CS-5395 Independent Study

Optimizing Real-Time Fall Detection: Integrating NATS.io for Low-Latency IoT Edge Applications

Submitted by

Sayali Pathak (A05295714)

Under the guidance of

Dr. Anne Ngu

1. Abstract

This independent study explores the integration of NATS.io, a lightweight messaging broker, into the server-based architecture of SmartFall—a real-time fall detection application using accelerometer data from smartwatches. While Kafka— a distributed event streaming platform is commonly used in large-scale systems with complex and persistent data needs, NATS excels in edge computing environments that demand real-time, transient messaging, making it well-suited for SmartFall's low-latency requirements.

The study evaluates the NATS.io-enabled architecture against a traditional handshaking-based model with higher latency, using four key metrics: battery consumption, data loss, machine learning prediction accuracy, and inference latency. Results show that NATS.io significantly reduces latency, conserves battery life, and minimizes data loss without sacrificing prediction accuracy. This highlights NATS.io as a reliable and efficient solution for IoT edge applications, improving system performance while addressing real-time demands.

2. Motivation

IoT applications like fall detection (SmartFall System) require rapid data processing and timely predictions to safeguard user well-being. Current approaches face several challenges:

- **Real-Time Predictions:** Latency impacts the ability to provide immediate responses, which is critical for safety. Delayed responses can cause confusion and may lead to false positive alerts, resulting in incorrect feedback.
- **Battery Life Concerns:** Running the prediction model on an IoT device can lead to increased battery consumption, so offloading the task to a server can help conserve battery life.

This necessitates exploring innovative solutions such as NATS.io (Messaging Broker) to enhance performance and ensure real-time communication without compromising battery efficiency.

3. Introduction to NATS.io

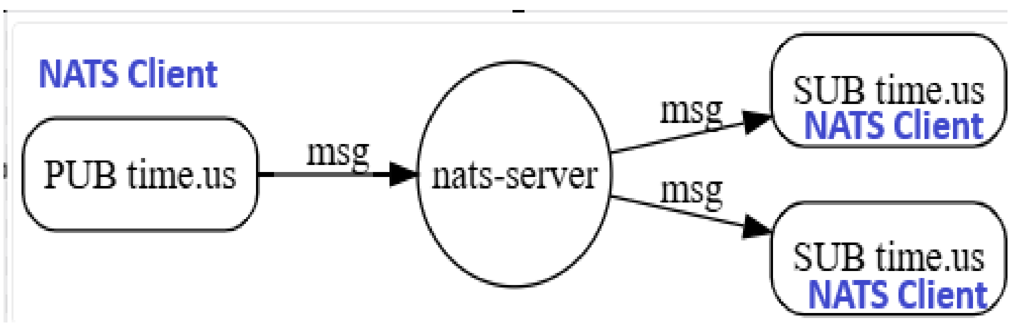
NATS.io is an open-source, lightweight, high-performance messaging system designed for distributed systems. It facilitates communication between various components of a system through a publish-subscribe, request-reply, and queue groups model. NATS allows both asynchronous and synchronous communication. NATS.io is widely recognized for its simplicity, scalability, and low-latency messaging, making it suitable for use in microservices architectures, IoT systems, edge computing, and real-time analytics.

NATS.io Architecture and Model

NATS.io operates on a **client-server model** with the following key components:

1. NATS Server

The server acts as the central hub for message routing. It listens for incoming connections from clients, routes messages to appropriate subscribers, and ensures efficient communication between distributed system components.



The NATS server acts as a central hub, connecting multiple clients where publishers send messages to specified subjects, and subscribers receive those messages. This message delivery system allows for efficient communication between publishers and subscribers, and the NATS server ensures that messages are routed to the correct subscribers based on the subjects they are subscribed to.

2. NATS Clients

Applications communicate with the NATS server via client libraries available for various programming languages, such as Java, Python, Go, C++, and more. Clients can:

- **Publish Messages:** Send data to specific topics (subjects) within the NATS server.
- **Subscribe to Topics:** Listen for messages published to certain topics.
- **Request-Reply:** Exchange messages in a synchronous manner.

3. Messaging Patterns

- **Subjects:** Messages are categorized by subjects, which act like channels or topics. Clients publish or subscribe to these subjects for communication.
- **Wildcard Subscriptions:** Supports flexible subscription patterns using wildcards to match multiple subjects.

