Comunicação 1:n

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

Pendências P1

Timeout

- Método *getDatagramTimeout* recebe como argumentos um identificador da mensagem e um tempo em milisegundos para timeout de operação bloqueante *readDatagramSocket*
- Utiliza um *alarm* que dispara um *signal* para o processo após um tempo determinado
- Caso um datagrama seja recebido antes do timeout, cancela-se o *alarm* e prossegue a execução
- Caso ocorra um timeout, ou seja, o *alarm* seja disparado, um *jump* é executado e o fluxo de execução é guiado para um *timeout*

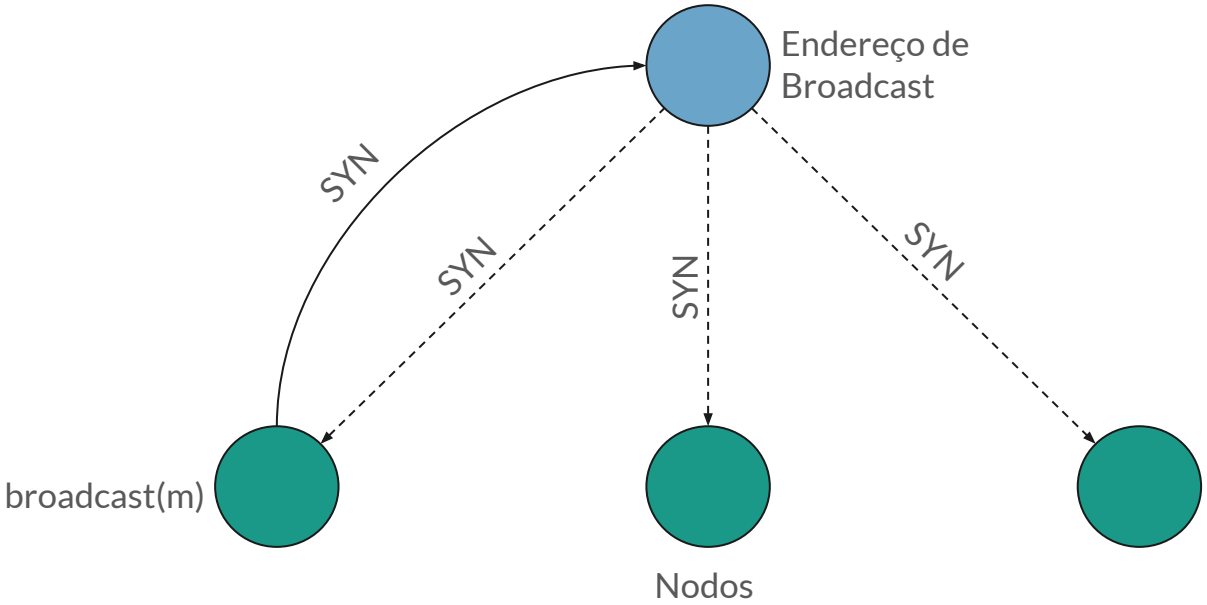
Timeout

```
void DatagramController::signalHandler(int) {
    if (waitingTimeout.load()) {
        waitingTimeout.store( i: false);
        longjmp(jumpBuffer,  val: 1);
    }
}
```

```
Datagram *DatagramController::getDatagramTimeout(const std::pair<unsigned int, unsigned short> &identifier,
                                                 int timeoutMS) {
    {
        std::shared_lock lock( [&]datagramsMutex);
        if (!datagrams.contains(identifier)) {
            return nullptr;
        }
    }
    sigset_t newmask, oldmask;
    sigemptyset(&newmask);
    sigaddset(&newmask,  signo: SIGALRM);
    pthread_sigmask( how: SIG_BLOCK, &newmask, &oldmask);
    if (setjmp( env: jumpBuffer) != 0) {
        pthread_sigmask( how: SIG_SETMASK, &oldmask,  oldmask: nullptr);
        ualarm( value: 0,  interval: 0);
        return nullptr;
    }
    std::signal( sig: SIGALRM, signalHandler);
    waitingTimeout.store( i: true);
    pthread_sigmask( how: SIG_UNBLOCK, &newmask,  oldmask: nullptr);
    ualarm( value: timeoutMS * 1000,  interval: 0);
    Datagram *datagram = datagrams.at(identifier)->pop();
    pthread_sigmask( how: SIG_SETMASK, &oldmask,  oldmask: nullptr);
    waitingTimeout.store( i: false);
    ualarm( value: 0,  interval: 0);
    return datagram;
}
```

