Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Publish\Subscribe Cache
Using Caffeine Cache and Apache Pulsar

Report: Distributed Systems

2023/2024

Gianfranco Branco - 1025565 - gianfranco.branco@studio.unibo.it

Contents

Chapter 1

Introduction

1.1 Objective

The project focuses on developing a library that implements a Publish\Subscribe cache system, leveraging Apache Pulsar and Caffeine Cache. This library serves as a practical demonstration of the foundational principles of distributed caching, emphasizing data propagation, consistency, and fault management. The objective is to explore how these technologies can collaborate to enable efficient data distribution and synchronization in real-time scenarios.

1.2 Definition of the Distributed Cache Library

The library integrates two core technologies: Apache Pulsar, a sophisticated platform for real-time event-driven communication, and Caffeine Cache, a high-performance solution for in-memory data storage. The project is primarily educational, aiming to provide an in-depth understanding of the dynamics of distributed caching without targeting an enterprise-grade solution.

Apache Pulsar ensures real-time data propagation across nodes, allowing efficient synchronization and reliable communication even under high traffic loads. Caffeine Cache enhances performance by storing frequently accessed data in memory, minimizing the need for queries to central storage and reducing latency.

Key aspects of this library include state propagation, anomaly detection, and fault tolerance, all critical components of any distributed system. The system ensures consistency across nodes, allowing each one to access an updated and synchronized view of the data. This approach reduces the risk of inconsistencies, even during frequent updates or system disruptions, and strengthens the system's overall reliability.

Ultimately, this project provides a functional and educational example of managing client-server interactions within a distributed system, focusing on consistency, resilience, and real-time performance. It highlights caching strate-

gies and fault tolerance mechanisms to maintain the efficiency and stability of a distributed caching environment.

1.3 Operational Scenarios

- **Message Subscription and Publishing:** Clients subscribe to specific Apache Pulsar data channels to receive real-time updates. Upon significant data changes, the system broadcasts updated messages to subscribers. This ensures near-instantaneous synchronization across all nodes and clients, supporting large-scale information consistency.
- **Local Caching:** When a user requests data available in the local Caffeine Cache, it is immediately retrieved from memory, bypassing the central database. This significantly reduces latency and alleviates the load on the central system.
- **Cache Synchronization with Apache Pulsar:** Data updates received via Apache Pulsar automatically trigger updates to the local cache. Subscribed nodes process incoming messages in real-time, ensuring consistent and synchronized data across the system.
- **Node State Management:** The system monitors node state changes (e.g., active to inactive) and propagates these events to all other nodes and the central system. This mechanism ensures system-wide awareness of node availability and operational continuity.
- **Cache and Node State Monitoring:** Regular integrity checks of the cache and node connections detect errors or disruptions. Notifications are sent to the central system, facilitating prompt issue resolution and maintaining data availability.
- **Data Query and Manipulation API:** External clients access or modify cached data through APIs that interact with both Apache Pulsar and Caffeine Cache. This dual interaction ensures synchronized and consistent data management across the distributed system.

1.4 Non-Functional Scenarios

- **Scalability and Elasticity:** The system dynamically scales resources to accommodate peak demands, such as by increasing the number of nodes or expanding cache resources. This ensures optimal performance under varying workloads and adapts to real-time operational needs.
- **Reliability and Fault Tolerance:** In the event of a node failure, the system triggers automatic recovery mechanisms, including reintegration of the affected node. Redundancy and failover strategies maintain system integrity and continuity during temporary disruptions.

- **Continuous Monitoring and Logging:** The system continuously monitors key metrics and logs critical events, such as synchronization activities and node state changes. These logs facilitate anomaly detection, swift troubleshooting, and detailed performance analysis.

Chapter 2

Requirements

Functional Requirements

1. **Subscription and Message Publishing**
   - Ability to subscribe to data channels (topics) on Apache Pulsar for receiving real-time updates.
   - Mechanism to send updated messages from the cache to subscribers when significant data changes occur.
2. **Local Caching**
   - Implementation of an in-memory cache using *Caffeine Cache* for local data storage.
   - Support for data removal functionality.
3. **Data Synchronization**
   - Automatic updating of the local cache when new messages are received from Apache Pulsar.
4. **Node State Event Management**
   - Propagation of state change events (e.g., active or inactive) for each node to ensure visibility of these changes across all nodes and the central system.
5. **Cache State Control**
   - Regular checks on cache and node statuses to maintain system reliability.
   - Notifications to the central system in case of synchronization errors or interruptions.

