Relatório - The One

Matheus Gonzaga Muniz[1] e Victor Augusto Andrade Silva[1]

[1] Instituto de Computação – Universidade Federal do Amazonas (UFAM)

{mgm2,vaas}@icomp.ufam.edu.br

Resumo. *Este relatório apresenta detalhes da solução para a Questão 7 proposta na atividade envolvendo o simulador The One.*

1. Especificação da questão - Comunicação Multicast

Nesta atividade, você vai precisar implementar uma comunicação multicast, isto é, quando uma mensagem é criada por um nó A com múltiplos destinatários B, C e D, por exemplo.

Por padrão, o The ONE implementa a comunicação unicast, com uma origem e um destino. Implemente uma classe de comunicação multicast, chamada *MessageMulticastEventGenerator*. Defina a parametrização utilizando seu arquivo de configuração, tal que contenha:

```
EventsX.class = MessageMulticastEventGenerator
EventsX.interval = 0,60
EventsX.size = 1M,5M
EventsX.hosts = 0,100
EventsX.tohosts = 0,25
EventsX.prefix = Message
EventsX.numdestinos = 10
```

O número de destinatários deve ser escolhido aleatoriamente ou definido via arquivo de configuração pela variável *numdestinos*. Os destinos devem ser definidos pela variável opcional *tohosts*.

É necessário que durante o roteamento, os nós da rede não arquivem e apaguem do buffer de envio as mensagens, caso ele seja um dos destinatários da mensagem. Assim, se um nó A envia uma mensagem com destinatários B, C e D e a mensagem chega ao nó B, então B deve considerar a mensagem como entregue e ao mesmo tempo continuar carregando a mensagem consigo para repassá-la aos nós C e D também.

Teste seu modelo de comunicação utilizando o algoritmo de roteamento epidêmico.

2. Modificações realizadas

As alterações foram realizadas em 3 arquivos pertencentes ao simulador. E são:

2.1 Classe *Message*

Tal classe que foi a base para a realização da comunicação multicast. Ela foi modificada em alguns dos seus atributos e métodos (sobrecarga). Conta com as seguintes modificações:

Criação do atributo:

```
private ArrayList<DTNHost> receivArrayList;
```

Que representa a lista de destinatários, ao invés de um destinatário único (unicast).

Um novo método construtor:

```
public Message(DTNHost from, DTNHost to, String id, int size, ArrayList<DTNHost> receivArrayList) {
    this.from = from;
    this.to = to;
    this.id = id;
    this.size = size;
    this.receivArrayList = new ArrayList<>();

    this.path = new ArrayList<DTNHost>();
    this.uniqueId = nextUniqueId;

    this.timeCreated = SimClock.getTime();
    this.timeReceived = this.timeCreated;
    this.initTtl = INFINITE_TTL;
    this.responseSize = 0;
    this.requestMsg = null;
    this.properties = null;
    this.appID = null;

    Message.nextUniqueId++;
    addNodeOnPath(from);
}
```

Além de métodos get/set para o novo atributo.

```
public ArrayList<DTNHost> getReceivArrayList(){
    try{
        return this.receivArrayList;
    } catch(Exception e){
        System.out.println(e.toString());
        return null;
    }

}

public void setReceivArrayList(ArrayList<DTNHost> receivArrayList){
    for(int i=0; i<receivArrayList.size();i++){
        this.receivArrayList.add(receivArrayList.get(i));
    }
}
```

2.2 Classe *MessageCreateEvent*

Essa classe é responsável pela criação das mensagens, e teve sua modificação no método *processEvent*, de modo a dar suporte ao multicast.

```
public void processEvent(World world) {
    if (isMultiCast) {
        DTNHost to = world.getNodeByAddress(this.toAddr);
        DTNHost from = world.getNodeByAddress(this.fromAddr);

        Settings settings = new Settings("Events1");

        Random random = new Random();
        int numAleatorio = random.nextInt(100);
        int numdestinos = 0;
        ArrayList<DTNHost> destinations = new ArrayList<>();
        int[] hosts = { 0, 25 };

        // acesso a variavel de numdestinos no arquivo
        if (settings.contains("numdestinos")) {
            numdestinos = settings.getInt("numdestinos", numAleatorio);
        }

        if (settings.contains("hosts")) {
            hosts = settings.getCsvInts("hosts");
        }

        // geração dos destinos aleatórios
        for (int i = 0; i < numdestinos; i++) {
            int destination = random.nextInt(hosts[1]);
            DTNHost destinationHost = world.getNodeByAddress(destination);
            destinations.add(destinationHost);
        }

        Message m = new Message(from, to, this.id, this.size, destinations);
        m.setReceivArrayList(destinations);
        from.createNewMessage(m);

    } else {
        DTNHost to = world.getNodeByAddress(this.toAddr);
        DTNHost from = world.getNodeByAddress(this.fromAddr);

        Message m = new Message(from, to, this.id, this.size);
        m.setResponseSize(this.responseSize);
        from.createNewMessage(m);
    }
}
```

Nela temos acesso aos dados do arquivo e configuração, temos acesso à quantidade de destinatários e os definimos de maneira aleatória, caso necessário. O procedimento de criação de mensagem segue normalmente após isso.

2.3 Classe *MessageRouter*

Essa classe teve seu método *messageTransferred* (que é chamado após uma mensagem ser tranferida com sucesso), modificado de modo a dar suporte tanto ao unicast, quanto ao multicast. Nele é acessado cada um dos destinatários presentes no ArrayList.

```
if (isMulticast) {

    ArrayList<DTNHost> recivers = aMessage.getReceivArrayList();

    for (int i = 0; i < recivers.size(); i++) {
        if (recivers.get(i) == this.host) {

            isFinalRecipient = true;
            isFirstDelivery = isFinalRecipient && !isDeliveredMessage(aMessage);

            if (!isFinalRecipient && outgoing!=null) {
                // adiciona a mensagem ao buffer
                addToMessages(aMessage, false);
            } else if (isFirstDelivery) {
                this.deliveredMessages.put(id, aMessage);
            } else if (outgoing == null) {

                this.blacklistedMessages.put(id, null);
            }

            for (MessageListener ml : this.mListeners) {
                ml.messageTransferred(aMessage, from, this.host,
                        isFirstDelivery);
            }
            break;
        }
    }
} else{
    isFinalRecipient = aMessage.getTo() == this.host;
    isFirstDelivery = isFinalRecipient &&
    !isDeliveredMessage(aMessage);
    if (!isFinalRecipient && outgoing!=null) {
        // not the final recipient and app doesn't want to drop the message
        // -> put to buffer
        addToMessages(aMessage, false);
    } else if (isFirstDelivery) {
        this.deliveredMessages.put(id, aMessage);
    } else if (outgoing == null) {
        // Blacklist messages that an app wants to drop.
        // Otherwise the peer will just try to send it back again.
        this.blacklistedMessages.put(id, null);
    }


    for (MessageListener ml : this.mListeners) {
        ml.messageTransferred(aMessage, from, this.host,
                isFirstDelivery);
    }
}
```