Broadcast

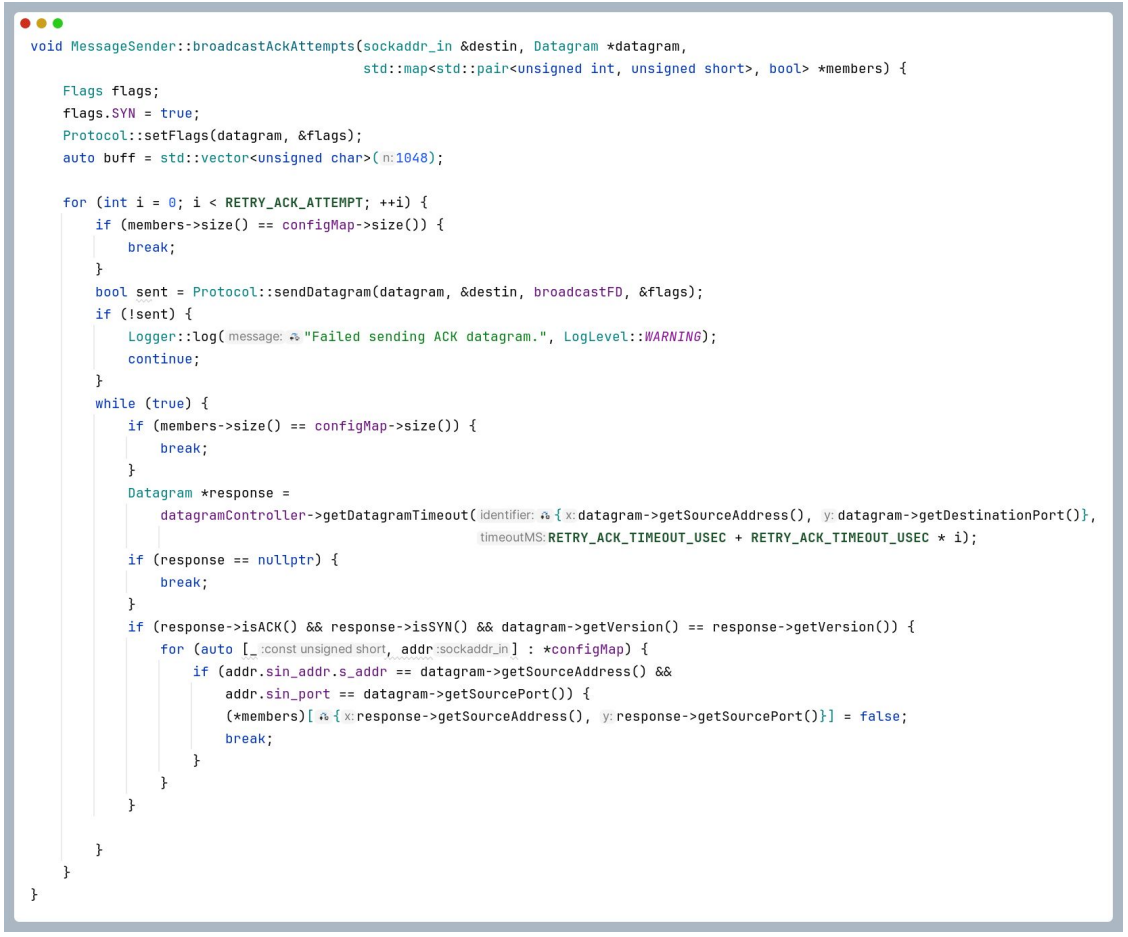
Começo do broadcast



Vendo a implementação

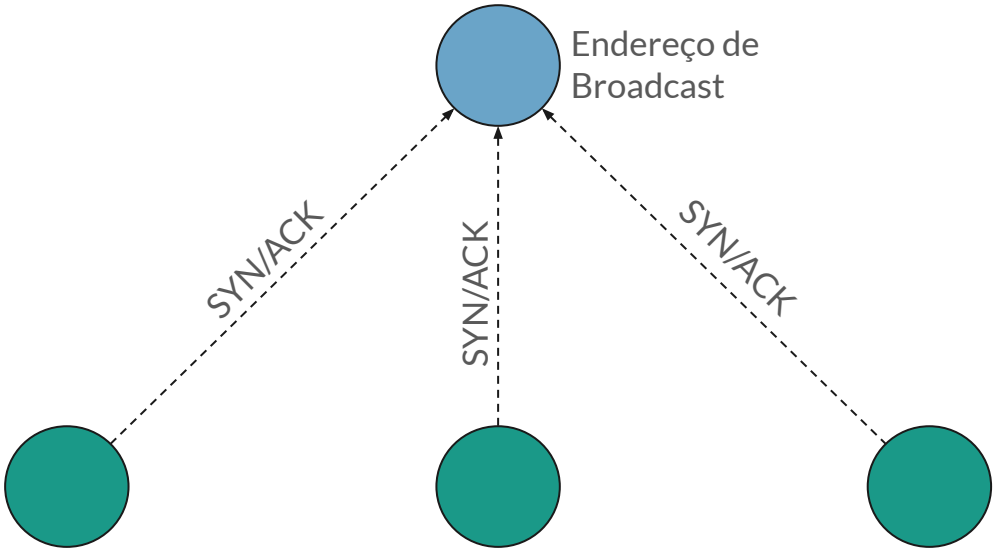
```
bool MessageSender::sendBroadcast(std::vector<unsigned char> &message) {

    std::pair<int, sockaddr_in> transientSocketFd = createUDPSocketAndGetPort();

    auto datagram = Datagram();
    unsigned short totalDatagrams = calculateTotalDatagrams( dataLength: message.size());
    datagram.setDatagramTotal(totalDatagrams);
    datagram.setSourceAddress(configAddr.sin_addr.s_addr);
    datagram.setSourcePort(configAddr.sin_port);
    datagram.setDestinationPort(transientSocketFd.second.sin_port);

    datagramController->createQueue( identifier: { [&]configAddr.sin_addr.s_addr,  [&]transientSocketFd.second.sin_port});

    sockaddr_in destin = Protocol::broadcastAddress();
    auto members = std::map<std::pair<unsigned int, unsigned short>, bool>();
    broadcastAckAttempts( [&]destin, &datagram, &members);
    if (members.empty()) {
        close(transientSocketFd.first);
        return false;
    }
```

Vendo a implementação



```
void MessageSender::broadcastAckAttempts(sockaddr_in &destin, Datagram *datagram,
                                         std::map<std::pair<unsigned int, unsigned short>, bool> *members) {
    Flags flags;
    flags.SYN = true;
    Protocol::setFlags(datagram, &flags);
    auto buff = std::vector<unsigned char>( n:1048);

    for (int i = 0; i < RETRY_ACK_ATTEMPT; ++i) {
        if (members->size() == configMap->size()) {
            break;
        }
        bool sent = Protocol::sendDatagram(datagram, &destin, broadcastFD, &flags);
        if (!sent) {
            Logger::log( message: "Failed sending ACK datagram.", LogLevel::WARNING);
            continue;
        }
        while (true) {
            if (members->size() == configMap->size()) {
                break;
            }
            Datagram *response =
                datagramController->getDatagramTimeout( identifier: { x:datagram->getSourceAddress(), y:datagram->getDestinationPort()},
                                                        timeoutMS:RETRY_ACK_TIMEOUT_USEC + RETRY_ACK_TIMEOUT_USEC * i);
            if (response == nullptr) {
                break;
            }
            if (response->isACK() && response->isSYN() && datagram->getVersion() == response->getVersion()) {
                for (auto [_ :const unsigned short, addr :sockaddr_in ] : *configMap) {
                    if (addr.sin_addr.s_addr == datagram->getSourceAddress() &&
                        addr.sin_port == datagram->getSourcePort()) {
                        (*members)[ { x:response->getSourceAddress(), y:response->getSourcePort()}] = false;
                        break;
                    }
                }
            }

        }
    }
}
```

Informado o recebimento do SYN



Vendo a implementação

```
void MessageReceiver::handleFirstMessage(Request *request, int socketfd, bool broadcast) {
    if (messages.contains( x: {x:request->datagram->getSourceAddress(), y:request->datagram->getDestinationPort()})) {
        sendDatagramSYNACK(request, socketfd);
    }
    auto *message = new Message(request->datagram->getDatagramTotal());
    if (broadcast) {
        message->broadcastMessage = true;
    }
    messages[{x:request->datagram->getSourceAddress(), y:request->datagram->getDestinationPort()}] = message;
    sendDatagramSYNACK(request, socketfd);
}
```

Vendo a implementação

```
void MessageReceiver::handleBroadcastMessage(Request *request, int socketfd) {
    std::shared_lock lock([&]messagesMutex);
    if (!verifyMessage(request)) {
        sendDatagramNACK(request, socketfd);
        return;
    }
    Protocol::setBroadcast(request);
    Datagram *datagram = request->datagram;
    // Data datagram
    if (datagram->getFlags() == 0) {
        handleBroadcastDataMessage(request, socketfd);
        return;
    }
```

