UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Uso do Network Simulator-NS para Simulação de Sistemas Distribuídos em Cenários com Defeitos

por

RENATA DE MORAES TRINDADE

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Profa. Ingrid Eleonora Schreiber Jansch-Pôrto
Orientadora

Prof. Antônio Marinho Pilla Barcellos
Co-Orientador

Porto Alegre, março de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Trindade, Renata de Moraes

Uso do Network Simulator-NS para Simulação de Sistemas Distribuídos em Cenários com Defeitos / por Renata de Moraes Trindade. – Porto Alegre: PPGC da UFRGS, 2003.

133p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientadora: Jansch-Pôrto, Ingrid Eleonora Schreiber; Co-Orientador: Barcellos, Antônio Marinho Pilla.

1. Simulação. 2. Tolerância a falhas. 3. Sistemas distribuídos. 4. Simulação de defeitos. 5. Colapso. I. Jansch-Pôrto, Ingrid Eleonora Schreiber. II. Barcellos, Antônio Marinho Pilla. III. Título.

Agradecimentos

Gostaria de agradecer aos meus pais por todo amor, carinho e apoio que sempre me deram. Obrigado por serem pais tão dedicados e sempre preocupados com o bem-estar de seus filhos. Amo vocês! Mãe, obrigado por largar tudo em Santa Maria e vir para Porto Alegre me dar força quando precisei.

Agradeço ao meu irmão Gustavo, que me emprestou seu computador todas as vezes que o meu parou de funcionar e ao meu irmão Serginho.

Aos meus amigos, colegas de trabalho e todos que de certa forma me ajudaram a concluir esse trabalho. Obrigado pelas conversas e palavras de apoio.

À minha orientadora, professora Ingrid e ao meu co-orientador, professor Marinho, pela orientação durante esse período.

A CAPES pelo suporte financeiro recebido.

Sumário

Lista de Siglas e Abreviaturas

ADU	Application-level Data Unit
CBR	Constant Bit Rate
CESIUM	Centralized Distributed-Execution Simulator with Failure Modeling
DARPA	Defense Advanced Research Projects Agency
LBNL	Lawrence Berkeley National Laboratory
LAN	Local Area Network
FTP	File Transfer Protocol
IP	Internet Protocol
NS	Network Simulator
NS-2	Network Simulator version 2
OTcl	MIT Object Tcl
RTP	Real-Time Transport Protocol
SRM	Scalable Reliable Multicast
TCL	Tool Command Language
TCP	Transmission Control Protocol
TTL	Time to Live
UC	University of California
UDP	User Datagram Protocol
UID	Unique Identifier
USC/ISI	University of Southern California/ Information Sciences Institute
VINT	Virtual InterNetwork Testbed
WAN	Wide Area Network

Lista de Figuras

Resumo

O desenvolvimento de protocolos distribuídos é uma tarefa complexa. Em sistemas tolerantes a falhas, a elaboração de mecanismos para detectar e mascarar defeitos representam grande parte do esforço de desenvolvimento. A técnica de simulação pode auxiliar significativamente nessa tarefa. Entretanto, existe uma carência de ferramentas de simulação para investigação de protocolos distribuídos em cenários com defeitos, particularmente com suporte a experimentos em configurações "típicas" da Internet. O objetivo deste trabalho é investigar o uso do simulador de redes NS (*Network Simulator*) como ambiente para simulação de sistemas distribuídos, particularmente em cenários sujeitos à ocorrência de defeitos.

O NS é um simulador de redes multi-protocolos, que tem código aberto e pode ser estendido. Embora seja uma ferramenta destinada ao estudo de redes de computadores, o ajuste adequado de parâmetros e exploração de características permitiu utilizá-lo para simular defeitos em um sistema distribuído. Para isso, desenvolveu-se dois modelos de sistemas distribuídos que podem ser implementados no NS, dependendo do protocolo de transporte utilizado: um baseado em TCP e o outro baseado em UDP. Também, foram estudadas formas de modelar defeitos através do simulador. Para a simulação de defeito de colapso em um nodo, foi proposta a implementação de um método na classe de cada aplicação na qual se deseja simular defeitos.

Para ilustrar como os modelos de sistemas distribuídos e de defeitos propostos podem ser utilizados, foram implementados diversos algoritmos distribuídos em sistemas síncronos e assíncronos. Algoritmos de eleição e o protocolo Primário-*Backup* são exemplos dessas implementações. A partir desses algoritmos, principalmente do Primário-*Backup*, no qual a simulação de defeitos foi realizada, foi possível constatar que o NS pode ser uma ferramenta de grande auxílio no desenvolvimento de novas técnicas de Tolerância a Falhas. Portanto, o NS pode ser estendido possibilitando que, com a utilização dos modelos apresentados nesse trabalho, simule-se defeitos em um sistema distribuído.

Palavras-chave: simulação, tolerância a falhas, sistemas distribuídos, simulação de defeitos, colapso.

TITLE: “USE OF NETWORK SIMULATOR-NS FOR DISTRIBUTED SYSTEMS SIMULATION IN SCENARIOS WITH FAILURES”

Abstract

The development of distributed protocols is a complex task. In fault-tolerant systems, mechanisms for detecting and masking failures represent a large part of the development effort. Simulation is a technique that can help significantly in this task. So, there is a demand for simulation tools that allow the investigation of distributed protocols in scenarios with failures, particularly with support for experiments in “typical” Internet configurations. The aim of this work is to investigate the use of the network simulator NS as an environment for the simulation of distributed systems, particularly in scenarios subject to failures.

NS is a multi-protocol network simulator that is distributed as open source and can be extended with new protocols. Although it is a tool for computer network study, the proper adjustment of its parameters and use of its characteristics allowed the simulation of failures in distributed systems. For that, two models of distributed systems were developed that can be implemented using NS. They depend on the transport protocol chosen: one is based on TCP and the other is based on UDP. Besides, approaches to simulate failures in this simulator were investigated. To simulate crash failures of a node, an implementation of a method in the application class was proposed in which the failure is supposed to occur.

To illustrate how the distributed system and those failure models could be used, several distributed algorithms in synchronous and asynchronous systems were implemented. Leader election algorithms and the Primary Backup protocol are examples of these implementations. From those algorithms, and particularly of the Primary Backup implementation in which the failure simulation was performed, NS showed to be a very useful tool for testing in the development of fault-tolerant new techniques. Therefore, NS can be extended in order to simulate failures in distributed systems, using the models presented in this study.

Keywords: simulation, fault tolerance, distributed systems, failure simulation, crash.

1 Introdução

Um sistema distribuído pode ser definido como um conjunto de computadores autônomos, separados geograficamente, que se comunicam por troca de mensagens. Este sistema deve lidar com várias fontes de indeterminismo: programas são executados em velocidades diferentes, utilizando um número qualquer de processadores, que estão distribuídos sobre uma topologia de rede desconhecida e com tempos de propagação e ordenamento de mensagens imprevisíveis [LYN96]. Por essas razões, desenvolver protocolos distribuídos é uma atividade complexa. Além disso, os mecanismos para detectar e mascarar defeitos representam tipicamente uma grande parcela do desenvolvimento desses protocolos. Assim, testar, depurar e validar tais protocolos são tarefas que ainda apresentam desafios.

