Remote Procedure Call

Lukasz Antoniak

26.01.2022

Agenda

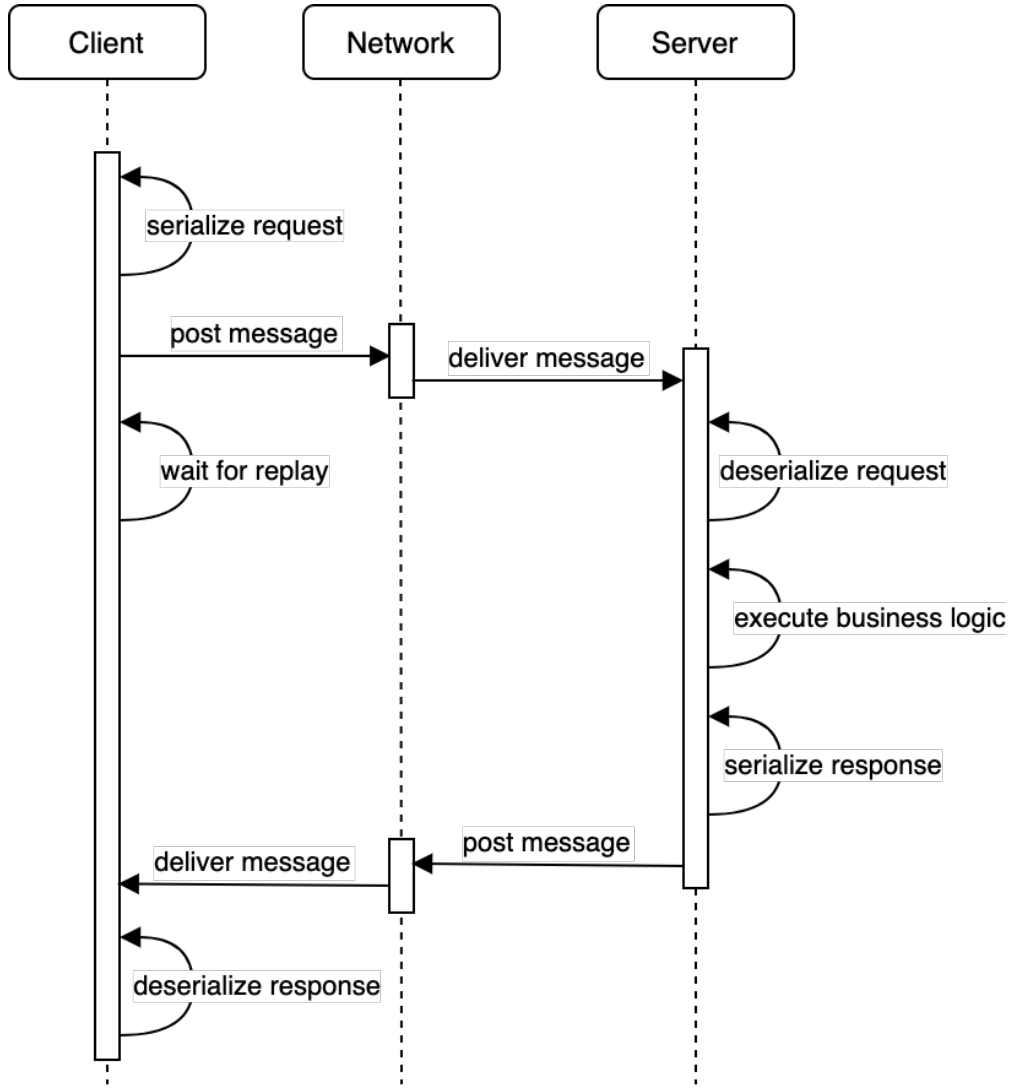
- The need of communication
- Message transport and format
- Synchronous and asynchronous communication patterns
- Traditional load balancers
- Service Registry and Discovery
- Service meshes
- Idempotent service design

Definition

Distributed system is a computing environment, where multiple processes running on different machines, communicate through the network and coordinate actions in order to appear to the end-user as a single coherent system.

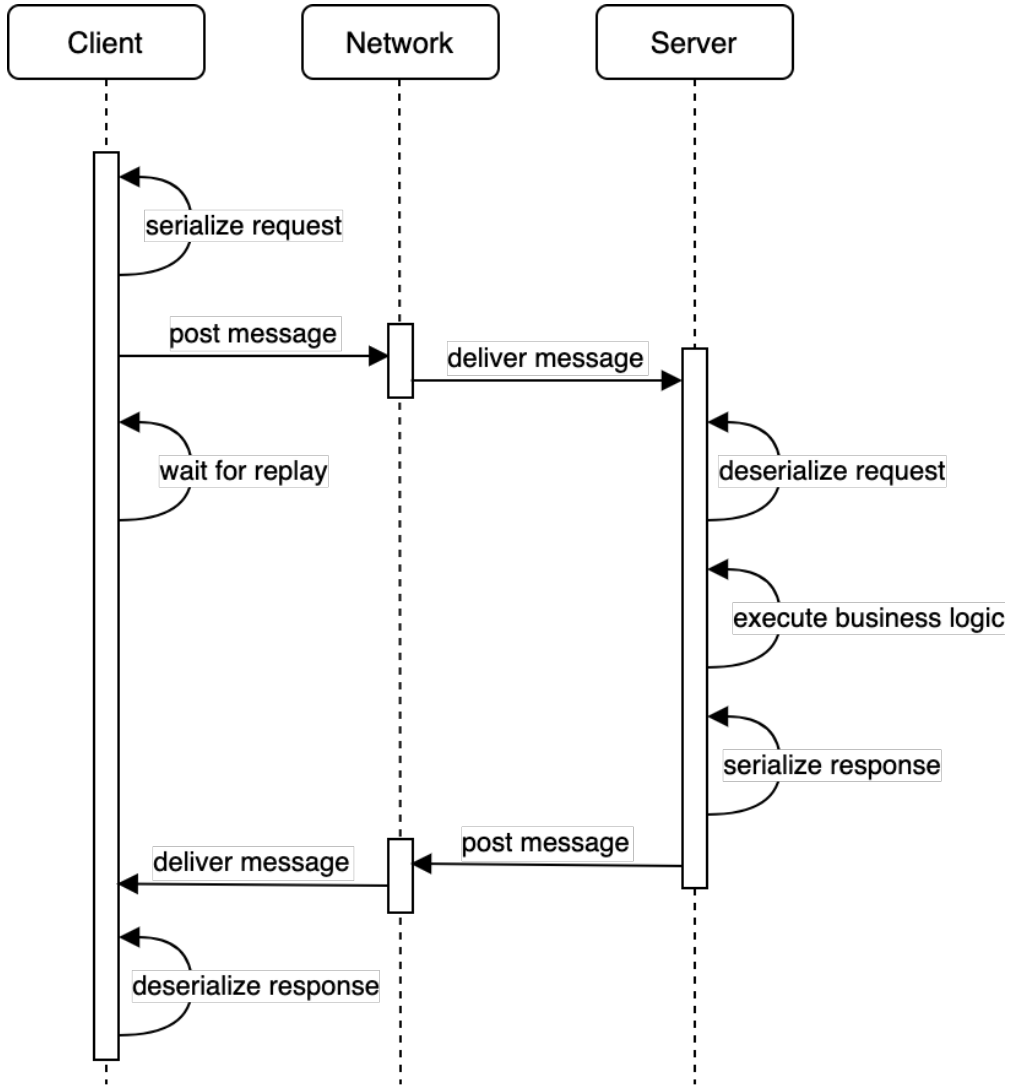

```
sayHello("Lukasz");

private String sayHello(String name) {
    return String.format(
        "Greetings %s!", name
    );
}
```