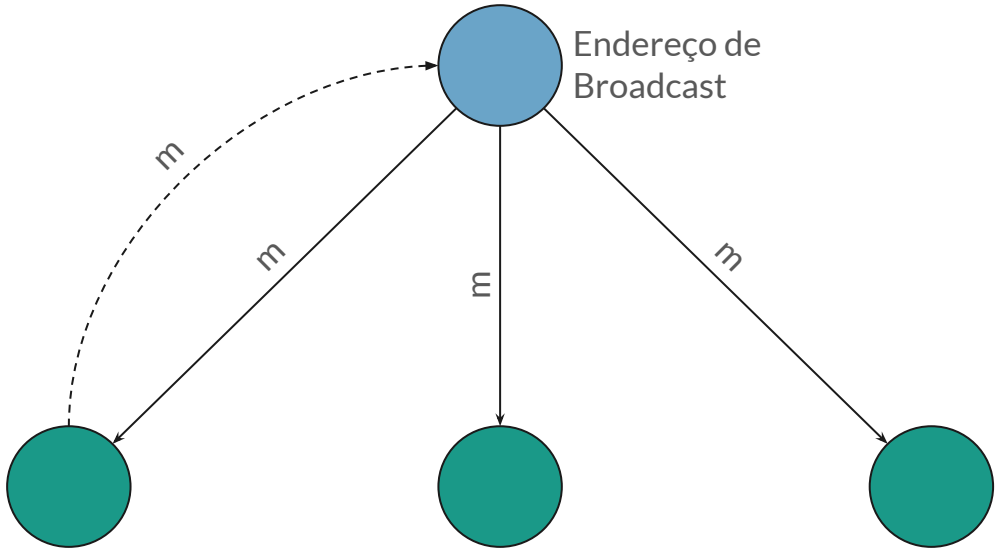
Vendo a implementação

```
}
if ((datagram->isSYN() && datagram->isACK()) || (datagram->isFIN() && datagram->isACK()) || datagram->isACK() ||
    datagram->isNACK() || datagram->isFIN()) {
    std::shared_lock lock([&]messagesMutex);
    Message *message = getMessage(datagram);
    if (message == nullptr)
        return;
```

Vendo a implementação

```
        if ((datagram->isFIN() || datagram->isSYN()) && datagram->isACK()) {
            std::shared_lock messageLock( [&]*message->getMutex());
            std::pair identifier = { x: datagram->getSourceAddress(), y: datagram->getSourcePort()};

            if (!message->acks.contains(identifier) && datagram->isSYN()) {
                message->acks[identifier] = false;
            }
            else if (!message->acks.contains(identifier) && datagram->isFIN()) {
                return;
            }
            else {
                message->acks[identifier] = datagram->isFIN();
                if (!message->delivered && datagram->isFIN()) {
                    deliverBroadcast(message);
                }
            }
        }
        datagramController->insertDatagram( identifier: { x: datagram->getSourceAddress(), y: datagram->getDestinationPort()}, datagram);
        return;
    }
    if (datagram->isSYN()) {
        handleFirstMessage(request, socketfd, broadcast: true);
    }
}
```

Transmissão da mensagem



Vendo a implementação

```
// build of datagrams
auto datagrams = std::vector<std::vector<unsigned char>>(totalDatagrams);
std::map<std::pair<unsigned int, unsigned short>, std::map<unsigned short, std::pair<bool, bool>>> membersAcks;
buildBroadcastDatagrams(&datagrams, &membersAcks, transientSocketFd.second.sin_port, totalDatagrams, [&]message,
                        &members);

unsigned short batchSize = BATCH_SIZE;
const double batchCount = static_cast<int>(ceil( x: static_cast<double>(totalDatagrams) / batchSize));
auto buff = std::vector<unsigned char>( n: 1048);
```

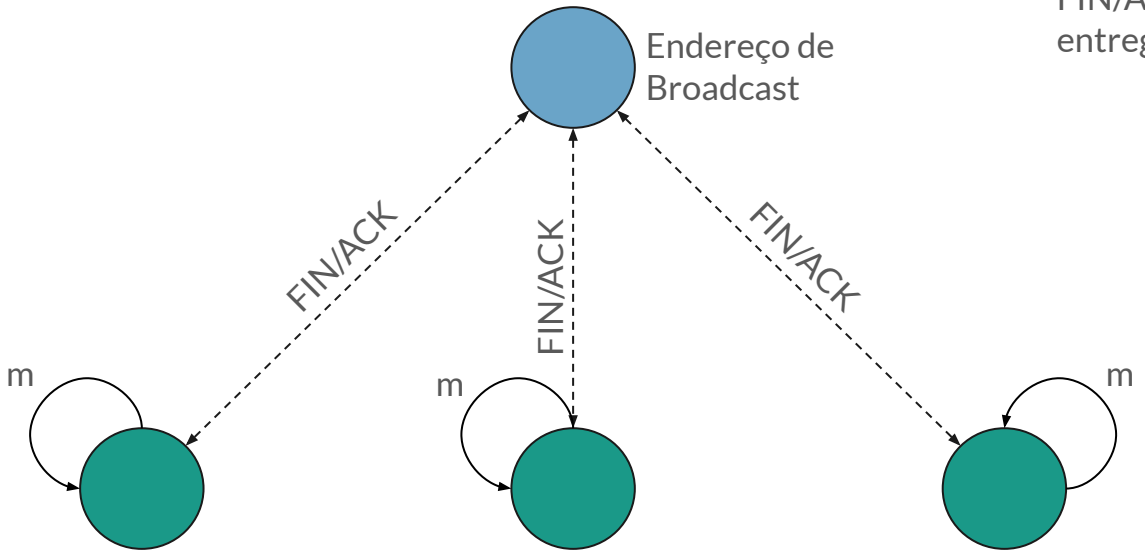
Vendo a implementação

```
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;

    for (int i = 0; i < totalDatagrams; ++i) {
        auto versionDatagram = Datagram();
        versionDatagram.setSourceAddress(configAddr.sin_addr.s_addr);
        versionDatagram.setSourcePort(configAddr.sin_port);
        versionDatagram.setDestinationPort(transientPort);
        versionDatagram.setVersion(i + 1);
        versionDatagram.setDatagramTotal(totalDatagrams);
        for (unsigned short j = 0; j < 1024; j++) {
            const unsigned int index = i * 1024 + j;
            if (index >= message.size())
                break;
            versionDatagram.getData()->push_back( x: message.at(index));
        }
        versionDatagram.setDataLength(versionDatagram.getData()->size());
        auto serializedDatagram :vector<unsigned char> = Protocol::serialize(&versionDatagram);
        Protocol::setChecksum(&serializedDatagram);
        (*datagrams)[i] = serializedDatagram;
        acknowledgments[i] = false;
        responses[i] = false;
        for (const auto &member :const pair<...>& : *members) {
            (*membersAcks)[ {member.first.first, member.first.second}][i]  = { x: false,  y: false};
        }
    }
}
```

