Технології створення Web-застосунків

Модуль 4.

Доц. Попівщий В.І. ЗНУ каф.ПЗАС 2020

ОСНОВНІ ДЖЕРЕЛА

1. Хорсдал К. Микросервисы на платформе .NET. – СПб. : Питер, 2018. 352 с.
2. Ньюмен С. Создание микросервисов. – СПб.: Питер, 2016. – 304 с.
3. Фаулер М. Шаблоны корпоративных приложений. – М.: Вильямс, 2016. – 544 с.
4. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. 544 с.
5. Cole Matt R. Hands-On Microservices with C#. – Packt Publishing, 2018. – 234 p.
6. Newman Sam Monolith to Microservices. – O'Reilly Media, 2020. – 256 p.
7. Gaurav Aroraa, Ed Price Hands-On Microservices with C# 8 and .NET Core 3 Third Edition. – Packt Publishing, 2020. – 451 p.

8. Morgan Bruce, Paulo A. Pereira **Microservices in Action**. – Manning Publications, 2019. – 366 p.
9. Essentials of Microservices Architecture. Paradigms, Applications, and Techniques. - CRC Press: 2020. - 293 p.
10. Tayo Koleoso Beginning Quarkus Framework: Build Cloud-Native Enterprise Java Applications and Microservices. – Apress, 2020. – 297p. https://scanlibs.com/beginning-quarkus-framework-cloud-native-apps/
11. Binildas Christudas Practical Microservices Architectural Patterns. – Apress, 2019. – 902 p.
12. Моуэт Т. Использование Docker – М.: ДМК Пресс, 2017. – 354 с.
13. .NET Microservices: Architecture for Containerized .NET Applications. – Microsoft Corporation, 2020

ТРЕНІНГИ

- Тренінг «Мікросервіси для Java розробників» https://itcluster.lviv.ua/paged/training-microservices-for-java-developers/
- Курси Microservices https://www.nobleprog.com.ua/kursy-microservices

Огляд відеокурсів

1. Linkedin - Learning Creating Your First Spring Boot Microservice 2019
2. **Lynda - Azure Microservices with .NET Core for Developers 2020**
3. Pluralsight - Building Microservices (2019)
4. Pluralsight – Microservices The Big Picture (2018)
5. ASP.NET Core 2.0 E-commerce Web Site Based on Microservices (2019)
6. Lynda - Microservices Design Patterns (2020)
7. Packt – A Beginner's Guide to a Microservices Architecture
8. O'Reilly - Building Microservice Systems with Docker and Kubernetes

Модуль 4. Зберігання даних мікросервісів. Взаємодія мікросервісів. Реалізація запитів в мікросервісній архітектурі.

Дослідження мікросервісной архітектури показує, що один з аспектів, які турбують найбільше, пов'язаний з реалізацією транзакцій, що охоплюють кілька сервісів.

Транзакції - незамінний компонент будь-якого промислового застосунку. Без них неможливо підтримувати узгодженість даних.

Транзакції типу ACID

- Транзакції типу ACID (Atomicity, Consistency, Isolation, Durability - «атомарность, узгодженість, ізольованість, довговічність») значно спрощують життя розробників, створюючи ілюзію того, що кожна з них має ексклюзивний доступ до даних.
- У мікросервісной архітектурі ACID-транзакції можуть використовувати навіть запити, що виконуються в рамках одного сервісу.
- Однак основна складність полягає в реалізації операцій, які **оновлюють дані, що належать різним сервісам**.

- Замість ACID-транзакцій операція, що охоплює кілька сервісів і прагне підтримувати узгодженість даних, повинна використовувати те, що називається **розповіддю (або сагою),** - послідовність локальних транзакцій на основі повідомлень.
- Одна з **проблем оповідань** пов'язана з тим, що за своєю природою вони є ACD (Atomicity, Consistency, Durability - «атомарность, узгодженість, довговічність»). Їм не вистачає підтримки ізольованості, яка є в ACID-транзакції.
- В результаті програма має використовувати так звані **контрзаходи** - методики проектування, які усувають або знижують вплив аномалій конкурентності, викликаних нестачею ізольованості.

Два способи координації

Далі ми поговоримо про два способи координації оповідань:

- **хореографії**, коли учасники обмінюються подіями без централізованої точки управління, і
- **оркестрації**, коли централізований контролер каже учасникам оповідання, які операції потрібно виконати.

Моделі зв'язку для мікросервісів

(9. Essentials of Microservices Architecture. Paradigms, Applications, and Techniques. - CRC Press: 2020. – Chapter 3)

Мета

При розробці програми на основі MSA існують різні ключові архітектурні елементи.

Мета цього розділу - дати огляд різних архітектурних елементів, що становлять MSA, та представити один з найважливіших архітектурних елементів: комунікаційні моделі для MSA.

Тут ми розглянемо як служби взаємодіють між собою синхронно та асинхронно, використовуючи стандартні архітектури, протоколи та брокери повідомлень.

Передмова

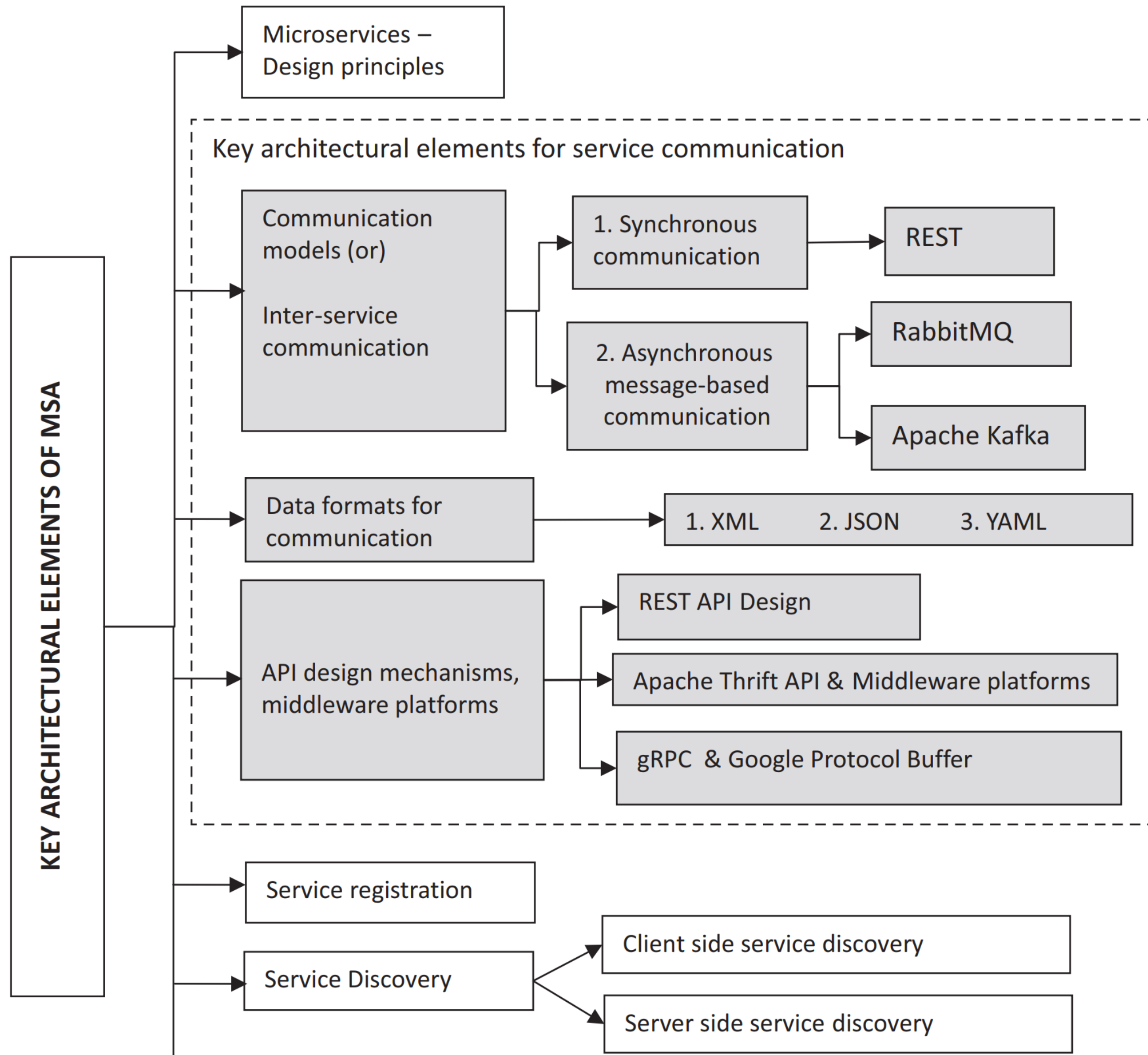
- З попередніх розділів ми дізналися, що метою MSA було забезпечення частого розгортання та безперервної доставки програм, які, як правило, розробляються за допомогою гнучкої моделі програмного процесу.
- В цьому розділі ми поступово розглянемо різні архітектурні елементи MSA.
- Спочатку в цьому розділі представлений огляд різних архітектурних елементів MSA та принципів проектування мікросервісів.
- Потім у цьму розділі описуються дві основні моделі комунікації, а саме модель синхронної комунікації та модель асинхронної комунікації.

- Особлива увага приділяється трьом загальновживаним протоколам зв'язку, а саме
  - Representational State Transfer (**REST**) для синхронного зв'язку;
  - Advanced Message Queuing Protocol (**AMQP**) та його реалізація з відкритим кодом,
  - **RabbitMQ**, для асинхронного спілкування на основі повідомлень; і
  - асинхронний брокер повідомлень з високою масштабованістю та високою пропускною здатністю, а саме **Apache Kafka**.

Основні архітектурні елементи MSA

- Оскільки MSA - це архітектура, що розвивається, правильне розуміння різних елементів архітектури стає важливим і є необхідною умовою для успішного розвитку програми на основі MSA. Основні елементи MSA показані на Рис. 3.1. Ключові елементи MSA включають:
- Принципи проектування мікросервісів
- Моделі зв'язку для мікросервісів
- Формати даних для мікросервісів
- Розробка API та проміжне програмне забезпечення для мікросервісів
- Реєстрація сервісу

- Виявлення сервісів
- Розробка точок входу до програми MSA – API gateway
- Композиція сервісів
- Транзакції з базами даних
- Розгортання служби
- Безпека


KEY ARCHITECTURAL ELEMENTS OF MSA
Microservices –
Design principles
Key architectural elements for service communication
Communication models (or)
Inter-service communication
1. Synchronous communication
REST
2. Asynchronous message-based communication
RabbitMQ
Apache Kafka
Data formats for communication
1. XML
2. JSON
3. YAML
API design mechanisms, middleware platforms
REST API Design
Apache Thrift API & Middleware platforms
gRPC & Google Protocol Buffer
Service registration
Service Discovery
Client side service discovery
Server side service discovery

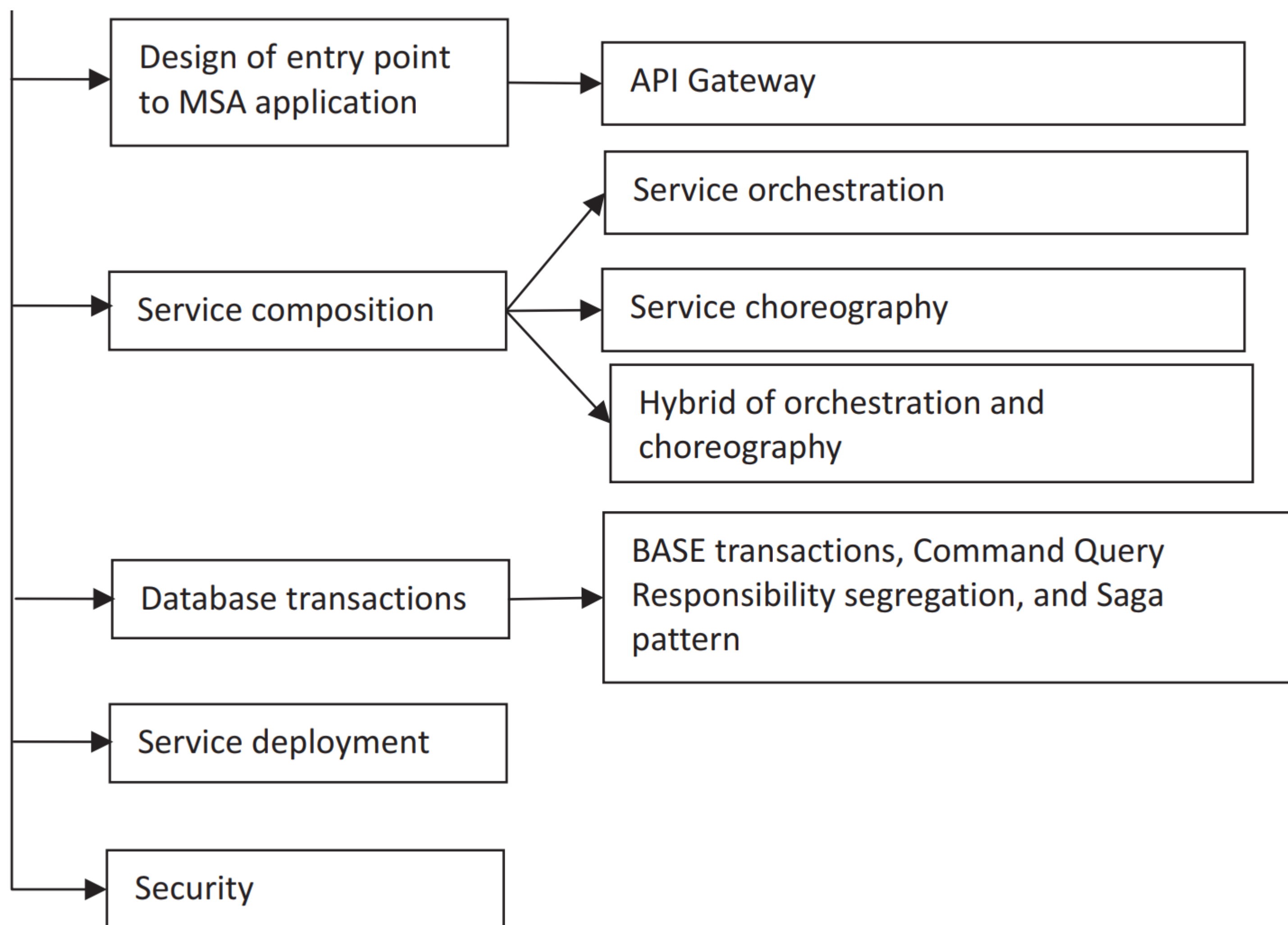


Figure 3.1 Key architectural elements of MSA

Microservices—Design Principles

- Додаток на базі MSA слід розділити на багато служб залежно від їх функціональних можливостей.
- Кожна послуга повинна виконувати лише одну функцію.
- Дизайн мікропослуг повинен базуватися на принципі єдиної відповідальності. Це робить послуги як незалежними, так і малими.
- Це полегшує індивідуальне розгортання, забезпечення індивідуальної масштабованості та просте обслуговування.