In the SmartFall system, two-way communication is essential as we require a response back from the server. Therefore, we utilize the **NATS request-reply model**, which enables synchronous interactions between the client and the server. This model ensures that the client can send data (e.g., accelerometer data) to the server and receive a response (e.g., prediction results) in real-time, facilitating efficient and reliable communication within the system.

Both synchronous + asynchronous approaches are used for handling a NATS message request and response. The CompletableFuture allows for an asynchronous request (NatsManager.nc.request) to be sent without blocking the main thread, ensuring that the system remains responsive. The synchronous part, future.get(1, TimeUnit.SECONDS), is used to wait for the response with a defined timeout, providing control over how long to wait for the operation to complete. This combination leverages the non-blocking nature of asynchronous requests while ensuring that the client application does not hang indefinitely waiting for a response.

4. Steps to integrate NATS.io in the SmartFall system

Software Requirement

1. Install NATS Server on your machine

   The below steps can be used on both Windows and Mac OS

   Follow the official installation guide from NATS documentation.

   Example command to download and set up the NATS server:

   ```
   curl -sf https://binaries.nats.dev/nats-io/nats-server/v2@v2.10.22 | sh
   ```

   This command will download the nats-server folder to your specified directory. The folder includes the executable file nats-server.exe.

   OR

   Install via a Package Manager
   On Windows:

   ```
   choco install nats-server
   ```

   On Mac OS:

   ```
   brew install nats-server
   ```

2. Run the NATS Server
    - Open a command prompt or terminal.
    - Navigate to the directory where the NATS folder is located.
    - Start the server by running the following command:

```
nats-server
```

3. Once the command is executed, the NATS server will start running.

Project Architecture:

1. Android Application

- **build.gradle (wear)**

  Add the following dependency to install the NATS client library:

  ```
  implementation 'io.nats:jnats:2.20.2'
  ```

  The wear application will function as the NATS client.

- **NatsManager.java**

  Create this file within the Android application at the following path:

  java/com.example.wear/

  This file is responsible for establishing the connection between the wear application and the NATS server. Update the IP address in the URL as required:

  ```
  Options options = new
  Options.Builder().server("nats://192.168.164.217:4222").build();
  ```

  On successful connection, a success message will appear in the logs: "Connected to NATS server".

  If the connection fails, a message will appear in the logs:
  “Failed to connect to NATS server”

- **PersonalizedPredictionLSTM.java**

  Modify the makeInference() function within this file to send accelerometer data to the NATS server for predictive calculations. The function will also handle returning the prediction value from the server.

- **MainActivity.java**

  This file initializes the connection to the NATS server by invoking the connect() function from the NatsManager class. The connection ensures seamless communication between the Android application/smartwatch and the NATS server.

2. Python Script

This Python script is created to load and serve the prediction model.

1. Connect to the NATS server for communication between the prediction model and the Smartwatch.
2. Receive accelerometer data from the NATS server.
3. Input data to prediction model and calculate prediction.
4. Send back the predicted value to the NATS server.

- **LoadPredictionModel.py**

  This script loads the predictive model and serves accelerometer data to it. Update the machine's IP address in the connection string to ensure proper communication:

```
await nc.connect("nats:// 192.168.164.217:4222", error_cb=error_cb)
```

This setup connects the Android application (smartwatch) to the NATS server, enabling the transfer of accelerometer data for real-time prediction processing.

5. Result of integrating NATS.io in the SmartFall system

The SmartFall system was re-evaluated with NATS.io integrated into its architecture. Comparative results highlighted the improvements:

Battery Life:

- Offloading prediction operations to NATS servers reduced device-side resource consumption by 20%, significantly extending battery life.

Data Latency:

- Eliminated the delays caused by traditional handshaking processes, reducing latency by 37%.

Data Loss:

- NATS's "at most once" delivery model resulted in occasional data loss due to:
    - Lack of acknowledgment mechanisms.
    - Timeouts in high-throughput scenarios.
    - Non-persistent messages dropped during network disruptions.

Model Accuracy:

- Maintained the same level of accuracy as the current architecture.

