Universidade Federal do Amazonas
Instituto de Computação
Simulação de Sistemas

DIEGO PINHEIRO - 20901612
FÁBIO LIMA - 20901556
GERSON CORRÊA - 20710307
MANDERSON CRUZ - 20901555
RENATO PACHECO - 20902172

Laboratório de Simulação de Sistemas
Políticas de Gerência de Buffer e Modelo de Mobilidade em redes DTN

Manaus
2012

1 - Introdução

Redes Tolerantes a Atraso e Desconexões

São um conjunto de redes que possuem em comum a dificuldade de manter uma comunicação fim-a-fim com baixa latência e pequena perda de pacotes. As principais características encontradas nas Redes Tolerantes a Atraso e Desconexões são:

Atrasos variáveis:

Uma Rede Tolerante a Atraso pode chegar a ter atrasos de horas ou mesmo dias. Esse atraso é formado basicamente por quatro componentes: tempo de espera, atraso nas filas, atraso de transmissão e atraso de propagação. A primeira componente diz respeito ao tempo de espera de cada nó pelo nó de destino ou pela chegada de um nó intermediário que possa encaminhar as suas mensagens. O atraso nas filas corresponde aos atrasos variáveis que ocorrem nas filas dos nós antes de uma mensagem corrente ser entregue. Por fim, tem-se o atraso de transmissão de mensagem e o atraso que corresponde ao tempo de propagação do sinal a cada contato entre dois nós.

Frequentes desconexões:

Inexistência de um caminho fim a fim entre dois nós dentro da rede, caracterizando uma conectividade intermitente. A causa dessas desconexões podem ocorrer devido a mobilidade provocada pelas constantes mudanças na topologia da rede, por condições de comunicação de baixa ou péssima qualidade, devido a economia de recursos como por exemplo em sensores sem fio onde sensores dormem para poupar energia.

A arquitetura

Devido as características citadas acima, o IRTF propôs uma arquitetura para redes DTN, o surgimento dessa arquitetura teve inicio no projeto Internet Interplanetária (IPN). O objetivo do projeto era estabelecer uma comunicação entre alguns pontos da terra com pontos espaciais.

A arquitetura que foi proposta faz parte da RFC 4838 e tem como princípio básico a comutação de mensagens e armazenamento persistente dos dados, ou seja, quando uma mensagem precisa ser enviada ela primeiramente será armazenada em um nó remetente para que possa ser repassada para outro nó que pode ou não ser o nó destino.

Para que seja possível o armazenamento persistente de dados, uma vez que o tempo de armazenamento pode chegar a até mesmo dias, é preciso que um dispositivo consiga armazenar os dados de uma forma robusta e persistente.

Com o intuito de aplicar tal tecnologia, foi criada uma sobrecamada (overlay) ao TCP/IP abaixo da camada de aplicação. Essa camada foi intitulada de camada de agregação (bundle layer).

O protocolo de agregação é o responsável por realizar o armazenamento persistente e a atividade de encaminhar e repassar os dados. As aplicações DTN podem enviar mensagens de tamanhos variáveis, essas mensagens são transformada pelo protocolo de agregação em

unidades de dados de protocolo(Protocol Data Units – PDU) que são denominadas agregados (bundle), o qual são armazenados e encaminhados pelos nós DTN. Dessa forma, um pedido de transfrência de um arquivo pode ser enviado contendo os dados necessários para a autenticação do usuário, o nome do arquivo desejado e o diretório local onde o arquivo deve ser entregue. Todas essas informações são então agregadas e enviadas de uma única vez com a intenção de evitar diversas trocas de mensagens que são realizadas numa transferência de arquivos realizada em uma rede TCP/IP convencional.

O tamanho do agregado pode ser reduzido de acordo com as características da rede regional atravessada. As funções de fragmentação e reagrupamento do agregado são executadas pelo protocolo de agreação. Após a fragmentação, cada fragmento continua sendo visto como um agregado que pode ser fragmentado outras vezes. Dois ou mais fragmentos podem ser reagrupados em qualquer lugar da rede, formando um novo agregado.