Communication Model/Inter-Service Communication

- Оскільки сервіси розроблені за принципом єдиної відповідальності, часто виникають ситуації, коли одна служба повинна спілкуватися з іншою службою.
- Наприклад, сервіс, який називається послугою пересилки, буде шукати платіжну службу, щоб перевірити, чи було здійснено оплату чи ні.
- Отже, взаємодія між службами є загальною вимогою MSA.
- Це досягається двома шляхами, а саме синхронним та асинхронним. При синхронному спілкуванні як відправник, так і одержувач перебувають у мережі. Протокол REST широко використовується для реалізації синхронного зв'язку між службами.

- При асинхронному спілкуванні відправник і одержувач перебувають в автономному режимі, і це обмін повідомленнями.
- AMQP зазвичай використовується для асинхронного спілкування, орієнтованого на повідомлення.
- Інструмент RabbitMQ - це проміжне програмне забезпечення, орієнтоване на повідомлення, яке реалізує AMQP.
- Для того, щоб мати високу пропускну здатність повідомлень, можуть використовуватися брокери повідомлень, такі як Apache Kafka.

API Gateway—Entry Point to an MSA Application

- У MSA програма розділена на багато мікросервісів відповідно до функціональних можливостей. Навіть невелика програма складатиметься із значної кількості мікросервісів.
- У цій ситуації, якщо клієнт безпосередньо спілкується із додатком, він повинен викликати кожну послугу через зовнішню мережу, таку як Інтернет. Цей тип прямого спілкування між клієнтом та додатком на основі MSA передбачає величезний мережевий трафік, і, отже, він не вважається доброю практикою.
- Архітектурний елемент, який називається шлюзом API, використовується як одна точка входу до клієнта. Він отримує запити від усіх клієнтів і робить необхідні подальші запити до відповідних служб у внутрішній мережі.
- Цей вид доступу до програми не тільки приховує пряме відкриття послуг клієнтом, але також покращує ефективність роботи постачальника, оскільки запити подаються у внутрішній мережі.

Service Composition

- Бувають ситуації, коли запит клієнта не може бути виконаний однією службою, а запит виконується шляхом складанням більше ніж однієї служби за певним шаблоном виконання. Процес об'єднання більше однієї послуги в певному шаблоні виконання, який називається робочим процесом (workflow), для досягнення заданого бізнес-процесу називається service composition.
- Композиція сервісу може бути виконана двома способами - оркестровка служби та хореографія служби. В service orchestration компонент центрального координатора, який містить логіку досягнення даного бізнес-процесу, використовується для виклику окремих служб відповідно до логіки. У хореографії послуг логіка обробки розподіляється між самими службами-учасниками. Наприклад, клієнт може викликати першу послугу, а в кінці першої послуги перша послуга потім викликає наступну послугу, і це триває до тих пір, поки бажаний процес не поверне клієнту результати.

Database Transactions

- Відповідно до принципів проектування мікросервісів, кожна служба повинна мати власну базу даних/таблицю.
- У системах управління базами даних транзакції використовуються як метод реалізації властивостей бази даних **A**tomicity (атомності), **C**onsistency (узгодженості), **I**solation (ізоляції), and **D**urability (довговічності) – **ACID**.
- Традиційно для реалізації транзакцій у розподілених базах даних використовується протокол Two-Phase-Commit (протокол 2PC).
- Але в мікросервісах, оскільки бази даних децентралізовані, впровадження 2PC вимагає як часу, так і ресурсів. Як принцип дизайну, <u>краще уникати транзакцій </u>у MSA.

- Якщо взагалі транзакції потрібні в обов’язковому порядку, на відміну від транзакції ACID у звичайних системах, транзакції **B**asically **A**vailable (В основному доступні), **S**oft state (М’який стан), **E**ventually consistent (Врешті-решт) (**BASE**) реалізуються із шаблонами дизайну, такими як шаблон Saga.
- **BASE** транзакції підходять для додатків з великими веб-сайтами та великою кількістю реплікованих наборів даних.
- У шаблоні Saga кожне атомне завдання розглядається як локальна транзакція, завдання здійснюється локально, а статус передається центральному координатору. Якщо будь-яка з локальних транзакцій зазнає збою, тоді координатор доручає кожній службі, що бере участь, відмінити здійснену зміну. Процес відкоту знову виконується як чергова локальна транзакція. Таким чином, транзакції BASE забезпечують можливу узгодженість даних.

- Крім того, MSA приймає шаблон розподілу відповідальності за командні запити - **C**ommand **Q**uery **R**esponsibility **S**egregation (**CQRS**), який передбачає використання двох різних екземплярів бази даних, один з яких використовується для операцій запису, а інший - для операцій читання.
- Наявність двох екземплярів бази даних допомагає зменшити проблеми зі збереженням даних та забезпечує високу доступність та безпеку. Таким чином, MSA забезпечує узгодженість даних із шаблоном Saga, високий рівень безпеки, доступність та продуктивність із шаблоном CQRS.

Принципи проектування мікросервісів

- Першим основним елементом MSA є мікросервіси.
- Мікросервіси створюються шляхом розділення програми на набір служб на основі функціональних можливостей, як правило, відповідно до Single Responsibility Principle Мартіна Фаулера (SRP).
- Додаток повинен бути розроблений таким чином, щоб складатися з декількох одноцільових мікросервісів. Основним критерієм проектування мікросервісів є те, що вони "роблять лише одне і роблять це добре".
- Послуги повинні бути автономними та незалежними. Одна служба не повинна втручатися в іншу службу за її функціональністю, тому вона здатна до незалежного розгортання.

Загальна практика декомпозиції програми на мікросервіси полягає у застосуванні принципу обмеженого контексту **D**omain **D**riven **D**esign (**DDD**). Простими словами, під час проектування мікросервісів виконайте такі дії:

1. Почніть із якнайменшого мікросервісу та створіть межу (boundary) навколо нього.
2. Перевірте, чи є взаємодія поза межами.
3. Якщо так, розширте межу, щоб включити взаємодію або дозвольте розслаблене спілкування (relaxed communications).
4. Таким чином, мікросервіс вдосконалюється на основі межі зв'язку, поки не виходить як окремий обмежений контекст. Наступні ключові моменти можуть бути розглянуті під час відокремлення обмеженого контексту.

- Якщо будь-яким двом службам потрібно багато взаємодіяти між собою, це означає, що вони працюють для однієї і тієї ж концепції або функції, і ці дві послуги слід об'єднати в одну службу.
- При розробці мікропослуг слід шукати окрему межу зв'язку (*communication boundary*).
- Межі зв'язку також відображаються в *автономності мікросервісів*. Якщо одна служба взаємодіє з іншими службами для досягнення своєї мети, очевидно, що вона не є автономною, і служба повинна бути перероблена, щоб включити залежність.
- Тут зауважте, що розмір служби не визначає розгортання; це вирішує автономний характер послуги.

- Таким чином, принципи проектування мікросервісів включають
  - (i) Принцип єдиної відповідальності,
  - (ii) Проектування мікросервісів відповідно до обмеженого контексту в Domain Driven Design. Це побічно підкреслює окремі межі зв'язку та автономність мікросервісів.
  - (iii) Перевірку, чи здатна служба незалежно розгортатися.

Приклад (Matt McLarty. Designing a System of Microservices. 2017)

- Розглянемо приклад. Цей приклад пояснює поняття домену, субдомену та послуг.
- Домен - це широке поняття, яке має один або кілька субдоменів.
- Субдомени - це незалежні області, і вони слабо пов'язані між собою.
- Кожен субдомен може мати одну або кілька мікросервісів, які ідентифікуються на основі однієї ділової можливості.

- Домен, задіяний у наведеному вище прикладі, - роздрібний (retail) банк.
- Роздрібні банківські послуги також називають споживчими (consumer) банківськими послугами, які орієнтовані на індивідуальність та пропонують широкий спектр послуг, таких як
- (i) ощадні рахунки,
- (ii) чекові рахунки,
- (iii) іпотечні кредити,
- (iv) особисті позики та
- (v) дебетові / кредитні картки для фізичних осіб.

- Роздрібний банкінг пропонує приватним особам онлайн-банкінг.
- Зазвичай клієнти використовують Інтернет-банкінг за допомогою веб- або мобільних додатків.
- Інший варіант, за допомогою якого клієнти здійснюють покупку, - це точка продажу Point of Sale (POS). Тут клієнт або гортає свою картку, або вводить свою карту з чіпом у термінал продавця. Ці операції можна здійснити за допомогою дебетової картки або кредитної картки.

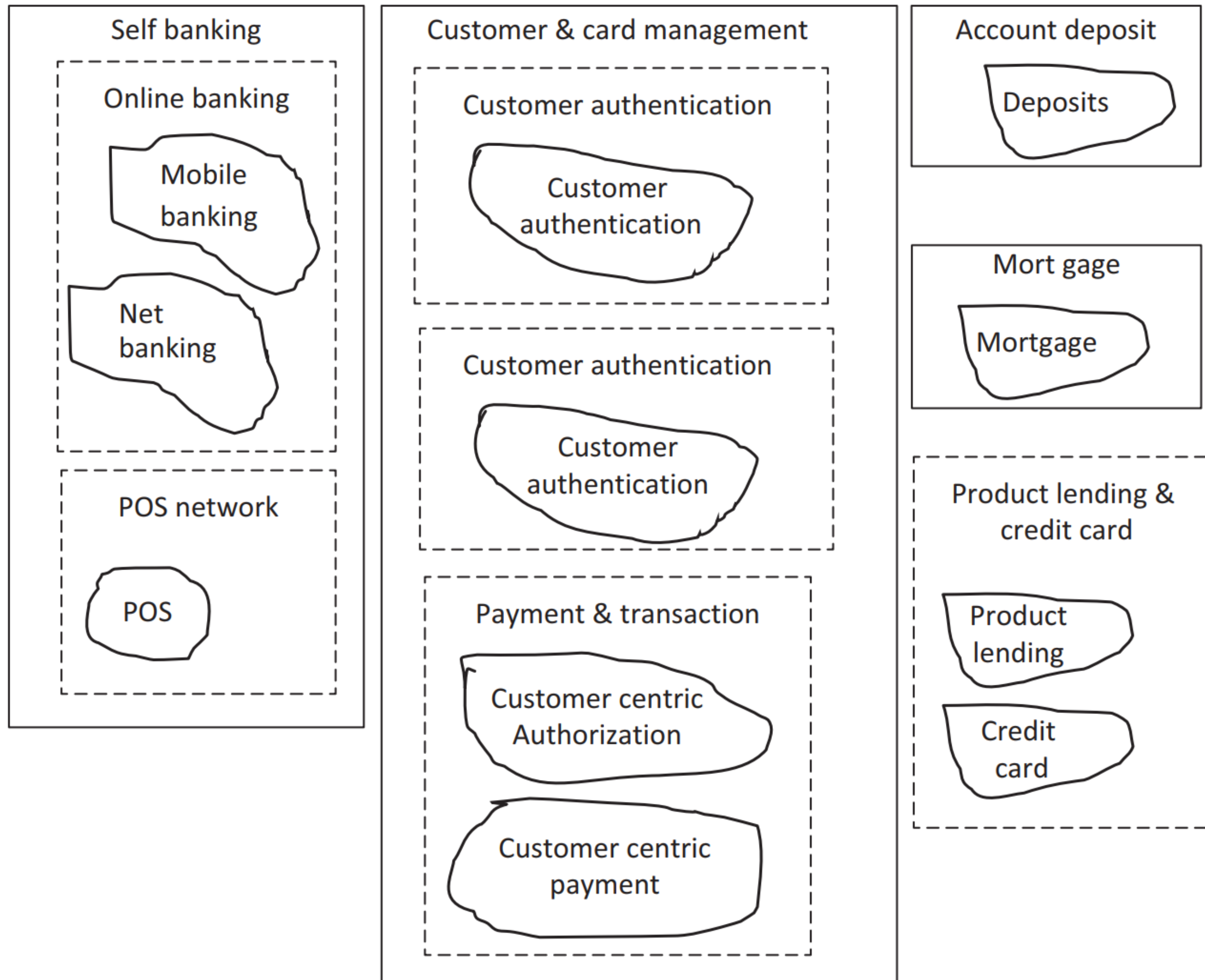
- Відповідно до Domain Driven Design, домен, тобто роздрібний банк, поділяється на субдомени.
- Крім того, згідно із законом Конвея, межі субдоменів частково визначаються комунікаційними структурами або організаційними структурами в організації.
- Таким чином, субдомени для роздрібних банківських операцій ідентифікуються з організаційної структури як
- (i) самообслуговування,
- (ii) управління клієнтами та картками,
- (iii) депозитний рахунок,
- (iv) кредитування та кредитування продуктів та
- (v) іпотека.

