Microservices architecture

© Ian Sommerville 2018

Software services

- A software service is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects.
  - The service is accessed through its published interface and all details of the service implementation are hidden.
  - Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.
- When a service request is made, the state information may be included as part of the request and the updated state information is returned as part of the service result.
- As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers.

© Ian Sommerville 2018:

Modern web services

- After various experiments in the 1990s with service-oriented computing, the idea of 'big' Web Services emerged in the early 2000s.
- These were based on XML-based protocols and standards such as SOAP for service interaction and WSDL for interface description.
- Most software services don't need the generality that's inherent in the design of web service protocols.
- Consequently, modern service-oriented systems, use simpler, 'lighter weight' service-interaction protocols that have lower overheads and, consequently, faster execution.

© Ian Sommerville 2018:

Microservices

- Microservices are small-scale, stateless, services that have a single responsibility. They are combined to create applications.
- They are completely independent with their own database and UI management code.
- Software products that use microservices have a *microservices architecture*.
- If you need to create cloud-based software products that are adaptable, scaleable and resilient then I recommend that design them around a microservices architecture.

© Ian Sommerville 2018:

A microservice example

- System authentication
  - User registration, where users provide information about their identity, security information, mobile (cell) phone number and email address.
  - Authentication using UID/password.
  - Two-factor authentication using code sent to mobile phone.
  - User information management e.g. change password or mobile phone number.
  - Reset forgotten password.
- Each of these features could be implemented as a separate service that uses a central shared database to hold authentication information.
- However, these features are too large to be microservices. To identify the microservices that might be used in the authentication system, you need to break down the coarse-grain features into more detailed functions.

© Ian Sommerville 2018:

Figure 6.1 Functional breakdown of authentication features

User registration

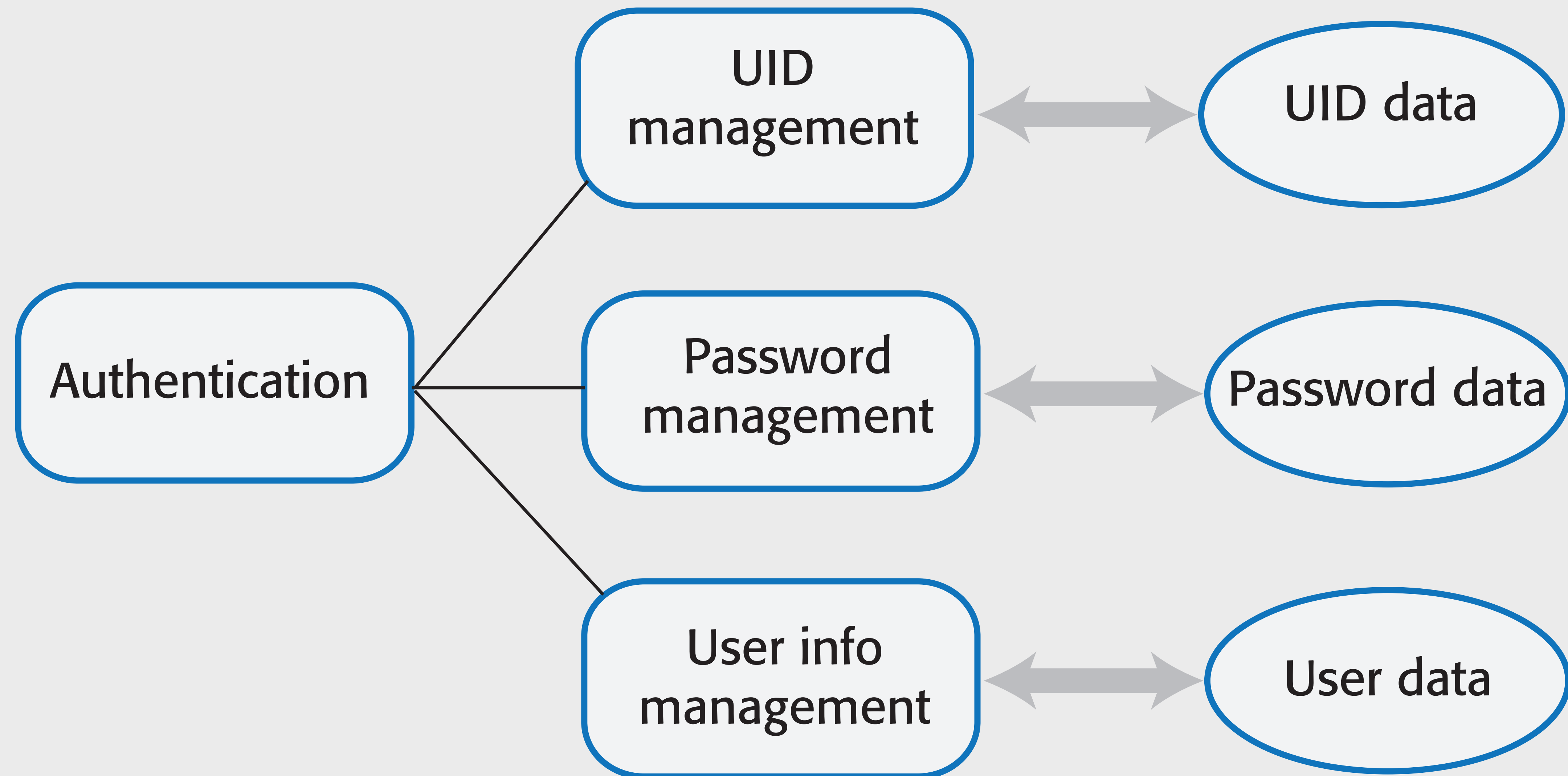
Setup new login id
Setup new password
Setup password recovery information
Setup two-factor authentication
Confirm registration

Authenticate using UID/password

Get login id
Get password
Check credentials
Confirm authentication

© Ian Sommerville 2018:

Figure 6.2 Authentication microservices



© Ian Sommerville 2018:

Table 6.1 Characteristics of microservices

Self-contained
Microservices do not have external dependencies. They manage their own data and implement their own user interface.

Lightweight
Microservices communicate using lightweight protocols, so that service communication overheads are low.

Implementation-independent
Microservices may be implemented using different programming languages and may use different technologies (e.g. different types of database) in their implementation.

Independently deployable
Each microservice runs in its own process and is independently deployable, using automated systems.

Business-oriented
Microservices should implement business capabilities and needs, rather than simply provide a technical service.

© Ian Sommerville 2018:

Microservice communication

- Microservices communicate by exchanging messages.
- A message that is sent between services includes some administrative information, a service request and the data required to deliver the requested service.
- Services return a response to service request messages.
    - An authentication service may send a message to a login service that includes the name input by the user.
    - The response may be a token associated with a valid user name or might be an error saying that there is no registered user.

© Ian Sommerville 2018:

Microservice characteristics

- A well-designed microservice should have high cohesion and low coupling.
    - Cohesion is a measure of the number of relationships that parts of a component have with each other. High cohesion means that all of the parts that are needed to deliver the component's functionality are included in the component.
    - Coupling is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.
- Each microservice should have a single responsibility i.e. it should do one thing only and it should do it well.
    - However, 'one thing only' is difficult to define in a way that's applicable to all services.
    - Responsibility does not always mean a single, functional activity.

© Ian Sommerville 2018:

Figure 6.3 Password management functionality

User functions
Create password
Change password
Check password
Recover password

Supporting functions
Check password validity
Delete password
Backup password database
Recover password database
Check database integrity
Repair password DB

© Ian Sommerville 2018:

Figure 6.4 Microservice support code

Microservice X

Service functionality	
Message management	Failure management
UI implementation	Data consistency management

Microservices architecture

- A microservices architecture is an *architectural style* – a tried and tested way of implementing a logical software architecture.
- This architectural style addresses two problems with monolithic applications
    - The whole system has to be rebuilt, re-tested and re-deployed when any change is made. This can be a slow process as changes to one part of the system can adversely affect other components.
    - As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions.

© Ian Sommerville 2018:

Benefits of microservices architecture

- Microservices are self-contained and run in separate processes.
- In cloud-based systems, each microservice may be deployed in its own container. This means a microservice can be stopped and restarted without affecting other parts of the system.
- If the demand on a service increases, service replicas can be quickly created and deployed. These do not require a more powerful server so 'scaling-out' is, typically, much cheaper than 'scaling up'.