Implementações corretas de protocolos devem satisfazer as propriedades de suas especificações para diversos cenários de defeitos e interações com o ambiente. Devido à complexidade dos sistemas distribuídos, é difícil assegurar que propriedades não sejam violadas como resultado de erros de especificação, de projeto ou de implementação. Métodos como verificação formal e modelagem analítica têm sucesso na modelagem do comportamento de sub-sistemas simples, mas em casos complexos ainda são necessárias simulações detalhadas para obter uma exatidão razoável [ALV2000].

Segundo Alvarez e Cristian [ALV2000], os métodos mais viáveis para testar implementações de protocolos submetem estas implementações a experimentos com execução normal e com injeção de falhas em sistemas práticos, testando-se o comportamento dos serviços sob cenários com defeitos. Entretanto, o teste de protocolos em plataformas distribuídas pode apresentar várias dificuldades. Primeiro, é necessário testar se o protocolo se comporta como especificado na presença de defeitos. Para isso, pode-se forçar a ocorrência de um conjunto de defeitos, mas em geral defeitos como perda ou atraso de mensagens podem ocorrer espontaneamente em uma plataforma de teste distribuída, o que diverge do cenário de defeitos pretendido. Além disso, defeitos devem ocorrer em pontos bem definidos do cenário e o momento certo de forçá-los pode ser difícil de determinar. Outra dificuldade é a necessidade de um projeto cuidadoso dos experimentos que serão executados, para levar o sistema distribuído a um caminho de execução particular. A geração de testes randômicos não é suficiente para obter um comportamento particular dentro de um tempo razoável. Além disso, muitas vezes é necessário repetir a mesma execução para cobrir a violação de uma propriedade; entretanto, em um sistema distribuído, o atraso na comunicação e a velocidade de execução mudam de um experimento para outro, tornando difícil controlar e repetir a mesma execução [ALV2000] [ALV97a].

Os simuladores contribuem para a solução destes problemas, pois oferecem um ambiente controlado para validar o comportamento de protocolos existentes, fornecem infra-estrutura para desenvolvimento de novos protocolos e oportunizam o estudo de suas interações. Entretanto, devido à complexidade dos sistemas distribuídos, ainda existem poucas iniciativas na área de simulação que explorem esse contexto com abordagens genéricas, notadamente com suporte adequado para o tratamento de situações com defeitos.

O desenvolvimento de protocolos para sistemas distribuídos robustos, no grupo de Tolerância a Falhas da UFRGS, levou à procura de ferramentas desta natureza, não se conseguindo identificar ferramentas prontas para o objetivo visado. A partir do

estudo de trabalhos relacionados, que serão apresentados na seção 1.1, decidiu-se verificar a viabilidade de uso de *software* disponível, no caso, o NS-2.

O *Network Simulator* versão 2 – NS-2 [NET2001] é uma ferramenta acadêmica proposta em 1989 na Universidade da Califórnia em Berkeley. A partir de 1995, o desenvolvimento do NS passou a ser apoiado pelo DARPA (*Defense Advanced Research Projects Agency*) através do projeto VINT (*Virtual InterNetwork Testbed* [VIN2000]): um projeto colaborativo que envolve USC/ISI (*University of Southern California/ Information Sciences Institute*), Xerox PARC, LBNL (*Lawrence Berkeley National Laboratory*) e UC Berkeley (*University of California, Berkeley*). Adicionalmente ao NS, o projeto VINT é composto por *softwares* que fornecem visualização, emulação, geração de topologias e cenários, entre outras funcionalidades. Além disso, este simulador tem código aberto e pode ser estendido por usuários para aplicações específicas.

O NS tem sido amplamente utilizado por pesquisadores da área de redes, inclusive para desenvolver aplicações ligadas à Internet [YUK2000], [RES2001]. Segundo Legout e Biersack [LEG2002], o NS é o melhor simulador para estudo de protocolos da Internet. O estudo de políticas de filas, controle de congestionamento, desempenho de protocolos, protocolos de *multicast* confiável em nível de transporte e qualidade de serviço, entre outros, são exemplos de uso do NS.

A imensa propagação do uso da Internet tem feito um número cada vez maior de aplicações comerciais migrarem de ambientes centralizados e controlados para uma rede distribuída padrão Internet, onde a instabilidade do sistema é maior. Isto despertou a importância de sistemas robustos – mais confiáveis e tolerantes a falhas. Assim, os sistemas de computação atuais, aliados à Internet, têm fomentado um crescimento contínuo na necessidade de aplicações distribuídas, tanto no âmbito de redes locais (LAN) quanto em redes de longa distância (WAN), sendo que os componentes desses dois tipos de redes se comunicam através de troca de mensagens tipicamente sobre os protocolos UDP/IP (*User Datagram Protocol/Internet Protocol*) ou TCP/IP (*Transmission Control Protocol/Internet Protocol*) [NUN2000]. Como o NS dá suporte para o desenvolvimento de aplicações ligadas à Internet, fornecendo implementações de vários protocolos inclusive UDP e TCP, decidiu-se investigar a possibilidade de utilizar essa ferramenta para simular sistemas distribuídos, incluindo cenários suscetíveis à ocorrência de defeitos. Tratando-se de ferramenta cujos recursos prevêem a descrição e aplicação em ambiente de redes, com ênfase na carga e uso de largura de banda, não era possível antecipar a viabilidade do NS para uso no contexto de sistemas distribuídos. Adicionalmente, uma investigação preliminar mostrou que apenas um modelo de defeitos restrito estava previsto na versão disponível.

Assim, este trabalho reporta uma investigação no uso do NS como ambiente para simulação de sistemas distribuídos, particularmente em cenários sujeitos à ocorrência de defeitos. Tanto na simulação de sistemas distribuídos quanto na simulação de defeitos, foram identificadas algumas dificuldades e limitações, que serão demonstradas ao longo do trabalho. Os resultados são explicados e tratados, nesta dissertação, através de exemplos de implementação.

Ainda é importante ressaltar que, devido à falta de uniformidade no uso dos termos “defeito” e “falha” na literatura, no decorrer desse trabalho, eles serão usados como tradução do inglês respectivamente dos termos *failure* e *fault*, de acordo com a terminologia adotada por Laprie [LAP98]. Segundo Laprie, ocorre um defeito (*failure*) no sistema quando o serviço fornecido não atende à sua especificação. Falha (*fault*) é a

causa hipotética ou identificada de um erro, que, ao se manifestar, resultará em um defeito. Neste trabalho, usa-se o termo defeito para referir-se às conseqüências de falhas internas cujos detalhes não interessam ao estudo, mas cuja resposta na interação com o restante do sistema deixa de atender à especificação.

1.1 Trabalhos Relacionados

Como mencionado, os simuladores possibilitam investigar o comportamento de protocolos de maneira controlada, sendo que a execução da mesma simulação pode ser repetida várias vezes para análise de resultados. Além disso, com o uso de simuladores não é preciso ter disponível o sistema real onde o protocolo será executado. Exemplos de simuladores encontrados na literatura relacionada a esse trabalho serão apresentados a seguir.

Ciarfella *et al.* [CIA94] descrevem um ambiente para desenvolvimento do protocolo Totem, um protocolo de comunicação tolerante a falhas que controla a difusão de mensagens de forma rápida, ordenada e confiável e implementa serviços de *membership*. Este ambiente é um simulador distribuído orientado a eventos discretos, desenvolvido para teste controlado e depuração do protocolo Totem. O código de implementação do protocolo simulado é executado diretamente em vez de ser modelado e defeitos de comunicação ocorrem de forma randômica durante a execução. O propósito primário dessa ferramenta foi o de testar os mecanismos especificamente desenvolvidos no protocolo Totem; portanto não é uma ferramenta que possa ser aplicada de forma genérica a outros tipos de protocolos tolerantes a falhas.