Vendo a implementação

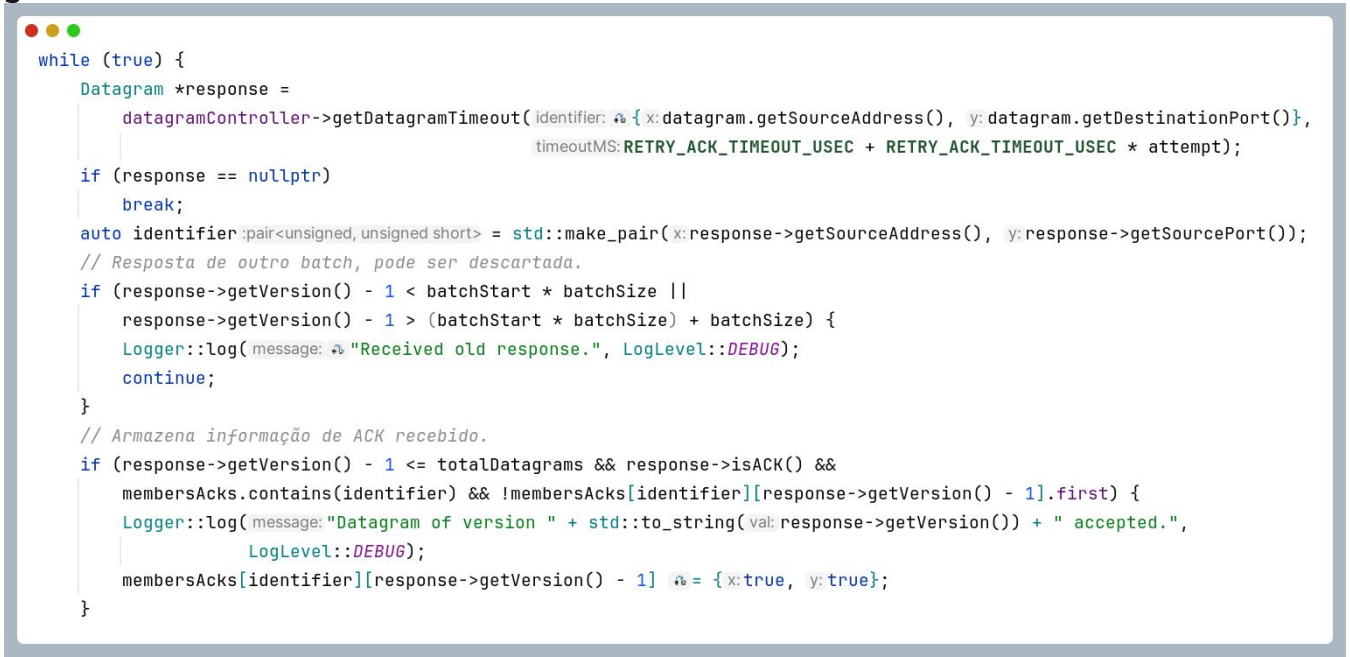
```
for (unsigned short batchStart = 0; batchStart < batchCount; batchStart++) {
    unsigned short batchIndex;
    if (verifyMessageAcked(&members) == totalDatagrams) {
        close(transientSocketFd.first);
        return true;
    }
    for (int attempt = 0; attempt < RETRY_DATA_ATTEMPT; attempt++) {
        if (verifyBatchAcked(&membersAcks, batchSize, batchIndex:batchStart, totalDatagrams)) {
            break;
        }

        for (unsigned short j = 0; j < batchSize; j++) {
            batchIndex = batchStart * batchSize + j;
            if (batchIndex >= totalDatagrams)
                break;
            bool datagramAcked = true;
            for (const auto &configPair : *configMap) {
                const auto &config :const sockaddr_in& = configPair.second;
                if (!membersAcks[ {config.sin_addr.s_addr, config.sin_port}][batchIndex].second) {
                    datagramAcked = false;
                    break;
                }
            }
            if (!datagramAcked)
                sendto(broadcastFD, buf:datagrams[batchIndex].data(), n:datagrams[batchIndex].size(), flags:0,
                       reinterpret_cast<sockaddr *>(&destin), addr_len:sizeof(destin));
```

Todos mandam e recebem ACK's



- Caso for **BEB**, a mensagem pode ser entregue assim que enviar o FIN/ACK
- Caso for **URB**, deve esperar receber a quantidade de FIN/ACK necessários para entregar a mensagem

Vendo a implementação



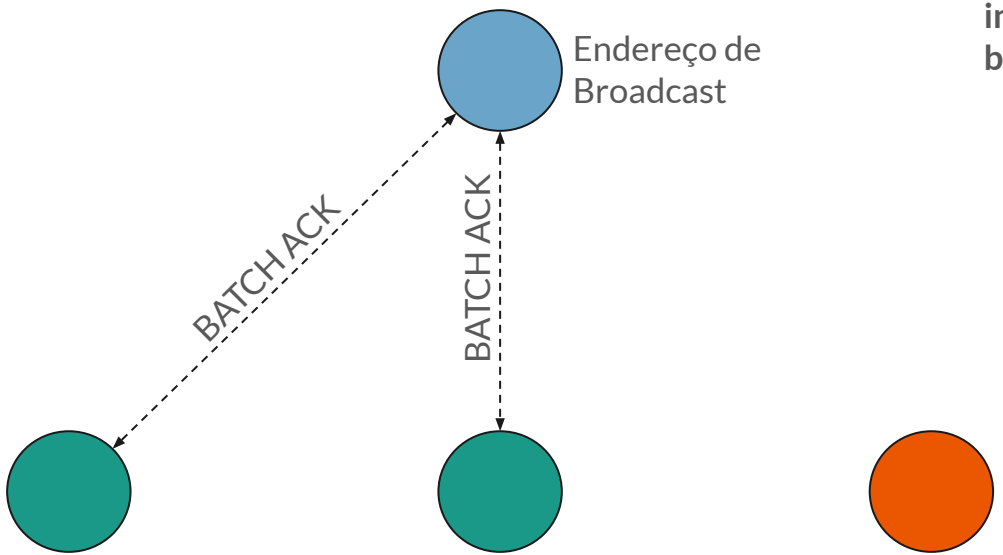
```
while (true) {
    Datagram *response =
        datagramController->getDatagramTimeout( identifier: { x: datagram.getSourceAddress(), y: datagram.getDestinationPort()},
                                                timeoutMS: RETRY_ACK_TIMEOUT_USEC + RETRY_ACK_TIMEOUT_USEC * attempt);
    if (response == nullptr)
        break;
    auto identifier :pair<unsigned, unsigned short> = std::make_pair( x: response->getSourceAddress(), y: response->getSourcePort());
    // Resposta de outro batch, pode ser descartada.
    if (response->getVersion() - 1 < batchStart * batchSize ||
        response->getVersion() - 1 > (batchStart * batchSize) + batchSize) {
        Logger::log( message: "Received old response.", LogLevel::DEBUG);
        continue;
    }
    // Armazena informação de ACK recebido.
    if (response->getVersion() - 1 <= totalDatagrams && response->isACK() &&
        membersAcks.contains(identifier) && !membersAcks[identifier][response->getVersion() - 1].first) {
        Logger::log( message: "Datagram of version " + std::to_string( val: response->getVersion()) + " accepted.",
                    LogLevel::DEBUG);
        membersAcks[identifier][response->getVersion() - 1] = { x: true, y: true};
    }
```

Vendo a implementação

```
            // Conexão finalizada, com ou sem sucesso.
            if (response->isACK() && response->isFIN()) {
                if (members.contains(identifier))
                    members[identifier] = true;
            }
        }
        // Batch acordado, procede para o proximo batch
        if (verifyBatchAcked(&membersAcks, batchSize, batchIndex:batchStart, totalDatagrams)) {
            break;
        }
        // Verifica se o batch foi ao menos respondido, caso tenha sido, mesmo que com algum NACK, procede para
        // retransmissão (se for o caso).
        if (verifyBatchResponded(&membersAcks, batchSize, batchIndex:batchStart, totalDatagrams)) {
            Logger::log( message: "Batch " + std::to_string( val: batchStart + 1) + " responded, but not aknowledged.",
                        LogLevel::DEBUG);
            break;
        }
    }
    // No final de uma tentativa, verifica se o batch foi acordado. Caso não, encerra o fluxo de envio.
    // Caso tenha finalizado de acordar todos os datagramas, finaliza o fluxo.
    if (!verifyBatchAcked(&membersAcks, batchSize, batchIndex:batchStart, totalDatagrams) || verifyMessageAcked(&members)) {
        break;
    }
}
```