6. **APIs for Data Querying and Manipulation**
    - Provision of APIs for querying and modifying cache data, enabling direct interaction with Apache Pulsar and Caffeine Cache for real-time updates and operations.

Non-Functional Requirements

1. **Performance and Low Latency**
    - Utilization of *Caffeine Cache* to guarantee low-latency access to in-memory data.
    - Optimization of communication with Apache Pulsar to minimize message transmission time and reduce network load.
2. **Scalability**
    - Support for horizontal scaling of both the cache and the Pulsar messaging system to accommodate a growing number of users and data volume.
3. **Reliability and Redundancy**
    - Implementation of failover mechanisms and redundancy to ensure continued operation despite node failures.
4. **Data Consistency**
    - Maintenance of data consistency across nodes during synchronization and state update propagation.
    - Accurate synchronization of updates to ensure all nodes have a unified view of the cache state.
5. **Maintainability and Configurability**
    - Adoption of a modular and configurable architecture to facilitate the addition of new nodes and adjustment of configurations (e.g., cache expiration policies).
6. **Monitoring and Logging**
    - Real-time monitoring of system performance, with detailed logs of key events such as synchronization and node state changes.
    - Immediate notifications for errors or anomalies to enable prompt resolution.
7. **Multi-Platform Support**
    - Compatibility with multi-platform environments, ensuring smooth operation on both cloud-based and on-premises infrastructures.

8. **Elasticity**
    - Dynamic adjustment of resources for cache and message handling to manage varying workloads during peak periods.
9. **Fault Tolerance**
    - Resilience to single-node failures, with automatic recovery and reintegration of nodes into the system.

2.1 Motivation for Technological Choices

The selection of Apache Pulsar and Caffeine Cache as core components for implementing the distributed cache library is justified by their robust features and alignment with the system's requirements for real-time distributed operations.

2.2 Why Apache Pulsar

Apache Pulsar is a highly scalable, low-latency messaging system suitable for real-time data streams. Initially developed by Yahoo and now maintained by the Apache Software Foundation, Pulsar supports publish-subscribe scenarios ideal for distributed systems requiring seamless communication.

Figure 2.1: Apache Pulsar Logo

Key advantages include:

- **Low Latency:** Designed for minimal delay, enabling nearly real-time updates for cache synchronization.
- **High Scalability:** Handles millions of topics and messages per second, making it apt for scaling with system growth.
- **Multi-Tenant Architecture:** Supports isolated data and resource management for multiple applications.
- **Message Persistence:** Ensures durability through disk-based storage using Apache BookKeeper, safeguarding messages even during node failures.
- **Fault Tolerance:** Automatically redistributes messages in case of node failures, ensuring consistent system operation.

2.3 Why Caffeine Cache

Caffeine Cache is a high-performance, in-memory caching library for Java. It provides fast data access, sophisticated expiration policies, and is highly suited for real-time caching needs.

Benefits of using Caffeine Cache:

- **High Performance:** Delivers extremely fast access times, ideal for GET and PUT cache operations.
- **Memory Efficiency:** Optimizes RAM usage, avoiding resource exhaustion.
- **Configurability:** Allows fine-tuning of parameters like cache size and expiration policies.
- **Data Lifetime Management:** Automatically handles expiration and invalidation of stale data.
- **Ease of Integration:** Seamlessly integrates with Java frameworks, supporting efficient in-memory operations.

Chapter 3

Design

3.1 Component Overview

The diagram below illustrates a high-level overview of the system's components, entities, and their interactions.