Spin [HOL97] é uma ferramenta para análise da consistência lógica de protocolos e algoritmos distribuídos. Este *software* usa uma linguagem de alto nível chamada Promela para especificar as descrições do sistema. Spin é, basicamente, composto de duas partes: uma que faz verificação formal exaustiva das propriedades do algoritmo, procurando identificar a existência de *deadlocks*, condições de corrida e código não executável, entre outras características, e outra que executa simulações. A linguagem Promela não permite que a arquitetura de um protocolo seja especificada diretamente. Além disso, o simulador do Spin não tem protocolos já implementados e suporte para redes típicas da Internet como o NS.

CESIUM (*Centralized Distributed-Execution Simulator with Failure Modeling*) [ALV97] [ALV9?] é um ambiente orientado a objetos para teste de implementações de protocolos distribuídos tolerantes a falhas. Este ambiente, que foi desenvolvido pelo *Dependable Systems Group* da Universidade da Califórnia em San Diego e teve sua versão 1.0 finalizada em julho de 1996 [CES2002], permite a execução de protocolos sobre condições controladas, incluindo a ocorrência de defeitos e ataques de segurança durante a execução. Por ser o ambiente mais próximo a este trabalho, será discutido em maior detalhe.

CESIUM foi implementado em Java e pode simular um conjunto de tarefas executando em paralelo em *N* nodos de um sistema distribuído. Para isso, esta ferramenta fornece a ilusão de uma execução distribuída, enquanto executa todas as tarefas em um único espaço de endereçamento de memória. Cada tarefa tem um identificador único e é implementada por uma ou mais *threads*, que podem ou não compartilhar variáveis de estado da tarefa. As tarefas são executadas e comunicam-se de

forma orientada a eventos: depois de completar sua inicialização, elas são bloqueadas à espera da ocorrência de eventos. Os eventos que influenciam no comportamento de uma tarefa T são: o recebimento de uma mensagem enviada por T ou por outra tarefa (um evento de entrada), um *timeout* de um temporizador iniciado por T, o colapso de T e seu reinício. Para o desenvolvimento da simulação, deve-se escrever o código de todas as tarefas em Java juntamente com extensões fornecidas pelo CESIUM para controle do experimento e programação orientada a eventos.

O sistema simulado é estruturado como uma hierarquia de serviços. No topo da hierarquia, está a tarefa de controle (*driver task*) que é um cliente centralizado para todos os nodos que estão executando o protocolo. A tarefa de controle é responsável pela geração de requisições e processamento de seus resultados. No próximo nível, estão as *N* tarefas S-server (uma para cada nodo) que implementam o protocolo. Na base da hierarquia, está a tarefa servidora de baixo nível (L-server), que simula os serviços de baixo nível dos quais uma implementação de protocolo depende.

Cada evento programado para ocorrer em uma tarefa T é armazenado em um descritor de eventos que contém o tipo de evento com seus parâmetros e o tempo global futuro no qual ele irá ocorrer. Cada tarefa T tem uma fila de eventos, ordenada pelo tempo global de ocorrência, que contém todos os descritores para os eventos de entrada programados para ocorrer em T no futuro (eventos pendentes de T). Para se comunicar, uma tarefa invoca uma primitiva do CESIUM, que constrói um descritor para o correspondente evento de entrada e o insere na fila de eventos da tarefa destino.

No CESIUM, **defeitos de comunicação** são simulados pela manipulação do conteúdo das filas de eventos: adicionando, modificando ou removendo descritores. O atraso de um pacote pode ser simulado fazendo-se uma alteração no tempo de um evento de entrada da fila do receptor. A corrupção de uma mensagem pode ser feita pela alteração de parâmetros do evento. Perdas de mensagens, duplicação, reordenamento e defeitos bizantinos são simulados de forma semelhante, já que as mensagens enviadas de um nodo são a única parte de seu comportamento visível pelos outros nodos. A manipulação de primitivas de filas fornecida pelo CESIUM permite que a tarefa de controle e a L-server inspecionem e modifiquem o conteúdo das filas. Essas tarefas também podem acessar e, opcionalmente, modificar os valores de variáveis das S-servers para grau de controle adicional. Além disso, a tarefa de controle e a L-server também podem invocar primitivas do CESIUM que causam **defeito de colapso** e reiniciam uma S-server. Este defeito é classificado como colapso com amnésia total: quando reiniciada, uma tarefa re-executa sua inicialização e os valores de suas variáveis são perdidos.

Após análise desse simulador, constatou-se que é possível utilizá-lo para simulação de defeitos em um sistema distribuído. Entretanto, o projeto foi abandonado em 1998, deixando a implementação do CESIUM pendente. Em contraste, o NS está em constante desenvolvimento, com uma lista de discussão ativa [NSU2002]. Outro fator que motivou a utilização do NS em vez do CESIUM, é que com o ajuste adequado de suas características ele dá suporte para simular sistemas distribuídos em redes típicas da Internet, contando com a base de protocolos já implementados como os protocolos de transporte TCP e UDP. Além disso, o NS fornece ferramentas de auxílio de simulação como *softwares* que dão suporte à visualização.

1.2 Metodologia empregada e estrutura do trabalho

Esta dissertação tem por objetivo investigar a possibilidade de empregar o simulador NS para simulação de algoritmos em sistemas distribuídos, considerando a possível ocorrência de defeitos em nodos destes sistemas. Portanto, as etapas básicas necessárias foram:

- demonstrar a viabilidade da modelagem de sistemas distribuídos através do NS;
- implementar funcionalidades correspondentes a defeitos em nodos, que pudessem ser usadas por esses algoritmos.

Para atingir os objetivos desse trabalho, primeiro foi feito um levantamento bibliográfico, no qual foram realizadas pesquisas de ferramentas para simulação de defeitos em sistemas distribuídos, verificando-se a disponibilidade de sistemas semelhantes ao proposto nesse trabalho.

Após, aprofundou-se o estudo do NS, para analisar como ele poderia ser usado para simular um sistema distribuído em cenários com defeitos. Então foram definidos dois modelos de sistema distribuído que podem ser implementados no NS: um baseado no protocolo de transporte UDP e outro no TCP.

A seguir, foram investigadas maneiras de simular defeitos de colapso no NS, verificando-se, primeiramente, a viabilidade de simular defeitos em um sistema distribuído usando apenas as características atuais do NS e, após, verificando-se a possibilidade de modificar alguns de seus componentes. Finalmente, estudou-se vários algoritmos que foram utilizados para exemplificar como os modelos de sistema distribuído e a inserção de defeitos podem ser simulados no NS.

O restante deste trabalho está organizado da seguinte maneira:

- no capítulo 2, serão apresentados conceitos de sistemas distribuídos e classificações de defeitos encontradas na literatura;
- no capítulo 3, uma visão geral do simulador *Network Simulator* – NS será apresentada, descrevendo-se algumas de suas características e as classes que têm maior relevância a esse trabalho;
- no capítulo 4, será feita a especificação do modelo de sistemas distribuídos e do modelo de defeitos assumidos nesse trabalho; também serão apresentadas propostas de simulação de defeitos estudadas;
- no capítulo 5, serão apresentadas implementações simuladas de algoritmos distribuídos em sistemas síncronos e assíncronos, com e sem a ocorrência de defeitos;
- no capítulo 6, serão avaliados os resultados obtidos, que se constituem nas conclusões desse trabalho, e propostas de trabalhos futuros.