Além dos pontos citados acima, destaca-se também a importância da gerência de buffer em redes DTN. Como o atraso entre as mensagens é variável, pode-se acumular muitas delas sem que haja um nó intermediário ou não para que possa ser repassada essa informação, assim, o buffer poderá não ter espaço para recebê-las. Logo, faz-se necessário o uso de políticas de gerência de buffer a fim de que a escolha pela remoção de uma mensagem seja a que causará menos impacto a rede.

2 - Implementação

A fim de simular a gerência de buffer quando ocorrer um *overflow* em redes DTN, primeiramente adotar-se-á um ambiente real para reproduzi-la e, em seguida, tratam-se os problemas em análise. No caso, um exemplo real pode ser dado pela locomoção de alunos, que representarão os nós da rede, em uma universidade os quais trocarão mensagens de acordo com o alcance de cada um. Assim, como variáveis reais tem-se:

- Ambiente: Universidade Federal do Amazonas - UFAM;
- Nós da Rede: Alunos;
- Movimentação dos Nós: locomoção dos alunos.

Para realizar esta simulação, é necessário a limitação do espaço em que os nós poderão se mover, UFAM. Com isto, adotam-se os nós, alunos, e informa-se como eles podem se mover. Após estes passos, duas políticas foram escolhidas para reproduzir o controle de gerência de buffer em redes DTN.

2.1 Coleta dos Dados

Afim de obter dados para geração do modelo adotado e posteriormente validação do mesmo, utilizou-se a aplicação *MyTracks*, que captura a trajetória de um usuário a partir do GPS de um aparelho celular. A aplicação foi executada dentro do ambiente do modelo, no caso a própria universidade, no intervalo de cinco dias. Contudo, em um dos dias, ocorreu uma falha na aplicação de modo que foi necessário reiniciá-la, sendo assim, neste dia foram gerados dois arquivos com a trajetória, totalizando seis arquivos.

O próximo passo seria gerar uma versão dos dados obtidos legível ao simulador, para posteriormente realizar a simulação com estes dados e com as políticas implementadas. Um tutorial foi disponibilizado pelo monitor da disciplina para a geração do arquivo.

Primeiramente, obteve-se os valores correspondentes à latitude e longitude mínima e máxima para cada arquivo, acrescentando uma linha ao arquivo de extensão .gpx gerado pela aplicação do celular. A linha adicionada é semelhante ao código a seguir:

<bounds minlat="-3.101157" minlon="-59.982712" maxlat="-3.087808" maxlon="-59.960546" />

Estes valores foram obtidos a partir do programa *track maker*.

Com os valores obtidos pelo celular e os valores máximos e mínimo de latitude e longitude adicionados ao arquivo, utilizou-se o programa *bonnmotion* para gerar um arquivo legível ao simulador para cada trajetória registrada. Contudo o arquivo de mobilidade para o simulador ONE precisa ser único e conter as informações de todos os arquivos. Deste modo, os valores foram mesclados em um único arquivo herdando as configurações máximas entre todas coletadas, identificando cada nó com um número e ordenando as posições dos nodos pelo tempo de simulação. Para gerar este arquivo, utilizou-se o comando *sort* no terminal do linux, aplicado aos arquivos correspondentes de cada nó.

2.2 Políticas de Gerência de Buffer

Para o controle da gerência de buffer no problema em estudo nas redes DTN, aplicam-se duas políticas: Drop LRU e Drop Social.

- Drop LRU: a mensagem recebida menos recentemente será a descartada do buffer;

- Drop Social: nesta política, cada nó da rede tem um laço social com os demais nós. Este laço é um número no intervalo [0, 1], assim, quanto maior o número, maior será o laço entre os nós. Se um nó não conhece o outro, então o nível social deles é zero. Logo, quando houver um overflow no buffer, a mensagem a ser descartada será aquela com menor laço social referente ao último nó que a enviou. Se houver mais de uma mensagem para ser descartada (mesmo nível social), remove-se aquela de maior tamanho.