Steps in RPC


Client
Network
Server
serialize request
post message
deliver message
wait for replay
deserialize request
execute business logic
serialize response
post message
deliver message
deserialize response

Faults in RPC

- What are the consequences of?
  - Client fails to serialize request or send message to the network
  - Server fails to deserialize request e.g., invalid format of the message
  - Server raises error during execution of business logic
  - Network fails to deliver reply to the client or server takes too long to respond (timeout)



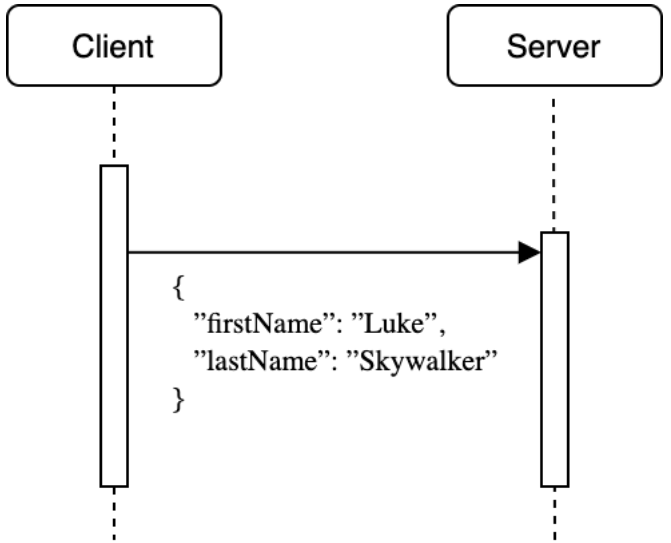
RPC Error Handling

```
(error, reply) = network.send(remote, actionData)
switch error
  case POST_FAILED:
    // handle case where you know server didn't get it
  case RETRYABLE:
    // handle case where server got it but reported transient failure
  case FATAL:
    // handle case where server got it and definitely doesn't like it
  case UNKNOWN: // i.e., time out
    // handle case where the *only* thing you know is that the server received
    // the message; it may have been trying to report SUCCESS, FATAL, or RETRYABLE
  case SUCCESS:
    if validate(reply)
      // do something with reply object
    else
      // handle case where reply is corrupt/incompatible
```

Source: https://aws.amazon.com/builders-library/challenges-with-distributed-systems

Message Transport and Format

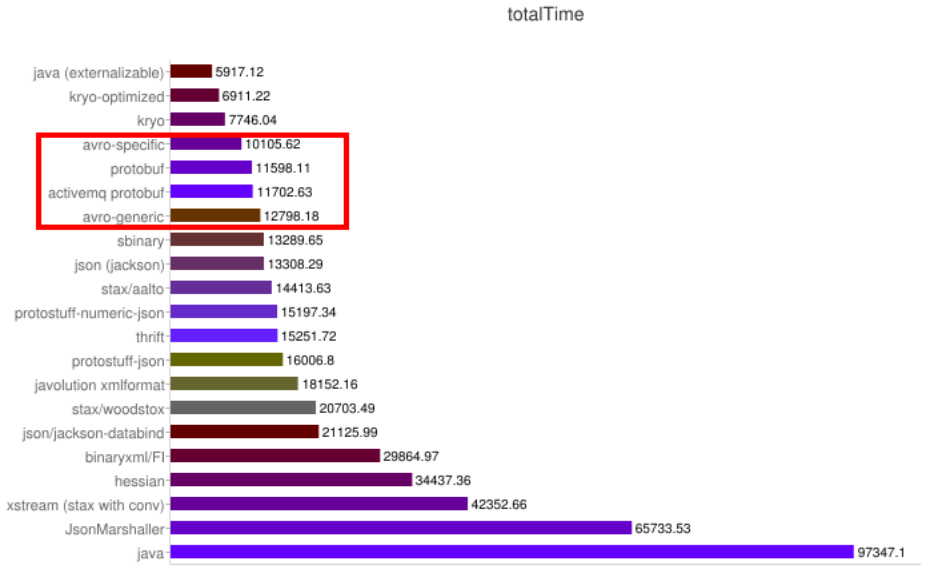
- Message transport defines a protocol used to exchange data between client and a remote server. Examples:
  - TCP or UDP sockets
  - FTP file exchange
  - HTTP
  - Messaging (e.g. JMS, Kafka, MQTT, RabbitMQ)
- Message format specifies how data is being represented. Examples:
  - XML
  - JSON
  - Google Protocol Buffers
  - Apache Avro

