Sezim Yertanatov, Douglas Yang

Repo: https://github.com/dougyster/collaboration

Apr 3, 2025

So, today we met to decide what kind of project we would do, and stopped on the idea of a collaborative document editing system similar to Google Docs. The system will allows users to create, edit, and share documents with other users in a collaborative environment, and our focus will lie on fault tolerance, consistency, and real-time collaboration.

It will be based on a distributed client-server architecture with the following key components:

- **Frontend**: A React-based single-page application
    - For previous assignments our team leveraged tkinter, but that library is not adapted to modern features and demands; hence, after we added a threading mechanims, the UI became very slow and laggy. We had to reclick 5-10 times on a button for it to properly function. So, we learned our lessons, and we assume that React would the best tool to develop a user-friendly, fast, and clean UI. Also, both of us have a vast experience working with it.

- **Backend**: Multiple Flask servers forming a distributed system.
    - Since we will be using web interface, we decided to use Flask. Again, we are choosing simplicity and reliability. Moreover, we have a previous knowledge of the framework.

- **Communication Layer**: gRPC for both server-server and client-server communication.
    - Up until now, we have been using custom communication protocol based on our own configurations or JSON-based communication between servers and clients. However, since gRPC is the most common protocol in the industry, as well as, because it is one of the most valuable lessons we have learned in this class, we decided to stick with it.

- **Storage**: JSON-based file storage with replication across servers.
    - Keeping in mind that we will have replicated backend, storing JSON files are much easier and less error-prone to distribute than maintaining an external database.

Apr 10, 2025

Today, we accomplished the first steps of implementing the project, and instead of immediately jumping into coding a distributed system, we thought it would be wiser to start with more straightforward design just to understand the nuances. In particular, today we created just the frontend using React, including the login/registration page, dashboard page, and the document-editing page, along with the single-server Flask backend. In order to not complicate the matter, we used REST API for client-server communication. Also, local JSON files serve as a storage tool. Hence, now we have a completely functioning system and any two registered users can collaboratively edit the same document. This simple application will serve as a backbone for our further improvements.

By the way, we have not implemented the conflict resolution mechanism yet. Operational Transformation will be one of the last steps.

Here are the key implementation details so far:

- Data storage:
    - The system uses two primary data models:
        1. **User**: Username, Password (note: stored in plaintext right now, would use hashing in production), List of document IDs the user has access to.
        2. **Document**: Unique ID, Title, Content, Last edited timestamp, List of users with access

Apr 17, 2025

Our first improvement is to convert the client-server communication from REST API to gRPC, which will require us to make several significant changes to both the frontend and backend of our collaboration system. As we searched about it, we learned that apart from protocol buffers and gRPC tools, we will also need a gRPC-Web proxy that will stand as a bridge between browser gRPC-Web and server gRPC (browsers can't directly use gRPC). One of the options for the proxy is Envoy.

The key benefits of using gRPC over REST API, as we see them, are:

- Smaller message size and faster serialization/deserialization.
- Compile-time type checking for requests and responses.

- Support for server-to-client streaming (useful for real-time updates).
- Clear service and message definitions in .proto files.

We had a very ambitious plan, but after spending hours on implementing the gRPC and debugging it, we have not still managed to achieve the goal. Integrating gRPC is much harder than we initially thought. Also, we have never worked with gRPC proxies; thus, we have very little knowledge of it.

We spend another couple of hours on transforming REST API into gRPC but still there are some hidden bugs that are stopping the communication between the server and the client. Apparently, gRPC is hard to use with web browsers, hence, we just decided to keep REST API for client-server communication and focus on more important parts of the system.

Apr 20, 2025 - Apr 25, 2025

Now, our main goal is to build the distributed backend that is 2-fault tolerant. We have gained enough experience from replicating the servers for the Chat app, and similar ideas will be applied inspired by the RAFT consensus algorithm.

Configuration:

The system consists of 3 replica servers, which allows it to tolerate up to 2 server failures while maintaining normal operation (If two servers fail, the remaining server automatically becomes the leader). Each server has:

- A unique server ID
- An HTTP endpoint for client communication
- A gRPC endpoint for server-to-server communication
- An independent database file for persistence

Leader Election Process:

- **Initial State**

  When the system starts:

  1. All servers begin in the follower state
  2. Each server initializes with a randomized election timeout (2-4 seconds)

3. Servers wait for heartbeats from a leader

- **Election**

  An election is triggered when a follower's election timeout elapses without receiving a heartbeat. In that case:

  1. The follower transitions to the candidate state
  2. The candidate increments its term number
  3. The candidate votes for itself and requests votes from other servers
  4. A candidate becomes the leader if it receives votes from a majority of servers (2 in our case) or remains alone

  If multiple candidates start elections simultaneously, votes may be split, with no candidate receiving a majority. So, after a timeout, a new election round begins with incremented terms. Often, randomized election timeouts help prevent continuous split votes.