2 Conceitos de sistemas distribuídos e classificações de defeitos

As definições de sistema distribuído e dos diferentes tipos de defeitos previstos nesse tipo de sistema são importantes para que se possa construir um modelo de simulação de defeitos consistente. Um sistema distribuído pode ser definido como um conjunto de vários computadores autônomos que estão separados geograficamente e são conectados por uma rede de comunicação. Estes computadores não compartilham relógio nem memória, sendo a troca de informações entre eles feita através de mensagens sobre a rede de comunicação [JAL94]. Além disso, existem outros conceitos relacionados aos sistemas distribuídos que são importantes para esse trabalho, tal como a divisão do modelo do sistema em físico ou lógico e o modelo de tempo. Esses conceitos serão apresentados na seção 2.1. Já na seção 2.2, serão apresentadas as principais classificações de defeitos encontradas na literatura. No entanto, o leitor já familiarizado com tais conceitos pode passar ao capítulo seguinte, e voltar a esse capítulo apenas quando for necessário.

2.1 Modelos de sistemas distribuídos

Um sistema distribuído pode ser visto a partir de duas perspectivas básicas, segundo Jalote [JAL94]: o **modelo físico** que define o sistema através dos componentes físicos, e o **modelo lógico** que o define do ponto de vista do processamento. Outra importante distinção, ao se modelar um sistema distribuído, é entre sistemas síncronos e assíncronos. Estes modelos serão explicados a seguir.

2.1.1 Modelo físico *versus* modelo lógico

No **modelo físico**, os computadores de um sistema distribuído são vistos como nodos, sendo que cada nodo é constituído de um processador com memória volátil, que não é acessível aos outros nodos, e um relógio próprio que define o momento de execução das instruções nesse processador. Cada nodo também tem uma interface de rede através da qual é conectado a uma rede de comunicação. Além disso, o nodo tem algum meio de armazenamento não volátil e um *software* que governa a seqüência de instruções a serem executadas. Portanto, os componentes principais de um sistema distribuído são: processadores, rede de comunicação, relógios, memória não-volátil e *software*. Entretanto, um sistema distribuído, freqüentemente, é modelado com nodos e rede de comunicação como componentes básicos, sendo que a estrutura interna dos nodos não é levada em conta. Cabe lembrar que o *hardware* dos nodos pode ser muito diverso, o que dificulta bastante a modelagem, caso esta estrutura precise ser considerada.

Dependendo do sistema considerado, a rede de comunicação de um sistema distribuído pode consistir de topologias ponto-a-ponto ou em barramento. Uma rede ponto-a-ponto é formada por um conjunto de enlaces que ligam cada par de nodos. Nesse modelo, um sistema distribuído pode ser representado por um grafo onde os

vértices representam os nodos do sistema e as arestas, os enlaces de comunicação. Existem vários tipos de topologias nas quais os nodos são ligados aos pares, sendo as mais comuns: nodos totalmente conectados, estrela, árvore e anel (Figura 2.1).

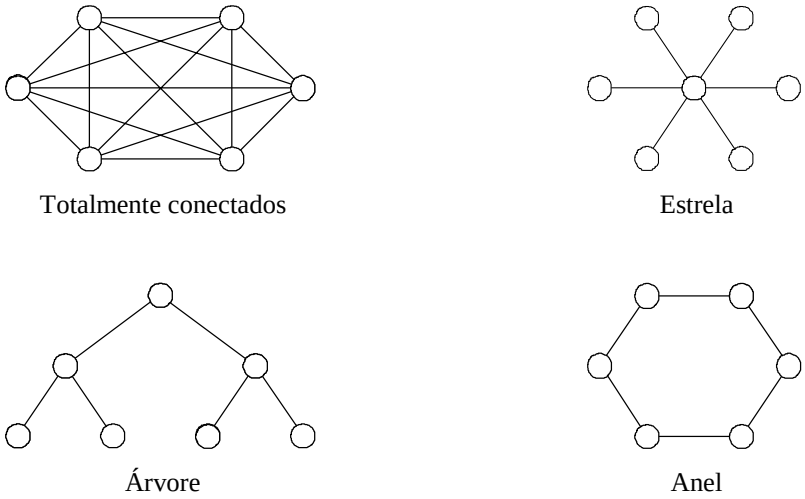


FIGURA 2.1 - Topologias ponto-a-ponto

Na topologia em barramento, ao invés de estarem conectados ponto-a-ponto, os nodos estão conectados a um barramento comum, como mostrado na Figura 2.2.

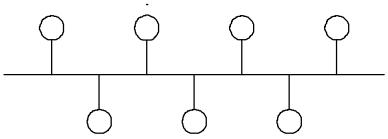


FIGURA 2.2 - Topologia em barramento

Para o programador, um sistema distribuído pode ser considerado somente através das características da aplicação, sem a preocupação com o sistema físico. Uma aplicação distribuída é composta por um conjunto de processos que são executados concorrentemente e cooperam uns com os outros para executar uma tarefa. No **modelo lógico**, os processos são executados em paralelo, comunicam-se através de troca de mensagens e estão em nodos diferentes do sistema.

Ao se empregar o nível lógico da aplicação, a rede física é tratada como se estivesse totalmente conectada, independentemente do tipo de protocolos e topologias empregadas nas camadas subjacentes. Portanto, assume-se que uma mensagem pode ser enviada de um nodo para qualquer outro nodo do sistema. O suporte a esta comunicação, entre os diferentes nodos da rede, é assegurado por protocolos subjacentes específicos. Na Internet, são tipicamente usados TCP e UDP, que, por sua vez, dependem de um conjunto de algoritmos de roteamento para transmissão de pacotes pela rede.

Os protocolos empregados na Internet incluem: o IP, que é o protocolo básico de envio de pacotes; o TCP e o UDP, que são protocolos de transporte que operam sobre o IP. Os protocolos da Internet foram originados em um trabalho feito pelo DARPA em 1970 e seu uso tem crescido gradualmente em redes de larga escala e de alto

desempenho que conectam milhões de computadores. O protocolo IP é um protocolo de transmissão de pacotes não orientado à conexão, denominador comum na Internet. O IP é responsável pela transmissão não confiável de pacotes de tamanhos variáveis de uma máquina origem para uma máquina destino. Uma limitação do protocolo IP é que ele opera com mensagens individuais e não garante confiabilidade. Mensagens podem ser perdidas por várias razões incluindo: defeitos em enlaces, defeitos em máquinas intermediárias que fazem roteamento, ruídos que causam corrupção de dados dos pacotes, ou falta de espaço em *buffer*. Por essa razão, é comum colocar um protocolo confiável sobre o IP de uma arquitetura de comunicação distribuída. O resultado é chamado de canal de comunicação confiável, que é garantido pelo protocolo TCP [BIR96].

TCP é o nome do protocolo orientado à conexão dentro do conjunto de protocolos da Internet. Uma sessão TCP é iniciada com o estabelecimento de uma conexão, na qual um dos programas fica no estado de escuta esperando para aceitar conexões, enquanto outro programa conecta-se a ele. Uma conexão TCP deve garantir que os dados sejam entregues na ordem em que foram enviados, sem perda ou duplicação [BIR96]. Assim, o TCP é responsável pela entrega dos dados de forma apropriada, retransmitindo e reorganizando pacotes, se necessário.

