Cleipnir

Resilient Functions

What is it?

A .NET framework

assisting with the **implementation**

of code

which needs to be executed in its

entirety

Resilient Functions Example - **Hello World!**

```
var store = new PostgreSqlFunctionStore (CONNECTION_STRING);

var functions = new RFunctions(store, new Settings(UnhandledExceptionHandler : Console.WriteLine));
```

Resilient Functions Example - **Hello World!**

```
var store = new PostgreSqlFunctionStore (CONNECTION_STRING);

var functions = new RFunctions(store, new Settings(UnhandledExceptionHandler : Console.WriteLine));

var rAction = functions. RegisterAction (

  functionTypeId : "HttpAndDatabaseSaga" ,

  inner: async (Guid id) =>

    var content = await (await HttpClient.PostAsync(URL, new StringContent (id.ToString()))).Content.ReadAsStringAsync();

    await connection.ExecuteAsync(

      "INSERT INTO Entities (Id, State) VALUES (@Id, @State) ON CONFLICT DO NOTHING"  ,

      new {State = content, Id = id}

    );

).Invoke;
```

Resilient Functions Example - **Hello World!**

```
var store = new PostgreSqlFunctionStore (CONNECTION_STRING);
var functions = new RFunctions(store, new Settings(UnhandledExceptionHandler : Console.WriteLine));
var rAction = functions. RegisterAction (
  functionTypeId : "HttpAndDatabaseSaga" ,
  inner: async (Guid id) =>
    var content = await (await HttpClient.PostAsync(URL, new StringContent (id.ToString()))).Content.ReadAsStringAsync();
    await connection.ExecuteAsync(
      "INSERT INTO Entities (Id, State) VALUES (@Id, @State) ON CONFLICT DO NOTHING"  ,
      new {State = content, Id = id}
    );
).Invoke;
await rAction(functionInstanceId : Id.ToString(), param: Id);
```

What you get (short version)!

Simply a way

to **ensure** your code

gets run

until you say

it is **done**

What you get (long version)!

1. ensuring a method invocation is **completed**
2. **synchronized invocation** across multiple process instances
3. no cluster management & transparent **scalability**
4. **cloud independance** & support for **multiple databases**
5. good **debug** experience
6. ability to **migrate** non-completed functions
7. **manual error handling** is first class citizen
8. simple **testability**

What can you **use it for** (use cases)?

- Any business process which must be executed in its entirety
  in order to avoid inconsistencies

- Do you have any methods/classes - in your code base - which communicates with multiple external systems?
  What happens if the flow is only partly completed?

- The situation is more common than we might expect in our microservice system’s landscape.

What can you **use it for** - Examples

Examples:

- Order Processing (we have already seen)

- Change Customer Payment Method Subscription
    a. Stop current payment method
    b. Start new payment method
    c. Persist the fact that the customer uses a new payment method

- Bank Transfer between two account

Repeated Example - **Order Processing**

```
var productPrices = await _productsClient.GetProductPrices(order.ProductIds).Sum(p => p.Price);

await _bankClient.Reserve(totalPrice);
await _logisticsClient.ShipProducts(order.CustomerId, order.ProductIds);
await _bankClient.Capture();

await _emailClient.SendOrderConfirmation(order.CustomerId, order.ProductIds);
await _ordersRepository.Insert(order);
```

Repeated Example - **Order Processing**

Making it resilient

Source code time!

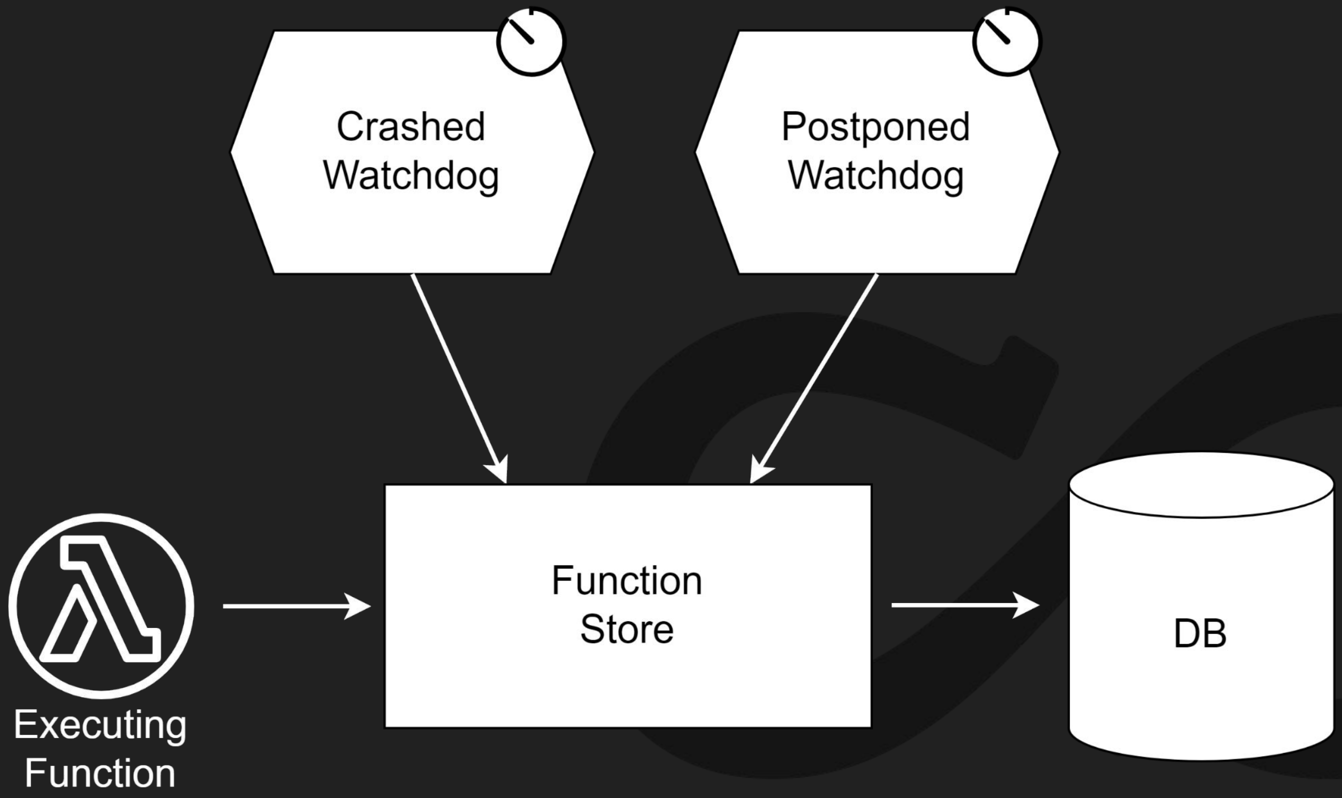
Technical **Overview**

Framework **Design Principles**

- Create a **simple** (low complexity) **abstraction** facilitating the implementation of **sagas**

- **Optimize** the developer’s **degrees of freedom**
(aka you are free to write the code how you like it)

Framework **Technical Architecture**



Distributed Systems

Theory (101)

Distributed Systems **Theory** (101)

- **Reality**, Microservice Architecture & The **Two Microservices Problem**
- **At-least-once** vs **At-most-once** message delivery and **Idempotency**
- Enterprise **Communication Patterns**

Distributed Systems Theory - **Reality**

What is the reality in which our programs are executed?

Distributed Systems Theory - **Reality**

What is the reality in which our programs are executed?

- A program may **crash at any point** during its execution

Distributed Systems Theory - **Reality**

What is the reality in which our programs are executed?

- A program may **crash at any point** during its execution
- A network package might be **lost**

The **Two Microservices Problem**

- Given two microservices (m_1 & m_2) can we implement a method sending a message to both services while guaranteeing that either:
  - Both receives the message
  - Neither receives the message

```
public static void SendToBoth(Message msg, Service s1, Service s2)

    s1.Send(msg);

    s2.Send(msg);
```

The **Two Microservices Problem**

- Given two microservices (m_1 & m_2) can we implement a method sending a message to both services while guaranteeing that either:
  - Both receives the message
  - Neither receives the message

```
~~public static void SendToBoth(Message msg, Service s1, Service s2)~~

~~    s1.Send(msg);~~

~~    s2.Send(msg);~~
```

It’s impossible!

