Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis

Design and Development of a Framework for Type-Consistent and Broker-Independent Messaging

Alexander Keck

Course of Study: Softwaretechnik

Examiner: Prof. Dr.-Ing. Steffen Becker

Supervisor: Sandro Speth, M.Sc.

Commenced: March 28, 2023

Completed: September 28, 2023

Abstract

In the context of modern distributed systems, such as microservices, the messaging stack is a highly distributed network of loosely coupled producers and consumers. These components communicate using various serialization protocols, such as *Apache Avro*, *protobuf*, or *JSON schema*, through a central message broker, such as *Apache Kafka*, *RabbitMQ*, *Amazon SQS*, *Azure Service Bus*, and *Google Pub/Sub*. Notably, solutions exist to centrally store schemata in a registry.

The central problem addressed in this work is how to ensure type consistency during message transmission. Existing broker-dependent solutions often prioritize their underlying messaging platform, leading to low-level client implementations. Consequently, this affects the developer experience and necessitates manual integration of application logic between producers and consumers.

The main objective of this thesis is to design and develop an open-source *Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework)* that offers a developer-centric approach to type-consistent messaging. Therefore, this framework aims to provide a unified messaging contract, a schema definition paradigm, and a modular client architecture, all designed to ensure seamless type consistency in communication between producers and consumers.

To achieve this goal, this work analyzes existing partial solutions and combines them into a developer-friendly framework. It begins by defining the fundamental concepts of type-consistent messaging. Among other things, a central registry serves as the single source of truth for schemata and contracts. Additionally, a unified message encoding format and a novel approach to message processing through message endpoints are presented. Next, standardized interfaces for messaging resources and components are defined. Following this, a modular client architecture is developed to ensure high adaptability to a wide range of messaging platforms.

The concepts have been evaluated through a case study involving a *NestJS* reimplementation of the *T2-Project* reference architecture. The evaluation shows that the implementation of a research prototype demonstrated the adaptability of the framework architecture and its seamless integration. Furthermore, the reimplementation of a reference architecture underscores its practical utility. Yet, there is room for improvement in areas like error handling and message reply strategies, which require further development, along with real-world evaluations. To summarize, this work tackles the issue of type inconsistency with a developer-centric approach and continues to show potential for future research.

Kurzfassung

Im Kontext moderner verteilter Systeme, wie etwa Microservices, ist der Messaging-Stack ein verteiltes Netzwerk von locker gekoppelten Produzenten und Konsumenten. Diese Komponenten kommunizieren mithilfe verschiedener Serialisierungsprotokolle, wie *Apache Avro*, *protobuf* oder *JSON schema*, über einen zentralen Nachrichtenbroker, wie etwa *Apache Kafka*, *RabbitMQ*, *Amazon SQS*, *Azure Service Bus* oder *Google Pub/Sub*. Dabei sei erwähnt, dass Lösungen existieren, um Schmata zentral in einer Registry zu speichern.

Als zentrale Problematik geht diese Arbeit der Frage nach, wie die Typkonsistenz während der Nachrichtenübertragung sichergestellt werden kann. Denn die existierenden brokerabhängigen Lösungen priorisieren vielfach ihre zugrundeliegende Messaging-Plattformen und führen in der Folge zu Low-Level-Client-Implementierungen. Als Konsequenz dessen leidet die Developer-Experience und es wird eine manuelle Integration der Anwendungslogik zwischen Produzenten und Konsumenten erforderlich.

Das Hauptziel dieser Arbeit ist es, ein brokerunabhängiges Open-Source Framework zu konzeptionieren sowie zu entwickeln, welches einen entwicklerzentrierten Ansatz für die Typkonsistenz von Nachrichten bietet. Das vorstehend beschriebene *BITCMT Framework* zielt darauf ab, einen einheitlichen Messaging-Contract, ein Schema-Definitionsparadigma und eine modulare Client-Architektur bereitzustellen. Dies soll eine nahtlose Typkonsistenz bei der Kommunikation zwischen Produzenten und Konsumenten sicherstellen.

Um dieses Ziel zu erreichen, analysiert diese Arbeit bereits existierende Teillösungen und kombiniert sie zu einem entwicklerfreundlichen Framework. Zunächst werden die grundlegenden Konzepte des Frameworks definiert, welche als Arbeitsgrundlage im Rahmen der weiteren Arbeit dienen. Unter anderem dient ein zentrales Registry als Single-Source-of-Truth für Schemata und Verträge. Darüber hinaus wird ein einheitliches Nachrichten-Codierungsformat sowie neuartige Ansätze zur Nachrichtenverarbeitung über Nachrichtenendpunkte dargestellt. In einem weiteren Schritt werden standardisierte Schnittstellen für Messaging-Ressourcen und -Komponenten definiert. Daran anschließend wird eine modulare Client-Architektur entwickelt, welche eine hohe Anpassungsfähigkeit an eine breite Palette von Messaging-Plattformen sicherstellen soll.

Die Konzepte wurden anhand einer Fallstudie durch eine *NestJS*-Reimplementierung der *T2-Project* Referenzarchitektur evaluiert. Die Evaluierung zeigte die Anpassungsfähigkeit der Framework-Architektur sowie die nahtlose Anwendungsintegration. Darüber hinaus unterstreicht die Reimplementation in eine Referenzarchitektur die praktische Nützlichkeit des vorgestellten Frameworks. In Bereichen wie integrierte Fehlerbehandlung sowie integrierte Strategien zur Nachrichtenbeantwortung besteht jedoch noch Raum für Verbesserungen, welche als Gegenstand weiterführender Studien weiter entwickelt werden müssen. Zusammenfassend ist feszuhalten, dass diese Arbeit das Problem der Typinkonsistenz mit einem entwicklerzentrierten Ansatz behandelt und fortbestehendes Potential für zukünftige Forschung aufzeigt.

Contents

1 Introduction **1**

2 Foundations and Related Work **3**
2.1 Foundations . . . 3
2.2 Related Work . . . 7

3 Concept **11**
3.1 Overview . . . 12
3.2 Messaging Contract . . . 17
3.3 Schema Compatibility . . . 18
3.4 Registry . . . 20
3.5 Schema Metaprogramming . . . 25
3.6 Message Encoding . . . 26
3.7 Validation . . . 27
3.8 Message Endpoint . . . 28

4 Architecture **31**
4.1 Architecture Overview . . . 32
4.2 Schema Adapter . . . 33
4.3 Registry Adapter . . . 34
4.4 Broker Adapter . . . 37
4.5 Client . . . 39
4.6 Message Production . . . 43
4.7 Message Consumption . . . 46

5 Implementation **53**
5.1 Framework Dependencies . . . 53
5.2 Repository Structure . . . 54
5.3 Minimal Application Example . . . 55
5.4 Summary . . . 58

6 Evaluation **59**
6.1 Case Study: T2-Project Reference Architecture . . . 60
6.2 Threads to validity . . . 62
6.3 Summary . . . 62

7 Conclusion and Future Work **65**

Bibliography **69**

List of Figures

2.1 Message Queuing Exchange Pattern. 4
2.2 Overview of the *Apache Pulsar Schema*. 9
2.3 Overview of the *Confluent Schema Registry* and client integration. 10

3.1 Concept: High-level overview of *type-consistent messaging*. 12
3.2 Simplified: Message processing by the *Producer*. 13
3.3 Concept: Overview of a *type-consistent Producer*. 14
3.4 Simplified: Message processing by the *Consumer*. 15
3.5 Concept: Overview of a *type-consistent Consumer*. 16
3.6 Concept: JSON representation of a messaging *Contract*. 17
3.7 Concept: High-level overview of the *Registry*. 21
3.8 Concept: Schema extraction with metaprogramming. 25
3.9 Concept: Decorating the `SchemaObject.class` with metaprogramming. 26
3.10 Concept: Unified message encoding format. 26
3.11 Concept: Data validation. 27
3.12 Concept: Message *Endpoint* and *Controller*. 29

4.1 The package hierarchy of the architecture. 32
4.2 Class diagram of all resource types within the application scope. 33
4.3 Class diagram with package hierarchy of the *Schema Adapter*. 34
4.4 Class diagram with package hierarchy of the *Registry Adapter*. 35
4.5 Class diagram with package hierarchy of the *Contract* and *Contract Version*. . . . 36
4.6 Class diagram with package hierarchy of the *Broker Adapter*. 37
4.7 Class diagram with package hierarchy of the *Producer Adapter*. 38
4.8 Class diagram with package hierarchy of the *Consumer Adapter*. 39
4.9 Class diagram with package hierarchy of the *Client*. 40
4.10 Class diagram with package hierarchy of the *Message*. 41
4.11 Class diagram with package hierarchy of the *Controller*. 42
4.12 Sequence diagram of the initial initialization of a `SchemaObject`. [Part 1/2] 44
4.13 Sequence diagram of the initial initialization of a `SchemaObject`. [Part 2/2] 45
4.14 Sequence diagram of the *Message Production* process. 47
4.15 Sequence diagram of the initialization sequence of a endpoint. [Part 1/2] 48
4.16 Sequence diagram od the initialization sequence of a endpoint. [Part 2/2] 49
4.17 Sequence diagram of the *Message Consumption* process. 50

Acronyms

API Application Programming Interface. 10, 21, 36

BITCMT Framework Broker-Independent Type-Consistent Messaging Template Framework. iii, 11, 31, 53, 59, 65

broker The remote message broker in the messaging cluster. 4, 12, 31, 55, 59, 65

broker adapter The adapter for the broker proposed by this thesis. 31, 54

channel The messaging channel, or topic, is a transportation route. 5, 11, 32, 56, 66

client The messaging client proposed by this thesis. 21, 31, 56, 66

consumer The consumer of a message. 3, 12, 39, 59, 65

consumer adapter The adapter for the consumer proposed by this thesis. 29, 37

contract The messaging contract proposed by this thesis. 12, 32, 63, 65

contract version The messaging contract version proposed by this thesis. 13, 32, 57, 63, 66

controller The messaging controller proposed by this thesis. 29, 38, 56, 66

DSL Domain-Specific Language. 6, 54

endpoint The messaging endpoint proposed by this thesis. ix, 11, 38, 63, 66

producer The producer of a message. 3, 12, 43, 59, 65

producer adapter The adapter for the producer proposed by this thesis. 37, 58

registry The remote schema and contract registry in the messaging cluster, as proposed by this thesis. 9, 12, 31, 54, 59, 65

registry adapter The adapter for the registry proposed by this thesis. 26, 34, 54

REST Representational State Transfer. 10, 21, 54, 65

schema The schema of a message. 6, 11, 31, 55, 59, 65

schema adapter The adapter for the schema proposed by this thesis. 26, 31, 54

SchemaID Schema Identification Number. 10, 12, 33

SchemaObject A class decorated by its schema, as proposed by this thesis. ix, 7, 26, 39, 56

ts-messaging research prototype. 53, 59, 66

1 Introduction

In highly distributed microservice architectures, messaging plays a pivotal role in facilitating data exchange among various components and systems. The messaging platforms are often at the heart of many distributed event-driven application architectures. Thereby, enabling asynchronous communication, scalability, and fault tolerance through replayability. However, as soon as applications leave their initial development cycle, further evolution becomes challenging. As the complexity of these systems grows and message schemata evolve, so does the challenge of ensuring type consistency during message transmission between various participants. In loosely coupled systems, where components may be developed independently, ensuring that all participants agree on a common message schema becomes a critical concern. Without such agreement, message processing can fail, or data corruption and system failures can occur.

Existing solutions to address type inconsistency in messaging often follow a platform-first approach and are highly broker-dependent. These solutions prioritize the needs of the underlying messaging infrastructure, resulting in low-level client implementations that lack developer experience. While they may provide essential functionality, they fall short in terms of true application-to-application business logic integration.

As a result, developers are frequently compelled to implement additional logic layers to ingest messages into the application logic and manually determine their schema using predeployed schemata and logic. Validation presents additional challenges, as there has been a lack of standardized definitions on a producer-to-consumer level regarding the content of message payloads. This absence of clear guidance further complicates the task of ensuring data integrity and consistency in messaging systems. Moreover, message schemata are often defined externally and integrated only through a code-generation step, introducing a separation between schema and application logic that adds complexity to the development process and may give rise to schema discrepancies. This misalignment between platform-centric solutions and the needs of developers underscores the need for a developer-centric framework template like BITCMT Framework to streamline type-consistent messaging.

To overcome these limitations, this thesis aims to design and develop a framework that focuses on the needs of developers, offering a developer-first approach to type-consistent messaging. Additionally, this framework is designed to be broker-independent, hence the choice of the working title BITCMT Framework. Moreover, the main objectives of this thesis are to define common interfaces for messaging resources and messaging stack component adapters. Furthermore, it aims to develop a uniform messaging contract and schema definition paradigm. Finally, it aims to develop a modular client architecture that empowers developers to integrate type consistency into their preexisting messaging stack. Therefore, the overall goal is to propose an open-source framework that empowers application-to-application messaging which is independent of the underlying messaging platform, promoting loose coupling while ensuring type consistency.

To provide a more comprehensive overview and underscore the primary contributions of this thesis they can be summarized as follows:

1. Conceptualizing a broker-independent, type-consistent messaging framework template, namely the BITCMT Framework.
2. The implementation of such a BITCMT Framework as a TypeScript prototype.
3. A case study, based on the reference architecture of the T2-Project[1], to demonstrate the feasibility and ease of use of the developer-centric approach offered by the BITCMT Framework.

To conclude, by the end of this thesis, the aim is to provide a modular and broker-independent framework template that empowers developers to achieve type-consistent messaging in highly distributed environments, thereby contributing to the robustness and reliability of modern distributed systems.

Thesis Structure

The structure of this thesis is as follows:

Chapter 2 – Foundations and Related Work provides an overview of the foundations and related work through an analysis of the current state of messaging ecosystems.

Chapter 3 – Concept illustrates the concept of type-consistent messaging and its implications.

Chapter 4 – Architecture presents the architecture of the proposed framework.

Chapter 5 – Implementation describes the implementation of the framework as a research prototype.

Chapter 6 – Evaluation evaluates the framework and its concepts.

Chapter 7 – Conclusion and Future Work concludes this thesis and outlines potential areas for future work.

[1] https://github.com/t2-project

2 Foundations and Related Work

This chapter introduces the foundations and related work of this thesis. First, Section 2.1 provides a simplified overview of the current messaging architecture and the concepts important for understanding this thesis. Then, Section 2.2 continues with an analysis of existing literature and solutions that address type inconsistency in messaging.

2.1 Foundations

This section establishes the key components of the messaging cluster and introduces fundamental concepts essential for a comprehensive understanding of this work. Their details and concepts are reduced to a level of relevance to the context of this thesis.

2.1.1 Messaging

Messaging, or *message queuing*, is a communication pattern used for exchanging messages between various applications or components within a cluster. The most prevalent message exchange pattern is the request-response style. In this pattern, a component, known as the producer, sends a message to a remote component, referred to as the consumer, and then waits for the result to be processed immediately. This form of communication operates on a synchronous point-to-point basis. Consequently, this straightforward method of interaction establishes a well-defined contract between producers and consumers, describing the order, the schema, the channel, and the format of the transmitted messages [HW03; JD20].

Additionally, this paradigm offers the advantage of simplicity and ease of comprehension for developers, as each interaction follows a procedural model. However, this simplicity leads to tight coupling between these deeply integrated systems, posing challenges in terms of evolution, scalability, and deployment. Nevertheless, these challenges can be addressed by implementing the *message queuing exchange pattern* [HW03; JD20].

As depicted in Figure 2.1, this particular *message queuing exchange pattern* is asynchronous. operates asynchronously. Unlike the request-response model, there is no immediate response sent back to the original producer. Instead, the producer receives confirmation that the message has been successfully transmitted into the messaging platform. The messages move through this system as if they were on a one-way street. Moreover, this analogy extends to the concept of multilane-one-way street lanes, known as *partitions*.

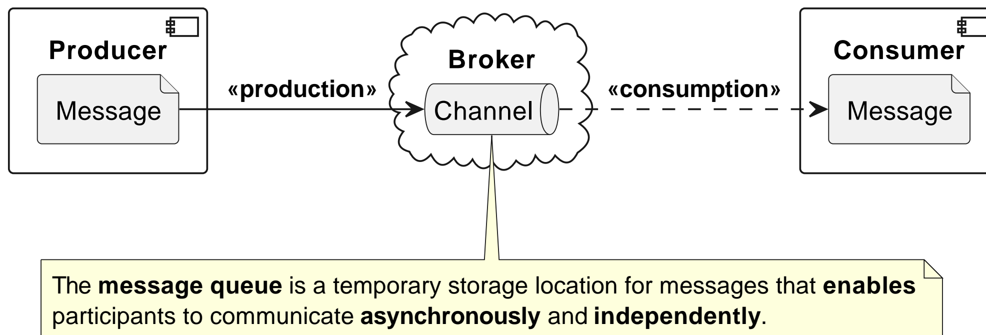

Figure 2.1: Message Queuing Exchange Pattern.

In this context, messages within a lane are often organized in a first-in, first-out queue. This arrangement ensures that messages are evenly distributed among the various lanes and processed sequentially by the consumers. Consequently, to facilitate this communication model, a mediator is required in the form of a broker, as introduced in Section 2.1.2 [Gon22].

2.1.2 Broker

Figure 2.1 also illustrates that the *broker* serves as the intermediary within the messaging cluster. Consequently, every produced message is received by the broker, which then proceeds to distribute it to the corresponding consumers. This distribution occurs in a queue, and as a result, the consumers. do not expect an immediate response.

The broker's primary role is to handle the storage and distribution of messages, and it acknowledges the receipt of new messages to the producers. When a message's payload requires a response, the roles of the producers and consumers are reversed, with the producer becoming a consumer of the response. Consequently, the communication is entirely asynchronous and decoupled [JD20].

It is worth noting that the content of these messages is irrelevant to the broker's distribution process, and traditionally, the broker does not verify the incoming payload [JD20]. Hence, the primary objective of this thesis is to establish a messaging system that ensures type consistency and operates independently of the underlying broker.

The most prominent broker platform is *Apache Kafka*, renowned for its numerous applications across industries, communication, and event persistence [ZK21]:

> «Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.»[1]

[1] Source: https://kafka.apache.org/

Numerous other messaging solutions exist, such as *RabbitMQ*[2], *Amazon SQS*[3], *Azure Service Bus*[4] and *Google Pub/Sub*[5]. Nevertheless, it is important to note that these solutions are not as widely adopted as *Apache Kafka*. As a result, this work primarily concentrates on integration with *Apache Kafka* and provides examples within that context.

Message

Messages are the core components transmitted during messaging. Initially, messages predominantly described state or state changes, essentially serving as events. However, in modern distributed systems, especially within microservices architectures, brokers and their messages are utilized for a wide range of purposes [ZK21]. Therefore, messages can have all kinds of content. In general, however, three fundamental components can be identified:

1. The **channel** channel signifies the path through which the message is transmitted (cf. Section 2.1.2).
2. The **payload**, a record containing the content of the message. Typically, the payload is represented as a serialized object in the form of a byte array. It is important to note that the payload's content is typically irrelevant to the broker.
3. The **metadata** includes various attributes and broker-specific information transmitted alongside the payload, such as headers, the partition number, or the message offset. It is important to note that this metadata is broker-specific and does not contribute to the proposed type consistency approach.

Channel

The *channel*, often referred to as a topic, acts as the central message transport path [Apa23a]. Channels serve as data container queues, enabling producers to publish data records and consumers to subscribe to and process these records. Additionally, channels can be subdivided into multiple partitions, allowing for high throughput and horizontal scalability in the *publish-subscribe messaging paradigm* [HW03; ZK21].

Producer

A *producer* is a core component of a messaging cluster and acts as the source of data or events. Its only responsibility is to publish and serialize messages. As a result, the producer is unaware of the consumers and their message processing. This decoupling implies that the payload is not tied to any specific schema. However, the producer does specify the channel to which the message is published and its payload [ZK21].

[2] https://www.rabbitmq.com/
[3] https://aws.amazon.com/de/sqs/
[4] https://azure.microsoft.com/
[5] https://cloud.google.com/pubsub

Consumer

The *consumer* is the core component responsible for processing data or events within the messaging cluster. Consumers subscribe to specified channels, allowing the broker to deliver messages published to those channels. One of the key responsibilities of a consumer is deserializing the message payload. As a result, the consumer must have knowledge of the schema for the message payload during execution [HW03; ZK21]. However, the concept of message serialization with a schema has developed over time. Consequently, the consumer was not originally designed to handle type-consistent message deserialization, which had to be added as an overlay concept. Thus, this work aims to extend the consumer to handle type-consistent message deserialization.

2.1.3 Schema

As already foreseen by Section 2.1.2 and Section 2.1.2, the producer's ignorance of the destination of his messages, as well as the uncertain origin of the messages arriving at the consumer, poses a significant problem in the processing of messages. Therefore, producers and consumers must agree on a common message schema to reach extended compatibility in distributed systems [Var16]. Hence, there are many approaches to defining descriptive languages for message serialization. The most prominent being *Apache Avor*[6], *protobuf*[7], and *JSON Schema*[8]. They are tailor-made *schema Domain-Specific Language (DSL)* [WB23] for the serialization and deserialization of messages. In general, however, the message broker has no interest in validating and distributing the message's schema since its only function is to route the bits and bytes of each message correctly [HW03]. Therefore, this work presents an efficient concept for schema transfer down to the business logic of the applications.

Schema Evolution

Schema evolution is the process of modifying a schema after its initial deployment. During development, schemata are subject to continuous changes, as it can be challenging to define a comprehensive schema for all potential interactions and interfaces from the outset. However, when these changes are performed in a productive system, they must be thoroughly documented and carefully planned. Changing the entire system to accommodate a new schema all at once is practically infeasible, which underscores the importance of a well-structured approach to schema evolution [Con23a].

In the context of this thesis, the schema evolution changes that are particularly relevant include *adding (optional) fields* and *removing (optional) fields*. While schema evolution can involve various unique features and edge cases depending on the schema format, many of these cases can be traced back to these two fundamental types of changes. For instance, renaming a field can be treated as a combination of removing the old field and adding a new one. Overall, these changes are key considerations when dealing with schema compatibility as discussed in Section 3.3.

[6] https://avro.apache.org/
[7] https://protobuf.dev/
[8] https://json-schema.org/

2.1.4 Type Safety

Type safety is a fundamental concept in programming that ensures that the data types are used consistently and predictably throughout the execution of an application [WF94]. In the context of messaging, type safety refers to a paradigm that ensures that a message sent by a producer and received by a consumer adhere to a predefined message schema. Furthermore, these implied implementation details prevent runtime errors and communication issues. Type safety is critical for consistency and reliability in distributed systems [ZK21].

Schema-First vs. Code-First

In the context of type-consistent messaging, two approaches are prevalent. The first is the *schema-first* approach, commonly seen in technologies that heavily rely on platform-first implementations. In this approach, the schema plays a central role in defining structured data formats and relationships, ensuring precise data consistency constraints. However, this approach has limited flexibility, as outlined in Section 2.1.3. Schemata tend to evolve continuously, and a consumer following a schema-first approach is bound to its predeployed schema. Moreover, when employing concepts like schema registries that facilitate dynamic assignment of message schemata, a code-first approach could be more suitable.

