Analysis of a Three-Component Distributed Voting System: Performance, Latency, Reliability, and Transparency Dimensions

Abstract:

In this report, we present an analysis of a distributed voting system built using modern cloud-based components and architectural patterns. The system is divided into three components, each responsible for different tasks: data analysis, real-time voting, and vote validation. The system utilizes Cloud Scheduler, Cloud Functions, Firestore, DataProc, Cloud Run, and Pub/Sub as its primary components, following a client-server and event-driven architecture. We analyze the system's performance, latency, reliability, and transparency dimensions and compare them to real-world distributed systems like Twitter.

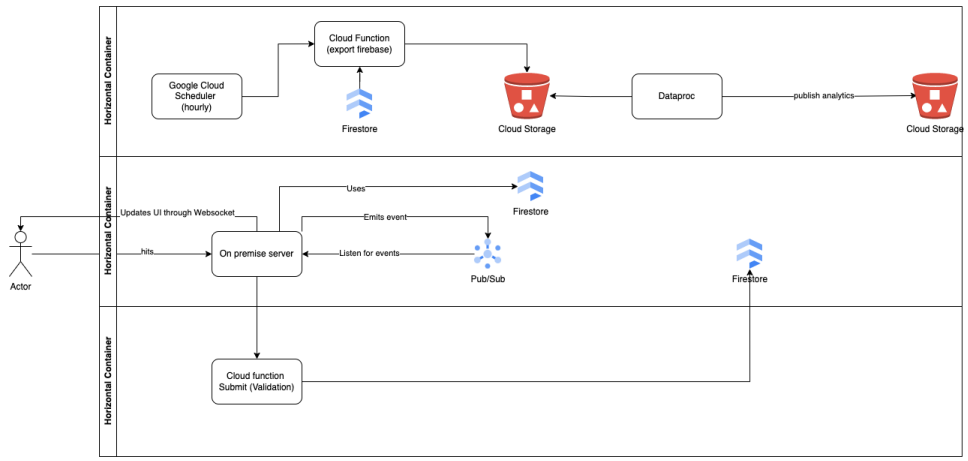
Introduction:

Distributed systems are essential in modern computing, powering applications ranging from social media to financial services. This report aims to analyze a distributed voting system's characteristics, divided into three components, and compare them to real-world systems like Twitter. We discuss the architectural choices, components used, and their impact on the system's performance, latency, reliability, and transparency.

System Overview:

Our distributed voting system is built using the following components:

- Component 1: Data analysis with Cloud Scheduler, Cloud Functions, Firestore, and DataProc (Constantin Elisabeta + Lazar Catalina)
- Component 2: Real-time voting with Cloud Run, Websockets, Firestore, and Pub/Sub (Ilas Silviu)
- Component 3: Add vote + Vote validation with Cloud Functions + UI for Component 2 (Prepelita Andrei)



Component 1:

Cloud Scheduler is a fully managed service that allows users to schedule tasks to be run in the future. In the real-time voting app, the Cloud Scheduler component is used to trigger a Cloud Function at a specific interval. This Cloud Function is used to export data from Firestore, which is Google's NoSQL document database.

The Cloud Function is triggered using cron expressions provided by Cloud Scheduler. The purpose of this function is to export a CSV file and upload it on a Cloud Storage Bucket to be used as input for the DataProc job.

DataProc is a fully managed analytics service that can be used to process large amounts of data. In our real-time voting app, the DataProc component is used to perform analytics on the CSV file exported from Firestore. The DataProc job is triggered by the Cloud Function after the CSV file is exported. The purpose of this job is to analyze the data and export the results as a JSON file.

The Cloud Storage bucket is a fully managed service that can be used to store data in the cloud. In our application it is used to store the CSV and JSON files exported by the Cloud Function and DataProc job, respectively.

Overall the combination of Cloud Scheduler, Cloud Function, Firestore, DataProc and Cloud Storage provides a scalable and efficient solution for our application.

Component 2 - On premise:

Key features:

- Reading from Firestore the initial poll values: This feature allows the application to retrieve the initial poll values from Firestore and display them to the user. This ensures that the user has the most up-to-date information on the poll.
- Calling Cloud Functions: This component allows the application to offload the functionality of adding new votes to a Cloud Function. This ensures that the application remains scalable and can handle a large number of votes without any performance issues.
- Emits a new event to Pub/Sub once a user votes: This feature ensures that the application is always aware of the latest votes and can update the UI accordingly. It sends a new event to Pub/Sub every time a user votes, which triggers an update to the UI.
- Connects with the UI through a websocket and listens to events from Pub/Sub to update the UI once a vote has been registered: This feature enables the UI to receive real-time updates whenever a user votes. The application uses a websocket to

establish a connection between the UI and the server. Whenever an event is received from Pub/Sub, the application updates the UI to reflect the latest vote count. This ensures that the user always has the most up-to-date information on the poll.
- Hosted on Google Cloud Run, a fully managed serverless platform that automatically scales up or down based on traffic.

Component 3 - Cloud Functions:

- According to the official Google cloud documentation, Cloud Functions is a *Scalable* pay-as-you-go function as a service (FaaS) to run your code with zero server management. Some of the key benefits of using cloud functions are: Cost, Scalability, Flexibility and Event-driven.
- In our real-time voting app the scope of the Component 3 is to process the request and to add a vote in Cloud Firestore. For that, I created a Cloud Function (name: addvote, 2nd gen) .Before adding the vote in the database it was necessary to validate the request in order to not allow users to vote twice. Also I verified that we have the options (answers) of a specific question saved in the database, otherwise I returned a status code: 404(Not found). The function (addvote) is triggered via HTTP request and the HTTP request is made by the Component 2

The system follows a client-server architecture, with a web-based frontend communicating with the backend server via RESTful APIs and Websockets. It also employs an event-driven architecture, utilizing Google Cloud Functions and Pub/Sub for processing and validating incoming votes.

Analysis & Discussion:

We analyze the voting system's performance, latency, reliability, and transparency dimensions and compare them to similar characteristics in real-world systems like Twitter.

- Performance: Our voting system leverages Firestore's real-time capabilities and DataProc for data analysis, providing fast and efficient data updates and insights. However, it may face performance challenges when handling a high volume of simultaneous requests, similar to Twitter during peak usage times.
- Latency: The system's latency is affected by the use of Google Cloud Functions and Websockets, which may introduce some overhead. However, by leveraging Cloud Run, we can achieve low-latency deployments, ensuring a responsive user experience.
- Reliability: Our system benefits from the inherent reliability of Google Cloud's infrastructure, with built-in redundancy and automatic failover. Twitter also relies on similar cloud-based infrastructure to maintain high availability.
- Transparency: The system employs a transparent architecture, allowing developers to easily understand the interactions between components. However, like Twitter, the use of proprietary cloud services may limit transparency in terms of data storage and processing.

Conclusion:

In conclusion, our three-component distributed voting system demonstrates the advantages and challenges of using modern cloud-based components and architectural patterns. By comparing our system to real-world systems like Twitter, we highlight the importance of performance, latency, reliability, and transparency in distributed systems design.