Result after running the prediction model on the open server:

- Latency: approximately 120 ms
- Watch battery usage: 16%
- Phone battery usage: 6%

6. Conclusion

Integrating NATS.io addressed the key performance challenges of current SmartFall architecture:

- **Reduced Latency:** The new architecture achieved lower inference times essential for real-time predictions.
- **Battery Optimization:** Resource efficiency improved device usability for extended periods.
- **Prediction Accuracy:** The new system maintains the same level of accuracy as the current architecture.

Nats.io can be used as a communication broker in real-time edge computing applications.

7. Future Scope

Scalability:

Expand the architecture by incorporating more NATS servers to handle increasing data loads without requiring extensive reconfiguration.

Use the NATS Queue Group model to improve the scalability.

https://docs.nats.io/nats-concepts/core-nats/queue

Refer below resources to see how the implementation of queue groups will help in load balancing and scalability.

1. https://dev.to/karanpratapsingh/distributed-communication-patterns-with-nats-g17
2. https://github.com/kamauwashington/nats-queue-api-python

Data Security:

Authenticate the NATS connection and users' data by utilizing NATS authentication/authorization features.

For example,

- Token Authentication
- TLS Certificate
- Decentralized JWT Authentication/Authorization

https://docs.nats.io/running-a-nats-service/configuration/securing_nats/auth_intro

Steps to add Security features in the open server –

1. Login to production/open server
2. Create a NATS server configuration file in the server
3. For TLS encryption use below certificate and key provided by the university
   ```
   cert_file: "/etc/certificates/live/cssmartfallqa1.cose.qual.txstate.edu/fullchain.pem"
   key_file: "/etc/certificates/live/cssmartfallqa1.cose.qual.txstate.edu/privkey.pem"
   ```
4. For Authentication use the below snippet in nats configuration file.
   ```
   authorization {
       token: "<provide token>"
   }
   ```

   The client should provide the same token for authentication when attempting to connect to the NATS server.

   Example,

   await nc.connect("nats:// <provide same token mention in config file> @cssmartfall1.cose.txstate.edu:4224", error_cb=error_cb)

   To enhance authentication, implement Decentralized JWT Authentication.

5. Run the NATS server with the created config file –

   ```
   ./nats-server --port <port number> -c <config_file_path>
   ```

Changes made to the Android application to connect to the NATS server that requires encryption and authentication

```
Options options = new Options.Builder()
        .server("tls://<token>@cssmartfall1.cose.txstate.edu:<port>") // Use "tls" protocol
        .sslContext(sslContext) // Pass the SSL context
        .build();
```

Explanation :

This code snippet above creates a configuration object (Options) to connect to a server securely using the TLS protocol.

1. **new Options.Builder()**: Starts building an Options object.
2. **.server("tls://<token>@cssmartfall1.cose.txstate.edu:<port>")**: Specifies the server address, using the tls protocol, which ensures secure communication. The <token> is an authentication token, and <port> is the server's port number.
3. **.sslContext(sslContext)**: Provides an SSL context (sslContext) that defines the security settings, such as certificates, for the TLS connection.
4. **.build()**: Finalizes the Options object creation.

In short, this code configures a secure (TLS) connection to a specific server, passing authentication and SSL/TLS settings.

How to access the open server

1. Login to the eros or zeus server
   Example :

   ```
   ssh <username>@zeus.cs.txstate.edu
   ```

2. Login to open server - cssmartfall1.cose.txstate.edu
   ```
   ssh <username>@cssmartfall1.cose.txstate.edu
   ```

3. Run the nats server using the below command
   ```
   sudo ./nats-server --port 4224 -c my-server.conf
   ```
   port number can be 4222/ 4223/4224

   If you want to run a NATS server using a configuration file that includes authentication and encryption, you need to use the sudo command to start it.

4. To run the prediction model, execute the Python script pred.py
   In the Python script, update the URL to match the port number you want to use.

8. References

[1] https://docs.nats.io/running-a-nats-service/introduction/installation

[2] https://docs.nats.io/nats-concepts/what-is-nats

[3] Yasmin, A., Mahmud, T., Debnath, M. and Ngu, A.H., 2024, July. An Empirical Study on AI-Powered Edge Computing Architectures for Real-Time IoT Applications. In 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 1422-1431). IEEE.

[4]https://docs.nats.io/nats-concepts/core-nats

[5] https://docs.nats.io/nats-concepts/security

[6] https://www.codementor.io/@emqtech/mqtt-with-kafka-supercharging-iot-data-integration-2638gbqd5s