



Detecção de Defeitos

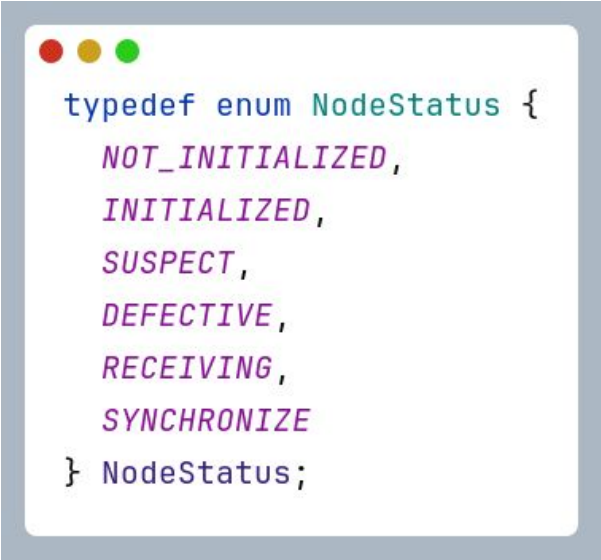
Ismael Coral Hoepers Heinzelmann, Marcos Tomaszewski, Matheus Paulon Novais, Sérgio Bonini



Estados possíveis de um nó

- **NOT_INITIALIZED** → Processo não inicializado
- **INITIALIZED** → Processo sincronizado com outros processos
- **SYNCHRONIZE** → Grupo possui um membro adentrando e existem mensagens a serem entregues
- **RECEIVING** → Utilizado no atomic broadcast para sincronização
- **SUSPECT** → Processo ficou dois *ticks* de *heartbeat* sem enviar um sinal
- **FAULTY** → Processo ficou três *ticks* de *heartbeat* sem enviar um sinal

Estados possíveis de um nó



```
typedef enum NodeStatus {  
    NOT_INITIALIZED,  
    INITIALIZED,  
    SUSPECT,  
    DEFECTIVE,  
    RECEIVING,  
    SYNCHRONIZE  
} NodeStatus;
```

Enumerador de estados de um nó.



Envio de *Heartbeats*

- Uma thread realiza o envio e verificação dos heartbeats a periodicamente;
- Este período é o parâmetro *alive* do arquivo de configurações, sendo assim esta será a precisão do tempo de verificação de falha dos nós.

Envio de *Heartbeats*

```
void MessageReceiver::heartbeat() {
    sigset_t newmask;
    sigemptyset(&newmask);
    sigaddset(&newmask, signo:SIGALRM);
    pthread_sigmask(how:SIG_BLOCK, &newmask, oldmask:nullptr);
    while (running) {
        {
            sendHEARTBEAT(target: { [&]channelIP, [&]channelPort}, broadcastFD);
            std::this_thread::sleep_for(rtime:std::chrono::milliseconds(rep:1 * aliveTimeMS));
            std::vector<std::pair<unsigned int, unsigned short>> removes;
            std::lock_guard lock([&]statusStruct->nodeStatusMutex);
            for (auto [identifier:const pair<unsigned, unsigned short>, time:time_point<system_clock>] : heartbeatsTimes) {
                auto oldStatus = statusStruct->nodeStatus[identifier];
                if (std::chrono::system_clock::now() - time >= std::chrono::milliseconds(rep:3 * aliveTimeMS) &&
                    statusStruct->nodeStatus[identifier] != NOT_INITIALIZED) {
                    statusStruct->nodeStatus[identifier] = DEFECTIVE;
                    // removes.emplace_back(identifier);
                }
                else if (std::chrono::system_clock::now() - time >= std::chrono::milliseconds(rep:2 * aliveTimeMS) &&
                    statusStruct->nodeStatus[identifier] != NOT_INITIALIZED) {
                    statusStruct->nodeStatus[identifier] = SUSPECT;
                }
                if (oldStatus != statusStruct->nodeStatus[identifier]) {
                    Logger::log(message:"New status for node " + std::to_string((*configs)[id].sin_port) + ": " +
                                Protocol::getNodeStatusString(statusStruct->nodeStatus[identifier]),
                                LogLevel::DEBUG);
                }
            }
        }
    }
}
```

```
[2024-11-25 10:19:11] [DEBUG] Listen thread started.  
Message type: 1 for unicast and 2 for broadcast  
[2024-11-25 10:19:18] [DEBUG] New status for node 47371: INITIALIZED  
[2024-11-25 10:19:20] [DEBUG] New status for node 47115: SUSPECT  
[2024-11-25 10:19:20] [DEBUG] New status for node 47115: DEFECTIVE  
[2024-11-25 10:19:24] [DEBUG] New status for node 47371: NOT_INITIALIZED  
[2024-11-25 10:19:25] [DEBUG] New status for node 47371: INITIALIZED
```