Message Delivery Guarantees

- **At-most-once**
  A message will be delivered at most once. That is - it might be lost.

- **At-least-once**
  A message will be delivered at least once. That is - it might be delivered multiple times.

- **Idempotency**
  An API endpoint property stating whether a message can be delivered multiple times without unintended side effects.

Repeated Example - **Order Processing**

Using an **idempotent** Bank API

Source code time!

Repeated Example - **Order Processing**

Using an **at-most-once** Logistics API

Source code time!

Handling **Failures**

Failure Handling - Human Intervention

Writing code addressing **all failure scenarios**
in a distributed setting
is often **infeasible**

As such, Resilient Functions is built around the tenet of using **human intervention**.

Failure Handling - Failing an invocation

A function invocation **fails** when:

- it throws an **unhandled exception**
- it returns a **Fail-instance**

A failed function invocation is *not* retried automatically by the framework.

In order to **re-invoke** the function the function's registration's ReInvoke-method must be invoked.

Failure Handling - Re-invoking a function

```
var registration = functions.RegisterAction(
    functionTypeId: "HttpAndDatabaseSaga",
    inner: (Guid id) => { … }
);

await registration.ReInvoke(
    id.ToString(),
    expectedStatuses: new[] { Status.Failed }
);
```

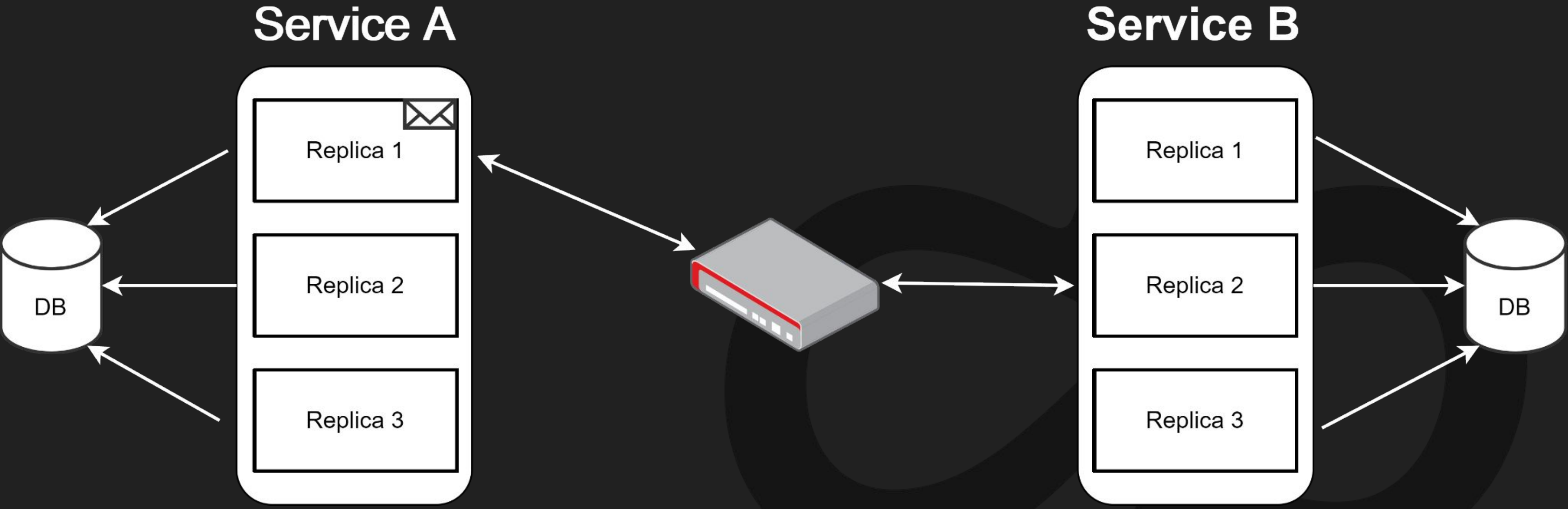
Failure Handling - Postponing an invocation

```
public static Result ProcessOrder(Domain.Order order)

{

    if (something)

        return Postpone.For(TimeSpan.FromHours(1));

    ...

}
```

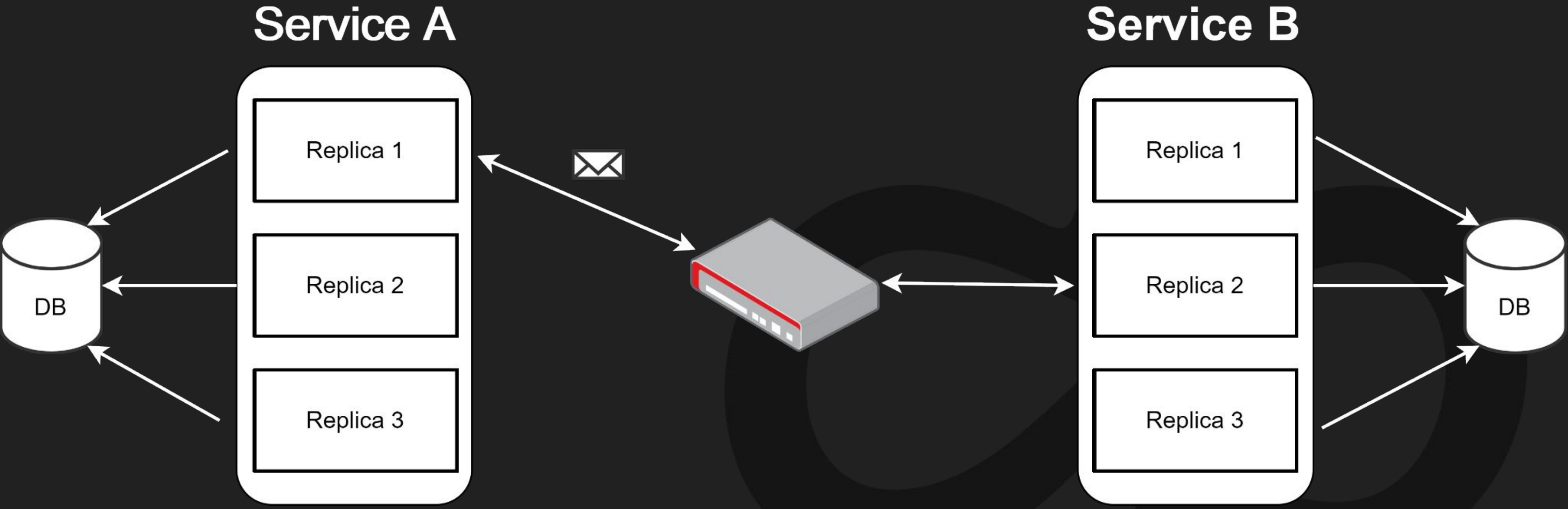
Communication

Patterns

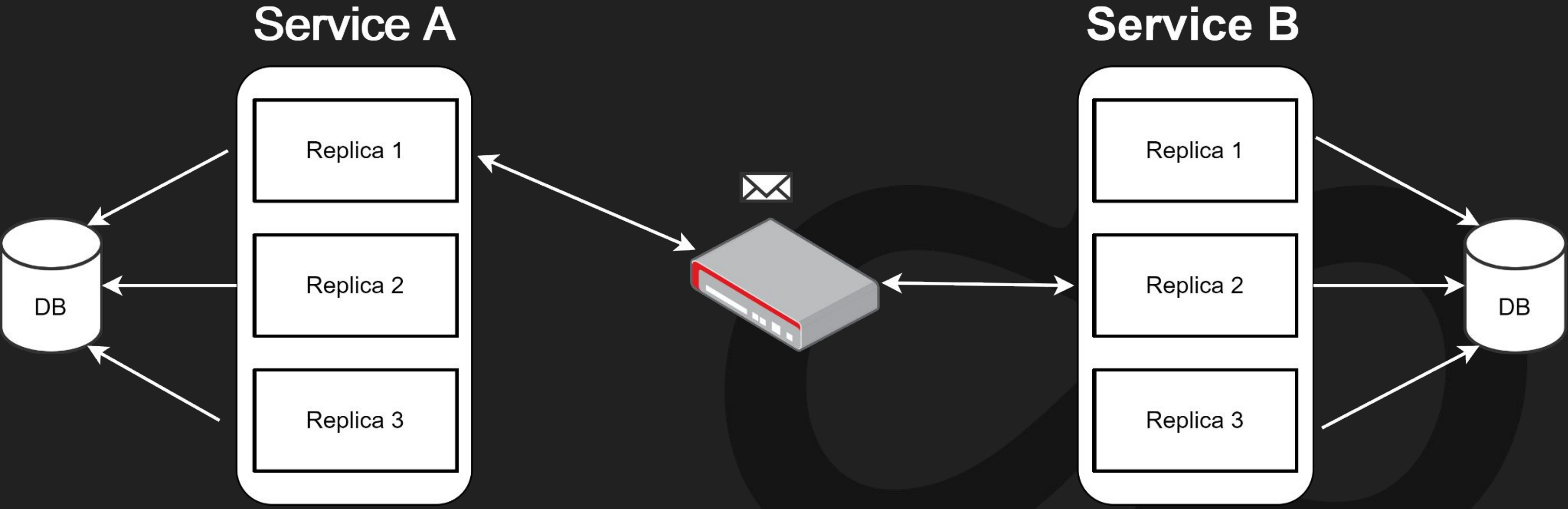
Communication Pattern - **Point-to-Point**