© Ian Sommerville 2018:

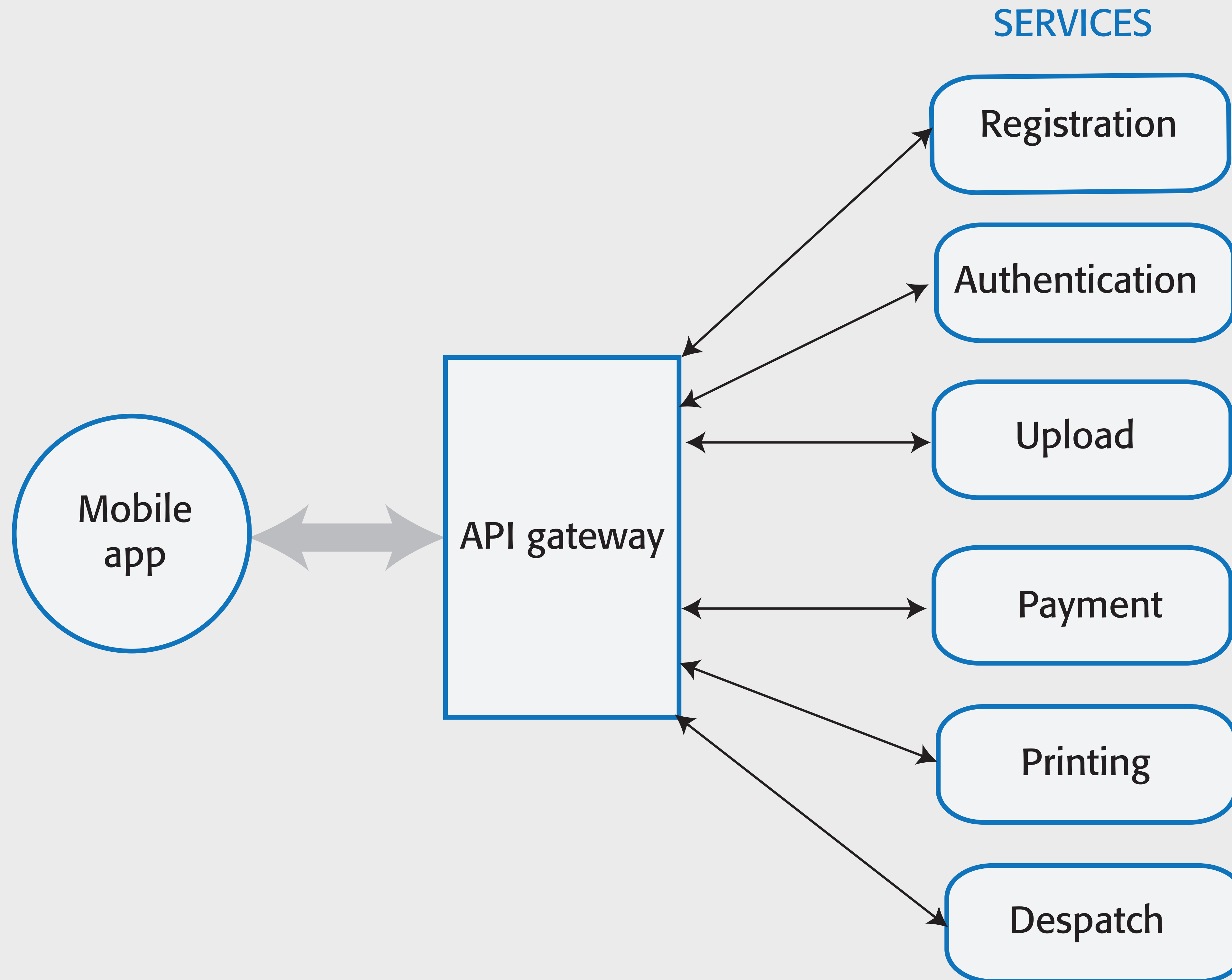
Table 6.2 A photo printing system for mobile devices

Imagine that you are developing a photo printing service for mobile devices. Users can upload photos to your server from their phone or specify photos from their Instagram account that they would like to be printed. Prints can be made at different sizes and on different media.

Users can chose print size and print medium. For example, they may decide to print a picture onto a mug or a T-shirt. The prints or other media are prepared and then posted to their home. They pay for prints either using a payment service such as Android or Apple Pay or by registering a credit card with the printing service provider.

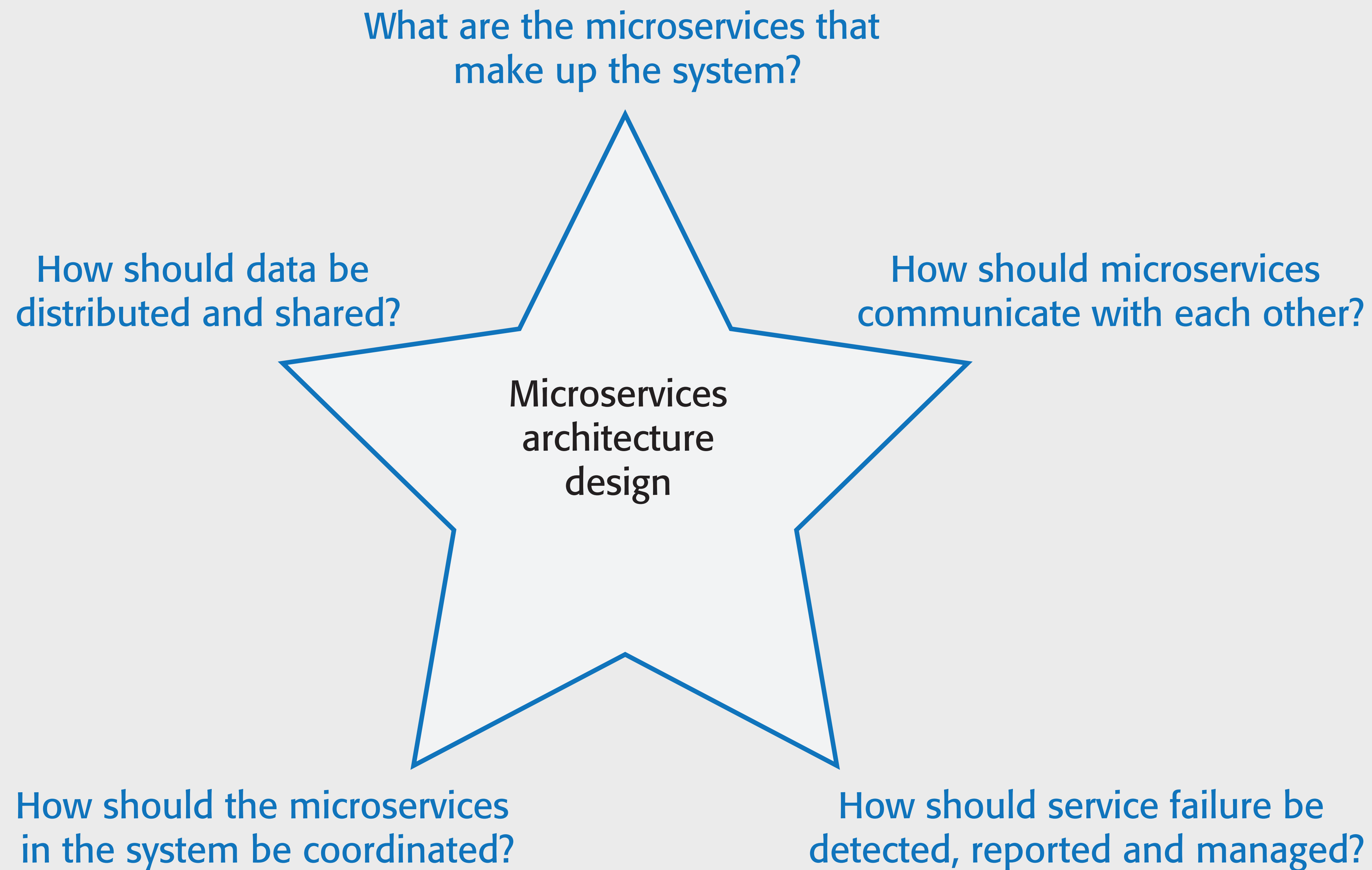
© Ian Sommerville 2018:

Figure 6.5 A microservices architecture for a photo printing system



© Ian Sommerville 2018:

Figure 6.6 Microservices architecture - key design questions



© Ian Sommerville 2018:

Decomposition guidelines

- Balance fine-grain functionality and system performance
  - Single-function services mean that changes are limited to fewer services but require service communications to implement user functionality. This slows down a system because of the need for each service to bundle and unbundle messages sent from other services.
- Follow the ‘common closure principle’
  - Elements of a system that are likely to be changed at the same time should be located within the same service. Most new and changed requirements should therefore only affect a single service.
- Associate services with business capabilities
  - A business capability is a discrete area of business functionality that is the responsibility of an individual or a group. You should identify the services that are required to support each business capability.
- Design services so that they only have access to the data that they need
  - If there is an overlap between the data used by different services, you need a mechanism to propagate data changes to all services using the same data.