- Тепер кожен субдомен має окремий або незалежний обмежений контекст. Для кожного субдомену виділено незалежні функціональні області, як зазначено в таблиці 3.1.
- Враховуйте, що роздрібний банк запроваджує *customer-centric payment services* (платіжні послуги, орієнтовані на клієнтів), для досягнення цілей утримання клієнтів.
- Ця послуга дозволяє покупцеві купувати їхні продукти на основі своїх інвестицій, рахунків, активів та взаємодії з банком, а не просто на основі залишку на рахунку.

Table 3.1 Bounded Contexts in Various Subdomains for a Retail Domain

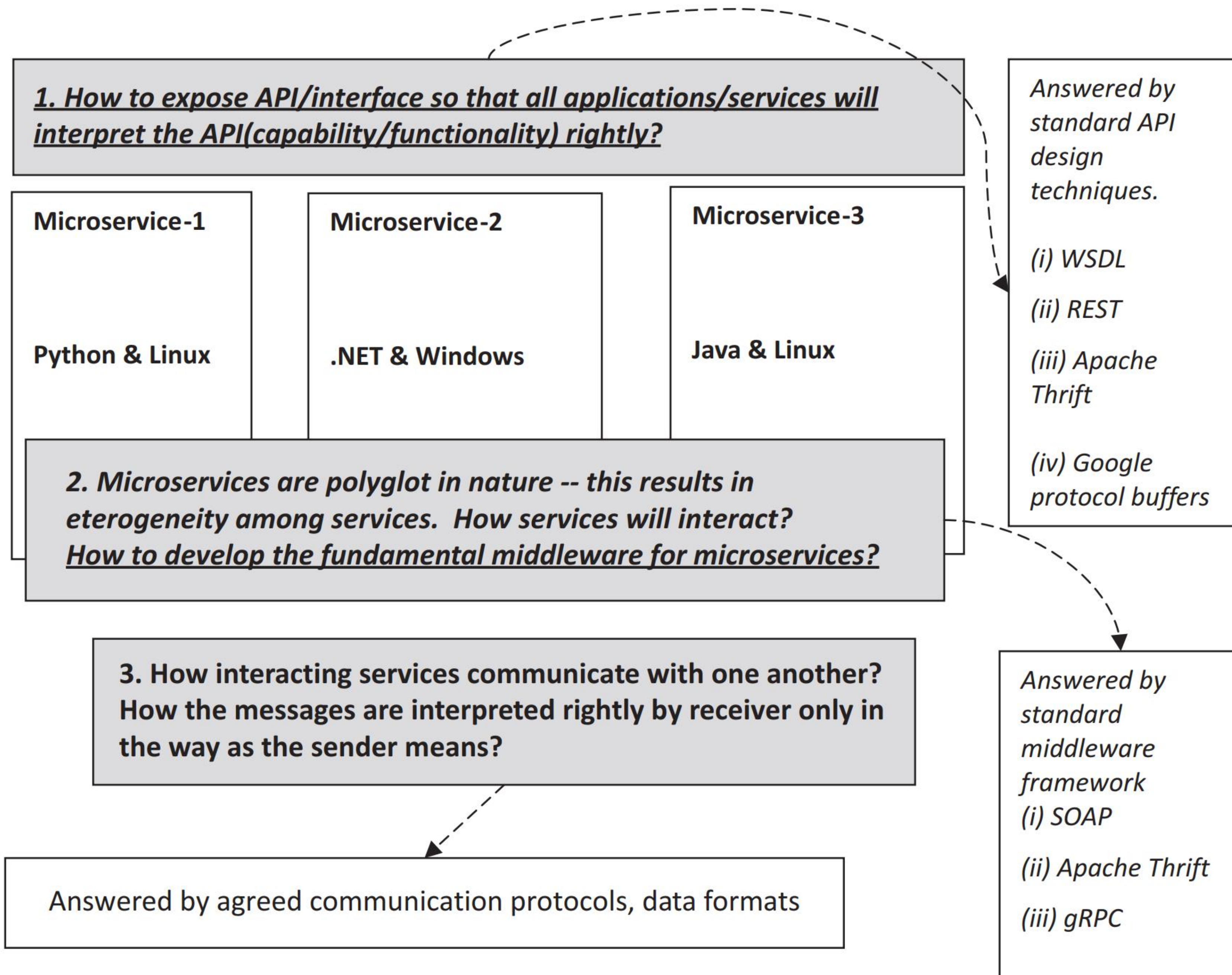
Subdomain	*Bounded Context*
Self-service banking	(i) online banking and (ii) Point of Sale
Customer and card management	(i) customer information, (ii) customer authentication, and (iii) payment and transactions
Deposit account	(i) savings account and (ii) checking account
Product lending and credit	(i) product lending and (ii) credit card
Mortgage	(i) mortgages

- Різні субдомени та обмежений контекст разом із новими *customer-centric payment services*, розміщуються у домені, як показано на Рис. 3.2.



Передумови для Service Interaction/ Communication

- Архітектура мікросервісів служить архітектурою для програм, які вимагають частого розгортання та постійного випуску.
- Ці програми розробляються з використанням гнучкої моделі програмного процесу.
- MSA дає свободу розробникам у виборі технологій, мов програмування та операційних систем, інструментів тощо для розробки мікросервісів; тобто MSA підтримує *polyglot development*.
- Кожен мікросервіс може мати власну мову програмування та платформу, як показано на Рис. 3.3.


1. How to expose API/interface so that all applications/services will interpret the API(capability/functionality) rightly?
Answered by standard API design techniques.
(i) WSDL
(ii) REST
(iii) Apache Thrift
(iv) Google protocol buffers
Microservice-1
Python & Linux
Microservice-2
.NET & Windows
Microservice-3
Java & Linux
2. Microservices are polyglot in nature -- this results in eterogeneity among services. How services will interact?
How to develop the fundamental middleware for microservices?
3. How interacting services communicate with one another? How the messages are interpreted rightly by receiver only in the way as the sender means?
Answered by standard middleware framework
(i) SOAP
(ii) Apache Thrift
(iii) gRPC
Answered by agreed communication protocols, data formats

- На Рис. 3.3 є три служби: Microservice-1 розробляється за допомогою Python та Linux, Microservice-2 розробляється за допомогою .NET та Windows, а Microservice-3 розробляється за допомогою Java та Linux.
- Це ілюструє труднощі в спілкуванні. Природно, що має бути створена платформа проміжного програмного забезпечення, щоб приховати неоднорідність у середовищі мікросервісів. Крім того, повинна існувати стандартизація
- (i) у способі розробки та експонування API та
- (ii) у форматах серіалізації даних.

Коли службам потрібно спілкуватися між собою, виникає низка питань.

- **(i)** Як одна служба дізнається про можливості іншої служби? Як служба надає свої функціональні можливості іншим?

Послуга надає свої можливості за допомогою Application Programming Interface. API описує всі деталі всіх сигнатур методів та їх семантику виклику.

- **(ii)** Як стандартизувати дизайн API, щоб служба інтерпретувала API правильно, так, як це означав / інтерпретував постачальник?

Існує кілька стандартних механізмів або методів розробки API. Це: WSDL, REST, Apache Thrift та Google Protocol Buffers.

- **(iii)** Оскільки в неоднорідному середовищі мови програмування, технології та платформи послуг різняться між собою, стає необхідним встановити проміжне програмне забезпечення, яке повинно приховувати всю неоднорідність.

Чи існує вже проміжне програмне забезпечення, що забезпечує основи?

Тож, щоб розробники могли зосередитись на основному додатку, а не на розробці підтримки проміжного програмного забезпечення для зв’язку мікросервісів?

Існує кілька стандартних комунікаційних платформ для MSA. Це: SOAP, Apache Thrift та gRPC

- **(iv)** Як правильно трактуються дані та повідомлення, надіслані від однієї служби до іншої?

Послуги повинні використовувати узгоджені протоколи та формати даних.

Communication Models for Microservices

- Перше очевидне запитання, яке виникає: Чому службам необхідно взаємодіяти між собою, якщо кожна служба незалежна?

- Ключ: Служби незалежні щодо конкретної бізнес можливості.

Це означає, що мікросервіс не повинен залежати від іншої служби, щоб завершити свої функціональні можливості.

- Наприклад, розглянемо мікросервіс: послугу автентифікації клієнта.

  Функціональність цієї послуги надається наступним чином:

- Візьміть облікові дані користувача, введені замовником, і перевірте вхідні дані у таблиці клієнта. Якщо облікові дані відповідають вимогам, служба затверджує користувача як автентифікованого користувача; в іншому випадку служба відхиляє користувача як неавторизованого користувача. Зараз ця послуга функціонально незалежна.

- Розглянемо ще один сервіс: customer-centric payment service (платіжна послуга, орієнтована на клієнта).

Враховуйте, що клієнт зацікавлений у придбанні товару у торговій точці (Point of Sale). Він проводить пальцем по своїй картці, і запит на покупку надходить до платіжної служби, орієнтованої на клієнта (customer-centric payment service).

Тут customer-centric payment service, повинен взаємодіяти з іншими сервісами,

- *user authentication service* та
- *customer-centric payment authorization service*.

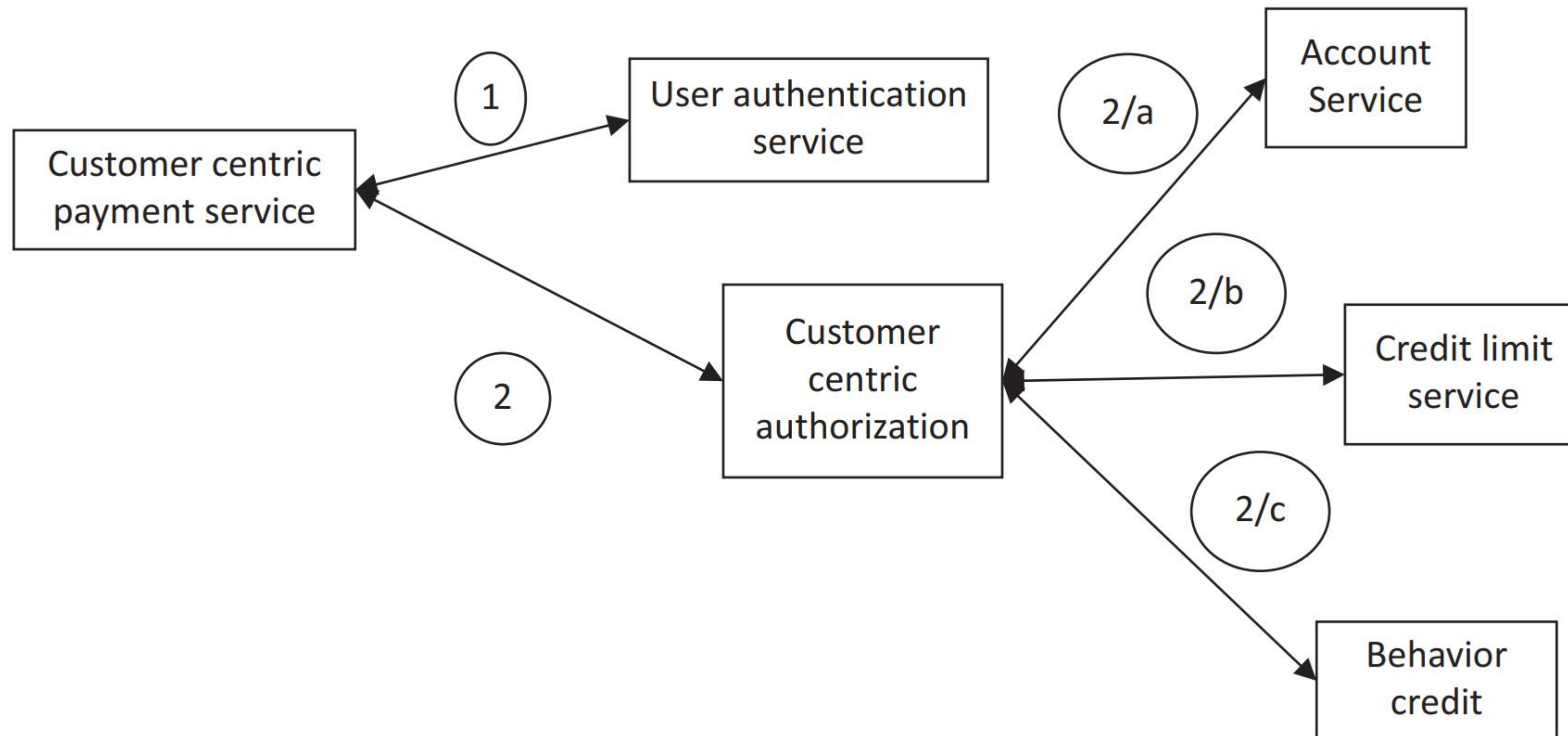
У цьому прикладі *customer-centric authorization service*, дозволяє здійснювати покупку, перевіряючи залишок на рахунку, його кредитні ліміти та взаємодію з банком. Врахуйте, що служба виконує перевірку наступним чином.

- Якщо

product price <=account balance +credit card limit+ credit gained by a user by his behavior,

дозвольте придбання, інакше забороніть покупку.