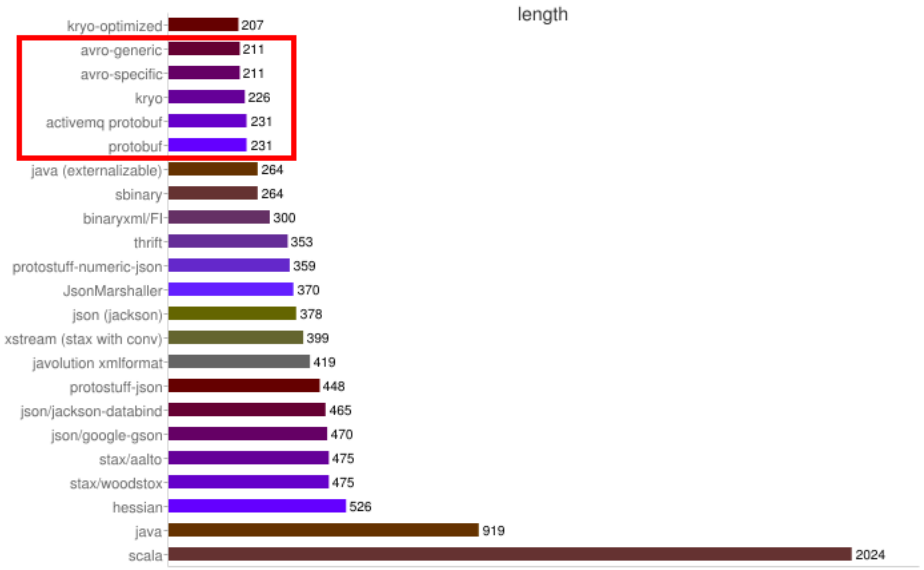



```
<customer>
   <firstName>Luke</firstName>
   <lastName>Skywalker</lastName>
</customer>
```

```
{
   "firstName": "Luke",
   "lastName": "Skywalker"
}
```

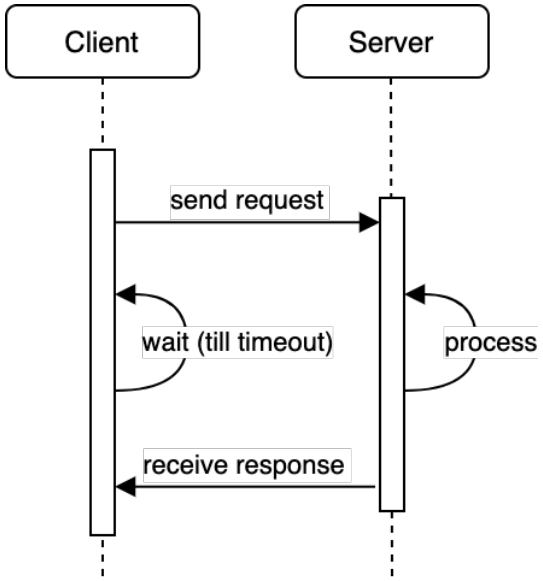
- Performance of modern binary serializers – Avro and ProtoBuf





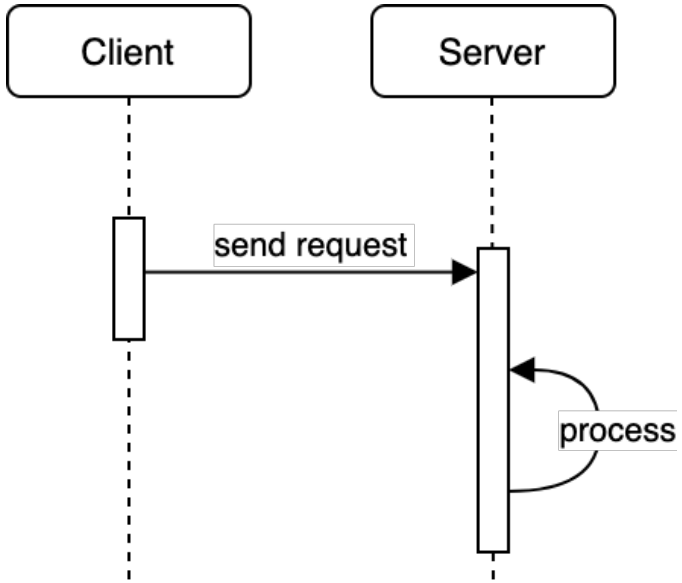
Source: https://blog.softwaremill.com/the-best-serialization-strategy-for-event-sourcing-9321c299632b

Synchronous (Request-Replay)



- Use for short running processes, without human intervention
- Always define timeout at client side
- Implement idempotent services for safe retry upon timeout

Asynchronous (Fire-and-Forget)

