Atomic Broadcast

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

Propriedades assumidas

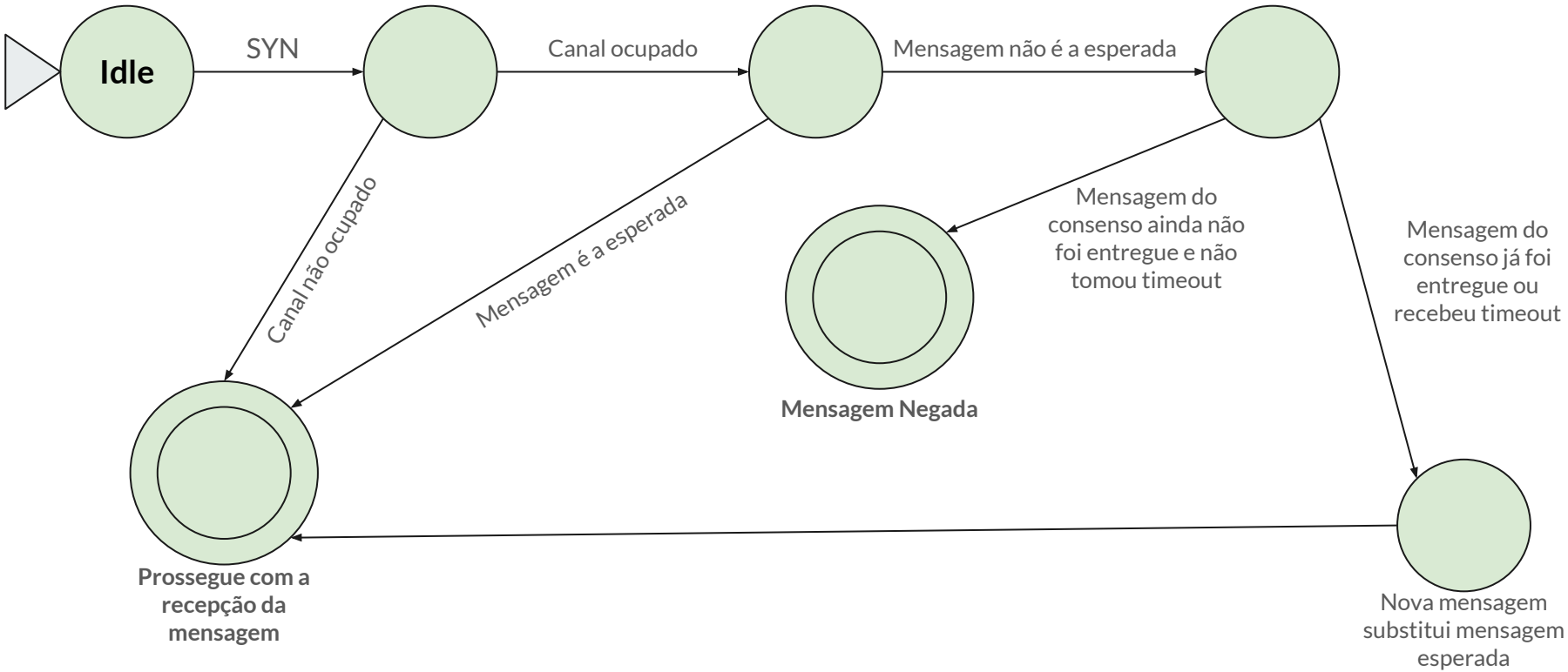
- Se um processo correto entrega, todos os processos corretos entregam (URB);
- Todos os processos ouvem todos as mensagens do canal (broadcast de rede);
- Existe uma tolerância falhas, onde para cada processo falho, deve-se existir ao menos mais que o dobro de processos corretos (2f + 1) para o sucesso de uma mensagem;
- Um processo com permissão para enviar um atomic broadcast possui um limite de tempo para enviar sua mensagem (3 segundos).

Ordem Total

- Para garantir a ordem total, o processo origem deve solicitar permissão para transmissão;
- O fluxo de permissão é feito de maneira passiva por parte dos destinatários, onde ao não concordar com a mensagem, apenas não respondem a solicitação;
- É utilizado um algoritmo de consenso para definição de qual processo possui permissão para enviar um *atomic broadcast*.

Consenso nos destinatários

```
// Não possui nenhuma mensagem esperada
if (broadcastType == AB) {
    // Cria uma mensagem mesmo que não vá ser utilizada para posterior consenso
    if (datagram->getFlags() == 2) {
        createMessage(request, socketfd);
    }
    // Caso haja alguma mensagem esperada
    if (channelOccupied) {
        std::pair messageID = {x: request->datagram->getDestinAddress(), y: request->datagram->getDestinationPort()};
        // Se a mensagem recebida for diferente da esperada
        if (channelMessageIP != messageID.first || channelMessagePort != messageID.second) {
            std::pair<unsigned int, unsigned short> channelMessageID = {[&]channelMessageIP, [&]channelMessagePort};
            std::pair consent = verifyConsensus();
            // Consenso espera alguma coisa
            if ((consent.first != 0 || consent.second != 0) &&
                (consent.first != channelMessageID.first || consent.second != channelMessageID.second)) {
                if (messages.contains(consent)) {
                    std::shared_lock messageLock([&]*messages[consent]->getMutex());
                    // Mensagem anterior não foi finalizada e recebeu um timeout, logo substitui a mensagem aceita.
                    if (!messages[consent]->delivered &&
                        messages[consent]->getLastUpdate() - std::chrono::system_clock::now() >
                            std::chrono::seconds(rep: 3)) {
                        channelMessageIP = consent.first;
                        channelMessagePort = consent.second;
                    }
                    else {
                        // Mensagem anterior não foi finalizada, porém ainda não tomou timeout
                        return;
                    }
                }
            }
        }
    }
}
```


