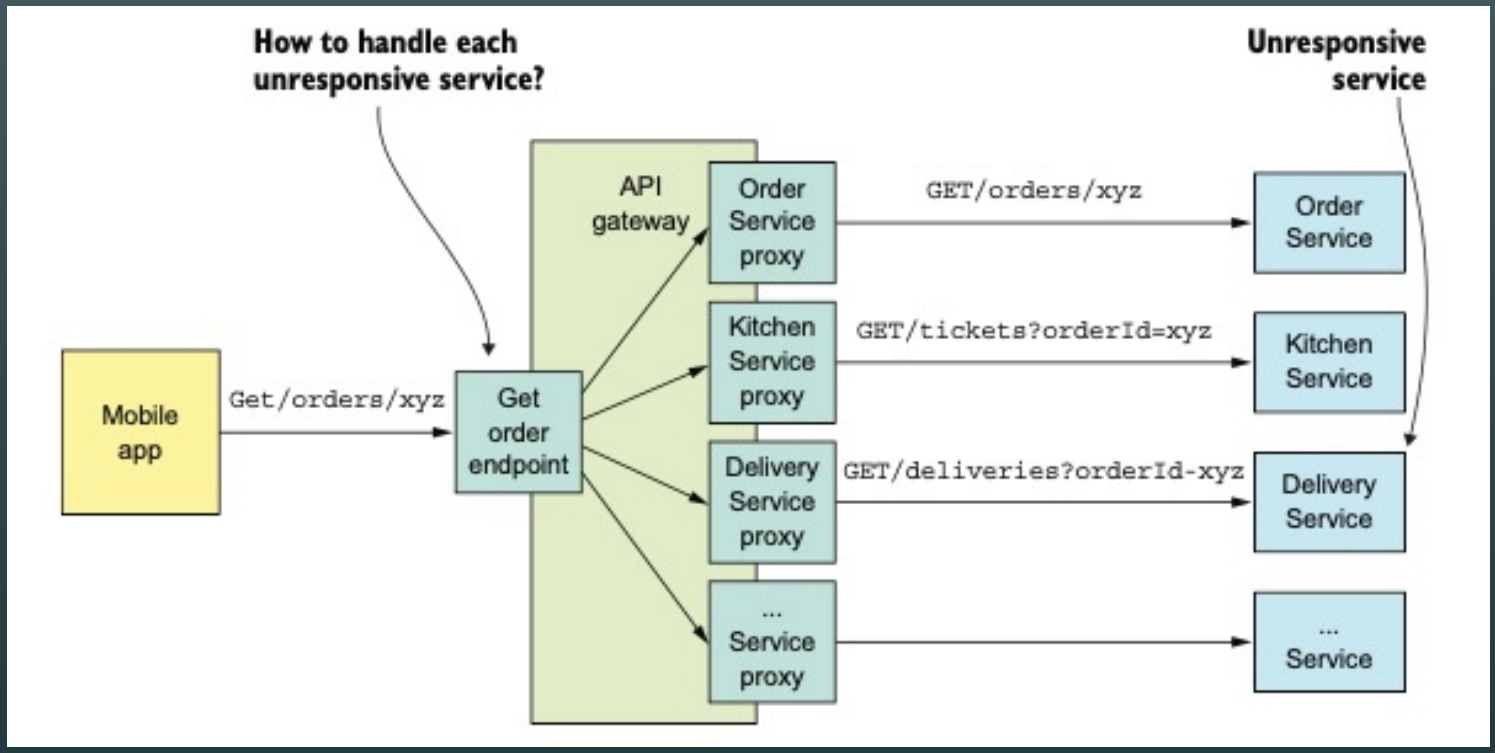
Part 1 – Developing Robust RPI Proxies

1. Network timeouts
   - Never block indefinitely and always use timeouts when waiting for a response.
2. Limiting the number of outstanding requests from a client to a service
   - Impose an upper bound on the number of outstanding requests that a client can make to a particular service.
3. Circuit breaker pattern
   - Track the number of successful and failed requests
   - If the error rate exceeds some threshold, trip the circuit breaker so that further attempts fail immediately.

Part 2 – Recovering from an unavailable server (1)

- We must also decide how your services should recover from an unresponsive remote service
- Option 1 - Return an error to its client
- Option 2 - Returning a fallback value

Part 2 – Recovering from an unavailable server (2)

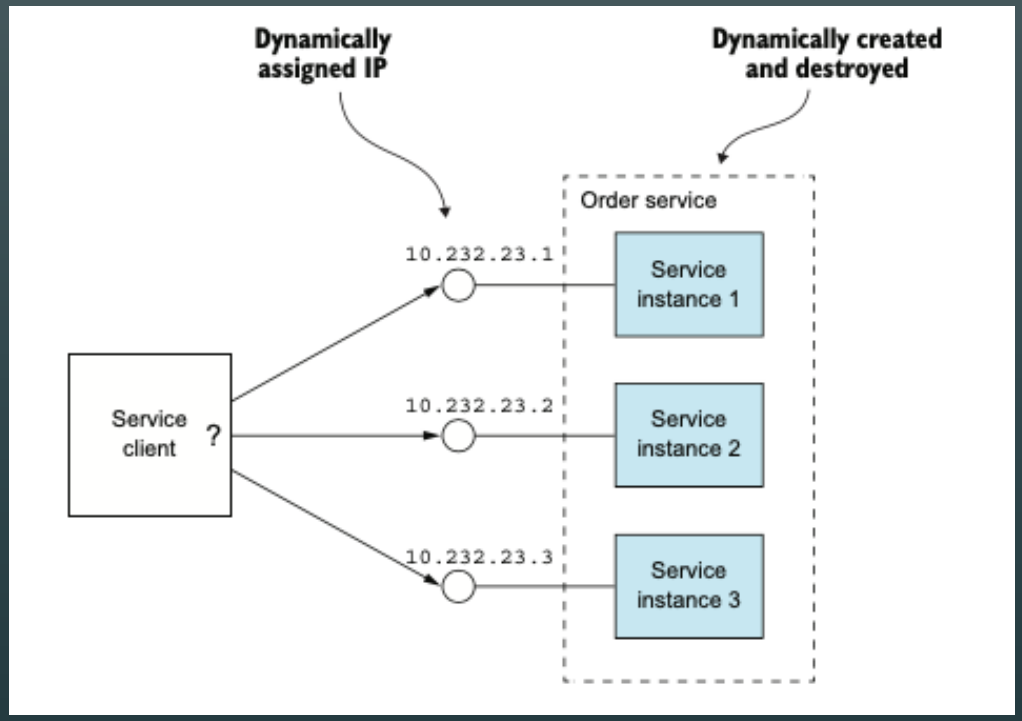


- The endpoint must have a strategy for handling the failure of each service that it calls.

Using Service Discovery (1)

- Assume that some code invokes a service that has a REST API
- The network location (IP address and port) of a service instance is needed
- In a traditional application, running on physical hardware,
  the network locations of service instances are usually static
- But in a modern, cloud-based microservices application,
  a modern application is more dynamic.