In a *code-first* approach, the schema is generated from the code itself, offering greater flexibility and accommodating dynamic schema evolution and compatibility strategies (cf. Section 3.3.1). Therefore, given this thesis's focus on a developer-centric framework, the code-first approach has been chosen. Nevertheless, it is recommended that future work explores a hybrid model that allows for schema-to-code generation.

2.1.5 Metaprogramming

Metaprogramming is the modification of object-oriented programs by annotating their classes with metadata that modifies the metaobject through preprocessing. Thus, new concepts are introduced to the application's programming language, which are not natively supported [Kru16]. In the context of this work, this metadata is mainly the schemata of payloads, which are added to the associated metaobject of the respective `SchemaObject.class` by applying metaprogramming paradigms.

2.2 Related Work

This section provides an overview of the literature review that was conducted for this paper. First, the literature methodology in Section 2.2.1 is discussed. Then, an overview of the meager academic findings is presented in Section 2.2.2. In contrast, the remaining sections present open-source and commercial solutions already emerging through high industrial demand.

2.2.1 Literature Research Methodology

The literature research has been done by using the following search engines: IEEE Xplore, Google Scholar, Springer Link, and the Catalog of the Universitätsbibliothek Stuttgart. The following keywords were used: *type consistency*, *type saftey*, *event streaming platform*, *messaging*, *event driven*, *schema registry*, brokers (*Kafka*, *RabbitMQ*, ...), schemata (*Apache Avro*, *protobuf*, ...), and their respective combinations. With these keywords, much literature on data consistency was discovered. However, no meaningful literature on type safety in messaging clusters could be found. Therefore, it must be assumed that researchers have paid little attention to type consistency in a distributed messaging system. With this in mind, the literature research was extended to include commercially sponsored literature like blog posts and documentation of open-source and enterprise solutions.

2.2.2 Academic Overview

The academic work is mainly limited to creating schema DSL and their associated tooling. Among other things, there exists primarily literature that formally defines the requirements for such a DSL [NK12; R N09]. However, the schema implementations mentioned in Section 2.1.3 already meet these standards - thus, we can exclude the schema DSL from the scope of this work. Instead, this thesis deals with the modern problem of how the messaging schemata can be exchanged efficiently between applications in a distributed system, such as a microservice architecture.

As mentioned, this problem has not yet been approached academically for messaging systems. Therefore, as a first approach, research was conducted toward a classic request-response model. During the research, the paper from Schürmann et al. [STS20] came into awareness. They presented a type-safe solution for collective adaptive systems. Here, the schemata are extended by a proper query description in a newly developed DSL. This DSL acted as a contract between the components so that code for the client applications could be generated. Furthermore, this work described three application lifecycle planes in which type safety is essential. Namely, exactly when the schema is created in the application development environment, when it is compiled, and when the application is executed [STS20]. Suppose this knowledge is transferred to the messaging context. In that case, it can be concluded that this work requires defining a messaging contract between the individual participants, which describes the schema and the endpoints of the transport path. Furthermore, the requirement to create and load these contracts during runtime is evident.

2.2.3 Apache Pulsar

> «Apache® Pulsar™ is an open-source, distributed messaging and streaming platform built for the cloud.»[9]

Apache Pulsar has become a top-priority project of the *Apache Foundation*. It offers proper client integration with a cloud-native messaging architecture [Sha22]. Thus, Apache Pulsar primarily offers high-level reliability concepts and solutions for message transmission. Many of these concepts

[9]Source: `https://pulsar.apache.org/`

can be transferred well to the concepts and the proposed framework architecture described in this paper. However, their approach is different. Pulsar is primarily an all-in-one messaging platform, so the implementation focuses on the messaging cluster, i.e., the broker. This work, on the other hand, is broker-independent. In Pulsar, for example, the broker plays a central role in that, in addition to the actual management and transmission of messages, it also takes on the task of a *schema registry* [Apa23b].

Such a schema registry is a central element of type-consistent messaging since it manages the schemata of each message participant. Hence, the registry serves as a *single source of truth*[10] for schemata. Thus, strongly typed schema definitions are a requirement. Furthermore, this concept, namely *Pulsar Schema* (cf. Figure 2.2), offers schema evolution, versioning, and compatibility strategies [Apa23c].

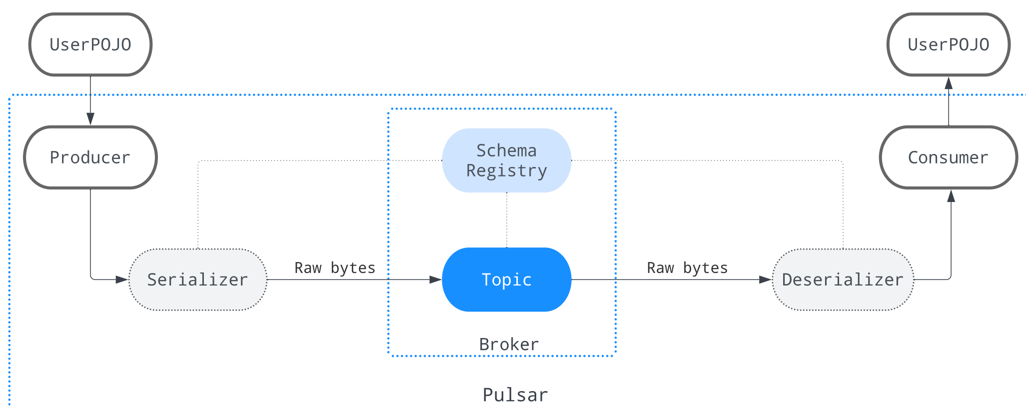

Figure 2.2: Overview of the *Apache Pulsar Schema*.
Source: https://github.com/apache/pulsar-site/blob/main/static/assets/schema.svg

Additionally, as Figure 2.2 shows, the platform handles the serialization and deserialization process, and the messages are transported in a raw byte format. Thus, before producers and consumers join the cluster, they must reevaluate and compare their schemata against the registry. Furthermore, it is also the responsibility of the registry to determine compatibility [Apa23b].

These proven concepts ensure type-consistency messaging. Therefore, they must be equivalently integrated through this work. However, the broker independence of this work requires a different approach. The broker cannot perform schema validations since it is not part of the proposed framework. Therefore, the clients must figure out the schema and consistency of their messages by themselves. Furthermore, the client libraries for such all-in-one platforms often lack the desirable developer experience and rather focus on low-level platform integration. This work, on the other hand, has a developer-first approach. Thus, the client libraries must be easy to use and integrate into the business logic, promoting concepts like metaprogramming.

[10] Apa23b.

2.2.4 Confluent Platform and Schema Registry

«Apache Kafka® Reinvented for the Data Streaming Era»[11]

Confluent is one of the leading innovators of enterprise messaging platforms. They identified common problems in large-scale messaging systems. These are: *data inconsistency*, *incompatible data formats*, *schema evolution* (cf. Section 2.1.3), *schema validation*, and *data governance* [Con23c]. Therefore, they proposed one of their most essential enterprise features, which is the creation of explicit contracts between individual messaging participants. Similar to other platforms, these versioned schemata are managed by a schema registry [Con23c]. Furthermore, as Figure 2.3 shows, they developed the concept of *message encoding* (cf. Section 3.6), which is an additional step after message serialization that identifies the schema of the message through on the basis of a Schema Identification Number (SchemaID).

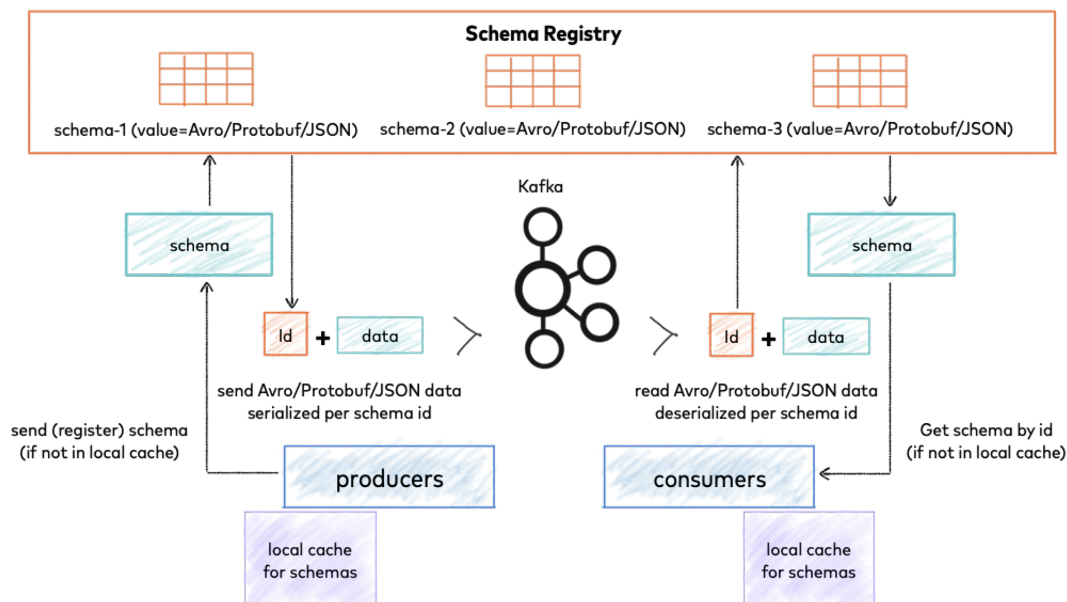

Figure 2.3: Overview of the *Confluent Schema Registry* and client integration.
Source: https://docs.confluent.io/platform/current/_images/schema-registry-and-kafka.png

However, in comparison to Apache Pulsar, their schema registry is broker-independent and accessible through a simple Representational State Transfer (REST) Application Programming Interface (API). Furthermore, they are the only entity to provide their schema registry as an open-source product[12], although with a community license. However, for now, their registry can be adapted to be broker-independently integrated into the framework proposed by this work.

[11] Source: https://www.confluent.io/

[12] https://github.com/confluentinc/schema-registry

3 Concept

This chapter outlines the essential concepts required for the transmission of a type-consistent message. As previously noted, the concepts are designed to complement and extend existing approaches and solutions such as those introduced in Chapter 1.

The central objective of this thesis is to develop a Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework) that ensures type consistency. As discussed in Section 2.2.1, the existing literature on this topic is relatively scarce. Therefore, many of the concepts elaborated upon in this chapter draw inspiration from pre-existing partial solutions, including both open-source and enterprise applications.

To facilitate the combination and subsequent evaluation of these distinct concepts, a small prototype[1] was developed at the beginning of this work. The impressions gained from this initial implementation have contributed significantly to the design and development of the proposed BITCMT Framework. It is worth noting that certain concepts that enabled true end-to-end type-saft development would have required them to exist within a single monorepository[2]. However, this has proven to be impractical in the context of a distributed and versatile messaging system. Consequently, functional approaches that enable the creation of type-safe remote procedure calls, such as tRPC[3], are automatically excluded. Instead, the concepts put forth in this work primarily stem from object-oriented programming paradigms. However, the proposed solution demands a certain level of code preparation, and modification metaprogramming serves as an ideal concept.

However, this chapter serves as a preparatory foundation by providing an overview of the fundamental concepts essential for achieving type-consistent messaging. It begins in Section 3.1 with an introduction to the general concept of type-consistent messaging. Following this introduction, the chapter delves into a detailed presentation of each individual concept. Section 4.3.1 and Section 3.3 are dedicated to the development of a contract between producers and consumers. This contract defines the message transmission channel and the message schema, therefore, governing the interaction between producers and consumers. To facilitate the management and storage of this contract, a central registry is proposed, as further discussed in section Section 3.4. Subsequently, section Section 3.5 introduces a paradigm for describing a schema within the application code. Building upon this foundation, the subsequent sections, Section 3.6 and Section 3.7, explore concepts designed to establish a uniform message encoding format. The chapter concludes in section Section 3.8 by introducing a novel message endpoint inside the consumer.

[1] https://github.com/unaussprechlich/messty

[2] https://monorepo.tools/

[3] https://trpc.io/

3.1 Overview

The general concept of type-consistent messaging is illustrated in Figure 3.1. In a conventional messaging stack, schema is either transmitted alongside the message payload through the broker or exchanged manually during the development process. However, when a fourth component, referred to as the registry as elaborated upon in Section 3.4, is introduced, the schema can be extracted from the message payload. Moreover, it can be substituted with a reference in the form of an Schema Identification Number (SchemaID) during transmission, as detailed in Section 3.6. The actual schema can then follow an indirect path via the registry. To optimize this process, caching mechanisms can be employed, enabling the schema to be exchanged only once between the producer and the consumer [Con23c].

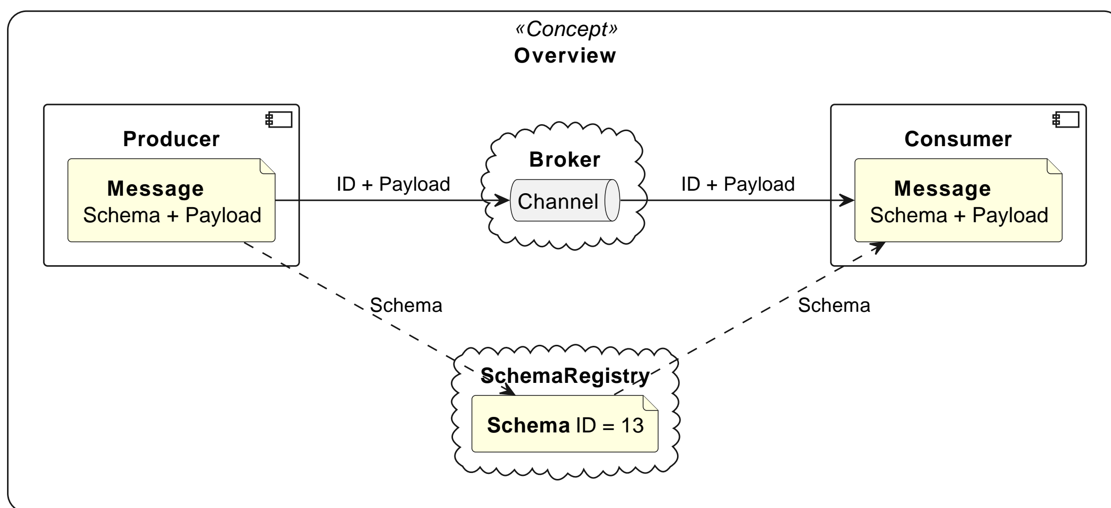

Figure 3.1: Concept: High-level overview of *type-consistent messaging*.

By leveraging concepts such as a *contract* (cf. Section 3.2), *schema metaprogramming* (cf. Section 3.5), and *schema compatibility strategies* (cf. Section 3.3), this form of type-consistent messaging can be significantly enhanced and deeply integrated into the development process.

To provide a more comprehensive understanding, it is beneficial to analyze the applications of producers and consumers separately. It's this very messaging ecosystem that facilitates the initial decoupling of consumers from producers. Consequently, consumer and producer applications are typically developed independently and often reside in distinct repositories. Chapter 3.1.1 offers a concise Producer Overview detailing the internal processes and essential concepts required to ensure type consistency. On the other hand, Chapter 3.1.2 presents an equivalent Consumer Overview.

To provide a more comprehensive understanding, it is beneficial to view the applications of producers and consumers separated and independent components. It is this very messaging paradigm that enables the decoupling of consumers from producers in the first place. Thus, consumer and producer applications are developed separately and rarely share the same repository. Therefore, Section 3.1.1 is a briefly *Producer Overview* detailing the internal processes and concepts necessary for type-consistency. whereas, Section 3.1.2 presents an equivalent *Consumer Overview*.

3.1.1 Producer Overview

The *producer* serves as the component responsible for generating messages within the messaging platform. To transform a message, intended for transmission by the producer, into a type-consistent message, it must undergo a defined sequence of processes. This sequential flow guarantees that the broker can effectively process the message, and subsequently, the consumer can successfully read and process its payload.

As depicted in Figure 3.2, this sequence of processes is illustrated. In essence, every message must undergo *validation*, *serialization*, and *encoding* before it becomes ready for transmission. Validation guarantees that the payload within the message is valid for the specific schema. Subsequently, the serialization step transforms the payload into a byte-readable format. Lastly, the encoder ensures that the broker can effectively process the message and adds a schema reference, the SchemaID, as a prefix to the previously serialized payload.

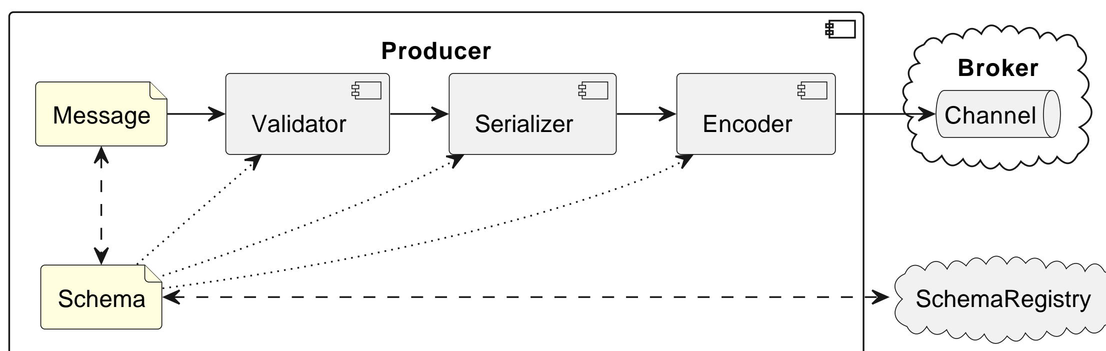

Figure 3.2: Simplified: Message processing by the *Producer*.

At first glance, the message processing procedure may appear straightforward. However, achieving complete type consistency is more intricate. While the producer possesses knowledge of the payload schema and can validate, serialize, and encode it effectively, the consumer lacks information regarding the original schema and the channel through which the message will be delivered. Consequently, the consumer cannot confidently decode, deserialize, or validate the message without assurance that type consistency will not be compromised. This underscores the necessity of comprehensive type-consistency mechanisms in the messaging ecosystem.

The solution to this challenge lies in establishing a well-defined message transmission format and ensuring the continuous exchange of all schemata. This paper introduces a concept that yields a distinct message format, as outlined in Section 3.6. The simplified concept operates as follows: Each schema is assigned a unique identification number, the Schema Identification Number (SchemaID). Subsequently, a prefix - referred to as a *magic byte* (`0000 0000`) - is appended, followed by the SchemaID, just before the serialized payload within a single byte array. This format is formalized using the messaging contract, as discussed in Section 3.2. By utilizing the schema reference, i.e., the SchemaID, in conjunction with the message channel, a unique contract version can be determined. Consequently, the schema, format, and transmission channel are all distinctly defined through this contract.

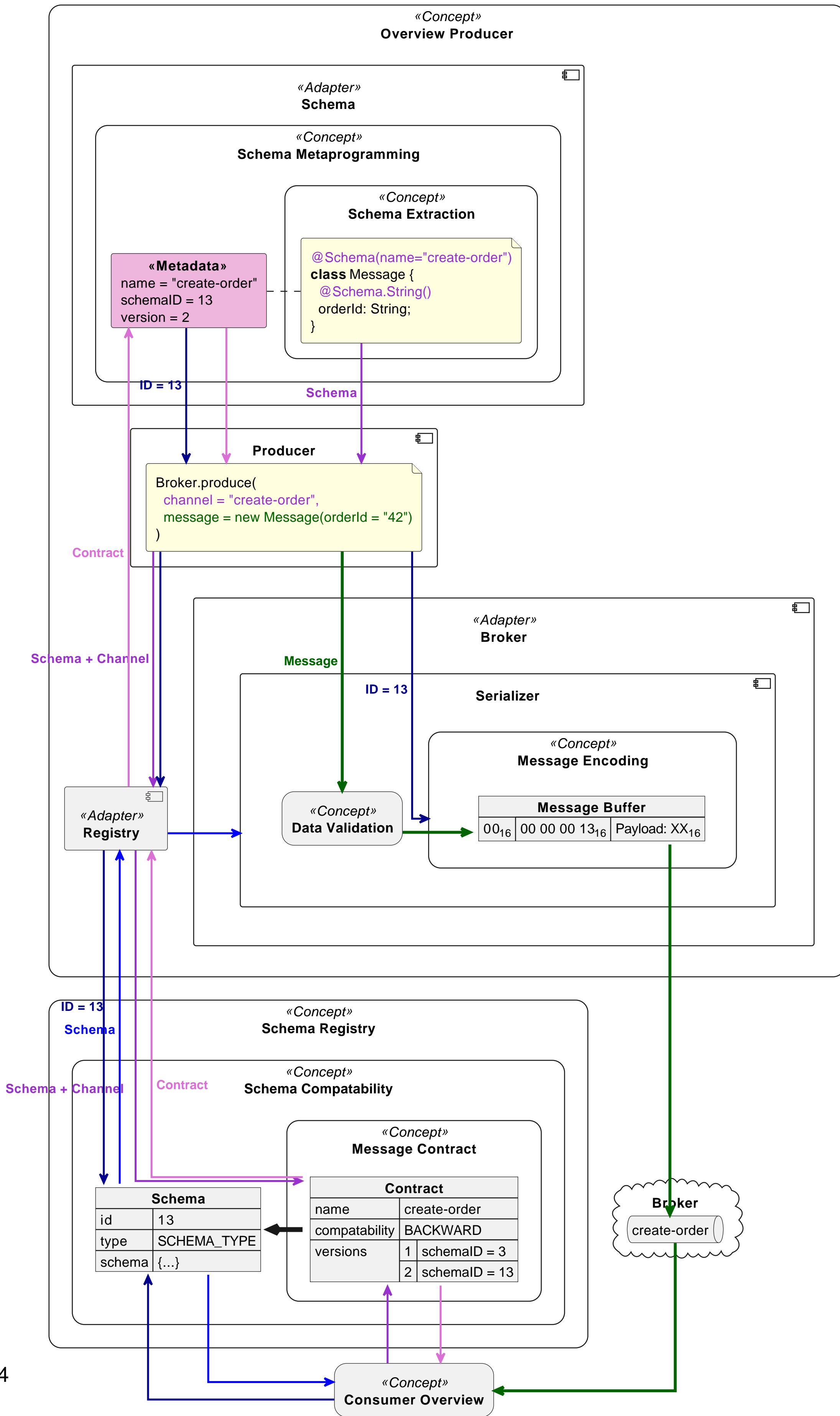

Figure 3.3: Concept: Overview of a *type-consistent Producer*.

Figure 3.3 offers an overview of the concepts that will be presented in the subsequent sections. Primarily, this diagram aims to illustrate the flow of messages, schemata, and contracts throughout the system. Detailed explanations of each mentioned concept can be found in their respective sections. Interactions with pre-existing components are facilitated through an adapter pattern, as discussed in Chapter 4.

3.1.2 Consumer Overview

The *consumer* plays a crucial role in processing the messages generated by producers. A typical consumer receives all messages produced within a channel, regardless of the format of the message or its payload. Consequently, it becomes the sole responsibility of the consumer to handle them correctly. This approach, however, can lead to issues, as it often necessitates extensive and error-prone message analysis to determine whether they align with the desired payload schema. In a conventional messaging system, without standardized agreements in place, ensuring type consistency becomes a challenging endeavor.