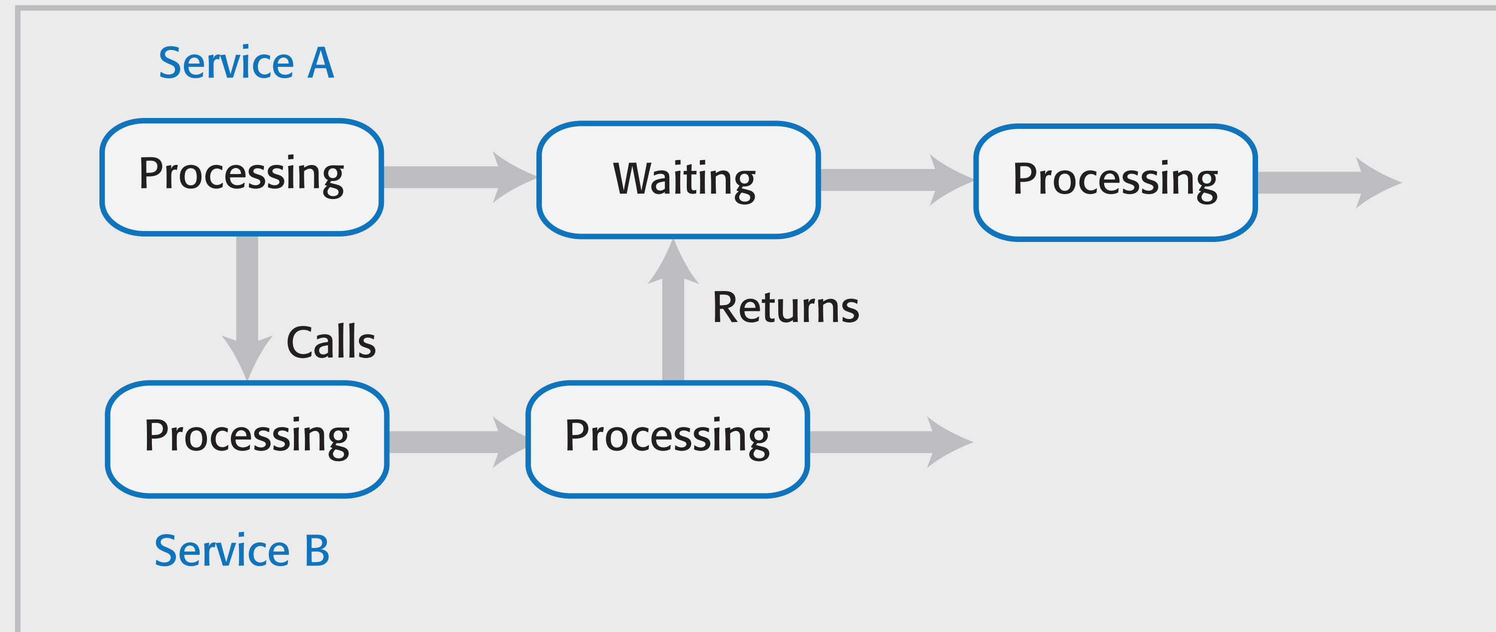
© Ian Sommerville 2018:

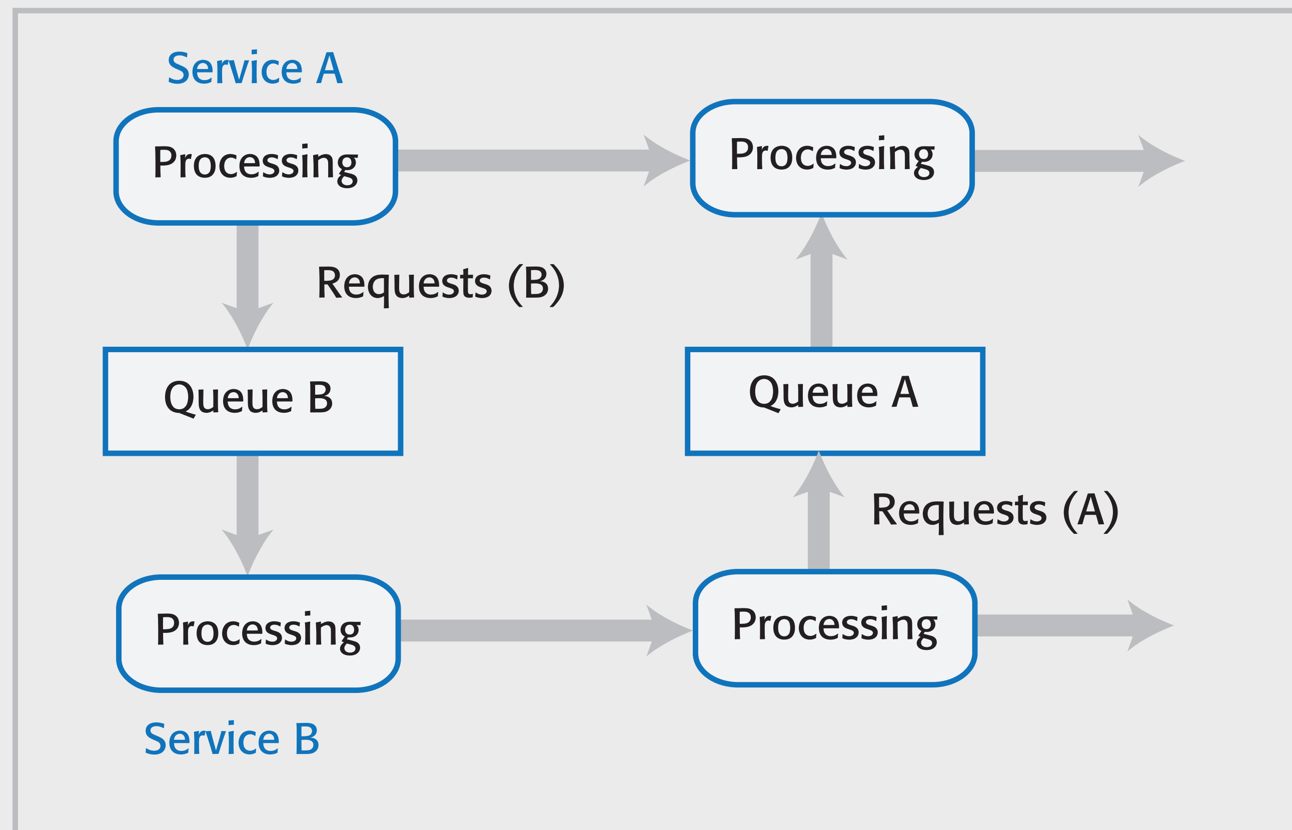
Service communications

- Services communicate by exchanging messages that include information about the originator of the message, as well as the data that is the input to or output from the request.
- When you are designing a microservices architecture, you have to establish a standard for communications that all microservices should follow. Some of the key decisions that you have to make are
  - should service interaction be synchronous or asynchronous?
  - should services communicate directly or via message broker middleware?
  - what protocol should be used for messages exchanged between services?