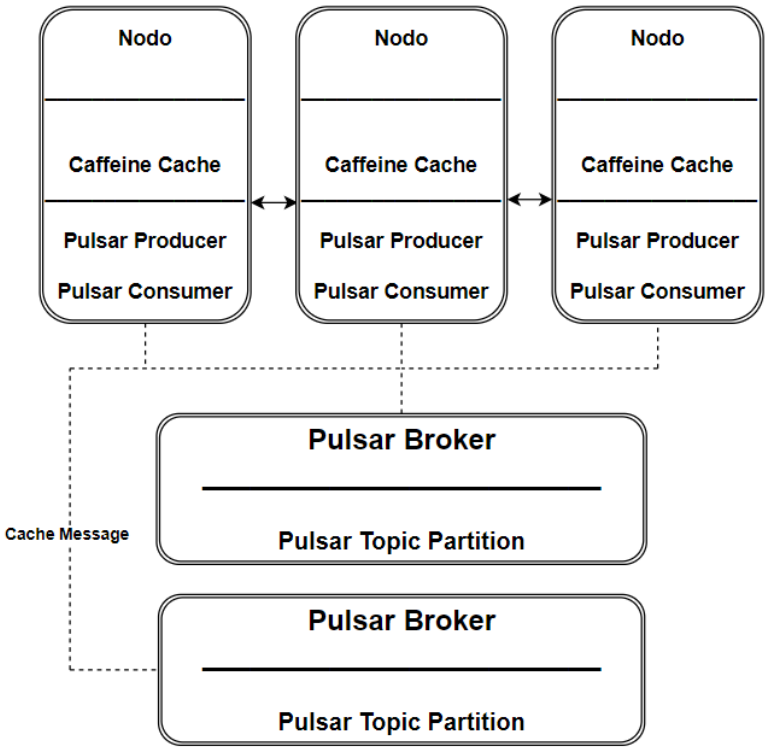

Figure 3.1: System Components

Figure 3.1 depicts the architecture of the main components in the distributed caching system. While three nodes are shown for illustration, the system is designed to scale horizontally, supporting a flexible number of nodes (n) as needed.

Node

A **Node** is an independent unit within the distributed system. It contains a local cache and the logic required for communication with other nodes. Nodes work collaboratively, utilizing Apache Pulsar to synchronize caching operations efficiently and consistently. Each node is composed of:

- **Caffeine Cache**: A local cache implemented with the Caffeine library to store and manage data within the node.
- **Pulsar Producer and Pulsar Consumer**: These components handle message production and consumption with the Apache Pulsar messaging system. The Pulsar Producer sends cache updates or invalidation messages to other nodes, while the Pulsar Consumer receives and processes incoming messages to reflect updates or changes in the local cache.

Pulsar

Apache Pulsar serves as the distributed messaging backbone for inter-node communication. Messages are organized into topic partitions, allowing nodes to process them in parallel and at scale. This partitioning ensures system resilience and enables efficient handling of high message throughput.

Cache Message

Cache Messages act as the communication mechanism between nodes. These messages contain key details, such as the action to be performed (e.g., PUT, FETCH REQUEST), the sender node's identifier, the key-value pair, and a timestamp indicating the event's occurrence. Pulsar facilitates the exchange of these messages, enabling nodes to update or invalidate their local caches in response to system-wide changes.

This architecture, based on autonomous nodes communicating through Apache Pulsar, ensures horizontal scalability by seamlessly adding new nodes. The distinct responsibilities of the Pulsar Producer and Pulsar Consumer within nodes allow for efficient message exchange, contributing to a robust and consistent distributed caching system.

3.2 System Architecture

To meet the outlined objectives and requirements, the system is composed of the following key components:

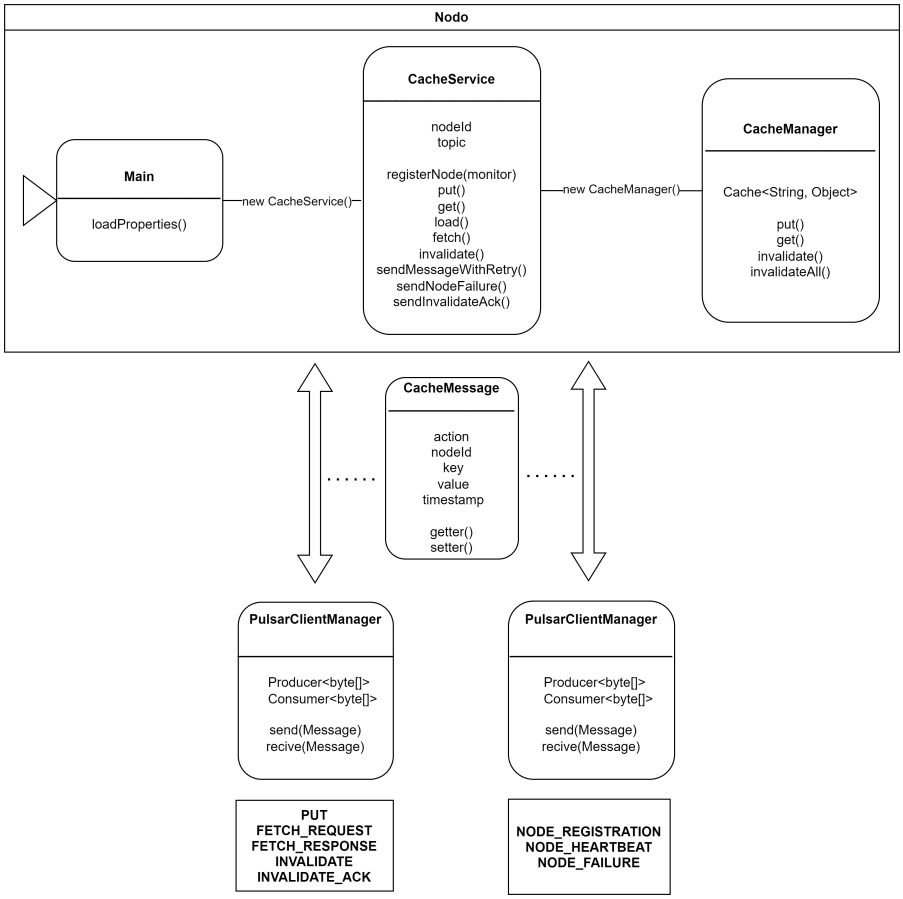

Figure 3.2: System Architecture Components

Node

A **Node** represents the foundational unit of the distributed system. It includes both a local cache and logic for communication with other nodes. Each node contains the following components:

- `CacheManager`: Manages the node's local cache, performing operations like `put`, `get`, and `invalidate`.
- `CacheService`: Facilitates synchronization with other nodes, ensuring consistency across the distributed system by propagating cache updates or invalidations.

Nodes collaborate to ensure data consistency across the system, propagating any cache changes to other nodes effectively.

CacheService

The `CacheService` is the core component of the system, responsible for managing cache operations such as `put`, `get`, `load`, `fetch`, and `invalidate`. It also handles node registration, inter-node communication, and acknowledgments (e.g., invalidation ACKs). The `CacheService` ensures seamless information exchange among nodes and manages error scenarios, such as unresponsive nodes.

CacheManager

The `CacheManager` is responsible for managing the local cache within a node. It handles data storage in memory using Caffeine Cache and implements operations like `put`, `get`, and `invalidate`. The `CacheManager` ensures rapid response times to data requests and appropriately invalidates local cache entries when needed.

PulsarClientManager

The `PulsarClientManager` manages communication between nodes through Apache Pulsar. Each node instantiates two instances:

- One instance handles message production and consumption for cache operations (e.g., PUT, FETCH REQUEST).
- The other instance manages state-related tasks, such as NODE REGISTRATION and NODE HEARTBEAT.

The `PulsarClientManager` also handles message serialization, ensuring all nodes receive and process updates consistently.

CacheMessage

A `CacheMessage` encapsulates the details of messages exchanged between nodes. It includes:

- The action to perform (e.g., PUT, INVALIDATE).
- The identifier of the originating node.
- Key-value data associated with the message.
- A timestamp indicating when the message was created.

This ensures nodes receive coherent messages and act in unison to maintain the consistency of the distributed cache.

Chapter 4

Implementation Details

This chapter outlines the implementation strategies adopted for the project, which is developed in Java. The primary components and their functionalities are described below.

4.1 Main

The `Main` class performs the following key operations:

- **Loading Configuration Properties:** The `loadProperties()` method reads a configuration file, whose path can be provided as a command-line argument. If no custom path is specified, or the file is not found, the application defaults to using system properties. These properties are set using `System.setProperty()`, making them globally available throughout the program.
- **Creating a `CacheService` Instance:** An instance of `CacheService` is initialized, which serves as the core component for cache management and integration with Apache Pulsar messages. The `CacheService` also sets up a Jetty server to expose REST APIs, allowing interaction with the cache.
- **Logger Initialization:** A `Logger` instance from the Java logging library is configured to handle logging, including information about property loading and error management.