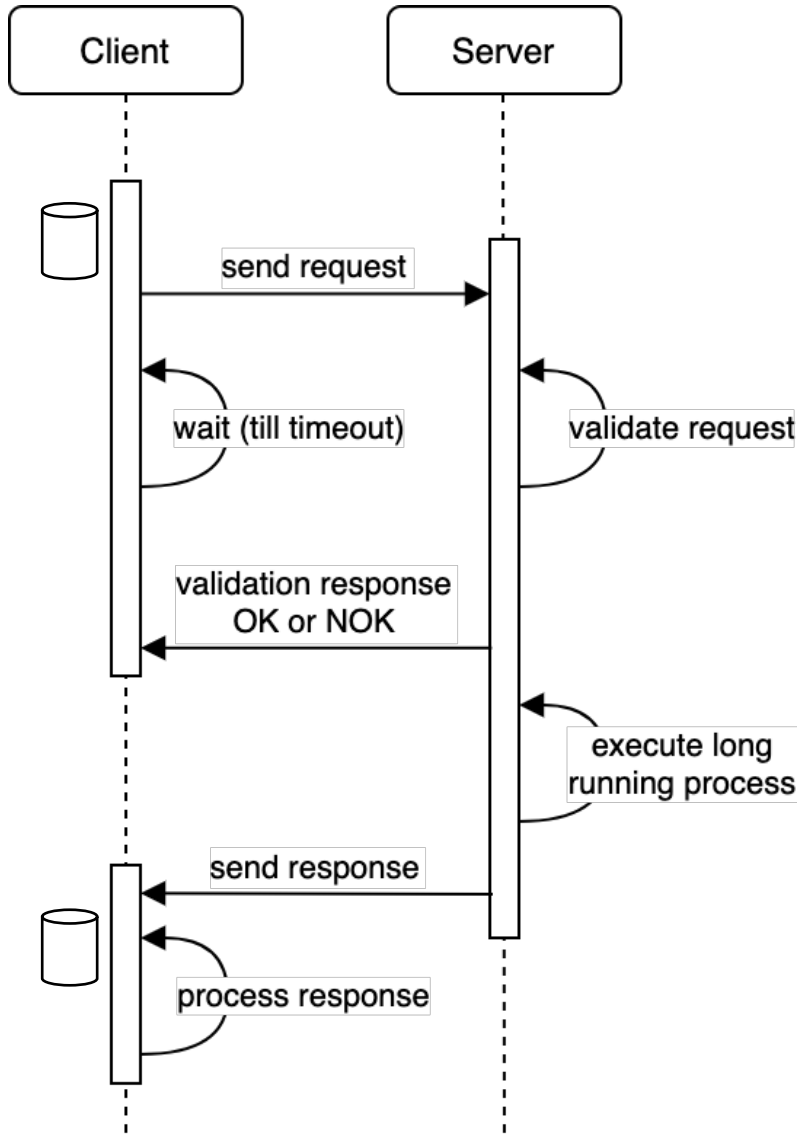


- Use for long running processes or when human intervention required
- Usually requires messaging solution to decouple service consumer from service provider

Communication Patterns

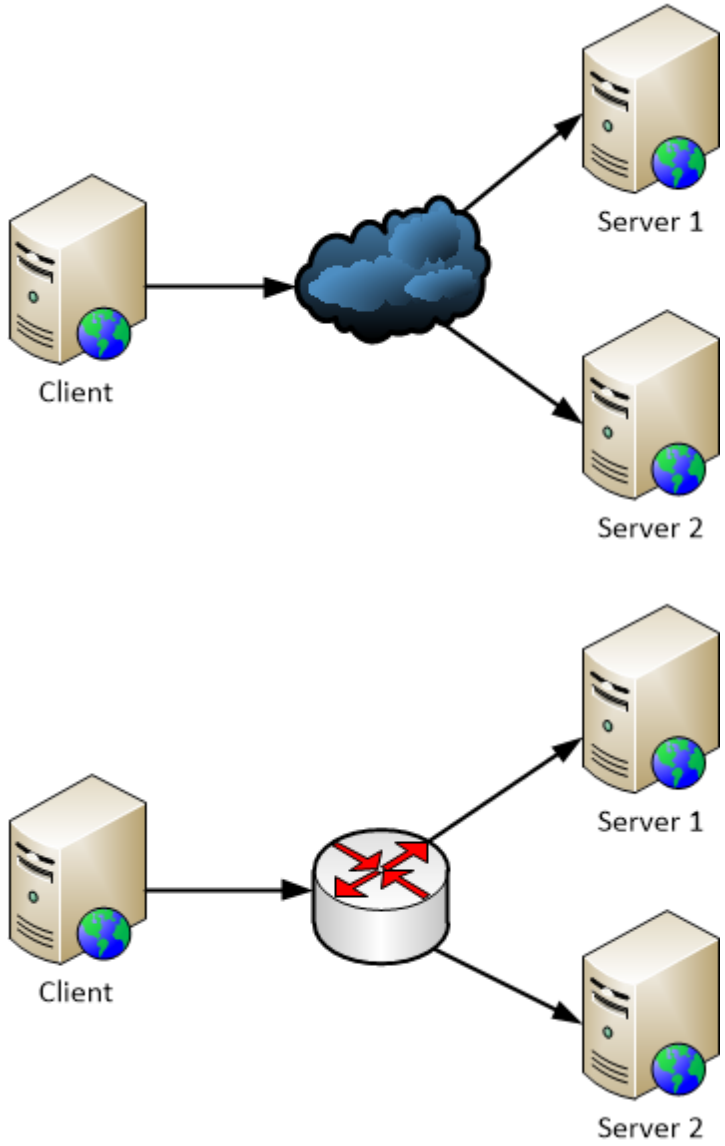
- Asynchronous with Confirmation
  - Server provides synchronous response only after quick validation of client's request
  - Final feedback is sent after completing long running process

- Use correlation ID in asynchronous communication
  - Client generates unique correlation ID and includes it in request message
  - Client stores request context and correlation ID in the database
  - Server provides response with given correlation ID
  - Client looks up request context by correlation ID in the database



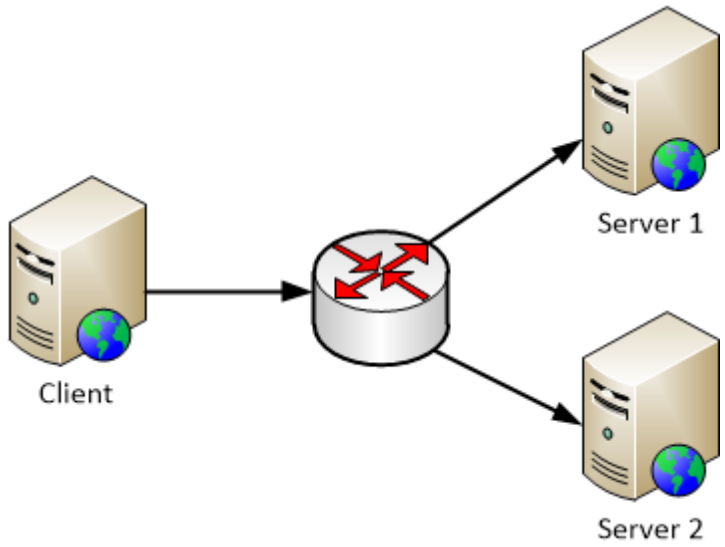
Traditional Load Balancers

- Hardware (or software) load balancers distribute traffic between configured number of backend servers
- If the backend server is unavailable, requests are forwarded to other servers from the pool
- Backend servers should expose “health-check” endpoints that will be invoked by load balancer every N seconds to confirm their availability
  - Verify availability of dependent components in health-check implementation