A implementação destas duas políticas envolveu a criação de duas classes: *DropLRU.java* e *DropSocial.java*, e alteração de outras duas no simulador ONE - *ActiveRouter.java* e *MessageRouter.java*. Basicamente, foi a adição de dois atributos à classe *MessageRouter.java* referentes a cada política e a inserção dos hosts a uma lista de hosts da classe *DropSocial.java*, e a modifcação de dois métodos para tratar a liberação de espaço no buffer na classe *ActiverRouter.java*. (o código das políticas e mais as alterações feitas se encontram no Apêndice)

2.2.1 DropLRU.java

Nesta política, a ideia principal é ter controle sobre o tempo em que cada mensagem chegou no nó. A fim de não fazer diversas alterações nos arquivos do simulador, criou-se uma classe que contém uma estrutura do tipo hash para armazenar o id da mensagem que está atualmente no buffer e o tempo de chegada.

A estrutura hash foi adotada pois em termos de complexidade é a que geraria um acesso mais rápido às mensagens existentes no buffer. Pois, dependendo do tamanho do buffer, podem-se ter diversas mensagens que se fossem armazenadas em uma lista ou mesmo vetor, teriam um tempo de acesso linear. Já o hash proporciona no melhor caso uma consulta em tempo constante. Outras estruturas como árvores não foram utilizadas, pois a complexidade da implementação e do tempo de acesso não seriam vantajosos já que a linguagem java fornece uma implementação padrão do hash.

Além disso, ressalta-se que o tempo de chegada da mensagem no buffer de cada nó será sempre atualizado independente da mensagem já ter sido ou não recebida. Ele é incrementado conforme as mensagens são recebidas, sendo comum a todos os nós da rede.

Os métodos presentes nesta classe são:

- *private void addOrReplace (String id)*: insere a mensagem recebida na estrutura hash com o respectivo tempo;

- *private void showHash (ActiveRouter router)*: este método apenas apresenta as mensagens atuais no buffer. Ele será utilizado apenas na remoção de uma mensagem para verificar se o espaço a ser liberado do buffer está de acordo com a política tratada;

- *private void dropMessage (String id, ActiveRouter router)*: realiza a liberação do espaço no buffer informando antes quais eram as mensagens atuais (método *showHash*) e qual delas será descartada;

- *protected boolean checkMessage (Message m)*: verifica se a mensagem existe no hash. Caso contrário, ela será inserida e um sinal de retorno emitido para que ela seja recebida pelo nó;

- *private String getLRUMessage ()*: verifica qual mensagem foi menos recentemente utilizada, no caso, recebida, e retorna-se o id da mesma;

- *private void updateHash (ActiveRouter router)*: atualiza o hash de acordo com as mensagens atuais no buffer;

- *protected boolean checkSpace (Message m, ActiveRouter router)*: verifica se a mensagem poderá ser armazenada no buffer. Se puder, libera-se espaço até que haja o suficiente para ser inserida.

Obs: O método *updateHash()* foi implementado pois havia casos em que, quando uma mensagem deveria ser recebida pelo nó em questão, ela não era inserida no buffer. Aparentemente, isto mostrou-se como sendo alguma manipulação do simulador ONE que não estava atribuindo corretamente as mensagens ao buffer. Então, para que o conteúdo armazenado no hash fosse igual ao do nó, utilizou-se essa função para garantir as informações.

2.2.2 DropSocial.java

Inicialmente, adotou-se uma estrutura hash para armazenar o nome dos hosts presentes na simulação, porém esta abordagem foi modificada para uma lista normal, pois qualquer que fosse a estrutura adotada, seria utilizada apenas uma vez quando o laço social para cada nó fosse inicializado, então não haveria necessidade de manter na memória uma estrutura como hash.

Logo, esta classe contém uma lista dos nós presentes na simulação comum a todos os objetos que ela terá. A lista é incrementada sempre que um novo nó é inicializado.

Cada objeto, no caso roteador, terá uma tabela hash (utilizada pelos mesmos motivos da classe *DropLRU.java*) cuja chave será o host e o conteúdo, o nível social. Então, esta tabela corresponderá ao laço social do nó atual com os demais.

Os métodos presentes nesta classe são:

- *protected void addHostToList (DTNHost host)*: adiciona um novo host a lista de hosts;
- *private void setSocialLevel ()*: para cada roteador presente na lista de hosts, associa-se na estrutura hash o roteador com um nível social - este será um número aleatório entre [0, 1]. Este método é executado apenas uma vez por cada nó;
- *private DTNHost getLastDTNHost (Message m)*: retorna-se o penúltimo salto (*hop*) da mensagem, pois no simulador ONE, o último salto é o nó atual;
- *private void showBuffer (ActiveRouter router)*: apresenta quais são as mensagens atuais no buffer;
- *private void dropMessage(Message m, ActiveRouter router)*: descarta-se a mensagem de maior tamanho em que o host do penúltimo salto tenha o menor nível social para o nó atual;
- *private String getLessSocialLevelMessage (ActiveRouter router)*: retorna-se o id da mensagem de menor nível social e de maior tamanho;

- *protected boolean checkSpace (Message m, ActiveRouter router)*: verifica se a mensagem pode se armazenada no buffer, se puder, libera-se espaço na memória até que ela possa ser armazenada.

É importante observar que tanto nesta política quanto na anterior, uma mensagem pode não ser descartada do buffer pois poderá estar sendo transferida no momento em que se tenta removê-la. Assim, o espaço ocupado por ela não será modificado.

3 - Resultados

Como visto no tópico 2, o modelo conceitual foi definido e a partir dele as informações foram coletadas para representá-lo na simulação. Dado o ambiente (UFAM) e com os dados produzidos pelo aplicativo MyTracks, pode-se reproduzir a área real e como os alunos (nós) se movimentam por ela.

Ainda com base no aplicativo, como descrito anteriormente, 6 arquivos foram gerados, o que proporcionou a representação de 6 nós na rede DTN e como seria a movimentação dos mesmos no problema real. Assim, o modelo conceitual pode ser representado como um arquivo o qual foi utilizado no simulador ONE.

Contudo, não foi possível realizar a simulação com os dados gerados devido a um erro apresentado pelo simulador. As variáveis de interesse foram definidas e o modelo de mobilidade acrescentado, porém este não pode ser validado com base em resultados conhecidos já que a simulação não foi possível.

Assim, com base nas técnicas de verificação e validação no modelo desenvolvido, observa-se que as características do ambiente foram bem definidas: área em análise, elementos da rede, como eles se movimentam e trocam mensagens (com o uso do simulador), o que pode ser visualizado como sendo um problema real de redes DTN.

Quanto as políticas de gerência de buffer, os resultados gerados mostraram-se válidos. Aplicando as técnicas de verificação e validação, elas foram aplicadas em um ambiente padrão do simulador em que o trace era apresentado na tela do usuário para verificar se a política de gerência aplicada liberava um espaço válido no buffer em questão, assim, mostrou-se consistente com o que foi proposto.

- Exemplo de Drop Social:

Router c62 - Buffer:

id M13 Nivel Social 0.25054771951214483
id M7 Nivel Social 0.41873227938151103
id M6 Nivel Social 0.640720358623506
id M17 Nivel Social 0.41873227938151103
id M2 Nivel Social 0.1025620111514578
id M1 Nivel Social 0.1349110502300065

Drop Message M2 from c62

- Exemplo de Drop Social:

Router w104 - Buffer:

id M13 time 136826
id M12 time 136827
id M10 time 136824
id M5 time 134013
id M26 time 136825
id M4 time 134014

Drop Message M5 from w104

Ressalta-se que elas não foram aplicadas ao modelo desenvolvido, pois, como apresentado acima, o simulador não permitiu a execução do mesmo.

4 - Conclusão

A etapa de modelagem é de suma importância para realizar uma boa simulação, pois por meio dela uma boa parte dos princípios do problema a ser resolvido pela aplicação serão considerados e testados antes mesmo de realizar a execução real. Esta fato pode poupar perdas imensuráveis de recursos utilizados pelo sistema, devido ao fato de que, através da simulação, erros podem ser detectados antes mesmo de ocorrerem em uma execução real proporcionando a possibilidade de contorná-lo anteriormente à sua execução.

Entretanto não basta que o modelo seja criado, e sim que possua características o mais próxima possível da aplicação real, pois assim os resultados de uma simulação utilizando este modelo tendem a ser também semelhantes aos resultados gerados pela aplicação real. É com este intuito que os processos de verificação e validação são utilizados no modelo.

Passadas as fases de validação e geração dos dados, a etapa de verificação foi de suma importância, pois as técnicas mencionadas na literaturas e em sala de aula ajudam a organizar e evitar falhas na codificação do modelo, permitindo que em fases futuras do trabalho a taxa de eficácia da simulação seja maior.

Outro fator importante foram as políticas implementadas, afinal, quanto mais realista for o resultado gerado pelo algoritmo mais fiel será o resultado esperado pelo conceito da política. Isso ajudará futuramente no processo de validação da simulação onde os resultados serão comparados com resultados reais para qualquer caso testado, além de poder ser utilizado no processo de verificação com entradas nas quais se conhece a respectiva saída.

Apêndice

Dois arquivos foram adicionados ao *routing/* do ONE: *DropLRU.java* e *DropSocial.java*. Porém, para que pudessem ser utilizados, duas alterações em arquivos distintos no mesmo diretório - *MessageRouter.java* e *ActiveRouter.java,* foram feitas.

MessageRouter.java

Dois atributos, referentes a cada política de gerência de buffer, foram adicionados, assim como, métodos para a manipulação dos mesmos. É importante também inicializá-los nos construtores desta classe.

```
private DropLRU dropList;
private DropSocial dropSocial;

public DropLRU getDropList() {
        return this.dropList;
}

public DropSocial getDropSocial() {
        return this.dropSocial;
}
```

Além disso, foi modificado o método *init()* para que sempre quando um novo nó fosse inicializado, o nome do mesmo fosse armazenado em uma lista de hosts da classe *DropSocial.java*, a fim de se ter o controle dos nós presentes na simulação.

```
public void init(DTNHost host, List mListeners) {
        this.incomingMessages = new HashMap();
        this.messages = new HashMap();
        this.deliveredMessages = new HashMap();
        this.mListeners = mListeners;
        this.host = host;
        this.dropSocial.addHostToList(host); // inserção do host no hash da classe DropSocial
}
```

ActiveRouter.java

Nesta classe, apenas dois métodos foram alterados: o primeiro libera espaço no buffer para que uma mensagem criada pelo próprio roteador seja armazenada, o segundo verifica se a mensagem pode ser recebida pelo roteador e onde as funções correspondentes a cada política serão adicionadas.

Obs: Como pode haver apenas uma política sendo utilizada por vez, ter-se-á as linhas da política Drop Social comentadas.

```
/*
 * método para liberar espaço no buffer a fim de que a mensagem criada
 * seja armazenada
 */
public boolean createNewMessage(Message m) {
	this.getDropList().checkSpace(m, this);
	//this.getDropSocial().checkSpace(m, this);
	return super.createNewMessage(m);
}

protected int checkReceiving(Message m) {
	if (isTransferring()) {
		return TRY_LATER_BUSY; // only one connection at a time
	}

	if (isDeliveredMessage(m)){
		return DENIED_OLD; // already seen this message -> reject it
	}

	if (m.getTtl() <= 0 && m.getTo() != getHost()) {
		// TTL has expired and this host is not the final recipient
		return DENIED_TTL;
	}

	/* Drop Least Recently Received */
	if (this.getDropList().checkMessage(m))
		return DENIED_OLD;
	if (!this.getDropList().checkSpace(m, this))
		return DENIED_NO_SPACE;

	/* Drop Social */
	// if (hasMessage(m.getId()))
	//	return DENIED_OLD;
	// if (!this.getDropSocial().checkSpace(m, this))
	//	return DENIED_NO_SPACE;

	return RCV_OK;
}
```