© Ian Sommerville 2018:

Figure 6.7 Synchronous and asynchronous microservice interaction





© Ian Sommerville 2018:

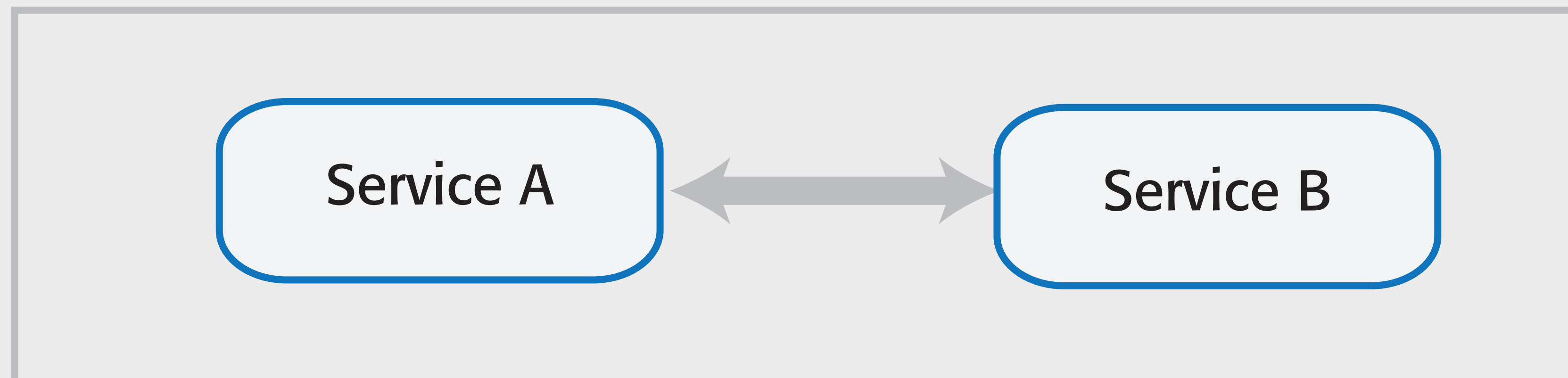
Synchronous and asynchronous interaction

- In a synchronous interaction, service A issues a request to service B. Service A then suspends processing while B is processing the request.
- It waits until service B has returned the required information before continuing execution.
- In an asynchronous interaction, service A issues the request that is queued for processing by service B. A then continues processing without waiting for B to finish its computations.
- Sometime later, service B completes the earlier request from service A and queues the result to be retrieved by A.
- Service A, therefore, has to check its queue periodically to see if a result is available.

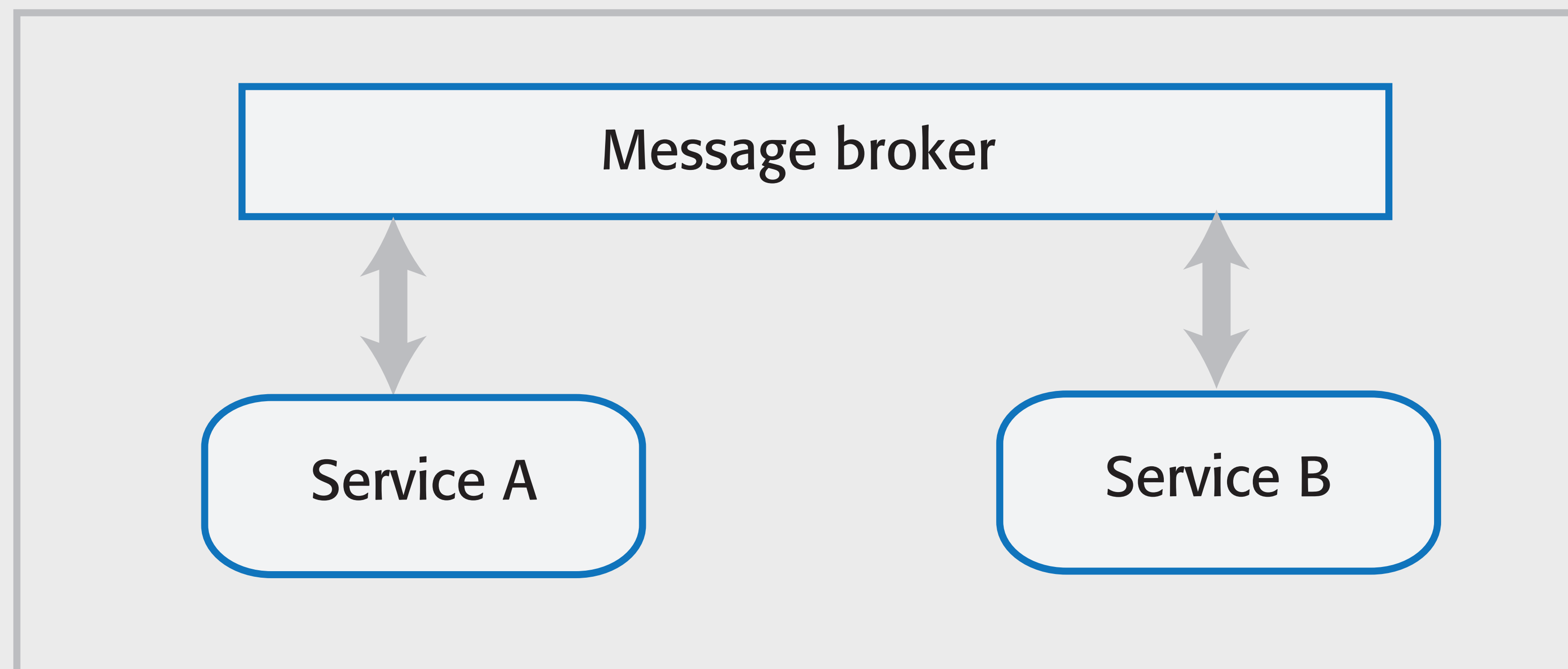
© Ian Sommerville 2018:

Figure 6.8 Direct and indirect service communication

Direct communication - A and B send messages to each other



Indirect communication - A and B communicate through a message broker



Direct and indirect service communication

- Direct service communication requires that interacting services know each other's address.
- The services interact by sending requests directly to these addresses.
- Indirect communication involves naming the service that is required and sending that request to a message broker (sometimes called a message bus).
- The message broker is then responsible for finding the service that can fulfil the service request.

© Ian Sommerville 2018:

Microservice data design

- You should isolate data within each system service with as little data sharing as possible.
- If data sharing is unavoidable, you should design microservices so that most sharing is 'read-only', with a minimal number of services responsible for data updates.
- If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Inconsistency management