O UDP, outro protocolo da Internet, é não orientado à conexão e não confiável. Não existe garantia que os pacotes serão entregues, que a ordem destes estará correta, ou que não serão duplicados [BIR96]. Portanto, por ser mais simples e não perder tempo com retransmissão de pacotes, o UDP é usado em aplicações onde a entrega dos pacotes, no menor tempo possível, é mais importante do que a entrega de todos os pacotes como, por exemplo, em aplicações multimídia de tempo real.

2.1.2 Sistema síncrono *versus* sistema assíncrono

Várias suposições podem ser feitas a respeito do tempo dos eventos em um sistema segundo Lynch [LYN96]. Em um extremo, processos podem executar comunicações e computações em perfeita sincronia. No outro extremo, os processos podem ser completamente assíncronos, executando tarefas em velocidades diferentes e em ordens arbitrárias. No meio desses dois extremos, existe uma grande variedade de suposições possíveis, que podem ser agrupadas na designação de sistemas parcialmente síncronos. Nesses sistemas, processos têm informações parciais sobre o tempo de eventos, por exemplo, processos podem ter velocidades limitadas, mas o atraso das mensagens é ilimitado ou as mensagens têm tempo de atraso limitado, mas desconhecido.

Segundo Fred B. Schneider [SCH93], um sistema é considerado **síncrono** quando as velocidades relativas de execução de processos são limitadas e os atrasos associados aos canais de comunicação podem ser modelados. Em contraste, um sistema é **assíncrono** quando não é possível fazer suposições sobre a velocidade de execução de processos nem sobre os atrasos na entrega de mensagens.

Outra definição é dada por Vassos Hadzilacos e Sam Toueg [HAD93], na qual um sistema, para ser **síncrono**, deve satisfazer as seguintes propriedades: a) existe um limite máximo conhecido de atraso da mensagem que é o tempo despendido no envio, transporte e recebimento da mensagem sobre um enlace; b) cada processo tem um relógio local com uma taxa conhecida e limitada no desvio com relação ao tempo real;

c) existem, e são conhecidos, limites mínimo e máximo no tempo necessário para um processo executar um passo. Já em um sistema **assíncrono** essas propriedades não são atendidas, portanto, não há limites nos atrasos de mensagens, nem no desvio dos relógios locais, nem no tempo necessário para execução de um passo.

Nancy Lynch [LYN96] define modelos de sistema síncrono e assíncrono do ponto de vista de programação de algoritmos distribuídos. No modelo **síncrono**, é assumido que todos os tempos de comunicação e processamento não são apenas limitados, mas também fixos e iguais para quaisquer enlaces e nodos. Adicionalmente, é assumida a total ausência de falhas. Estes pressupostos permitem que um sistema distribuído seja modelado em termos de rodadas síncronas. No modelo de sistema **assíncrono**, é assumido que componentes separados executam passos em uma ordem arbitrária com velocidades relativas arbitrárias. Apesar de corresponder à maior parte dos sistemas reais, este modelo é mais difícil de programar que o modelo síncrono devido à incerteza na ordem dos eventos. A solução de um problema em um sistema síncrono pode ser útil como um passo intermediário na solução de um problema em um sistema real. No entanto, pode ser impossível ou pelo menos ineficiente implementar modelos síncronos na maioria dos sistemas distribuídos, segundo Lynch.

2.2 Classificações de defeitos em sistemas distribuídos

Como foi visto na seção 2.1.1, os principais componentes de um sistema distribuído são: processadores, rede de comunicação, relógios, memória não-volátil e *software*; com exceção do *software*, todos os componentes são físicos. A maioria dos protocolos para tolerância a falhas, segundo Jalote [JAL94], tem seu foco em falhas de componentes físicos, principalmente falhas de nodos e da rede de comunicação já que estas são as principais causadoras de defeitos dos componentes do sistema. Assim, nessa seção, serão apresentadas quatro classificações de defeitos encontradas na literatura.

2.2.1 Classificação de Cristian

A classificação de defeitos em sistemas distribuídos, sistematizada por Cristian [CRI91], enquadra os defeitos nas seguintes categorias: omissão, colapso, temporização, resposta e arbitrários. A representação gráfica dessas categorias, que identifica a generalidade dos mecanismos empregados para sua detecção, foi proposta por Jalote [JAL94] e é mostrada na Figura 2.3.

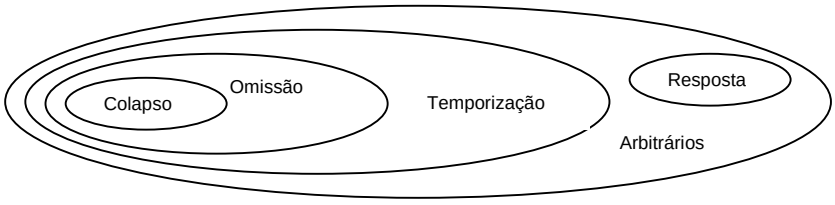


FIGURA 2.3 - Classificação de defeitos por Cristian [JAL94]

Defeito de omissão: este defeito ocorre quando um componente não responde a uma solicitação.

Defeito de colapso (*crash*): este defeito ocorre quando, depois da primeira omissão em produzir um resultado, um componente omite a produção de resultados subseqüentes até ser reiniciado. Dependendo do estado do componente ao ser reiniciado, pode-se distinguir entre vários tipos de comportamentos de defeitos de colapso:

- colapso com amnésia (*amnesia-crash*) ou colapso com amnésia total – ocorre quando o componente reinicia em um estado inicial predeterminado que não depende do que foi recebido antes do colapso;
- colapso com amnésia parcial (*partial-amnesia-crash*) – ocorre quando, no reinício, parte do estado do componente é o mesmo que antes do colapso e o restante é reiniciado para um estado inicial predefinido;
- colapso com pausa (*pause-crash*) – ocorre quando o componente reinicia com o estado que tinha antes do colapso;
- colapso com *halting* (*halting-crash*) – ocorre quando o componente nunca reinicia.

Defeito de temporização: ao ocorrer este tipo de defeito, um componente responde corretamente mas a resposta está fora do intervalo de tempo especificado, podendo ser fornecida muito cedo ou muito tarde; também é chamado de defeito de desempenho.

Defeito de resposta: com esse tipo de defeito, um componente produz resultados incorretos para determinadas solicitações ou realiza uma transição de estado incorretamente.

Defeito arbitrário ou bizantino: faz com que um componente se comporte de maneira totalmente imprevisível.

Com exceção dos defeitos de resposta, os tipos de defeitos formam uma hierarquia. O modo de defeito mais simples é o de colapso, no qual, o componente se comporta de maneira benigna, pois simplesmente pára quando ocorre o defeito. O modo mais geral é o de defeitos arbitrários, no qual, não se pode fazer qualquer suposição sobre o comportamento do componente que está com defeito.

2.2.2 Classificação de Schneider

A classificação de defeitos, proposta por Schneider [SCH93], inclui várias definições previamente empregadas na literatura de sistemas distribuídos e apresenta os seguintes modelos:

***Failstop*:** o processador pára e permanece neste estado. O defeito no processador pode ser detectado por outros processadores.