As discussed in Section 3.1.1, this paper has already presented an approach to ensure that the producer generates messages in the desired schema and informs the cluster of its schema. In a messaging system designed for type consistency, it becomes the responsibility of consumers to process this information and guarantee that the business logic for this message is invoked only when type consistency is assured. Figure 3.4 offers a simplified representation of the steps required to unpack a received message. This involves *decoding*, *deserialization*, and *validation*. This processing pipeline serves as the sole means to ensure that a transmission is type-consistent.

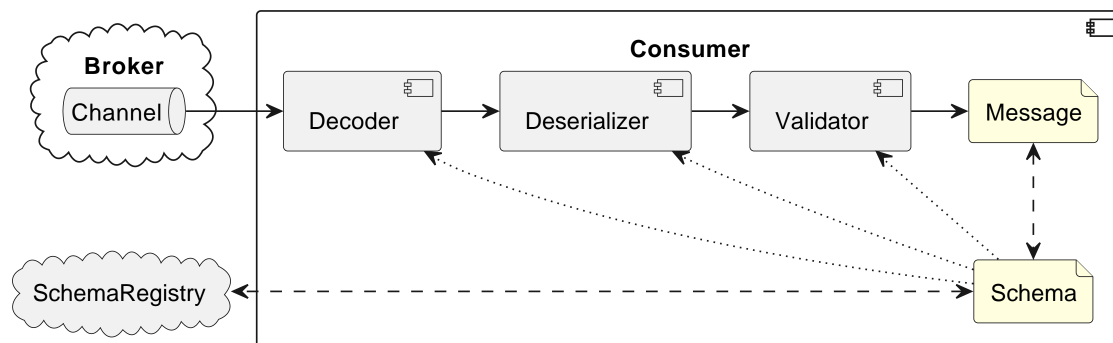

Figure 3.4: Simplified: Message processing by the *Consumer*.

Thanks to the well-defined transfer format described in Section 3.6, the Schema Identification Number can be extracted from the message. Consequently, the schema stored in the registry can be retrieved by the consumer. This schema is then utilized to deserialize the message. However, in certain environments, it may be necessary to revalidate the payload, especially if there is no guarantee that the message was produced by a type-consistent producer.

With the message endpoint concept introduced in Section 3.8, the consumer can explicitly specify which messages it can process. As a result, the consumer is no longer the entry point into the application context. Instead, this responsibility is transferred to the endpoint, which is only invoked when type consistency has been ensured. This architectural shift enhances the overall robustness and reliability of the messaging system.

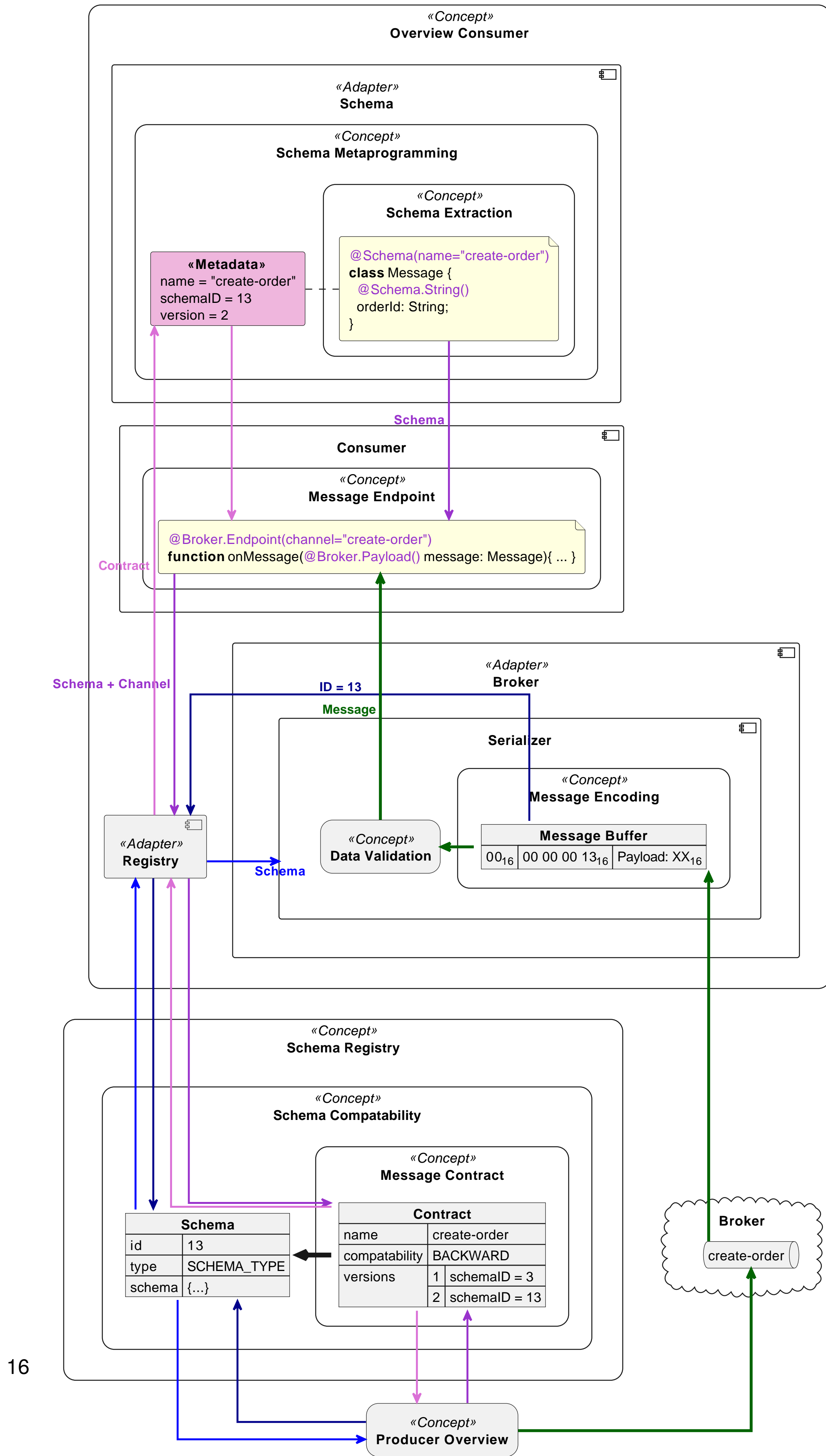

Figure 3.5: Concept: Overview of a *type-consistent Consumer*.

Given the intricacies of these processes, Figure 3.5 provides an overview, emphasizing the flow of information. Collectively, these two diagrams encompass a comprehensive depiction of a complete and type-consistent messaging paradigm.

3.2 Messaging Contract

The messaging system facilitates the transmission of messages between senders and receivers. This communication involves multiple producers and consumers operating over a shared message channels. As this transmission occurs asynchronously within a distributed system, producer has no confirmation of successful message delivery. Furthermore, it is unclear whether the consumer can process a message that has been produced by the producer. To eliminate these ambiguities, it is imperative to establish a precise definition of the messaging scheme and the transport pathway. This definition is called the messaging *contract*.

The messaging *contract* represents an agreement between the sender, referred to as the *producer* (cf. Section 2.1.2), and the receiver, known as the *consumer* (cf. Section 2.1.2). This contract provides a clear and explicit definition comprising two fundamental components: the *channel* (cf. Section 2.1.2) and the message *schema* (cf. Section 2.1.3).

The channel serves solely as a transportation pathway, facilitating message direction, without any knowledge of the content or format of the messages it conveys. In contrast, the schema is concerned solely with the format, encompassing the structure and constraints of the message payload.

Therefore, a comprehensive contract must include, at a minimum, these essential elements:

1. The **schema** defines the format of the message and should describe it precisely.
2. The **channel** describes the message route for which the contract is valid.

The contract can share its name with the channel, effectively providing a clear reference to the associated channel. To ensure a unique assignment between a channel and a schema, versioning is the preferred method. Additionally, it is essential to avoid overwriting previous contract versions with newly registered schemata. Consequently, each contract version should be a uniquely incremented integer.

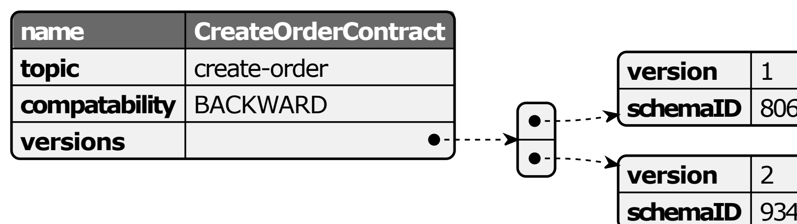

Figure 3.6: Concept: JSON representation of a messaging *Contract*.

Figure 3.6 provides an illustrative representation of a contract. Since this agreement must be machine-readable, a straightforward format, such as JSON[4], is strongly advocated. Furthermore, this foundational format can be extended to accommodate additional descriptions or impose further constraints within the same contract.

Additionally, the contract needs to be stored and managed centrally by a registry (cf. Section 3.4). Instead of storing the entire schema in the contract, the schema is given a unique identifier, its SchemaID, and stored separately from the contract in the registry. This approach allows a single schema to be reused in multiple contracts.

In such a distributed messaging system, updating all participants simultaneously to a contract version is virtually impossible. Therefore, the contract should also define information about the *schema compatibility* (cf. Section 3.3) between the individual contract versions and the update sequence of the producers and consumers in a *schema updating strategy* (cf. Section 3.3.3).

3.3 Schema Compatibility

The concept discussed in Section 3.3.1 for handling schema evolution and the resulting compatibility strategies was primarily influenced by *Confluent, Inc.*[5][Con23a]. Notably, the *Apache Foundation* shares a similar perspective on schema compatibility. In their documentation titled «*Understanding Pulsar Schema*», they define comparable compatibility modes for their *Pulsar* platform[Apa23c]. Therefore, the following sections will first introduce the concept of schema compatibility and then discuss the compatibility strategies in detail.

As outlined in Section 2.1.3, the key scenarios pertinent to schema compatibility revolve around the introduction and elimination of (optional) fields. The inclusion of as many fields as optional within the schema proves crucial. Unlike non-optional fields, these optional ones can be easily removed or substituted with default values. Consequently, it becomes evident that there should be three distinct types of compatibility between schema version X and version $X - 1$:

Backward compatibility implies that consumers using schema version X can successfully read data produced with version $X - 1$ [Con23a].

In practical terms, it allows for the removal of old fields that have become obsolete for future processing. The deletion of optional fields follows a similar principle. Additionally, optional fields can be added without issues, as they can remain empty for processing by definition. However, introducing a required new field will trigger errors, as existing messages with the previous schema version lack this field.

Forward compatibility signifies that data produced with schema version X remains readable by consumers using version $X - 1$, even if they may not fully utilize the capabilities of the new schema [Con23a].

[4] https://www.json.org/
[5] https://www.confluent.io/

In essence, it permits the addition of fields since older consumers can simply ignore them during processing. The addition of optional fields follows a similar pattern. Furthermore, the deletion of optional fields is also allowed since they were optional by definition. However, deleting a required field will result in errors for older consumers, as this data will now be missing during processing.

Full compatibility This denotes that schemas inherit both backward and forward compatibility. Schemas evolve in a fully compatible manner, allowing old data produced with schema $X - 1$ to be read with the new schema X, and vice versa [Con23a].

This means that only changes within the intersection of forward and backward compatibility are permitted. These permitted changes are the removal and addition of optional fields.

In most cases, it is adequate to assess compatibility between two schemas since a schema typically undergoes updates by only one version in a standard update process. However, there are scenarios where transitive compatibility is required. Transitive schema compatibility extends beyond the relationship between X and $X - 1$, therefore, encompassing the inductive compatibility across the entire sequence from X to $X - n : \forall n \in \mathbb{N} \land n < X$.

3.3.1 Schema Compatibility Strategies

For the contract and the registry, which is responsible for verifying the compatibility of newly registered schemas, these compatibility strategies align with those employed by Confluent and the Apache Foundation for their implementation [Con23a].

In this context, schema X represents a schema contract version. Consequently, $X - 1$ corresponds to the preceding contract version, and $X - n$ pertains to any prior contract version in the sequence.

`BACKWARD` The consumer of schema version X can read data produced with version $X - 1$.

`BACKWARD TRANSITIVE` The consumer of schema version X can read data produced with version $X - n$.

`FORWARD` The consumer of schema version X can read data produced with version $X + 1$.

`FORWARD TRANSITIVE` The consumer of schema version X can read data produced with version $X + n$.

`FULL` The consumer of schema version X can read data produced with version $X + 1$ and $X - 1$.

`FULL TRANSITIVE` The consumer of schema version X can read data produced with version $X + n$ and $X - n$.

`NONE` The consumer of schema version X can only read data produced with version X.

3.3.2 Schema Normalization

The process of *schema normalization* can increase compatibility between schemata. Essentially, it allows for certain schema modifications without altering the underlying schema semantics. As a general rule, any changes that solely impact formatting, such as reorganizing the string representation, can be implemented seamlessly. In this thesis, we refer to this initial form of normalization as *level-1 normalization*.

Level-2 normalization encompasses modifications that extend beyond mere formatting adjustments. Depending on the schema type, this can include normalizing the order of fields or their names without affecting the schema's core semantics. It is important to emphasize that these alterations are strictly syntactical and do not impact the functionality of the schema.

3.3.3 Updating Strategies

The updating strategy defines the sequence in which applications must be updated to ensure the preservation of the messaging contract. The following strategies are available:

Consumer First

This strategy applies when the compatibility strategy is either `BACKWARD`, or `BACKWARD TRANSITIVE`. In this scenario, a consumer using schema version X is only capable of reading previous versions. Consequently, all consumers must be updated to version $X + 1$ before a producer can send a message with version $X + 1$.

Producer First

This strategy applies when the compatibility strategy is of the `FORWARD` or `FORWARD TRANSITIVE` type. Consumers using schema version X can read version $X + 1$ without issues. However, before bringing the consumers to version $X + 1$, the producers have to write messages only with version $X + 1$, as consumers with schema version $X + 1$ are not compatible with version X. Therefore, in this case, the producer has to be updated first.

Any First

If the compatibility strategy is of type `FULL` or `FULL TRANSITIVE`, the producer and consumer can undergo a schema version update process in any order.

No update sequence can be defined for the compatibility strategy `NONE` because the absence of compatibility restrictions makes it impossible to determine a compatibility conclusion.

3.4 Registry

The *registry* serves as a centralized interface within a messaging cluster for the management and storage of schemata and messaging *contracts*. It empowers individual participants within the messaging platform to attain exceptional message integrity and consistency, all without the need for direct communication amongst themselves. An overarching view of the distinct components and responsibilities of the registry is presented in Figure 3.7.

The primary responsibility of the registry is to allocate a distinctive SchemaID to each schema within the messaging cluster. Moreover, it is tasked with offering a *RESTful API*[6] that makes this mapping easily accessible through straightforward HTTP[7] requests. Consequently, this component can aptly be labeled as *schema management* (cf. Section 3.4.1).

The registry undertakes another crucial role, which is the management of message contracts, referred to as *contract management* (cf. Section 3.4.2). In simple terms, it must offer a unified interface for handling the contract and contract versions associated with the channel through which communication is intended between producers and consumers.

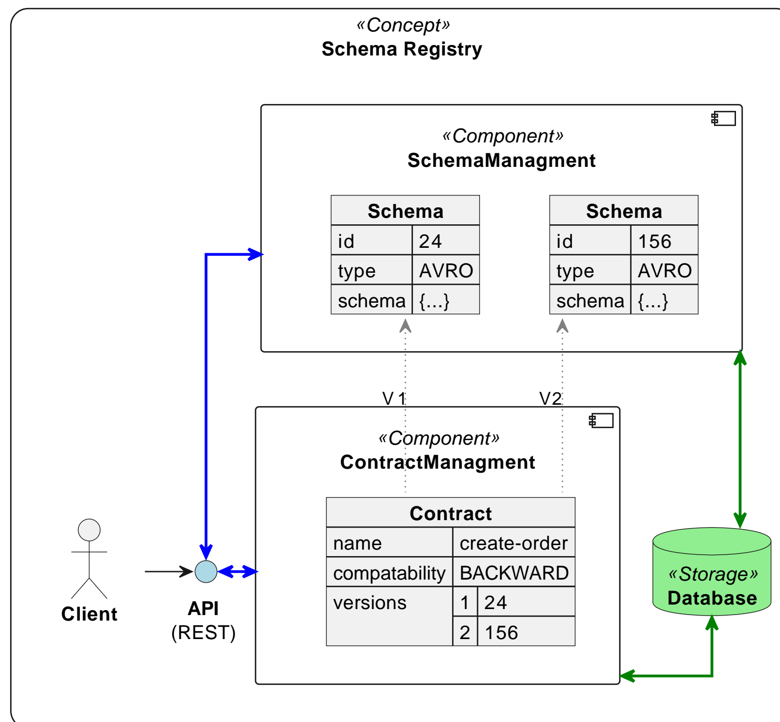

Figure 3.7: Concept: High-level overview of the *Registry*.

As outlined in Figure 3.7, the registry also bears the responsibility of providing a unified Application Programming Interface (API) – preferably, a straightforward Representational State Transfer (REST) interface – for the clients. To gain a deeper insight into the prerequisites of this API, the following two sections outline the REST endpoints relevant to this thesis. In this description, we focus solely on error-free communication with these endpoints. Therefore, the discussion of classic errors within the `404` or `500` segment is omitted. These endpoints serve as the foundational requirements for identifying a suitable registry or forming the basis for a custom implementation.

[6] https://aws.amazon.com/what-is/restful-api/

[7] Hypertext Transfer Protocol

3.4.1 Schema Management

The *schema management* is the core task of the registry. The creation and administration of schemata should exclusively occur through the contract endpoint, thereby preventing the deletion of schemata which are still referenced by a contract versions (cf. Section 3.4.2). Moreover, schema management also incorporates the crucial task of schema validation. Consequently, the registry confirms and validates specific schema specifications, such as *Apache Avro*[8]. This validation process holds paramount importance for the registry as it ensures the compatibility of the schema with both the registry itself and the messaging platform in use.

The schema management component maintains persistent storage of the schemata in an external storage medium, which could be a database or a cache equipped with persistent object storage. Per design, most interactions with the registry occur during the startup phase of applications. Hence, they are relatively infrequent during the actual processing of messages. Therefore, the primary objective of this storage solution is not focused on speed, as individual applications can proactively cache the unique schemata locally.

GET `/schemas/:id`
Returns the schema associated with the unique SchemaID of the schema.

Request
Path Parameters:

- **id** *(Integer)* The unique SchemaID of the schema.

Response
Body (JSON):

- **type** *(String)* The type of the schema, e.g. `AVRO`, `PROTOBUF`, ...
- **schema** *(String)* The string representation of the schema identified its unique SchemaID.

3.4.2 Contract Management

The *contract management* endpoints grant producers and consumers the ability to oversee their contracts. As outlined in Section 3.2 each contract possesses a unique identifier in the form of its distinct name. Significantly, the contract version is a consecutive sequence of numbers mapped to the related SchemaID of each particular contract version.

The contract management component also necessitates the persistent storage of the contracts. The requirements for the storage solution mirror those detailed in Section 3.4.1.

GET `/contracts/:name`
Returns the contract identified by its unique contract name.

Request
Path Parameters:

[8] https://avro.apache.org/docs/1.11.1/

- **name** *(String)* The contract name.

Response

Body (JSON):

- **name** *(String)* The contract name.
- **channel** *(String)* The channel name with which this contract is associated.
- **compatibility** *(String)* The compatibility strategy of the contract as outlined in Section 3.3.1, e.g. `BACKWARD`, `FORWARD`, ...
- **version** *(Integer)* The current contract version.

POST `/contracts/:name`

Register a new schema, essentially create a new contract version. The returned SchemaID can be used to get the *level-1 normalized*(cf. Section 3.3.2) schema via the `GET /schema/:id` endpoint.

Request

Path Parameters:

- **name** *(String)* The name of the contract.

Query Parameters:

- **normalize** *(Boolean, default=false)* Activates a *level-2 normalization* process, as defined in Section 3.3.2.

Body (JSON):

- **type** *(String)* The type of the schema, e.g. `AVRO`, `PROTOBUF`, ...
- **schema** *(String)* The string representation of the schema identified its unique SchemaID.

Response

Body (JSON):

- **id** *(Integer)* The unique SchemaID.
- **version** *(Integer)* The contract version for this schema.

POST `/contracts/:name/versions`

Check if a schema has already been registered in a contract. If a contract version exists, it will be returned with the associated SchemaID.

Request

Path Parameters:

- **name** *(String)* The contract name.

Query Parameters:

- **normalize** *(Boolean, default=false)* Activates a *level-2 normalization* process, as defined in Section 3.3.2.

Body (JSON):

- **type** *(String)* The type of the schema, e.g. `AVRO`, `PROTOBUF`, ...
- **schema** *(String)* The string representation of the schema identified its unique SchemaID.

Response

Body (JSON):

- **id** *(Integer)* The unique SchemaID.
- **version** *(Integer)* The contract version for this schema.

GET `/contracts/:name/versions/:version`

Returns the associated SchemaID, which is uniquely identified by the contract name and the contract version.

Request

Path Parameters:

- **name** *(String)* The contract name.
- **version** *(Integer)* The contract version of the contract.

Response

Body (JSON):

- **schemaID** *(Integer)* The SchemaID associated with the contract version.

POST `/contracts/:name/compatability`

Sets the compatibility strategy of this contract.

Request

Path Parameters:

- **name** *(String)* The contract name.

Body (JSON):

- **compatibility** *(String)* The compatibility strategy of the contract defined in Section 3.3.1, e.g. `BACKWARD`, `FORWARD`, ...

Response

Body (JSON):

- **success** *(Boolean)* Return `true` if the compatibility strategy was successfully set.

3.5 Schema Metaprogramming

The concept illustrated in Figure 3.8 establishes a strong coexistence between the schema and the business logic of the application. Conventionally, it has been common practice to store schemata as text files within application resources and subsequently generate data classes from these files during development. However, this integration is highly manual, often requiring ongoing background processes for file generation. Consequently, this approach imposes challenges due to the existence of two representations of the technically identical schema. Furthermore, a lack of developer experience is evident.

In contrast, leveraging *metaprogramming* (cf. Section 2.1.5) offers the advantage of developing the schema alongside the data class within the same file. Consequently, any schema modifications become immediately apparent in the application code, allowing for early detection of potential complications. This integration proves especially beneficial in scenarios involving schema evolution.

Additionally, during the development phase, a language linter plugin could assess the compatibility between the client's schema against the schemata stored in the registry. Thus, this deep integration into the development environment could help identify compatibility issues in an early development stage.

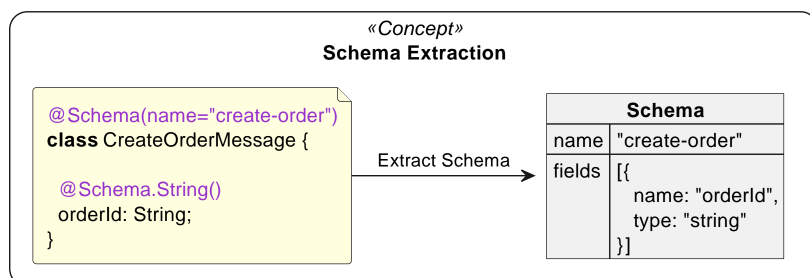

Figure 3.8: Concept: Schema extraction with metaprogramming.

