Universidade Federal do Rio Grande do Sul **Instituto de Informática**
Professor Alberto Egon Schaeffer Filho **alberto@inf.ufrgs.br**
INF01120 Sistemas Operacionais II N **Turma B (2021/1)**

Trabalho Prático (Parte 2)

Iron P. da Silva (00231590)
Luiza Schmidt (00281954)
Mayra Cademartori (00301639)
Tiago S. Ceccon (00278142)

10 de novembro de 2021

1 Especificações

Durante o desenvolvimento foi utilizado o sistema operacional Linux/Ubuntu com compilador GCC 7.5.0. Para gerar os makefiles e compilar foi utilizado CMake 3.21.3.

1.1 Vídeo

O vídeo para demonstração das funcionalidades básicas se encontra em https://youtu.be/bVOQmIIWVT0.

2 Implementação

2.1 Eleição

2.1.1 Heartbeat e Perda de Conexão

O recebimento constante de pacotes estilo "I'm alive"é utilizado para monitorar o estado de conexões. A cada 5 segundos, pelo menos um pacote do tipo `HEARTBEAT` é recebido a fim de garantir o funcionamento correto dos Servers. Caso aconteça alguma espécie de falha, é estourado um timeout por falta de resposta e sua conexão é dada como perdida.

```
void ReplicaConnection::_receivePacket(bool ignoreTimeout) {
...
       if (difftime(now, _lastReceivedHeartbeat) > _timeout) {
           _lostConnection();
       }
...
   }
```

Com a falha reconhecida, é encerrada qualquer conexão pendurada com o Server defeituoso. Caso o erro tenha ocorrido com o Server cujo id é equivalente ao id do `ReplicaManager` líder, `ElectionManager` começa o processo de eleição.

```
void ReplicaConnection::_lostConnection() {
    if (!_connected) return;

    _connected = false;
```

```
    if (_isServer) {
        ServerConnectionManager::closeConnection(_sfd);

    } else {
        _cm->closeConnection();
    }

    _sfd = -1;

    std::cout << "Lost server " << _otherID << std::endl;

    if (_otherID == _em.getLeaderID()) {
        _em.unsetLeaderIsAlive();
    }
}

void ElectionManager::unsetLeaderIsAlive() {
    _leaderIsAlive = false;
    _startElection();
}
```

2.1.2 Eleição de líder: Algoritmo do Valentão/Bully

No início do processo de eleição, instâncias backup de ReplicaManager tem a propriedade `_state` de seu `ElectionManager` (classe para manejamento de estados) inicializado como réplica. É definida a próxima ação do processo como `SendElection` e estabelecido valor de timeout.

```
    void ElectionManager::_startElection() {
        _state = State::REPLICA;
        _isLeader = false;
        _action = Action::SendElection;
        _epoch += 1;

        time(&_waitAnswerTimer);
    }
```

A eleição segue a partir do seguinte laço presente na função `start()` de `ReplicaManager`, executando com todos outros Servers backups a função `electionState` a partir da ação de seu `ElectionManager` até um novo líder ser eleito (ou o processo em questão ser terminado):

```
void ReplicaManager::start() {
    ...
    while (signaling::_continue && !_em.unlockedLeaderIsAlive()) {
        ElectionManager::Action action = _em.action();
        for (auto con : _connections) {
            con->electionState(action);
```

```
        }
        _em.step();
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

    }
    ...
```

Em `electionState`, A RM com ação equivalente a `SendElection` envia pacotes do tipo `ELECTION` informando seu id para outros processos cujos ids tem valor menor que o seu (maior hierarquia), avisando-os da nova eleição.

```
    void ReplicaConnection::electionState(ElectionManager::Action action) {
        switch (action) {
            case ElectionManager::Action::SendElection:
                if (_thisID > _otherID) {
                    ServerConnectionManager::dataSend(_sfd,
                        PacketBuilder::serverSignal(_thisID,
                        PacketData::PacketType::ELECTION, _em.epoch()));
                } break;
    ...
```

Esta RM agora espera, no laço de `start()`, respostas do tipo `ANSWER` de outros processos que lhe informarão caso essa tenha perdido a eleição para RMs de maior valor hierárquico.

```
    void ElectionManager::step() {
        switch(_action) {
            case SendElection:
                _action = Action::WaitAnswer;
                ...
                break;
        ...
    }

    void ReplicaConnection::electionState(ElectionManager::Action action) {
        switch (action) {
            ...
            case ElectionManager::Action::WaitAnswer:
            case ElectionManager::Action::WaitElection:
                _receivePacket(true);
                break;
            ...
    }

    void ReplicaConnection::_receivePacket(bool ignoreTimeout) {
        ...
            switch(packet.type) {

                case PacketData::PacketType::ELECTION:
```

```
            ServerConnectionManager::dataSend(_sfd,
                PacketBuilder::serverSignal(_thisID,
                PacketData::PacketType::ANSWER, _em.epoch()));

        case PacketData::PacketType::ANSWER:
            _em.receivedAnswer(packet.seqn);
            break;
        ...
```

