UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Master of Science in Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

Real-Time Stock Market Data

Students:

Giuseppe Soriano

TEMFACK Derick

Examiner: Prof. Alessio Bechini
University of Pisa, Italy

Academic Year 2023-2024

Contents

1. Introduction

This project is designed to keep track of the real-time performance of certain stock market tickers. These tickers comprise BITCOIN in USD, ETHEREUM in USD, and the stocks of major companies like Amazon, Apple, Tesla, and Microsoft. By constantly monitoring the behaviour of these tickers, the project aims to provide valuable insights into the movements of the stock market and help investors make informed decisions.

A stock, also known as a **share**, is a unit of ownership in a company. When you buy a stock, you essentially buy a piece of the company's assets and profits. The market determines the price of a stock, and it reflects the collective judgment of investors about the company's value.

A symbol, also known as a **ticker**, is a short, alphanumeric code that uniquely identifies a particular stock. Stock symbols are standardized and are typically used to represent the company's name or ticker symbol. For example, the stock symbol for Apple Inc. is AAPL, and the stock symbol for Microsoft Corporation is MSFT.

Here is some helpful information for accessing different artefacts and testing the application.

To access the web application, visit the following URL in a web browser:

http://10.2.1.72:8084

To query stock information starting from a specific date, such as Tesla's stock performance from a given date, make a GET request to:

http://10.2.1.73:8080/database/stock-api?ticker=TSLA&date=1

This URL fetches stock data for the specified ticker (in this case, TSLA) from the Database Manager, querying data recorded from the specified date onward. Ensure that the date parameter is provided in epoch seconds format.

2. Functional Requirements

2.1 Functional Requirements

- As a user, I want to see real-time stock values for BTC/USD, ETH/USD, TSLA, AMZN, MSFT, and AAPL at the top of the app to stay updated on the market.
- As a user, I want to click on a specific ticker in the top bar and view all the historical data on a stock chart.

2.2 Implementation Functional Requirements

- Real-Time Market Data Collection and Management with Java Server
  - **Objective:** To collect and manage real-time stock market data streams from various data sources.
  - **Description:** Utilize a Java server with EJB Singleton hosted on GlassFish to gather market data from different services, each handling specific data for various companies. This server acts as the primary data collection point before forwarding data to the middleware for further processing.
- Real-Time Data Processing and Distribution through Erlang Cowboy Bridge
  - **Objective:** To process and distribute real-time market data to client applications.
  - **Description:** The Cowboy Bridge middleware processes the incoming data from the Java server and routes it to the appropriate consumers, including the web application and the Database Manager. This setup ensures efficient data processing and timely distribution to end users.
- Reliable web-socket endpoint
  - **Objective:** To ensure reliable and concurrent management of market data updates.
  - **Description:** The system is designed to maintain high reliability and performance, even under substantial loads, ensuring that data updates are timely and consistent across all client applications.
- Data Interaction and Management through Web Application
  - **Objective:** To provide users with real-time access and interaction with the market data.
  - **Description:** The web application, developed with servlets and JSP and hosted on Tomcat, communicates with the Cowboy Bridge via WebSocket for real-time data updates. It also interacts with the Database Manager using HTTP GET requests to fetch and display historical data insights.

- Database Management for Data Persistence Using Redis
  - **Objective:** To manage the storage and retrieval of historical market data using Redis.
  - **Description:** The Database Manager, implemented with WebServlets on GlassFish, handles both the storage of data in Redis and the servicing of data requests from the web application. Redis is utilized for its performance in handling large volumes of data with minimal latency, ensuring that both historical and real-time data are efficiently managed and accessible.

3. Architecture and Implementation

3.1 Distributed Architecture

The architecture of our system is structured as a network of interconnected components, each designed to manage specific aspects of communication and data processing within a distributed environment. This configuration ensures operational efficiency and scalability, essential for meeting the modern demands of an advanced information system. The following section provides a detailed overview of each node, highlighting its functions, the technologies used, and its interactions with other system components. From a Java node responsible for data production, through an Erlang node serving as central middleware, to a dynamic web-app and a responsive database manager, each component is tailored to support the overall functionality. The interactions between these nodes facilitate a seamless and timely flow of data and establish a robust framework for ongoing support and future expansion. In the Figure 3.1, the schema that describes the overall architecture is shown. More detailed analysis of each component is presented next, illustrating their critical roles in the overall system architecture.