- Після того, як *customer-centric payment service*, отримає необхідну інформацію та схвалення від служби *customer payment authorization service*, вона продовжує приймати платіж від клієнта. Взаємодія між різними службами наведена на Рис. 3.4.



- У наведеному вище прикладі, після вибору продукту, користувач проводить картку та вводить PIN-код.
- Тепер його запит надходить до *customer-centric payment service* (припустимо, що користувач вже встановив свої уподобання в банку, і, отже, банк схвалив його орієнтовану на клієнта оплату, оскільки він є довіреним користувачем із хорошим кредитом). Цей сервіс виконує наступне:
- (i) Перевірка автентифікації користувача.
- (ii) Якщо він знаходить користувача як автентифікованого користувача, він викликає службу customer-centric authorization service. Ця послуга, в свою чергу отримує доступ до трьох різних служб, а саме: обслуговування рахунків, обслуговування кредитних лімітів та обслуговування поведінки.

Synchronous Communication

- В основному, існує два типи стилів спілкування мікросенвісів, а саме, синхронне спілкування та асинхронне спілкування, як показано на Рис. 3.5.
- При синхронному спілкуванні відправник та одержувач повинні бути зв’язані між собою під час зв’язку.
- Відправник (або клієнт) відправляє запит одержувачу і переходить у заблокований стан або чекає, поки не отримає відповідь від одержувача (або сервера).
- У MSA одна служба може викликати іншу службу для отримання інформації, і служба, що викликає, повинна почекати, поки не отримає відповідь від іншої служби. Це показано на Рис 3.6. Відправник та одержувач взаємодіють між собою в point-to-point стилі.

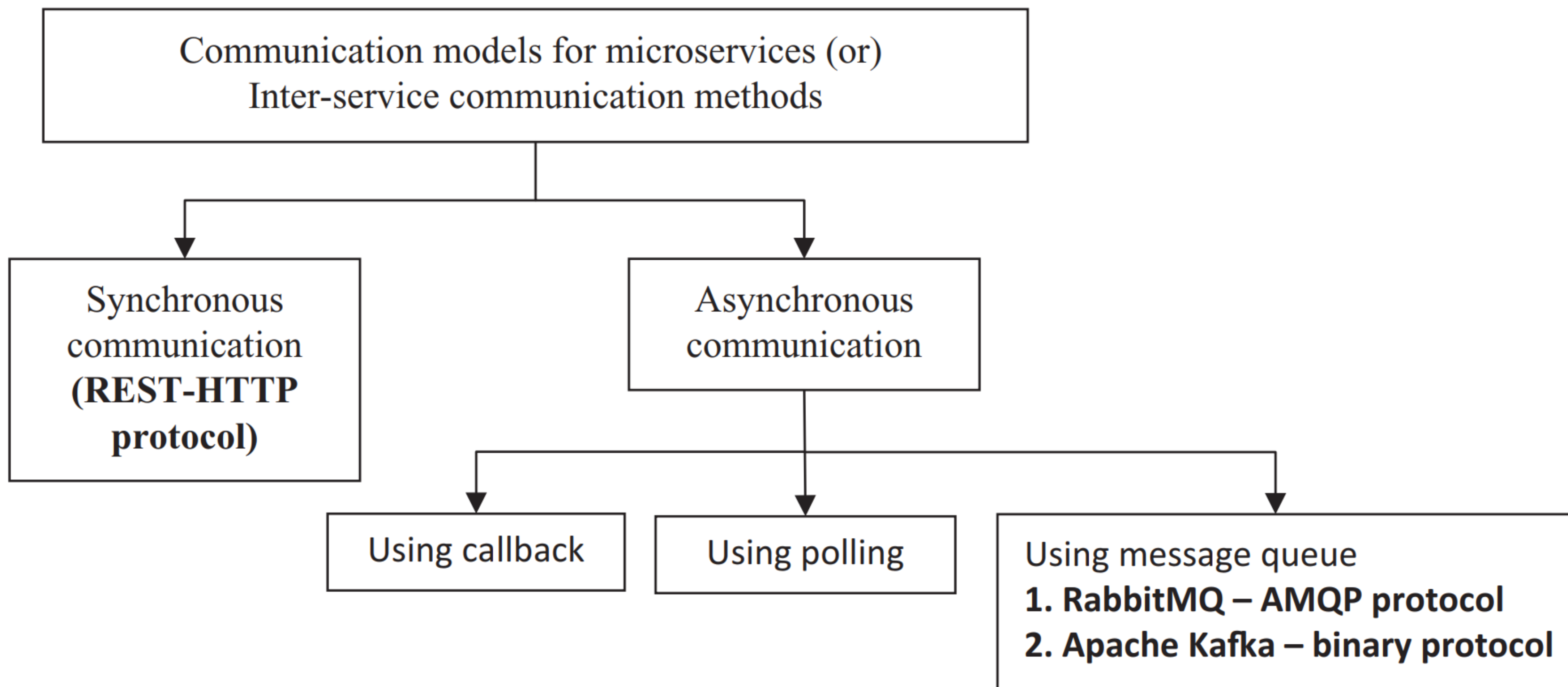


Figure 3.5 Communication models for microservices.

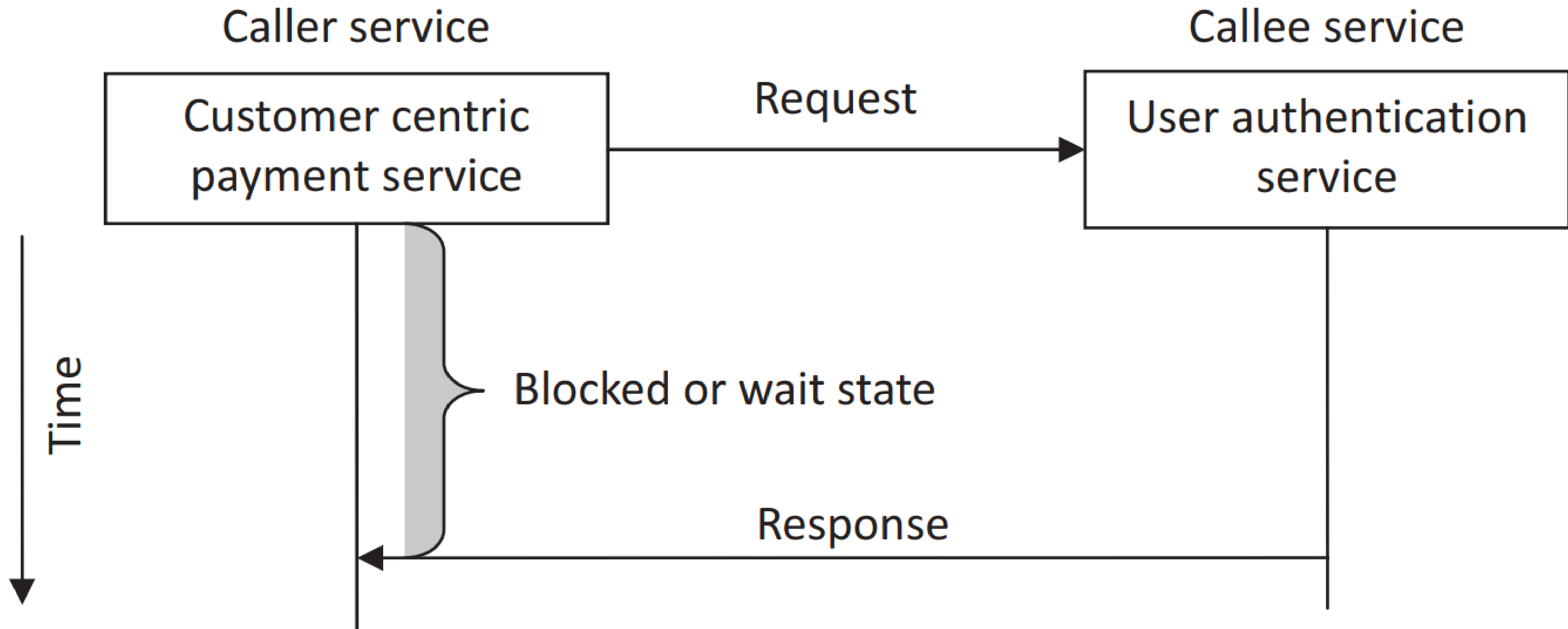


Figure 3.6 Synchronous communication.

- Мікросервіси використовують архітектурний стиль REST, який неявно використовує протокол HTTP для синхронного зв'язку між собою.
- HTTP - це синхронний протокол, в якому клієнт надсилає запит і чекає відповіді від сервера.
- Клієнт може продовжувати свою роботу лише за умови отримання відповіді від сервера.

Representational State Transfer (REST) Architecture

- Архітектура мікросервісів зародилася у 2014 році. Спочатку додатки на базі мікросервісів розроблялися з використанням архітектурного стилю Representational State Transfer (REST), який був розроблений Роєм Філдінгом у 2000 році. як засіб спілкування.
- REST в основному розглядає мікропослуги як подібні до веб / Інтернет-ресурсів. тобто, подібно до того, як веб-ресурси ідентифікуються та розміщуються за допомогою URL-адреси у Всесвітній павутині, REST розглядає мікросервіси як веб-ресурси, а також ідентифікує та знаходить мікросервіси за допомогою URL-адреси. Навіть сам Інтернет є прикладом реалізації архітектури REST. Архітектура REST складається з трьох компонентів, як показано на Рис. 3.7.

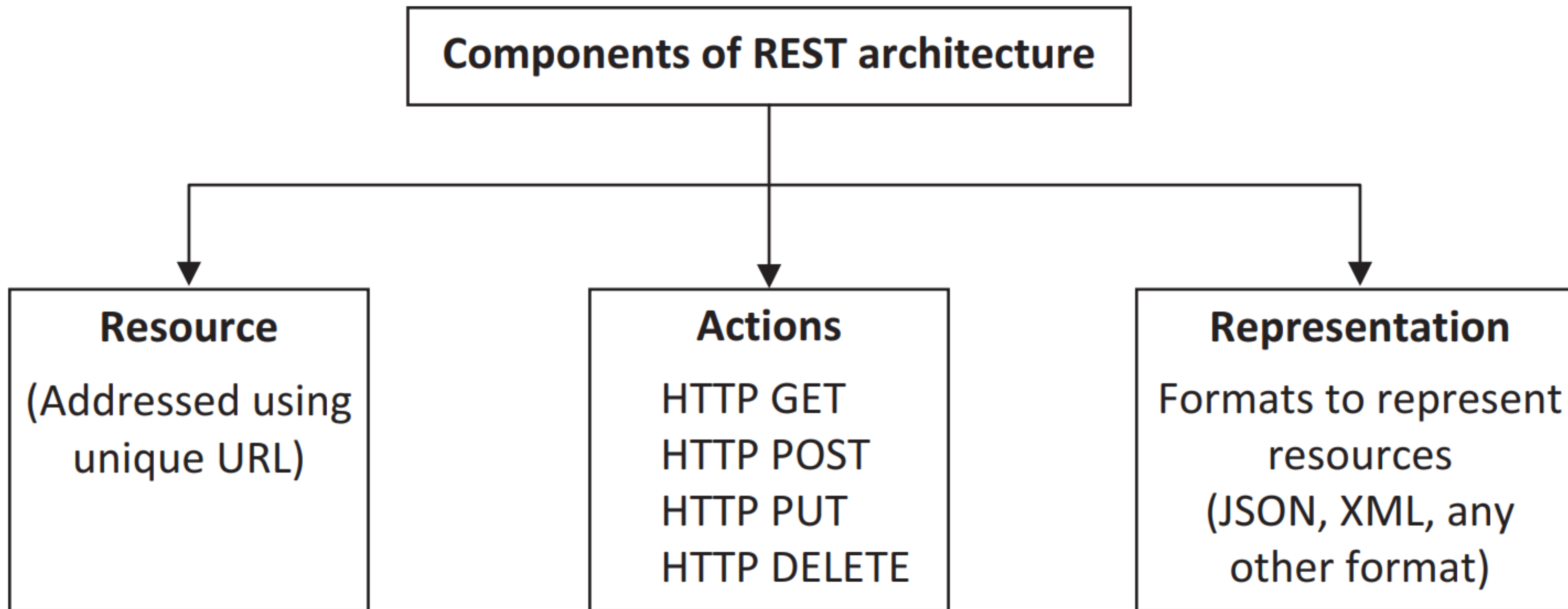


Figure 3.7 Components of the REST architecture.

Як два мікросервіси, service-A та service-B, взаємодіють за допомогою запиту REST-HTTP GET, показано на Рис. 3.9.