Colapso (*Crash*): o processador pára e permanece neste estado. Este defeito pode não ser detectado por outros processadores.

***Crash+link*:** o processador pára e permanece neste estado; o enlace perde algumas mensagens, mas não atrasa, não duplica, nem corrompe mensagens.

Omissão na recepção: o processador recebe somente um subconjunto das mensagens que foram enviadas para ele, ou então pára e permanece nesse estado.

Omissão no envio: o processador transmite apenas um subconjunto das mensagens que ele efetivamente tenta enviar, ou pára e permanece nesse estado.

Omissão geral: o processador recebe somente um subconjunto de mensagens que foram enviadas para ele; envia somente um subconjunto das mensagens que ele efetivamente tenta enviar, e/ou pára e permanece neste estado.

Defeito bizantino: o processador exibe um comportamento arbitrário.

Nessa classificação, os defeitos *failstop* correspondem à classe mais fácil de lidar, porque hipoteticamente os processadores não executam operações incorretas e os defeitos podem ser detectados. Assim, outros processadores podem, de forma segura, assumir atividades em lugar do processador que parou de forma defeituosa. Defeitos de colapso em sistemas assíncronos são mais difíceis de lidar que defeitos *failstop* porque, nesse tipo de sistema, é difícil de identificar se o processador parou ou se está muito lento. Já em sistemas síncronos, defeitos de colapso e *failstop* são equivalentes.

Os modelos de defeitos de *crash+link*, omissão de recepção, omissão de envio e omissão geral lidam com perdas de mensagens. Os defeitos bizantinos são os mais problemáticos, pois fazem com que o processador se comporte de qualquer maneira. Portanto, um sistema que tolera defeitos bizantinos, também está apto a lidar com os demais tipos de defeitos caracterizados anteriormente.

2.2.3 Classificação de Hadzilacos e Toueg

A classificação proposta por Hadzilacos e Toueg [HAD93] é dividida em duas categorias: defeitos de processos e defeitos de comunicação. A seguir, serão mostrados os modelos de defeitos de cada uma dessas categorias.

Defeitos de processos

- **Colapso (*crash*):** o processo com defeito pára prematuramente e não faz nada a partir deste ponto. Antes de parar, o processo tem comportamento correto.
- **Omissão no envio:** o processo pára prematuramente e/ou ocasionalmente omite o envio de mensagens que deveria enviar.
- **Omissão na recepção:** o processo pára prematuramente e/ou ocasionalmente omite a recepção de mensagens que são enviadas a ele.
- **Omissão geral:** o processo é submetido a defeitos de omissão no envio e/ou omissão na recepção.
- **Arbitrário** (também chamado de bizantino ou malicioso): o processo pode exibir qualquer tipo de comportamento defeituoso.
- **Arbitrário com autenticação de mensagens:** o processo pode se comportar de maneira arbitrária, mas há um mecanismo para autenticação de mensagens disponível. Um processo com defeito arbitrário pode afirmar que recebeu uma determinada mensagem de um processo sem defeito mesmo que nunca tenha recebido essa mensagem. O mecanismo de autenticação de mensagens permite ao processo não defeituoso verificar essa afirmação.

Nesses modelos, os defeitos arbitrários são considerados os mais “severos” porque não se pode fazer nenhuma restrição ao comportamento do processo com defeito. Por outro lado, defeitos de colapso são os menos severos. A classificação desses

modelos, na interpretação dos autores, organizados do menos severo ao mais severo, é mostrada na Figura 2.4.

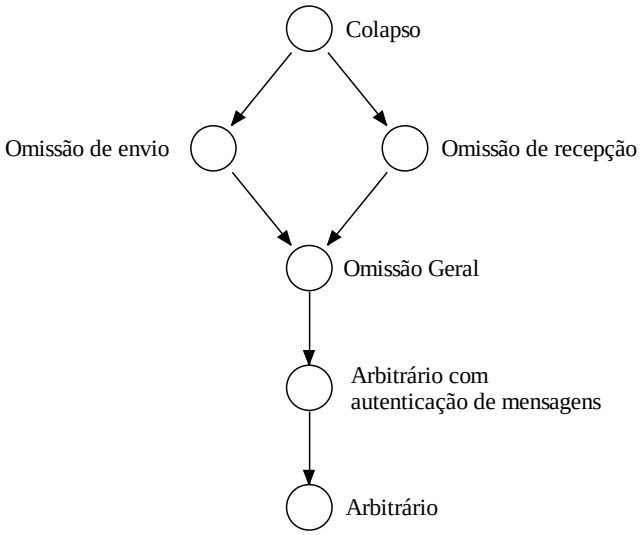


FIGURA 2.4 - Classificação de defeitos por Hadzilacos e Toueg

Defeitos de comunicação

- **Colapso (*crash*):** o enlace de comunicação pára de transportar mensagens. Antes de parar, o enlace se comporta corretamente.
- **Omissão:** o enlace, com esse tipo de defeito, ocasionalmente omite o transporte de mensagens enviadas através dele.
- **Arbitrário** (também chamado de bizantino ou malicioso): o enlace defeituoso pode ter qualquer tipo de comportamento. Por exemplo, pode gerar mensagens falsificadas.

2.2.4 Classificação de Birman

A classificação apresentada por Birman [BIR96] modela defeitos da seguinte forma:

Defeito de *halting*: nesse modelo, um processo ou computador simplesmente pára sua execução, em caso de defeito, sem executar ações incorretas. Nesse modelo, não é possível detectar que o processo parou, exceto por limite de tempo (*timeout*).

Defeito *failstop*: nesse caso, um processo apresenta defeito da mesma forma que no modelo de *halting*, entretanto outros processos que estavam interagindo com este processo podem detectar tal defeito de forma bastante precisa.

Defeito de omissão no envio: ocorre quando deixa de ser enviada uma mensagem que, de acordo com a lógica dos sistemas de computação distribuída, deveria ter sido enviada. Defeitos de omissão de envio são geralmente causados por falta de espaço no *buffer* da interface de rede ou do sistema operacional, que podem fazer com que a mensagem seja descartada depois que o programa de aplicação a enviou e antes de deixar o computador de origem.

Defeito de omissão na recepção: este tipo de defeito é semelhante ao defeito de omissão de envio, mas ocorre quando uma mensagem é perdida próxima do processo de destino, geralmente devido à falta de memória para armazená-la ou devido à descoberta de evidências de corrupção dos dados.

Defeito de rede: este defeito ocorre quando a rede perde mensagens trocadas entre determinados pares de processos.

Defeito de partição de rede: este defeito ocorre quando a rede se fragmenta em subredes desconexas; neste caso, mensagens enviadas entre essas subredes são perdidas.

Defeito de temporização: este tipo de defeito ocorre quando acontece uma violação da propriedade temporal do sistema.

Defeito bizantino: este termo abrange uma grande variedade de defeitos, incluindo corrupção de dados e programas que não seguem o protocolo adequado ou com comportamento malicioso.