4.2 API

The `API` provides RESTful endpoints designed to test and verify the correct functioning of the distributed cache system, enabling the execution of fundamental cache operations. Implemented using `Jakarta REST`, the APIs are accessible via HTTP and can be easily tested with tools like Docker and Postman, thereby simplifying the testing process.

The available endpoints include:

- **GET /cache/get/key** Retrieves the value associated with the specified key from the local cache. If successful, it returns the value in JSON format; otherwise, it responds with "Key not found."
- **GET /cache/fetch/key** Attempts to retrieve the value by exploring the distributed nodes. If the value is located on another node, it is returned. Any network or Pulsar-related issues result in an internal server error response.
- **POST /cache/load** Adds a new key-value pair to the cache. The endpoint accepts the key and value as query parameters, stores the entry in the local cache, and returns a confirmation message.
- **POST /cache/put** Provides an alternative for inserting or updating entries in the cache, ensuring flexibility in management operations.
- **DELETE /cache/invalidate/key** Removes the specified key from the cache. Upon completion, the system returns a confirmation; communication issues, particularly with Pulsar, trigger an internal server error response.

In detail, the `CacheApplication` class handles API configuration, exposing the endpoints and integrating with the `CacheResource` class, which implements the core operational logic.

4.3 CacheService

The `CacheService` class serves as the central component of the application, enabling nodes to share a distributed cache. This is achieved by publishing and subscribing to messages through Apache Pulsar. Each node can perform operations such as `put`, `get`, `invalidate`, and `fetch` on the cache.

The class manages two distinct Pulsar clients:

- One client handles sending and receiving messages related to cache operations, such as `PUT` and `FETCH_REQUEST`.
- The other client is responsible for node monitoring and communication, such as sending registration messages and periodic heartbeats to indicate node activity.

The core functionalities implemented in this class include:

- **Server Setup and Listener** In the constructor, a listener is set up to process incoming Pulsar messages, and the system is configured to monitor nodes' status. Additionally, a Jetty server is started, which listens on port 8080 and exposes RESTful APIs for interacting with the cache.

- **Adding Data to the Cache (`put`)** When a node adds a value to its cache using the `put` method, it broadcasts a `PUT` message to all other nodes. If the node is in *degraded mode* (e.g., experiencing connection issues with Pulsar), this operation cannot be performed.
- **Retrieving Data from Local Cache (`get`)** Nodes can retrieve values from their local cache using the `get` method. This operation does not involve sending messages to other nodes.
- **Fetching Data from Other Nodes (`fetch`)** If a requested key is not present in the local cache, a node can issue a `FETCH_REQUEST` message to other nodes through Pulsar. The node that holds the corresponding data responds with a `FETCH_RESPONSE` message. The retrieval process is managed asynchronously using `CompletableFuture`.
- **Cache Invalidation (`invalidate`)** To remove a key from the cache, a node sends an `INVALIDATE` message to other nodes. Upon receiving this message, each node invalidates the key locally and sends back an acknowledgment (`INVALIDATE_ACK`). The originating node tracks these acknowledgments to ensure all nodes have completed the invalidation.
- **Node Heartbeat and Monitoring Active Nodes** Nodes periodically transmit `NODE_HEARTBEAT` messages to indicate their activity. Additionally, nodes monitor active peers by processing `NODE_REGISTRATION` and `NODE_HEARTBEAT` messages. A node is considered inactive and removed from the active node list if it fails to send heartbeats within a specified time window.

4.4 CacheMessage

The `CacheMessage` class represents a serializable object used to transfer messages between nodes. Its key functionalities include:

- **Serialization:** The class implements the `Serializable` interface, enabling `CacheMessage` objects to be serialized. The `serialVersionUID = 1L` field is a unique identifier that ensures compatibility of serialized versions when deserialized. This is essential to prevent incompatibility issues arising from future modifications to the class fields.
- **Attributes:** The class defines five fields representing the message data:
  - `action`: A string specifying the action to be performed, such as "PUT", "INVALIDATE", or "FETCH_REQUEST". This indicates the cache operation to execute.
  - `nodeId`: A string identifying the node in the distributed system that generated the message. It is useful for tracking the message's origin.
  - `key`: A string representing the key of the value to insert or update.

