Scalable and Anonymous Pub/Sub in Kotlin: A Model-Driven and Component-Based Approach with Kafka

Youngjae Moon

***Abstract*—This paper presents an innovative software engineering methodology using a unique combination of UML diagrams, Petri-nets, Kotlin, and formal verification tools. We detail our methodology, which involves:**

1) **Specifying model and components with UML diagram, Petri-nets**
2) **Generating unoptimised code in Kotlin**
3) **Optimize Kotlin code using formal verification tools, returning efficient, verifiable, and performant Kotlin code as a monolith code written in Java.**

Moreover, this paper presents a novel approach to implementing Publisher/Subscriber (Pub/Sub) systems using a combination of message queues and efficient message routing techniques through Apache Kafka. The proposed approach leverages RabbitMQ, a popular open-source message broker, to deliver reliable and scalable messages. It employs a hierarchical topic-based routing mechanism to ensure efficient message dissemination to targeted subscribers. The resulting pub/sub system handles many messages with low response times. This work contributes to the field by introducing an efficient and scalable pub/sub implementation that caters to the demands of modern distributed applications. Thus, it shows the potential of how more complex distributed systems, such as the Raft consensus protocol, could be implemented more skillfully based on the proposed software engineering methodology.

***Index Terms*—Apache Kafka, automated verification, automatic code generation, Coloured Petri-nets, component-based software engineering, model-driven software engineering, consensus algorithm, distributed systems, functional programming, FP, input-output automata, IntelliJ IDEA, IOA, JSON, Kotlin, KRaft, logic programming, Object-oriented programming, OOP, Petri-Nets, Pub/Sub, Publisher/Subscriber, RabbitMQ, Raft, UML**

I. Introduction

Building complex systems from individual components offers numerous benefits, as Xiaoming Liu et al. highlighted in their work on high-performance communication systems [17]. This approach facilitates easier design, development, testing, and optimisation of individual components. It also promotes adaptability to new environments, enables system extension with new components at runtime, and allows for formal specification and verification of individual components.

However, despite these advantages, component-based development faces significant challenges. First, the overhead is imposed by abstraction barriers between components, which also disables various compiler optimisation techniques due to the separate compilation of each component. Second, configuring systems from components can be difficult. Third, there are challenges in generating and verifying the code of individual components. Despite these challenges, the trend towards componentisation has led to the developing of complex systems such as operating systems, cloud servers, and cyber-physical systems with more functionality but not necessarily more reliability or performance.

In addition, programming in Kotlin is generally considered easier than programming in Java for five main reasons.

1) Kotlin is a hybrid functional and object-oriented programming language. It has first-class support for functional programming concepts like immutability, higher-order functions, and pattern matching. This support can lead to more expressive and concise code and also helps in writing code that is easier to reason about, test, and debug.
2) Kotlin encourages using immutable objects, which can lead to safer and more predictable code. Immutability is a key concept in functional programming and helps manage side effects, making it easier to write concurrent and parallel programs.
3) Kotlin has a more sophisticated type system compared to Java. Features like variance annotations and type bounds can provide more control and expressiveness.
4) Kotlin has null safety features that help avoid null pointer exceptions, a common source of errors in Java.
5) Kotlin's syntax is more concise than Java's. Kotlin reduces boilerplate code, which is prevalent in Java. For instance, Kotlin allows semicolons to be optional, making code more concise and readable.

These challenges highlight the need for an alternative approach to component-based software engineering that addresses the limitations of both traditional component-based development and Java. The approach presented in this paper aims to achieve the benefits of component-based development while overcoming the difficulties associated with both component configuration and programming in Java.

II. Project Goals

This project aims to develop and implement a novel Publish/Subscribe architecture with the following key goals:

1) Robust and Scalable Message Delivery:

- Use UML diagrams and Petri-Nets for modelling the system behaviour, enabling formal verification and ensuring correct message delivery.

Youngjae Moon is a Master's in computer science and Engineering Graduate Fellowship recipient at Vanderbilt University. e-mail: youngjae.moon@Vanderbilt.Edu

Manuscript received November 23, 2023

- Leverage Apache Kafka for using KRaft consensus protocol to achieve fault tolerance and high availability, ensuring consistent message delivery despite node failures [2].

2) Efficient and Anonymous Communication:

- Generate the initial Kotlin code using the RPC design pattern to promote cleaner separation of concerns and improve code maintainability.
- Implement anonymity mechanisms within the architecture to decouple publishers and subscribers, enhancing security and privacy.

3) Formal Verification:

- Model a portion of the software system as Petri nets to enable rigorous verification of the system's functionality and behaviour.

By achieving these goals, the project aims to build a robust, scalable, and efficient Pub/Sub system that delivers reliable message exchange while ensuring anonymity and fault tolerance. This system will be valuable for applications requiring secure and reliable communication in distributed environments.

III. Methodology

A. Overview of the proposed methodology

Our methodology leverages the combination of UML diagram and Petri-Nets for design and specification, Kotlin for code generation, and Stainless for verification and optimisation. This involves three main steps as described below:

1) Define the Publisher/Subscriber architecture and Kafka for distributed systems in UML diagrams and Petri-nets, establishing a robust framework for system interactions.
2) Generate the unoptimized code in Kotlin.
3) Perform code verification and optimisation using formal verification tools, enhancing system efficiency and reliability.

Technologies Used: Coloured Petri-nets, CPN IDE, IntelliJ IDEA, Kotlin.

This is different from the component-based software engineering methodology presented in Xiaoming Liu et al.'s paper in two main ways [17]. First, Kotlin is chosen as the target language instead of OCaml. This is mainly due to a broader range of frameworks and libraries in Kotlin than in Ocaml. Thus, Kotlin is widely used in Android and Micro-services development. Second, we have proposed using other formal verification tools to verify and optimise the code instead of Nuprl. Nuprl is not designed to verify Kotlin code. Thus, Nuprl is open-sourced and free to use, it requires permission from the Computer Science department at Cornell University to be used.

The proposed software engineering methodology in this paper will be more valuable than the methodology proposed by Xiaoming Lu et al. [15].

1) Modelling in UML class diagrams before modelling in Input/Output Automata (IOA) organises code into well-defined modules, making it easier to design, understand and maintain. Object-oriented programming (OOP) is based on the concept of objects, which encapsulate data and behaviour. This encapsulation promotes modularity and code reusability. By creating well-defined interfaces, OOP enables clear separation of concerns, making systems easier to understand, develop, and maintain.
2) Petri-nets excel in modelling concurrent processes and their interactions. They can naturally represent situations where multiple processes operate independently but need to synchronize at specific points, which is less intuitive in the IOA model.
3) Petri-nets can be more expressive in specific scenarios, especially when dealing with complex synchronization and concurrency requirements. Their graphical nature aids in visualizing the flow of the process and the states of the system.
4) Petri-nets provide solid analytical capabilities, allowing for analysing properties like reachability, liveness, and deadlock-freeness. These analyses can be crucial in verifying the correctness and efficiency of distributed systems.
5) Kotlin has a larger and more active community than OCaml, leading to more readily available resources, libraries, and frameworks.
6) Kotlin integrates seamlessly with the Java Virtual Machine (JVM), allowing you to leverage existing Java libraries and frameworks. This can be beneficial for projects with existing Java code or requiring interoperability with Java applications.
7) Third, Kotlin's type system is more flexible and expressive than OCaml's, enabling advanced type-level programming techniques and cleaner code.
8) Kotlin has an excellent Integrated Development Environment (IDE) compared to OCaml. IntelliJ IDEA and Android Studio are two excellent IDEs that facilitate software development in Kotlin.

B. Formal Behavioural Specification of Desired Behavior Using Petri-nets

This project employs formal behavioural specification using Petri-Nets to define the system's desired behaviour. Specifically, the following requirements are encoded using Petri-Net models:

1) Leader Election:

- The model guarantees that at most one leader exists at any given time.
- Transitions between leaders are modelled to occur efficiently, ensuring minimal downtime during leadership changes.

2) Command Execution:

- Petri-Net models enforce the execution of commands on all nodes in the same order.
- This ensures that the system maintains a consistent state across all nodes, even in the presence of concurrent commands.

3) Availability:

- The system is designed to remain available even when nodes fail or network partitions occur.

- Petri-Net models capture the system's behaviour under these conditions, ensuring graceful degradation and continued operation.

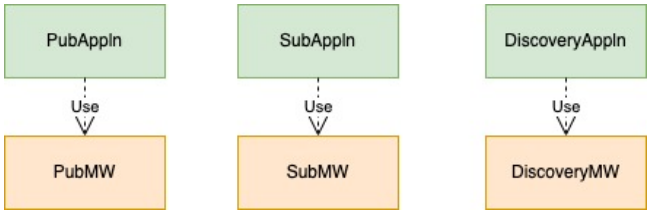
4) Log Consistency:

- The replicated log serves as the single source of truth for the system's state.
- Petri-Net models guarantee that all nodes agree on the contents and order of entries within the replicated log, even if crashes or restarts occur.

This formal approach to behavioural specification provides several advantages. It allows for unambiguous communication of system requirements, facilitates rigorous verification and validation, and enables efficient simulation and analysis of the system's behaviour under various conditions. By leveraging Petri-Nets, the project ensures that the system adheres to its intended behaviour and delivers robust performance.

IV. System Architecture

A. Overview of the complete system



A novel approach has been used to implement a Publisher/Subscriber architecture in Kotlin. The architecture leverages the power of the Remote Procedure Call (RPC) pattern but addresses the anonymity limitations of RabbitMQ. While RabbitMQ excels at message delivery, it requires explicit connections between publishers and subscribers, hindering the ideal decoupling of the publish/subscribe pattern.

To address this issue and achieve greater anonymity, a dedicated middleware layer has been introduced. This lightweight layer sits on RabbitMQ, shielding the application logic from the underlying messaging details. As a result, publishers and subscribers interact solely with the middleware API, remaining completely anonymous to each other. The middleware then handles the necessary communication and routing through RabbitMQ, ensuring efficient message delivery without compromising anonymity.

Further enhancing the system's robustness and scalability, the project implements warm-passive fault tolerance for the Discovery service. This is achieved by integrating Kafka for using its KRaft consensus protocol, facilitating coordinated decision-making amongst multiple servers. This ensures that even if one server fails, the system continues to operate smoothly, promoting high availability and resilience.

Leveraging these design patterns, the project offers substantial advantages compared to other existing methods. Separating concerns between application logic and middleware promotes cleaner code structure and easier maintenance. Furthermore, the anonymity feature provides enhanced security and flexibility, while the Raft consensus protocol guarantees reliable operations despite failures. These advancements contribute to a more robust, scalable, and efficient Publisher/Subscriber architecture.

B. Integration of components

1) Inter-Component Communication: The system uses message passing as its primary communication protocol. This enables asynchronous and decoupled communication between components, promoting modularity and scalability. Components communicate by sending and receiving messages through message queues, ensuring efficient and reliable data exchange.

2) Data Flow: Data flows through the system in a well-defined manner. Publishers send messages to the message bus, a central hub for routing messages to subscribed components. Subscribers register their interest in specific topics and receive relevant messages through the message bus. This ensures subscribers receive only the desired messages, minimizing unnecessary data transmission.

3) Component Interfaces: Each component exposes a well-defined interface that outlines its functions and services to other components. This promotes a modular architecture where components interact without tight coupling, simplifying development and maintenance. The interfaces utilize standard data structures and communication protocols, enabling interoperability with other systems.

C. Workflow description

To address the anonymity issue and achieve a clear separation of concerns, a multi-layered architecture was designed:

- Application Logic (Appln.kt): This layer handled application-specific functionalities, oblivious to the underlying communication details.
- Middleware (MW.kt): This layer acted as an abstraction, providing an API for anonymous communication and RabbitMQ interaction.

The Remote Procedure Call (RPC) pattern was employed for efficient communication between components.

The focus shifted to implementing anonymity within the middleware. Techniques were devised to mask the identities of publishers and subscribers, ensuring secure and private messaging. RabbitMQ was chosen as the underlying messaging platform for its reliable, ordered message delivery while remaining transparent to the application logic through the middleware layer.

Warm-passive fault tolerance was implemented for the Discovery service, a crucial component for maintaining network information. The KRaft consensus protocol was chosen for its ability to coordinate decision-making among multiple servers and its simplicity compared to other consensus protocols, such as Paxos and Practical Byzantine Fault Tolerance (PBFT), ensuring consistency and high availability even in server failures.

V. Discussion

A. Interpretation of results

B. Limitations of the current study

Most notably, there is no formal verification tool specifically designed for Kotlin. While the current implementation demonstrates promising performance and efficiency, further optimi-

sation opportunities exist for enhancing system performance and resource utilization.

1) Explicit Memory Management: The current implementation utilizes implicit garbage collection, which can lead to unpredictable performance overhead. Implementing explicit memory management, where the programmer takes control of memory allocation and deallocation, could offer significant performance gains, particularly in resource-constrained environments.

2) Optimization of Data Marshalling/Unmarshalling: The current architecture involves marshalling and unmarshalling data between different layers, which can introduce additional processing overhead. Exploring techniques to minimize or eliminate this overhead, such as shared memory or message queue optimizations, could lead to smoother system operation and improved performance.

3) Parallel Data Transmission and Buffering: The current implementation transmits data sequentially, which can introduce bottlenecks in high-throughput scenarios. Implementing parallel data transmission with efficient buffering mechanisms can significantly improve data transfer rates and ensure seamless execution.

4) Message Header Compression: Message headers carry necessary metadata but can affect message size and communication overhead. Message header compression techniques can significantly reduce bandwidth consumption and improve overall system efficiency.

5) Code Optimisation and Grouping: Frequently used code sequences within the system can be grouped and optimized for faster access and execution. This can involve code inlining, loop optimizations, and other techniques tailored to the specific code structures.

The system can improve performance, enhance resource efficiency, and remain competitive in demanding real-world applications by exploring and implementing these optimisation strategies. Performing these optimisations manually will require a lot of workers and time; hence, we shall automate these processes as much as possible.

Formal verification tools can be used for static optimisation, enabling partial optimisation process automation. The component-based and model-driven approach often leads to the excessive layering of code, which causes inefficiency. Hence formal verification can be used to optimise code efficiency layer by layer, relying heavily on automated theorem proving for efficiency. However, human intervention remains crucial for checking the appropriateness of applied optimisations, such as function inlining and directed equality substitution. This ensures that the optimised code remains semantically equivalent to the original while improving performance. Combining automatic and human-driven optimisation strategies within the Stainless tool provides a powerful approach for enhancing code efficiency and maintaining correctness.

Formal verification tools can also be used for dynamic optimisation, focusing on reducing unnecessary layers for a streamlined architecture. This optimisation is primarily automated through theorem proving, identifying and eliminating redundant layers. A concept known as Common Collapse Patterns (CCPs) can be applied to identify these redundant layers. When a specific condition, the Common Case Predicate (CCP), is met, a ”bypass code” could be automatically generated. This bypass code effectively jumps over unnecessary layers, significantly boosting performance for frequently occurring scenarios. However, human intervention remains critical for verifying the CCPs and ensuring the generated bypass code complies with all system requirements and security considerations. This combination of automation and human oversight allows for efficient and reliable dynamic optimisation within the Stainless tool.

VI. Related Work

In addition to Xiaoming et al.'s research on reliable and efficient communication systems, other relevant work has been done [16]. For example, J. R. Wilcox et al. used Verdi, a framework for verifying distributed systems in Coq, to implement the Raft consensus algorithm. They then automatically generated equivalent OCaml code using Coq's extraction feature [6]. Thus, at least two independent implementations of Raft in OCaml exist. Mitsunori Komatsu developed ORaft [8], while Heidi Howard created ocaml-raft [4]. There are multiple independent implementations of Raft in Scala as well. One of the most well-known example of raft implementation in Scala is zio-raft by Aris Koliopoulos [7].

VII. Future work

Our project aims to extend its scope beyond the current implementation with short-term and long-term plans. Regarding short-term plans, JSON can be replaced by Protocol Buffers (Protobuf) for higher performance and less resource consumption. Messages in Protobufs are often much smaller than their corresponding JSON counterparts, leading to reduced bandwidth usage and faster transmission times. This is because Protobuf uses binary encoding, which packs data more efficiently than the text-based format of JSON. Moreover, parsing and serialization operations are significantly faster with Protobuf than with JSON. This is due to the pre-defined schema and binary encoding of Protobuf, which allows for optimized parsing routines. Besides, Protobuf messages are strongly typed, meaning that each field has a specific data type defined in a schema file. This helps to prevent data corruption and ensures that messages are interpreted correctly by all applications. JSON, on the other hand, is dynamically typed, which can lead to errors if the data is not formatted correctly.

Another next step in this project is to automate converting Petri-Nets into Kotlin code. This will eliminate manual effort, significantly improve efficiency, and allow for more rapid development and experimentation.

The proposed approach leverages the strengths of functional programming paradigms, resulting in concise and expressive code. Additionally, the formal semantics of Petri-Nets guarantee the correctness and reliability of the generated code, ensuring that the system behaves as intended without compromising functionality.

The automated code generation process will facilitate easier development and maintenance by employing high-level operations and data structures. Furthermore, automatic garbage

collection and memory allocation will free developers from low-level concerns, allowing them to focus on higher-level design and implementation.

This automation will streamline the development process and open doors for further research and exploration within Petri-nets-based software engineering.

There are three main ways to convert a model such as Petri-nets into code:

1) Visitor-based: This approach uses the visitor pattern, commonly found in object-oriented programming, to process each element of the input structure and generate corresponding code. A similar technique is recursion, which is often preferred in functional languages.
2) Template-based: This approach focuses on the final output program. The complete code is written out, except for specific placeholders that the generator fills in based on the input structure.
3) Hybrid: This approach combines the benefits of both visitor-based and template-based methods. It uses a template for the overall program structure and applies algorithmic traversals of the input data to fill in the details [16].

Further work is needed to figure out which method is most appropriate for converting Petri-Nets into Kotlin will be required.

In terms of long-term trajectory, more complex software systems could be developed with this proposed methodology. For instance, it can be applied to implement complex consensus algorithms such as Raft, simple Paxos, Mutli-Paxos, Disk Paxos, Cheap Paxos, Fast Paxos, Generalized Paxos, Stoppable Paxos and Mencius [15]. Thus, the proposed software engineering methodology could be applied to implementing key-value database management systems, showcasing the versatility and robustness of our approach in practical applications. Additionally, further optimization techniques could be explored, potentially integrating other tools and frameworks to enhance the performance and reliability of our methodology. We would also like to employ our methodology to safety-critical embedded systems and algorithms, potentially transforming how these systems are developed, optimized, and verified for safety and performance.

VIII. Conclusion

This paper presents the design and development of a scalable and anonymous Pub/Sub system in Kotlin, using a model-driven and component-based approach with Apache Kafka for KRaft distributed consensus to provide fault-tolerant message delivery. The system leverages Petri nets for formal modelling and verification, guaranteeing correct behaviour and reliable message exchange. Furthermore, the RPC design pattern promotes modularity and maintainability, while the implementation of anonymity mechanisms ensures privacy and decoupling between publishers and subscribers.

The implemented Pub/Sub system demonstrates the potential of integrating model-driven and component-based methodologies for building robust and scalable distributed systems. The combination of Scala's multi-programming paradigm, KRaft for fault tolerance, and anonymity mechanisms provides a strong foundation for secure and reliable message exchange in diverse applications.

Overall, this project has demonstrated the potential and benefits of a model-driven and component-based approach to building a complex software system. By leveraging Scala, KRaft, and formal modelling, the system offers a robust and efficient solution for applications requiring reliable and secure message exchange in distributed environments.

IX. Acknowledgements

My gratitude goes to four esteemed professors and one dedicated teaching assistant at Vanderbilt University who have impacted my academic journey.

First, I express my most profound appreciation to Professor Abhishek Dubey and Samir for their invaluable support in CS 6376 - Foundation of Hybrid and Embedded Systems (Fall 2023). Their support played a crucial role in trying out the research proposal that I have made.

Second, I am immensely grateful to Professor Aniruddha Gokhale for teaching CS 6381 - Distributed Systems Principles. His insightful instruction and mentorship broadened my knowledge of distributed systems and aided me in navigating my academic path at Vanderbilt University.

Third, I extend my heartfelt thanks to Professor Taylor Johnson for teaching CS 6315 - Automated Verification (Spring 2023). The knowledge and skills acquired in this course will be a valuable foundation for my future research endeavours.

Fourth, I was honoured to learn from Professor Douglas Schmidt. He taught me CS 5253 - Parallel Functional Programming in Fall 2023. The knowledge and skills acquired in this course were valuable in completing this project.

Their dedication to teaching and their commitment to their students have inspired me. I am deeply indebted to them for their support and guidance.

References

[1] Chandra, Tushar, et al. "Paxos Made Live: An Engineering Perspective." Annual ACM Symposium on Principles of Distributed Computing: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing; 12-15 Aug. 2007, ACM, 2007, pp. 398–407, https://doi.org/10.1145/1281100.1281103.

[2] D. Ongaro. Consensus: Briding Theory and Practice. PhD thesis, Stanford University, Aug. 2014.

[3] D. Ongaro. The Raft consensus website, Nov. 2014. http://raft.github.io/.

[4] Howard, Heidi. "Analysis of Raft Consensus." Technical Report No. 857, Computer Laboratory, University of Cambridge, July 2014. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf

[5] Jensen, Kurt. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. 1st ed., Springer Nature, 2009, https://doi.org/10.1007/b95112.

[6] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and verifying distributed systems. In PLDI 2015, June 2015.

[7] Koliopoulos, Aris. "Implementing Raft using ZIO in Scala." Aris K. https://softwaremill.com/implementing-raft-using-a-functional-effect-system/, 19 Nov. 2021, https://ariskk.com/scala-raft-zio/. Accessed 8 Dec. 2023.

[8] Komatsu, Mitsunori. "ORaft: Raft Consensus Algorithm in OCaml." GitHub repository, https://github.com/komamitsu/oraft. Accessed 8 Dec. 2023.

[9] Lamport, Leslie. "Paxos made simple." ACM Sigact News 32, no. 4 (2001): 18–25.

[10] Lamport, Leslie. “The Part-Time Parliament.” ACM Transactions on Computer Systems, vol. 16, no. 2, 1998, pp. 133–69, https://doi.org/10.1145/279227.279229..
[11] Ongaro, Diego, and John Ousterhout. ”In Search of an Understandable Consensus Algorithm.” 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, Philadelphia, PA, pp. 305-319. USENIX Association, https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.
[12] Patterns of Distributed Systems. Addison-Wesley Professional, 2023.
[13] Planning for change in a formal verification of the Raft consensus protocol” by Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. In CPP 2016: 5th ACM SIGPLAN Conference on Certified Programs and Proofs, (St. Petersburg, FL, USA), Jan. 2016, pp. 154-165.
[14] S. J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In Foundations of Component-based Systems. Cambridge University Press, 2000.
[15] Van Renesse, Robbert, and Deniz Altinbuken. “Paxos Made Moderately Complex.” ACM Computing Surveys, vol. 47, no. 3, 2015, pp. 1–36, https://doi.org/10.1145/2673577.
[16] Wasowski, Andrzej, and Thorsten Berger. Domain-Specific Languages: Effective Modeling, Automation, and Reuse. 1st ed., Springer International Publishing AG, 2023, https://doi.org/10.1007/978-3-031-23669-3.
[17] XIAOMING LIU, et al. “Building Reliable, High-Performance Communication Systems from Components.” Operating Systems Review, vol. 33, no. 5, Association for Computing Machinery, 1999, pp. 80–92, https://doi.org/10.1145/319344.319157