Falha em receber ACK's

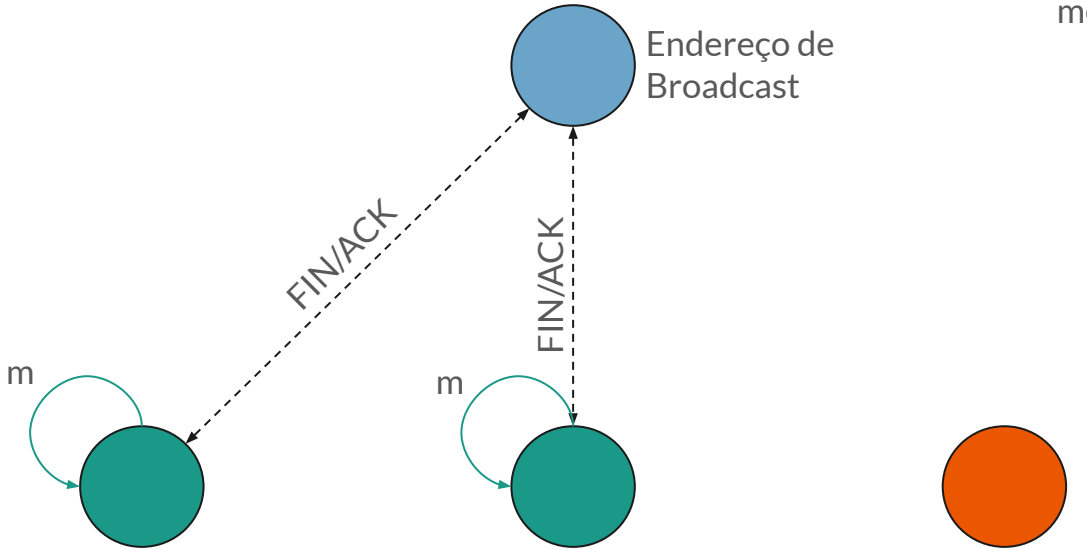
Falha



- Caso um um processo que respondeu **SYN/ACK** durante a fase inicial do broadcast falhe em responder **ACK** para um batch, o envio falha **independente do tipo de broadcast**.

Falha em receber FIN/ACK's em BEB

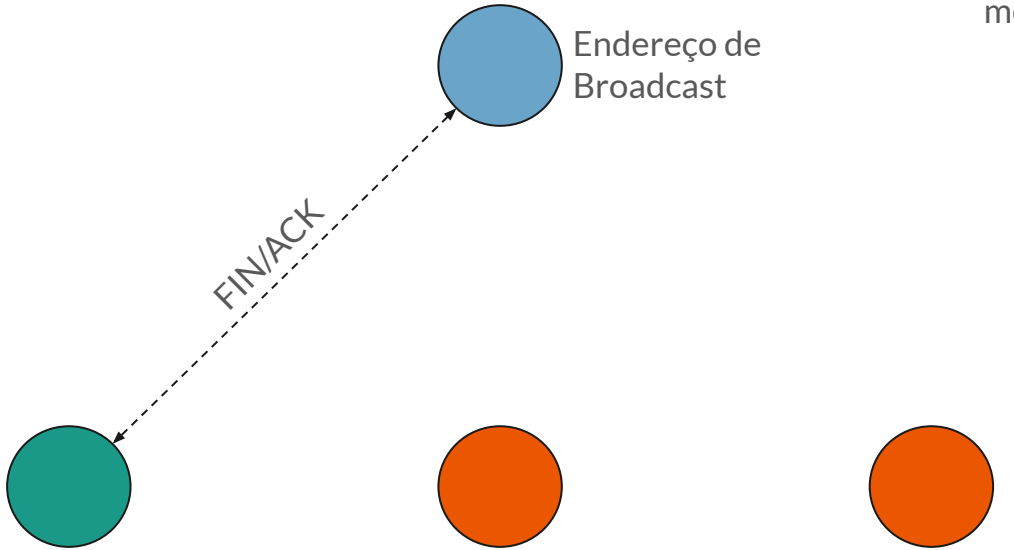
Sucesso



- Em um broadcast do tipo **BEB**, é necessário que **apenas metade** dos processos iniciais responda **FIN/ACK** para informar o recebimento total de uma mensagem.

Falha em receber FIN/ACK's em BEB

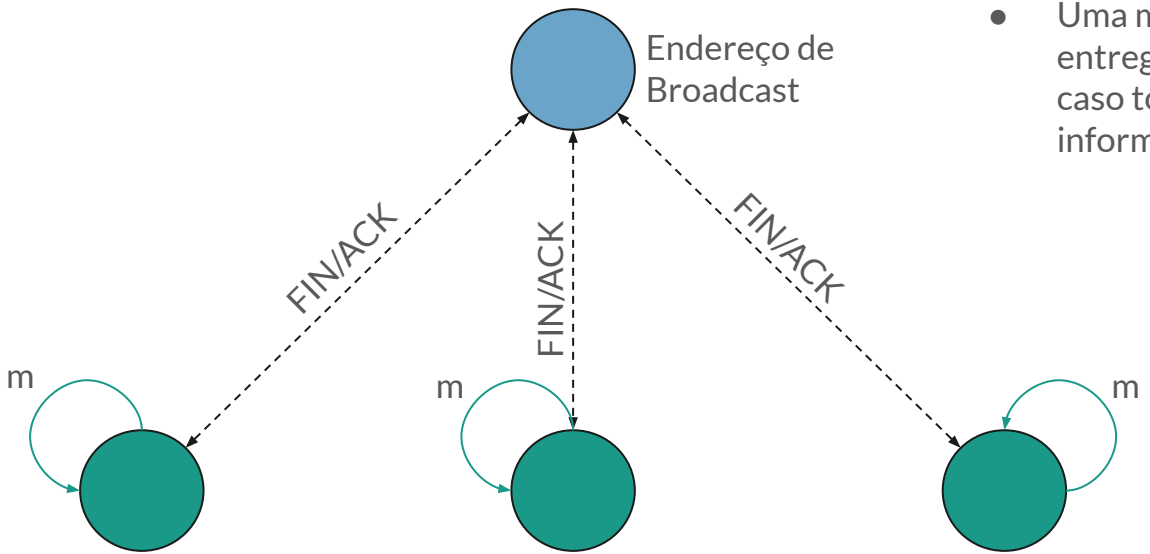
Falha



- Em um broadcast do tipo **BEB**, é necessário que **apenas metade** dos processos iniciais responda **FIN/ACK** para informar o recebimento total de uma mensagem.