2.2.5 Comparação entre as classificações de defeitos

Verificando-se as definições apresentadas nas seções anteriores, pode-se notar que existem algumas diferenças de nomenclatura e funcionalidades. Cristian, na definição básica de defeito de colapso, não caracteriza particularmente a atividade interna de um componente, após o momento de ocorrência do defeito [JAN2002]. Mas, em seguida, divide defeito de colapso em quatro categorias, dependendo do estado do componente quando ele é reiniciado, o que permite deduzir o comportamento interno deste componente após o colapso. O defeito de colapso com *halting* (*halting-crash*), definido por Cristian, é equivalente nas outras classificações aos seguintes defeitos: de colapso segundo Schneider, de colapso segundo Hadzilacos e Toueg e de *halting* segundo Birman. Schneider e Birman ainda apresentam uma classe de defeitos denominada *failstop*, que trata de um caso particular de defeitos de colapso, detectável a partir de outros processadores [JAN2002].

Os defeitos de omissão propostos por Cristian, nas outras classificações apresentadas nesse trabalho, são divididos em omissão na recepção e omissão no envio. Cristian trata somente da omissão do ponto de vista do destinatário, que detecta a ausência de resposta. Adicionalmente, Hadzilacos e Toueg assim como Schneider englobam problemas de envio e recepção, através dos defeitos de omissão geral.

Outra diferença é que Cristian e Birman classificam os defeitos de acordo com a propriedade temporal do sistema, chamando tal defeito de temporização ou desempenho, o que não é feito pelas outras duas classificações. Além disso, Hadzilacos e Toueg classificam defeitos arbitrários de duas formas dependendo da existência de autenticação de mensagens.

Hadzilacos e Toueg, Schneider e Birman estendem suas propostas com a caracterização de defeitos de comunicação. Schneider apenas menciona este tipo de defeito através da classe “colapso+*link*”. Hadzilacos e Toueg apresentam três classes de defeitos de comunicação análogas àquelas definidas para nodos: colapso, omissão e arbitrário [JAN2002]. Birman define dois tipos de defeitos específicos de comunicação: defeito de rede e defeito de partição de rede.

Nesse trabalho, são tratados principalmente defeitos de colapso, sendo que a classificação escolhida como base é a definida por Cristian, já que é caracterizado o

estado de um componente após sua recuperação (se ocorrer). Além disso, para cobrir defeitos de comunicação, adota-se a classificação de Hadzilacos e Toueg.

3 O simulador de redes VINT NS-2

Nesse capítulo, será apresentada uma visão geral do *Network Simulator* – NS, mostrando-se suas características básicas, as linguagens de programação utilizadas por esse simulador e a descrição de algumas de suas classes. Na seção 3.1, serão apresentadas características do NS, como a existência de ferramenta de visualização que auxilia no entendimento de protocolos. Na seção 3.2, serão apresentadas as linguagens que compõem o NS, assim como suas funcionalidades. Na seção 3.3, estarão as classes do NS que têm maior relevância a esse trabalho. Tais classes serão apresentadas para auxiliar no entendimento dos experimentos relatados nos capítulos seguintes. Entretanto, o leitor que já tem conhecimento de simulações do NS pode passar ao próximo capítulo, retornando a esse quando necessário.

3.1 Características

O NS versão 2 é um simulador discreto orientado a eventos. O NS é orientado a objetos e escrito em C++ e OTcl [WET95] (extensão de Tcl orientada a objetos) o que fornece um modelo de programação dual: o processamento de pacotes de comunicação é feito em uma linguagem de sistema – C++, enquanto que a configuração da simulação é feita em uma linguagem *script* – OTcl.

Para realizar uma simulação, o usuário tipicamente escreve um *script* OTcl que define o cenário (topologia e eventos). Uma topologia de rede é construída usando-se basicamente quatro elementos: nodos, enlaces, agentes (que são usados na implementação de protocolos de várias camadas e são anexados aos nodos) e geradores de tráfego (que são anexados aos agentes).

O projeto VINT, através do simulador NS e de *softwares* relacionados, apresenta várias características como: abstração, emulação, geração de cenários, visualização e extensibilidade [BRE2000]. As características de maior relevância são apresentadas a seguir.

Geração de Cenários

No NS, as configurações de entrada para a execução de uma simulação são feitas pela definição de cenários de simulação. Alguns dos componentes desses cenários são:

- a topologia da rede: inclui as interconexões físicas entre os nodos e as características estáticas de enlaces e nodos;
- os modelos de geração de tráfego: definem os padrões de uso da rede e a localização dos geradores;
- a dinâmica da rede: inclui simulação de defeitos em enlaces.

O NS dá suporte para a geração de topologias predefinidas, que podem ser escolhidas de uma biblioteca. Além disso, existem ferramentas que geram topologias automaticamente, fornecendo a possibilidade de criar topologias arbitrárias.

O NS fornece uma grande variedade de modelos de geração de tráfego que podem ser usados juntamente com protocolos de transporte *multicast* ou *unicast*.

Atualmente, o NS suporta protocolos de transporte como UDP, RTP (*Real-Time Transport Protocol*), SRM (*Scalable Reliable Multicast*) e muitas variantes de TCP. Os modelos de geração são aplicações que ficam localizadas sobre os agentes de transporte no NS. Essas aplicações são divididas em geradores de tráfego e aplicações simuladas. Geradores de tráfego disponíveis para aplicações sem fluxo controlado incluem CBR (*Constant Bit Rate*) e geração de tráfego a partir de um arquivo de rastro (*trace*). Aplicações simuladas incluem FTP (*File Transfer Protocol*) e Telnet.

Visualização

Ferramentas para visualização adicionam uma representação dinâmica ao comportamento da rede e fornecem um melhor entendimento dos protocolos, assim como facilitam a sua depuração. Para isso, o NS fornece uma ferramenta de animação chamada Nam (*Network Animator*) [EST2000].

Nam possui animação de pacotes e gráficos de protocolos específicos que ajudam a projetar e depurar novos protocolos. Nam interpreta um arquivo de rastro, armazenado em disco, que contém eventos de rede indexados por tempo para fazer a animação de tráfegos de rede de maneiras diferentes.

O arquivo de entrada do Nam contém todas as informações necessárias para a animação: leiaute estático da rede, eventos dinâmicos como pacotes que chegam ou que partem de um nodo e falhas de enlace. Simulação de redes sem fio incluem localização e movimento dos nodos.

A animação de pacotes é simples, como explicado a seguir. Eventos de rastro indicam quando os pacotes entram e saem de enlaces e filas. Pacotes são mostrados como retângulos com flechas na frente e filas são quadrados (Figura 3.1). Os pacotes e nodos podem ser coloridos. Quando as filas enchem, os pacotes são descartados e aparecem como pequenos quadrados caindo. A janela de animação é interativa: dando-se um clique nos pacotes, enlaces ou nodos, são mostradas informações pertinentes, incluindo estatísticas. Nam permite que a velocidade de animação seja ajustada; pode-se ir à frente ou retornar a visualização, para assim encontrar e analisar ocorrências de interesse.