The optimal implementation of this pattern involves annotating classes with class-level annotations, as exemplified in Figure 3.8. These class annotations serve as a metaobject for essential metadata related to the respective schema, including schema attributes like the name, type, or descriptions.

For schema fields, annotations are applied directly to the fields within the data class. For instance, a schema field of type `'string'` would correspond to a decorated `String` field within the data class. In this context, annotations can also be used to define metadata such as descriptions, constraints, or default values.

Furthermore, these metaprogramming concepts can be extended to support dynamic metaprogramming during runtime. As discussed in Figure 3.8, a data class that has had its schema extracted can be dynamically tagged with its SchemaID at runtime.

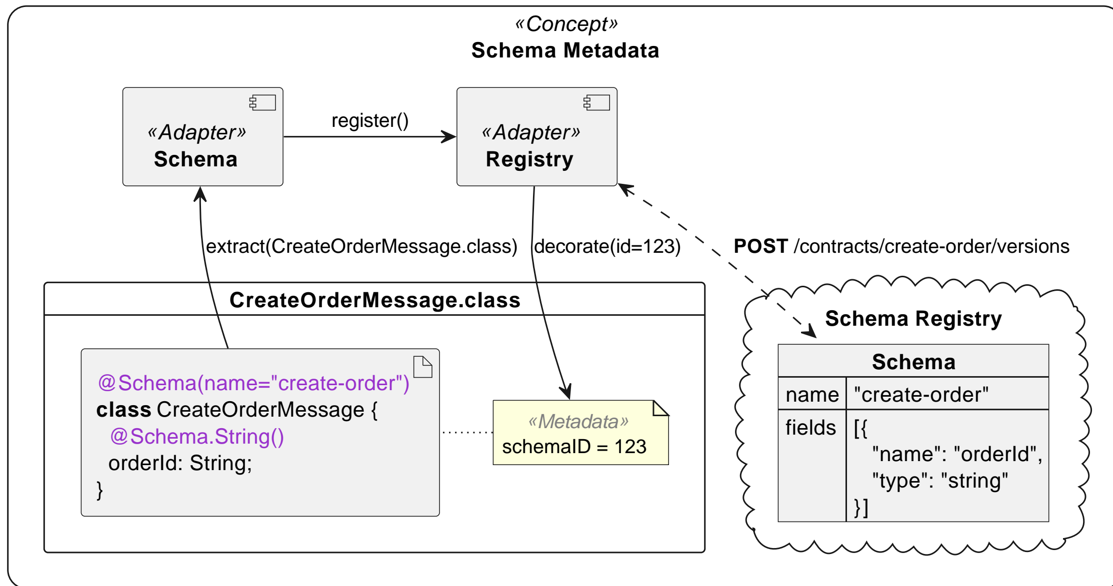

Figure 3.9: Concept: Decorating the `SchemaObject.class` with metaprogramming.

This concept is schematically illustrated in Figure 3.9. To enable this type of runtime metaprogramming, two adapters, the schema adapter (cf. Section 4.2) and the registry adapter (cf. Section 4.3), come into play. Through the interaction with these adapters, the class can be annotated at runtime with crucial metadata, that is only known by the registry, such as the SchemaID or the contract version. This metadata can then be used to identify the correct schema within the registry.

3.6 Message Encoding

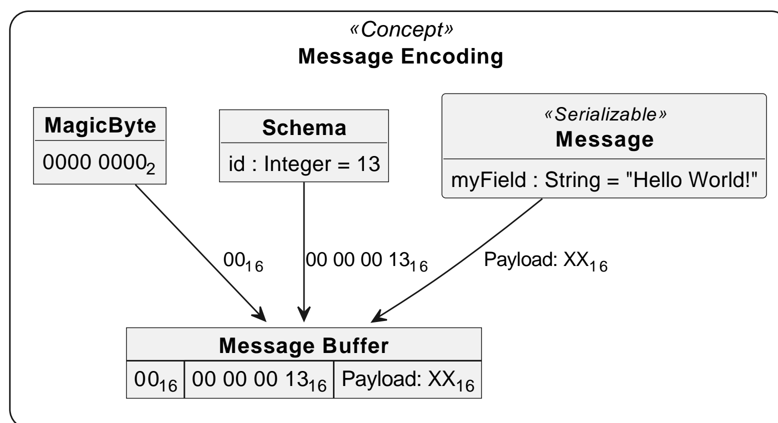

Figure 3.10: Concept: Unified message encoding format.

Figure 3.10 illustrates the concept of *message encoding*, which should serve as the cluster-wide unified message format. Consequently, by encoding the SchemaID together with the payload into a single buffer, we ensure that a transmitted message carries all the essential information required for future processing.

The design of this format intentionally maintains simplicity and aligns with previously established formats, such as those used by Confluent. Its structure commences with a magic byte as the initial marker, followed by the unique SchemaID integer represented in the four following bytes. Subsequently, the payload, serialized according to the schema, follows suit [Con23b]. This encoded message can also be referred to as a *type-consistent message*.

3.7 Validation

The concept of the *schema validator* stems from the generally employed paradigm of validators. A particular schema initializes the validator, which can subsequently ascertain whether a dataset complies with the specifications of that schema or not. This concept is illustrated schematically in Figure 3.11.

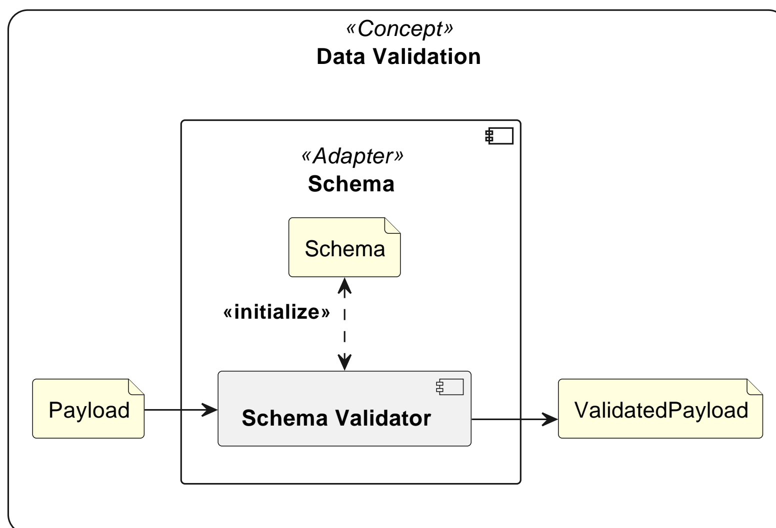

Figure 3.11: Concept: Data validation.

When the validator paradigm is applied to a messaging platform, two key concepts emerge where a system can conduct schema validation of message payloads. The optimal moment for validation would be right when the broker receives a message. However, this approach presents certain drawbacks, making client-side validation, performed by participating producers and consumers, a potentially preferred alternative. These two approaches are discussed in the following sections.

3.7.1 Validate with Broker

The broker stands as the central component within a messaging platform. It is responsible for accepting and distributing messages. Consequently, it represents the ideal point for message validation. Therefore, the type-consistency of messages could be assessed before it is accepted by the broker. In cases of invalid messages, the broker can reject them, providing the producer with information regarding the validation error. This approach offers enhanced consistency and reliability but places greater resource demands on the broker and poses compatibility challenges with existing solutions. Furthermore, the broker must establish a direct connection with the registry and integrate logic for the validation of each schema type. However, this approach can often be impractical, impeding the broker's primary function of swiftly distributing messages.

Furthermore, the approach of this thesis is to design a BITCMT Framework. Consequently, the aforementioned approach isn't compatible, as it necessitates substantial broker involvement, which is contrary to the framework's core principle of broker independence.

3.7.2 Validate with Client

In contrast to validation through the broker, outsourcing the validation to the participants is a further strategy. The messaging architecture typically guarantees that individual clients are highly scalable. Given that these clients are responsible for serializing the messages, they often already incorporate the necessary logic for validation within this process.

In cases where type-consistent producers generate each message, one can reasonably assume that all messages within such a system are type-consistent. Given this assumption, there may theoretically be no need for consumers to revalidate messages since they are already inherently type-consistent. Nevertheless, it is essential to consider scenarios where messages could be manually created or altered by external systems. Hence, it is advisable to conduct revalidation during the deserialization process by the consumer as an added layer of assurance.

Overall, this approach offers greater flexibility and aligns more seamlessly with existing solutions. Additionally, it ensures that the broker is not further burdened by additional validation processes. Moreover, this approach is compatible with the proposed BITCMT Framework, as it does not require any broker involvement.

3.8 Message Endpoint

In a type-consistent messaging system, it is important to note that the consumer cannot serve as the direct interface to the application's business logic. This is because it does not inherently enforce type-consistency, as discussed in Section 2.1.2.

Furthermore, the responsibilities of a single consumer extend to receiving messages from various channels and handling multiple messages from diverse messaging contracts. To address this complexity, this work introduces a new layer above the consumer level, the *messaging endpoint*. The fundamental principle underlying this concept is that a single message endpoint is designed to exclusively receive messages from a specific channel with a particular contract version.

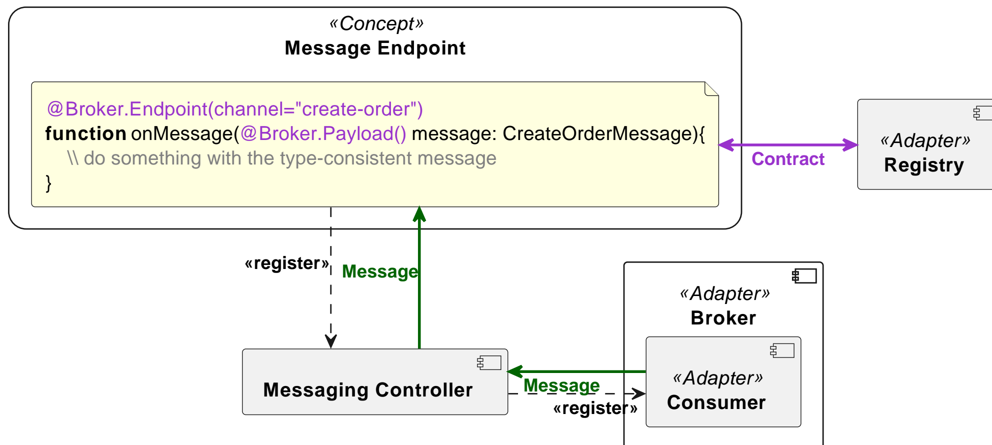

Figure 3.12: Concept: Message *Endpoint* and *Controller*.

Figure 3.12 illustrates the structure of a messaging endpoint. Another integral component in this concept is the controller, responsible for managing the endpoints. Consequently, the endpoints are registered within the controller, which, in turn, is registered with the consumer adapter (cf. Section 4.4.2).

Given that the messages are already deserialized and validated by the consumer, they can also be regarded as type-consistent. Therefore, it becomes the responsibility of the consumer to invoke an endpoint only if it corresponds to the same contract version. However, in cases where no endpoint has been defined for a particular contract version, the controller should raise an error, which has to be handled by the application. Alternatively, the controller should use compatibility strategies (cf. Section 3.3.1) to determine an appropriate endpoint.

The process of determining the specific contract and its corresponding contract version is facilitated through the registry adapter (cf. Section 4.3), which establishes a direct connection to the registry. As mentioned in Section 3.2, the precise contract version can be derived from the combination of the contract name and the associated schema.

To streamline this information retrieval process, a metaprogramming approach (cf. Section 2.1.5) utilizing annotations is ideally suited. Thereby, the schema and contract data can be directly extracted from the decorated endpoint. Subsequently, this metadata can be employed to correctly register the endpoint within the controller.

4 Architecture

Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework), as the name suggests, is the framework architecture proposed by this thesis. It represents a modular approach to achieving a messaging paradigm that is both broker-independent and type-consistent. The subsequent sections offer an overview of a potential architecture of the BITCMT Framework. Which is a composition of concepts previously outlined in Chapter 3 into a tangible client application. However, within the scope of this work, it is nearly impossible to describe the many details and edge cases of the application solely through text and diagrams. Consequently, the architecture components presented in this chapter should serve as nothing more than a foundational framework architecture template for a more detailed implementation, as outlined Chapter 5.

Similar to the preceding chapter, this chapter commences with an overview in Section 4.1. This overview encompasses aspects such as the structure of individual packages and the resources present in the messaging cluster. In the following three sections, we delve into the adapters [GHJV95] as mentioned in Chapter 3: Section 4.2 deals with the schema adapter, serving as the bridge between the application and the schema. Section Section 4.3 concentrates on the integration of the registry into the application context. Lastly, Section 4.4 defines the broker adapter, functioning as the interface between the application and the broker. In Section 4.5, the three adapters are further merged into the concept of a messaging client.

Subsequently, Section 4.6 and Section 4.7 outline how these individual classes and components interact. Through sequence diagrams, these sections provide insights into the instantiation processes, as well as the execution sequence during message production and consumption.

4.1 Architecture Overview

First, Figure 4.1 we present the structural hierarchy and dependencies among the individual packages of the BITCMT Framework. A crucial consideration in this architectural design is to maintain a high level of compatibility and individual package substitutability. Consequently, there should be no direct horizontal dependencies between the individual packages. This architecture allows for the possibility of replacing the `broker-kafka` package with another package like `registry-rabbitmq`. The individual components are thus intentionally designed to be strongly decoupled, with transitive dependencies existing only through interfaces within the `common` package. Consequently, the `common` package serves as the central package compatibility interface.

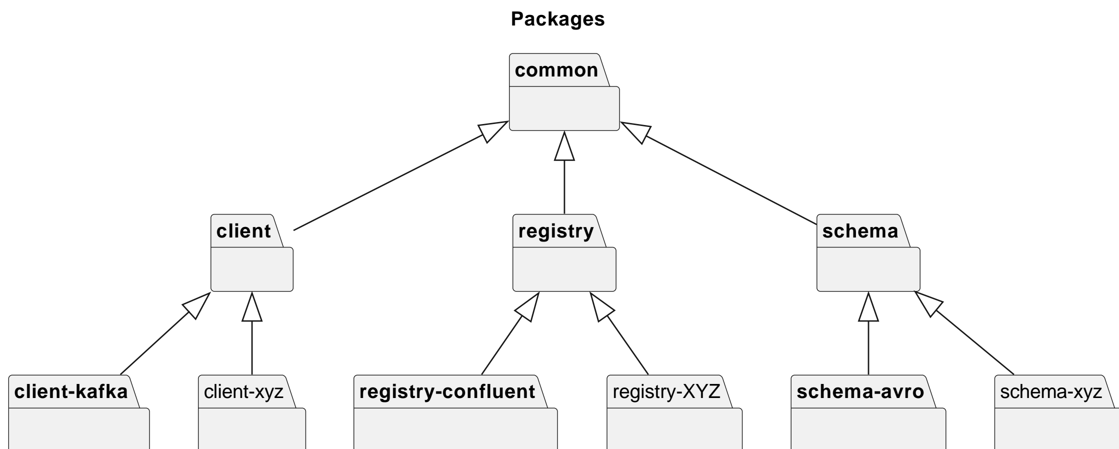

Figure 4.1: The package hierarchy of the architecture.

At the underlying level, the architecture ensures that all packages of a specific type, whether it is client, broker adapter, or schema adapter, share a common abstract foundation. This layer establishes the essential groundwork, including the implementation logic for automated caching, among other functionalities.

The bottom layer functions as the dependency-specific implementation. In essence, it represents the concrete realization of the platform-specific messaging logic.

On the other hand, Figure 4.2 illustrates the overview of the individual resources within the client application. In this context, these resources refer to remote instance objects in the broker or registry, which are replicated by the application for consumption by various components or concepts. These connections between resources unveil certain dependencies. For instance, a contract version cannot exist without its contract, and a message is inherently dependent on its channel. It is also illustrated that the schema can be extracted from the message, and following of schema registration process, the message can be serialized using the normalized schema. In summary, the diagram illustrates the direct and independent dependencies among resources within the application.

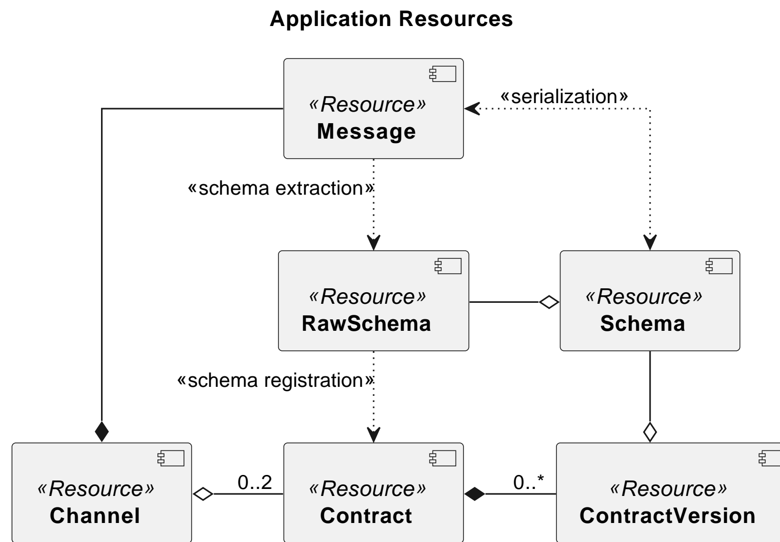

Figure 4.2: Class diagram of all resource types within the application scope.

4.2 Schema Adapter

The schema adapter serves as the pivotal connection point between the metaprogramming strategy outlined in Section 3.5 and the BITCMT Framework. Consequently, the adapter encapsulates all the business logic essential to deliver the schema functionalities as mandated by the framework design. In Figure 4.3, a more detailed structural representation of the schema is presented through a class diagram.

The `SchemaObject` is a class that has been decorated with the necessary annotations to enable the extraction of a `RawSchema` from it, employing the concept outlined in Section 3.5. Its primary function of the `SchemaObject` is to serve as a placeholder for any suitable class while also promoting additional type restrictions. It is the task of the BITCMT Framework to determine whether a `SchemaObject` is decorated and whether a `RawSchema` can be extracted via metaprogramming. Additionally, the schema-specific package must implement a schema factory [GHJV95]. This factory takes the `RawSchema` as input and generates a schema adapter.

Thus, if a `SchemaObject` can be successfully extracted this process results in a `RawSchema` interface. The `RawSchema` is a combination of the schema type, exemplified in Figure 4.3 as `'AVRO'`, and the raw schema representation, which, in the case of Avro, is in JSON[1] format [Con23b]. The actual schema adapter, which includes not only its schema type but also its unique Schema Identification Number (SchemaID), is created when the raw schema is registered in the Registry. With the assistance of external dependencies, the schema-specific implementation facilitates the `AvroSchemaFactory`, which can convert a `AvroRawSchema` into an `AvroSchema`. This enables the `AvroSchema` class to possess the logic required for validation, serialization, and deserialization of its associated `AvroObject`.

[1] `https://www.json.org/`

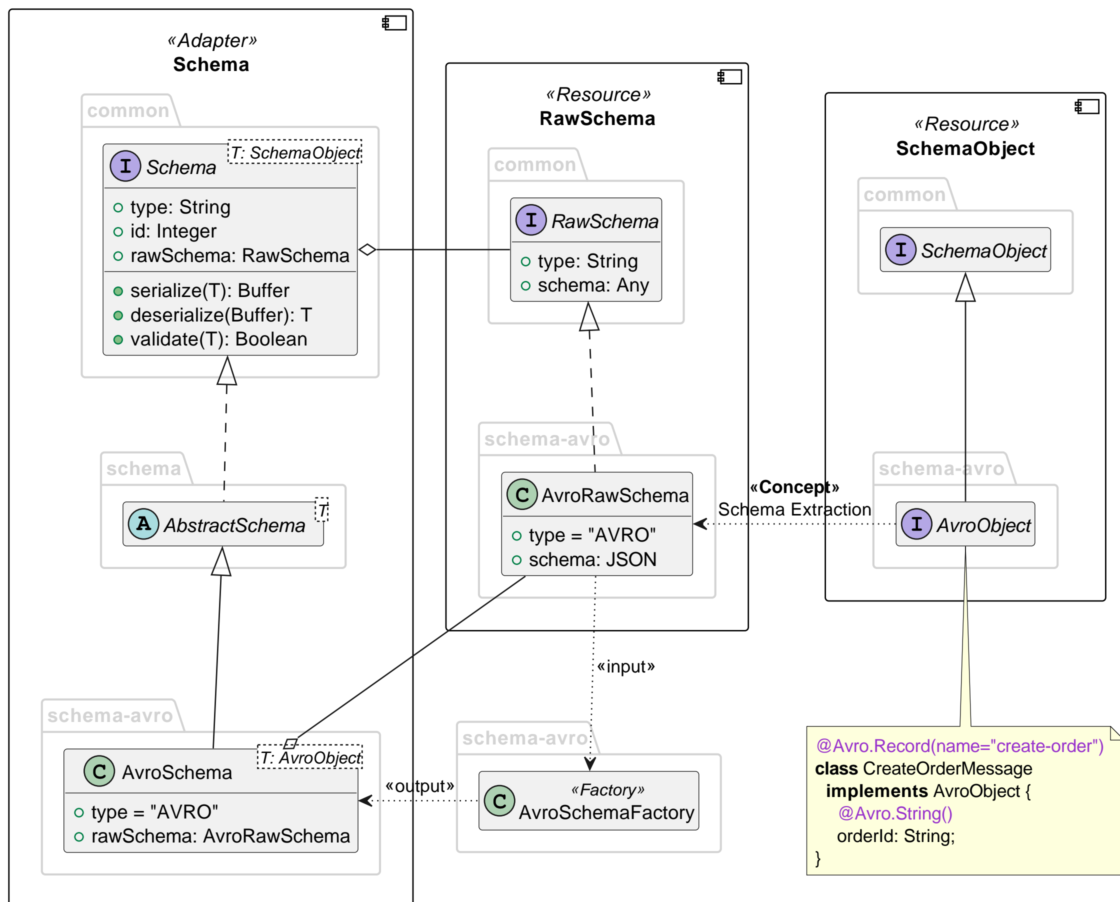

Figure 4.3: Class diagram with package hierarchy of the *Schema Adapter*.

4.3 Registry Adapter

The *registry adapter* serves as the intermediary between the external registry distributed in the cluster and the client of the BITCMT Framework. Given that the resources of the registry are distinct, they suit themselves well to an optimistic caching pattern. As a result, the registry-specific adapter only needs to incorporate the logic for retrieving the schema, the contract, and contract version from the remote registry. Figure 4.4 further illustrates the structure of the registry adapter.

Following the retrieval of the `RawSchema` from the remote registry, registry adapter is responsible for generating the schema-specific schema adapter. As partially outlined in Section 4.2, a factory [GHJV95] that can produce a `Schema` from a `RawSchema` is particularly well suited. These factories must be initially registered in the registry adapter during framework initialization.

In line with the schema management outlined in Section 3.4.1, the registry adapter primarily plays a minor role in schema management. The responsibilities for schema registration and the management of schemata are primarily handled by the contract architecture, as detailed in the following Section 4.3.1.