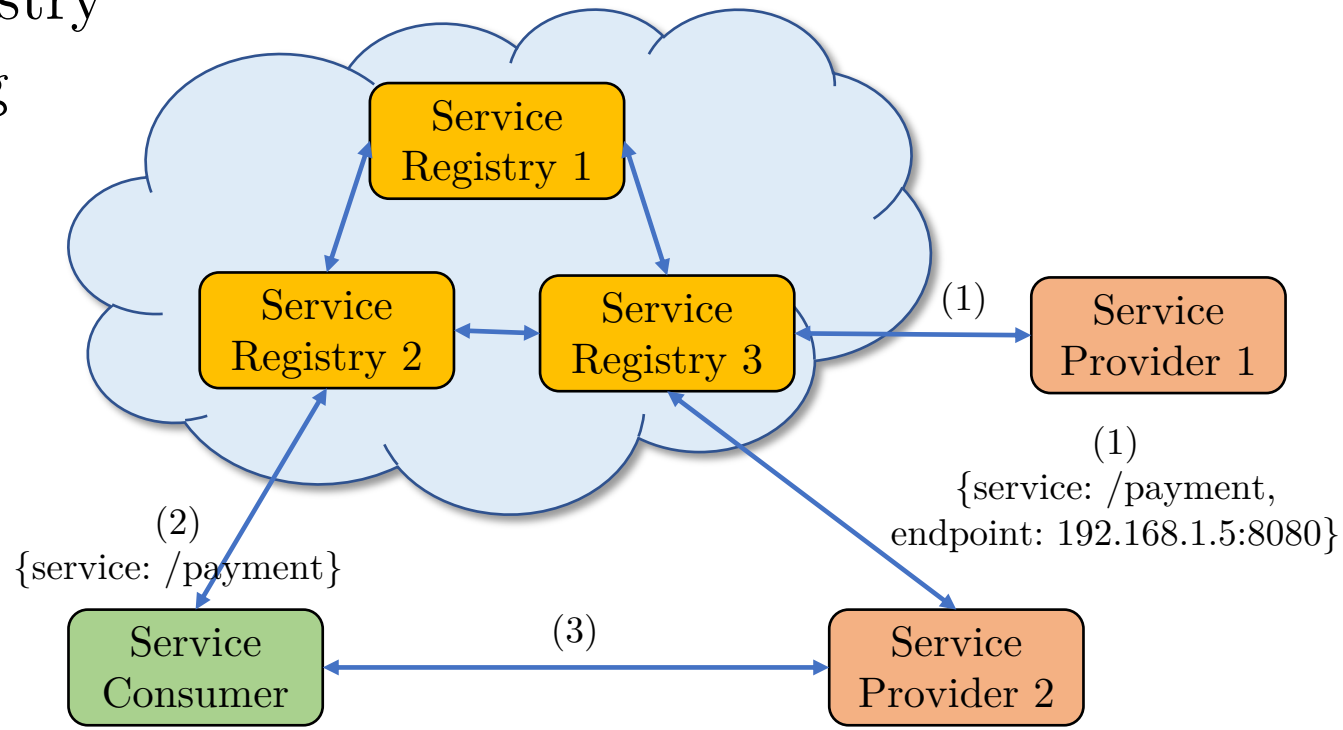
Apache Web Server or NGINX can be easily configured as software load balancer

```
http {
  upstream myApp1 {
    server srv1.example.com;
    server srv2.example.com;
    server srv3.example.com;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://myApp1;
      health_check interval=3 fails=3 passes=2 uri=/health;
    }
  }
}
```



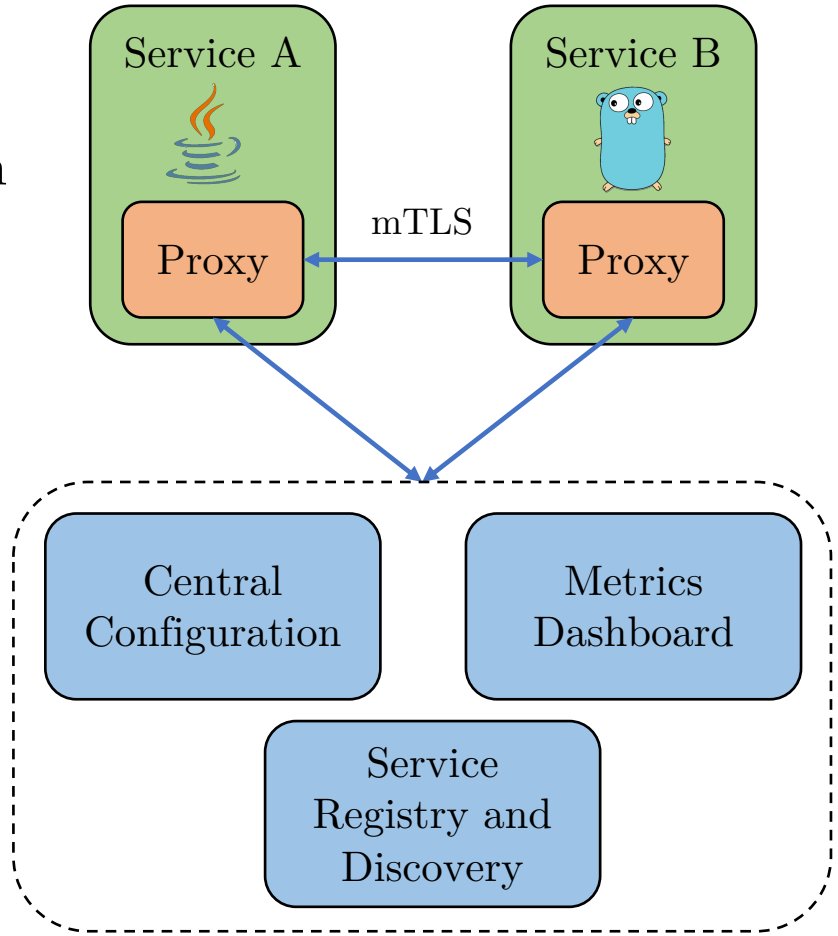
Single Point of Failure!