Falha em receber FIN/ACK's em URB

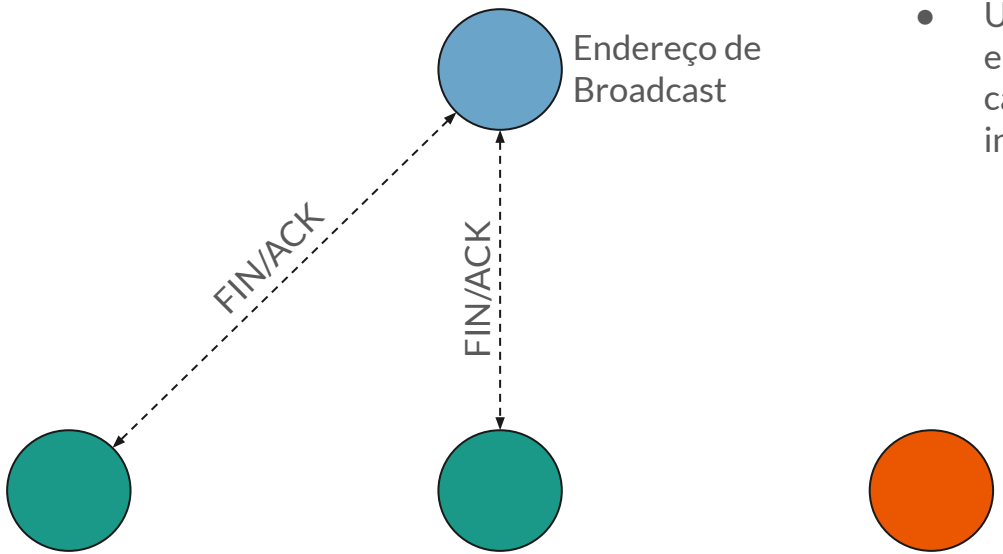
Sucesso



- Em um broadcast do tipo **URB**, é necessário que **todos** os processos iniciais responda **FIN/ACK** para informar o recebimento total de uma mensagem.
- Uma mensagem só será entregue a um processo caso todos os processos informem enviem **FIN/ACK.**

Falha em receber FIN/ACK's em URB

Falha



- Em um broadcast do tipo **URB**, é necessário que **todos** os processos iniciais responda **FIN/ACK** para informar o recebimento total de uma mensagem.
- Uma mensagem só será entregue a um processo caso todos os processos informem enviem **FIN/ACK.**

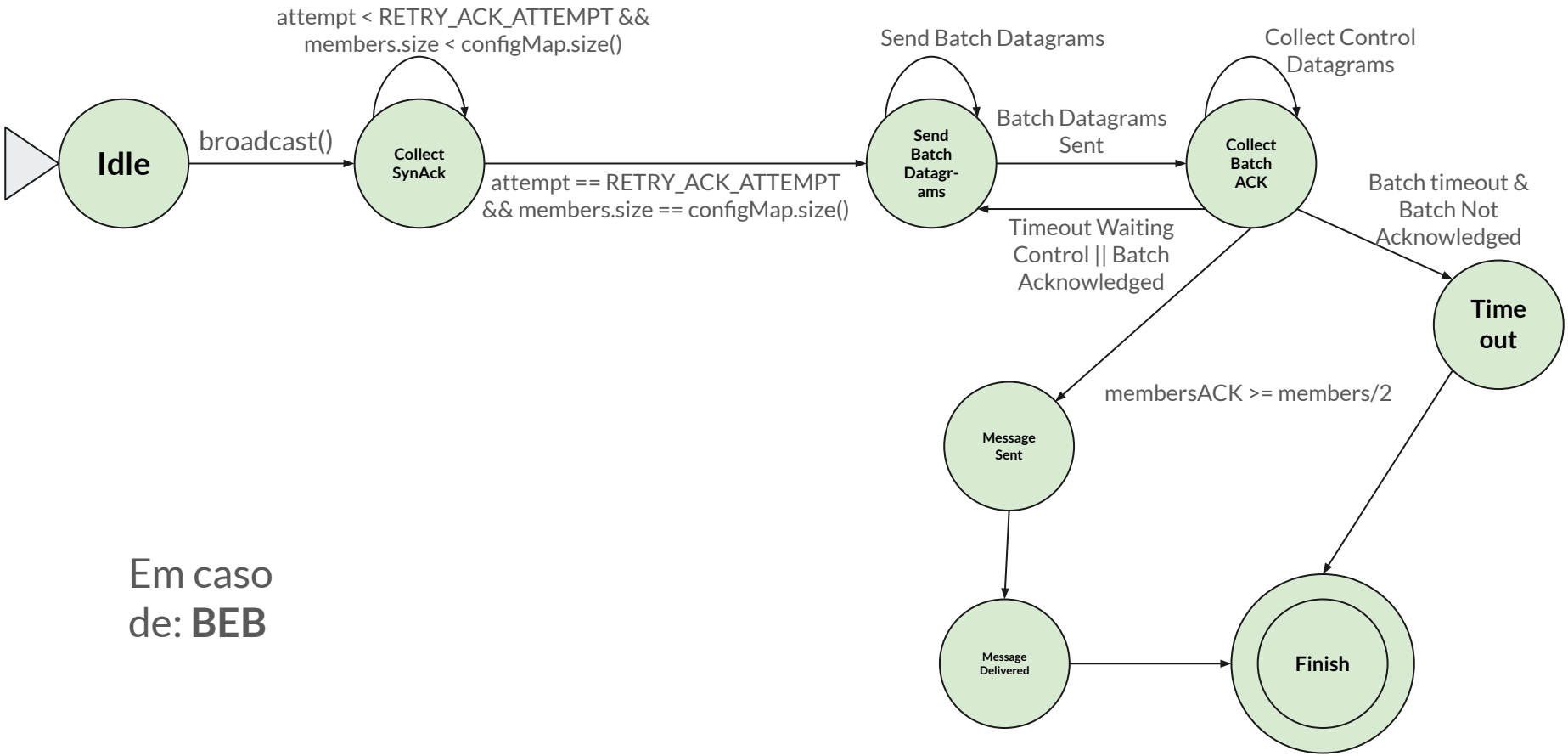
Vendo a implementação

```
return broadcastType == "BEB" ? verifyMessageAckedBEB(&members) : verifyMessageAckedURB(&members);
```