Using Service Discovery (2)

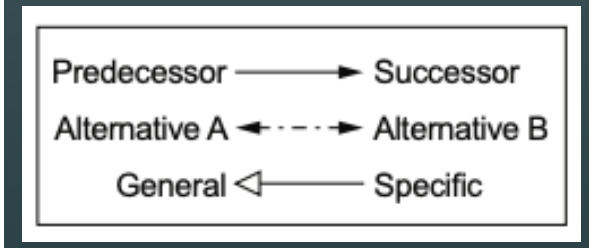


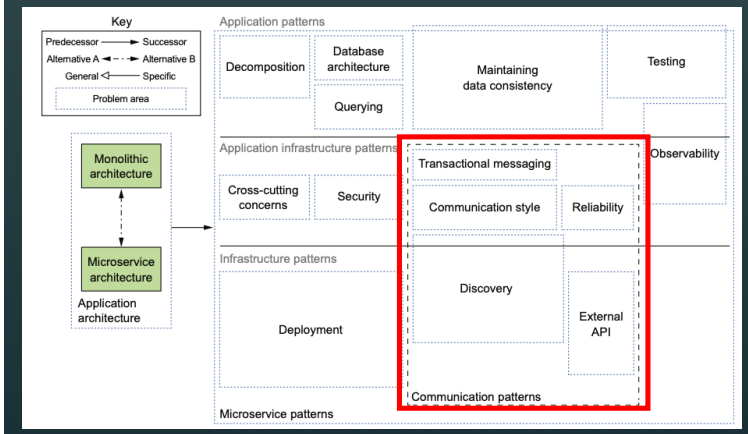
- Service instances have dynamically assigned IP addresses.

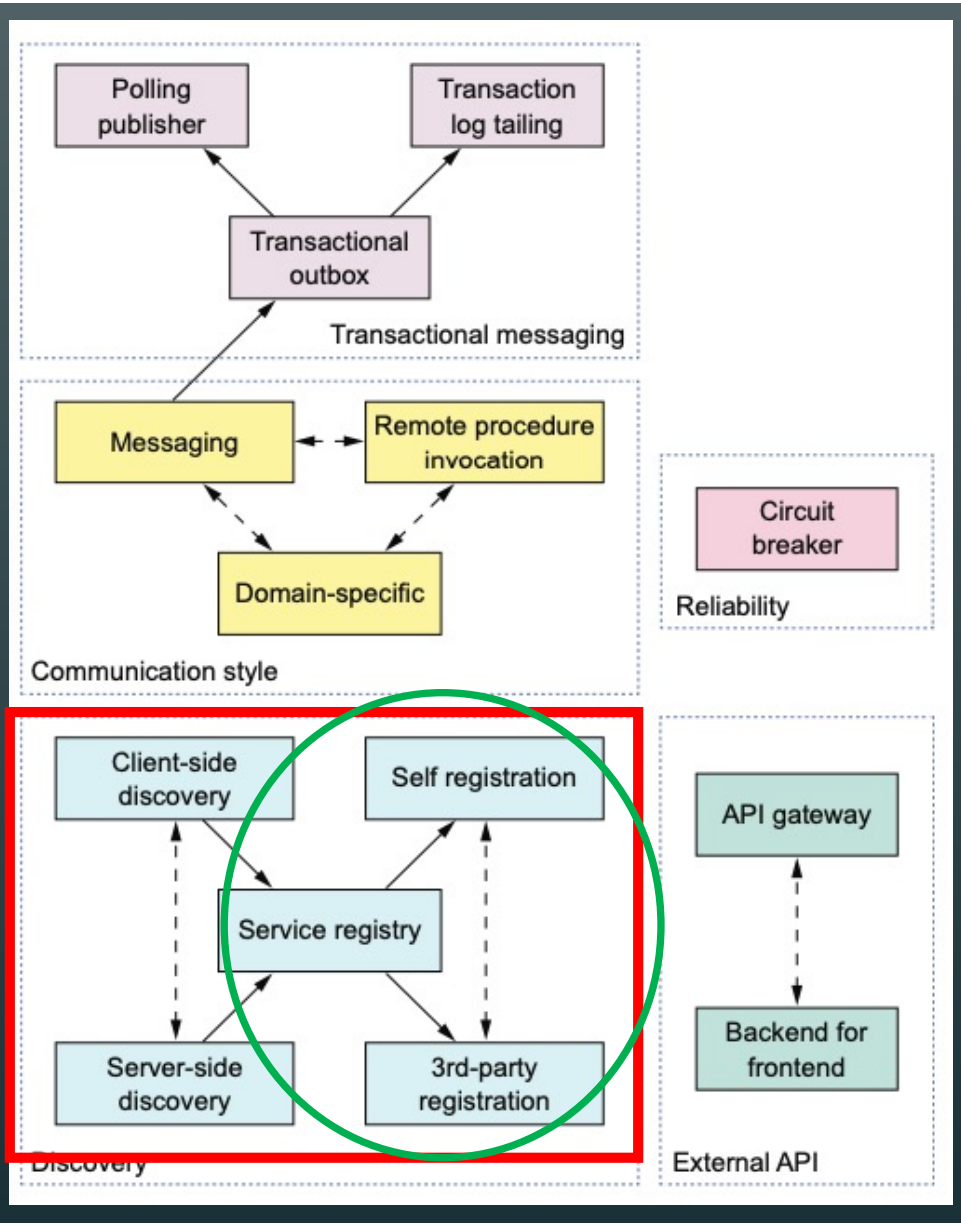
Using Service Discovery (3)