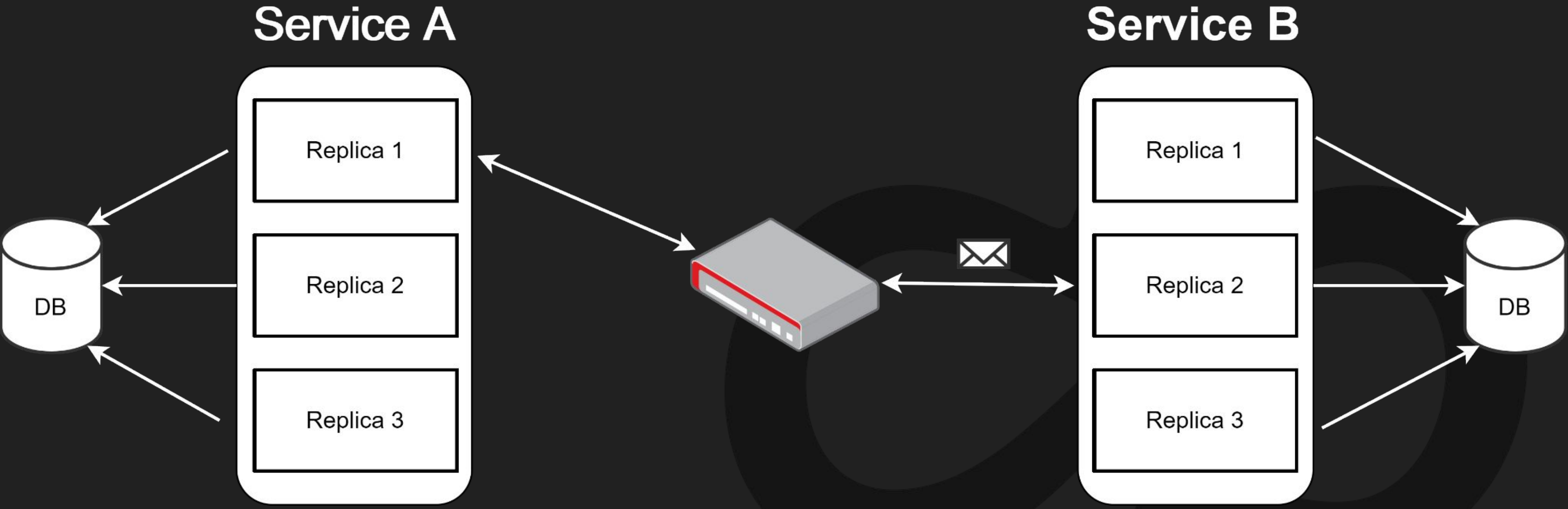
Communication Pattern - **Point-to-Point**



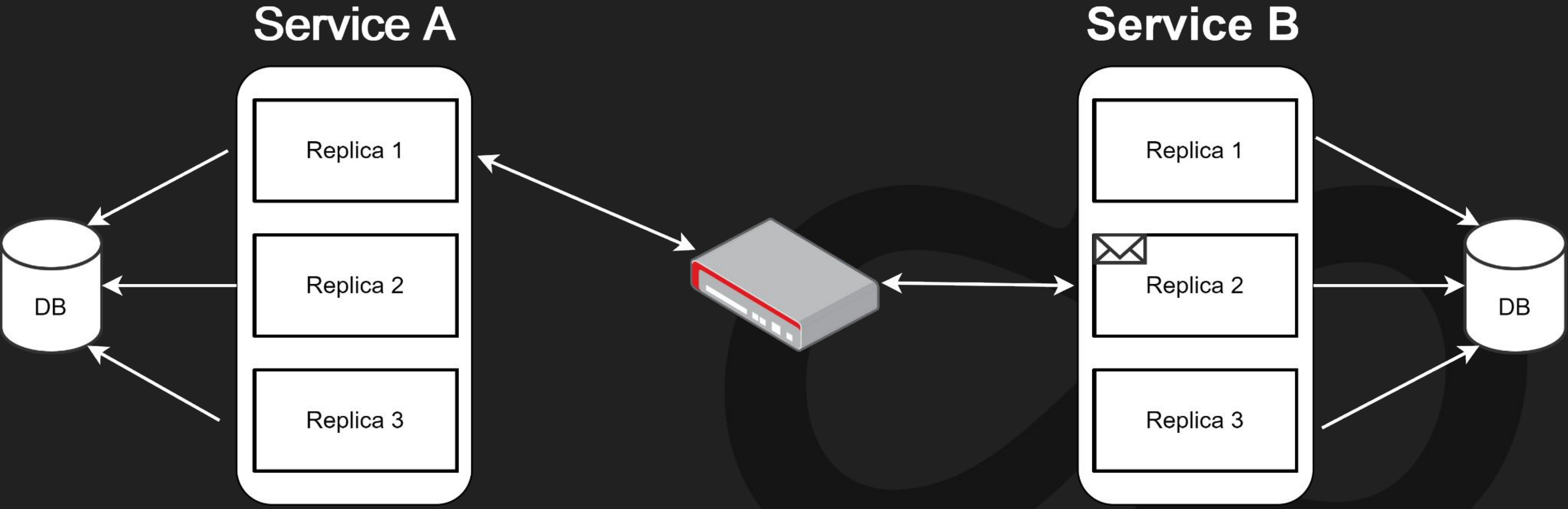
Communication Pattern - **Point-to-Point**



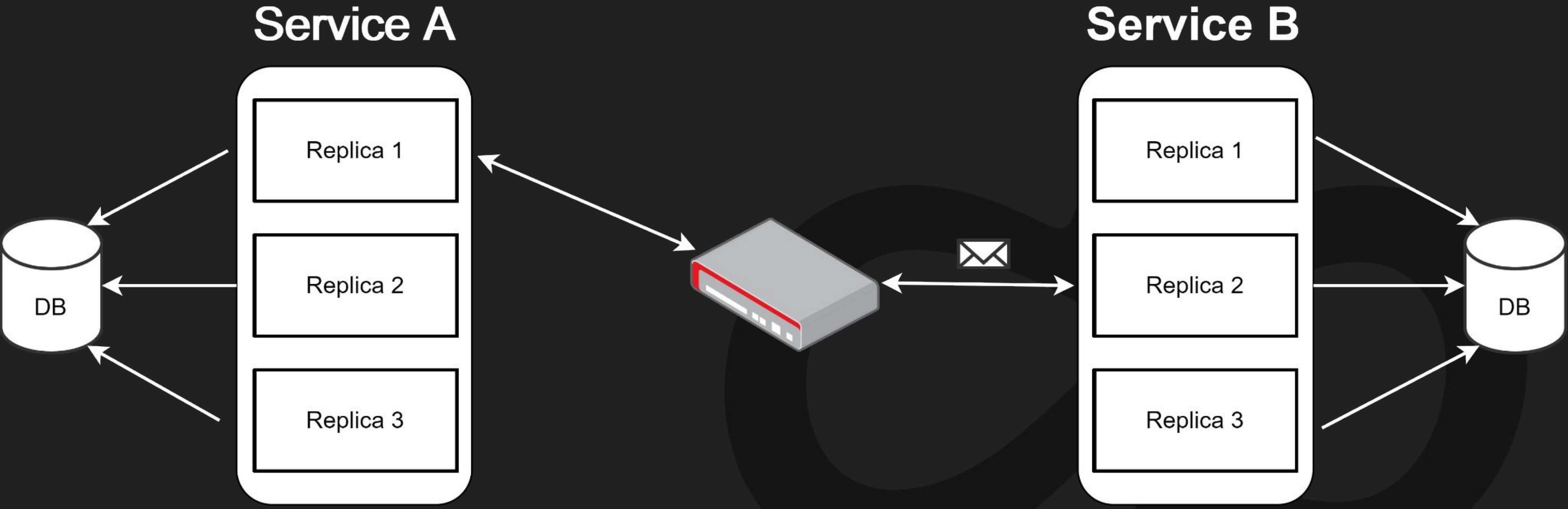
Communication Pattern - **Point-to-Point**