Client Connection Process

- **Initial connection**
  1. It initializes with a list of all server addresses (from `config.js`)
  2. It attempts to connect to the first server in the list
  3. It sends requests to this server until a failure occurs
- **Write Request Handling**

  For write operations (POST, PUT, DELETE):

  - If the client connects to a follower, the follower redirects to the leader
  - The client follows the redirection and sends the request to the leader
  - The leader processes the request and replicates it to followers
  - The leader responds to the client after successful replication
- **Read Request Handling**

  For read operations (GET), any server (leader or follower) can serve read requests. This provides load balancing for read-heavy workloads. However, reads from followers might return stale data if replication is delayed

Failure Scenarios and Handling

If the leader fails, then followers stop receiving heartbeats. So, the new leader takes over and continues processing client requests. Clients that were connected to the failed leader will retry and eventually connect to the new leader. In case a follower fails, the other servers keep functioning normally, and the clients that were connected to the failed follower will retry connecting to remaining servers. Whatever situation happens, the user will not have to intervene as every key process is handled automatically.

So, now we have 3 backend servers with consistent and persistent storage that inter-communicate via gRPC. However, note that the current design is not a direct copy of the RAFT consensus algorithm but rather a self-implementation of it. Also, it is not really scalable.

Apr 28, 2025

We are almost done with our project, and now we have to work on integrating a simplified version of the operational transformation algorithm into our system in order to handle edit conflicts between two users on the same document.

The core of the implementation is found in the `BusinessLogic` class, specifically in the `update_document_content_with_merge` method and the supporting `_merge_diffs` method.

Key Components of the OT Implementation:

1. **Three-Way Merge Approach**

   The system uses a three-way merge strategy, which requires:

   - **Base Content**: The original document version that both users started with
   - **Current Content**: The version currently stored in the database
   - **New Content**: The version being submitted by the user

   This approach differs from classic OT, which typically transforms operations directly. Instead, this implementation:

   - Compares the base content with both the current and new content

- Identifies the changes made in each version
- Attempts to merge these changes intelligently

2. **Diff-Based Change Detection**

   The system uses Python's difflib.Differ to detect changes between versions. This produces a line-by-line comparison showing:

   - Lines that are unchanged (starting with ' ')
   - Lines that were added (starting with '+ ')
   - Lines that were removed (starting with '- ')

3. **Conflict Resolution Strategy**

   The merge algorithm in `_merge_diffs` handles several scenarios:

   - **Non-conflicting changes**: Changes to different lines are both applied.
   - **Line deletion conflicts**: If both users delete the same line, the deletion is applied once.
   - **Modification conflicts**: If both users modify the same line differently, both changes are kept with a conflict marker.
   - **Mixed conflicts**: If one user deletes a line while another modifies it, the modification is preserved.

4. **Fallback Mechanism**

   The system includes a fallback to "last-write-wins" if the merge algorithm fails. This ensures that the system remains operational even if the merge algorithm encounters unexpected situations.

The three key design choices are:

1. Use line-based diffing rather than character-based operations since:
   - Simpler to implement and understand
   - Sufficient for most text-editing scenarios
   - Leverages Python's built-in difflib library
   - Reduces computational complexity

   However, this method cannot handle fine-grained edits within the same line optimally.

2. Require the client to send the base version it was working with since:
   - Simplifies the merge algorithm by providing a common reference point
   - Reduces the need for complex transformation matrices

    - Makes it easier to detect and resolve conflicts
3. Use a relatively simple conflict resolution strategy because:
    - Easier to implement and maintain
    - Sufficient for the needs of this collaborative system
    - Predictable behavior for users
    - Prioritizes preserving changes over perfect merging

May 4, 2025

We presented our project at the fair, and the TFs liked our idea and the implementation. Today, we performed some unit and interaction testing, and all the test cases verified the validity and correctness of our codebase.

In general, through building this final project and working on the previous assignments, we gained a wealth of practical knowledge about distributed systems. To be particular,

- Fault-Tolerance
    - **Practical Implementation of Raft**: Students gain hands-on experience implementing a simplified version of the Raft consensus algorithm, understanding leader election, log replication, and safety guarantees.
    - **State Machine Replication**: Learning how to maintain consistent state across multiple servers using replicated logs.
    - **Failure Handling**: Understanding how distributed systems can continue functioning despite node failures.
    - **Persistent and Consistent Storage**: • Learning how to replicate data across multiple nodes for reliability.
- Distributed System Architecture
    - **Service-Oriented Design**: Experience with separating concerns into distinct layers (controller, gateway, distributed server, business logic, data access).
    - **Client-Server Communication**: Understanding the tradeoffs between different communication protocols (REST vs gRPC vs Custom protocol vs JSON).
    - **Server-Server Communication**: Learning how to design efficient inter-server protocols using gRPC.
- System Design Skills
    - Rather than jumping straight into coding, we learned to build oru project brick-by-brick by clearly outlining each step and evaluating the consequences of each of choice we made. It is important to be aware of

what you are doing and why you are doing regarding every component of the system. Especially, maintaining an engineering notebook was a stimulus for us to keep track of design choices and document every step. Moreover, we learned balancing read operations across multiple servers and minimizing network overhead in server-to-server communication as well as identifying bottlenecks and designing the application while keeping the back of the mind the horizontal scaling.

- Thorough testing is important, even at the college level.
    - We suppose this point does not require any elaboration. At the least case, unit and integration testin are must-have.
- Concurrency and Consistency Models
    - **Optimistic Concurrency Control**: Understanding how to handle concurrent operations without locking.
    - **Operational Transformation**: Implementing a simplified OT algorithm for collaborative editing.

This project along with the past assignments served as an excellent bridge between theoretical distributed systems concepts and their practical implementation, giving us a deeper understanding of the challenges and solutions in building robust distributed applications.