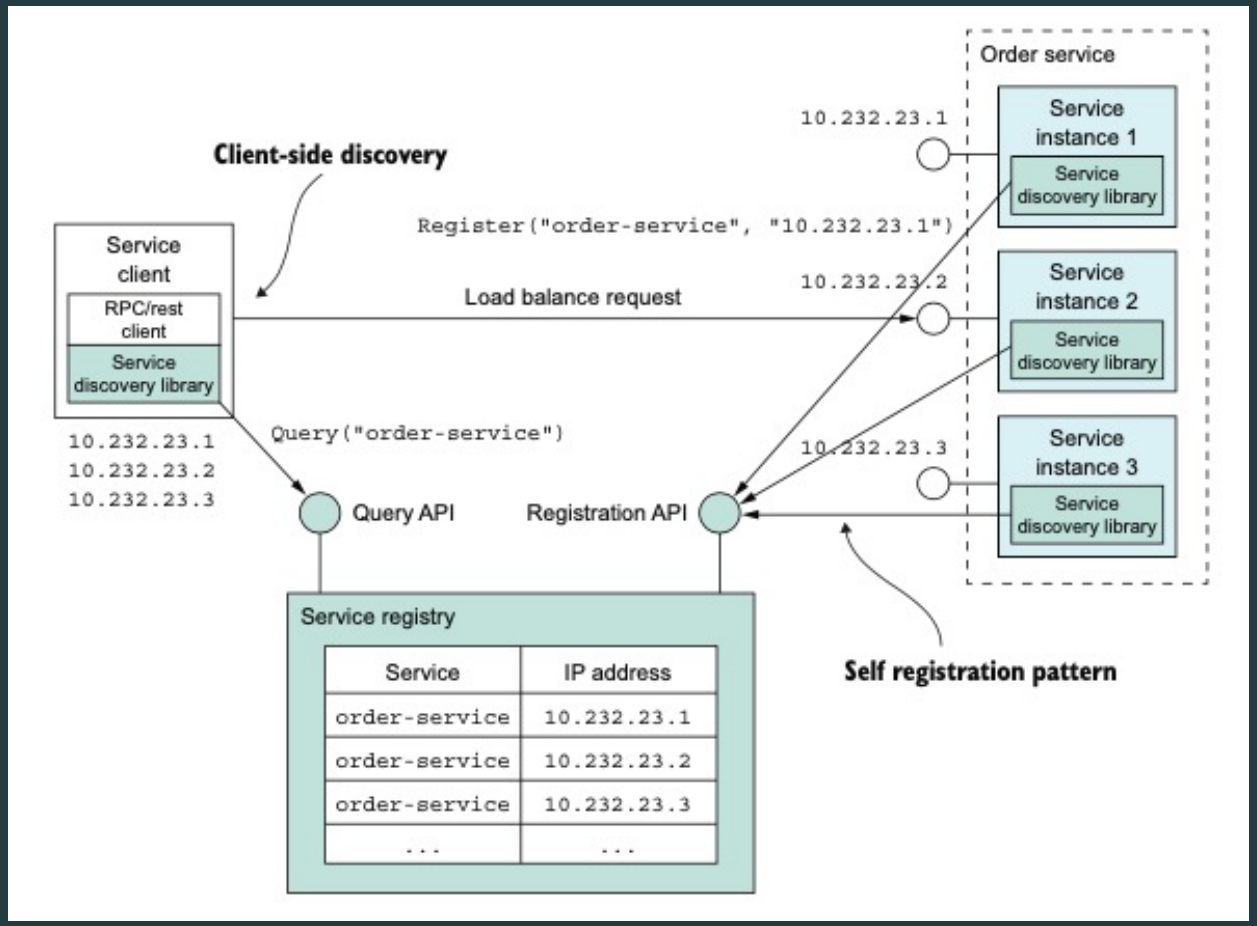
- Service discovery key component is a service registry
  - A database of the network locations of an application's service instances.
- The service discovery mechanism updates the service registry when service instances start and stop
- There are two main ways to implement service discovery:
  1. The services and their clients interact directly with the service registry.
  2. The deployment infrastructure handles service discovery.

The five groups of communication patterns



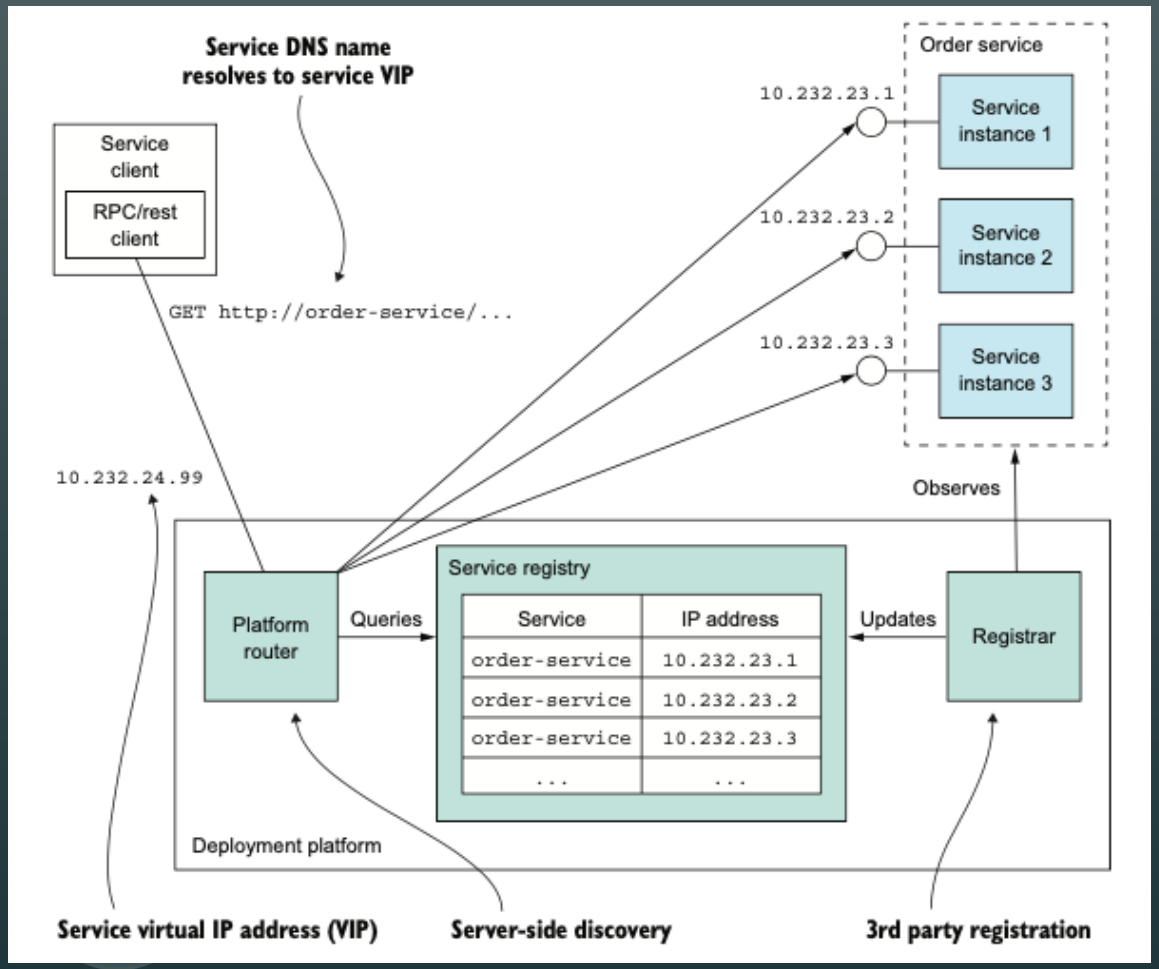






Option 1 – Apply the application-level service discovery pattern

- Implement service discovery for the application's services and their clients
- This approach combine two patterns
  - Self registration pattern
  - Client-side discovery pattern



Option 2 – Apply the platform-provided service discovery pattern

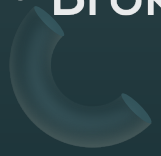
- Use a built-in service registry of modern deployment platforms
  - e.g. Docker and Kubernetes
- This approach combine two patterns
  - 3rd party registration pattern
  - Server-side discovery pattern