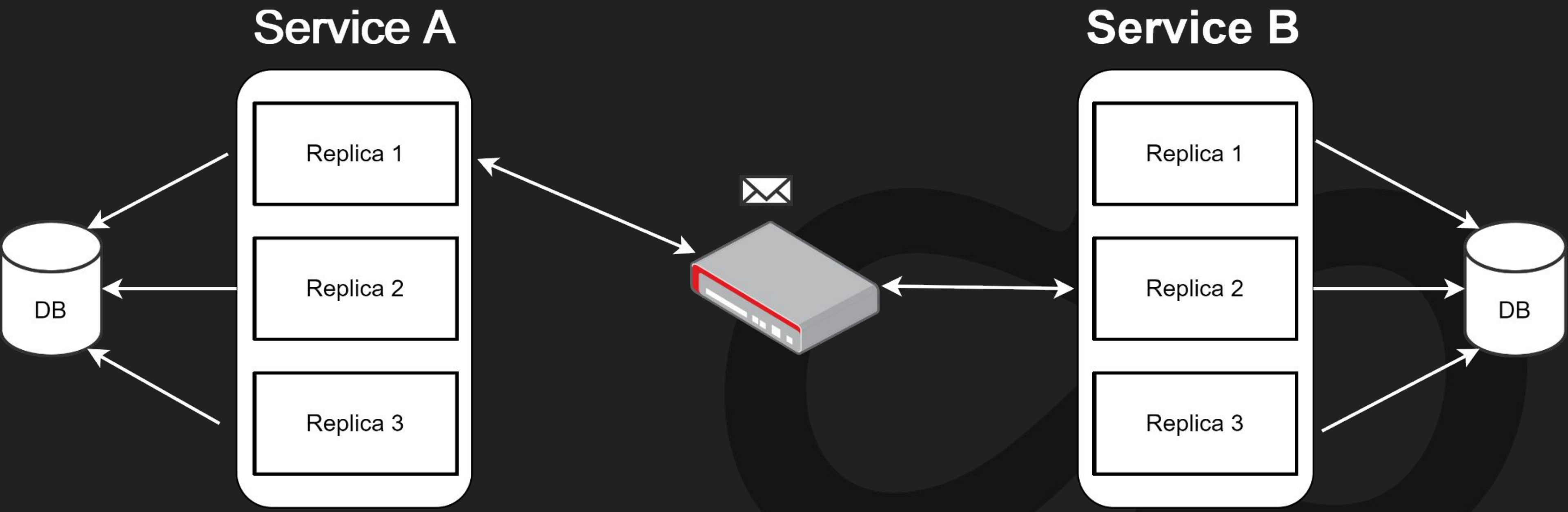
Communication Pattern - **Point-to-Point**



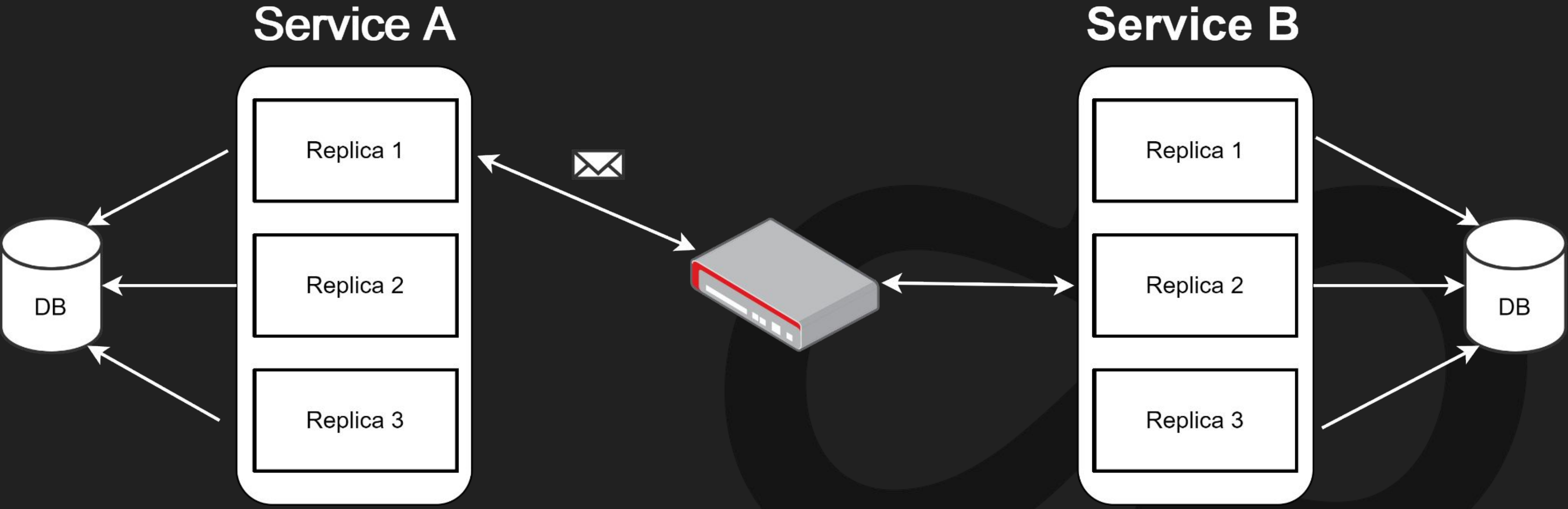
Communication Pattern - **Point-to-Point**



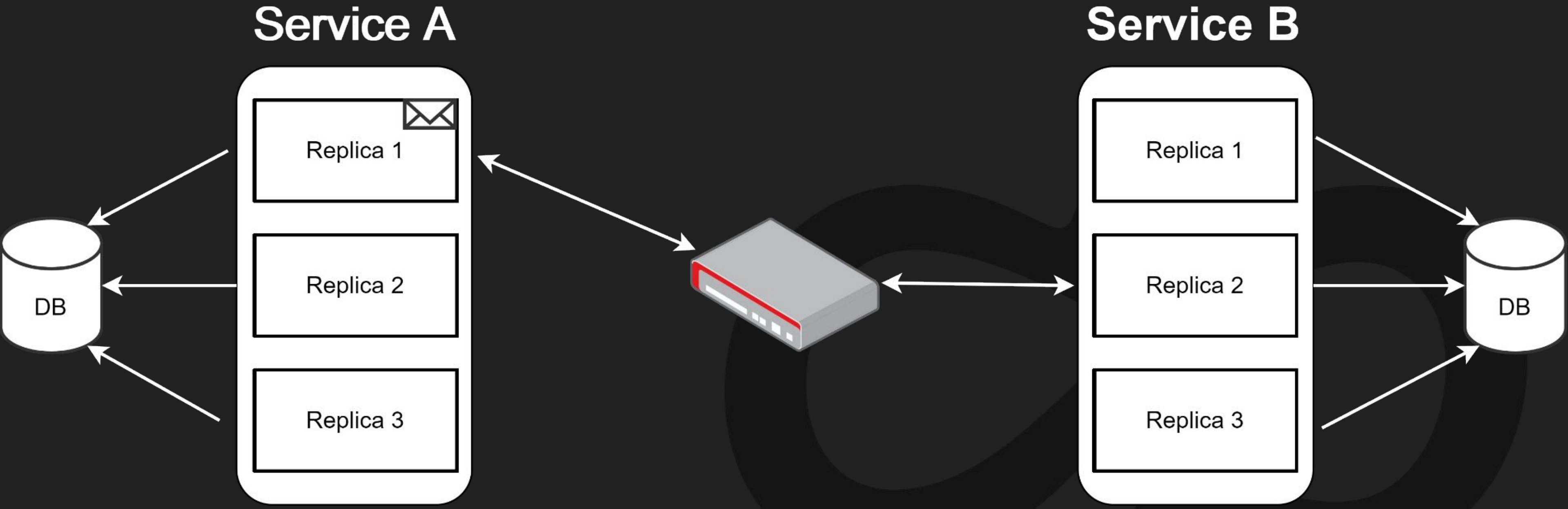
Communication Pattern - **Point-to-Point**



