67519 Reliability of Distributed Systems
Final Project

Yosef Edery, Eli Chouatt

April 2024

Contents

1 Main idea

1.1 Pseudocode

Algorithm 1 Pay Operation - Client side (Payment to a client x)

Given: a token t with version v
Send $\langle \text{pay}, t, x, v+1 \rangle$ to all
Wait for $n - f$ acknowledgements $\langle \text{ackPay} \rangle$

Algorithm 2 Pay Operation - Server side

Upon receiving $\langle \text{pay}, t, x, v+1 \rangle$ from client
if t.version $< v + 1$ **then**
    Update t.owner $\leftarrow x$ and t.version $\leftarrow v + 1$
end if
Send $\langle \text{ackPay} \rangle$ to client

Algorithm 3 GetTokens Operation - Client side

Send $\langle \text{getToken} \rangle$ to all
Wait for $n - f$ acknowledgements $\langle \text{ackToken}, t, *, * \rangle$
Let $\langle \text{ackToken}, t, x, r \rangle$ be the most recent version r
Send $\langle \text{pay}, t, x, r \rangle$ to all
Wait for $n - f$ acknowledgements $\langle \text{ackPay} \rangle$

Algorithm 4 GetTokens operation - Server side

1: Upon receiving ⟨getToken⟩ from client
2: **Send** ⟨ackToken, t, t.owner, t.version⟩ to client

1.2 Implementation

For the implementation, I've utilized the asyncio library. This library allows multiple operations, which are executed concurrently in a single-threaded program, and is capable to handle multiple network communications simultaneously. The implementation includes:

1.2.1 A command line interface

```
(base) user@MBP-de-Yosef ex4-Distributed % python main.py client
Client UI commands:
  pay <tokenid,version,newowner>  - Send payment or transfer ownership of a token.
  gettokens <ownerid>             - Retrieve tokens based on ownership.
  check                           - Verify connectivity with all servers.
  transform                       - Convert this client node into a server.
  help                            - Display this help message.
  quit                            - Exit the client interface.

>>
```

Figure 1: Server command line UI

```
(base) user@MBP-de-Yosef ex4-Distributed % python main.py server
Server UI commands:
  transform  - Convert the server into a client mode.
  quit       - Shut down the server.
  help       - Display this help message.

>>
```

Figure 2: Client command line UI

In which we can run client and server and perform operations from the command line. The server can run 2 commands:

1. **Transform** Transforms the server to a client.
2. **Quit** Closes the server connection and terminates the server's operation.

The client can run 5 commands:

1. **Pay** Initiates the payment command, where the user inputs the tokenId, version, and new owner.

2. **Get Tokens** Retrieves tokens by sending a gettokens command. The user inputs the owner ID to fetch tokens owned by the specified user.

3. **Transform to Server** Converts the client into a server without exceeding the maximum limit (7 servers).

4. **Quit** Terminates the client application.

5. **Check Server Connections** Checks and displays the current number of servers running the protocol.

```
Client UI commands:
  pay <tokenid,version,newowner>  - Send payment or transfer ownership of a token.
  gettokens <ownerid>             - Retrieve tokens based on ownership.
  check                           - Verify connectivity with all servers.
  transform                       - Convert this client node into a server.
  help                            - Display this help message.
  quit                            - Exit the client interface.

>> check
Checking server connections...
127.0.0.1:8889: Connected
127.0.0.1:8890: Connected
127.0.0.1:8891: Connected
127.0.0.1:8892: Failed to Connect
127.0.0.1:8893: Failed to Connect
127.0.0.1:8894: Failed to Connect
127.0.0.1:8895: Failed to Connect
>> gettokens 1
[(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1)]
>> pay 1,2,2
OK
>> gettokens 2
[(1, 2), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1), (18, 1), (19, 1), (20, 1)]
```

Figure 3: A running example for the client

1.2.2 Simulation tests

To test the implementation and assess liveness and safety, I've conducted operations involving gettokens and pay that run concurrently, including random faulty servers satisfying $n/2 > f$ and using random delays.

1.3 Detailed Code Implementation Overview

The implementation of the distributed token management system uses the asyncio library to handle asynchronous operations efficiently. This section provides a detailed overview of the core functionalities of the server and client, specifically highlighting the operational code and interactions within the system.

1.3.1 Server Implementation

The server side of the system is designed to manage tokens, handle client requests, and simulate network faults for testing the system.

- **Token Management:** The server maintains a list of token objects where each token is represented as a tuple containing the token's ID, version, and owner. This list is managed by the `TokenManager` class, which provides methods for updating token data and retrieving token information based on token IDs.
- **Network Communication:** Servers use the `asyncio` library to listen for incoming connections and handle requests asynchronously. The `Server` class encapsulates the functionality to start the server, listen for client requests, process these requests, and send responses back to the client.
- **Request Handling:** The server distinguishes between *pay* and *gettokens* commands. Upon receiving a *pay* command, it updates the token's ownership if the provided version is higher than the current version. For *gettokens* commands, it returns the state of the requested token.
- **Fault Simulation:** Faults and delays can be introduced deliberately as part of the server's functionality to test how well the system copes with adverse conditions. This is managed by randomizing delays or omitting responses.
- **ransform to client:** The server can change to a client during runtime.

1.3.2 Client Implementation

The client component is responsible for interacting with servers to perform token transactions and query token states.

- **Sending Requests:** Clients send asynchronous requests to servers using the `asyncio.open_connection` method. This includes sending *pay* and *gettokens* commands and handling server responses.
- **Response Handling:** Clients await responses from the server, ensuring that actions such as payments or queries are acknowledged by a majority of the servers.
- **Server Connectivity Checks:** Clients can initiate checks to determine the availability of servers.
- **Transform to Server:** Clients can transition to server mode.