Agenda

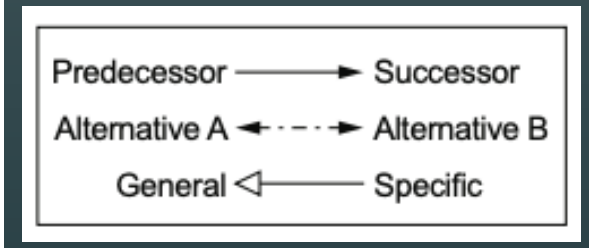
- Overview of interprocess communication in a microservice architecture
- Communicating using the Synchronous Remote procedure invocation pattern
- Communicating using the Asynchronous messaging pattern

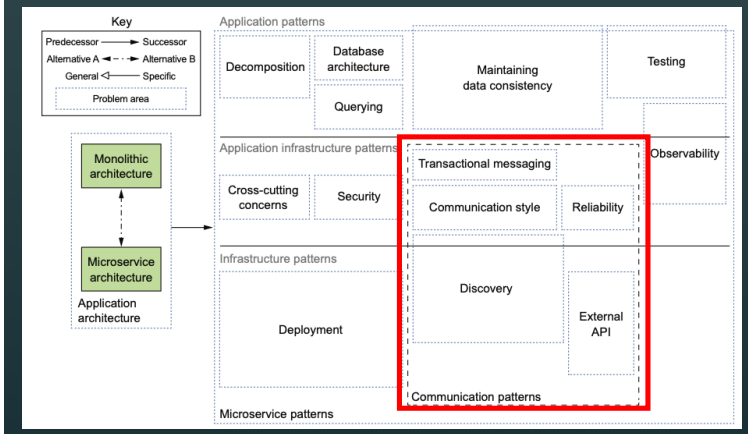
Communicating using the Asynchronous messaging pattern

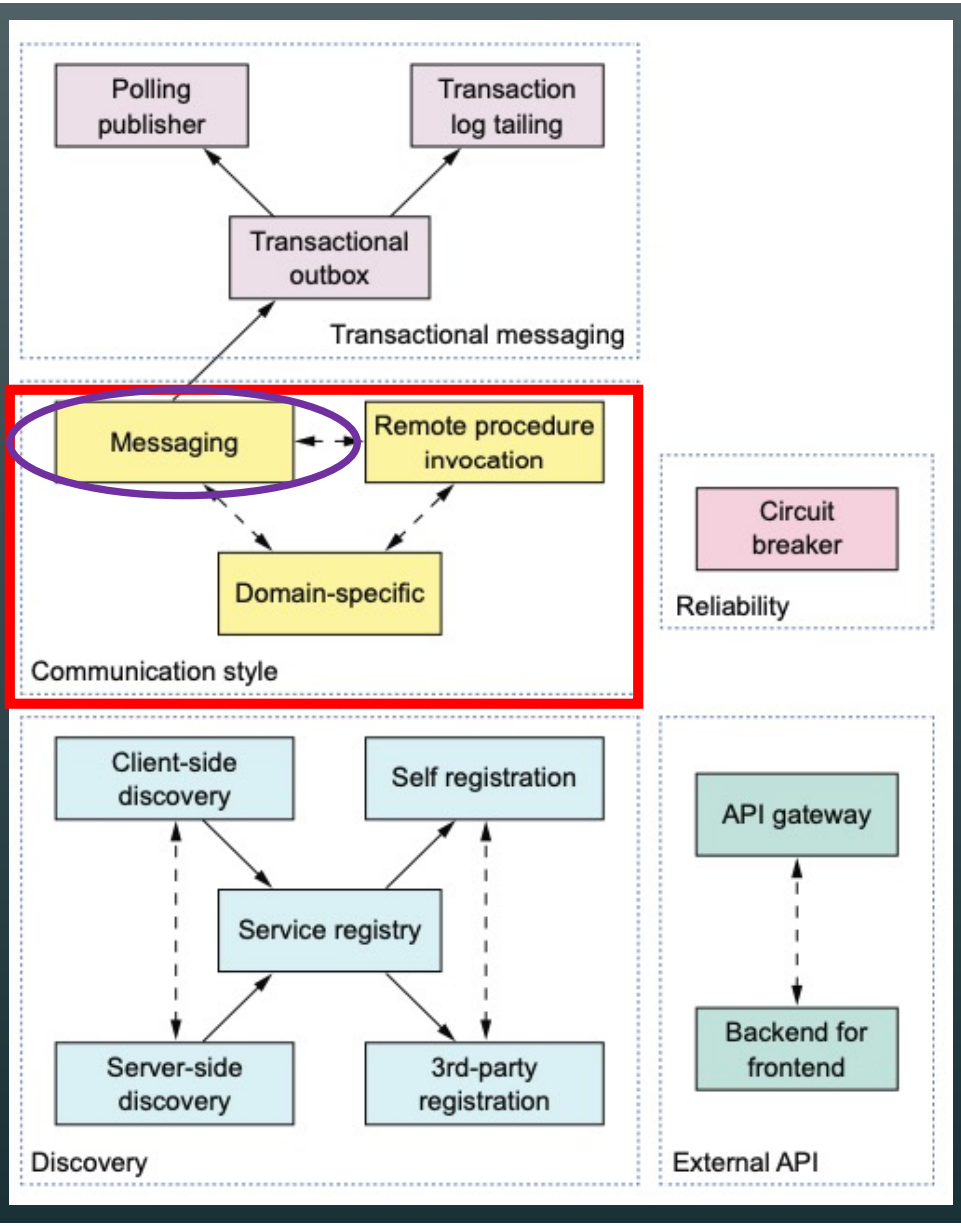
- Services communicate by asynchronously exchanging messages
  - A service client makes a request to a service by sending a message
  - The client doesn't block waiting for a reply.
  - Instead, the client is written assuming that the reply won't be received immediately.
- Brokerless vs broker-based architectures