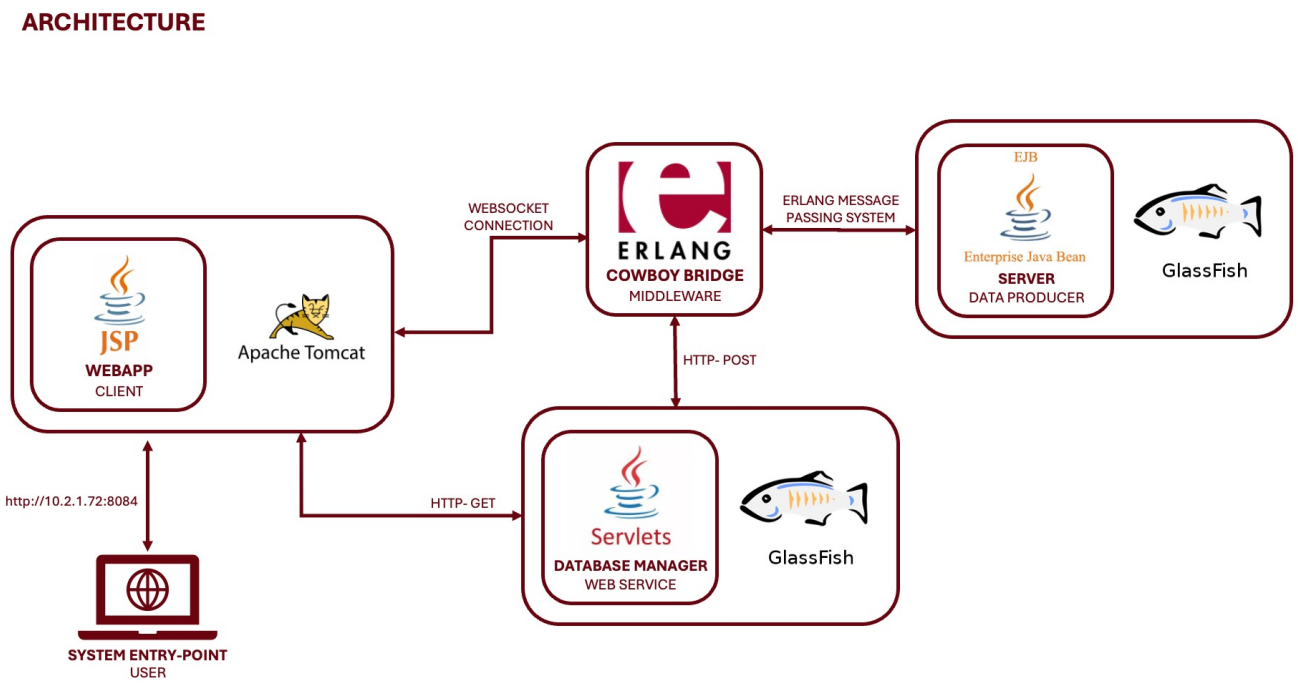

Figure 3.1: Architecture schema

3.2 Server

Our server is responsible for producing data for BTC/USD, ETH/USD, TSLA, AMZN, MSFT, and AAPL in specific intervals. The server is implemented as a singleton EJB with methods to generate the required data. We make use of the TimerService to execute scheduled operations as per the cron pattern. This approach of generating fake data is used because access to real-time data is not free, and it allows us to test our distributed system. The data produced by the server is sent to our middleware Erlang node for further processing.

3.3 Database Manager

The Database Manager is an essential microservice within our architecture, implemented as a Jakarta servlet and deployed on Glassfish. This component manages the storage of data within a Redis database and provides the web app with historical data access. Data flows into the Database Manager from the Erlang node via a POST request, facilitating efficient data storage. In addition, the web app retrieves historical data through a GET request to display specific ticker information for end users.