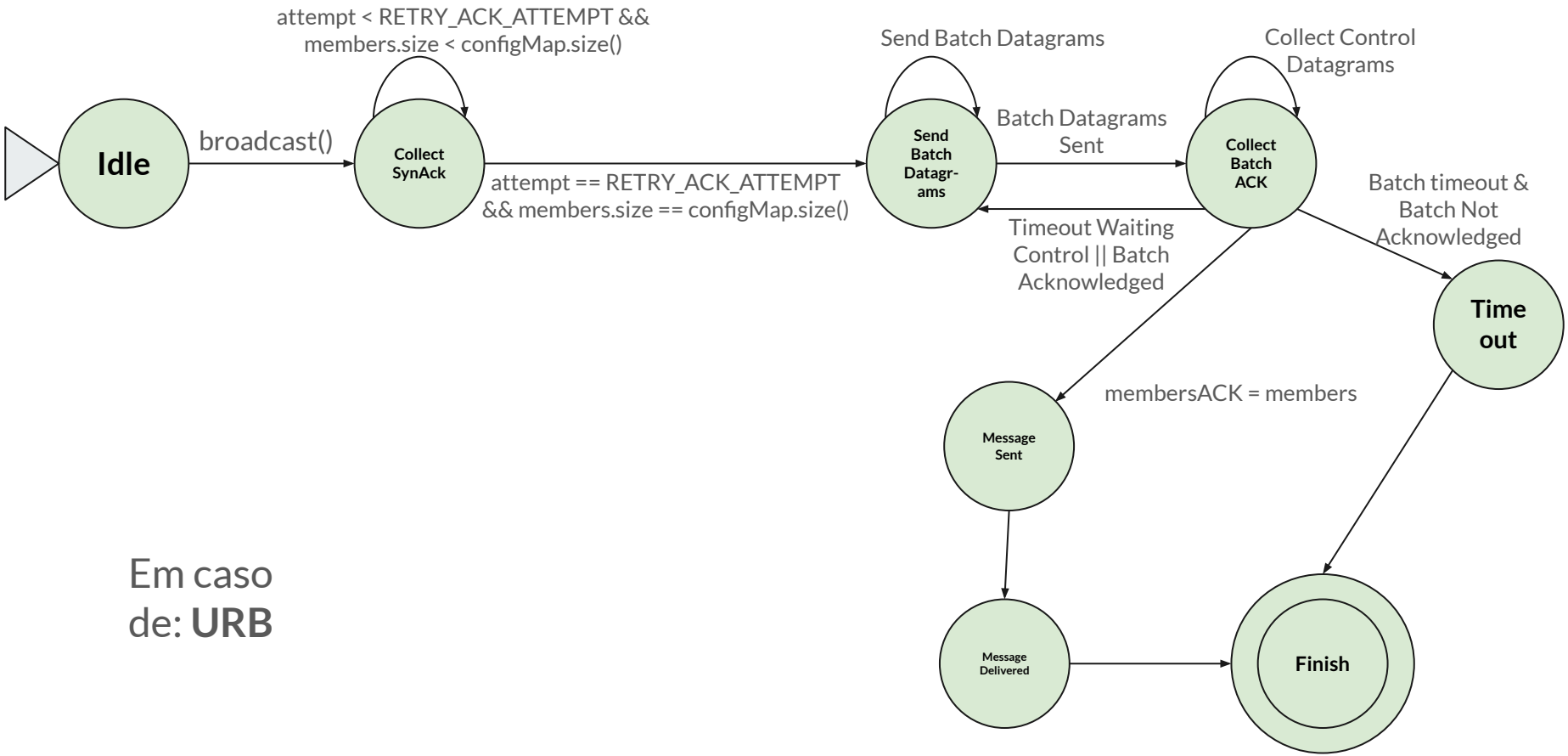
Vendo a implementação

```cpp
bool MessageSender::verifyMessageAckedURB(std::map<std::pair<unsigned int, unsigned short>, bool> *membersAcks) {
    for (auto &&member :pair<const pair<unsigned, unsigned short>, bool>& : *membersAcks) {
        if (!member.second)
            return false;
    }
    return true;
}

bool MessageSender::verifyMessageAckedBEB(std::map<std::pair<unsigned int, unsigned short>, bool> *membersAcks) {
    unsigned short totalSYNACK = 0;
    for (auto &&member :pair<const pair<unsigned, unsigned short>, bool>& : *membersAcks) {
        if (member.second)
            totalSYNACK++;
    }
    return totalSYNACK >= membersAcks->size() / 2;
}
```