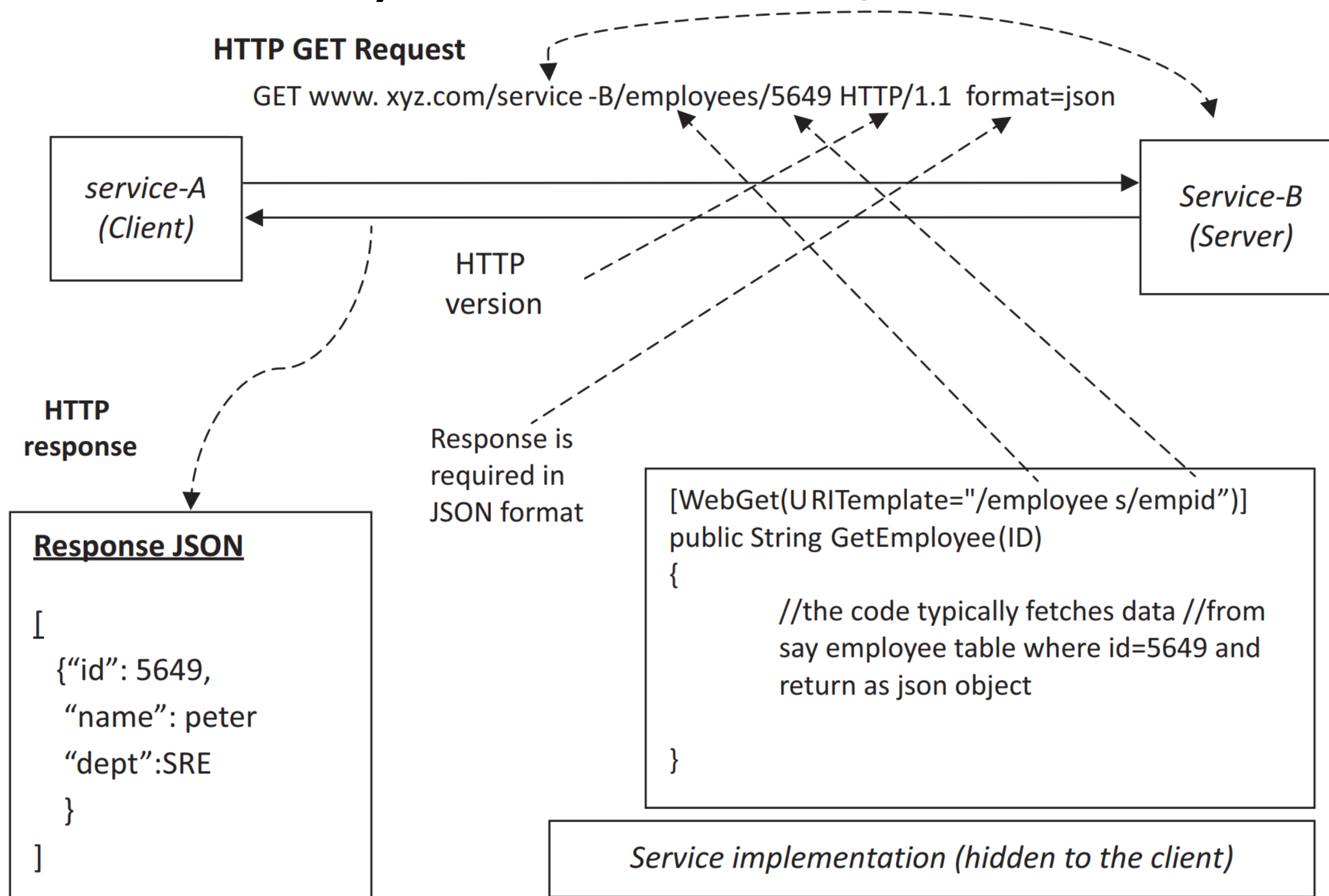


Figure 3.9 Typical REST GET request and response.

Чому в мікросервісах не заохочується синхронне спілкування?

- Розглянемо простий синхронний зв’язок між чотирма службами, скажімо, serviceA, service-B, service-C та service-D, як показано на Рис. 3.13.
- Як на Рис. 3.13, service-A робить запит до service-B і чекає, поки не отримає відповідь від service-B. Служба service-B надсилає запит до service-C, яка, в свою чергу, робить запит до service-D.
- У кожній взаємодії абонент повинен чекати, поки caller надішле відповідь. Через стан блокування або очікування синхронного зв'язку (як показано на Рис. 3.13), цикли запит-відповідь стануть довгими. Продуктивність програми стане дуже низькою.

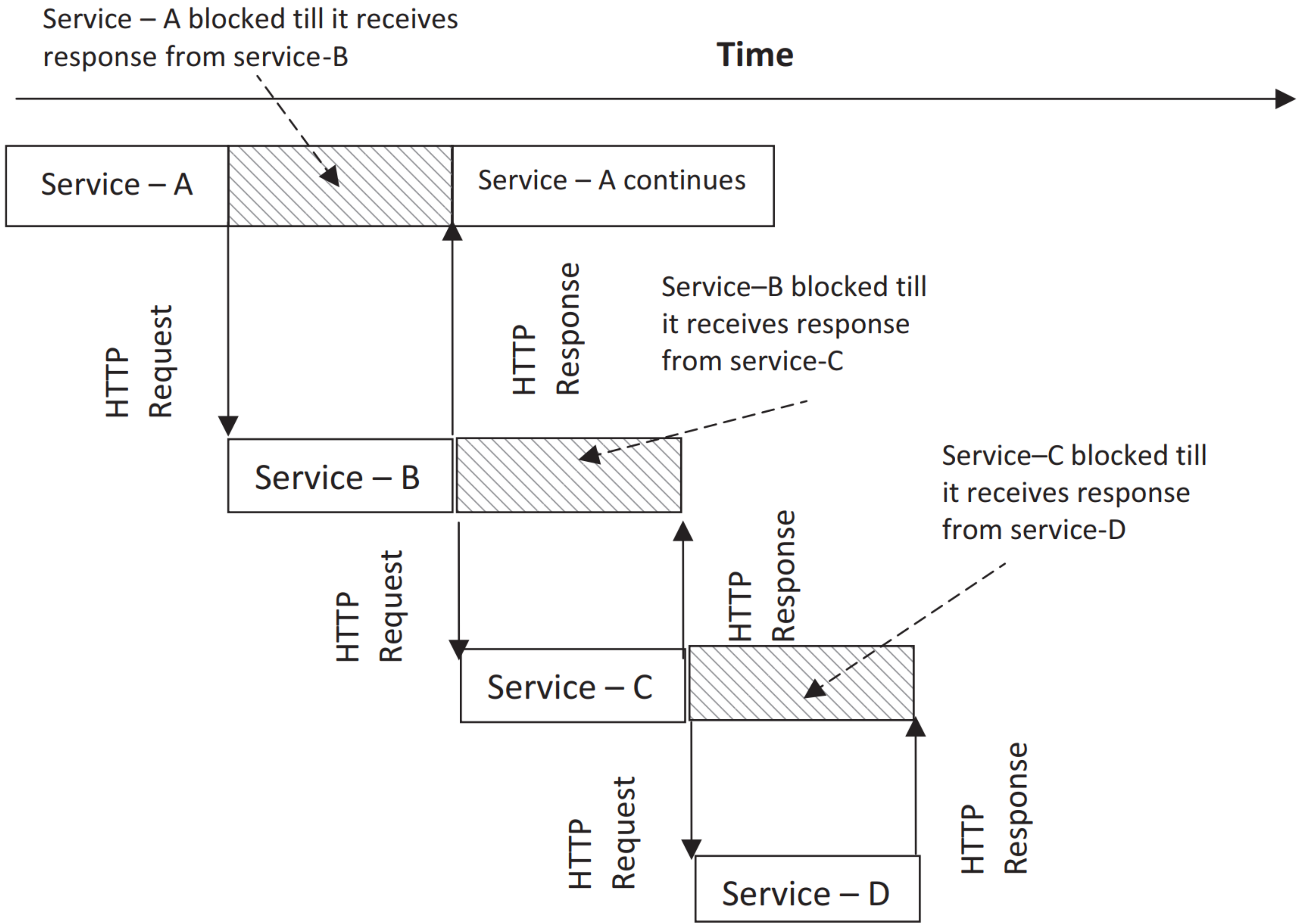


Figure 3.13 Blocked state in synchronous communication.

Asynchronous Communication

При асинхронному спілкуванні клієнт (caller) надсилає запит на сервер (callee) і не чекає відповіді від сервера.

Існує три способи отримати відповідь від сервера:

- (i) за допомогою функції зворотного виклику,
- (ii) за допомогою методу опитування та
- (iii) за допомогою посередника повідомлень.

(i) Using the Callback Method

- У цьому методі клієнт (caller) надсилає запит на сервер. Відправляючи запит на сервер, клієнт також надсилає посилання на функцію зворотного виклику (функція зворотного виклику - це функція в клієнті), яку буде викликати сервер після обробки запиту.
- Отже, як тільки запит обробиться, сервер викличе функцію зворотного виклику в клієнті. За допомогою цього виклику функції зворотного виклику клієнт розуміє, що відповідь готова, і він піде на подальшу обробку. Це показано на Рис. 3.14.

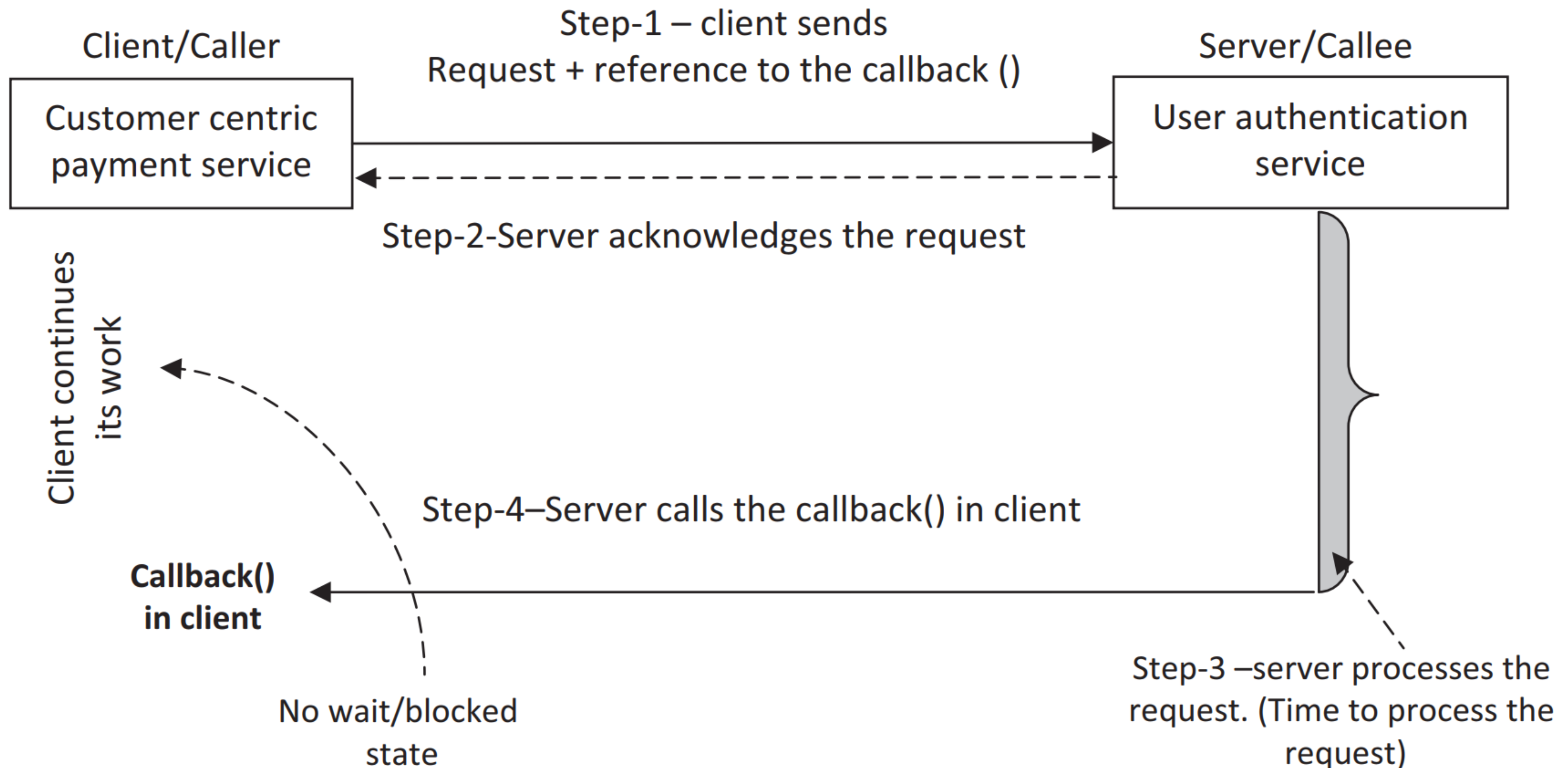


Figure 3.14 Asynchronous communication using the callback() function.

(ii) Using the Polling Method

- Асинхронний зв'язок із використанням методу опитування показаний на Рис. 3.15.
- У методі опитування клієнт надсилає запит на сервер. Сервер інформує клієнта про отримання запиту. Клієнт продовжує свою роботу. Крім того, клієнт опитує сервер щодо статусу запиту через рівні проміжки часу, поки сервер не завершить запит.