The Database Manager outputs data as a JSON array where each object represents one trading day's data for a specific stock ticker. The data objects include the following fields: `day` (the date of the trading data), `open value` (the price at the start of the trading day), `low value` (the lowest price during the trading day), `high value` (the highest price during the trading day), and `close value` (the price at the end of the trading day). This structured data format allows the web app to present detailed historical trading data, providing end users with insights into market trends and the performance of stocks over specified periods.

3.4 Web App

The WebApp serves as the primary gateway to our distributed architecture. It is a pivotal component that enables users to interact with our system. The WebApp is built on Jakarta Servlet technology and comprises two servlets and two JSP pages. The home page of the application is the default page and doesn't pass through any servlet. The **StockServlet** is responsible for implementing the GET request and displaying the historical data of a specific ticker. Finally, the **StockEndpoint** is a REST Endpoint that the JavaScript file uses to initialize the stock chart with all the historical data. The servlet retrieves the requested data from the Database Manage micro-service and sends it to the JavaScript file. Additionally, the WebApp connects to the Erlang WebSocket endpoint to display real-time data on all pages' top bars. In summary, the WebApp is the primary means by which users interact with our application.

3.5 Cowboy Bridge

The *cowboy bridge* acts as a central communication hub within our distributed architecture, integrating the Java backend, the web application, and the Database Manager. Utilizing Erlang's efficient concurrency features and the Cowboy framework, it ensures seamless data handling and distribution across the system.

The *cowboy_bridge* employs two essential GenServers, *stock_storage* and *internal_comm_manager*, to manage and orchestrate the system's communication processes:

- **stock_storage**: This GenServer is tasked with managing and storing stock data that flows into the system, maintaining a quick-access state for current stock information. It enables rapid data updates and retrieval, crucial for the responsiveness and reliability of the system.
- **internal_comm_manager**: Serving as the communication backbone within the *cowboy_bridge*, this GenServer coordinates the flow of messages across the system, ensuring data consistency and managing synchronization between different system components.

Operational Dynamics

The operational dynamics within the *cowboy_bridge* are structured as follows:

- **Data Reception and Distribution**: The *message_receiver* module initially receives data inputs, which are then forwarded to the *internal_comm_manager*. This GenServer plays a pivotal role in routing these incoming messages to the appropriate destinations within the system. Specifically, it directs the stock data to the *stock_storage* module, where the data is updated and maintained in its current state.
- **Data Retrieval and Web Communication**: The *websocket_handler* module actively queries the *stock_storage* every second to retrieve the latest stock values. Once retrieved, these values are sent directly to the client through an established WebSocket connection. This continuous polling and data transmission ensure that the client-facing web application displays real-time, up-to-date stock information.

This flow illustrates a clear division of responsibilities within the *cowboy_bridge*, where each component is optimized for specific tasks - data reception, processing, storage, and final delivery to the client - ensuring efficient management of data throughout the system.

4. Communication and Synchronization Issues

4.1 Communication

We rely on Erlang as a central component for communication between our various microservices, leveraging its built-in concurrency handling and fault tolerance features. Our database producer, which is an EJB, sends data to the Erlang Node through JInterface. Once the message reaches the Erlang node, it is distributed across different microservices. Specifically, when the Erlang node receives a message, it forwards that message to the web app using a WebSocket endpoint. At the same time, it sends the data to the Database Manager micro-service to store in the database. The WebApp communicates with the Erlang Node via WebSocket to display real-time stock data. It also communicates with the Database Manager to obtain historical data.

A key aspect of our system's architecture is its inherent capacity for scalability, particularly in terms of handling an increasing volume of data producers. The use of Erlang as middleware not only simplifies the integration of additional Java producers but also ensures that scaling the number of data nodes does not compromise the system's performance or reliability. This is achieved through Erlang's efficient message-passing capabilities, which facilitate seamless data flow between an increasing number of producers and the central Erlang node.