2 Pseudo code

2.1 Server Class

```
Class Server:
    Initialize(token_manager, simulate_faults, delay):
        Set up token management, fault simulation settings, and operational delay.

    Run server asynchronously:
        Start main server task to handle incoming connections.
        Start user input listener for server control commands.


    Handle client connection:
        Read and parse incoming request.
        Simulate network delay if configured.
        Randomly drop connections if fault simulation is enabled.
        Execute command from request:
            PAY_COMMAND: Update token ownership if the request has a higher version.
            GET_TOKENS_COMMAND: Retrieve and send token data.
        Respond to client with result.
        Close connection after sending response or on simulated fault.
```

```
    Transform to client:
        Close server and switch operation mode to client.

    Input listener:
        Provide interactive command interface
```

2.2 Client Class

```
Class Client:
    Initialize:
        Set up server list and define majority requirements for operations.

    Transform to server:
        Switch from client mode to server operation.

    Send request to a single server:
        Establish connection
        Send request
        Close connection after receiving response.

    Broadcast request for token data (GETTOKENS):
        Send request to all servers asynchronously.
        Collect and aggregate responses to determine the most recent token version.
        Update token ownership after n-f responses with the most rencent version.
        Initiate a payment operation on the most recent version received.

    Broadcast request for payment (PAY):
        Send payment request to all servers asynchronously.
        Await confirmation from a majority of servers before finalizing the payment.

    Execute payment operation:
        Prepare payment details and initiate broadcast to servers.
        Return operation status.

    Execute gettokens operation:
        Split token retrieval into batches.
        Process each batch asynchronously.
        Wait all token results and return token ownership.

    Check server connectivity:
        Verify active connections with each server and return status.

    Run client interface:
        Provide an interactive command interface.
```

3 System Architecture Overview

The application is structured to handle dynamic connectivity and role transformations between clients and servers. It operates over a set of 7 predefined ports, each potentially hosting a server. The operations are detailed as follows:

3.1 Client to Server Communication

- **Connection Establishment:** Before performing any operation, the client attempts to establish connections with all servers across the predefined ports. This step ensures that the client is aware of all active servers at the time of the operation.
- **Broadcasting Requests:** Once the connections are established, the client broadcasts its request to all connected servers. This approach allows the client to dynamically determine the number of active servers (denoted as n) and calculate the necessary acknowledgments (denoted as $n - f$) for operations to be considered successful.

3.2 Server Transformation to Client

- **Deactivation of Server Role:** When a server decides to transform into a client, it discontinues its server functions by closing the server socket associated with its port, rendering the port inactive.
- **Impact on Connectivity:** The deactivation of a server results in the failure of new connection attempts to that port. This signals to other nodes that the server has left the network, allowing them to adjust their operational parameters accordingly.
- **Client Instance Initialization:** Post-deactivation, the former server initializes a new client instance to participate in the network as a client.

3.3 Client Transformation to Server

- **Acquiring a Server Role:** A client can transform into a server by attempting to bind to one of the predefined ports. It iterates through the ports until it finds one that is free, indicating no active server connection.
- **Connection Handling:** Unlike a server, which maintains persistent connections, a client typically establishes connections only during active command operations and closes them immediately afterward. This behavior facilitates the transformation process without the need for managing ongoing connections.
- **Network Recognition of New Server:** Other nodes recognize a new server in the same manner they would detect a server joining the network externally. Each client continuously checks the status of server ports during their operations and adapts to the changes by connecting to newly activated ports.

4 Implementation of Liveness Tests

The liveness tests are implemented using Python's `unittest` framework, specifically designed to facilitate asynchronous testing of network operations and server-client interactions.

4.1 Test Setup

- A `ServerSimulation` instance is created to simulate server behavior and handle client requests.
- A `Client` instance is prepared to perform operations against the simulated servers.

4.2 Automated Liveness Test

The `test_system_responsiveness` coroutine performs the following operations:

1. Initializes servers with fault simulation and introduces artificial delays to mimic real-world operational challenges.

2. Simultaneously sends multiple 'gettokens' requests from the client to all servers.
3. Gathers and evaluates all responses to ensure none are null or empty, indicating successful server response despite simulated faults.
4. Validates that each response adheres to expected formats and contains valid data

5 Safety Testing

Using Python's `unittest` framework, we simulate both normal and adverse conditions to validate the system's responses to client operations including network delays and server faults, to ensure that the system maintains correct operation.

5.1 Concurrent Payment Operations Test

This test validates the system's ability to handle multiple concurrent payment operations under fault conditions.

- **Test Logic:** Servers are initialized with intentional faults and delays. Concurrent payment requests are sent to the servers, and the system's response to each request is verified to be successful.
- **Validation:** The test confirms that all payment operations return a successful acknowledgment, ensuring the system's resilience to faults.

5.2 Concurrent GetTokens Operations Test

Tests the system's response to concurrent 'gettokens' requests, ensuring that the system provides accurate and complete token information despite simulated server faults.

- **Test Logic:** Similar to payment tests, but focuses on retrieving token ownership information under fault conditions.
- **Validation:** Ensures that responses contain complete token data for each request.

5.3 Comprehensive Safety Test

A rigorous test that compares system states after simulated fault runs with control runs to ensure consistent system behavior.

- **Faulty and Control Runs:** The system is tested under both faulty conditions and normal conditions. Operations are performed in both scenarios, and the resulting system states are captured.
- **State Comparison:** Uses cryptographic hashing to compare states from faulty runs and control runs, verifying that the system maintains consistency and correctness across different operational conditions.