Communication Pattern - **Point-to-Point**



Communication Pattern - **Point-to-Point**



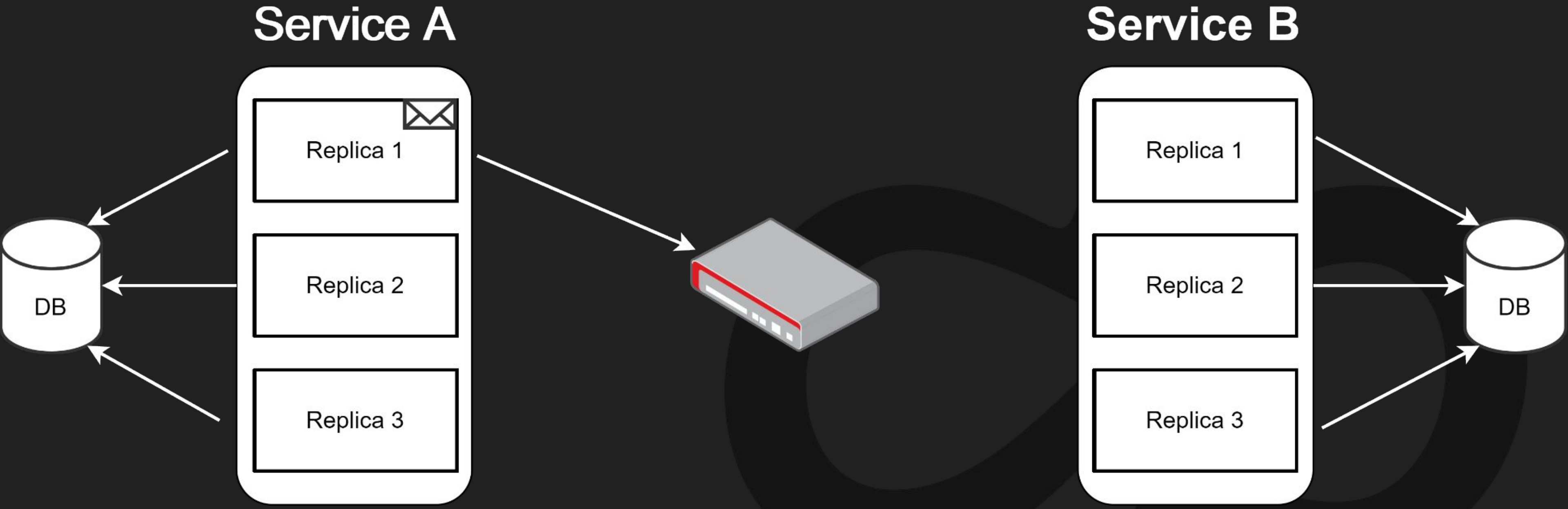
Communication Patterns - **Point-to-Point**

How does this affect

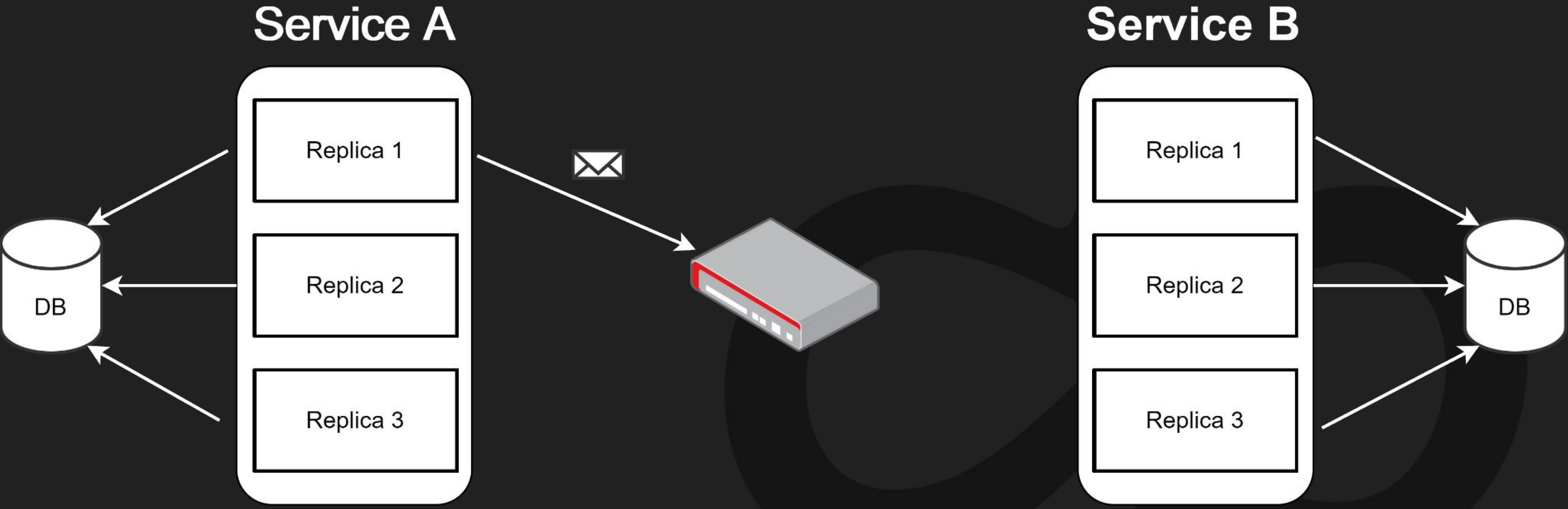
the way we can implement

our resilient logic?