Atualização de estados para entrada tardia de um processo, saída do mesmo e então entrada novamente.

Modificações

```
void MessageSender::buildBroadcastDatagrams(  
    std::vector<std::vector<unsigned char>> *datagrams,  
    std::map<std::pair<unsigned int, unsigned short>, std::map<unsigned short, std::pair<bool, bool>>> *membersAcks,  
    in_port_t transientPort, unsigned short totalDatagrams, std::vector<unsigned char> &message,  
    std::map<std::pair<unsigned int, unsigned short>, bool> *members) {  
    std::map<unsigned short, bool> acknowledgments, responses;  
    {  
        std::shared_lock lock([&]statusStruct->nodeStatusMutex);  
        for (auto [identifier:const pair<unsigned, unsigned short>, nodeStatus] : statusStruct->nodeStatus) {  
            if (nodeStatus != NOT_INITIALIZED && nodeStatus != DEFECTIVE)  
                (*members)[identifier] = false;  
        }  
    }  
}
```

Broadcast utilizando apenas processos válidos para solicitação de ACKS de nova mensagem.

Entrada tardia

Processo interessante



Processo ingressante

- Este processo irá solicitar algumas vezes a entrada utilizando JOIN, aguardando que o grupo responda com JOIN-ACK;
- Se o grupo estiver vazio (nenhum *heartbeat*), inicializa;
- Se o grupo estiver com integrantes e não receber JOIN-ACK dos mesmos, repete o processo;
- Os JOIN-ACKS possuem a informação de quantas mensagens o canal possui, se for zero, inicializa;
- Se for diferente de zero, envia periodicamente mensagens SYNCHRONIZE, até que as mensagens no seu buffer de mensagens seja igual ao número de mensagens;
- Caso o canal fique vazio no processo (os processos válidos tiveram defeito), inicializa;

Processo interessante

```
void ReliableCommunication::configure() {
    Datagram joinDatagram = Datagram();
    joinDatagram.setSourcePort(this->configMap[id].sin_port);
    joinDatagram.setSourceAddress(this->configMap[id].sin_addr.s_addr);

    Flags flags;
    flags.JOIN = true;
    std::set<std::pair<unsigned, unsigned short>> joinACKS;
    unsigned messagesCounter = 0;

    auto broadcastAddr::sockaddr_in = Protocol::broadcastAddress();
    datagramController.createQueue(Identifier: { [&]configMap[id].sin_addr.s_addr, [&]configMap[id].sin_port});
    for (int i = 0; i < JOIN_RETRY; i++) {
        Protocol::sendDatagram(&joinDatagram, &broadcastAddr, broadcastInfo, &flags);
        while (true) {
            // Grupo inteiro já concordou
            if (joinACKS.size() == configMap.size() - 1)
                break;
            Datagram *response = datagramController.getDatagramTimeout(
                Identifier: { [&]this->configMap[id].sin_addr.s_addr, [&]this->configMap[id].sin_port}, timeoutMS: 100);
            if (response == nullptr)
                break;
            if (response->isJOIN() && response->isACK() && response->getData()->size() == 4) {
                messagesCounter = TypeUtils::buffToUnsignedInt(*response->getData(), 0);
                joinACKS.insert({x: response->getSourceAddress(), y: response->getSourcePort()});
            }
        }
    }
}
```

Método utilizado na inicialização da biblioteca, a fim de restaurar o estado do grupo.