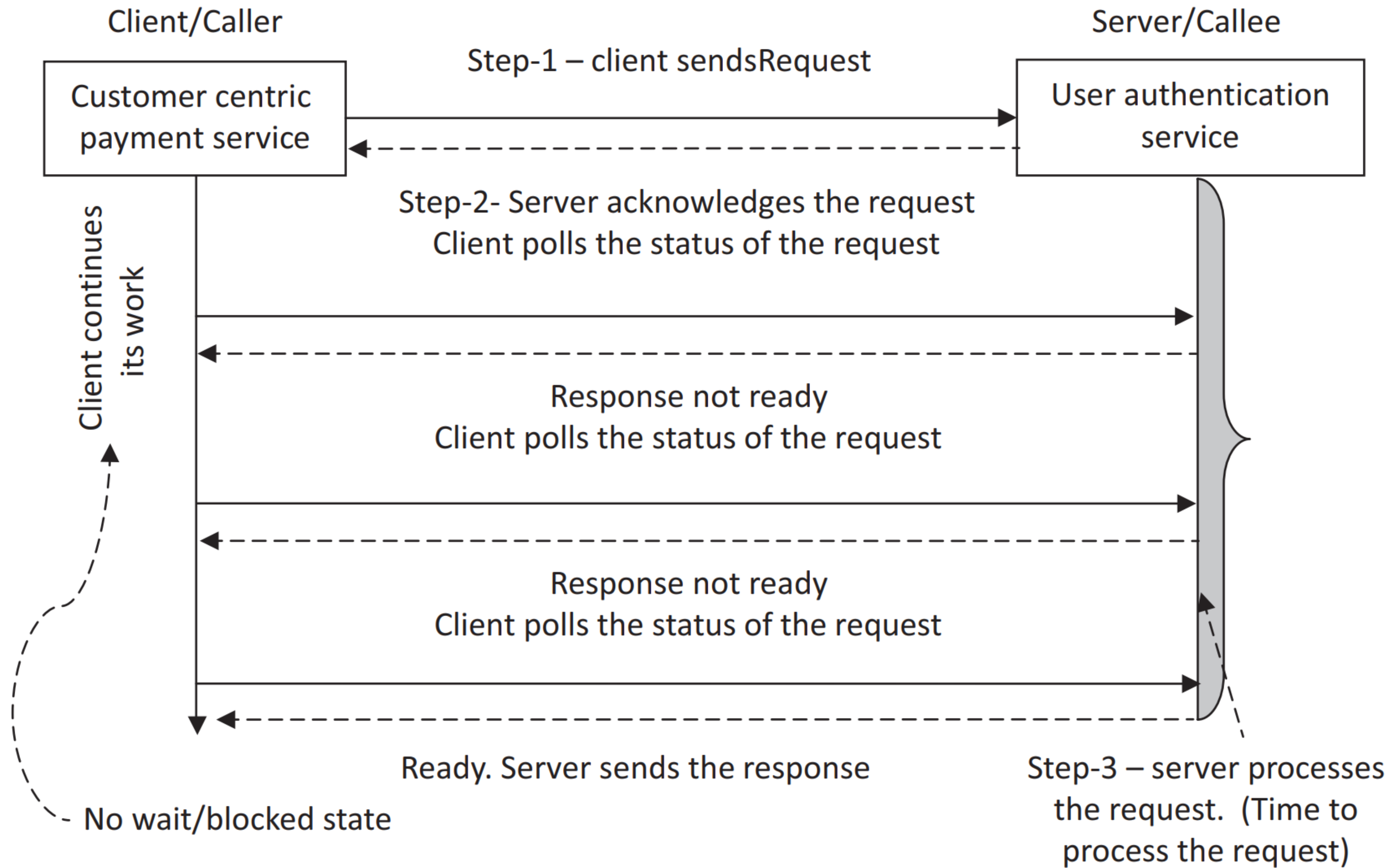


Figure 3.15 Asynchronous communication using polling.

(iii) Using Message Queue

- У цьому методі відправник надсилає повідомлення в чергу, а одержувач приймає повідомлення з черги, як показано на Рис. 3.16.
- У методі черги повідомлень відправник та одержувач повідомлення не пов’язані безпосередньо один з одним.
- Відправник та одержувач з'єднані через чергу повідомлень. Їм не потрібно одночасно підключатися до черги. Відправник відправляє повідомлення в чергу, а черга зберігає повідомлення, поки одержувач не отримає повідомлення.
- Основна перевага черги повідомлень полягає в тому, що вона забезпечує високий ступінь роз'єднання між відправником і одержувачем. Це дозволяє нам досягти високої продуктивності, масштабованості тощо.

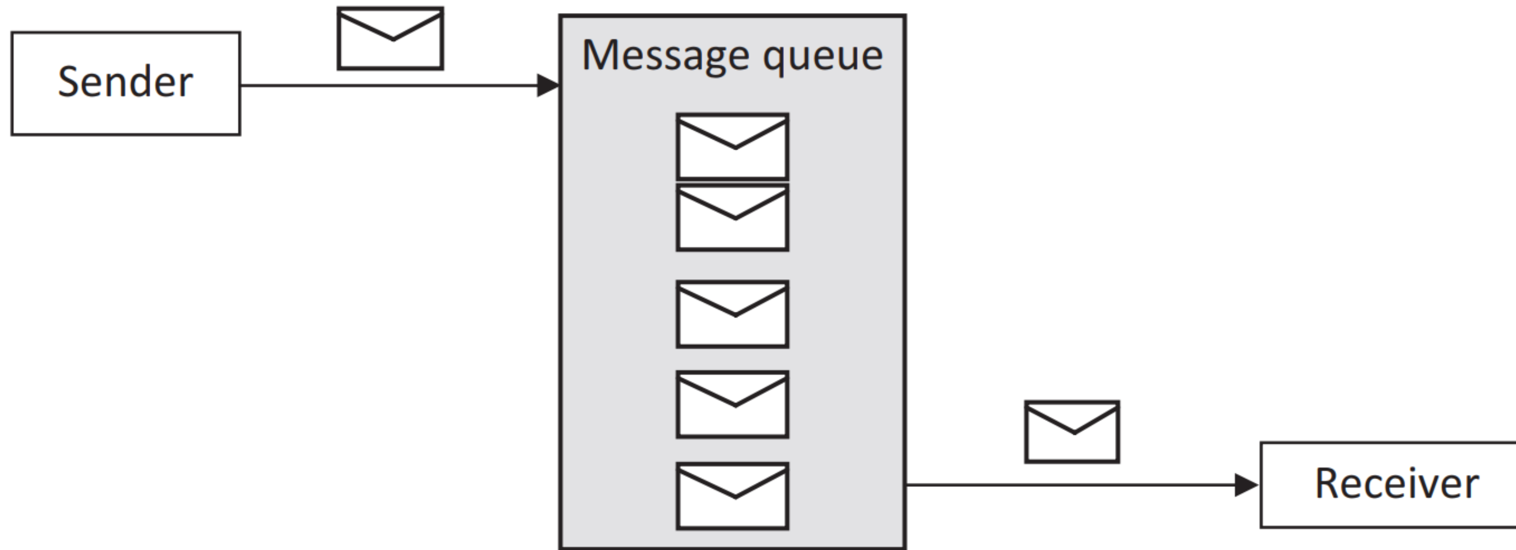


Figure 3.16 Asynchronous communication using a message queue.

Традиційно черги повідомлень використовують власні приватні протоколи, які не забезпечують міжплатформну та міжмовну підтримку. Отже, сьогодні черги повідомлень реалізуються за допомогою стандартних протоколів.

До загальновживаних відкритих протоколів для черги повідомлень належать (i) **AMQP**, (ii) Simple/Streaming Text Oriented Messaging Protocol (**STOMP**) та (iii) Message Queuing Telemetry Transport (**MQTT**).

- **Advanced Message Queuing Protocol (AMQP)**— Протокол розширених черг повідомлень - це проміжне програмне забезпечення, орієнтоване на повідомлення з відкритим кодом, яке забезпечує асинхронні повідомлення, включаючи надійні черги, повідомлення на основі публікації та підписки на основі тем, гнучку маршрутизацію, транзакції та безпеку. AMQP забезпечує надійність, масштабованість та сумісність між різними постачальниками.
- **Simple/Streaming Text Oriented Messaging Protocol** — STOMP - це простий текстовий протокол, що використовується для передачі даних між програмами, розробленими різними мовами та платформами. Він не використовує поняття черг і тем, але посилає семантику з цільовим рядком. Цей протокол може бути використаний, коли потрібен простий додаток для чергування повідомлень без потреби у чергах.

- **Message Queuing Telemetry Transport** — MQTT надає повідомлення для публікації та передплати (без черг). Його невеликий розмір робить його придатним для вбудованих систем, програм на базі Інтернету речей (IoT) тощо. Наприклад, брокери на базі MQTT можуть підтримувати багато тисяч одночасних з'єднань між пристроями. MQTT має компактний двійковий пакет корисного навантаження, що робить його придатним для таких додатків, як оновлення датчиків, оновлення біржових цін, акції, мобільні повідомлення тощо.

Існує багато інструментів черги повідомлень, таких *як IBM MQ, Java Message Service (JMS), Apache ActiveMQ, Apache Kafka, Apache RocketMQ, Sun Open Message Queue, Amazon Simple Queue Service*, обмін повідомленнями *JBoss* тощо.

У MSA асинхронна комунікація зазвичай реалізується за допомогою двох популярних посередників повідомлень з відкритим кодом, а саме:

- (i) Rabbit MQ
- (ii) Apache Kafka

Ці два брокери повідомлень докладно описані в наступних розділах.

Asynchronous Communication with Rabbit MQ Message Broker

- Брокери повідомлень використовуються як <u>проміжне програмне забезпечення між мікросервісами</u>, де мікросервіс може надсилати / доставляти (публікувати) свої повідомлення брокеру та іншим мікросервісами, які можуть підключатися до брокера та отримувати / передплачувати повідомлення.
- <u>Основною метою посередника повідомлень є забезпечення роз'єднання між виробником повідомлення та споживачем повідомлення</u>. Отже, у випадку з брокером виробник повідомлень звільняється від аспектів, пов’язаних із спілкуванням, і зосереджується на основній роботі. Брокер відповідає за надійну доставку повідомлення до одержувача.

- *Java Message Service* - це найпопулярніший асинхронний API Java. Але обмеження JMS полягає в тому, що він підтримує лише мову Java, і, таким чином, стає недостатнім для спілкування з додатками, що не є Java.
- Інший популярний і традиційний брокер повідомлень - *RabbitMQ*. *RabbitMQ* був розроблений компанією Rabbit Technologies Ltd, яку придбало SpringSource, частина VMware. Проект RabbitMQ став частиною програмного забезпечення Pivotal і був випущений в 2013 році.
- *Початковою метою RabbitMQ було забезпечити взаємодіючого брокера повідомлень. Він забезпечує підтримку різних мов та операційних систем. На відміну від попередніх черг повідомлень, таких як IBM qieue або JMS або Microsoft Message Queue, RabbitMQ є нейтральним до платформи, мовно нейтральним брокером повідомлень з відкритим кодом*.

- RabbitMQ першим застосував специфікацію AMQP. Специфікація AMQP містить два аспекти, як показано на Рис. 3.17:

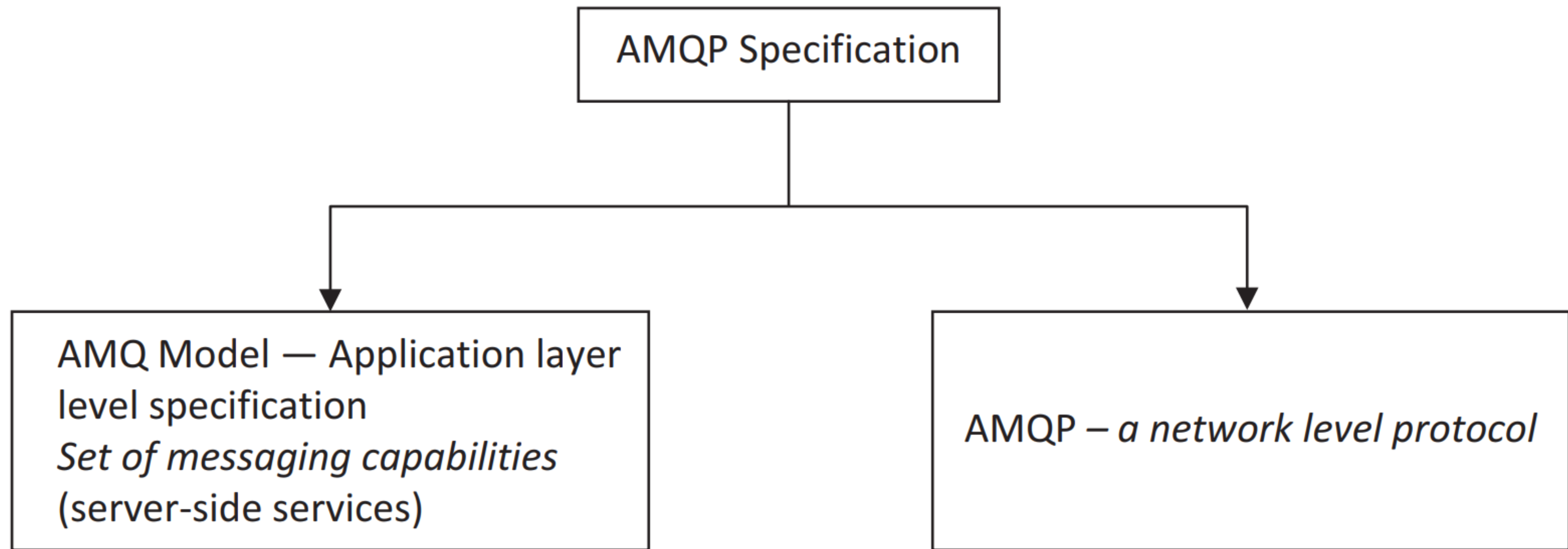


Figure 3.17 Parts of AMQP Specification.

1. Набір можливостей обміну повідомленнями (рівень прикладного рівня, серверні послуги), що називається моделлю AMQ. Модель AMQ складається з набору компонентів, які спрямовують та зберігають повідомлення на сервері, та набору правил, які пов'язують ці компоненти разом. Точно ця частина специфікації надає архітектуру вищого рівня для брокерів повідомлень.

2. Протокол мережевого рівня під назвою AMQP за допомогою клієнтської програми може взаємодіяти з брокером повідомлень.

AMQ Model

Основні компоненти моделі AMQ показані на Рис. 3.18.

Основними компонентами моделі AMQ є

- (i) повідомлення,
- (ii) виробники повідомлень,
- (iii) споживачі повідомлень та
- (iv) брокери / сервери повідомлень.

