Chat App with Replication

Github Repository: https://github.com/sezimy/replication.git

System Overview

This distributed chat application implements a fault-tolerant messaging platform using primary-backup replication. The system maintains consistency across multiple server instances, allowing continuous operation even when servers fail. Using a Raft-inspired leader election protocol, the system automatically elects new primary servers when failures occur, ensuring high availability.

Architecture

Client Component

1. Frontend Technologies:
    a. Python with socket programming for server communication
    b. Threading for concurrent operations
    c. JSON-based wire protocol
2. Key Features:
    a. Automatic reconnection to available servers
    b. Transparent primary server discovery
    c. Fault-tolerant message delivery

(Individual) Server Component

1. Backend Technologies:
    a. Python for socket programming for client and inter-server communication
    b. Threading for concurrent client handling
    c. Threading for concurrent inter-server communication
    d. Primary-backup replication protocol
2. Core Components:
    a. Replication Manager for server coordination
    b. Leader election protocol
    c. Client request handler
    d. State replication mechanism

Replication Architecture

1. Primary Server
    a. Stores server state (Primary, backup, or candidate)
    b. Handles all client write operations
    c. Replicates the state to backup servers
    d. Sends periodic heartbeats
    e. Coordinates backup servers via write operation replication
2. Backup Servers:
    a. Forward client requests to the primary server
    b. Monitor primary through heartbeats
    c. Participate in leader elections
    d. Maintain replicated state

Communication Protocol

Message Types

```
{
    "HEARTBEAT": "Primary-to-backup heartbeat",
    "REQUEST_VOTE": "Election vote request",
    "VOTE_RESPONSE": "Election vote response",
    "REPLICATE": "State replication message",
    "FORWARD": "Forwarded client request"
}
```

Message Structure

```
{
    "type": "MESSAGE_TYPE",
    "term": number,         // Election term
    "server_id": string,    // Sender's ID
    "payload": any          // Data for the message
}
```

Server States

```
class ServerRole(Enum):
    PRIMARY = "PRIMARY"
    BACKUP = "BACKUP"
```

```
    CANDIDATE = "CANDIDATE"
```

Replication Protocol

Leader Election

1. Random election timeouts (1.5-3.0 seconds)
2. Term-based voting system
3. Majority vote requirement
4. Automatic primary failure detection
5. Automatic primary detection if the server is the candidate for too long (indicating that we are the only server left)

State Replication

1. Primary processes all write operations
2. Synchronous replication to backups

Deployment Instructions

Server Deployment

1. Configure server addresses in `setup_distributed.sh`
2. Set up a virtual environment and install dependencies
3. Run the setup script for each server:

```
./setup_distributed.sh 1  # For server 1
./setup_distributed.sh 2  # For server 2
./setup_distributed.sh 3  # For server 3
```

Client Deployment

1. Configure the client with server addresses
2. Install required dependencies
3. Run the client application

```
SERVER_ADDRESSES = {
    ('10.250.103.230', 8081): "replica1",
    ('10.250.103.230', 8082): "replica2",
    ('10.250.145.247', 8083): "replica3"
}
```

Troubleshooting

Common Issues

1. Candidate timeout: Constant polling of the election_timeout_loop prevents the last candidate from becoming primary (solved)
2. Client request forwarding to primary: The backup correctly forwards the request to the primary, but the server doesn't handle it appropriately (solved).
3. Replication occurring in dead servers: We suppose this isn't a bad thing since this would be bonus consistency.
4. Inefficient lock holding during inter-server communication causes client requests to hang when the server receives them.

Performance Considerations

Scalability

1. Supports multiple backup servers across different devices
2. Automatic failover capability if the primary goes down.
3. Distributed request handling

Limitations

1. Single primary bottleneck for handling client communication.
2. Client iterates through servers to find the primary - inefficient.

Future Enhancements

1. Asynchronous replication option
2. Dynamic server addition/removal
3. Read-from-backup capability

Security Considerations

1. Inter-server encryption
2. Client-server secure channel
3. Vote verification
4. State validation

(*Generative AI was used to write parts of this documentation)