Processo interessante

```
// Nenhuma mensagem no grupo
if (messagesCounter == 0) {
    statusStruct.nodeStatus[{{&this->configMap[id].sin_addr.s_addr, &this->configMap[id].sin_port}}] = INITIALIZED;
    handler->configure();
    return;
}

Datagram synchronizeDatagram = Datagram();
synchronizeDatagram.setSourcePort(this->configMap[id].sin_port);
synchronizeDatagram.setSourceAddress(this->configMap[id].sin_addr.s_addr);

flags.JOIN = false;
flags.SYNCHRONIZE = true;
while (handler->getBroadcastMessagesSize() != messagesCounter) {
    bool channelALive = true;
    {
        std::shared_lock lock(&statusStruct.nodeStatusMutex);
        for (auto [_:const pair<unsigned, unsigned short>, status:NodeStatus] : statusStruct.nodeStatus) {
            if (status == DEFECTIVE) {
                channelALive = false;
            }
        }
    }

    // Não há ninguém no canal
    if (!channelALive)
        break;

    Protocol::sendDatagram(&synchronizeDatagram, &broadcastAddr, broadcastInfo, &flags);
    std::this_thread::sleep_for(rtime:std::chrono::milliseconds(rep:250));
}

statusStruct.nodeStatus[{{&this->configMap[id].sin_addr.s_addr, &this->configMap[id].sin_port}}] = INITIALIZED;
handler->configure();
}
```

Método utilizado na inicialização da biblioteca, a fim de restaurar o estado do grupo.

Membros



Membros

- Membros do grupo responderam com JOIN-ACK quando o canal estiver disponível para a entrada de um ingressante, com a quantidade de mensagens do canal;
- Ao receber um SYNCHRONIZE, o processo válido com menor porta irá responder esta requisição, se a porta for igual, irá ser o processo com menor IP;
- Um processo válido é um processo já configurado o contexto do grupo;
- O processo escolhido irá criar uma *thread* que então irá retransmitir a mensagem para o grupo, membros que já tiverem recebido a mensagem apenas a responderão, porém sem entregar novamente.

```

void MessageReceiver::handleBroadcastMessage(Request *request) {
    std::shared_lock lock([&messagesMutex]);
    Protocol::setBroadcast(request);
    Datagram *datagram = request->datagram;
    if (datagram->isHEARTBEAT()) {
        std::pair identifier = {x: datagram->getSourceAddress(), y: datagram->getSourcePort()};
        std::unique_lock hblock([&heartbeatsLock]);
        heartbeats[identifier] = {x: datagram->getDestinAddress(), y: datagram->getDestinationPort()};
        heartbeatsTimes[identifier] = std::chrono::system_clock::now();
        treatHeartbeat(request->datagram);
        return;
    }
    // É uma mensagem mas o nó ainda não esta inicializado
    if (datagram->isSYNCHRONIZE() && status == NOT_INITIALIZED)
        return;
    // Recebeu concordância com join
    if (datagram->isJOIN() && datagram->isACK()) {
        datagramController->createQueue(identifier: {x: datagram->getDestinAddress(), y: datagram->getDestinationPort()});
        datagramController->insertDatagram(identifier: {x: datagram->getDestinAddress(), y: datagram->getDestinationPort()}, datagram);
        return;
    }
}