The five groups of communication patterns

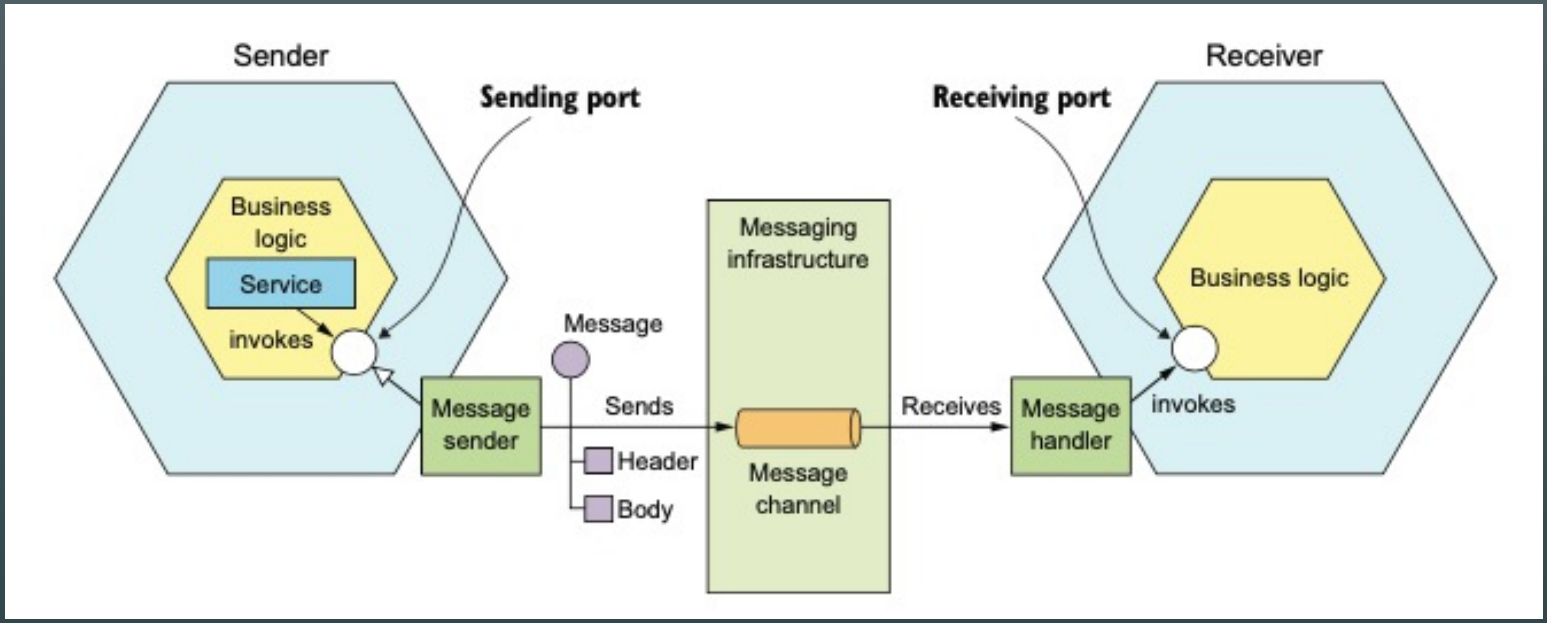






Overview of messaging (1)

- Messages are exchanged over message channels.
  - A sender (an application or service) writes a message to a channel.
  - A receiver (an application or service) reads messages from a channel.
- There are several different kinds of messages:
  - Document—A generic message that contains only data.
  - Command—A message that's the equivalent of an RPC request.
  - Event—A message indicating that something notable has occurred in the sender.



1. The business logic in the sender invokes a *sending port* interface.
2. The *message sender* sends a message to a receiver via a *message channel*.
3. The *message channel* is an abstraction of messaging infrastructure.
4. A *message handler* adapter in the receiver is invoked to handle the message.
5. It invokes the *receiving port* interface implemented by the receiver's business logic.

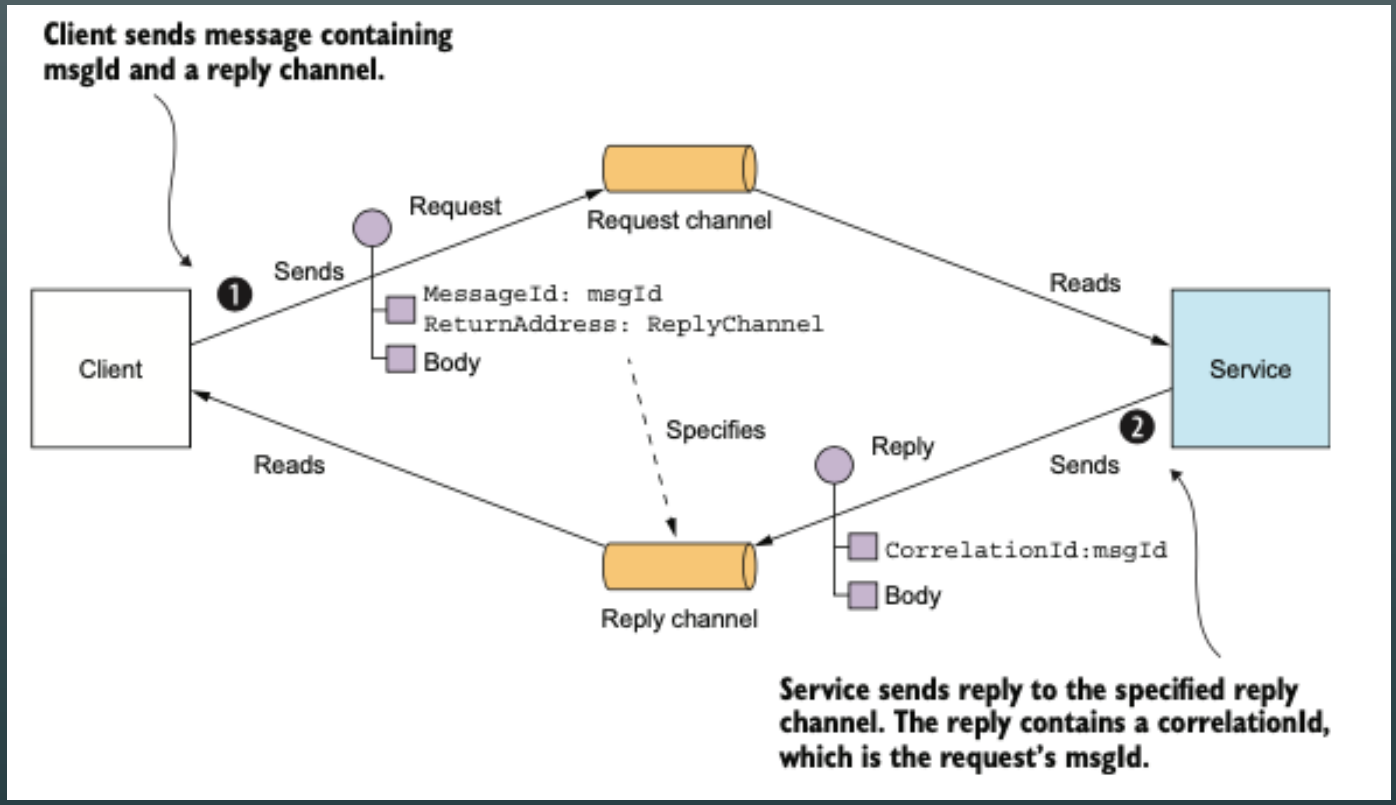
Overview of messaging (2)

- There are two kinds of channels:
- A point-to-point channel
  - Delivers a message to exactly **one** of the consumers
  - For example, a command message is sent over a point-to-point channel.
- A publish-subscribe channel
  - Delivers each message to **all** of the attached consumers.
  - For example, an event message is sent over a publish-subscribe channel.