- `value`: A string containing the value associated with the key in the cache.
- `timestamp`: A long value representing the message creation time, useful for ordering events temporally.

- **Getter and Setter Methods:** The class provides getter and setter methods for each field, enabling controlled access and modification of private attributes.

4.5 CacheManager

The `CacheManager` class leverages the Caffeine library to implement cache management, allowing efficient storage, retrieval, and removal of objects. Key features are:

- **Cache Initialization:** In the class constructor, the cache is initialized using Caffeine's builder pattern:

  ```
  this.cache = Caffeine.newBuilder()
      .expireAfterWrite(10, TimeUnit.MINUTES)
      .maximumSize(100)
      .build();
  ```

  Explanation of the code:

  - `expireAfterWrite(10, TimeUnit.MINUTES)`: Sets an expiration policy. Objects are automatically removed 10 minutes after their last write (insert or update).
  - `maximumSize(100)`: Limits the cache to a maximum of 100 objects. When this limit is reached, the least-used objects are evicted.
  - `build()`: Creates a cache instance configured with the specified policies.

- **Cache Management Methods:** The class provides several methods to manage the cache:

  - `put(String key, Object value)`: Inserts an object into the cache, associating it with the specified key. Updates the value if the key already exists.

    ```
    public void put(String key, Object value) {
        cache.put(key, value);
    }
    ```

  - `get(String key)`: Retrieves an object from the cache by its key. It uses `getIfPresent()`, which returns the object if it exists in the cache or `null` otherwise.

```
public Object get(String key) {
    return cache.getIfPresent(key);
}
```

– `invalidate(String key)`: Removes a specific object from the cache using its key.

```
public void invalidate(String key) {
    cache.invalidate(key);
}
```

– `invalidateAll()`: Clears all objects from the cache.

```
public void invalidateAll() {
    cache.invalidateAll();
}
```

4.6 PulsarClientManager

In a distributed system with multiple cache nodes, it is crucial to share data between nodes to reduce database load. If one node has the required data in its cache, other nodes do not need to store or fetch it from the database. For this purpose, **Pulsar**, a real-time *publish/subscribe* messaging system, is used. Nodes connected to Pulsar can receive updates about new data and respond to requests from other nodes.

The Publish-Subscribe Paradigm: In a Pub/Sub system, *publishers* (nodes) send messages to a *topic* (channel) without knowing the recipients. *Subscribers* receive messages from topics they are subscribed to. Pulsar acts as a broker, ensuring decoupling between publishers and subscribers while delivering messages efficiently.

Description

The `PulsarClientManager` class represents a Pulsar client that enables a node to send and receive messages within a distributed system. It provides an encapsulated interface for producer and consumer operations. Key functionalities include:

- **Client Creation:** Initializes a client and connects it to the Pulsar service URL.
- **Message Producer:** Sends messages to a shared topic.
- **Message Consumer:** Receives messages from the topic with a unique subscription for each node.
- **Resource Cleanup:** Provides methods to close the client and release associated resources.

4.7 CacheInvalidationException

The `CacheInvalidationException` class extends the `PulsarClientException` and is designed to handle exceptions specific to cache invalidation.

Main Functionality

The constructor of the `CacheInvalidationException` class accepts two parameters:

- **message:** A descriptive string of the error.
- **cause:** A `Throwable` object representing the original cause of the error.

This allows for custom error messages and the inclusion of the original exception, facilitating improved error handling.

Usage and Context

This exception is raised when an error occurs during cache invalidation, such as connection issues with Pulsar or other messaging-related errors. By extending `PulsarClientException`, it integrates seamlessly into the Pulsar client context and is easily handled by interacting components.

Chapter 5

Automated Tests

The primary goal of these tests is to ensure the application functions correctly in a client-server environment, verifying that all components interact seamlessly.

During testing, the following aspects were validated:

- Data transmission and reception between client and server operated reliably and consistently.
- Client requests were correctly processed by the server.
- Responses were delivered within acceptable time limits.
- Session and authentication management was robust, ensuring uninterrupted user access to their data.

5.1 Automated Tests

To guarantee stability and proper operation, automated tests were implemented to monitor various features, including node creation and deletion within the system. These tests simulated realistic scenarios to verify:

- Correct registration and association of all necessary information during node creation.
- Proper handling of node deletion without residual data or errors.
- Reliable client-server interactions throughout the process.