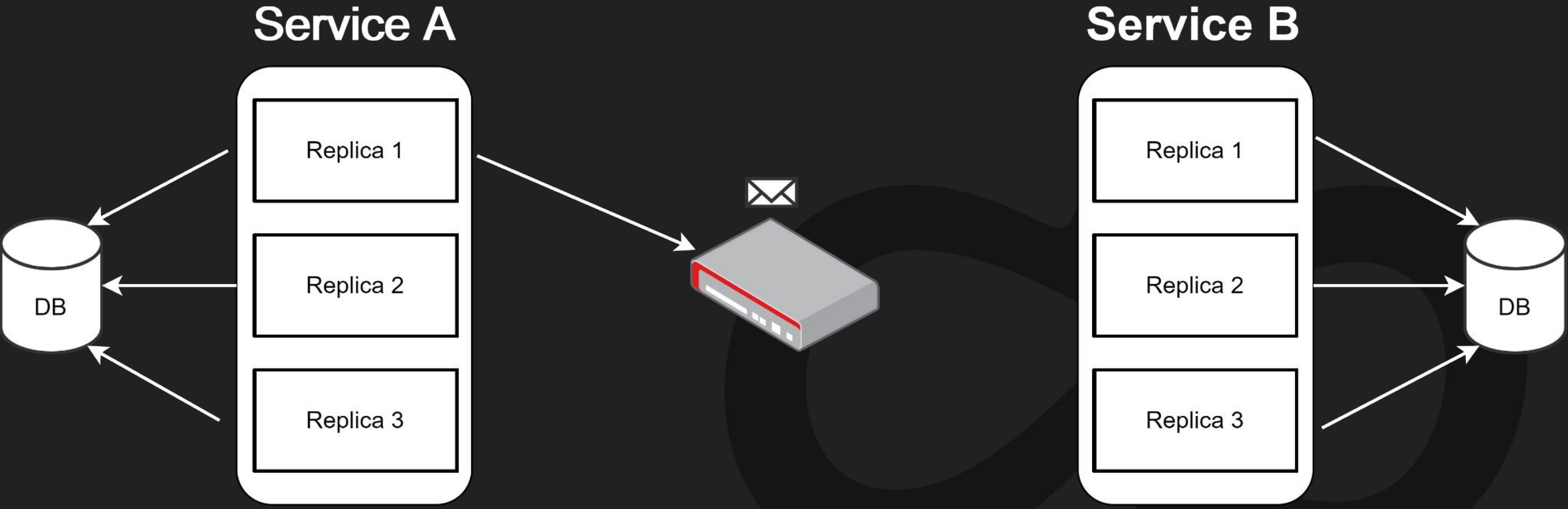
Communication Pattern - **Brokered**



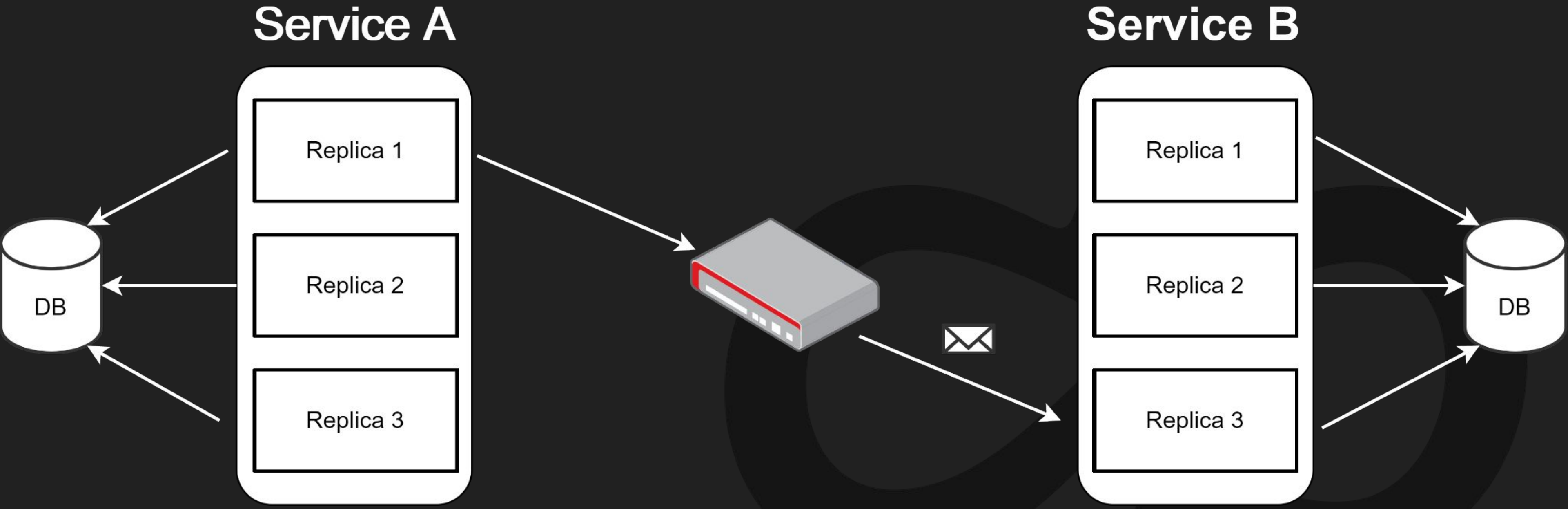
Communication Pattern - **Brokered**



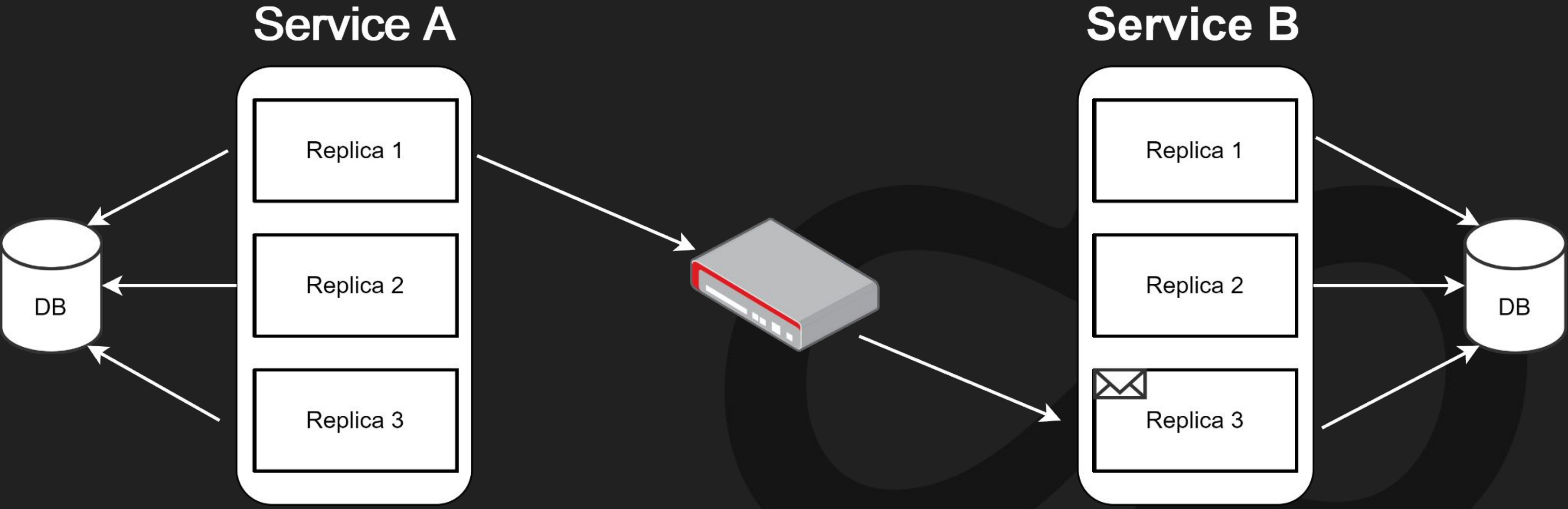
Communication Pattern - **Brokered**



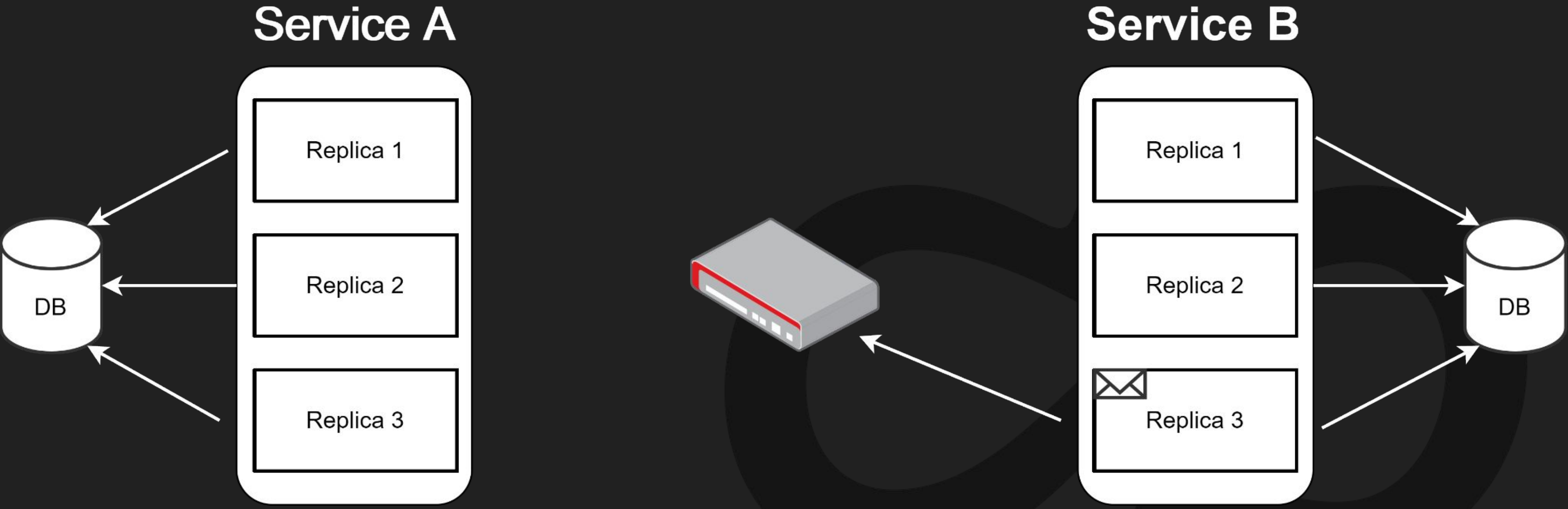
Communication Pattern - **Brokered**



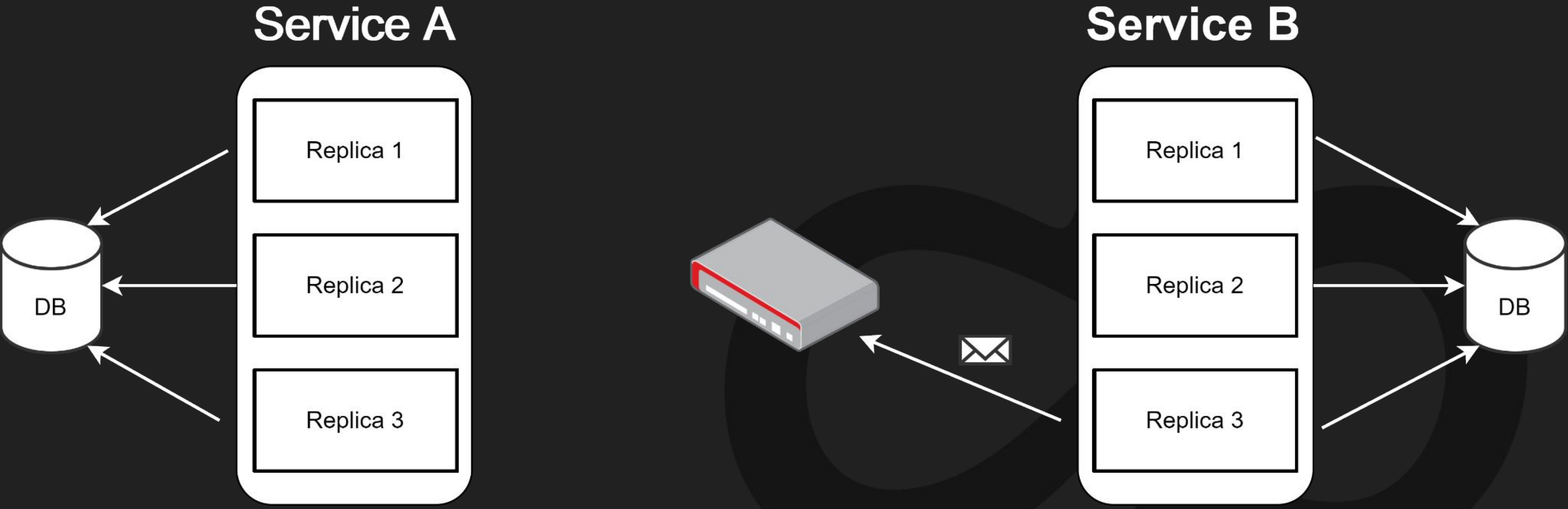
Communication Pattern - **Brokered**



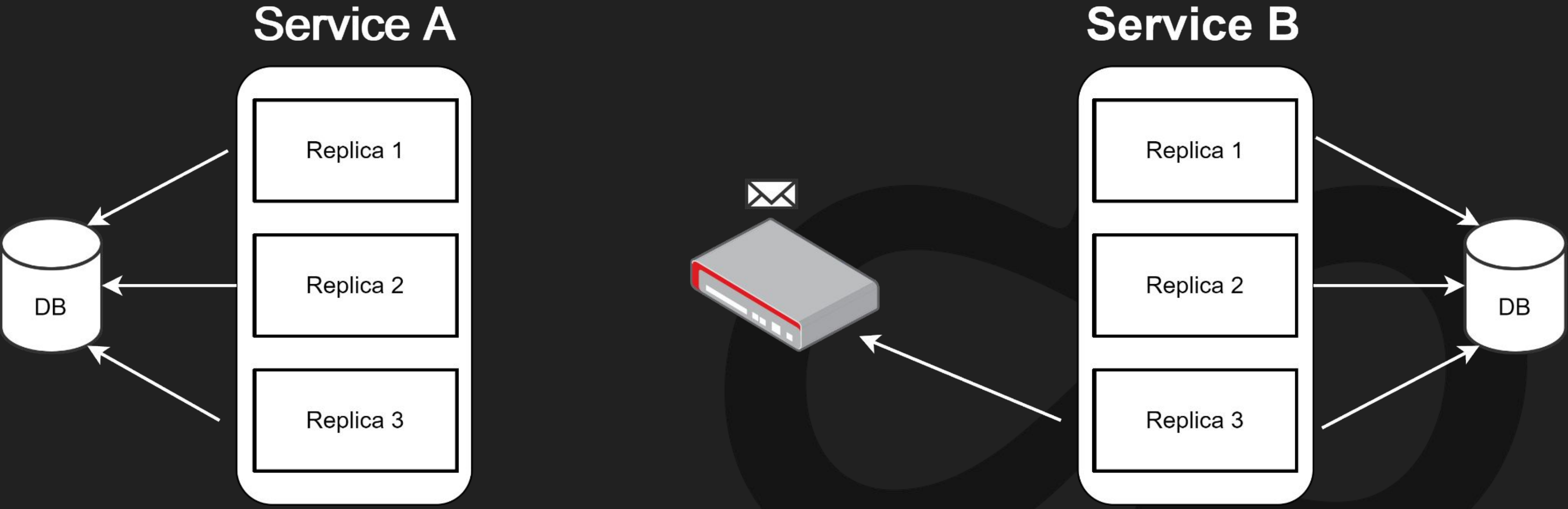
Communication Pattern - **Brokered**



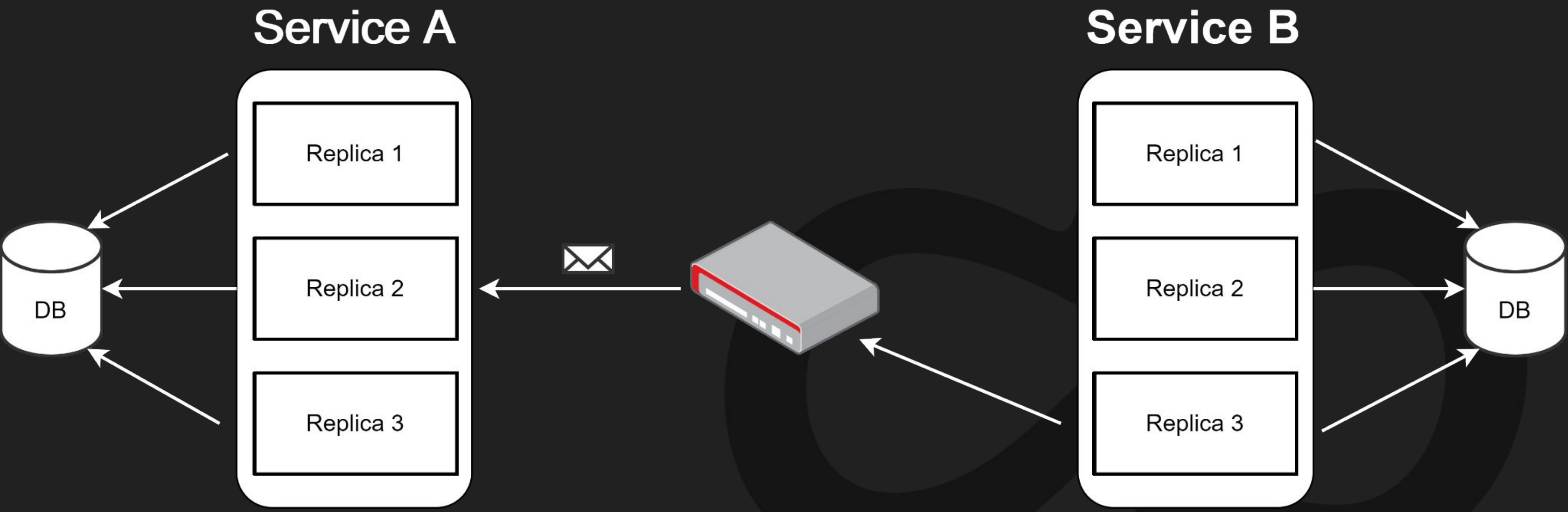
Communication Pattern - **Brokered**



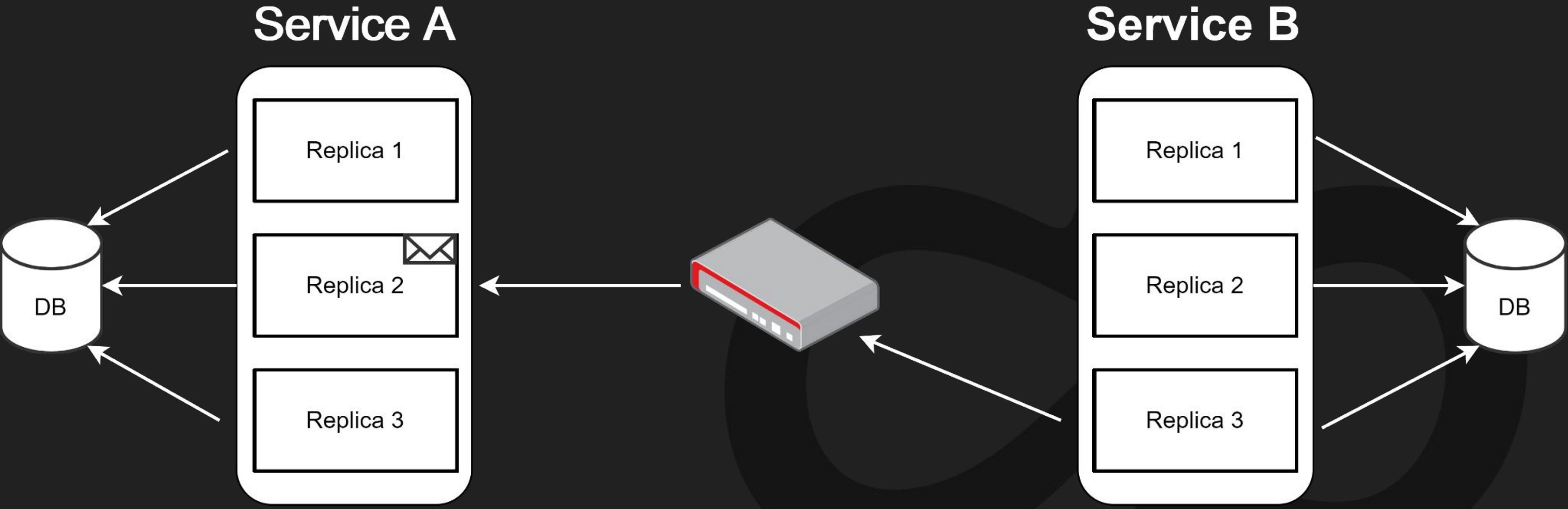
Communication Pattern - **Brokered**



Communication Pattern - **Brokered**



Communication Pattern - **Brokered**



Communication Patterns - **Brokered**

How does this **affect**

the way we can implement

our resilient logic?

Communication Patterns

- **Point-to-Point**
  Just use vanilla Resilient Functions

- **Brokered**
  Use the Messaging extension

Repeated Example - **Order Processing**

Using a **message driven** e-mail API

