PubSub with Raft in Scala

CS 6376 Foundations of Hybrid and Embedded Systems Final Project

Youngjae Moon

VANDERBILT UNIVERSITY

Contents

- Motivation
- Steps involved in this component based SWE methodology
- Domain chosen – Raft
- Project Goals
- Etcd for Raft distributed consensus
- Overall software design
- Behavioral Specification using Petri-Nets
- Verification by Petri-Nets
- Key Takeaways
- Code Generation: Why Scala
- Future Work
  - Advanced Optimization
  - Stainless for Static Optimization
  - Stainless for Dynamic Optimization
  - Automatic Code generation
  - Other Ideas

Motivation

- Programming in Java is harder to code than other JVM-based languages such as Scala or Kotlin.
- I found a way to code with Java-like efficiency without using the same methods from Xiaoming Liu et al.'s paper – “Building reliable, high-performance communication systems from components.”
- It's not the same, but it borrows key ideas.

Steps invovled in this SWE methology

1. Specify through UML diagram and Petri-nets
2. (Automatically) Generate unoptimized Scala code from the Petri-net model.
3. Perform code optimization of the Scala code using the Stainless verification tool, enhancing system efficiency and reliability of the code, and leverage Scala’s multi-paradigm capabilities for efficient, Java-like performance, and ensuring functional equivalence for reliable deployment.
4. Technologies used: Petri-nets, Scala, Stainless

Domain chosen – Raft

- Final aim of this project: Build a tool to implement distributed systems, like Raft, easier.
- Current methods are like building with bricks – complex and not ideal for low-level languages.
- My approach is like using LEGO® – simpler and lets you optimize communication without layering hassles.
- More ways to optimize than you might think – check slides 17-19 for details!

Project Goals

1. Develop and implement a Publish/Subscribe architecture using Petri nets and the Scala programming language.
   a. This architecture leverages Raft algorithm for the consensus protocol to achieve robust and scalable message delivery.
2. Model a portion of the software system as Petri nets.
   a. This will enable the verification of the model's correctness, ensuring the system's functionality.
3. Generate the initial, unoptimized Scala code in RPC design pattern.
   a. This approach promotes cleaner separation of concerns, improves anonymity and fault tolerance, and ultimately makes the system more robust and scalable.

Etcd for Raft distributed consensus

- Message queueing
  - Used RabbitMQ message queueing framework for reliable, ordered message delivery.
- Data format
  - Used JSON for data serialization/deserialization.
- Structure
  - Leader/Followers: Has a designated leader who tells everyone what to do. Everyone else follows.
- Calls to Action
  - RPC (Remote Procedure Call): Imagine a Raft node asking another node to do something (vote, add a log entry, etc.) like a remote function call. ZeroMQ delivers these requests and responses.
  - Raft Messages as RPCs: Turn each Raft message type (e.g., vote request, add log entry) into an RPC call for cleaner code

Overall software design

- Have implemented a Publisher/Subscriber architecture in Scala using the Remote Procedure Call (RPC) pattern. To achieve better decoupling, I split the code into two main categories:
  - ~Appln.scala: Contains all application-level logic.
  - ~MW.scala: Contains all middleware-level code, including RabbitMQ communication (~MW files).
- While RabbitMQ excels at messaging, it lacks anonymity. Subscribers must explicitly connect to publishers by IP and port, compromising the ideal decoupling of publish/subscribe (time, space, and synchronization independence).
- Therefore, I sought a solution where publishers and subscribers remain anonymous to each other. Of course, someone needs to manage these associations, technically breaking the "ideal" definition, but as long as the application logic adheres to it, that's acceptable.

Overall software design

- To achieve this, I designed a middleware layer. Instead of directly using RabbitMQ, application logic (~Appln.scala files) interacts with the middleware's API. This lightweight pub/sub middleware sits on top of RabbitMQ, enabling anonymity.
- Furthermore, I've implemented warm-passive fault tolerance for the Discovery service using the Raft consensus protocol for coordination.
- This approach promotes cleaner separation of concerns, improves anonymity and fault tolerance, and ultimately makes the system more robust and scalable.

Behavioral Specification using Petri-Nets

- Requirements-driven
  - Leader election: Ensure at most one leader at a time, with efficient transitions between leaders.
  - Command execution: Ensure all nodes execute commands in the same order, following the replicated log.
  - Availability: Maintain system availability even with node failures and network partitions.
  - Log consistency: Guarantee all nodes agree on the replicated log, even during crashes and restarts.

Behavioral Specification using Petri-Nets

- Context: Message Publishing Process in PubMW.
- Modeling Scope: The process of establishing a connection, declaring an exchange, and publishing a message.
- Petri Net Design:
  - Places: Connection Idle, Connection Established, Exchange Declared, Message Ready, Message Published.
  - Transitions: Establish Connection, Declare Exchange, Prepare Message, Publish Message.
  - Tokens: Represent the state of the connection and message readiness.
  - Arcs: Connect transitions to places, representing the flow from establishing a connection to message publication.

Behavioral Specification using Petri-Nets

- Context: Message Subscription and Consumption in SubMW.
- Modeling Scope: The process of connecting to the server, setting up a subscriber queue, and consuming messages.
- Petri Net Design:
  - Places: Connection Idle, Connection Established, Queue Setup, Listening for Messages, Message Received.
  - Transitions: Connect to Server, Setup Queue, Start Listening, Receive Message.
  - Tokens: Indicate the state of the subscription process and message reception.
  - Arcs: Show the sequence from connection establishment to message consumption.