This approach facilitated rapid identification and resolution of issues, enhancing overall system quality and reducing development time.

5.2 Example of Test:

The `testInvalidateAck` test method verifies that invalidation of a key (`key`) on one node (`node2`) is propagated and acknowledged by all other nodes (`node1` and `node3`). Specifically, it checks that the key, after being invalidated on `node2`, is removed from the caches of the other nodes.

Test Behavior

The test simulates the following scenario:

1. A value is inserted into the cache of `node1` for a specific key.
2. A brief wait ensures the data propagates to `node2` and `node3`.
3. The key is invalidated on `node2`.
4. Another short wait allows the invalidation acknowledgment (ACK) to be received.
5. The test confirms that the key has been invalidated on `node1` and `node3`.

Code Snippet

```
@Test
public void testInvalidateAck() {
    // Insert value into node1's cache
    cacheManagerNode1.put("key", "value");

    // Wait for propagation
    Thread.sleep(100);

    // Invalidate the key on node2
    cacheManagerNode2.invalidate("key");

    // Wait for ACK reception
    Thread.sleep(100);

    // Verify that the key was invalidated on all nodes
    assertNull(cacheManagerNode1.get("key"));
    assertNull(cacheManagerNode3.get("key"));
}
```

Explanation of the Code

- **Initial Insertion**: A value (`testValue`) for a key (`testKey`) is inserted into the cache of `node1`.

- **Propagation Wait**: A delay (`Thread.sleep(1000)`) ensures the value propagates to `node2` and `node3`.
- **Key Invalidation**: The key is invalidated on `node2` using the `invalidate` method.
- **Wait for ACK**: Another delay (`Thread.sleep(2000)`) ensures all nodes acknowledge the invalidation.
- **Final Verification**: The `assertNull` method confirms the key has been removed from the caches of `node1` and `node3`.

Test Considerations

This test underscores the importance of synchronizing system nodes and using acknowledgment mechanisms (ACKs) to confirm the propagation of changes across the network. The final verification ensures that data is invalidated throughout the cache network, preventing outdated access.

Expected Results

The test is considered successful if:

- After invalidation on `node2`, the key is no longer accessible from `node1` and `node3`.
- No exceptions occur during the test execution.

Chapter 6

Deployment

This section provides an overview of how to deploy the application and perform basic testing using tools like Postman after the services are started via Docker.

6.1 How to deploy

Prerequisites

To deploy the application, ensure the following are installed on your machine:

- Docker
- Docker Compose
- Maven

Deployment Steps

1. **Clone the repository:**

   ```
   git clone

   https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/


   ds-project-branco-ay2324.git


   cd  ds-project-branco-ay2324
   ```

2. **Build the project:** Use Maven to build the application:

```
mvn clean install -DskipTests
```

3. **Build the Docker image:**

```
docker build -t pubsubcache .
```

4. **Start the services:** Use Docker Compose to launch the necessary services, including the application and Apache Pulsar:

```
docker-compose up
```

Components and Ports

After starting the services, the following components will be running:

- **The application:** Runs on dynamic ports, with each instance exposing its APIs.
- **Apache Pulsar:** Handles messaging and runs on fixed ports, such as 6650 (Pulsar service) and 8080 (Pulsar web interface).

To check the running containers and their port mappings, execute:

```
docker ps
```

This command displays the active containers and the host ports mapped to them.

Testing the APIs

Once the application is running, you can test its APIs using Postman or curl. The APIs provide functionality to manage cache entries and test distributed cache operations.

Base URL: `http://localhost:<dynamic_port>` *Note:* The dynamic port can be retrieved from the `docker ps` output under the "PORTS" column.