Ao receber resposta, a RM em questão se conforma ao seu estado de réplica e espera o resultado final da eleição, quando receberá informações do novo processo coordenador. Caso demore demais, um timeout executa um novo processo de eleição.

```
void ElectionManager::receivedAnswer(uint16_t epoch) {
    if (_leaderIsAlive) return;
    if (epoch < _epoch) return;

    _state = State::REPLICA;
    _action = Action::WaitElection;
    time(&_waitElectionTimer);
}
```

Caso a RM não obtenha resposta do tipo `ANSWER` em um limite de tempo, é coroada como novo processo coordenador e deve comunicar sua vitória para outras RMs.

```
ElectionManager::Action ElectionManager::action() {
    time_t now;
    time(&now);

    if (_action == Action::WaitAnswer && difftime(now, _waitAnswerTimer) >
        _waitAnswerTimeout) {
        receivedCoordinator(_id, _epoch);
        _leaderIsAlive = true;
        _action = Action::SendCoordinator;
        ...
}

void ReplicaConnection::electionState(ElectionManager::Action action) {
switch (action) {
    ...
    case ElectionManager::Action::SendCoordinator:
        ServerConnectionManager::dataSend(_sfd,
            PacketBuilder::serverSignal(_thisID,
            PacketData::PacketType::COORDINATOR, _em.epoch()));
        break;
    ...
}
```

Os processos envolvidos na eleição (possuindo estado provisório equivalente à `REPLICA`)

agora atualizam seus devidos estados de acordo com o resultado final do algoritmo bully.

```
void ElectionManager::receivedCoordinator(unsigned short id, uint16\_t epoch) {
    _epoch = epoch;

    if (_state == State::REPLICA) {
        _leaderID = id;
        if (id == _id) {
            _isLeader = true;

            _state = State::LEADER;
        } else {
            _state = State::REPLICA;
        }

        _leaderIsAlive = true;
        _action = Action::None;
    }
}
```

2.1.3 Justificativa

Apesar de o algoritmo bully requerer mais volume de banda durante seu funcionamento comparado à outros protocolos de eleição, este foi priorizado por ser um algoritmo relativamente simples.

2.2 Replicação

2.2.1 Implementação

Para permitirmos a funcionalidade de replicação, criamos uma abstração acima do conceito de Server que tínhamos na Etapa 1, a classe `ReplicaManager`. Agora, todo `Server` herda de `ReplicaManager`. A ideia principal é que agrupamos em `ReplicaManager` a lógica de coordenação e sincronização entre os vários processos existentes de servidores. Exemplos disso são métodos para verificar se o servidor é o primário, se está esperando uma eleição, comitar e esperar por confirmação de commits das operações, etc.

Além disso, criamos a classe `ReplicationManager` para agrupar o gerenciamento da troca de mensagens entre os servidores indicando mudanças de estado no sistema. Exemplos são métodos para processar pacotes genéricos recebidos, gerar as informações de uma nova mudança que precisa ser replicada (struct mostrado abaixo), etc.

Para facilitar o tratamento dessas mensagens entre as réplicas, cada mensagem está sempre associada a um estado (`enum ReplicationState`, definido na class `ReplicationManager`) que ajuda a determinar qual ação deve ser tomada para tratá-la. Ainda, cada mensagem contém uma série de informações a ser respeito que são definidas no struct `ReplicationData`:

ReplicationManager.hpp

```
...

class ReplicationManager {
 public:
    typedef enum {
        SEND,      // Used by the leader to send a message for replication
        SENT,      // Used by the leader to wait confirmations
        CONFIRM,   // Used by the passive replicas to confirm replication
        CONFIRMDUP,
        COMMIT,    // Used by all replicas to indicate state propagation
        COMMITTED,  // Used by all replicas
        CANCELED
    } ReplicationState;

    typedef struct s_replication_data {
        PacketData::packet_t packet;
        uint64_t receivedAt;
        ReplicationState state;
        std::vector<bool> sentTo;
        unsigned short numFailures;
    } ReplicationData;

 ...
};
```