Implementing the interaction styles using messaging (1)

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

1. Request/response and Asynchronous Request/response
2. One-way notifications
3. Publish/subscribe
4. Publish/asynchronous subscribe



1. Implementing asynchronous request/response by including a *reply channel* and *message identifier* in the request message.
2. The receiver processes the message and sends the reply to the *specified reply channel*.

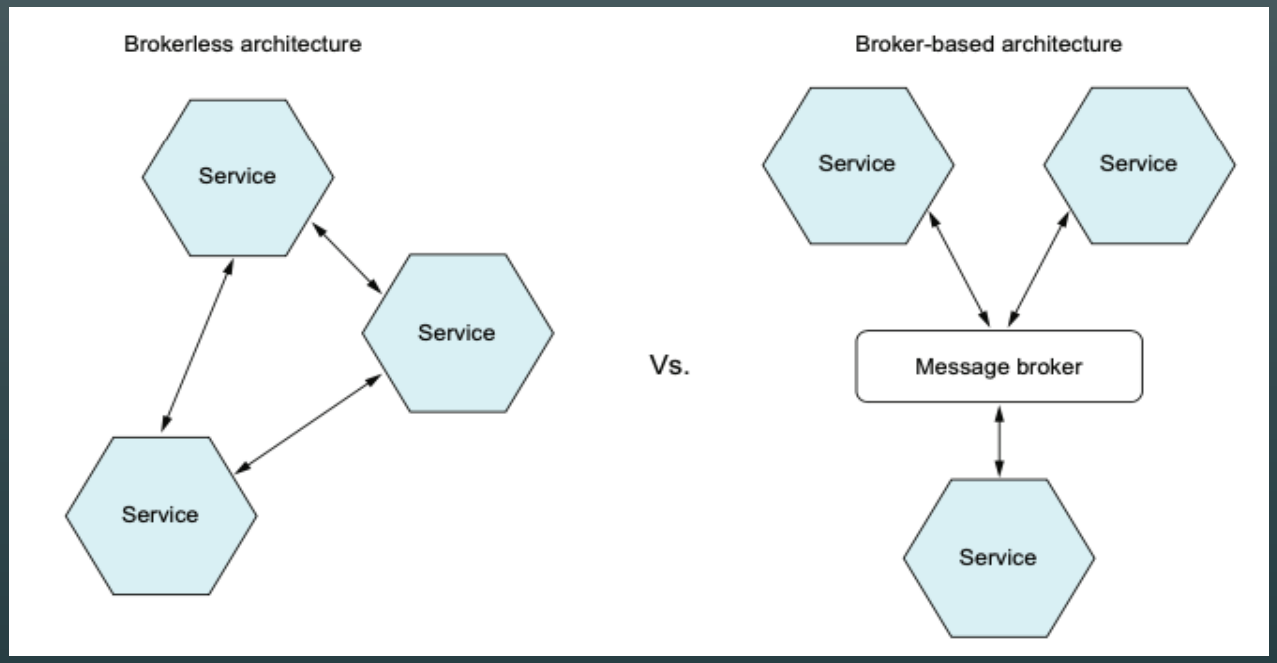
Implementing the interaction styles using messaging (2)

- Request/response and Asynchronous Request/response
  - Messaging is inherently asynchronous, so only provides asynchronous request/response.
  - But a client could block until a reply is received.
- One-way notifications
  - The service subscribes to the channel and processes the message.
  - But it doesn't send back a reply.

Implementing the interaction styles using messaging (3)

- Publish/subscribe
  - A client publishes a message to a publish-subscribe channel that is read by multiple consumers
  - A service that's interested in a particular domain object's events only has to subscribe to the appropriate channel.
- Publish/asynchronous subscribe
  - Combining elements of publish/subscribe and request/response

Brokerless vs Broker-based architecture



- We will focus on broker-based architecture
- But it's worthwhile to take a quick look at the brokerless architecture

Benefits and drawbacks of Brokerless architecture

- Benefits:
  - Allows lighter network traffic and better latency,
  - Eliminates the possibility a performance bottleneck or a single point of failure
  - Features less operational complexity
- Drawbacks:
  - Services must use one of the discovery mechanisms
  - Availability is reduced, because both the sender and receiver must be available while the message is being exchanged.

Overview of Broker-based messaging

- A message broker is an intermediary through which all messages flow
- Benefit of broker-based messaging:
  - The sender doesn't need to know the network location of the consumer
  - A message broker buffers messages until the consumer is able to process them
- Popular open source message brokers:
  - ActiveMQ, RabbitMQ, Apache Kafka
- cloud-based messaging services
  - AWS Kinesis, AWS SQS

Benefits and drawbacks of broker-based messaging

- Benefits:
  - Loose coupling – The client doesn't need to use a discovery mechanism
  - Message buffering
  - Flexible communication - Messaging supports all the interaction styles
  - Explicit interprocess communication
- Drawbacks:
  - Potential performance bottleneck
  - Potential single point of failure
  - Additional operational complexity

Competing receivers and message ordering (1)

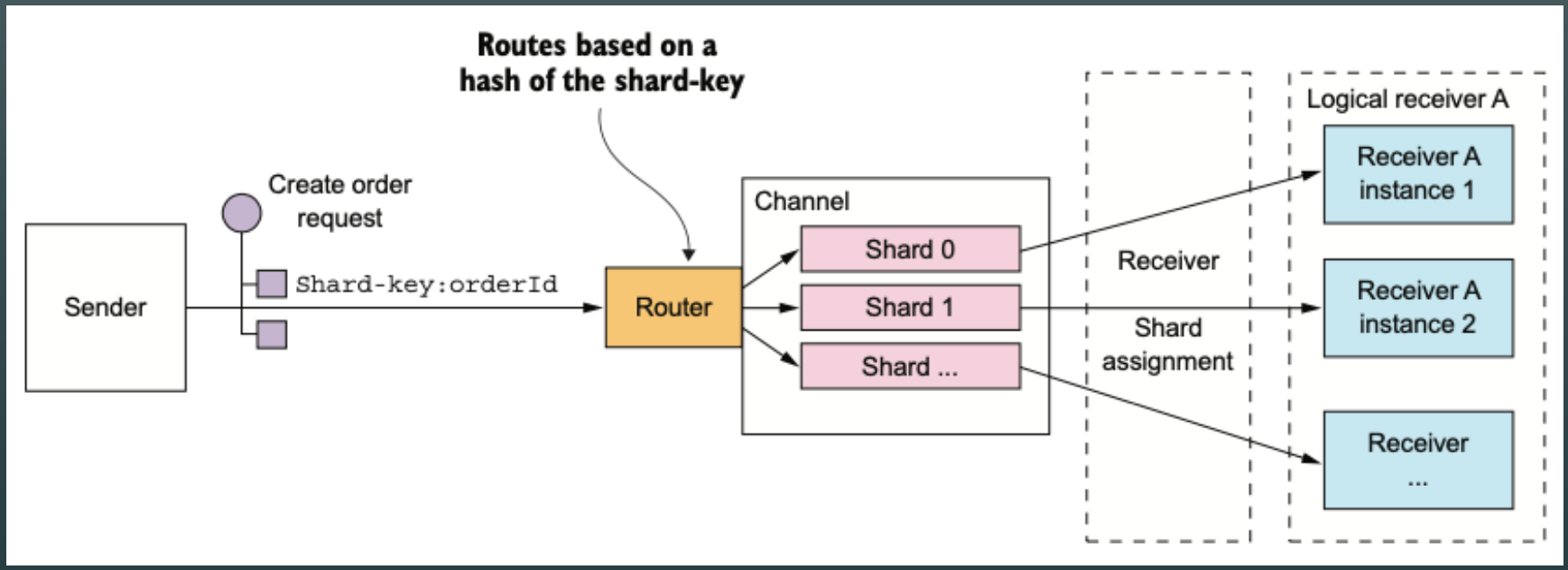
- Imagine that there are three instances of a service reading from the same point-to-point channel