- Service producers register in distributed Service Registry (1)
  - Service Registry keeps track of all alive instances of given service
  - Each instance of service provider periodically heartbeats with registry
  - Easy dynamic scaling (up and down) of service producers
- Clients query any of Service Registry nodes to get all endpoints hosting given service (2)
- Clients contact service providers directly (3)
  - Client-side load balancing based on various metrics
  - No extra hops between client and a service



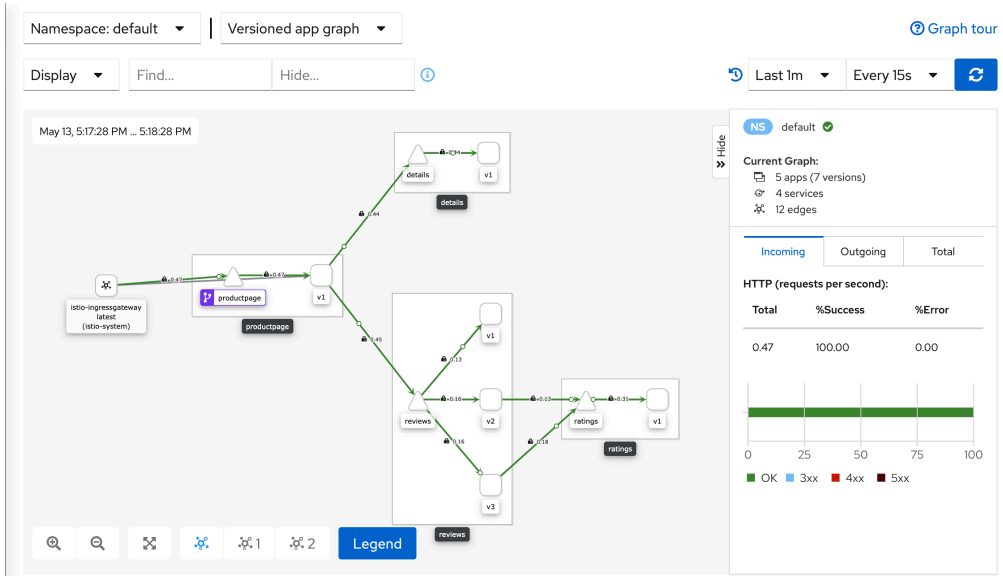
Service Mesh

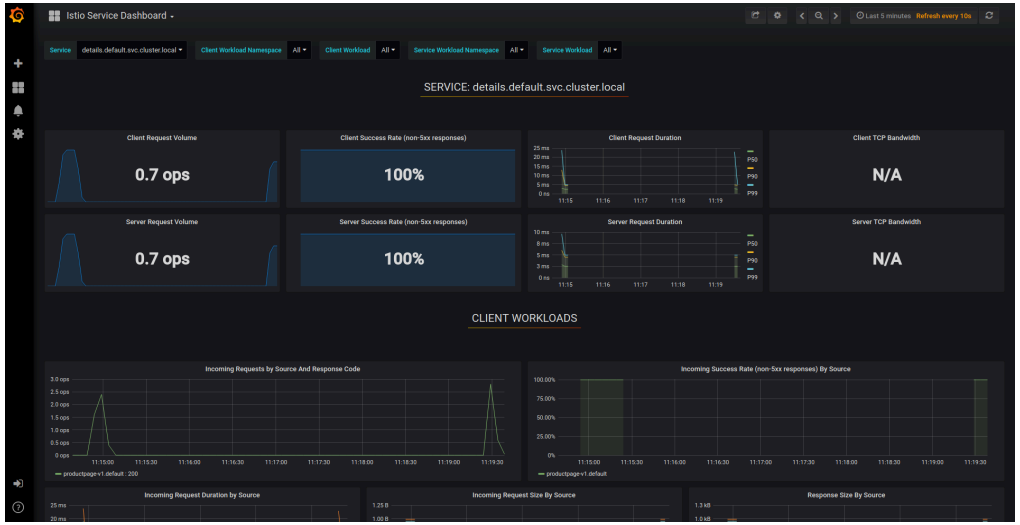
- Leverage Proxy Sidecar pattern to abstract inter-service communication
- Service Mesh features:
  - Service discovery and registry
  - Request load balancing and routing
  - Retry logic, rate limiting and Circuit Breaker pattern
  - Transport-level security (mTLS) and ACLs
  - Performance metrics
  - Distributed track and trace
  - Traffic splitting and Canary Deployments pattern
- Programming language independent inter-communication
- Example frameworks:
  - Linkerd 1.x (standalone) and 2.x (Kubernetes)
  - Istio



Service Mesh

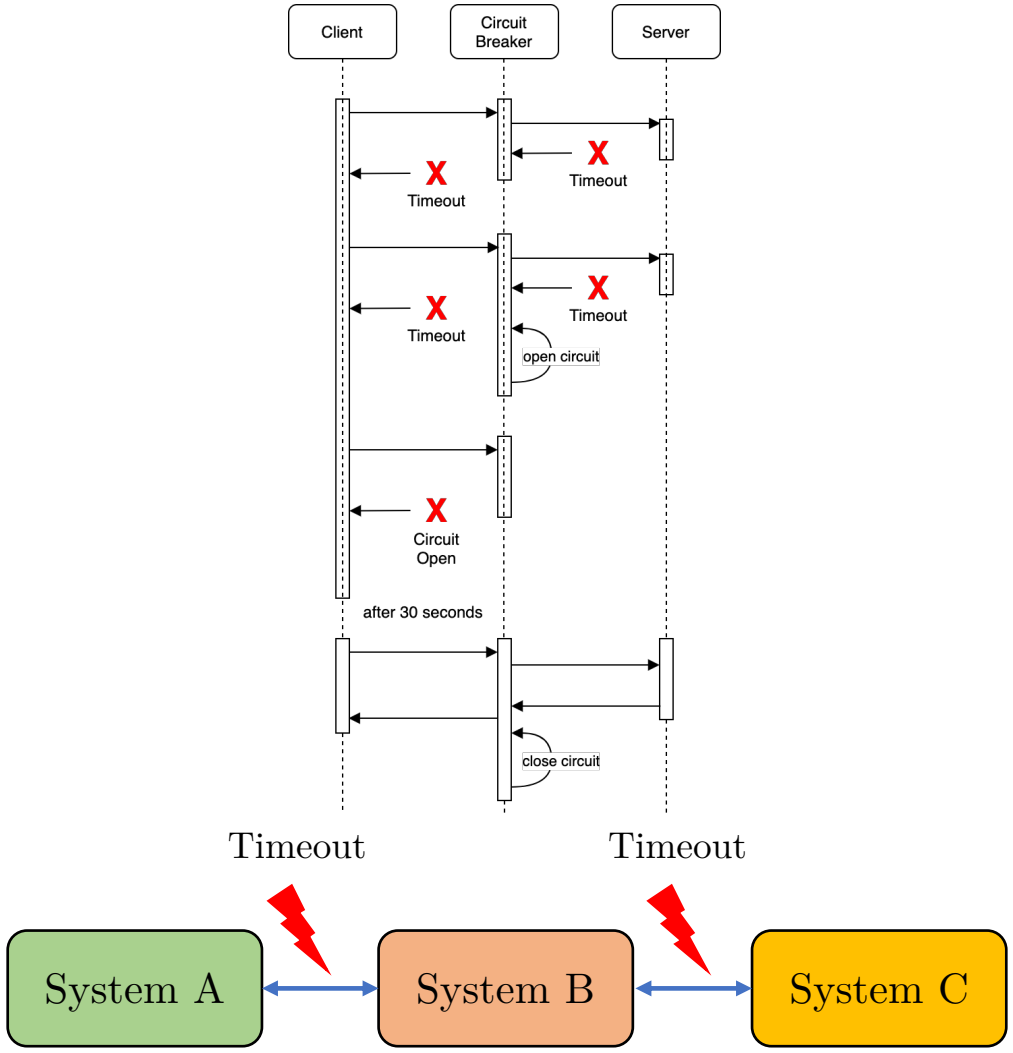
```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: jwt-per-host
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: ALLOW
  rules:
  - from:
    - source:
        requestPrincipals: ["*@example.com"]
    to:
    - operation:
        hosts: ["example.com", "*.example.com"]
```