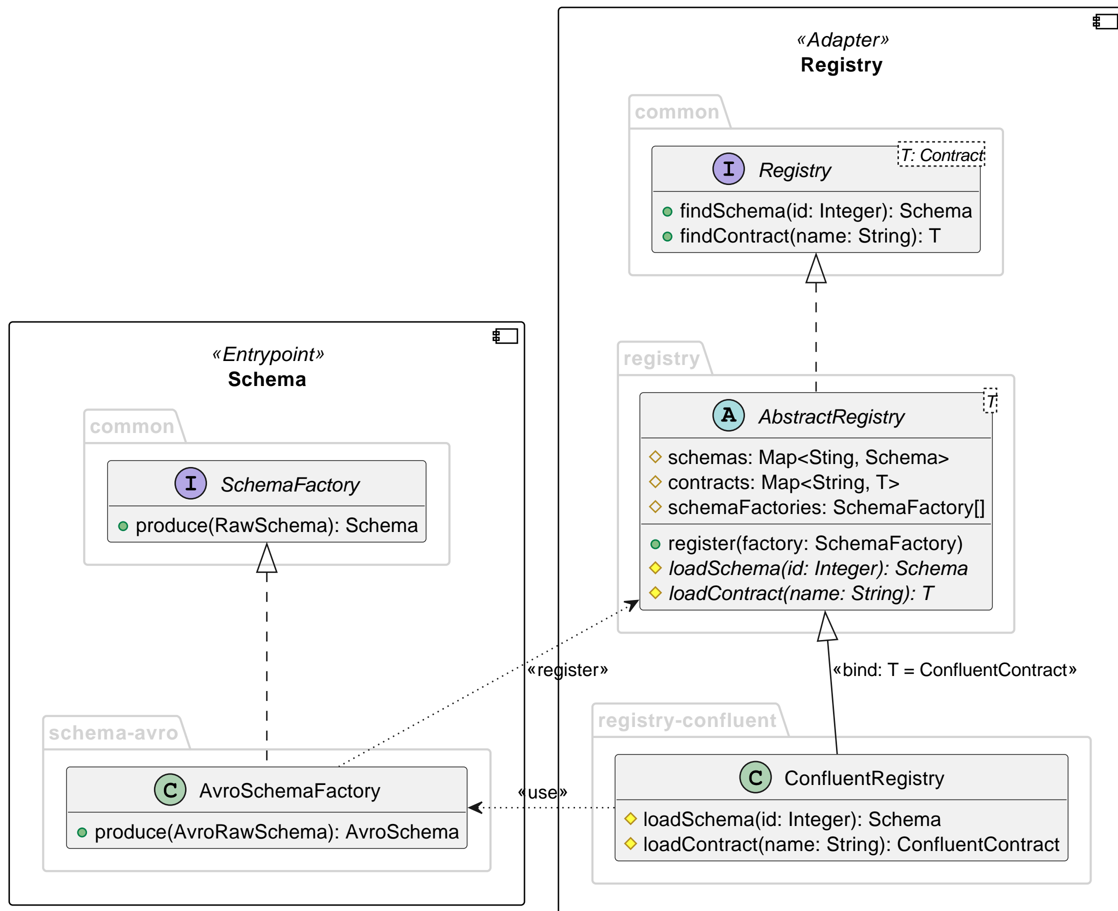

Figure 4.4: Class diagram with package hierarchy of the *Registry Adapter*.

4.3.1 Contract

The primary resource managed by the registry adapter is the concept of a messaging contract, as defined in Section 3.2. The contract architecture within the BITCMT Framework adheres to the principles outlined in Section 3.4.2. The simplest resource in this architecture is the `ContractVersion`, which is essentially a data class. It contains only the contract version number, the associated schema adapter, and the contract to which it belongs, with no further logic behind this structure. Due to its straightforward design and its unique fields, this data class is particularly well-suited for caching within the application. Consequently, a specific contract version and its associated schema need only be retrieved once from the remote registry.

Conversely, a substantial portion of the central interface of contract management is allocated to the registry-specific `Contract` classes. The `RawSchema` functions as the medium for transmission. It is only after the registration process, i.e., when the schema is loaded by the registry adapter, that it can be guaranteed that this `RawSchema` is normalized and meets all compatibility requirements. Indeed, the architecture is intentionally designed to disallow the explicit use of schemata for messaging without first subjecting them to validation by a remote registry. This approach ensures that each schema employed in the messaging process has undergone verification and complies with the

compatibility requirements. Furthermore, the registry-specific implementation must support the functionalities imposed by the corresponding Application Programming Interface (API) endpoints as described in Section 3.4.2.

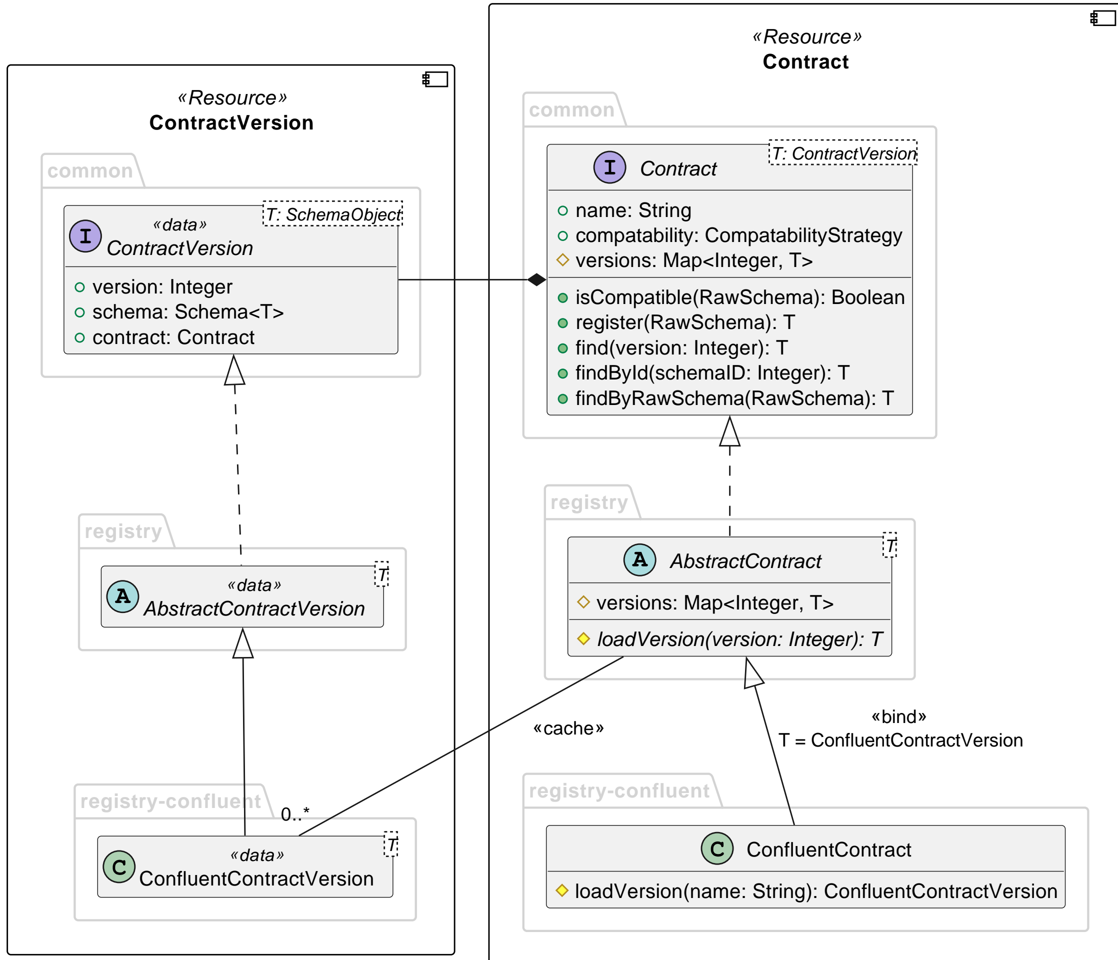

Figure 4.5: Class diagram with package hierarchy of the *Contract* and *Contract Version*.

4.4 Broker Adapter

The broker adapter serves as the connection to the remote broker within the messaging cluster. Its key responsibilities include the loading of the channels (cf. Section 2.1.2), along with the creation of producer adapters (cf. Section 4.4.1) and consumer adapters (cf. Section 4.4.2).

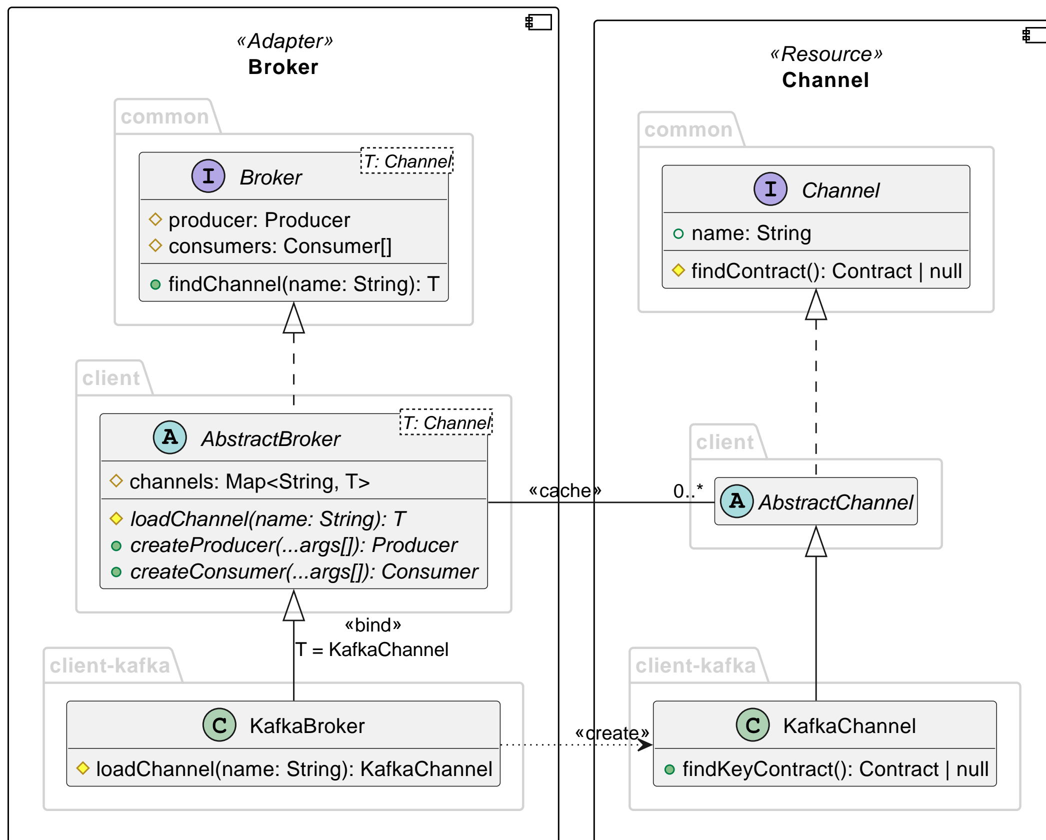

Figure 4.6: Class diagram with package hierarchy of the *Broker Adapter*.

The distinctive feature of the channel is that, in addition to its name, it also includes the functionalities to find its contract. Leveraging the uniqueness of the channel name, it is also feasible to load the channel and its associated contract just once. Therefore, an optimistic caching strategy, as illustrated in Figure 4.6, can be implemented. Overall, it can be concluded that the loading of the composition of channel, contract, contract version, and schema should occur just once, by fetching them from the remote registry and remote broker into a cached the application context.

4.4.1 Producer Adapter

The *producer adapter* represents the common interface for sending messages. As Figure 4.7 illustrates, the architecture is minimal. Through the extension of the `Connectable` interface the producer adapter must allow for opening and closing connections with the broker. When reduced

down to its core functionalities, the producer adapter is tasked with only one additional responsibility, the message production and transmission to the broker. However, this logic is determined by the broker-specific implementation of the broker adapter.

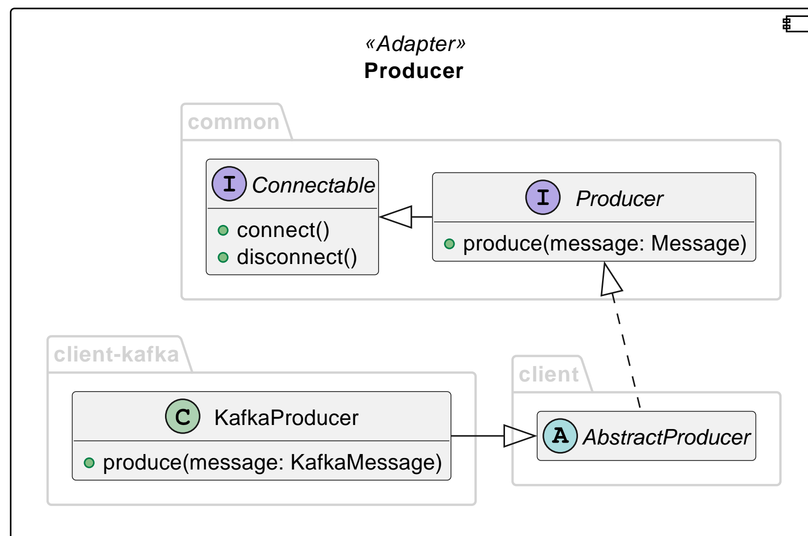

Figure 4.7: Class diagram with package hierarchy of the *Producer Adapter*.

4.4.2 Consumer Adapter

The *consumer adapter* has significantly greater responsibilities within this type-consistent architecture. Due to the concept outlined in Section 3.8, the consumer adapter no longer serves as the interface to the application logic but acts as a mediator for all controllers and message endpoints. The structure of the consumer adapter is depicted in Figure 4.8 as a class diagram. Similar to the producer adapter, the consumer adapter must implement the `Connectable` interface. In addition, the consumer can subscribe to a specified channels, effectively receiving future messages from these channels. Depending on the broker-specific implementation, such as Kafka, the subscription can only be initialized if the consumer adapter does not have an open connection with the broker adapter.

However, the most crucial task of the consumer adapter is to receive and distribute messages to the controllers that have been registered previously. Since the relationship between consumer adapter and controller follows a fundamental observer pattern [GHJV95], the controller should decide whether it can further process a given message. It is vital to note that within this observer pattern, there must be a mechanism in place to ensure that a message has been handled by at least one endpoint. Furthermore, if the controller cannot find a suitable endpoint, it must not commit the message (cf. Section 4.5.3).

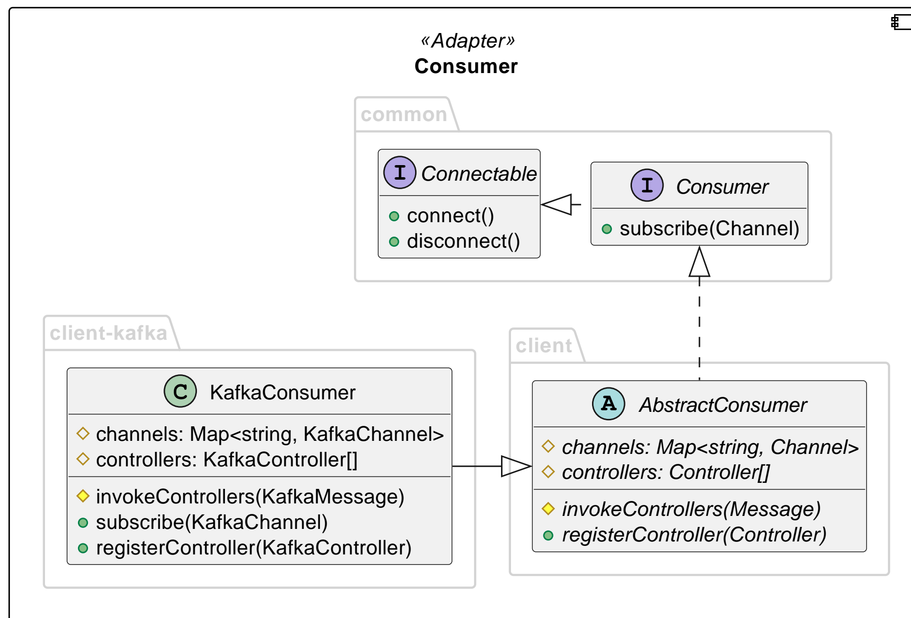

Figure 4.8: Class diagram with package hierarchy of the *Consumer Adapter*.

4.5 Client

The *client* functions as the component that orchestrates and unifies all elements of the BITCMT Framework. It holds instances of both the broker adapter and the registry adapter, which, in turn, holds instances of schema adapters. Thanks to the vertical transitive integration between these packages, many instances can be effectively reused. For example, the registry adapter instance can be shared among multiple clients, enabling a specific schema with a given SchemaID to be loaded into the application context only once. Consequently, this architecture facilitates the use of schemas across multiple client instances.

Additionally, the client is tasked with managing individual controllers. Initially, a controller is essentially the decorated class as described in Section 4.5.2, and it is solely instantiated by the client. Once this controller instance is initialized, the client registers it with the associated consumer adapter. Since the controller specifies which channel the consumer adapter has to subscribe to, and a consumer needs to rebalance with the broker whenever subscription changes occur, dynamic initialization during runtime is not feasible. Consequently, the controller must be statically registered in the client during framework initialization.

In addition to these functionalities, the client also incorporates the default producer adapter of the broker adapter for basic message production capabilities. At this stage, the client possesses all the essential information to independently determine the channel, the contract, and, the contract version. Therefore, it is sufficient to provide the channel's name and an instance of a `SchemaObject` to the client. With the channel name, the client locates the associated channel via the broker adapter. The contractcan also be determined using the channel name through the registry adapter. The contract version can be identified through the resource contract and the runtime-decorated metadata of the `SchemaObject`. The contract version also includes the SchemaID, which enables querying the

registry adapter for the associated schema adapter. With all these resources in hand, the message can finally be encoded and dispatched. Given the complexity of this process, it will be revisited in Section 4.6 and illustrated with the support of sequence diagrams.

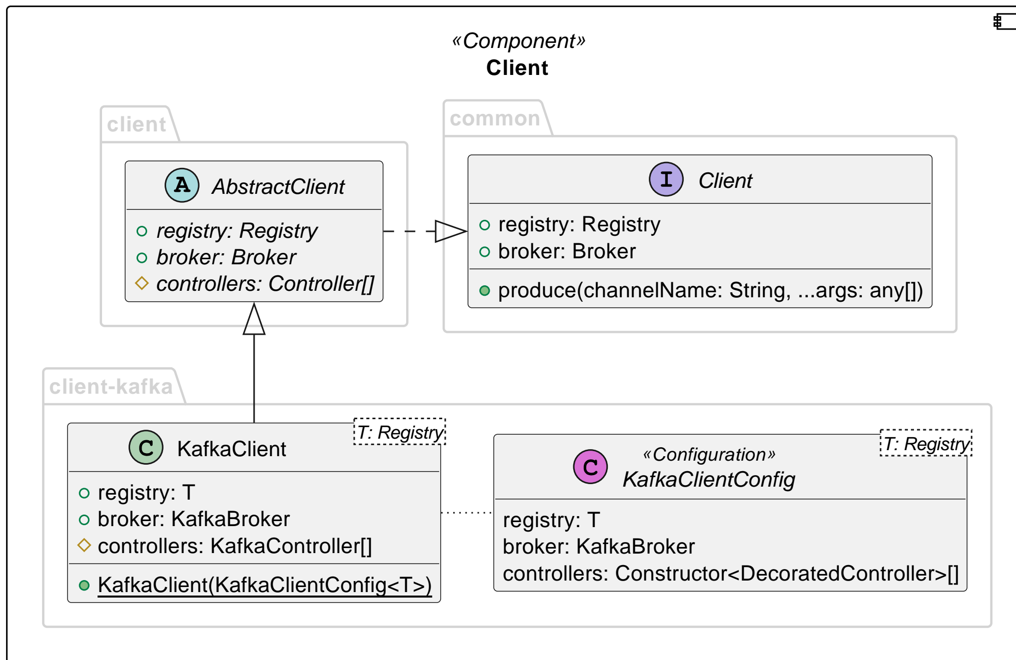

Figure 4.9: Class diagram with package hierarchy of the *Client*.

4.5.1 Message

The BITCMT Framework requires the message to be sharable across its modular components. To achieve this, a central interface is indispensable, one that can uniformly describe a message. A review of existing solutions indicates that each message comprises at least three abstract components (cf. Section 2.1.2). The first component is the *channel*, specifying the communication route over which the message is transmitted. The second component is the *payload*, signifying the content of the message. Moreover, if the payload can be associated with a `SchemaObject`, the entry also contains the corresponding *contract version*. The final component is the *message metadata*, which encompasses additional meta-information about the message. This metadata may include headers, the partition number, or the message offset, among other elements. It's crucial to note that the metadata is defined at the broker-specific level and does not play any role in achieving type consistency.

Furthermore, inheritance offers a means of achieving broker-specific extensibility. For instance, in the context of Apache Kafka, as illustrated in Figure 4.10, there is also the option to include a message key in addition to the payload1 [ZK21].

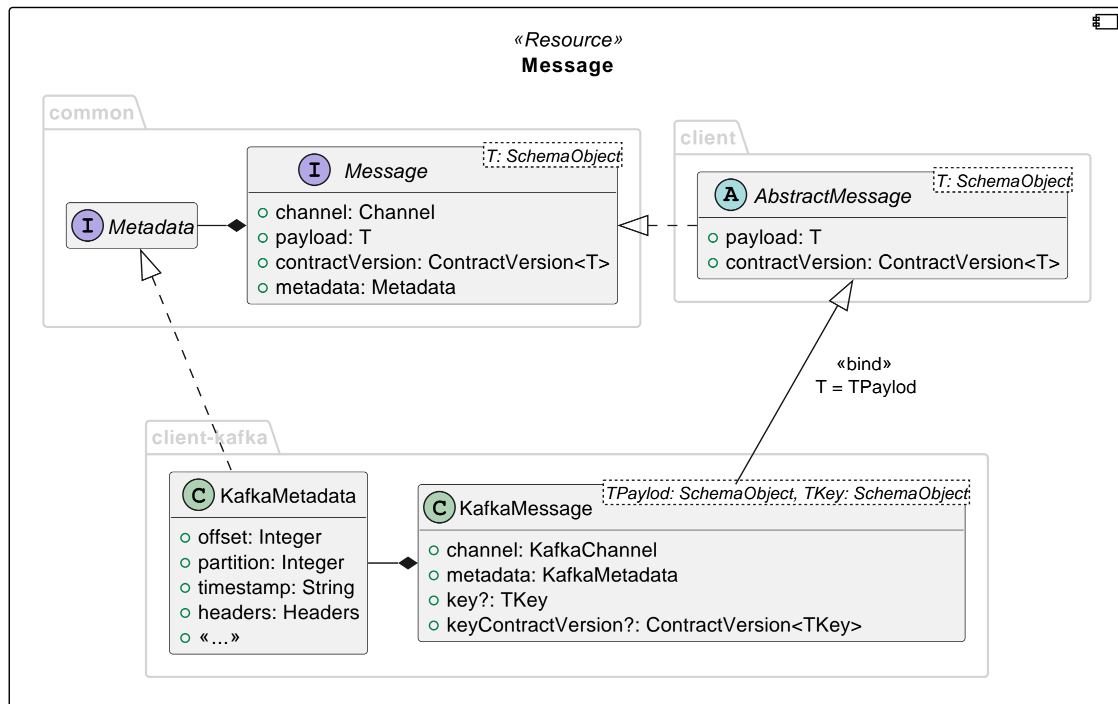

Figure 4.10: Class diagram with package hierarchy of the *Message*.

4.5.2 Controller

As discussed in Section 2.1.2, in a type-consistent messaging system, the consumer can no longer serve as the interface between the received message and the application's business logic. To address this, this work introduced the novel endpoint concept in Section 3.8. However, an additional level is necessary to facilitate communication and management between the consumer and the endpoint. This newly created communication level is the controller.

The controller serves as an intermediate consumption level, undertaking two crucial responsibilities. Firstly, it handles the reception and distribution of messages, ensuring they reach the appropriate endpoint. Secondly, it manages errors produced by the BITCMT Framework and its endpoints.

Figure 4.11 illustrates the structure of the controller as a class diagram. The controller maintains a reference to its consumer, which serves as the source of its messages. Additionally, it includes a field for the name of the decorated class to which it supplies messages. During the initial prototyping phase, metaprogramming has proven to be a suitable method for defining controllers. A factory [GHJV95] pattern is employed to transform these decorated classes into controllers. Moreover, the BITCMT Framework automatically generates and integrates controllers, reducing the likelihood of errors and enhancing the developer experience. Consequently, a controller can be reused and instantiated by multiple clients, further enhancing the framework's versatility.

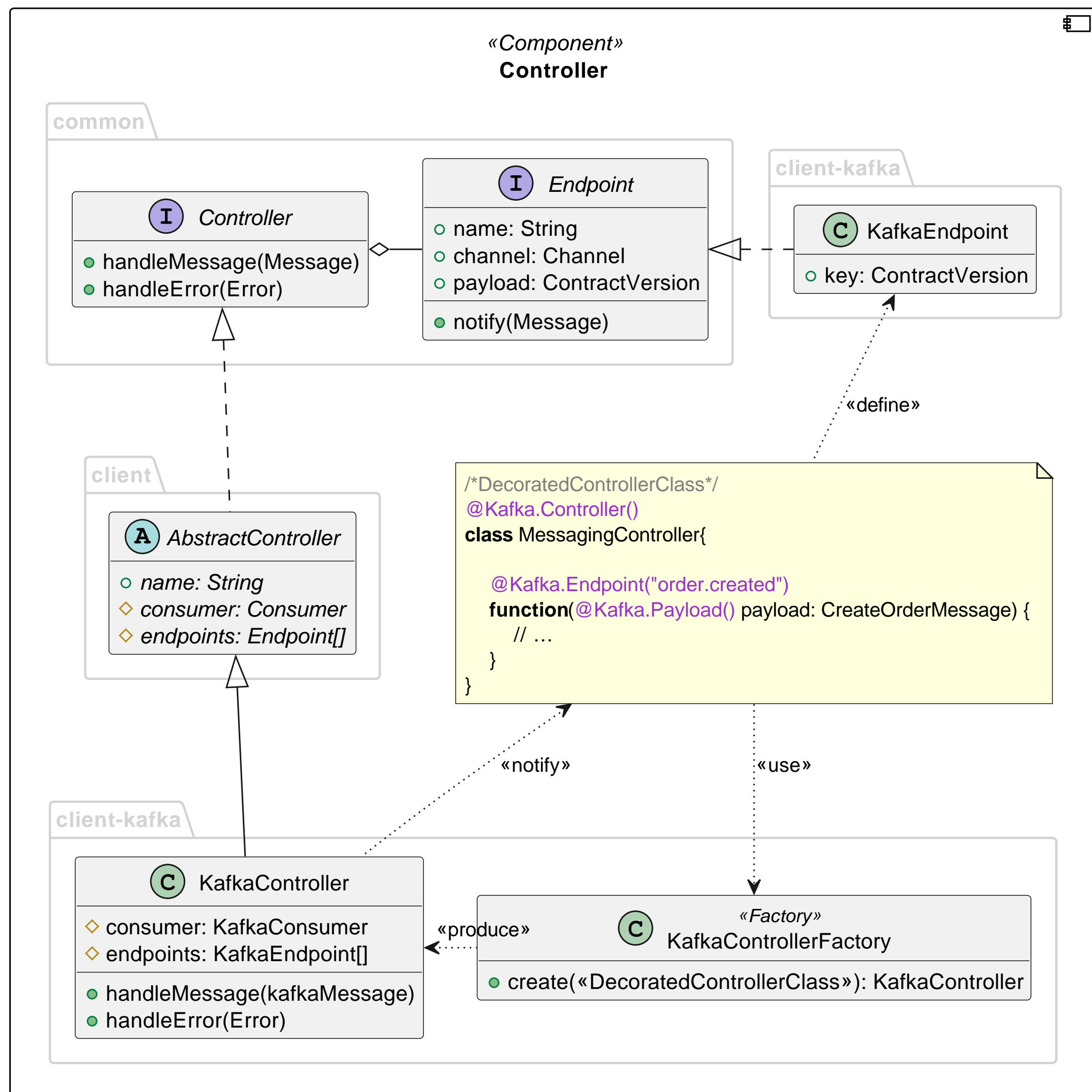

Figure 4.11: Class diagram with package hierarchy of the *Controller*.

4.5.3 Endpoint

The annotations illustrated in Figure 4.11 are intended to serve as a template for a broker-specific implementation. These annotations play a critical role in defining the channel and contract version of the endpoint, they are crucial for ensuring type-consistent message transmission.

Ideally, the channel can be defined through a method annotation on the endpoint function. If the parameter representing the message payload is an annotated `SchemaObject`, the contract version can be derived from its metadata. Thereby, when the endpoint has to be notified, the controller also knows at which parameter index it has to insert the payload. This way, when the endpoint needs to be notified, the controller also knows the parameter index at which to insert the payload.

It is important to note that such implementations are highly broker-specific. Therefore, this work can only be abstractly defined that before the controller can notify an endpoint of a new message, it must ensure that this endpoint precisely supports the message's contract version. If the controller cannot find a suitable endpoint, it can algorithmically determine a suitable endpoint using the contract's corresponding *compatibility strategy* (cf. Section 3.3.1). If no matching endpoint is found, further actions must be taken.

Therefore, any unhandled message should be dealt with by an error-handling strategy, which is not in the scope of this work. Common solutions like a *retry queue*, or a *dead letter queue* [Apa23a] could be considered. To be explicit, an unhandled message should never be committed by the consumer adapter. However, it is worth noting that the concept of the endpoint could be extended to encompass a general subscription model, similar to the *channel subscriptions* offered by Apache Pulsar [Apa23a], but for endpoints. This extension would allow for a more fine-grained control over the message distribution, and retry strategies. However, this concept is not further elaborated in this work.

4.6 Message Production

This section focuses on the transmission of type-consistent messages. Specifically, it presents possible executions and interactions of individual components and classes through sequence diagrams. To enhance clarity, many implementation-dependent details, especially the passing of parameters in functions, have been omitted. These diagrams describe error-free executions, assuming that all schemata and contracts are already registered in the registry, either through the application or external capabilities.

To provide insight into the nature of processes they have been color-coded. Synchronous processes are marked in light-green, while asynchronous processes are marked in dark-green. Processes that might involve external network requests are marked in light-red. This distinction highlights that due to caching - following an initialization process - the continuous message processing is mainly synchronous, without involving network requests.

4.6.1 Initialization

After the initialization of the client, Figure 4.12 and Figure 4.13 illustrate the initial, one-time registration, and initialization process that occurs when a `SchemaObject` is first produced by the producer for a designated channel. To accomplish this, a new instance of a `SchemaObject` is initially created within the application.

Suppose an application intends to send messages in a type-consistent manner. In such cases, it necessitates access to several resources, as repeatedly described in this thesis: a channel, a contract, the contract version, and consequently, the schema. Gathering all these resources over the network within the cluster can be time-consuming. Therefore, it is not feasible to perform this process for every message. However, once a channel, contract, or schema has been loaded once, they can be optimistically cached within the application runtime due to their uniqueness.

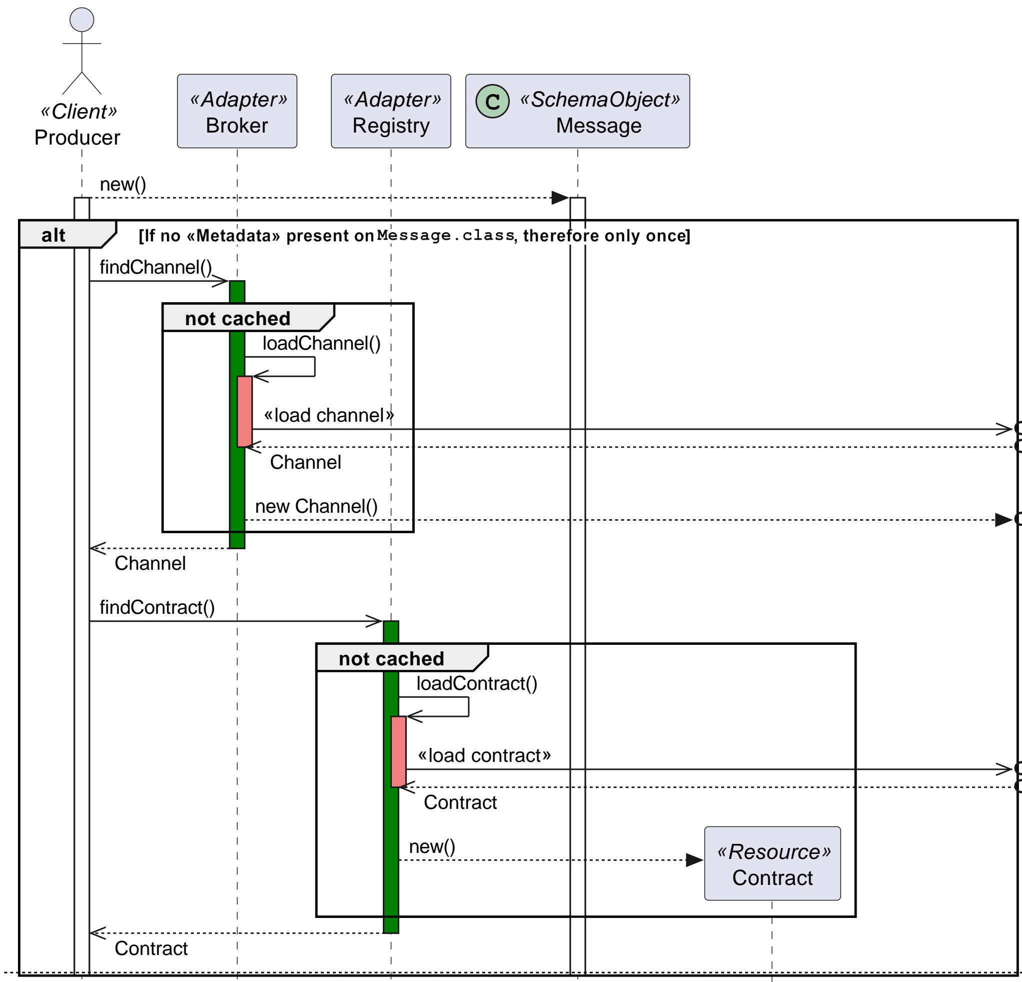

Figure 4.12: Sequence diagram of the initial initialization of a `SchemaObject`. [Part 1/2]

This caching mechanism is extended by decorating metadata to the metaobject of the `SchemaObject.class` that has been transmitted at least once, meaning it has undergone this particular process. Consequently, a `SchemaObject` is tagged, precisely specifying which schema is appropriate for a specified channel and contract. Due to the uniqueness of the schema, conflicts are eliminated. The actual process of annotating a `SchemaObject` with the required metadata can be traced using Figure 4.13. In total, there are five essential steps.

In the initial step, as illustrated in Figure 4.12, the broker adapter is queried to identify the channel for message production. If no such channel is found, the entire process must be aborted. However, it can also be the broker adapter's responsibility to automatically register the channel with the remote broker. This step should always be executed first, as without it, none of the subsequent steps can be executed.

The second step involves, as depicted in Figure 4.12, locating the appropriate contract, or, in the case of Apache Kafka, the contracts associated with channel. However, in Figure 4.12, the presence of only one *SchemaObject* can be assumed, and consequently, only one contract. Hence, from this

point onward, all subsequent steps must be executed individually for each `SchemaObject` or contract. A contract that has been retrieved once from the registry can be optimistically cached by the registry adapter, facilitating synchronous processing for all subsequent requests.

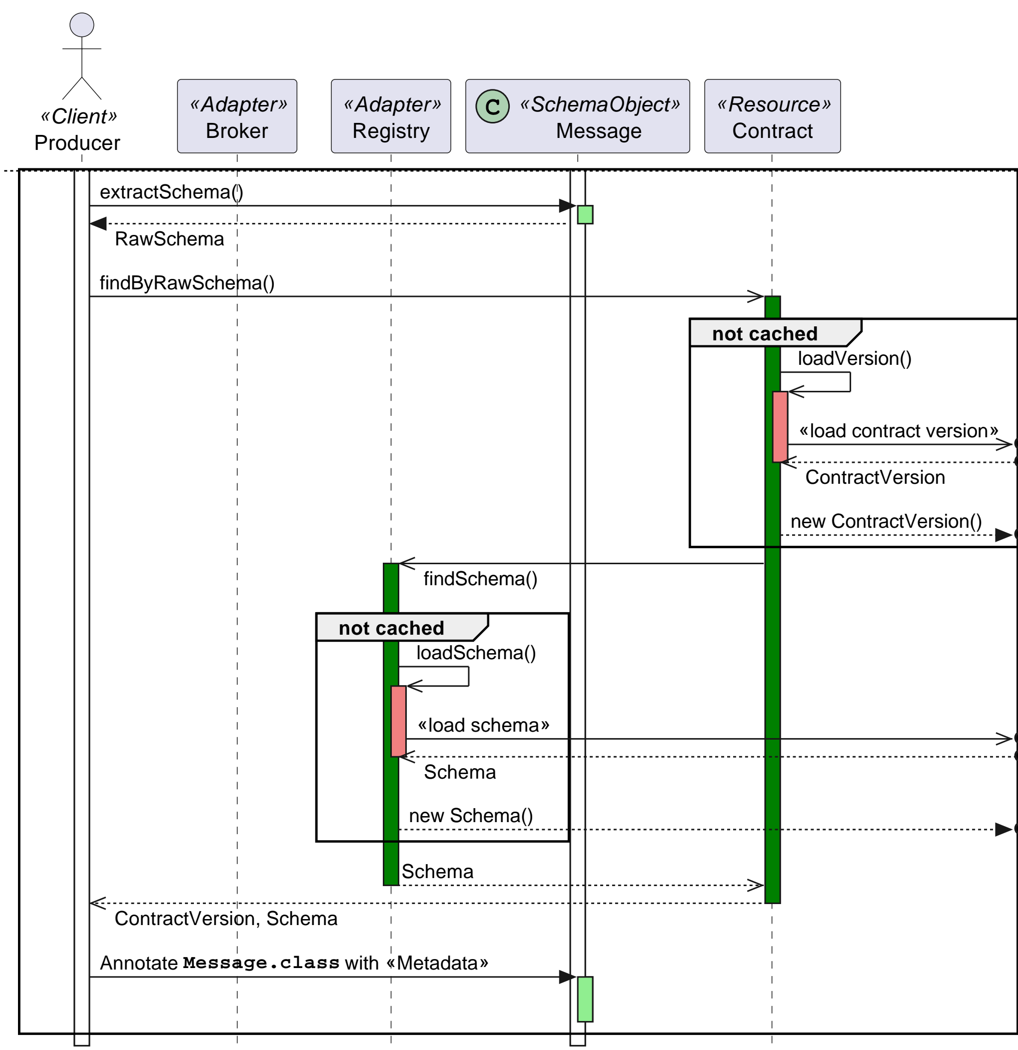

Figure 4.13: Sequence diagram of the initial initialization of a `SchemaObject`. [Part 2/2]

The third step involves, as illustrated in Figure 4.13, the extraction of the schema and is carried out based on the schema extraction concept outlined in Section 3.5. As this process is highly schema-specific, a more detailed architectural explanation is beyond the scope of this thesis. However, the implementation of this concept (cf. Chapter 5) has demonstrated that this process can be executed synchronously. Furthermore, efficiency can be enhanced by performing this operation only once and then decorating the metaobject of the `SchemaObject.class` with the `RawSchema` as metadata.

In the fourth step, as depicted in Figure 4.13, the previously extracted schema is employed to determine the specific contract version, thus loading the validated and normalized schema into the application context. To achieve this, the previously loaded contract is utilized to initially establish

the contract version using the `RawSchema` as a reference. This contract version inherently includes an SchemaID reference to the normalized and validated schema, which can subsequently be retrieved from the registry by the registry adapter. Optimistic caching is also applied in this context, ensuring that each contract version and schema need only be loaded into the application context once.

The fifth and final step involves, as illustrated in Figure 4.13, decorating the metaobject of the `SchemaObject.class` with the associated metadata. The metadata contains information that uniquely identifies this initialization process, enabling the bypass of the initialization in the future. Therefore, it stores information about the registry, the channel, the contract, the contract version, and the SchemaID of associated schema. While these resources are inherently unique within their respective contexts, the BITCMT Framework also permits their usage in several clients and registry adapters. Consequently, an appropriate key for this metadata consists of a generated token of unique identifiers for the registry, the channel, the contract, and the contract version. The SchemaID serves as the corresponding value - a reference to the previously created and cached schema adapter.

To summarize this section, these five steps constitute the complete initialization process for producing messages of a specific contract version within its associated channel.

4.6.2 Production

The actual production and sending of the message occur only after the one-time registration process, as described previously in Section 4.6.1. In this section, it can be assumed that an instance of `SchemaObject` has already been instantiated, and the metaobject of the `SchemaObject.class` has been decorated with the necessary metadata. Consequently, consistent message production follows six sequential steps, as illustrated in the sequence diagram shown in Figure 4.14.

In the first step, the metadata attached to the metaobject of the `SchemaObject.class` during the registration process is loaded into the current execution context. This metadata is used to determine the unique SchemaID. Consequently, in the second step, the previously cached schema adapter can be requested from the registry adapter. This schema adapter is employed for message validation in step three, followed by serialization in step four. Step five involves encoding the serialized message to ensure a type-consistent format, as elaborated in Section 3.6. Finally, in the sixth step, the producer adapter asynchronously sends the message to the registry and returns an acknowledgment to the application once the broker has accepted the message.

Overall, these steps align with the process originally outlined in Section 3.1.1 and offer a method for producing type-consistent messages synchronously.

4.7 Message Consumption

This section focuses on message consumption, specifically the initialization of controllers and endpoints. What sets this process apart is that the incoming message dictates the schema, and there is no attempt to serialize it with an arbitrary schema, as is typically the case. Thanks to optimistic caching, the registration, and the initialization process by the controller, this procedure also predominantly operates synchronously. Similar to Section 4.6, the process is illustrated using sequence diagrams.

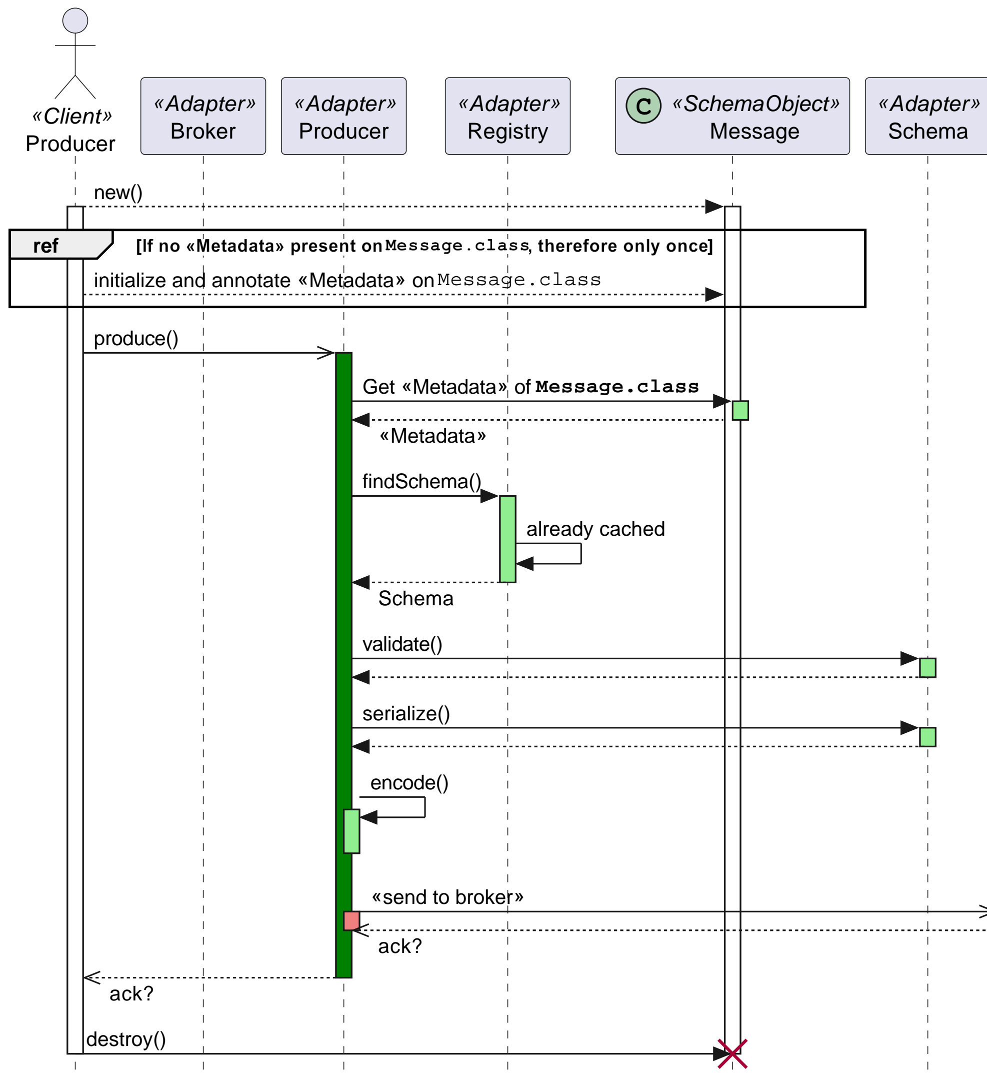

Figure 4.14: Sequence diagram of the *Message Production* process.

4.7.1 Initialization

The initialization process of the consumer primarily involves registering the controllers, which are annotated with the endpoints, in the consumer adapter. While the steps described bear a strong resemblance to those outlined in Section 4.6.1, only the differences will be explained further below.

Figure 4.15 illustrates the initialization procedure in a sequence diagram. Initially, a factory called the `ControllerFactory` takes responsibility for initializing and registering the controller, as most of the required dependencies for this task are irrelevant for subsequent execution. When the `ControllerFactory` is invoked, each of the defined endpoints within it must undergo an initialization process.

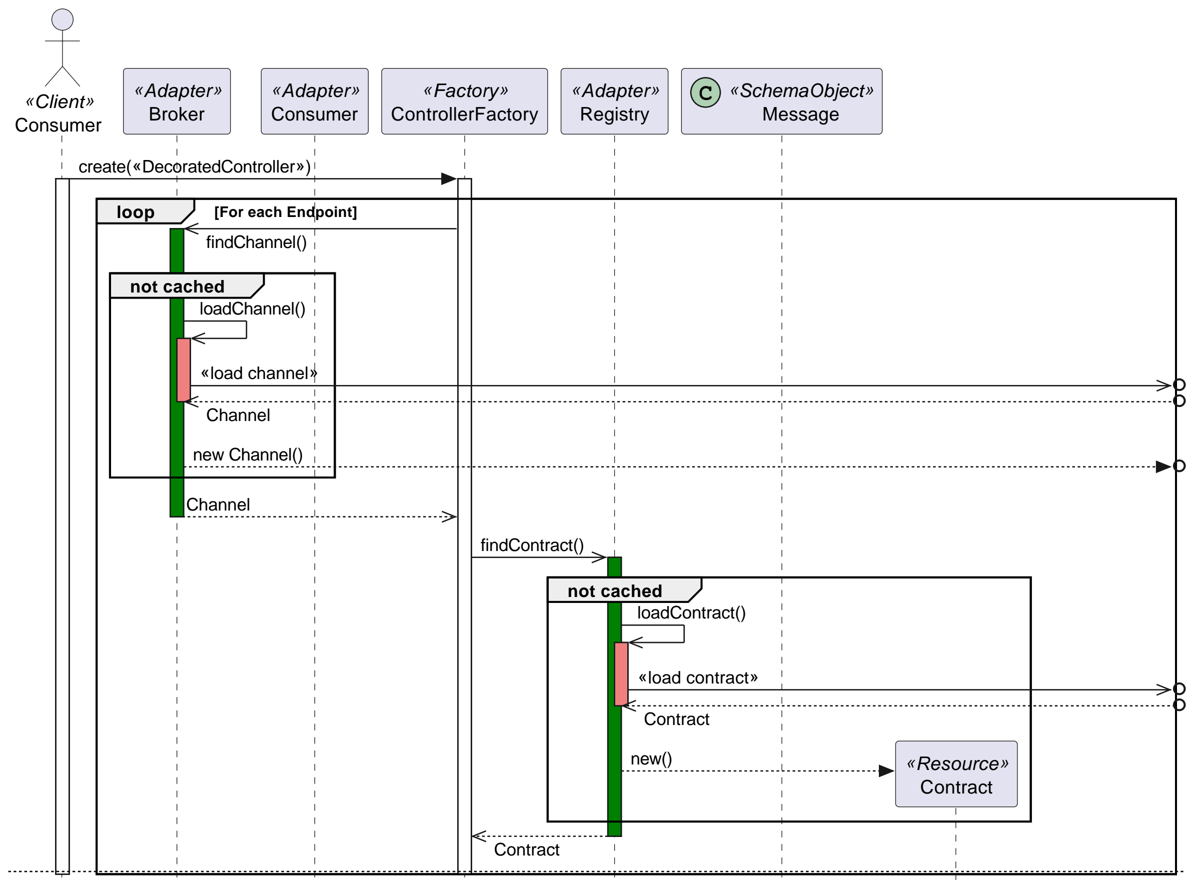

Figure 4.15: Sequence diagram of the initialization sequence of a endpoint. [Part 1/2]

First, the *brokeradapter* loads the channel into the application context. For certain brokers, such as Apache Kafka, the following steps must be repeated for multiple contracts. Hence, the *contract* must be loaded with the assistance of the registry adapter. Subsequently, the `RawSchema` is extracted from the `SchemaObject` associated with the endpoint. Using this `RawSchema`, the contract version and the corresponding schema can be loaded. Finally, an instance of the endpoint class must be created for each endpoint. This instance contains its name, channel, contract version, and a reference to the method initially decorated within the class that was decorated as a controller.

Following this, as illustrated in Figure 4.16, an instance of a `Controller.class` is generated along with its associated endpoints. Subsequently, this instance is registered in the consumer adapter, which subscribes to the channels specified by the endpoints. It is important to note that the consumer adapter must subscribe to all channels defined by the controller's endpoints. Consequently, as already stated in Section 4.5.3, this process can only be executed when the consumer adapter is not already connected to the broker.

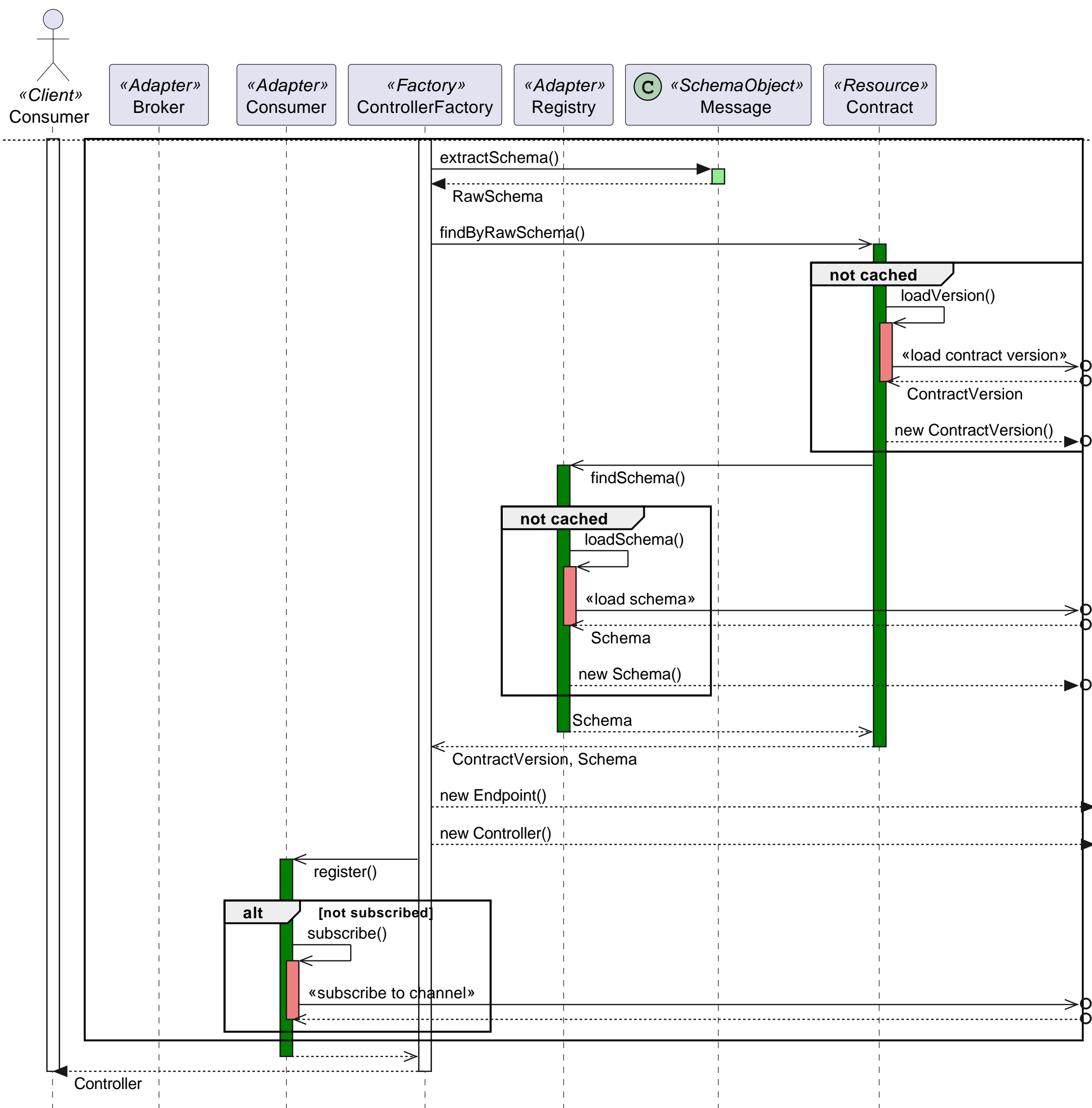

Figure 4.16: Sequence diagram od the initialization sequence of a endpoint. [Part 2/2]

4.7.2 Consumption

After the completion of the initialization process, as outlined in Section 4.7.1, the subsequent reception and processing of messages, as illustrated in Figure 4.17, primarily occurs synchronously.

Before a controller can process a message, four steps must be completed. In the first step, the message is decoded. This step is facilitated by the fact that a type-consistent message, as described in Section 3.6, already contains a reference, namely the SchemaID, to its schema. With this reference, the registry adapter can load the associated schema adapter. Thanks to optimistic caching, this process only has to be performed once for each unique SchemaID. Subsequently, the message can be deserialized using the schema adapter and then undergo revalidation as defined in Section 3.7.

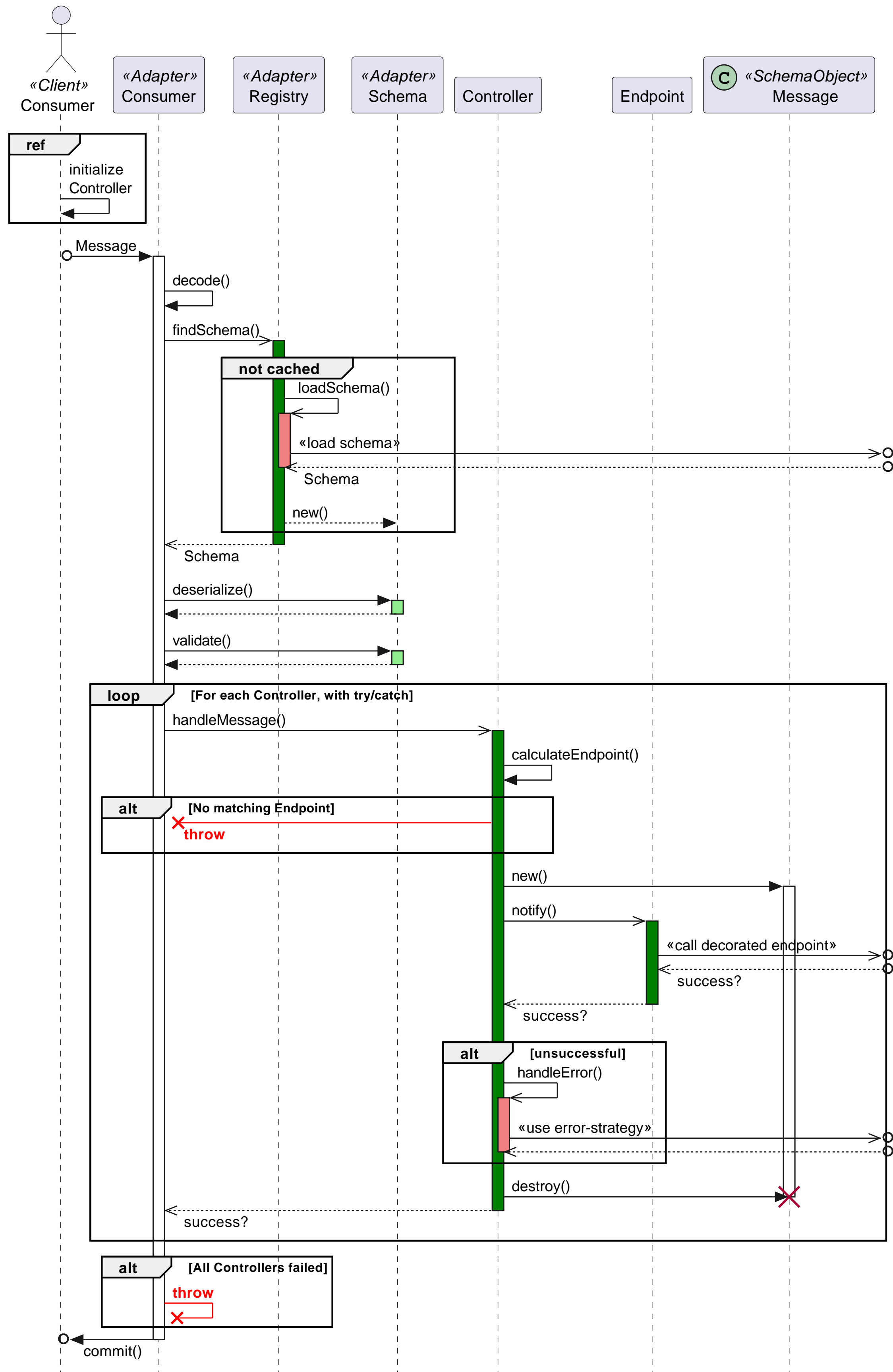

Figure 4.17: Sequence diagram of the *Message Consumption* process.

Subsequently, each controller is tasked with processing the message, ensuring that it reaches all its intended endpoints. The controller utilizes the channel and the SchemaID to determine the relevant endpoint. If it cannot find a suitable endpoint, depending on the controller or endpoint definition, a compatibility strategy, as proposed in Section 3.3.1, may be employed to identify a compatible endpoint. This intricate procedure may necessitate network requests to the registry for determining the message's precise contract version, although this falls outside the scope of this thesis. However, if no endpoint is found, the controller will raise an error.

In any case, an instance of the `SchemaObject` registered for this endpoint can be created, and the endpoint can be notified to invoke the initially decorated function. At this point, the BITCMT Framework transfers the message to the application's business logic. Subsequently, the endpoint reports whether the call was successful or not. If the invocation is successful, the `SchemaObject` can be safely destroyed, and the execution can proceed. If not, the controller may apply a predefined error strategy. These strategies are not addressed within the scope of this thesis and are therefore considered topics for future work, as outlined in Section 4.5.3. If these error strategies are successful, the previously instantiated `SchemaObject` can be destroyed.

In the subsequent process, there is one more noteworthy scenario to consider before the consumer adapter can commit a message. This situation arises when all controllers have encountered failures. Such a scenario unfolds when no matching endpoint exists within any of the controllers. Alternatively, it can also happen when all eligible endpoints have encountered exceptions, and the error strategy within their respective controllers has also failed. In this particular case, the consumer adapter is unable to commit the message under any circumstances.

Lastly, as illustrated in Figure 4.17, if the message has successfully reached its intended destination, the endpoint, and the execution has been deemed successful, it can then be committed by the consumer adapter. This signifies that the message has reached its final destination and has been processed successfully. Consequently, after the successful commit of the message - and only then - the type-consistent message transfer can be considered complete.

5 Implementation

In order to gain further experience and evaluate the architecture of the Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework), as presented in Chapter 4, a research prototype (ts-messaging) has been created. Significantly, metaprogramming concepts are challenging to represent in diagrams and text, so additional insight through a possible implementation is vital for the overall understanding of the in Chapter 3 presented concepts. The research prototype (ts-messaging) is made publicly available on GitHub (https://github.com/unaussprechlich/ts-messaging)[1] as an open-source project with an MIT[2] license.

The language TypeScript[3] has been chosen as the appropriate language for the implementation of ts-messaging, as it can model object-oriented programming. Furthermore, the growing relevance of web technologies is leading to an increasing effort to integrate the backend architecture into a homogenous web stack. Therefore, in recent years, the spread of TypeScript as a backend language has increased enormously, especially in the professional environment[4]. Furthermore, it is noticeable that the TypeScript ecosystem, despite its relevance through online e-commerce, has received little attention from developers of enterprise messaging solutions. Furthermore, they do not provide first-party TypeScript clients and rely on the open-source community for implementations. Another reason for choosing TypeScript was that the provided metaprogramming is particularly developer-friendly in its implementation.

The following are some decisions made during implementation - specifically, the challenges while integrating third-party libraries and systems with the adapters. These decisions are primarily based on the requirements derived from the proposed BITCMT Framework, as outlined in Chapter 4. It should also be mentioned that performance played a minor role in implementing ts-messaging, and therefore, performance was not a decision-making metric. Furthermore, all integrated dependencies are open-source and available on npm[5].

5.1 Framework Dependencies

This section outlines the framework dependencies of the developed research prototype (ts-messaging) of the BITCMT Framework.

[1] https://doi.org/10.5281/zenodo.8387363
[2] https://opensource.org/license/mit/
[3] https://www.typescriptlang.org/
[4] https://www.jetbrains.com/lp/devecosystem-2022/ and https://survey.stackoverflow.co/2023/
[5] https://www.npmjs.com/

For the implementation of the schema adapter (cf. Section 4.2), Apache Avro[6] was chosen as the preferred schema Domain-Specific Language (DSL). Apache Avro was favored over Protobuf due to its superior open-source model and wider adoption in messaging. JSON Schema was not selected as an adapter because it solely defines the message JSON representation without a serialization process. The avro-schema adapter was implemented using the avsc[7] package, which is the most popular on npm and follows a modern approach. Additionally, for message validation, the zod[8] library was utilized.

The implementation of the registry adapter (cf. Section 4.3) presented significant challenges due to the limited availability of open-source registry solutions. The Confluent Schema Registry[9], which is released under a community license, being the only viable option, ruling out all other commercial alternatives. Given the central role of the registry in ensuring type-consistent messaging and managing schema compatibility and contracts, building a custom implementation within the project's scope was not feasible. Consequently, the decision was made to utilize the Confluent Schema Registry. However, no suitable client could be found, therefore, a custom client has been developed. This client, written in TypeScript, is accessible on GitHub (https://github.com/unaussprechlich/ts-schemaregistry) as an open-source project under the MIT license. It is built on top of the Confluent Schema Registry Representational State Transfer (REST) API[10] for compatibility.

The broker adapter was developed for Apache Kafka, a widely used open-source messaging broker. Kafka's widespread adoption and relevance within the messaging domain made it the natural choice for this implementation. Furthermore, selecting a Kafka client was straightforward, with KafkaJS[11] being the most popular and available under the MIT license, making it the ideal candidate for the task.

The client implementation is self-contained and does not require any additional external dependencies.

5.2 Repository Structure

Since the package hierarchy is part of the BITCMT Framework, this structure has been embraced and ts-messaging was implemented in a mono repository[12]. Therefore, the same tooling can be used for all packages. The disadvantage is that these tools have to support mono repositories.

[6] https://avro.apache.org/
[7] https://www.npmjs.com/package/avsc
[8] https://zod.dev/
[9] https://github.com/confluentinc/schema-registry
[10] https://docs.confluent.io/platform/current/schema-registry/develop/api.html
[11] https://kafka.js.org/
[12] https://monorepo.tools/

NX[13] was used for the build system because it has a first-class mono repository integration and is published under an MIT license. The dependency management was supplemented by pnpm[14]. Additional build tools are ESlint[15] for static code analysis, therefore, code linting. Prettier[16] for consistent code formatting and enforcement of the code style. Changesets[17] as a changelog generator and tool for version management of the individual packages. As a modern testing framework for the TypeScript ecosystem, Vitest[18] has been selected. In addition, VitePress[19] generates the documentation from markdown files. This documentation is automatically deployed to the GitHub Pages of the ts-messaging repository through GitHub Actions.

5.3 Minimal Application Example

A minimal, reproducible example serves as an excellent introduction to a framework, offering a clear understanding through self-explanatory code, even without delving into intricate implementation details. Moreover, it effectively demonstrates the framework's ease of use, providing developers with an initial sense of the development experience.

To facilitate this understanding and developer experience, a minimal example is included in the same repository as the ts-messaging implementation (`https://github.com/unaussprechlich/ts-messaging/tree/main/examples/minimal-kafka`)[20]. The purpose of this minimal example is to simulate a type-consistent message exchange via a broker and a registry. In this scenario, messages are both produced and consumed by the same application. Consequently, for this minimal example to function, two prerequisites are essential: an accessible messaging cluster and a registry must already be in place. To address this, Confluent offers a comprehensive quick-start guide (`https://docs.confluent.io/operator/current/co-quickstart.html`) for deploying these resources in a Kubernetes[21] cluster. Additionally, all the necessary packages can be found on npm.

As Listing 1 shows, the first step is to configure and instantiate the registry adapter and the schema adapters as schema providers. The configuration option `baseUrl` is the address of the REST API provided by the Confluent Schema Registry. With the `autoRegisterSchemas` option set to `true`, the registry will automatically register new schemata.

[13] `https://nx.dev/`
[14] `https://pnpm.io/`
[15] `https://eslint.org/`
[16] `https://prettier.io/`
[17] `https://github.com/changesets/changesets`
[18] `https://vitest.dev/`
[19] `https://vitepress.dev/`
[20] `https://doi.org/10.5281/zenodo.8387363`
[21] `https://kubernetes.io/`

Listing 1 Minimal example: Initialization of the registry adapter and schema adapter.

```
import { Confluent } from '@ts-messaging/registry-confluent';
import {Avro} from "@ts-messaging/schema-avro";

const confluentSchemaRegistry = new Confluent({
    clientConfig: {
        baseUrl: 'http://localhost:8081',
    },
    schemaProviders: [new Avro()],
    autoRegisterSchemas: true,
})
```

Listing 2 shows the creation of a new `SchemaObject`. The class that implements `AvroObject` gets the schema annotated as metadata through decorators. The detailed definition of the possible schema annotations is not part of this work. However, it should be mentioned that the `AvroObject` metaprogramming described in Listing 2 produces an Apache Avro schema with exactly one field of type `'integer'` and the name `'my_field1'`.

Listing 2 Minimal example: Definition of a `SchemaObject`.

```
import { Avro, AvroObject } from '@ts-messaging/schema-avro';

@Avro.Record({
    name: 'sampleRecord',
    namespace: 'com.mycorp.mynamespace',
    doc: 'Sample schema to help you get started.',
})
export class SampleRecord implements AvroObject {
    @Avro.Int({
        doc: 'The int type is a 32-bit signed integer.',
    })
    my_field1: number | undefined;

    //If the schema is used in a endpoint all constructor arguments must be optional.
    constructor(my_field1?: number) {
        this.my_field1 = my_field1;
    }
}
```

Next, in this procedure, a broker adapter - which is also the client - must be created. As shown in Listing 3, this client is configured by providing a registry adapter with its schema adapters and the constructors of the client controllers. The configuration option `autoRegisterChannels` allows the automatic registration of new channels in the broker. After its successful definition, the `init()` method has to be called to trigger its internal initialization process.

Listing 3 Minimal example: Initialization of the broker adapter and client.

```
import { Kafka, KafkaBroker } from '@ts-messaging/client-kafka';

const client = new Kafka({
    broker: { brokers: ['localhost:9092'] },
    registry: confluentSchemaRegistry,
    controllers: [MinimalController],
    autoRegisterChannels: true,
});

await client.init();
```

In the next step, as shown in Listing 4, such a definition of a controller can be created by decorating a class using the controller decorator. A new consumer instance is created in the background through the configuration, passed to the controller annotation, or the default consumer defined by the client is used. The endpoints' channels are also automatically subscribed to by the consumer in use. The actual endpoint can then be created by annotating a function with the endpoint decorator. This endpoint will automatically receive all messages the consumer received through the channel `'minimal.example'` and has a matching contract version. The annotation of the individual function parameters determines where the controller must insert the respective parameter when the function is called. Furthermore, these decorators also define the contract version. Thus, annotating a parameter with `Kafka.Payload()` defines that the payload of the message must have the type `SampleRecord`, and its annotated metadata can be used to register the schema through the registry adapter. Moreover, the default producer or a newly configured producer can be injected into the controller with the `@Kafka.producer(...config?)` annotation.

Listing 4 Minimal example: Definition of a controller.

```
import { Kafka, KafkaProducer, KafkaMetadata, KafkaController } from '@ts-messaging/client-kafka';

@Kafka.Controller({ consumer: { groupId: 'minimal-example' } })
class MinimalController implements KafkaController {

    @Kafka.Producer()
    readonly producer: KafkaProducer;

    @Kafka.Endpoint('minimal.example')
    async onMessage(
        @Kafka.Key() key: string,
        @Kafka.Payload() payload: SampleRecord,
        @Kafka.Metadata() meta: KafkaMetadata
    ) {
        // ...
    }
}
```

Last, Listing 5 shows how such a message can be produced for the channel `'minimal.kafka'`. For this purpose, a suitable producer adapter is created via the broker adapter, ord the default adapter can be used as well. Subsequently, a type-consistent message can be produced by simply calling the method `produce()`. The channel is defined by its name `'minimal.example'`. The `payload` option indirectly defines the contract version. The loading of the channel, the contract versions, and its schemata, as well as the sending and serialization of the message, is done by the BITCMT Framework in the background.

Listing 5 Minimal example: Producing a type-consistent message.

```
1  await client.broker.defaultProducer.produce({
2      channel: "minimal.example",
3      key: '::1',
4      payload: new SampleRecord(123),
5  });
```

5.4 Summary

The implementation is a substantial part of this work, serving as the practical realization of the concepts outlined in Chapter 3 and the architecture of the BITCMT Framework as described in Chapter 4. Implementing ts-messaging was challenging, especially since the adapters have introduced far-reaching requirements and various edge cases. It is worth noting that the BITCMT Framework presented in this paper is only an implementation template. Therefore, many individual details can extracted directly from the code and its accompanying documentation. Despite its prototype nature, the repository incorporates numerous essential development and deployment tools, making it ready for future development into a production-ready project. Overall, as further discussed in Chapter 6, this implementation serves as the foundation for evaluating this work.

6 Evaluation

The overall evaluation objective is to assess the feasibility and applicability of the Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework) template and the research prototype (ts-messaging) presented in this paper. To achieve this goal, the evaluation process is divided into three stages: a code review, a minimal application example (cf. Section 5.3), and the reimplementation of a reference architecture (cf. Section 6.1) in a case study. The code review focused on evaluating the technical implementation as a whole, while the minimal application example served as a proof-of-concept, assessing its initial feasibility. Lastly, the reimplementation of the T2-Project reference architecture aims to assess the capabilities and limitations of the framework within a simulated real-world scenario [SSB22].

Moreover, the primary goal of this thesis is to conceptualize and construct a framework template that facilitates type-consistent messaging in a broker-independent fashion. In order to evaluate the viability of the outcome of this objective, four primary technical questions (TQs) have been proposed:

TQ1 Is it possible to distribute a type-consistent message in a messaging cluster by employing code-first schema principles?
This question aims to investigate the feasibility of distributing messages within a messaging cluster while adhering to code-first schema principles. It seeks to understand if code-first approaches can ensure type consistency in this context.

TQ2 Can type-consistent messaging be performed broker-independent?
Explores the possibility of achieving type-consistent messaging without being dependent on a specific messaging broker. It examines whether type consistency can be maintained in a broker-independent messaging environment.

TQ3 How can common interfaces for the registry, schema, broker, consumer, producer, and message be defined?
This question focuses on defining common interfaces that can be used across various components of the messaging system, including the registry, schema, broker, consumer, producer and message. It aims to establish standardized framework interfaces.

TQ4 How can an application-to-application messaging paradigm be implemented in a developer-friendly manner?
Delves into the implementation of an application-to-application messaging paradigm while prioritizing a developer-friendly approach. It seeks to identify strategies and practices that enhance the developer experience in this context.

The subsequent sections elaborate on the evaluation process of this case study and the results of the evaluation. Furthermore, the evaluation's validity is assessed in Section 6.2.

6.1 Case Study: T2-Project Reference Architecture

To assess the BITCMT Framework presented in this paper, a case study involving the reimplementation of a reference architecture, specifically the *T2-Project*[1], was conducted using the research prototype developed earlier (cf. Chapter 5). This reference architecture, utilizing the *saga pattern*[Gon22; Ric18], poses additional architectural challenges to the proposed BITCMT Framework [SSB22]. In essence, the case study serves as a means to further evaluate the framework's feasibility, practicality, and developer experience. Furthermore, it delves into exploring the challenges and limitations of the current BITCMT Framework template.

To provide a more comprehensive overview of these primary objectives and their associated goals:

O1 **Reimplement the T2-Project reference architecture using the ts-messaging:**
This objective involves the process of recreating the T2-Project reference architecture, utilizing the ts-messaging framework as a foundation.

O2 **Assess the feasibility and practicality of the BITCMT Framework:**
This objective aims to evaluate the extent to which the BITCMT Framework framework can be realistically applied and how well it functions in practical scenarios.

O3 **Evaluate the developer experience of the BITCMT Framework:**
This objective focuses on assessing the overall satisfaction and usability of the BITCMT Framework framework from the perspective of developers who work with it.

O4 **Identify the challenges and limitations of the BITCMT Framework:**
Here, the objective is to recognize and document any difficulties or restrictions encountered using the BITCMT Framework framework.

O5 **Propose future work and enhancements for the BITCMT Framework:**
This objective involves suggesting potential areas for future development and improvement of the BITCMT Framework framework, based on the findings and insights gained during the evaluation.

The technical prerequisite of this case study are as follows: Given that the reference architecture is originally implemented in Java, a separate project clone, tailored to the *Node.js*[2] ecosystem, as dictated by the existing research prototype's requirements, had to be created. For this clone, the backend framework chosen was *NestJS* (NestJS Official Website). The stripped-down reimplementation of the reference architecture can also be accessed on GitHub within the mono repository of the research prototype (`https://github.com/unaussprechlich/ts-messaging/tree/main/examples/t2-nestjs`)[3].

6.1.1 Results

The evaluation of the BITCMT Framework and ts-messaging presented in this paper yielded several noteworthy results:

[1] https://github.com/t2-project
[2] https://nodejs.org
[3] https://doi.org/10.5281/zenodo.8387363

R1 **Successful Reimplementation (Proof-of-Concept):**
The BITCMT Framework demonstrated its adaptability and robustness by successfully reimplementing the reference architecture. Furthermore, this demonstrated the framework's ability to be further developed into a production-ready template.

R2 **Minimal Challenges (In the scope of this work):**
The framework's ability to minimize challenges during the reimplementation process suggests that the BITCMT Framework is user-friendly. It offers clear guidelines for developers to produce messages and define endpoints. Therefore facilitating a smooth transition from theory to practice.

R3 **Compatibility with NestJS:**
The ability to seamlessly clone and adapt the reference architecture from Java to the *NestJS* ecosystem is a testament to the framework's flexibility. Therefore, this implementation further highlights the framework's compatibility with modern technologies. Furthermore, this showcases the framework's potential to be adaptable to diverse technology stacks, making it suitable to various project requirements.

R4 **Sage Pattern Challenge:**
The integration of the *saga pattern* by the *T2-Project* challenged the proposed BITCMT Framework and added a further dimension to the evaluation. Hence, this demonstrates that the BITCMT Framework can handle advanced architectural patterns and remains robust in complex design challenges.

R5 **Challenge: Error Handling:**
The BITCMT Framework lacks a clear error-handling strategy. The research prototype only implements a single error handler per controller. Therefore, the implementation of patterns like dead letter queue is still highly manual and lacks developer experience. This is a significant shortcoming that needs to be addressed in future development through common endpoint-error strategies.

R6 **Challenge: Message Replies:**
NestJS proposed message reply strategy has a much better developer experience than the one implemented in the research prototype. Therefore, the BITCMT Framework should be extended to support a similar approach. It should be possible to define a reply channel and produce a new message through the return object of the endpoint.

R7 **Open Source:**
The framework's open-source nature encourages further development, research, and contributions from a wider community.

In summary, the BITCMT Framework presented in this paper has demonstrated promising capabilities in successfully implementing a reference architecture. It exhibits adaptability to various technology ecosystems and effectively handles advanced architectural patterns. Nevertheless, the BITCMT Framework needs further development, particularly in terms of proposing an integrated error-handling strategy and an integrated message-reply strategy. Additionally, it should be expanded to support a broader spectrum of integrated messaging patterns. Lastly, the BITCMT Framework should undergo further evaluation in a production environment to ascertain its reliability and performance.

6.2 Threads to validity

In this section, the four aspects of validity - construct, internal, external, and reliability, as defined by Runeson and Höst [RH09], are discussed.

In terms of external validity, it is important to acknowledge that the proposed modular architecture may have limitations when applied to other messaging platforms beyond the analysis scope of this thesis. Due to time constraints, the analysis of messaging platforms was restricted, and this thesis primarily focuses on *Apache Kafka* and *Apache Avro* as implementation targets. Consequently, other platforms might employ different and potentially incompatible concepts, potentially limiting the generalizability of the BITCMT Framework. To enhance external validity in a future BITCMT Framework iteration, conducting expert interviews with experienced everyday developers and architects could provide valuable insights into further requirements and potential shortcomings of existing solutions.

Regarding internal validity, there is a risk that crucial aspects of type-consistent and broker-independent messaging may have been inadvertently overlooked. Noteworthy is, that the proposed architecture is primarily a composition of existing partial solutions and established concepts. They have been deployed in production environments and demonstrated reliability. Therefore, the internal validity of these described concepts extends beyond this specific set of biased technologies. However, it is important to recognize that the internal validity of the implemented research prototype and architecture is constrained to the technologies analyzed during the research phase and those that are supported by the BITCMT Framework.

Furthermore, the chosen evaluation and research methods may have limitations in fully assessing the framework's capabilities. Evaluating the research prototype and, by extension, the BITCMT Framework itself could benefit from user studies involving experienced developers. However, such a study was not intended within the scope of this thesis, which primarily aimed to demonstrate application-to-application type-consistent and broker-independent messaging. Nevertheless, for future work, conducting additional user studies is highly recommended to comprehensively evaluate the framework's effectiveness and the developer experience.

Finally, the reliability of the research prototype and the proposed BITCMT Framework is limited by the lack of a production-ready implementation. The research prototype is a proof-of-concept implementation that has not yet been tested in a production environment. Therefore, establishing reliability requires real-world deployment, testing, scalability assessment, performance optimization, and handling diverse use cases in a production setting. Such endeavors could provide significantly different results compared to the current state of the proposed BITCMT Framework. However, the research prototype's reliability is not a primary concern of this thesis, as its primary purpose was to demonstrate the feasibility of the proposed concepts. Nevertheless, the reliability of the BITCMT Framework should be further assessed in future work.

6.3 Summary

Chapter 6 offers an initial assessment of the BITCMT Framework and the concepts introduced earlier in the thesis. Due to the complexity, the evaluation has been carried out in three phases: a code review for implementation assessments, a minimal application example as a proof-of-concept,

and a case study involving the reimplementation of the T2-Project reference architecture. The objective of this evaluation was to address critical technical inquiries and gauge the framework's viability and capabilities.

The technical questions (TQs) raised can be answered as follows:

TQ1 Is it possible to distribute a type-consistent message in a messaging cluster by employing code-first schema principles?
Indeed, one of the primary objectives of this thesis has been achieved. The minimal application example (cf. Section 5.3) and the case study (as outlined in Section 6.1) have effectively demonstrated that the BITCMT Framework framework can distribute type-consistent messages within a messaging cluster, all while adhering to a code-first schema principle. The BITCMT Framework ensures that messages maintain their consistency as they depart from the producer in a predefined type-consistent format and only proceed to their intended endpoint if said endpoint has been implemented with a compatible contract version. Furthermore, the schema is directly extracted from the code-first schema definition and registered within a contract in the proposed registry. This feature holds immense significance in upholding data integrity and ensuring the reliability of distributed systems.

TQ2 Can type-consistent messaging be performed broker-independent?
Yes, type-consistent messaging can be performed broker-independently with the BITCMT Framework. In comparison to Apache Pulsar (cf. Section 2.2.2), one of the notable advantages of the proposed BITCMT Framework lies in its modular architecture, which is entirely independent of the underlying messaging infrastructure. This broker-independence is essential for system portability and flexibility in choosing messaging infrastructure.

TQ3 How can common interfaces for the registry, schema, broker, consumer, producer, and message be defined?
In Chapter 4 this work provides common interfaces for key components such as the registry, schema, broker, consumer, producer, and message. The implementation of the research prototype revealed that these interfaces are sufficient for the framework's implementation. Furthermore, the interfaces are designed to be easily extendable and adaptable to future requirements.

TQ4 How can an application-to-application messaging paradigm be implemented in a developer-friendly manner?
Yes, in comparison to many existing messaging clients, the BITCMT Framework proposed in this paper promotes the separation of business logic from the messaging architecture. Utilizing metaprogramming, type-consistent messaging is decoupled from business-related logic. This separation fosters clean code practices and a clear separation of concerns, consequently enhancing maintainability, scalability, and testability.

In summary, the evaluation of the BITCMT Framework confirms that it is capable of enabling type-consistent messaging in a broker-independent manner. Furthermore, with the provided common interfaces, the business logic has been successfully decoupled from the messaging system. These findings suggest that the BITCMT Framework holds promise for building robust and flexible distributed systems. However, there are undefined areas, such as error handling and message replies, that require further development and refinement to realize the potential of the BITCMT Framework.

Moreover, the analysis of the aspects of validity in Section 6.2 have revealed limitations in the chosen evaluation and analysis methods. Therefore, for a future iteration of the BITCMT Framework, it is advisable to incorporate additional research methods, e.g. expert interviews, and expand evaluation methods, including user studies, for a more comprehensive assessment. Consequently, the reliability of the BITCMT Framework has to be further assessed in future work.

7 Conclusion and Future Work

In this concluding chapter, the results of this thesis are summarized, highlighting the achievements and contributions made. Additionally, possible avenues for future work and potential expansions of the Broker-Independent Type-Consistent Messaging Template Framework (BITCMT Framework) are outlined.

Conclusion

In highly distributed systems, such as messaging stacks, loose coupling is the most important design principle. However, this loose coupling can lead to type inconsistency, since the individual participants of the messaging stack are not aware of each other. Yet, to ensure correct message processing all participants have to agree on a common schema. However, all examined solutions have a platform-first approach and lack developer experience though providing only low-level client implementations. Therefore, the core objective of this thesis was to develop the BITCMT Framework that enables application-to-application, type-consistent messaging in a broker-independent manner.

To achieve this goal, the thesis first analyzed the current state of messaging stacks and identified preexisting solutions. The analysis revealed that the current implementations are platform-first solutions and lack true application-to-application business logic integration.

To address this shortcoming this thesis reconsidered their approach and concepts to develop a developer-centric and type-consistent messaging paradigm. To summarize the approach, this work formalized the concept of a messaging contract and its compatibility implications. Furthermore, this work redefined possible Representational State Transfer (REST) endpoints for a central contract and schema repository, the registry. This registry acts as a single source of truth within the messaging stack. Hence, it becomes possible to identify potential schema discrepancies between the producer and the consumer. To ensure type consistency during message transmission, this thesis proposed a common message encoding format. Moreover, the process of message consumption needed to be reimagined in order to guarantee type-safe business logic invocation, a concept that was introduced through the concept of message endpoints.

To further develop the proposed concepts, this thesis presented an architecture for such a messaging client, BITCMT Framework. The architecture is based on previously established concepts and provides a modular foundation that is designed to be easily extendable and adaptable to future requirements. The architecture is divided into three layers: the *common layer*, the *framework layer*, and the *implementation layer*. The common layer provides the interfaces for the framework layer, which in turn provides a shared framework logic for the specific implementation layer. The individual components exist of common adapters for the broker, consumer, producer, registry, and schema. Furthermore, the BITCMT Framework provides common interfaces for messaging stack

resources like the message, contract, contract version, and channel. These interfaces are designed to be easily combined in a modular client architecture. This client is the final component of the BITCMT Framework and provides the interface for the application. The client is responsible for the initialization of the individual adapters and the registration of the message endpoints in its corresponding controller.

Evaluation has been performed by implementing the proposed framework template as a research prototype. Moreover, this research prototype was then utilized to assess the feasibility and practicality of the proposed architecture. Consequently, a case study was conducted, involving the reimplementation of the T2-Project reference architecture using NestJS. During this evaluation phase, the BITCMT Framework showed advanced adaptability and clean application integration through the utilization of metaprogramming techniques. Thereby, the BITCMT Framework demonstrated application-to-application type-consistent messaging. However, the evaluation also revealed that the BITCMT Framework lacks an integrated error-handling strategy and message reply strategy. Therefore, these are areas that require further development and refinement to realize the potential of this proposed BITCMT Framework. However, the modularity and openness of the BITCMT Framework demonstrated its ability to be further developed into a production-ready project.

Future Work

As stated, this section provides an outlook on possible future work and future BITCMT Framework enhancements.

First, it's worth noting that there is a limited presence of technical literature on the topic of type-consistent messaging. However, many enterprise and open-source solutions address this problem as a core feature. This apparent gap in the literature presents a compelling opportunity for future research since there is a clear real-world demand. Moreover, the lack of literature on this topic also presents a challenge for the development of the BITCMT Framework. Therefore, the only quotable source of inspiration has been the existing solutions and their documentation. Consequently, this is an invention for researchers to further explore this topic and develop a theoretical foundation for type-consistent messaging.

Furthermore, the research prototype (ts-messaging) presented in this thesis is a prototype, and the BITCMT Framework is a template. Therefore, these are not production-ready and require further development. Regarding further requirements, the ts-messaging implementation requires additional adapters to work with a diverse set of preexisting schemata, registries, and broker. Thereby, the BITCMT Framework can evolve to support a wide range of messaging platforms. Furthermore, the BITCMT Framework should be extended to support integrated message replies, such as the request-response messaging concept[1] that has been retrofitted by NestJS. Additionally, it should propose an integrated error-handling strategy and implement a subscription approach similar to Apache Pulsar[2], but at the endpoint level. Lastly, the BITCMT Framework should include schema-specific code generators to expedite the integration of existing schemas into the application context.

[1] https://docs.nestjs.com/microservices/kafka

[2] Apa23a.

Finally, as Chapter 6 has revealed, the BITCMT Framework should be further evaluated in a production-ready environment. This productive system would enable a more in-depth evaluation of the framework's performance, scalability, and compatibility. Moreover, additional analysis through expert interviews and user studies should be conducted to further evaluate the framework's capabilities and developer experience. These evaluations could serve as the foundation to guide a future iteration of the BITCMT Framework.

Lastly, the research prototype and additional documentation have been released as an open-source project (`https://github.com/unaussprechlich/ts-messaging`)[3]. This initiative fosters continued development, research, and contributions from a broader and more diverse community of developers and researchers.

[3] `https://doi.org/10.5281/zenodo.8387363`

Bibliography

[Apa23a] Apache Foundation. *Messaging with Pulsar*. 2023. URL: https://pulsar.apache.org/docs/3.1.x/concepts-messaging/ (cit. on pp. 5, 43, 66).

[Apa23b] Apache Foundation. *Overview Pulsar Schema*. 2023. URL: https://pulsar.apache.org/docs/3.1.x/schema-overview/ (cit. on p. 9).

[Apa23c] Apache Foundation. *Understanding Pulsar Schema*. 2023. URL: https://pulsar.apache.org/docs/3.1.x/schema-understand/ (cit. on pp. 9, 18).

[Con23a] Confluent, Inc. *Schema Evolution and Compatibility*. 2023. URL: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html (cit. on pp. 6, 18, 19).

[Con23b] Confluent, Inc. *Schema Registry Key Concepts*. 2023. URL: https://docs.confluent.io/platform/current/schema-registry/fundamentals/index.html (cit. on pp. 27, 33).

[Con23c] Confluent, Inc. *Schema Registry Overview*. 2023. URL: https://docs.confluent.io/platform/current/schema-registry/index.html (cit. on pp. 10, 12).

[GHJV95] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995 (cit. on pp. 31, 33, 34, 38, 41).

[Gon22] S. R. Goniwada. “Cloud Native Architecture and Design Patterns”. In: *Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples*. Berkeley, CA: Apress, 2022, pp. 127–187. ISBN: 978-1-4842-7226-8. DOI: 10.1007/978-1-4842-7226-8_4. URL: https://doi.org/10.1007/978-1-4842-7226-8_4 (cit. on pp. 4, 60).

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (cit. on pp. 3, 5, 6).

[JD20] L. Johansson, D. Dossot. *RabbitMQ essentials : build distributed and scalable applications with message queuing using RabbitMQ, Second edition*. Packt Publishing, 2020, p. 147. ISBN: 9781789131666. URL: https://portal.igpublish.com/iglibrary/search/PACKT0005711.html (cit. on pp. 3, 4).

[Kru16] P. Krusenotto. *Funktionale Programmierung und Metaprogrammierung*. Jan. 2016. ISBN: 978-3-658-13743-4. DOI: 10.1007/978-3-658-13744-1 (cit. on p. 7).

[NK12] R. Neswold, C. King. *Further Developments in Generating Type-Safe Messaging*. 2012. arXiv: 1210.1223 [physics.ins-det] (cit. on p. 8).

[R N09] C. K. R. Neswold. “Generation of Simple, Type-Safe Messages for Inter-Task Communications”. In: *Proceedings of ICALEPCS2009*. 2009, pp. 137–139 (cit. on p. 8).

[RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14 (2009), pp. 131–164. URL: https://api.semanticscholar.org/CorpusID:207144526 (cit. on p. 62).

[Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN: 9781617294549 (cit. on p. 60).

[Sha22] R. Sharma. *Cloud-Native Microservices with Apache Pulsar : Build Distributed Messaging Microservices*. eng. 1st ed. 2022. Berkeley, CA: Apress, 2022. ISBN: 9781484278390 (cit. on p. 8).

[SSB22] S. Speth, S. Stieß, S. Becker. “A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis”. In: *Companion Proceedings of 19th IEEE International Conference on Software Architecture (ICSA-C 2022)*. IEEE, Mar. 2022. DOI: 10.1109/ICSA-C54293.2022.00029 (cit. on pp. 59, 60).

[STS20] J. Schürmann, T. Tegeler, B. Steffen. “Guaranteeing Type Consistency in Collective Adaptive Systems”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*. Ed. by T. Margaria, B. Steffen. Cham: Springer International Publishing, 2020, pp. 311–328. ISBN: 978-3-030-61470-6 (cit. on p. 8).

[Var16] E. Varga. “Schema-Based Messages”. In: *Creating Maintainable APIs: A Practical, Case-Study Approach*. Berkeley, CA: Apress, 2016, pp. 215–227. ISBN: 978-1-4842-2196-9. DOI: 10.1007/978-1-4842-2196-9_14. URL: https://doi.org/10.1007/978-1-4842-2196-9_14 (cit. on p. 6).

[WB23] A. Wąsowski, T. Berger. *Domain-Specific Languages*. Springer International Publishing, 2023. DOI: 10.1007/978-3-031-23669-3 (cit. on p. 6).

[WF94] A. Wright, M. Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115.1 (1994), pp. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935 (cit. on p. 7).

[ZK21] A. Zelenin, A. Kropp. *Apache Kafka*. München: Carl Hanser Verlag GmbH & Co. KG, 2021. DOI: 10.3139/9783446470460. eprint: https://www.hanser-elibrary.com/doi/pdf/10.3139/9783446470460. URL: https://www.hanser-elibrary.com/doi/abs/10.3139/9783446470460 (cit. on pp. 4–7, 40).

All links were last followed on September 28, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 28. September 2023

place, date, signature