O servidor primário se encarrega de receber pacotes dos aplicativos clientes com comandos executados pelos usuários, assim como o único servidor que tínhamos na Etapa 1 já fazia. Agora, no entanto, esse servidor notifica todos os servidores backup e espera por confirmações deles antes de efetuar a mudança de estado e dar uma resposta para a operação solicitada:

Server.cpp

```
...
void Server::_handleUser(std::string username, int& csfd, SessionController*
    session) {

            ...

            if (packet.type == PacketData::PacketType::CLOSE) {
                is_over = true;
                waitCommit(PacketBuilder::replicateSession(username, "CLOSE," +
                    std::to_string(csfd)));

                _sm.closeSession(username, csfd);
                csfd = -1;
                std::cout << "Closed user " << username << " session\n" <<
```

```
                std::endl;

            } else if (packet.type == PacketData::PacketType::FOLLOW) {

                ...

                bool success = true;
                if (username != packet.extra) {
                    _sm.getControl();

                    if
                        (waitCommit(PacketBuilder::replicateFollower(packet.extra,
                        username))) {
                        SessionController* followee =
                            _sm.getSession(packet.extra);
                        if (followee != nullptr) {
                            followee->addFollower(username);
                        } else { // No session open
                            User user = _pm.loadUser(packet.extra, false);

                            if (user.name() == packet.extra) {
                                user.addFollower(username);
                                _pm.saveUser(user);
                            } else {
                                success = false;
                            }
                        }
                    }
                    _sm.freeControl();

                    ...
                }
            } else if (packet.type == PacketData::PacketType::MESSAGE) {
                std::string messageContent = packet.payload;

                ...

                auto messagePacket = PacketBuilder::replicateMessage(username,
                    messageContent);
                if (waitCommit(messagePacket)) {
                    session->sendMessage(messageContent,
                        messagePacket.timestamp);

                }
            }
        ...
};
```

O ponto principal a se notar no trecho de código acima são as chamadas a `waitCommit`,

das quais o código só avança após os backups vivos terem confirmado que replicaram o comando executado. Dentro de `waitCommit` é utilizado o método `newReplication`, de `ReplicationManager`, para produzir e enfileirar a mensagem que determina a execução desejada, mensagem essa que é posteriormente enviada para cada backup. Quando todos tiverem confirmado a replicação do comando em seus estados internos, a chamada para `_rm.getMessageState(replicationID);` (como vista no código abaixo) indicará que o estado da mensagem é `COMMITTED`, permitindo que o fluxo saia de `waitCommit` e o servidor primário possa por fim encerrar a execução dessa operação.

ReplicaManager.cpp

```
...
bool ReplicaManager::waitCommit(PacketData::packet_t commandPacket) {
    bool success = false;

    uint64_t replicationID = 0;

    ReplicationManager::ReplicationState state =
        ReplicationManager::ReplicationState::SEND;
    commandPacket.seqn = _em.epoch();

    _rmSem.beginRead();
    if (_rm.newReplication(commandPacket, replicationID)) {
        _rmSem.endRead();
        do {
            _rmSem.beginRead();
                state = _rm.getMessageState(replicationID);
            _rmSem.endRead();

            std::this_thread::sleep_for(std::chrono::milliseconds(25));

        } while (
            signaling::_continue
            && state != ReplicationManager::ReplicationState::COMMITTED
            && state != ReplicationManager::ReplicationState::CANCELED
        );

    } else {
        _rmSem.endRead();
    }

    if (state == ReplicationManager::ReplicationState::COMMITTED) {
        success = true;
    }

    return success;
}
...
```

2.3 Reconexão do cliente

Para que os clientes possam continuar funcionando após uma troca de líder, optamos por implementar a estratégia de manter no 'frontend' um socket que pode ser utilizado pelos servidores para informar ao cliente quem é o novo líder, ou seja, para qual processo ele deve encaminhar as requisições. Para isso, é possível informar via linha de comando ao iniciar o aplicativo cliente qual porta deve ser utilizada para ouvir esse sinal específico (tendo como valor default a porta "42366").