```

Tratamento de mensagens de entrada por parte de um membro.

```
if (datagram->isJOIN() && status != NOT_INITIALIZED) {  
    // Algum nó já está sincronizando  
    if (status == SYNCHRONIZE &&  
        (datagram->getSourceAddress() != channelIP || datagram->getSourcePort() != channelPort)) {  
  
        std::shared_lock statusLock(&statusStruct->nodeStatusMutex);  
        // Se o nó em sincronização morreu substitui o nó  
        if (statusStruct->nodeStatus[0] == NOT_INITIALIZED) {  
            return;  
        }  
    }  
  
    channelIP = datagram->getSourceAddress();  
    channelPort = datagram->getSourcePort();  
    sendDatagramJOINACK(request, broadcastFD);  
    return;  
}
```

Tratamento de mensagens de entrada por parte de um membro.


```
if (datagram->isSYNCHRONIZE()) {  
    auto smallestProcess :pair<unsigned int, unsigned short> = getSmallestProcess();  
    // Não há processo configurado  
    if (smallestProcess.first == 0 || smallestProcess.second == 0)  
        return;  
    // Este processo não deve responder a solicitação, pois não é o menor.  
    if (smallestProcess.first != configs->at(id).sin_addr.s_addr &&  
        smallestProcess.second != configs->at(id).sin_port) {  
        return;  
    }  
    // Não há thread dedicada para recepção.  
    if (!synchronizing) {  
        synchronyzeThread = std::thread(f: [this] ->void { synchronize(); });  
        synchronyzeThread.detach();  
        return;  
    }  
}
```

Tratamento de mensagens de entrada por parte de um membro.

```

void MessageReceiver::synchronize() {
    synchronizing = true;
    for (auto message : broadcastOrder) {
        if (message->delivered) {
            for (auto [identifier :const pair<unsigned, unsigned short>, m:Message*] : broadcastMessages) {
                if (m == message) {
                    if (!message->delivered)
                        continue;
                    auto data :vector<unsigned char> = *message->getData();
                    sender->synchronizeBroadcast( [&]data, identifier, target: { [&]channelIP, [&]channelPort}, m->origin);
                }
            }
        }
    }
    status = INITIALIZED;
    synchronizing = false;
}

```

Tratamento de mensagens de entrada por parte de um membro.

Resultados

```
Message type: 1 for unicast and 2 for broadcast
2
Broadcast type: URB
Write the message:
Mensagem 1
Message content: Mensagem 1
[2024-11-25 12:26:43] [INFO] Time spent: 0ms
Message sent successfully.
Message type: 1 for unicast and 2 for broadcast
2
Broadcast type: URB
Write the message:
Mensagem 2
Message content: Mensagem 2
[2024-11-25 12:26:45] [INFO] Time spent: 0ms
Message sent successfully.
Message type: 1 for unicast and 2 for broadcast
2
Broadcast type: URB
Write the message:
Mensagem 3
Message content: Mensagem 3
[2024-11-25 12:26:47] [INFO] Time spent: 51ms
Message sent successfully.
Message type: 1 for unicast and 2 for broadcast
```

Mensagens enviadas antes do segundo processo entrar, utilizando 2% de perda de pacote.

```
[2024-11-25 12:26:55] [DEBUG] Listen thread started.  
[2024-11-25 12:26:56] [DEBUG] New status for node 47627: NOT_INITIALIZED  
[2024-11-25 12:26:56] [FAULT] Packet received will be dropped and ignored.  
[2024-11-25 12:26:56] [FAULT] Packet received will be dropped and ignored.  
Message type: 1 for unicast and 2 for broadcast  
Message content: Mensagem 1  
Message content: Mensagem 2  
Message content: Mensagem 3
```

Mensagens recebidas em ordem pelo processo tardio, com 2% de perda de pacote.



Conclusão

- A biblioteca agora integra sistemas de *heartbeat* para obter o status de outros processos, auxiliando assim na tomada de decisões;
- Processos agora conseguem entrar de maneira tardia no grupo, além de entrar novamente no grupo em caso de defeitos.