Source code time!

The Message Queue **Panacea**

Message Queues and **Sagas**

- Message Queues are often used for implementing Sagas
- At first glance this seems like a good idea
- However, after implementing the first state machine representation of a saga by hand the assumption quickly **falls apart**

Message Queues and **Sagas** - Problems

Difficult, because one still have to handle:

- **Re-deliveries** & **Out-of-order** messages
- **Poison**-messages and **Dead-Letter-Queues**
- **Synchronization** - Ensuring only one saga is executioning at a time
- **Failures** - How to Retry or Postpone an invocation

Message Queues and **Sagas** - Problems

As a saga is *not* a first-order concept; how do you:

- Check if the saga has failed
- Manually retry it
- Migrate existing running sagas

Final Thoughts

Conceptual **benefit** of using Resilient Functions

- A persistent function invocation becomes a **first-order** “**thingamajig”**
  You can reference it, check its status and start it again

- Similar to when delegates (Func/Action) where introduced and functions became a first-order thingy in .NET

Food for thought…

1. Regarding **Versioning** of functions and **migration** of executing functions. What if we create a new resilient function type but at most one of the two versions are allowed to execute?

2. Can you create a **recurring job** from the constructs you have seen so far?

3. Any issues to be aware of when changing “**crashed check frequency**”
   What if it increased?
   What if it decreased?

Passing the **Baton** on

- How is the **relay** related to **distributed systems**?
- How do we ensure *not* losing the **baton**?

  When using **Resilient Functions**?

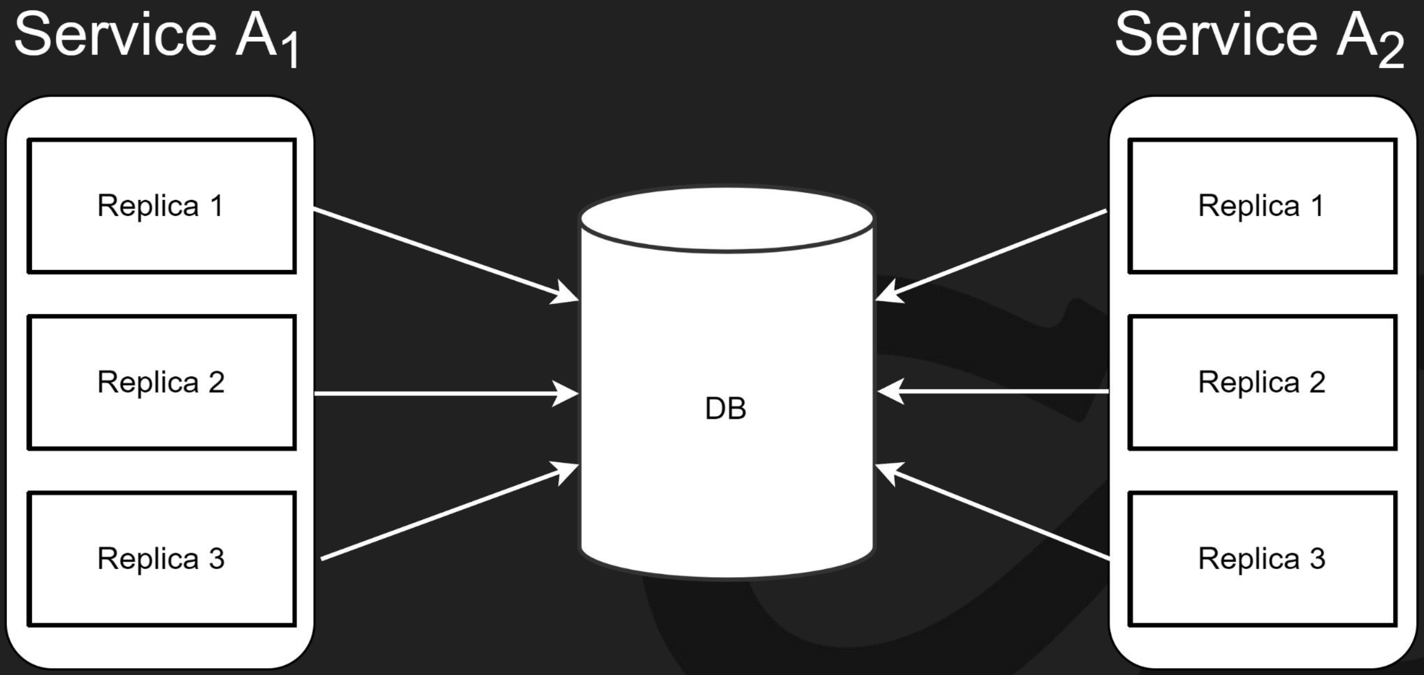
Passing the **Baton** on

- How is the **relay** related to **distributed systems**?

- How do we ensure *not* losing the **baton**?

  When using **Message Queues**?

Food for thought - **Deployment & Versioning**


Service A_1
Replica 1
Replica 2
Replica 3
DB
Service A_2
Replica 1
Replica 2
Replica 3