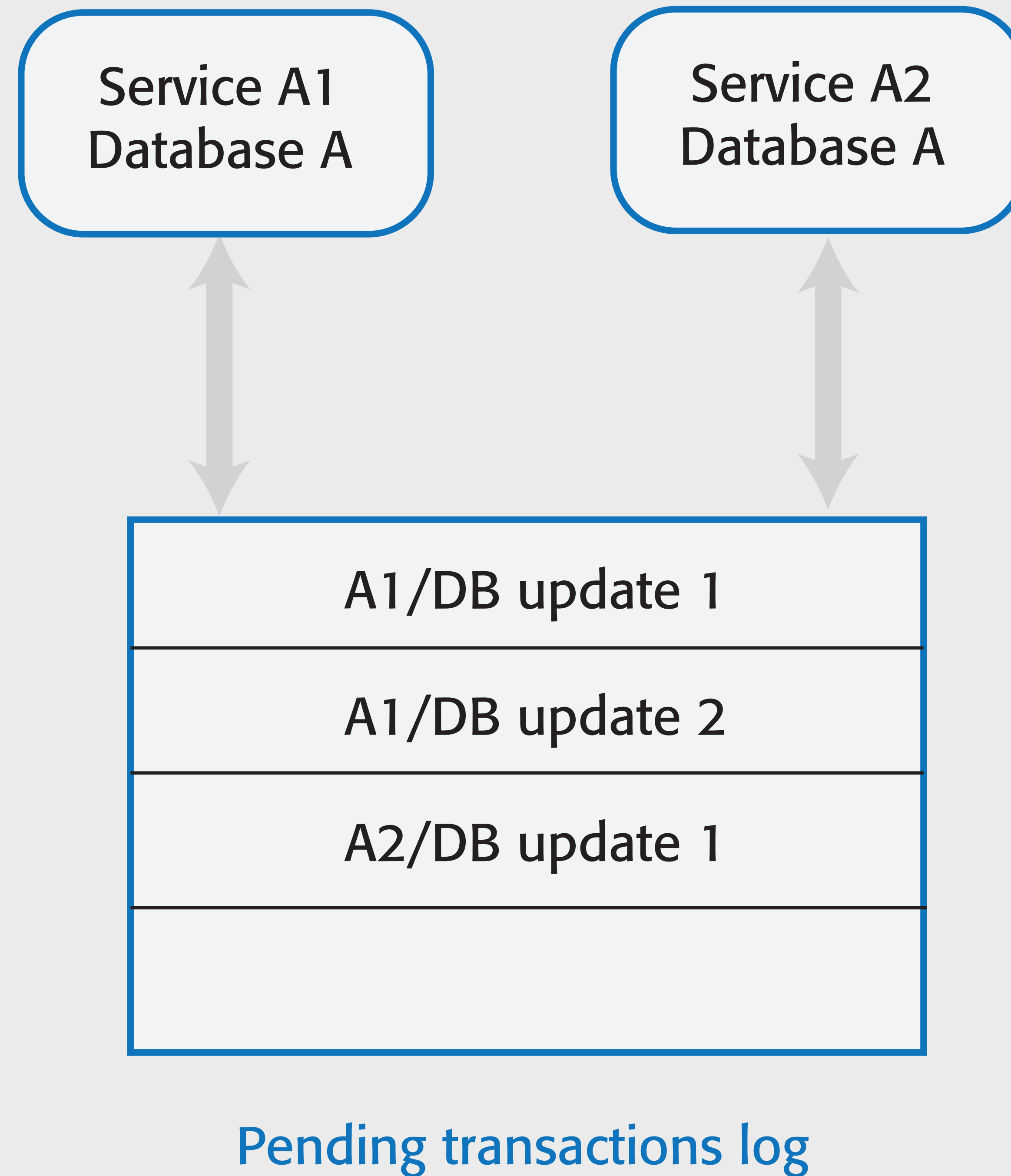
- An ACID transaction bundles a set of data updates into a single unit so that either all updates are completed or none of them are. ACID transactions are impractical in a microservices architecture.
- The databases used by different microservices or microservice replicas need not be completely consistent all of the time.
- Dependent data inconsistency
  - The actions or failures of one service can cause the data managed by another service to become inconsistent.
- Replica inconsistency
  - There are several replicas of the same service that are executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases 'eventually consistent' so that all replicas are working on the same data.

© Ian Sommerville 2018:

Eventual consistency

- Eventual consistency is a situation where the system guarantees that the databases will eventually become consistent.
- You can implement eventual consistency by maintaining a transaction log.
- When a database change is made, this is recorded on a ‘pending updates’ log.
- Other service instances look at this log, update their own database and indicate that they have made the change.

Figure 6.9 Using a pending transaction log



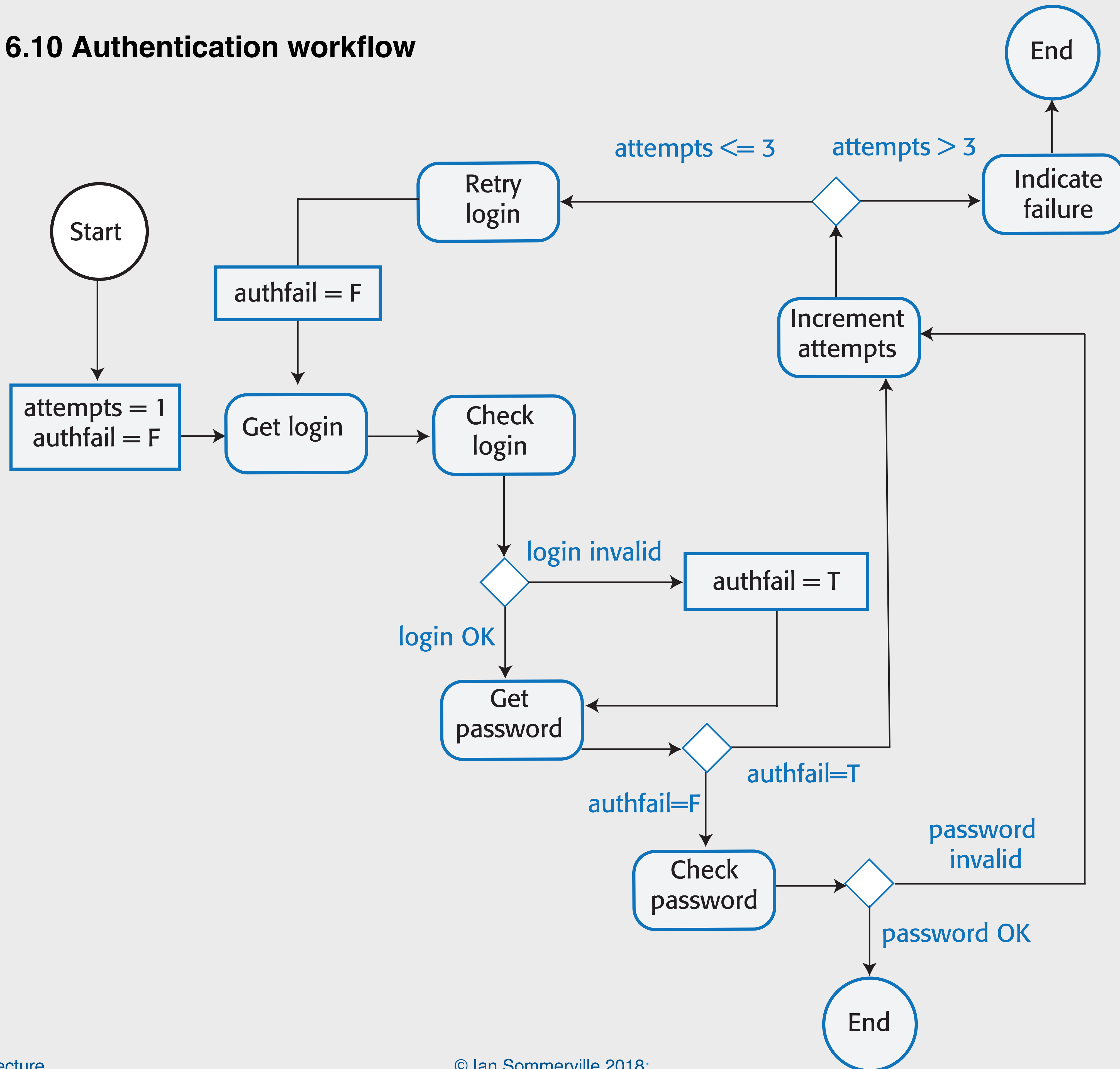
© Ian Sommerville 2018:

Service coordination

- Most user sessions involve a series of interactions in which operations have to be carried out in a specific order.
- This is called a workflow.
    - An authentication workflow for UID/password authentication shows the steps involved in authenticating a user.
    - In this example, the user is allowed 3 login attempts before the system indicates that the login has failed.

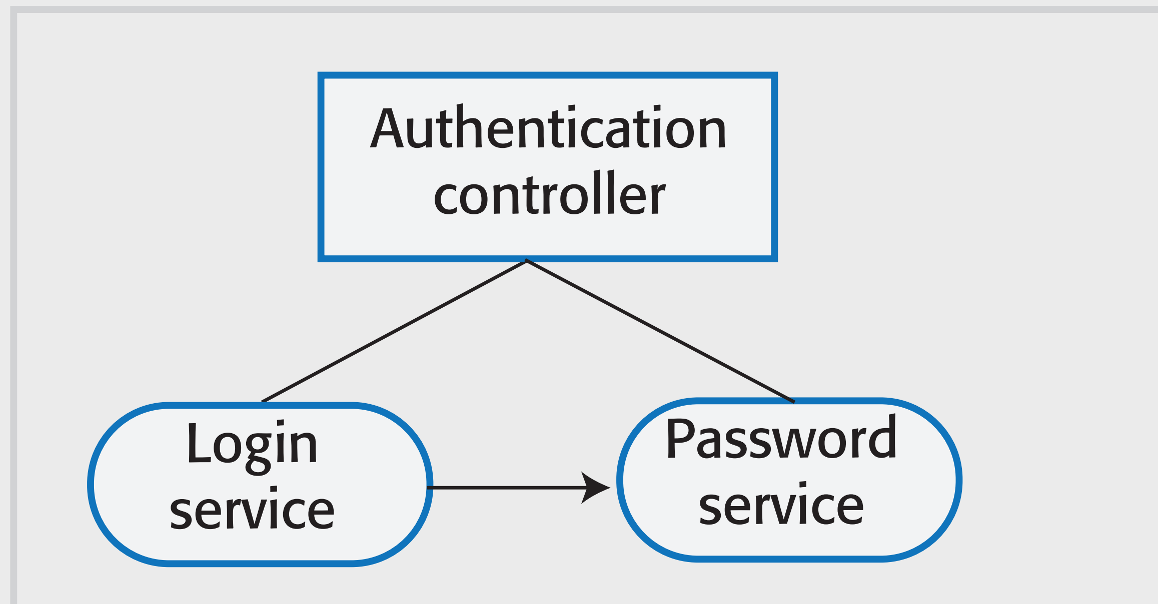
© Ian Sommerville 2018:

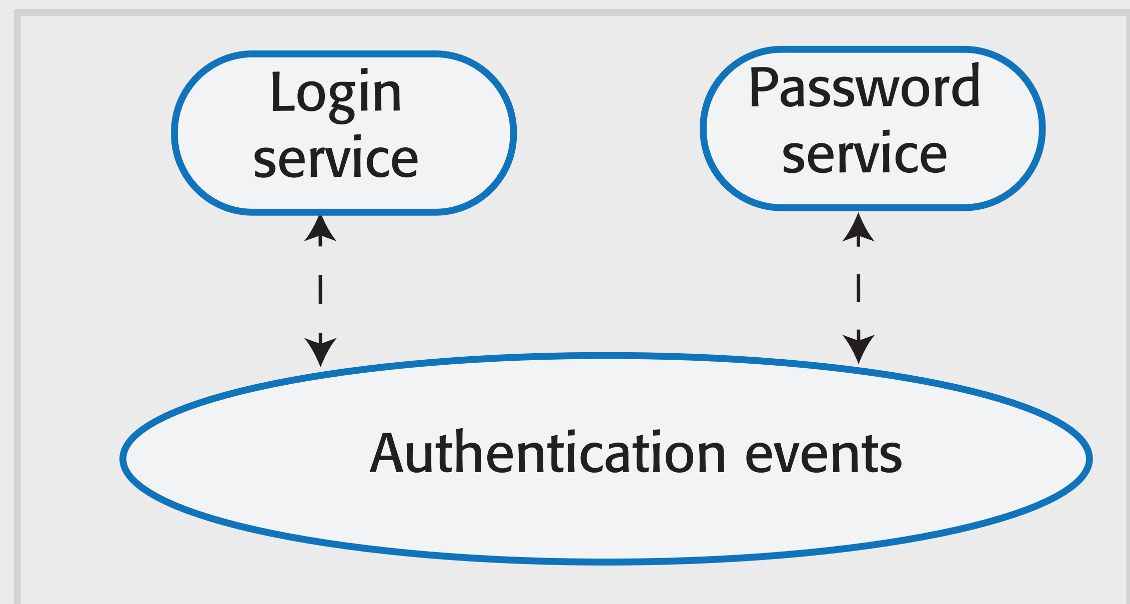
Figure 6.10 Authentication workflow



© Ian Sommerville 2018:

Figure 6.11 Orchestration and choreography





© Ian Sommerville 2018:

Table 6.3 Failure types in a microservices system

Internal service failure

These are conditions that are detected by the service and can be reported to the service client in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.

External service failure

These failures have an external cause, which affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to restart the service.

Service performance failure

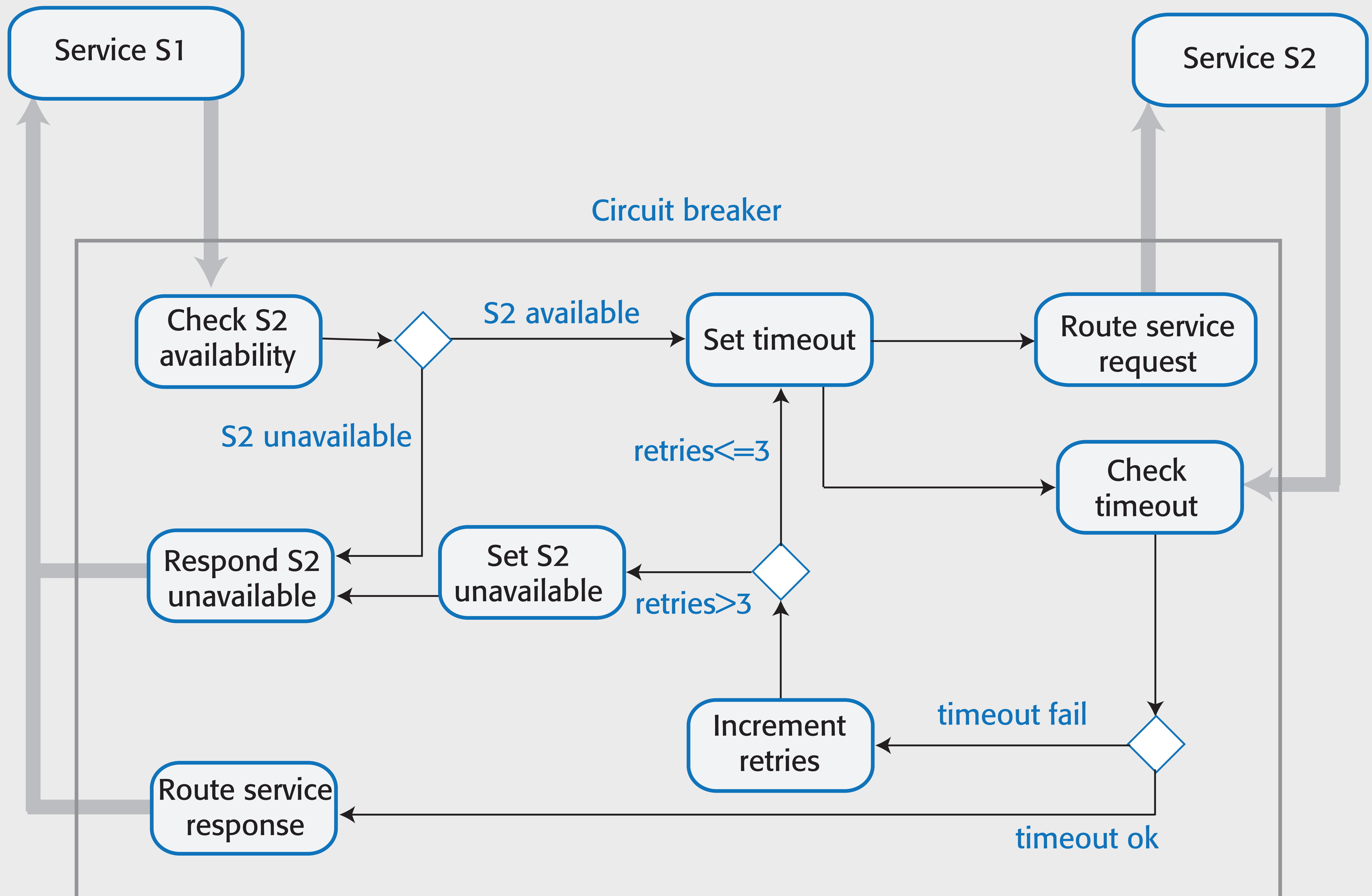
The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

© Ian Sommerville 2018:

Timeouts and circuit breakers

- A timeout is a counter that this associated with the service requests and starts running when the request is made.
- Once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly.
- The problem with the timeout approach is that every service call to a 'failed service' is delayed by the timeout value so the whole system slows down.
- Instead of using timeouts explicitly when a service call is made, he suggests using a circuit breaker. Like an electrical circuit breaker, this immediately denies access to a failed service without the delays associated with timeouts.

Figure 6.12 Using a circuit breaker to cope with service failure



RESTful services

- The REST (REpresentational State Transfer) architectural style is based on the idea of transferring representations of digital resources from a server to a client.
    - You can think of a resource as any chunk of data such as credit card details, an individual's medical record, a magazine or newspaper, a library catalogue, and so on.
    - Resources are accessed via their unique URI and RESTful services operate on these resources.
- This is the fundamental approach used in the web where the resource is a page to be displayed in the user's browser.
    - An HTML representation is generated by the server in response to an HTTP GET request and is transferred to the client for display by a browser or a special-purpose app.