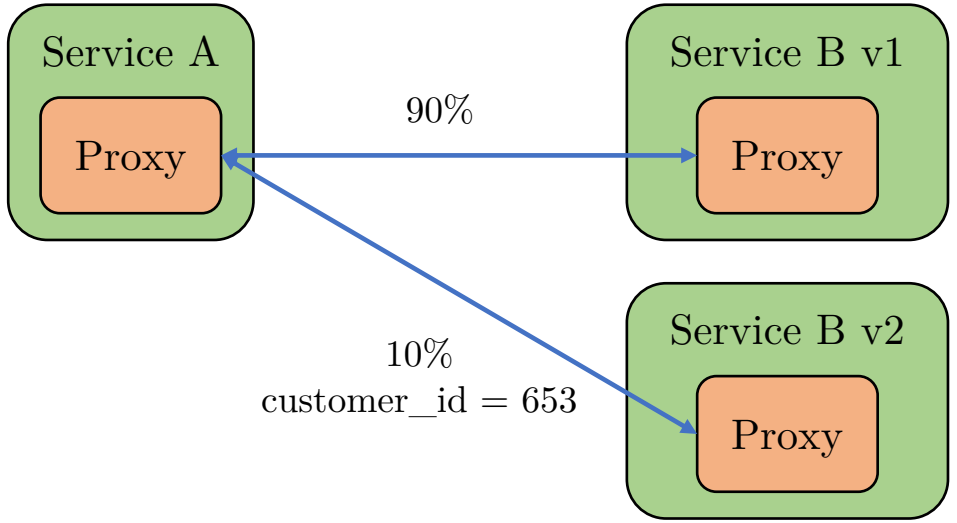


Source: [Istio Documentation]

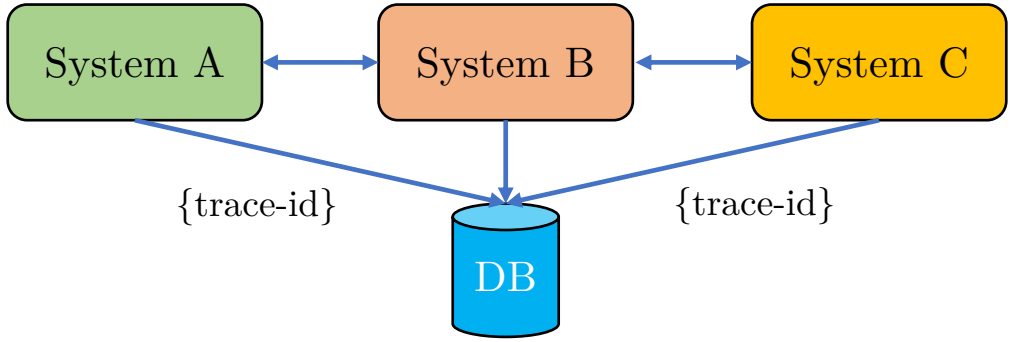
- Circuit Breaker



- Canary Deployment



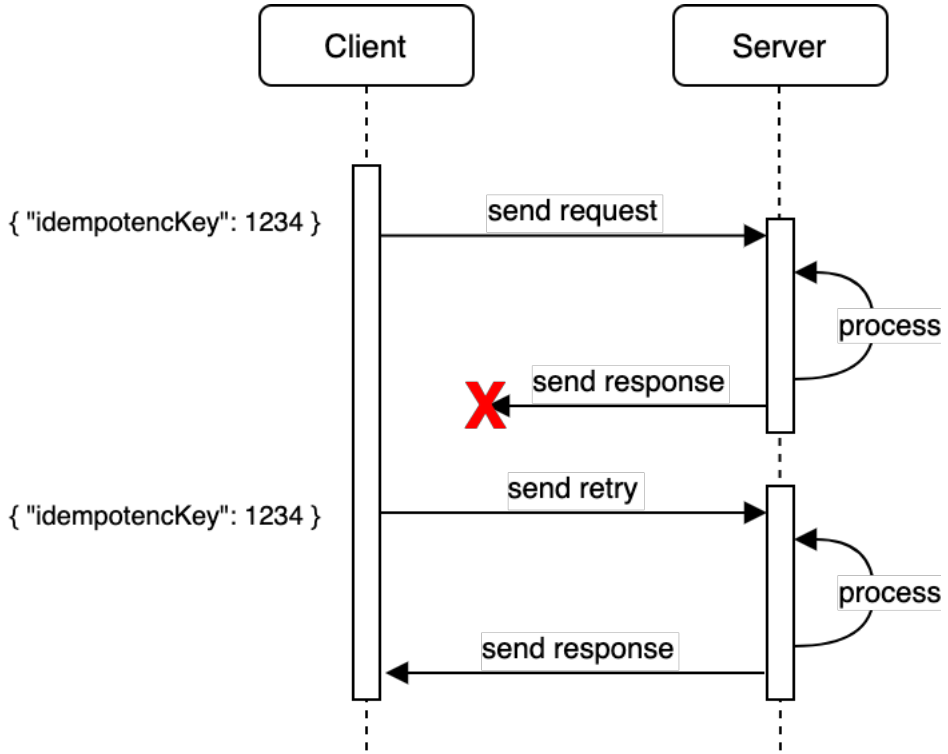
- Distributed Tracing



Definition

An idempotent service can be called many times without different outcomes, provided all requests share the same, unique idempotency key (generated by service consumer).