DropLRU.java

```
package routing;

import java.lang.Integer;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;

import core.Message;

class DropLRU {

	private static int time;
	private HashMap<String, Integer> hash;

	public DropLRU () {
		this.time = 1;
		this.hash = new HashMap<String, Integer>();
	}

	private void addOrReplace (String id) {
		this.hash.put(id, this.time++);
	}

	/* informa quais as mensagens que estão no buffer */
	private void showHash (ActiveRouter router) {

		int mTime;
		String id;
		Iterator it = hash.entrySet().iterator();
		Map.Entry mapEntry;

		System.out.println("Router " + router.getHost() + " - Buffer:\n");
		while (it.hasNext()) {
			mapEntry = (Map.Entry) it.next();
			mTime = Integer.parseInt(mapEntry.getValue().toString());
			id = mapEntry.getKey().toString();
			System.out.println("id " + id + " time " + mTime);
		}
	}

	private void dropMessage (String id, ActiveRouter router) {
		showHash(router);
		this.hash.remove(id);
		router.deleteMessage(id, true); // true representa um drop nesta função
		System.out.println("\nDrop Message " + id + " from " + router.getHost() + '\n');
	}

	/* se a mensagem encontra-se no buffer, atualiza-se o tempo de chegada */
```

```
protected boolean checkMessage (Message m) {

	if ( !this.hash.containsKey(m.getId()) )
		return false;

	addOrReplace(m.getId());
	return true;
}

/* retorna-se o id da mensagem que menos recentemente foi utilizada */
private String getLRUMessage () {

	int lruTime = -1, mTime;
	String lru = null;
	Iterator it = hash.entrySet().iterator();
	Map.Entry mapEntry;

	while (it.hasNext()) {
		mapEntry = (Map.Entry) it.next();
		mTime = Integer.parseInt(mapEntry.getValue().toString());
		if (mTime <= lruTime || lruTime == -1) {
			lruTime = mTime;
			lru = mapEntry.getKey().toString();
		}
	}

	return lru;
}

/* atualiza-se o hash de acordo com as mensagens atuais no buffer */
private void updateHash (ActiveRouter router) {

	String id;
	Object objs[] = hash.keySet().toArray();

	for (Object obj: objs) {
		id = obj.toString();
		if (!router.hasMessage(id))
			this.hash.remove(id);
	}
}

/* remove-se, se necessário, alguma mensagem do buffer para que a nova mensagem entre */
protected boolean checkSpace (Message m, ActiveRouter router) {

	int freeBuffer, messageSize = m.getSize();
	Message lru;
	String id;
```

```
                if (messageSize > router.getBufferSize())
                        return false; /* tamanho da mensagem maior que o buffer */

                updateHash(router);
                /* remove-se mensagens do buffer até que haja espaço */
                freeBuffer = router.getFreeBufferSize();
                while (freeBuffer < messageSize) {

                        id = getLRUMessage();
                        if (id == null)
                                return false; // nenhuma outra mensagem pode ser deletada
                        lru = router.getMessage(id);
                        dropMessage(id, router); /* remove-se a mensagem especificada */
                        freeBuffer += lru.getSize();
                }

                addOrReplace(m.getId());
                return true;
        }

}
```