© Ian Sommerville 2018:

Table 6.4 RESTful service principles

Use HTTP verbs

The basic methods defined in the HTTP protocol (GET, PUT, POST, DELETE) must be used to access the operations made available by the service.

Stateless services

Services must never maintain internal state. As I have already explained, microservices are stateless so fit with this principle.

URI addressable

All resources must have a URI, with a hierarchical structure, that is used to access sub-resources.

Use XML or JSON

Resources should normally be represented in JSON or XML or both. Other representations, such as audio and video representations, may be used if appropriate.

© Ian Sommerville 2018:

Table 6.5 RESTful service operations

Create
Implemented using HTTP POST, which creates the resource with the given URI. If the resource has already been created, an error is returned.

Read
Implemented using HTTP GET, which reads the resource and returns its value. GET operations should never update a resource so that successive GET operations with no intervening PUT operations always return the same value.

Update
Implemented using HTTP PUT, which modifies an existing resource. PUT should not be used for resource creation.

Delete
Implemented using HTTP DELETE, which makes the resource inaccessible using the specified URI. The resource may or may not be physically deleted.

© Ian Sommerville 2018:

Road information system

- Imagine a system that maintains information about incidents, such as traffic delays, roadworks and accidents on a national road network. This system can be accessed via a browser using the URL:
  - https://trafficinfo.net/incidents/
- Users can query the system to discover incidents on the roads on which they are planning to travel.
- When implemented as a RESTful web service, you need to design the resource structure so that incidents are organized hierarchically.
  - For example, incidents may be recorded according to the road identifier (e.g. A90), the location (e.g. stonehaven), the carriageway direction (e.g. north) and an incident number (e.g. 1). Therefore, each incident can be accessed using its URI:
  - https://trafficinfo.net/incidents/A90/stonehaven/north/1

Table 6.6 Incident description

Incident ID: A90N17061714391

Date: 17 June 2017

Time reported: 1439

Severity: Significant

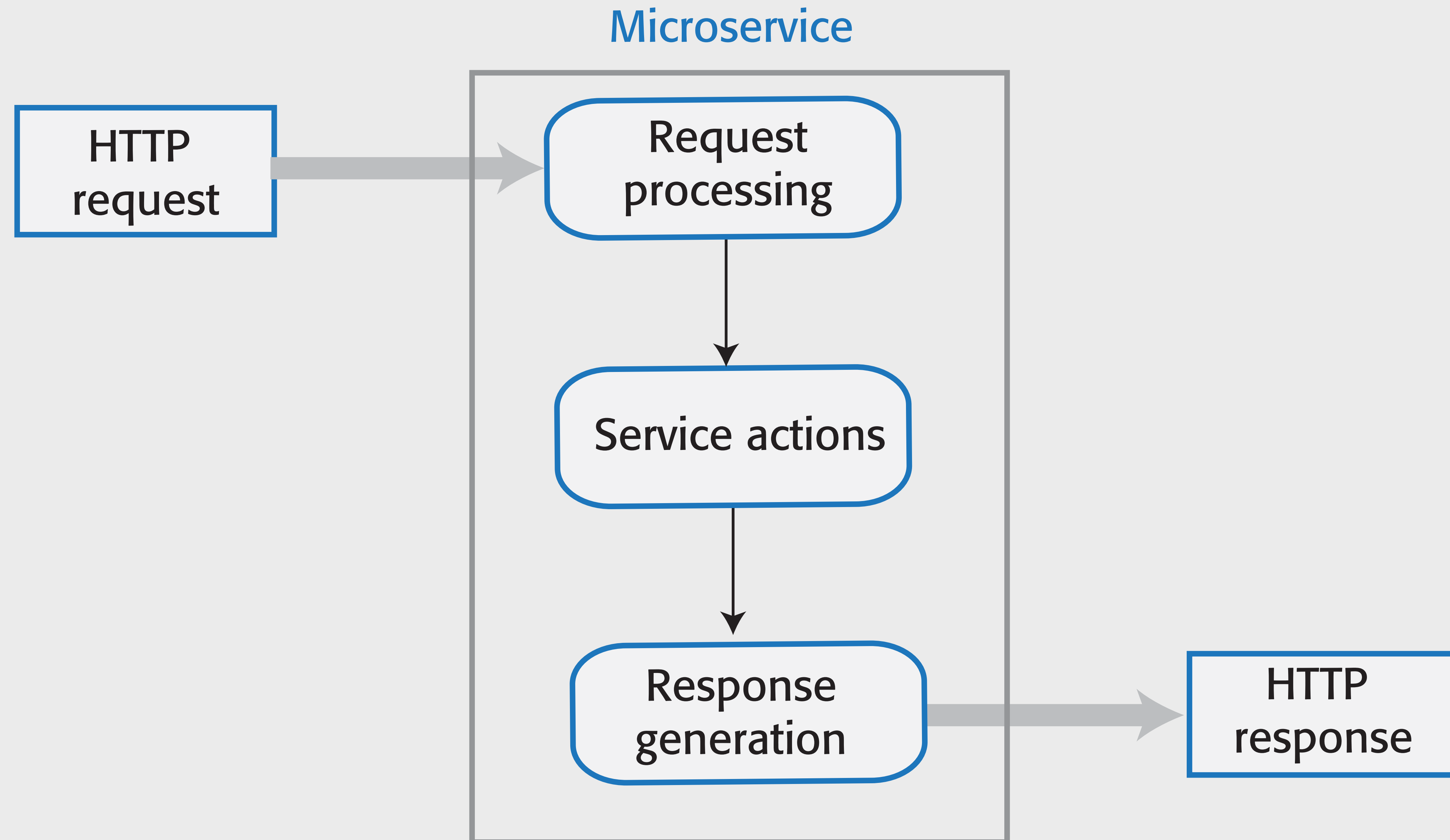
Description: Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes

© Ian Sommerville 2018:

Service operations

- Retrieve
  - Returns information about a reported incident or incidents. Accessed using the GET verb.
- Add
  - Adds information about a new incident. Accessed using the POST verb.
- Update
  - Updates the information about a reported incident. Accessed using the PUT verb.
- Delete
  - Deletes an incident. The DELETE verb is used when an incident has been cleared.

Figure 6.13 HTTP request and response processing



© Ian Sommerville 2018:

Figure 6.14 HTTP request and response message organization

© Ian Sommerville 2018:

Table 6.7 XML and JSON descriptions

JSON

{
id: “A90N17061714391”,
“date”: “20170617”,
“time”: “1437”,
“road_id”: “A90”,
“place”: “Stonehaven”,
“direction”: “north”,
“severity”: “significant”,
“description”: “Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.”
}

Table 6.7 XML and JSON descriptions

XML

<id>
A90N17061714391
</id>
<date>
20170617
</date>
<time>
1437
</time>
…
<description>Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.
</description>

© Ian Sommerville 2018:

Figure 6.15 A GET request and the associated response

REQUEST

GET	incidents/A90/stonehaven/	HTTP/1.1

Host: trafficinfo.net
...
Accept: text/json, text/xml, text/plain
Content-Length: 0

RESPONSE

HTTP/1.1	200

...
Content-Length: 461
Content-Type: text/json

{
“number”: “A90N17061714391”,
“date”: “20170617”,
“time”: “1437”,
“road_id”: “A90”,
“place”: “Stonehaven”,
“direction”: “north”,
“severity”: “significant”,
“description”: “Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.”
}
{
“number”: “A90S17061713001”,
“date”: “20170617”,
“time”: “1300”,
“road_id”: “A90”,
“place”: “Stonehaven”,
“direction”: “south”,
“severity”: “minor”,
“description”: “Grass cutting on verge. Minor delays”
}

Service deployment

- After a system has been developed and delivered, it has to be deployed on servers, monitored for problems and updated as new versions become available.
- When a system is composed of tens or even hundreds of microservices, deployment of the system is more complex than for monolithic systems.
- The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is no 'standard' deployment configuration for all services.
- It is now normal practice for microservice development teams to be responsible for deployment and service management as well as software development and to use continuous deployment.
- Continuous deployment means that as soon as a change to a service has been made and validated, the modified service is redeployed.