- Benefits:
  - Handling of duplicate requests
  - Clients have the possibility to retry timed out requests safely



- Verification scenarios:
  - Any operation within service implementation fails, customer retries
  - Complete service invocation succeeds, but consumer times out and retries
  - Two identical requests picked up at the same time by the system

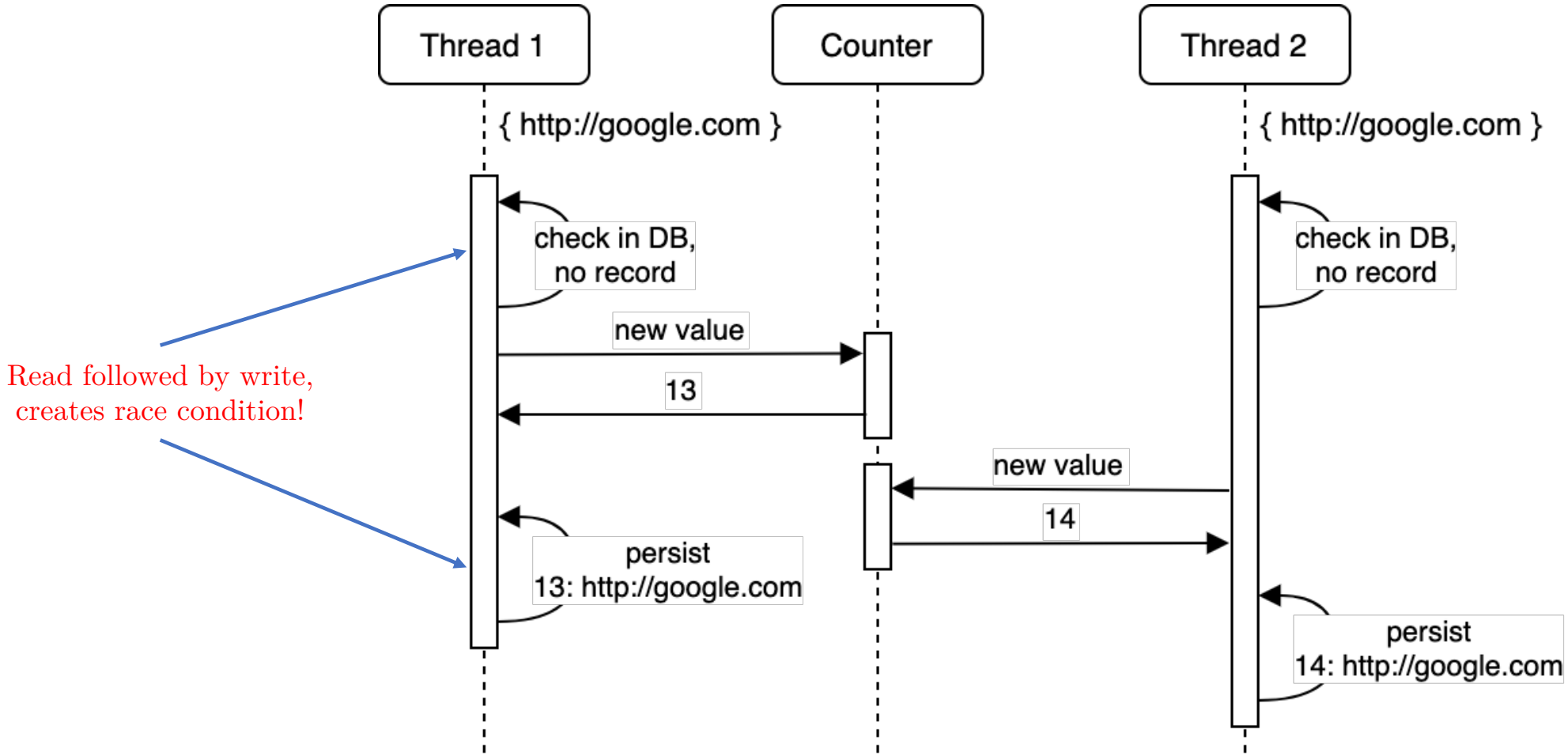
```
public String shortenUrl(String longUrl) {
    long id = generateNextId();
    String shortUrl = String.format(
        "http://tiny.com:8080/%s",
        Long.toString(id, 36));
    insertMapping(shortUrl, longUrl);
    return shortUrl;
}
```

Loose one ID number

Nothing happens, local operations only

Orphaned mapping inserted
Client will retry the request

- Two identical requests picked up at the same time by the system



Idempotent Service Design

- Invoking non-idempotent services
  - Keep track whether we tried to invoke given service with idempotency key
  - We cannot retry the service invocation for the second time
  - Raise alert for operations team if we have called the service before

```
if (! didInvokeBefore("serviceA", idempotencyKey)) {
    persistStatus("serviceA", TPC_INVOKE, idempotencyKey);
    response = invokeServiceA(idempotencyKey, request);
    updateStatus("serviceA", INVOKED, idempotencyKey);
}
public boolean didInvokeBefore(String serviceName, String key) {
    Status status = queryStatus(serviceName, key);
    if (TPC_INVOKE.equals(status)) throw new UnsupportedRetryException(serviceName, key);
    return INVOKED.equals(status);
}
```

Summary

- Client and server communicate over the network
- Message transport and format
- Request-replay and Fire-and-forget communication patterns
- Service Registry and Discovery
- Service Meshes
- Idempotent service design