Behavioral Specification using Petri-Nets

- Context: Leader Election Process in Etcd (Raft algorithm)
- Modeling Scope: The leader election process, including candidacy declaration, voting, and leader establishment.
- Petri Net Design:
  - Places: Idle, Candidate, Voted, Leader Elected, Follower.
  - Transitions: Declare Candidacy, Vote, Elect Leader, Become Follower.
  - Tokens: Represent the state of each node in the Etcd cluster.
  - Arcs: Illustrate the transitions between different states in the election process.

Behavioral Specification using Petri-Nets

- Context: State Replication in Etcd (Raft)
- Modeling Scope: The process of replicating state across nodes in the Etcd cluster.
- Petri Net Design:
  - Places: Leader Idle, Log Entry Created, Log Entry Replicated, Commit Entry.
  - Transitions: Create Log Entry, Replicate Entry, Commit Entry.
  - Tokens: Indicate the state of the log entry from creation to commitment.
  - Arcs: Depict the flow of log entries across the Raft cluster.

Verification by Petri-Nets

- Used reachability analysis to confirm leader election eventually reaches a stable state.
- Future work:
  - Use liveness analysis to ensure log entries are eventually replicated and committed on all nodes.
  - Future work: Use boundedness analysis to guarantee the Log and Vote Granted tokens remain finite.
  - Use model checking to verify specific properties like safety (no inconsistencies) and liveness (desired behaviors always occur).

Key Takeaways

- Petri-Nets is not sufficient for modeling complex software systems and hence other methods are used in addition (prior) to modelling in Petri-Nets.
  - E.g., UML diagram
- Not executable yet
  - Petri-Nets provide a high-level framework for Raft's behavior, not a concrete implementation.
  - Specific message formats JSON, timeouts, and error handling need further design.
  - Petri-Nets serve as a blueprint for verifying and reasoning about Raft's correctness and Pub/Sub architecture's correctness.
  - Not a production-ready codebase.

Code Generation: Why Scala?

- Scala supports both OOP and FP.
  - OOP can be used to organize code into well-defined modules, making it easier to design, understand and maintain.
  - FP’s immutability and pure functions lead to predictable and less error-prone code, simplifying reasoning and debugging.
  - Breaking down complex tasks into smaller, reusable functions promotes code modularity and facilitates parallel execution.
  - Small, well-defined functions and objects are easier to isolate and test in isolation, improving test coverage and confidence.
  - Parallel programming can be most effectively done with functional programming and lazy evaluation.
- Scala has a rich-set of libraries.
- IntelliJ IDEA is a well-designed IDE for Scala programming.

Future Work: Advanced Optimization

- Future Optimization Ideas:
  - Ditch implicit garbage collection and take control with explicit memory management for peak performance.
  - Eliminate the overhead of marshalling/unmarshalling data between layers for smoother operation.
  - Buffer data while transmitting in parallel for seamless execution.
  - Compress message headers for leaner communication.
  - Group frequently used code sequences together for faster access and improved efficiency.
  - These optimizations promise to squeeze even more performance and efficiency out of the project, making it a lean, mean, coding machine!
- Future work: How to automate the process?
  - Concepts from CS 6315 Automated Verification should be applied.
  - Plan to work on these tasks during this winter break with Professor Dubey.

VANDERBILT UNIVERSITY

Future Work: Stainless for Static Optimization

- Layer-by-layer optimization for efficiency.
- Mostly automated with theorem proving, but human intervention still needed for checking appropriateness.
- E.g. Function in-lining, directed equality substitution, and other functional magic for simpler code.

Future Work: Stainless for Dynamic Optimization

- Layer Reduction:
  - Involves identifying and eliminating redundant layers, resulting in a streamlined architecture.
- Automation with Theorem Proving
- Identifying Common Collapse Patterns (CCP)
- Bypass Code Generation:
  - When a specific condition, known as the Common Case Predicate (CCP), is met, the system automatically generates "bypass code."
  - This bypass code facilitates directly jumping over redundant layers, significantly improving performance for frequent scenarios.
- Human Intervention for final Verification
  - Developers are responsible for validating the CCPs and ensuring that the generated bypass code adheres to system requirements and security considerations.

Future work: Automatic Code generation

- Automate the process of turning Petri-Nets into Scala code, eliminating manual effort and boosting efficiency.
  - I will do this on next semester for CS 5278 Principles of Software Engineering Final Project.
- Functional programming paradigms will lead to concise and expressive code.
- Formal semantics will guarantee code correctness and reliability.
- Should not sacrificing functionality.
- Leverage high-level operations and data structures will allow for easier development.
- Automatic garbage collection and memory allocation will prevent low-level concerns.

Future Work: Other Ideas

- Use Protocol Buffers instead of JSON to reduce network bandwidth and storage requirements.
- Implement a wider range of sophisticated consensus algorithms like Single/Multi/Disk/Cheap/Fast/Generalized/Stoppable Paxos and Mencius, unlocking new possibilities for distributed systems.
- Build a key—value database using our Petri-Net to Scala pipeline, leveraging its inherent security and reliability.
- Delve deeper into optimization techniques, potentially integrating other tools to push the performance and dependability of our approach to new heights across diverse domains.

Thank you

Any Questions?

VANDERBILT UNIVERSITY