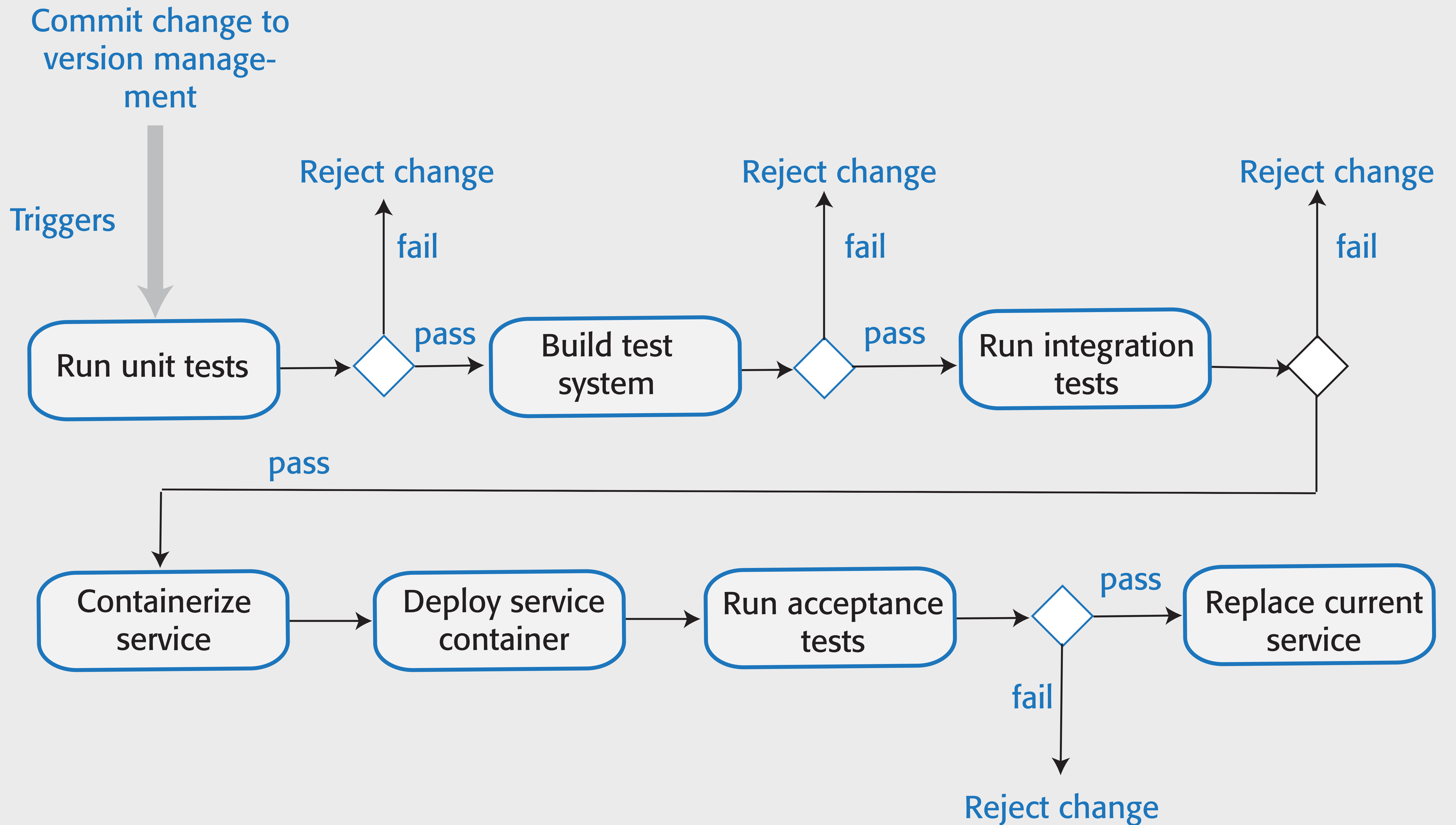
© Ian Sommerville 2018:

Deployment automation

- Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software.
- If the software ‘passes’ these tests, it then enters another automation pipeline that packages and deploys the software.
- The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git.
- This triggers a set of automated tests that run using the modified service. If all service tests run successfully, a new version of the system that incorporates the changed service is created.
- Another set of automated system tests are then executed. If these run successfully, the service is ready for deployment.

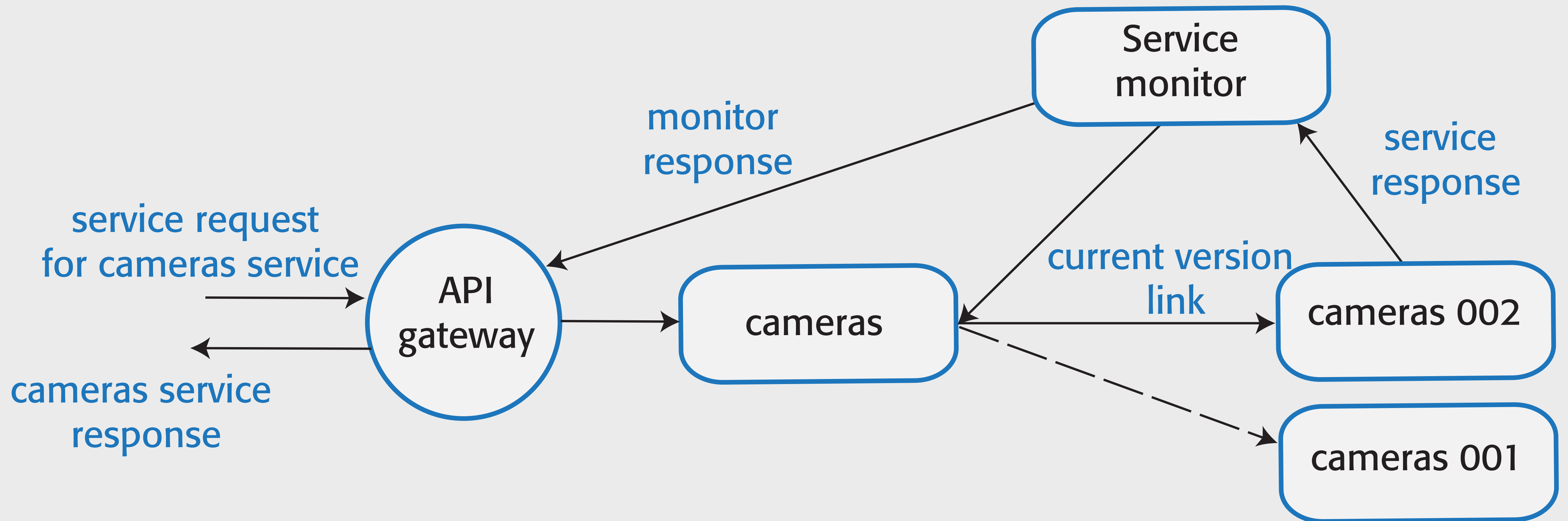
© Ian Sommerville 2018:

Figure 6.16 A continuous deployment pipeline



© Ian Sommerville 2018:

Figure 6.17 Versioned services



© Ian Sommerville 2018:

Key points 1

- A microservice is an independent and self-contained software component that runs in its own process and communicates with other microservices using lightweight protocols.
- Microservices in a system can be implemented using different programming languages and database technologies.
- Microservices have a single responsibility and should be designed so that they can be easily changed without having to change other microservices in the system.
- Microservices architecture is an architectural style in which the system is constructed from communicating microservices. It is well-suited to cloud based systems where each microservice can run in its own container.
- The two most important responsibilities of architects of a microservices system are to decide how to structure the system into microservices and to decide how microservices should communicate and be coordinated.

© Ian Sommerville 2018:

Key points 2

- Communication and coordination decisions include deciding on microservice communication protocols, data sharing, whether services should be centrally coordinated, and failure management.
- The RESTful architectural style is widely used in microservice-based systems. Services are designed so that the HTTP verbs, GET, POST, PUT and DELETE, map onto the service operations.
- The RESTful style is based on digital resources that, in a microservices architecture, may be represented using XML or, more commonly, JSON.
- Continuous deployment is a process where new versions of a service are put into production as soon as a service change has been made. It is a completely automated process that relies on automated testing to check that the new version is of 'production quality'.
- If continuous deployment is used, you may need to maintain multiple versions of deployed services so that you can switch to an older version if problems are discovered in a newly-deployed service.

© Ian Sommerville 2018: