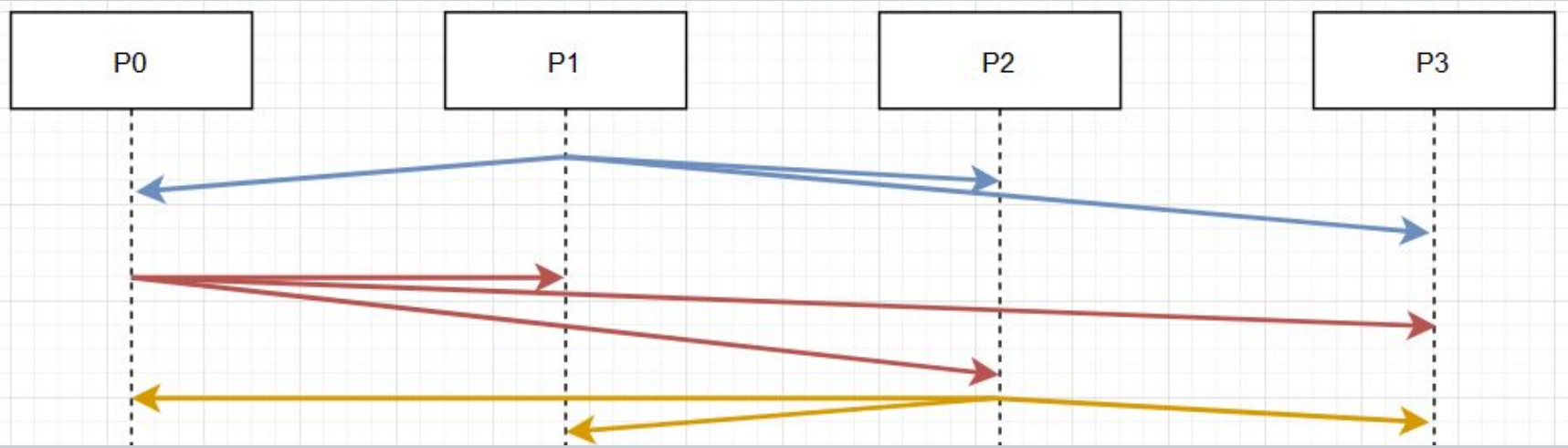
P5

INE5424-06208B (20242) – Sistemas Operacionais II | Curso de Ciências da Computação

Alexis Mendes Sequeira (16100717)
Luis Henrique Goulart Stemmer (18105165)
Luiz Maurício do Valle Pereira (21104157)
Rafael Neves de Mello Oliveira (17102816)

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Troca de heartbeats



- 0, 1 e 2 enviam seus heartbeats para todos os outros processos durante uma janela de tempo
- Cada heartbeat traz consigo a informação de quais processos o emissor sabe que estão vivos
- A verificação é realizada após a janela determinada, iniciando um novo round logo depois

Funcionamento

P1

```
[INFO] Detecting defective processes
Node 0: Uninitialized
Node 1: Active
Node 2: Active
Node 3: Active
Node 4: Uninitialized
```

P2

```
[INFO] Detecting defective processes
Node 0: Uninitialized
Node 1: Active
Node 2: Active
Node 3: Active
Node 4: Uninitialized
```

P3

```
[INFO] Detecting defective processes
Node 0: Uninitialized
Node 1: Active
Node 2: Active
Node 3: Active
Node 4: Uninitialized
```

- Os processos em destaque são capazes de se detectar com sucesso, assim como detectar processos ainda não inicializados (P0 e P4) com sucesso

Adições - Novo tipo de mensagem

```
Failure detection
   HTB - Heartbeat
   HSY - Heartbeat sync states of processes - NOT USED
```

Adições - atributos e threads

```
// Witness of nodes (map of process ID to set of process IDs that mark it as alive)
std::map<int, std::set<uint8_t>> witness_of_nodes;

// Participant states (map of process ID to state)
std::map<int, ParticipantState> participant_states;

// Buffers to store messages for uninitialized processes
std::map<int, std::queue<Message>> message_buffers;
```

```
// Start the heartbeat and defect detection threads
std::thread heartbeat_thread(&AtomicBroadcastRing::send_heartbeat, this);
heartbeat_thread.detach();

std::thread defect_detection_thread(&AtomicBroadcastRing::detect_defective_processes, this);
defect_detection_thread.detach();
std::thread htb_handler(&AtomicBroadcastRing::htb_handler_thread, this);
htb_handler.detach();
```

Enviando Heartbeats

```cpp
// Heartbeat message sending logic
void AtomicBroadcastRing::send_heartbeat() {
    while (true) {
        std::vector<uint8_t> heartbeat_msg;
        for (const auto& entry : participant_states) {
            heartbeat_msg.push_back(static_cast<uint8_t>(entry.second));  // Serialize all participants states
        }

        // Send heartbeat to all participants (except self)
        for (const auto& node : nodes) {
            if (node.first != process_id) {
                //log("Sending heartbeat to node " + std::to_string(node.first), "INFO");
                channels->send_message(node.first, process_id, Message(process_address, msg_num, "HTB", heartbeat_msg));
            }
        }

        // Sleep for the heartbeat interval
        std::this_thread::sleep_for(std::chrono::milliseconds(heartbeat_interval));
    }
}
```

Recebendo Heartbeats

```cpp
// Process received heartbeat messages
void AtomicBroadcastRing::process_heartbeat(const Message& msg) {
    //log("Processing heartbeat message", "INFO");

    int key = find_key(msg.sender_address);

    // If the sender is uninitialized, mark it as active
    if (participant_states[key] == ParticipantState::Uninitialized) {
        participant_states[key] = ParticipantState::Active;

        witness_of_nodes[key].insert(process_id);   // Insert this process as witness of sender
        // For every node sender knows is alive, mark it as a witness
        for (int i=0; i<static_cast<int>(msg.content.size()); i++) {
            if (msg.content[i] == 1) {
                witness_of_nodes[i].insert(key);
            }
        }
        // Answer with a heartbeat of our own witnesses to sender
        std::vector<uint8_t> heartbeat_msg;
        for (const auto& entry : participant_states) {
            if (entry.first != process_id && entry.second == ParticipantState::Active) {
                heartbeat_msg.push_back(static_cast<uint8_t>(entry.second));  // Serialize all participants states
            }
        }
        channels->send_message(key, process_id, Message(process_address, msg_num, "HTB", heartbeat_msg));
    }
```

Detectando - verificando se há quorum

```
void AtomicBroadcastRing::detect_defective_processes() {
    send_htb = true;
    while (true) {
        // two steps for a round:
        // 1- collect HTBs for alloted time
        // 2- Make final decision based on the collected information

        // Sleep before checking again
        std::this_thread::sleep_for(std::chrono::milliseconds(5*heartbeat_interval));
        log("Detecting defective processes", "INFO");

        // Check amount alive
        int amount_alive = 0;
        for (auto&entry :participant_states) {
            if (entry.second == ParticipantState::Active) {
                amount_alive++;
            }
        }
```

Detectando - Se não houver quorum

```
if (amount_alive <= (2*failures)) {
    //log("Not enough nodes alive alive, no need to continue", "INFO");
    print_states();
    for (const auto& node : nodes) {
        participant_states[node.first] = ParticipantState::Uninitialized;
    }
    participant_states[process_id] = ParticipantState::Active;
    // initialize other nodes responses to suspect everyone
    for (const auto& node : nodes) {
        witness_of_nodes[node.first].clear();
    }
    // put process as a witness of itself
    witness_of_nodes[process_id].insert(process_id);
    continue;
```

Detecção - se tivermos quorum

```
    for (auto& entry:witness_of_nodes) {
        if (static_cast<int>(entry.second.size()) < (2*failures)+1) {
            //log("Node " + std::to_string(entry.first) + " is down", "INFO");
            participant_states[entry.first] = ParticipantState::Uninitialized;
        } else {
            participant_states[entry.first] = ParticipantState::Active;
        }
    }
}

// reset witness of nodes for next round
print_states();
for (const auto& node : nodes) {
    participant_states[node.first] = ParticipantState::Uninitialized;
}
participant_states[process_id] = ParticipantState::Active;
// initialize other nodes responses to suspect everyone
for (const auto& node : nodes) {
    witness_of_nodes[node.first].clear();
}
witness_of_nodes[process_id].insert(process_id);
```

Starts new round after this, sleeping for a set amount

Envio de mensagens transmitidas antes da entrada no grupo

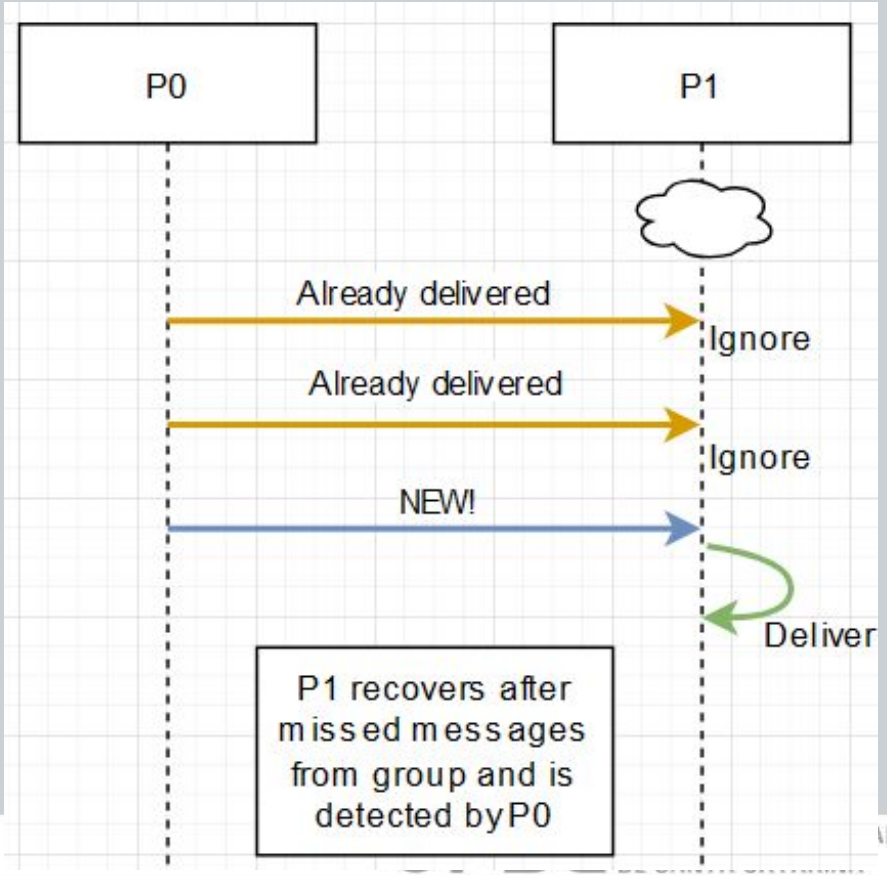
- Infelizmente não foi feita a implementação completa deste requisito

```cpp
// Buffer messages for uninitialized nodes
void AtomicBroadcastRing::buffer_message_for_uninitialized(const Message& msg) {
    if (participant_states[find_key(msg.sender_address)] == ParticipantState::Uninitialized) {
        message_buffers[find_key(msg.sender_address)].push(msg);
    }
}

// Deliver buffered messages when the node becomes active
void AtomicBroadcastRing::deliver_buffered_messages(int node_id) {
    if (participant_states[node_id] == ParticipantState::Active) {
        while (!message_buffers[node_id].empty()) {
            Message msg = message_buffers[node_id].front();
            message_buffers[node_id].pop();
            deliver_queue.push(msg);  // Deliver message to the application
        }
```

Estados de um processo - Decisão de projeto

- Somente dois estados: **Ativo e não inicializado**
- Facilita a entrega de mensagens a processos que falharam temporariamente
- Estes são tratados como novos ao grupo e recebem todas as mensagens até então
- Caso haja envio de mensagens já entregues, basta ignorar



Equipe

Alexis Mendes Sequeira (16100717)

Rafael Neves de Mello Oliveira (17102816)

Luiz Maurício do Valle Pereira (21104157)

Luis Henrique Goulart Stemmer (18105165)