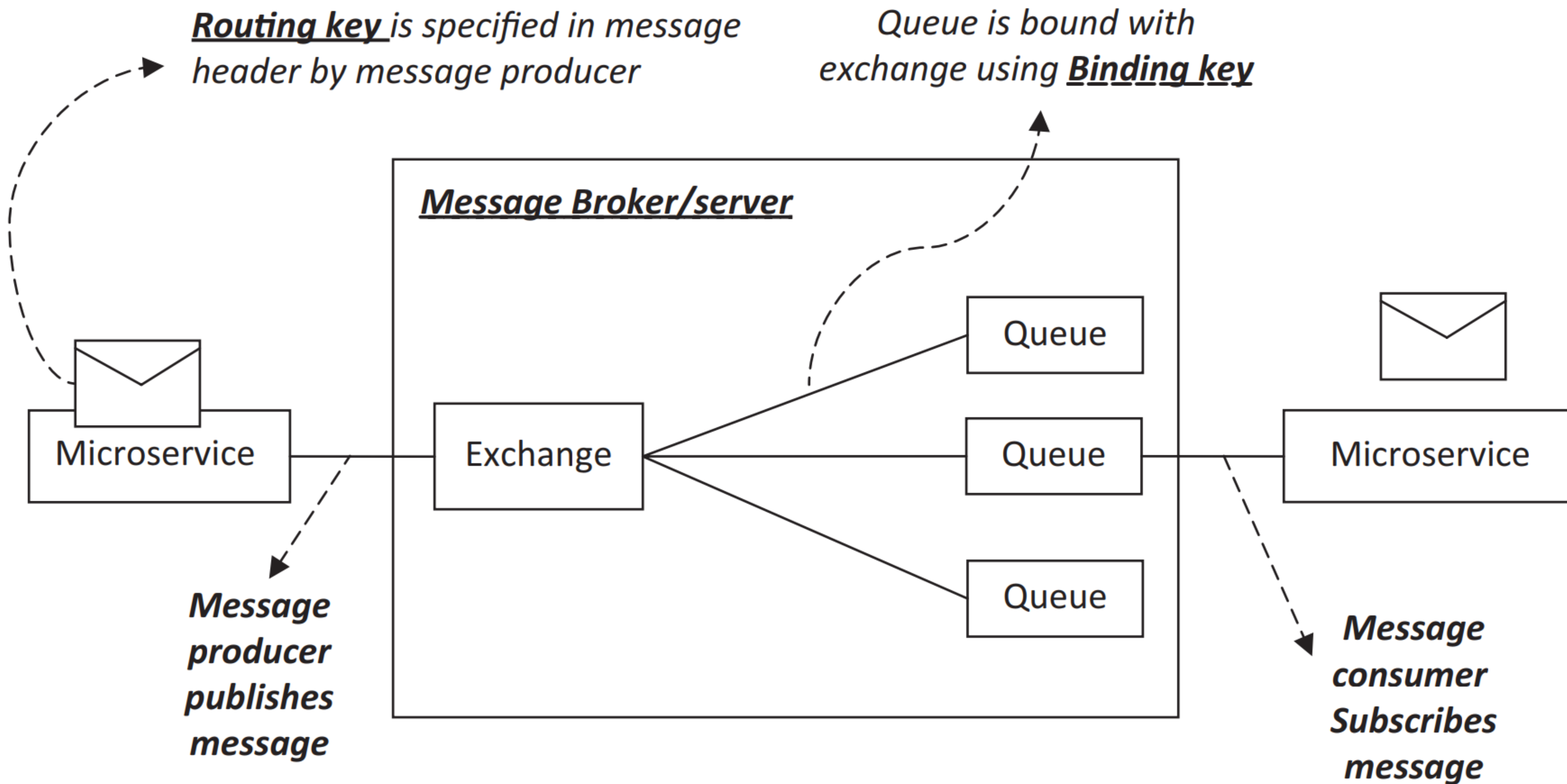


Figure 3.18 The AMQ model.

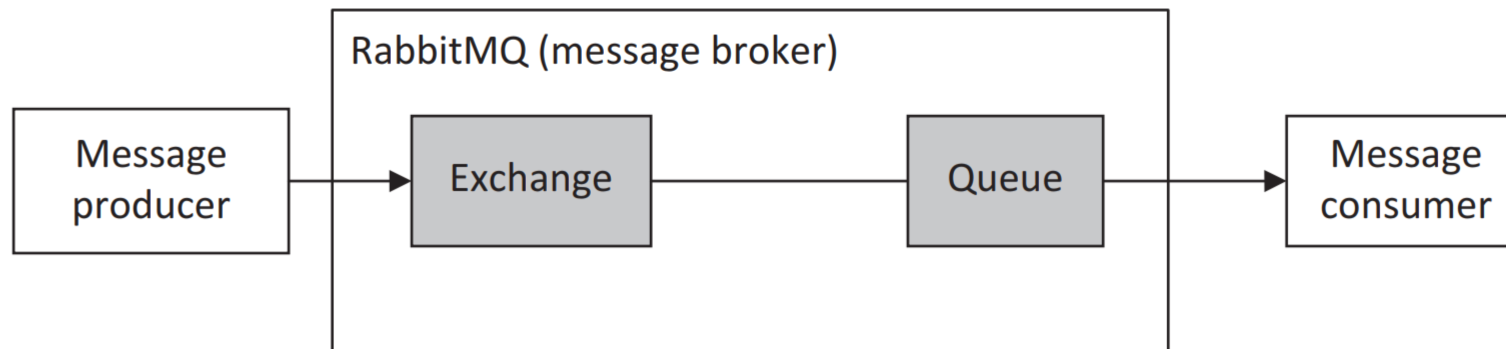
- **Messages** — Повідомлення - це інформаційний пакет, що складається з двох частин, а саме заголовка та тіла.
- **Message producers** — Виробники повідомлень - це програми, які створюють повідомлення та надсилають їх брокеру. Виробник повідомлень публікує свої повідомлення за допомогою так званого ключа маршрутизації (routing key), який використовується компонентом обміну для направлення повідомлення зацікавленому споживачеві. Ключ маршрутизації - це атрибут повідомлення. Ключ маршрутизації дає віртуальну адресу, на яку обмін направляє повідомлення.
- **Message consumers** — Споживачі повідомлень - це програми, які підключаються до черги брокера та підписуються або отримують їх повідомлення.

Message Brokers/Servers

Сервери / брокери повідомлень дозволяють повідомленням, опублікованим виробником, асинхронно дістатись до споживачів. Брокер має дві ключові складові:

1. Exchanges (Біржі)
2. Queues (Черги)

Розглянемо посередника повідомлень RabbitMQ. Це посередник повідомлень з відкритим кодом, який реалізує специфікацію AMQP.



Exchange

- Біржа виконує функцію маршрутизації повідомлень. Повідомлення публікуються на біржі всередині брокера. Потім біржа розподіляє копії цих повідомлень у черги згідно з певними правилами, визначеними розробником, і ці правила називаються прив'язками (bindings).
- Прив’язка - це відношення між біржею та чергою. Кожна черга пов'язана обміном через binding key (прив'язувальний ключ - див. Рис. 3.18).

Queues

- Черга - це іменована сутність, яка зберігає повідомлення в пам'яті або на диску (які були направлені біржею) і доставляє повідомлення одному або кільком споживачам.
- Черги - це двосторонні компоненти.
- Вхідна сторона черги отримує повідомлення з однієї або декількох бірж, тоді як вихідна сторона черги підключена до одного або декількох споживачів.
- Кожне повідомлення зберігатиметься в черзі, доки його не отримає споживач повідомлення. Кожна черга повідомлень повністю незалежна і є досить розумним об'єктом.

Черга має різні властивості, такі як

- private/shared,
- durable/temporary,
- server-named/client-named тощо.

Ці властивості корисні для реалізації черги з різними функціональними характеристиками.

Наприклад, приватна черга призначена для доставки повідомлення певному або одному споживачеві, тоді як загальна черга доставляє повідомлення кільком споживачам. Подібним чином, черга передплати - це черга з іменем сервера, яка підключена до одного споживача.

Розподіл повідомлень у RabbitMQ / Типи обміну

- Виробник повідомлень публікує повідомлення на названій біржі (named exchange), а споживач витягує повідомлення з черги, яка пов’язана з біржею. Тут розуміють, що споживач повинен створити чергу і приєднати її до необхідної біржі. Також мається на увазі, що споживач повинен знати назву біржі.
- Може бути кілька бірж на чергу, або кілька черг на біржу, або одна черга на біржу (з відображенням один на один). Тепер виникає питання: Як відбувається розповсюдження повідомлень? Це залежить від типу обміну. Залежно від типу обміну, обмін направляє повідомлення до відповідної черги.

Існують різні типи обмінів залежно від того, як черги зв’язані з обміном, і як обмін відповідає прив’язці черги до атрибутів повідомлення, наприклад, routing key або header values. Вони є:

1. Прямий обмін (Direct exchange)
2. Обмін вентилятором (Fan out exchange)
3. Обмін темами (Topic exchange)
4. Обмін заголовками (Headers exchange)

1. ***Direct exchange*** — Прямий обмін направляє повідомлення по збігу ключа маршрутизації (routing key) з назвою черги. Це показано на Рис. 3.20.

Як показано на Рис. 3.20, ключ маршрутизації, що містить значення “Queue-1”, відповідає імені черги (“Queue-1”). Цей тип обміну направляє повідомлення для зв'язку точка-точка. У цьому типі обміну символи підстановки, такі як "*" або "#", не підтримуються.

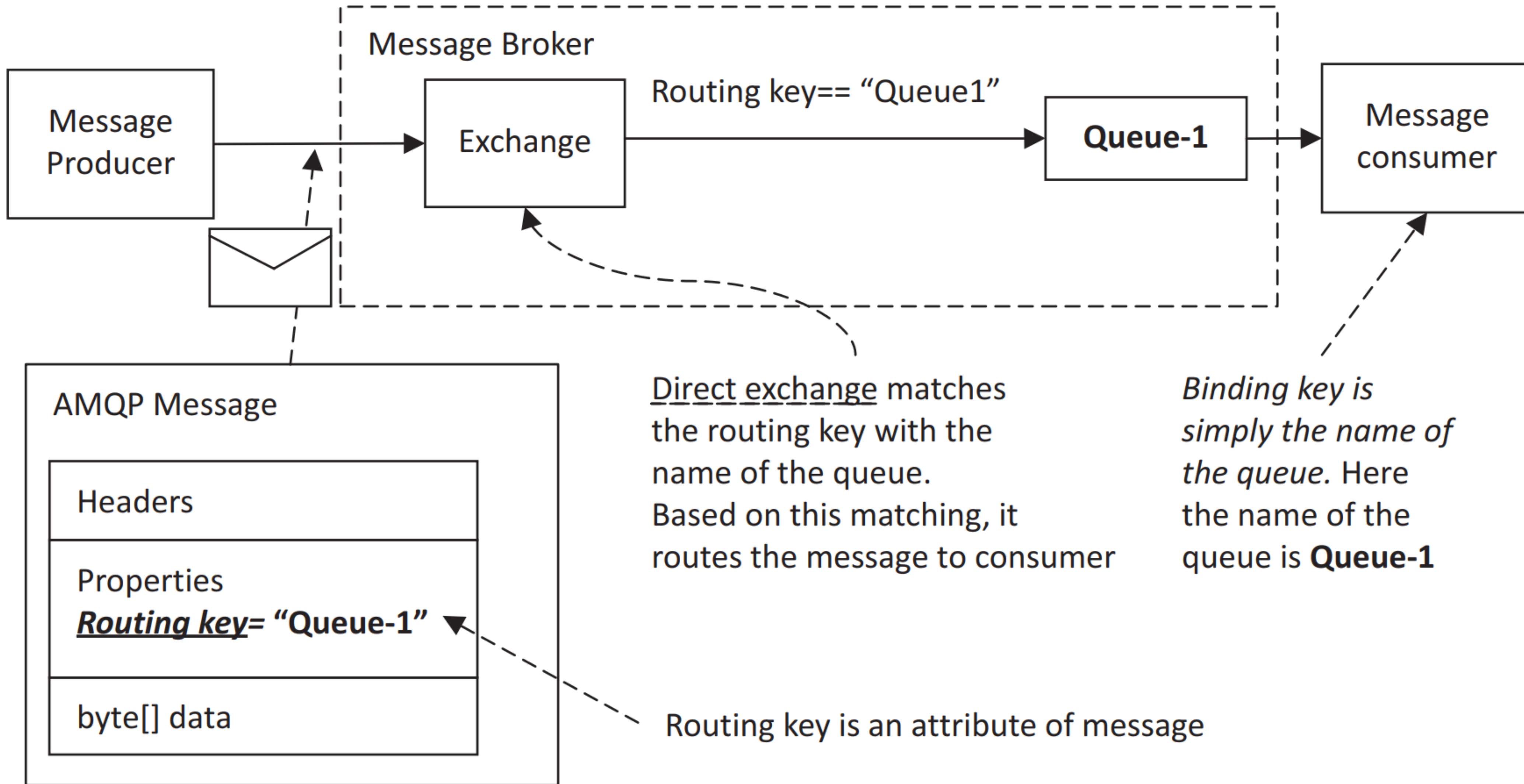


Figure 3.20 Direct Exchange.

2. Fan out exchange — При обміні вентилятором повідомлення копіюється та надсилається у всі черги, підключені до цієї біржі. Поняття обміну вентилятором показано на Рис. 3.21.

У цьому обміні ключ маршрутизації ігнорується. Коли нове повідомлення публікується на біржі вентилятором, воно надсилається у всі черги, прив'язані до біржі. Ця біржа більше підходить для трансляції повідомлення для всіх зацікавлених споживачів.