DropSocial.java

```
package routing;

import java.lang.Double;
import java.lang.Math;
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;

import core.Message;
import core.DTNHost;

class DropSocial {

        /* lista dos hosts presentes na simulação */
        private static List<DTNHost> hostsList = new ArrayList<DTNHost>();
        private HashMap<DTNHost, Double> knownRouters;

        public DropSocial () {
                this.knownRouters = new HashMap<DTNHost, Double>();
        }
        protected void addHostToList (DTNHost host) {
                hostsList.add(host);
```

```
}

/*
 * Para cada roteador da simulação, atribui-se um nível social entre 0 e 1.
 * Esta função é realizada somente uma vez em cada roteador, pois o nível
 * social é fixo durante toda a simulação.
 */
private void setSocialLevel () {
	Iterator it = this.hostsList.iterator();
	DTNHost host;
	while (it.hasNext()) {
		host = (DTNHost) it.next();
		if (host == router.getHost())
			knownRouters.put(host, 1.0);
		else
			knownRouters.put(host, Math.random());
	}
}

/*
 * retorna-se o penúltimo salto da mensagem já que o último é o roteador
 * atual.
 */
private DTNHost getLastDTNHost (Message m) {
	if (m.getHopCount() == 0)
		return null;
	return m.getHops().get(m.getHopCount()-1);
}




private void showBuffer (ActiveRouter router) {

	int mTime;
	String id;
	Map.Entry mapEntry;

	System.out.println("Router " + router.getHost() + " - Buffer:\n");
	Message ms[] = router.getMessageCollection().toArray(new Message[0]);
	for (Message m: ms)
		System.out.println("id " + m.getId());
}

/* Elmina-se a mensagem com menor nível social e com o maior tamanho */
private void dropMessage(Message m, ActiveRouter router) {

	DTNHost lastHost;

	showBuffer(router);
```

```
		lastHost = getLastDTNHost(m);
		if (lastHost == null) return;
		router.deleteMessage(m.getId(), true);
		System.out.println("\nDrop Message " + m.getId() + " from " + router.getHost() + '\n');
}

/*
 * Para cada mensagem do buffer, analisa-se qual veio do host de menor nível
 * social e tem o maior tamanho.
 */
private String getLessSocialLevelMessage (ActiveRouter router) {

		Message mToDrop = null;
		DTNHost hostM, hostDrop;
		Collection<Message> messages = router.getMessageCollection();

		for (Message m: messages) {
			/*
			 * quando não há host, então a mensagem foi criada no atual e ainda não
			 * foi saiu do buffer
			 */
			if (getLastDTNHost(m) == null) continue;
			if (mToDrop == null)
				mToDrop = m;
			else {
				hostM = getLastDTNHost(m);
				hostDrop = getLastDTNHost(mToDrop);
				if (knownRouters.get(hostM) == knownRouters.get(hostDrop) &&
							m.getSize() > mToDrop.getSize())
					mToDrop = m;
				else if (knownRouters.get(hostM) < knownRouters.get(hostDrop))
					mToDrop = m;
			}
		}
		if (mToDrop == null) return null;
		return mToDrop.getId();
}

/*
 * Verifica se o tamanho da mensagem é menor ou igual ao do buffer, caso
 * seja aceitável, elimina-se mensagens do buffer de acordo com a política
 * Drop Social se necessário.
 */
protected boolean checkSpace (Message m, ActiveRouter router) {

		int freeBuffer, messageSize = m.getSize();
		Message lessSocial = m;
		String id;
```

```
                /* inicializa-se os hosts com níveis sociais aleatórios */
                if (knownRouters.size() == 0)
                        setSocialLevel();
                if (messageSize > router.getBufferSize())
                        return false; /* mensagem não por ser armazenada no buffer */

                /* elimina-se mensagens do buffer até que haja espaço suficiente */
                freeBuffer = router.getFreeBufferSize();
                while (freeBuffer < messageSize) {
                        id = getLessSocialLevelMessage(router);
                        /* não há mais mensagens para serem removidas */
                        if (id == null)
                                return false;
                        lessSocial = router.getMessage(id);
                        dropMessage(lessSocial, router); /* remove-se a mensagem */
                        freeBuffer += lessSocial.getSize();
                }
                return true;
        }
}
```

Referências

[1] MOTA, Edjair. Notas de Aula. 2012

[2] KERÄNE, Ari; OTT, Jörge; KÄRKKÄINEN, Teemu. The ONE Simulator for DTN Protocol Evaluation. Kelsinki University of Technology. 2009

[3] Carina T. de Oliveira, Marcelo D. D. Moreira , Marcelo G. Rubinstein, Luís Henrique M. K. Costa e Otto Carlos M. B. Duarte. Redes Tolerantes a Atrasos e Desconexões. COPPE/Poli - Universidade Federal do Rio de Janeiro. PEL/DETEL - FEN - Universidade do Estado do Rio de Janeiro.