- **GET /cache/get/{key}** Retrieves a value associated with the given key from the local cache. **Example Request:**

```
curl -X GET http://localhost:<dynamic_port>/api/cache/get/sampleKey
```

- **GET /cache/fetch/{key}** Fetches a value from the distributed cache by querying other nodes if necessary. **Example Request:**

```
curl -X GET http://localhost:<dynamic_port>/api/cache/fetch/sampleKey
```

- **POST /cache/load** Loads a key-value pair into the local cache. **Example Request:**

```
curl -X POST "http://localhost:<dynamic_port>/api/cache/load?key=

sampleKey&value=sampleValue"
```

- **POST /cache/put** Inserts or updates a key-value pair in the cache and synchronizes it across the cluster. **Example Request:**

```
curl -X POST "http://localhost:<dynamic_port>/api/cache/put?key=

sampleKey&value=sampleValue"
```

- **DELETE /cache/invalidate/{key}** Invalidates a key in the cache, removing it from the system. **Example Request:**

```
curl -X DELETE http://localhost:<dynamic_port>/api/cache/invalidate/sampleKey
```

Example Test Scenarios

Use the following scenarios to test the application's functionality:

1. **Test LOAD Operation:** Use the `POST /cache/load` endpoint to load a key-value pair into the cache. Verify the value using the `GET /cache/get/{key}` endpoint.
2. **Test FETCH Operation:** Load a key-value pair into one node using `POST /cache/load`, then verify it can be fetched from another node using `GET /cache/fetch/{key}`.
3. **Test PUT Operation:** Use the `POST /cache/put` endpoint to update a key-value pair. Confirm the change using the `GET /cache/get/{key}` endpoint.
4. **Test INVALIDATE Operation:** Load a key-value pair, then invalidate it using `DELETE /cache/invalidate/{key}`. Verify the key no longer exists using the `GET /cache/get/{key}` endpoint.

These tests can be performed manually using Postman by constructing requests and verifying responses. The dynamic port allocation ensures no conflicts between nodes, providing a robust environment for testing distributed caching behaviors.

Chapter 7

Conclusions

This project has served as a significant opportunity for growth, allowing me to deepen the skills acquired during the course while addressing and solving complex challenges related to real-time communication and distributed data management. The implementation of data invalidation and synchronization functionalities using Apache Pulsar presented a valuable challenge, enabling me to translate theoretical knowledge into practical applications, strengthen my technical expertise, and refine my approach to designing distributed systems.

One of the primary goals of the project was to develop a modular and scalable solution that could seamlessly accommodate future extensions. To achieve this, the design emphasized the integration of a Publish\Subscribe architecture. This approach enhanced the system's flexibility and resilience, enabling nodes to operate asynchronously and independently while providing strong support for fault tolerance.

A particularly advantageous feature of Apache Pulsar is its ability to configure clients to interact with multiple brokers, thereby increasing the system's overall resilience. This setup ensures that even if one broker becomes unavailable, the nodes can continue to communicate and synchronize data through the remaining brokers, significantly mitigating the risk of service disruptions.

Despite the constraints of limited time, I concentrated on delivering the core functionalities essential for the project's objectives. Furthermore, I adopted a modular development approach to ensure that the solution could be easily extended in the future, allowing for the integration of new features without compromising the system's architecture.

7.1 Future Improvements

Outlined below are potential areas for future enhancements:

- **Enhanced Failure Management:** An area for improvement is the handling of node failures and reconnection strategies. Currently, the system

employs a limited number of reconnection attempts in case of errors. Future work could involve implementing more robust failure management techniques, such as dynamic reconnection time adjustments, advanced failover mechanisms, and automated node resilience strategies.

- **Improved Scalability:** While the system performs well under moderate loads, scalability enhancements are necessary to support a larger number of nodes and operations. Potential improvements could include distributing workloads across multiple Pulsar brokers, optimizing data propagation, and employing advanced caching strategies to enhance performance in high-concurrency scenarios.
- **Security Enhancements:** At present, the system lacks advanced security measures to protect data and communication between nodes. Future developments could include integrating encryption mechanisms to secure data transmission, implementing robust authentication for accessing the nodes and Pulsar system, and introducing access controls to ensure that only authorized users can interact with the cache system.

7.2 Conclusion

In summary, this project proved to be an immensely rewarding educational experience. Applying advanced techniques for distributed system design provided a hands-on opportunity to explore and understand the intricacies of building resilient and scalable solutions. This journey has significantly enhanced my approach to designing complex systems, fostering a perspective that prioritizes both system robustness and design flexibility.

Bibliography

Apache Pulsar, *A distributed messaging and streaming platform.* `https://pulsar.apache.org/`

Caffeine Cache Repository, *High performance caching library for Java.* `https://github.com/ben-manes/caffeine`