  1. Order Created, 2. Order Updated, and 3. Order Cancelled

- Because of delays due to network issues or garbage collections
  → messages might be processed out of order

- A common solution, used by modern message brokers like Apache Kafka and AWS Kinesis, is to use sharded (partitioned) channels

Competing receivers and message ordering (2)

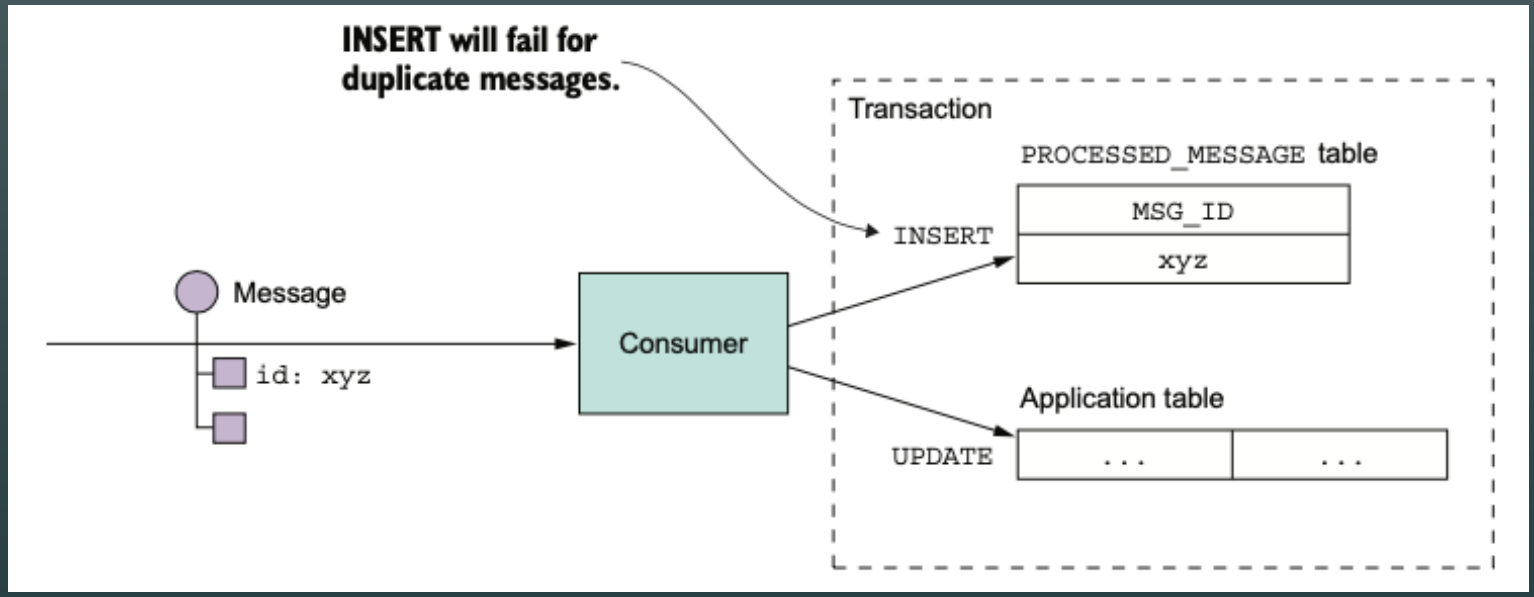


- Each Order event message has the *orderId* as its shard key
- Each event is published to the same shard, which is read by a single consumer instance

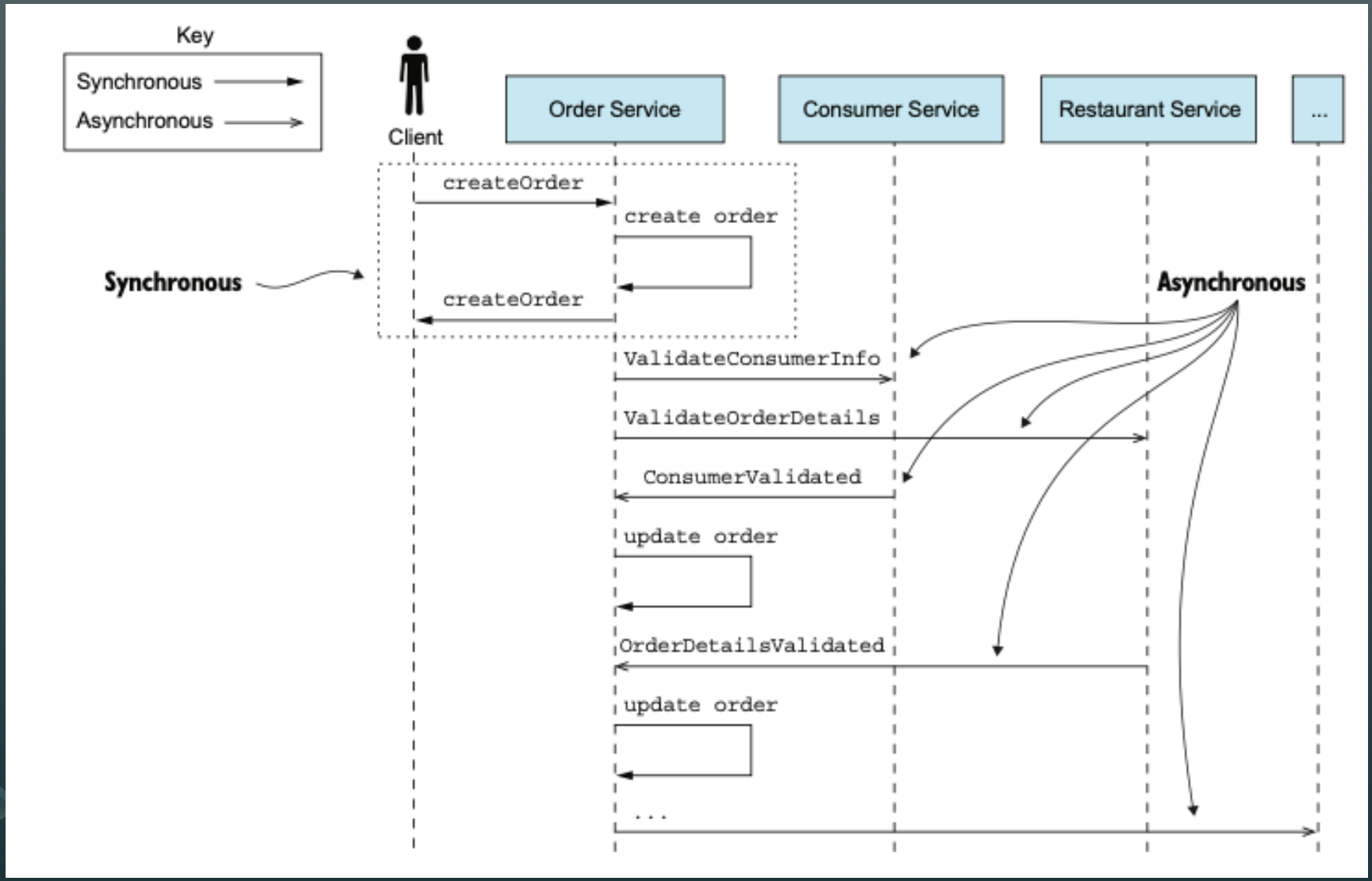
Handling duplicate messages (1)