O cliente então abre um socket, utilizando a porta informada, ao inicializar o `ClientConnectionManager`. Alteramos o pacote para mensagens de login desenvolvido na Etapa 1 do trabalho para conter também a informação de qual a porta o cliente estará utilizando para ouvir notificações de mudança de líder, utilizando o campo "payload"para esse propósito:

PacketBuilder.cpp

```
...
packet_t PacketBuilder::login(std::string username, std::string listenerPort) {
    packet_t packet;

    packet.type = LOGIN;
    packet.rtype = R_NONE;
    packet.seqn = 0;
    packet.length = 0;
    packet.timestamp = std::chrono::duration_cast<std::chrono::milliseconds>
                       (std::chrono::system_clock::now().time_since_epoch()).count();
    strcpy(packet.payload, listenerPort.c_str());
    strcpy(packet.extra, username.c_str());

    return packet;
}
...
```

Do lado do servidor, essa informação é salva junto com as demais informações da sessão do cliente, portanto sendo replicada pelo servidor primário para todos os backups. Uma vez que haja uma eleição e um novo servidor seja eleito como primário, ele solicita que todos os backups limpem as informações antigas de sessões e informa todos os clientes sobre a mudança, que então prontamente encerram a conexão com o antigo líder e estabelecem uma nova com o novo servidor primário que passa a ser utilizada para as operações normais do sistema.

client/app/src/main.cpp

```
...
bool reconnect(std::string user, ClientConnectionManager& cm) {
...
    // Leader sent new info
    cm.closeConnection();
```

```
    ClientConnectionManager::closeConnection(serverSFD);

    // Connecting to Leader
    cm.setAddress(std::string(packet.extra));
    cm.setPort(std::string(packet.payload));
    std::cout << "Preparing reconnection..." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(3));

    if (cm.openConnection(true, false)) {

        ...
        // <Logica de confirmacao e prevencao de timeouts, etc.>
        ...

    }
...
    return accepted;
}
...
```

3 Dificuldades e Desafios

Algumas dificuldades surgem na necessidade de adaptações do código da primeira etapa para aportar os novos requerimentos. Essas foram consideradas de maneira geral já no início do desenvolvimento desta etapa. Modificações à classes auxiliares, como `ServerConnectionManager` e `ClientConnectionManager` foram implementadas no início para permitir um maior grau de liberdade em seu uso, como, por exemplo, que a última possa ser instanciada e mantida sem uma conexão ativa - requerido para reconexão sem fechar a aplicação cliente.

Outras alterações que influem diretamente da necessidade de replicação foram consideradas no planejamento inicial, mas implementadas apenas ao final da fase de replicação. Uma das grandes dificuldades surgiu a partir desta tentativa, a de abstrair as alterações de código a serem feitas após a implementação das novas funcionalidades de eleição e replicação. Essa abstração gerou algumas tentativas falhas até que o sequenciamento das ações sob responsabilidade do `ReplicaManager` estivesse bem definida e funcionando de maneira correta, fornecendo o serviço devido às aplicações cliente enquanto mantém também a coerência da comunicação entre réplicas.

Uma outra grande dificuldade inicial foi definir o que cada integrante deveria desenvolver e como integrar os códigos com funcionalidades distintas, tudo enquanto considerando os problemas já traçados. Uma primeira tentativa, onde o `ReplicaManager` trataria cada conexão com as outras réplicas de maneira concorrente, logo demonstrou-se muito complexa e propensa a falhas. Foi então desenhado um sistema de manipulação de estados

completamente síncrono dentro de cada `ReplicaManager`. A partir deste paradigma, a integração entre módulos passou a ser mais simples, clara e direta. Ainda assim, a integração correta dos módulos que compõe o `ReplicaManager` exigiu um esforço grande, com minúcias que, quando mal tratadas, culminavam em erro. Junto disso, como na primeira etapa, não conseguimos desenvolver um mecanismo eficaz para testar o código de maneira automatizada, devido à falta de conhecimento e experiência em como projetar e implementar esse tipo de teste automatizado. Isto certamente tornou a atividade mais exigente em termos de atenção, memória e paciência.

Após conseguir integrar as conexões entre réplicas à funcionalidade de eleição, determinar as estruturas de dados necessárias para a replicação do estado do servidor líder e como propagar tal estado nas réplicas tornou-se prioridade. Algumas estruturas já eram imaginadas e estavam implementadas, porém alguns erros estavam presentes por conta da granularidade das medidas de tempo em uso e, novamente, problemas de integração. Nesta fase do desenvolvimento, além dos esforços de integração da mudança de estados apenas dentro do `ReplicaManager`, grandes esforços também foram necessários na modificação do código anterior de modo a permitir a propagação das informações de estado recebidas. Algumas estruturas tiveram de ser alteradas de modo a tornar o acesso às suas outras estruturas internas mais flexível, além da modificação da parte da funcionalidade do backend que trata dos clientes individualmente.

Em suma, este foi um trabalho bastante desafiador em diversos aspectos. Entre eles, destacam-se a necessidade de um bom planejamento, a compreensão detalhada dos algoritmos para avaliar as modificações necessárias em código, trabalho em equipe constante e um bocado de paciência e resiliência.