Idle
SYN
Canal ocupado
Mensagem não é a esperada
Canal não ocupado
Mensagem é a esperada
Mensagem do consenso ainda não foi entregue e não tomou timeout
Mensagem do consenso já foi entregue ou recebeu timeout
Mensagem Negada
Prossegue com a recepção da mensagem
Nova mensagem substitui mensagem esperada

Heartbeat para consenso

- A informação de qual mensagem será aceita (ou seja, qual mensagem tem a vez no canal) é armazenada em cada nó;
- Periodicamente um sinal de *heartbeat* é enviado ao canal atualizando os membros se o mesmo espera uma mensagem, caso sim, identifica esta mensagem;
- Todos os nós do canal mantém a informação de acordo de cada nó;
- A mensagem com mais nós concordando com a mesma é acatada pelos demais nós.

Desempate no consenso

- Em casos de quantidade par de nós, metade do canal pode concordar com uma mensagem, e a outra metade concordar com outra;
- Para estes casos, contabiliza-se um critério de desempate, onde caso um critério falhe em desempatar, o próximo será utilizado:
    - Menor **ip** de origem;
    - Menor **porta** de origem;
    - Menor **identificador de mensagem** de origem.

Desempate no consenso

```
std::pair<unsigned int, unsigned short> MessageReceiver::verifyConsensus() {
    std::map<std::pair<in_addr_t, in_port_t>, int> countMap;

    for (const auto &entry :const pair<...>& : heartbeats) {
        countMap[entry.second]++;
    }

    std::vector<std::pair<std::pair<in_addr_t, in_port_t>, int>> sortedCounts( first:countMap.begin(), last:countMap.end());
```

Desempate no consenso

```
    std::sort( first: sortedCounts.begin(),  last: sortedCounts.end(),  comp: [](const auto &a, const auto &b) {
        if (a.second != b.second) {
            return a.second > b.second;
        }
        if (a.first.first != b.first.first) {
            return a.first.first < b.first.first;
        }
        return a.first.second < b.first.second;
    });
    if (!sortedCounts.empty()) {
        const auto &result :const pair<unsigned, unsigned short>& = sortedCounts.front().first;
        return {result.first, result.second};
    }

    return { x: 0,  y: 0};
}
```

Atualização dos *heartbeats*

- Uma *thread* envia um sinal de *heartbeat* a cada 1 segundo informando os demais nós qual mensagem está sendo acordada no momento;
- Adicionalmente, sempre que a mensagem acordada for modificada (a mensagem foi entregue ou recebeu *timeout*), o nó atualiza este estado através de um *heartbeat*.

Atualização dos *heartbeats*

```
void MessageReceiver::heartbeat() {
    while (running) {
        {
            sendHEARTBEAT( target: { [&]channelMessageIP, [&]channelMessagePort}, broadcastFD);
            std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 1));
            std::vector<std::pair<unsigned int, unsigned short>> removes;
            for (auto [identifier :const pair<unsigned, unsigned short>, time :time_point<system_clock>] : heartbeatsTimes) {
                if (std::chrono::system_clock::now() - time >= std::chrono::seconds( rep: 3)) {
                    removes.emplace_back(identifier);
                }
            }
            for (auto identifier :pair<unsigned int, unsigned short> : removes) {
                if (heartbeats.count(identifier))
                    heartbeats.erase(identifier);
                if (heartbeatsTimes.count(identifier))
                    heartbeatsTimes.erase(identifier);
            }
        }
    }
}
```

Atualização dos *heartbeats*

```
void MessageReceiver::handleBroadcastMessage(Request *request, int socketfd) {
    //...//
    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();
        return;
    }
```

Consenso no remetente

Consenso no remetente

- Resumidamente, a permissão para envio de mensagem é adquirida no remetente ao conseguir o SYN+ACK de mais da metade do grupo.

Consenso no remetente

```
bool 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) {
        // Consensus
        if (broadcastType == AB && members->size() > configMap->size() / 2) {
            return true;
        }
        //..//
```

Conclusão

- O consenso sobre a ordem total das mensagens foi atingido permitindo que apenas um membro do canal envie um broadcast por vez;
- Portanto, o consenso sobre a ordem foi transformado em consenso sobre quem deve conseguir mandar a mensagem, que foi resolvido como descrito na apresentação atual.