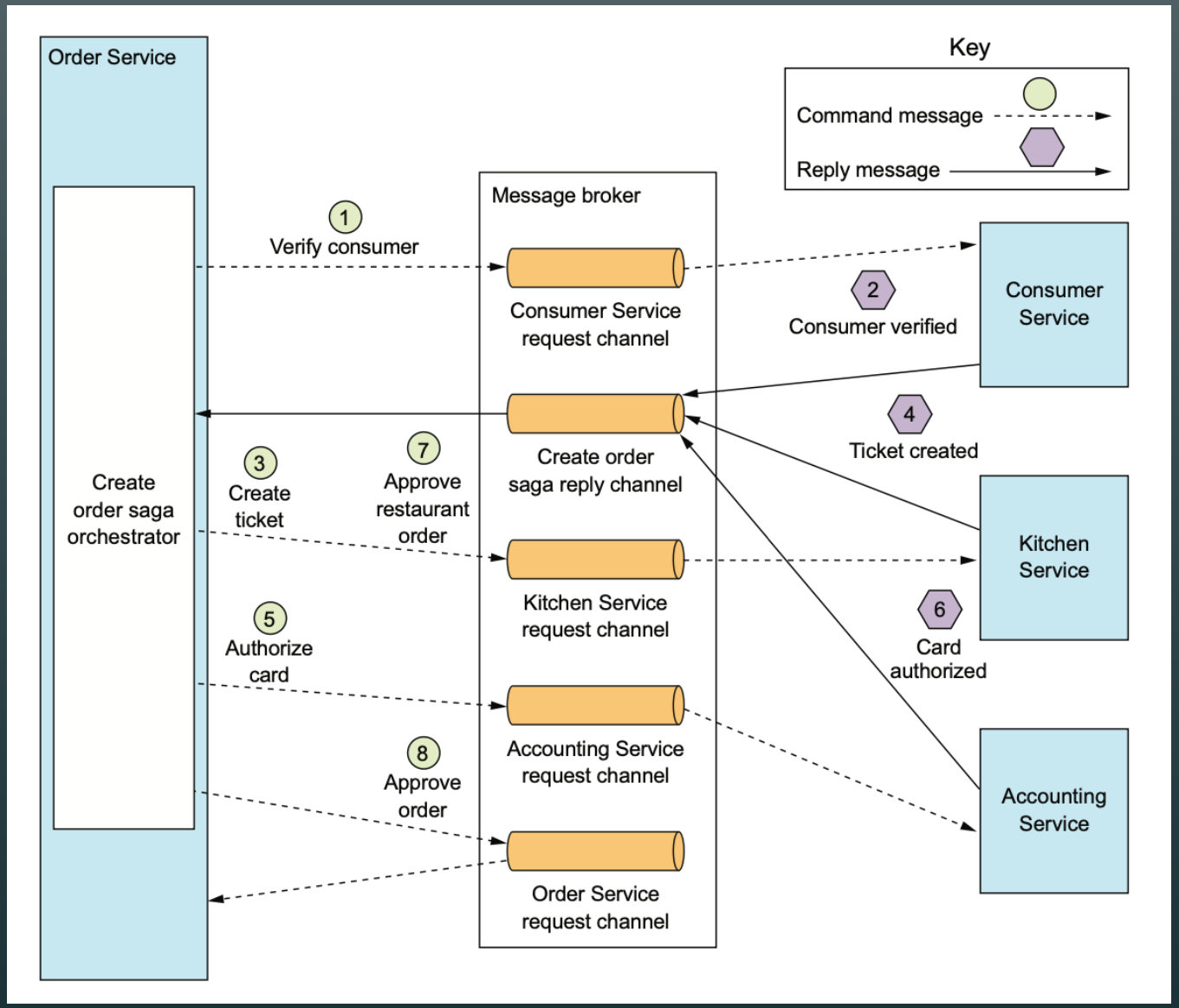
- A failure of a client, network, or message broker can result in a message being delivered multiple times
  - For example, a message handler that authorizes a consumer credit card.
- A message consumer has to discard any duplicates message
  - Tracking by the message id

Handling duplicate messages (2)



- A consumer record the IDs of processed messages in a database table.
- If a message has been processed before, the INSERT into the PROCESSED_MESSAGES table will fail.


Key
Synchronous
Asynchronous
Client
Order Service
Consumer Service
Restaurant Service
...
createOrder
create order
createOrder
Synchronous
Asynchronous
ValidateConsumerInfo
ValidateOrderDetails
ConsumerValidated
update order
OrderDetailsValidated
update order
...


Order Service
Key
Command message
Reply message
Message broker
Create order saga orchestrator
1
Verify consumer
Consumer Service request channel
2
Consumer verified
Consumer Service
Create order saga reply channel
4
Ticket created
3
Create ticket
7
Approve restaurant order
Kitchen Service request channel
Kitchen Service
6
Card authorized
5
Authorize card
Accounting Service request channel
Accounting Service
8
Approve order
Order Service request channel

Summary (1)

- The microservice architecture is a distributed architecture, so interprocess communication plays a key role.
- It's essential to carefully manage the evolution of a service's API.
- There are numerous IPC technologies, each with different trade-offs.
- To prevent failures, a service client that uses a synchronous protocol must be designed to handle partial failures

Summary (2)

- An architecture that uses synchronous protocols must include a service discovery mechanism to determine the network location of a service instance.
- A good way to design a messaging-based architecture is to use the messages and channels model, typically message broker–based