As more Java producers are added to the system, they can easily send their data to the Erlang node, which then acts as a dispatcher, routing this data to the appropriate microservices, be it the WebApp for real-time display or the Database Manager for storage. This model supports dynamic scaling, allowing our system to accommodate growth without requiring significant architectural changes or additional complexity in data routing.

The design of our Erlang-based system, particularly the `internal_comm_manager`, is crafted to ensure robust internal communication. This component is crucial for orchestrating the interaction between various modules within the system. By managing the data flow and ensuring that messages are directed correctly, the `internal_comm_manager` enhances the system's responsiveness and coherence.

Looking forward, this structure provides a solid foundation for potential system upgrades or expansions. Should the need arise to deploy Erlang modules on separate machines or to introduce additional functionalities, the existing communication framework, centered around the `internal_comm_manager`, will allow for straightforward integration and interaction between distributed components. This flexibility is pivotal in maintaining system efficiency and reliability as it evolves.

4.2 Synchronization

In distributed systems dealing with real-time data, such as stock price updates, maintaining data consistency and ensuring timely delivery are paramount. The Erlang modules `stock_storage` and `websocket_handler` play critical roles in achieving these objectives by managing the synchronization of data updates and their dissemination to clients. Here, we explore how these modules handle potential synchronization challenges through tailored strategies, enhancing both system reliability and user experience.

The `stock_storage` module serves as the nerve center for handling incoming stock data. It employs a sophisticated mechanism to ensure that only the most recent data updates modify the system state. This is crucial because stock data is highly time-sensitive, and any errors in the order or timeliness of updates can lead to incorrect information being relayed to users. When new data arrives, it is processed sequentially due to Erlang's inherent design, which naturally queues messages as they come in. The `handle_info` function within `stock_storage` performs a critical check, comparing the timestamp of incoming data against what is currently stored. If the incoming data is newer, it updates the state; if not, it is disregarded. This approach not only prevents older data from overwriting newer data but also eliminates the need for locks, making the system more efficient and less prone to deadlocks or race conditions.

Parallel to the data storage and validation process, `websocket_handler` ensures that all clients receive the latest stock data at consistent intervals every second. This consistency is vital for client applications that rely on up-to-the-minute data to make financial decisions.

The module leverages Erlang's built-in timing functions to send out data updates at fixed intervals, independent of when data updates are received. This decoupling of data reception from data delivery allows the system to maintain a steady and predictable flow of data to clients, ensuring a smooth user experience even under varying network conditions or data arrival rates.

Moreover, to handle potential issues such as network congestion or client-side processing limitations, `websocket_handler` employs a buffering strategy. By aggregating data updates over the interval of one second, the system can minimize the number of messages sent, reducing the load on the network and the processing demand on clients. Just before dispatch, it performs a final check to ensure that the data being sent is the most current, thus minimizing the chances of delivering stale information.

The Database Manager within our system architecture plays a pivotal role, but it operates under different conditions compared to the real-time data handling of `stock_storage` and `websocket_handler`. Unlike these components, the Database Manager is largely unaffected by the synchronization issues that impact the real-time delivery of stock data.

Data is forwarded to the Database Manager as soon as it is received from the Java producer, along with the associated timestamp. This immediate forwarding ensures that the Database Manager stores every piece of data exactly as it arrives, regardless of whether it passes the timestamp validation or is stored in the `stock_storage` buffer.

The inclusion of timestamps with each data point allows the Database Manager to record the exact sequence of events as they are received, without needing to reorder them. This feature is

crucial because it means that the Database Manager's functionality remains unaffected by any discrepancies in the order of data arrival. Its primary task is to ensure data is logged accurately with the correct temporal data, maintaining the integrity of historical records.

The operation of the Database Manager is thus distinctly separated from the real-time data processing and synchronization challenges that other components handle. It acts as a straightforward repository that logs incoming data, with each entry marked by its reception time. This process isolates the Database Manager from the complexities faced by the real-time operational components, such as ensuring data freshness and order, which are managed upstream.