Máquina de estados



Em caso de: **BEB**



Em caso de: **URB**

Diagramas de mensagens

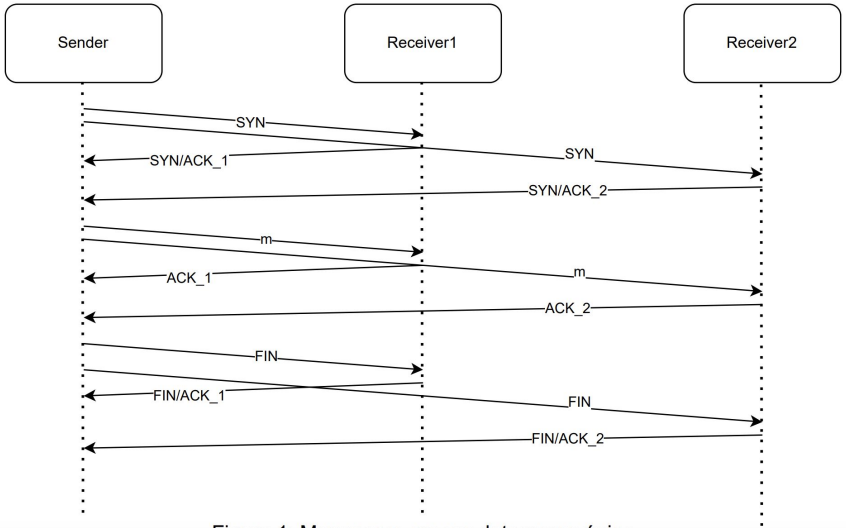


Figura 1: Mensagem em um datagrama único

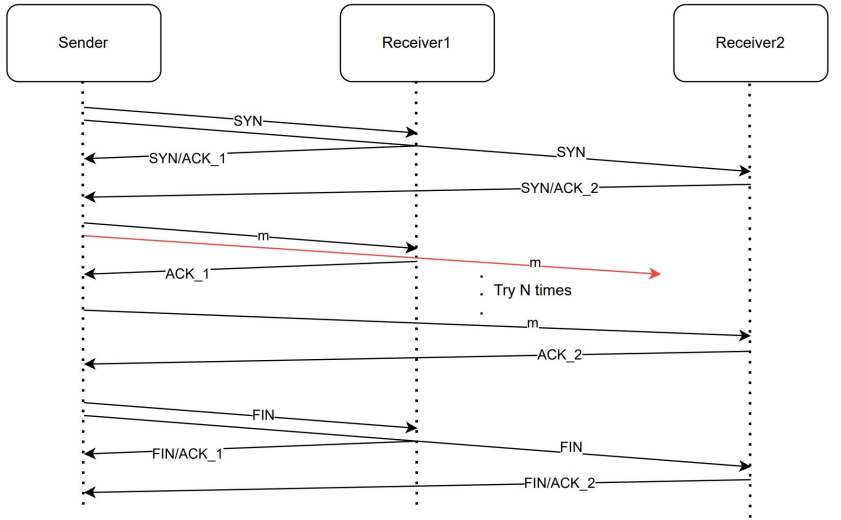


Figura 2: Mensagem em um datagrama único com retransmissão

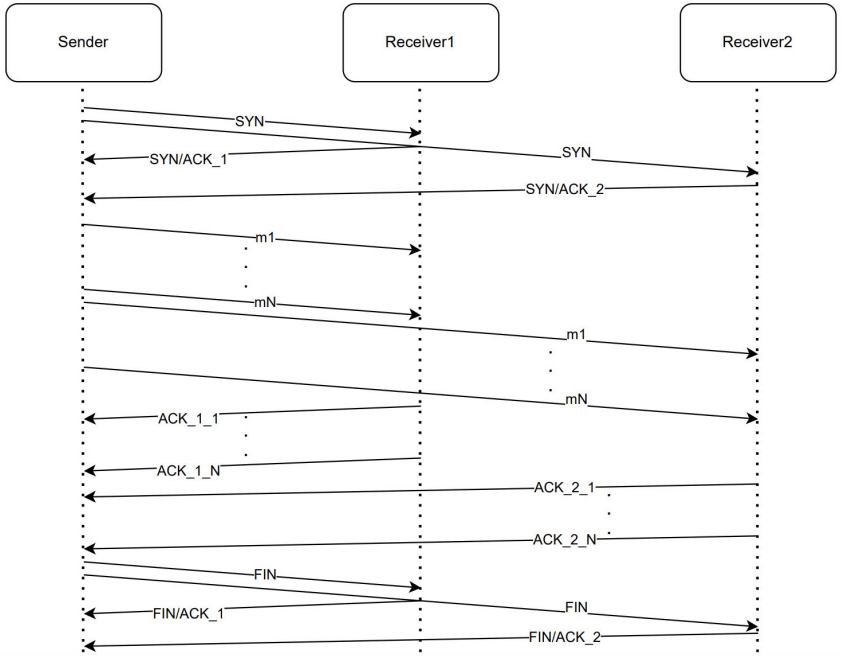


Figura 3: Mensagem com fragmentação

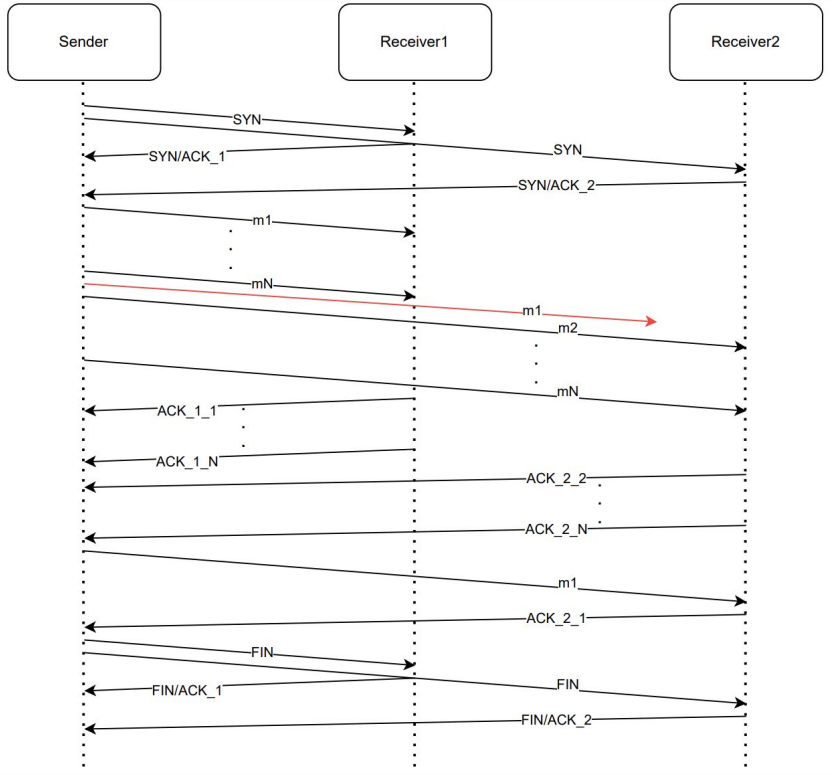


Figura 4: Mensagem com fragmentação e retransmissão

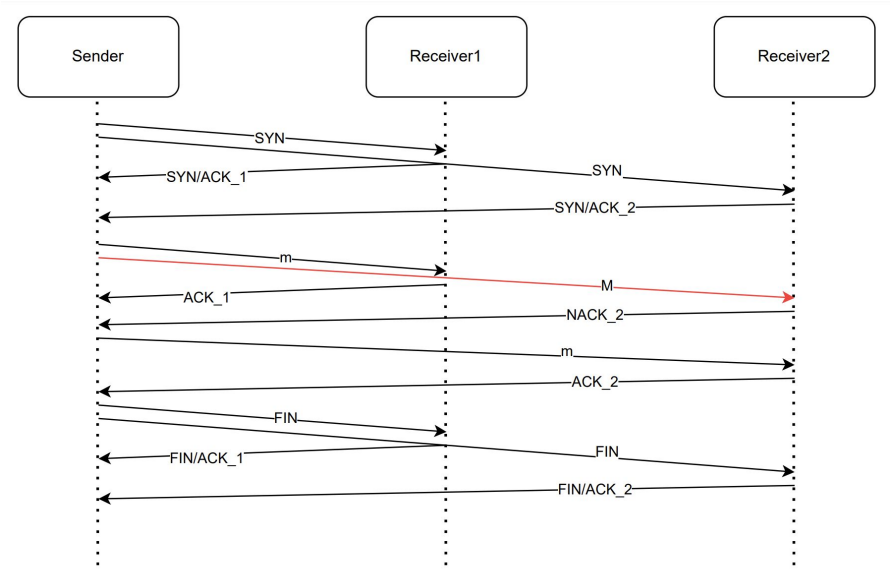


Figura 5: Mensagem em um datagrama único e corrompida

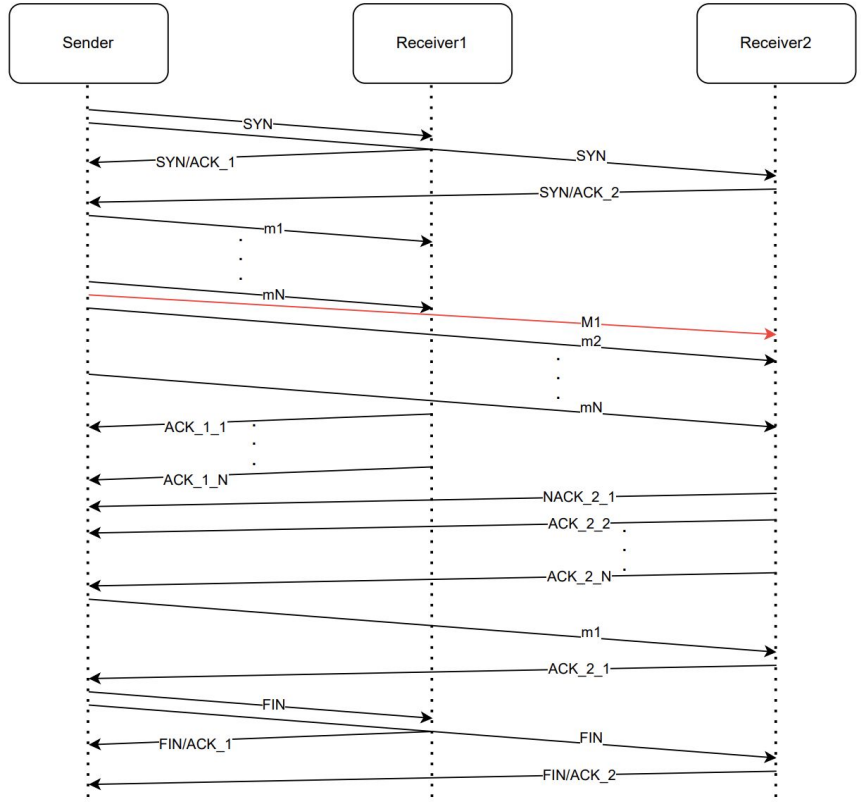


Figura 6: Mensagem com fragmentação e corrompida