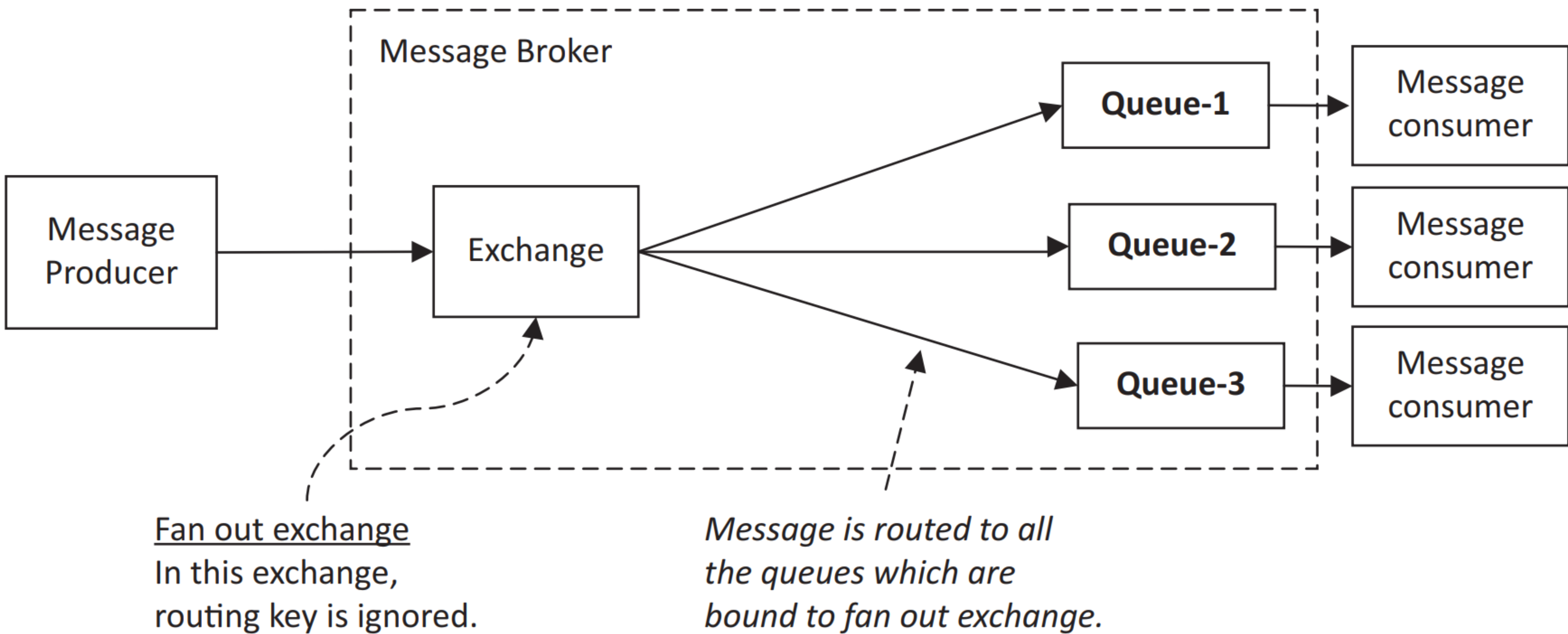


Figure 3.21 Fan out exchange.

3. Topic exchange — Розподіл повідомлень при обміні темами показано на Рис. 3.22. При обміні темами повідомлення направляється до деяких черг шляхом зіставлення ключа маршрутизації із шаблонами ключів прив'язки, представлених із використанням символів підстановки.

Цей обмін, в основному, забезпечує шаблон публікації-передплати повідомлень, за допомогою якого видавці (виробники повідомлень) класифікують свої повідомлення на різні класи (логічні канали, що називаються темами) без відома абонентів (споживачів повідомлень).

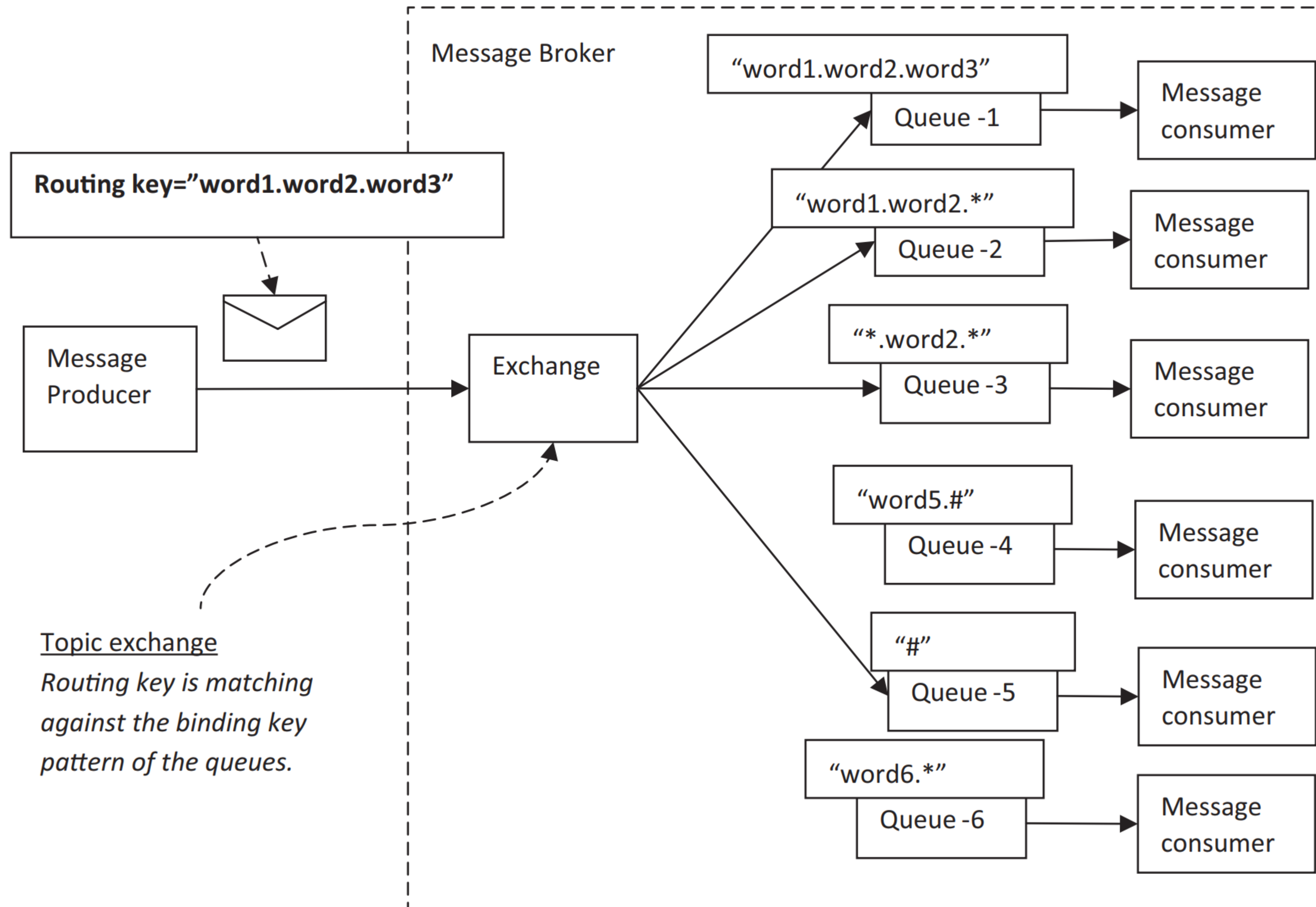


Figure 3.22 Topic exchange.

- При обміні темами публікується повідомлення з ключем маршрутизації, що містить ряд слів, розділених крапкою, скажімо, наприклад, ключем маршрутизації є «word1.word2.word3».
- Черги, прив’язані до обміну темами, забезпечують сервер відповідним шаблоном для використання під час маршрутизації повідомлення. Шаблони можуть містити зірочку «*», щоб відповідати слову в конкретному положенні ключа маршрутизації, або хеш «#», щоб відповідати нулю або більше слів. Як і на Рис. 3.22, оскільки ключі прив'язки Queue-1, Queue-2, Queue-3 і Queue-5 знайшли збіг з ключем маршрутизації, повідомлення пересилається до цих черг, тоді як між маршрутизацією немає відповідності шаблон ключа та прив'язки Queue4 та Queue6. Отже, повідомлення не пересилається в Queue-4 та Queue-6.

4. Headers exchange — Розподіл повідомлень при обміні заголовками показано на Рис. 3.23. При обміні заголовками замість відповідності ключа маршрутизації ключу прив’язки один або кілька заголовків повідомлення узгоджуються із очікуваними заголовками, визначеними чергою.

При обміні темами існує лише один критерій, а саме ключ маршрутизації відповідає ключу прив'язки. Але при обміні заголовками більше одного критерію вказується як пари ключ-значення в заголовку повідомлення та узгоджується з парами значень ключа, вказаними в черзі. При обміні заголовками обмежена черга вказує, чи слід поєднувати всі пари ключ-значення із заголовками (тобто через параметр збігу “всі”), чи потрібно зіставляти будь-яку з пар ключ-значення (тобто через варіант відповідності “будь-який”).

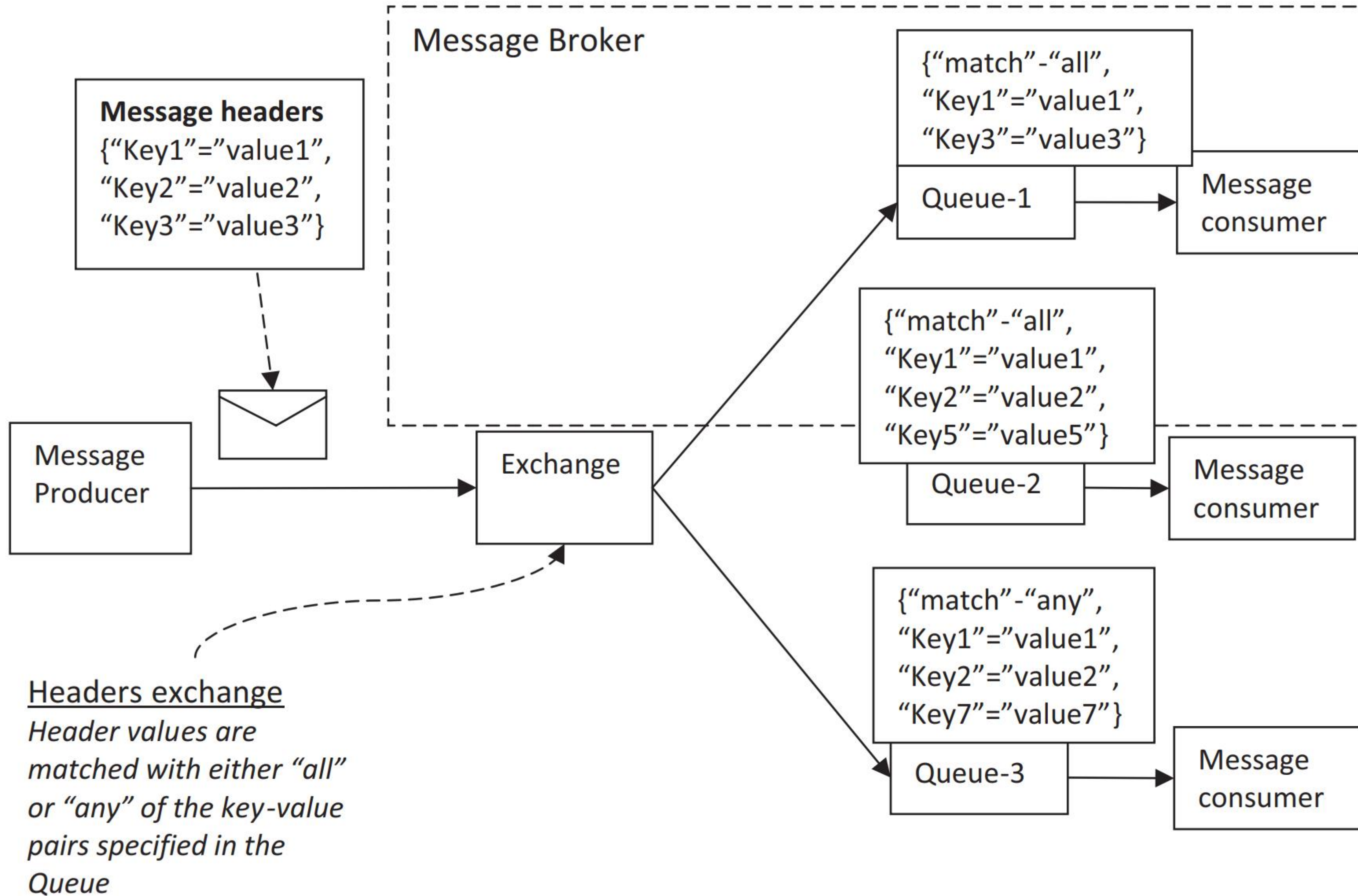


Figure 3.23 Headers exchange.

- Розглянемо обмін заголовками на Рис. 3.23. Його заголовки повідомлень складаються з трьох пар ключ-значення. Вони мають значення {«key1» = «value1», «key2» = «value2», «key3» = «value3»}. Також є три черги, Queue-1, Queue-2 та Queue-3.
- Queue-1 вказує, що обмін повинен відповідати всім вказаним ключам-значенням. Є два ключові значення. Вони мають значення {« Key1 »=« value1 »,« Key3 »=« value3 »}. Оскільки ці два знаходяться в заголовках повідомлень, біржа пересилає повідомлення в Queue-1.
- Для Queue-2, яка вказує, що всі параметри ключ-значення мають узгоджуватися із заголовками, оскільки заголовок не містить “Key7” = “value7”, повідомлення не пересилатиметься в Queue-2.
- У випадку Queue-3, черга вказує відповідність для будь-якої з пар ключ-значення проти заголовків. Оскільки для "Key1" і "Key2" існує два збіги, повідомлення буде переадресовано в Queue-3.