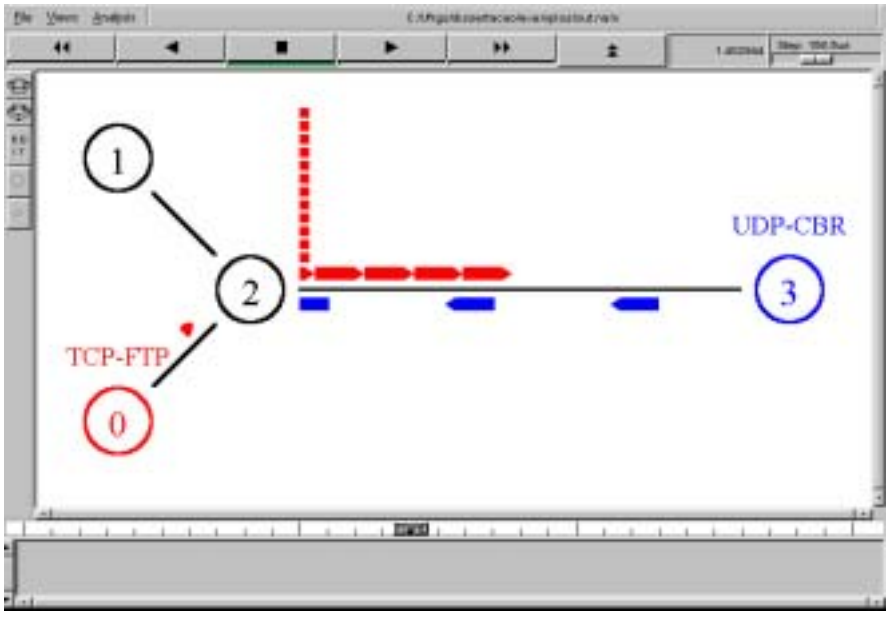


FIGURA 3.1 - Tela de simulação no Nam

Extensibilidade

Permitir a extensão de funcionalidades é uma característica desejável de um simulador. Com esta flexibilidade, pode-se explorar uma variedade de cenários e estudar novos protocolos. O *software* do NS foi construído para que seus usuários possam estendê-lo para atender aos seus próprios interesses.

Em conjunto com essas características, outros aspectos têm impacto substancial no uso do simulador. O primeiro é a disponibilidade de uma grande variedade de módulos. Isso permite a comparação fácil de diferentes abordagens e reduz o tempo de desenvolvimento da simulação, permitindo que se focalize o estudo nos aspectos relevantes do projeto que está sendo estudado. O segundo é a distribuição pública do sistema (NS e *softwares* relacionados) que tem ajudado a reduzir a duplicação de esforço na pesquisa.

3.2 Linguagens no NS

O NS é escrito em duas linguagens C++ e OTcl, que é uma extensão de Tcl orientada a objetos. Tcl (*Tool Command Language*) é uma linguagem *script* interpretada, criada em 1988 por John K. Ousterhout. Esta linguagem é de uso geral e pode ser utilizada para controlar e estender aplicações, fornecendo facilidades de programação genéricas como variáveis, laços e procedimentos. Seu interpretador é uma biblioteca de procedimentos C, que pode ser incorporada a aplicações, podendo ser estendida para atender às necessidades da aplicação [OUS94].

OTcl, abreviação de MIT Object Tcl [WET95a], é uma extensão de Tcl para gerência de tipos de dados complexos e programação orientada a objetos dinâmica, em geral. OTcl foi projetada para o VuSystem [LIN94] e foi usada por dois anos, antes de ser apresentada no TCL/TK Workshop de 1995, como uma distribuição independente.

Quando OTcl foi desenvolvida, seus projetistas procuraram estender Tcl, de forma compacta, com suas próprias facilidades de programação orientada a objetos, em vez de trazer uma linguagem orientada a objetos existente, como C++, para dentro de Tcl. Assim, buscaram um pequeno conjunto de primitivas de objetos que deveriam manter e usar a sintaxe Tcl, sua extensibilidade, simplicidade e compatibilidade com a linguagem C.

Em linguagens orientadas a objetos como C++, as classes são definidas como um bloco único; já em OTcl, as classes são definidas incrementalmente através de tantos blocos quanto forem necessários. Tipicamente, usa-se um bloco por método com pouca preocupação com a ordem. Essa diferença, na definição de classes em OTcl e C++, está associada à extensibilidade; em OTcl métodos e classes podem ser modificados individualmente em qualquer momento.

Cada objeto em OTcl é manipulado por um comando único. Por exemplo, objetos simples são criados com o comando “Object”. OTcl não suporta encapsulamento: todas as variáveis de instância e métodos são públicos. Os projetistas de OTcl consideram que não é necessária proteção quando os programas são bem escritos.

Objetos permitem que dados e procedimentos relacionados sejam agrupados. Mecanismos de herança fortalecem esse modelo por permitir compartilhamento de funcionalidades entre objetos diferentes, assim estimulando a reutilização de código.

Herança em OTcl é baseada em classes e suas superclasses, pois cada objeto tem uma classe e, além de suas próprias especificações, herda funcionalidades da sua classe e da superclasse. OTcl suporta herança múltipla, na qual cada classe pode ter muitas superclasses.

Em Tcl, todos os comandos e valores são representados como *strings*. Assim, é simples passar dados entre o interpretador Tcl e comandos escritos em código C em uma aplicação. Para isso, o código C só precisa estar apto para converter representações internas para *strings*. OTcl mantém a mesma flexibilidade da combinação de Tcl e C da linguagem Tcl. Métodos podem ser implementados em C++ ou OTcl quando apropriado, levando a objetos com implementações mistas.

3.2.1 Utilização de OTcl e C++ no NS

O NS utiliza duas linguagens, porque o simulador executa duas tarefas diferentes. De um lado, simulações detalhadas de protocolos precisam de uma linguagem que possa manipular *bytes* e cabeçalhos de pacotes eficientemente, além de poder implementar algoritmos que contenham um grande conjunto de dados; neste caso, a velocidade de execução é importante.

Por outro lado, uma grande parte das pesquisas envolve variação de parâmetros e configurações, ou explora diferentes cenários. Nesses casos, o tempo de iteração, isto é, a mudança da configuração e sua re-execução, é mais importante do que o tempo de execução, já que a configuração é executada somente uma vez no início de cada experimento.

A primeira característica é encontrada em C++, que, por ser uma linguagem compilada, tem execução rápida. Assim, C++ é adequada para implementação de protocolos detalhados. Já OTcl, como é interpretada, tem execução mais lenta que C++, mas pode ser modificada mais rapidamente pois não é preciso recompilação, sendo ideal para a configuração da simulação.

Os desenvolvedores do NS aconselham o uso de OTcl para operações de controle, configuração, inicialização e manipulação de objetos C++ existentes. E sugerem o uso de C++ para qualquer operação que necessite de processamento de cada pacote de um fluxo ou para mudar o comportamento de uma classe C++ existente. Por exemplo, enlaces são objetos OTcl; então, se o experimento é feito com enlaces que são compostos de atraso, fila e possivelmente módulos de perda, usa-se OTcl. Mas se o experimento necessita alguma característica específica (como comportamento diferente de uma fila ou outro modelo de perda), então é necessário construir um novo objeto C++.

Um problema do uso destas duas linguagens é que o usuário precisa ter bom conhecimento delas, principalmente de OTcl, que é extremamente necessária para o uso do simulador. Nota-se que é possível fazer uma simulação sem utilizar C++, o que não é possível sem OTcl. Por exemplo, *scripts* simples, leiaute da topologia e variação de parâmetros podem ser feitos exclusivamente em OTcl.

O NS inclui uma extensão simples de OTcl chamada TclCL (Tcl com classes), que fornece uma ponte entre C++ e OTcl e permite que uma implementação de um objeto seja dividida entre as duas linguagens.

3.2.2 Exemplo de um *script* OTcl

Como mencionado anteriormente, no NS, uma simulação é formada basicamente por nodos, enlaces, agentes e geradores de tráfego. Toda a simulação depende de um *script* OTcl, que define o cenário da simulação (topologia e eventos). Este *script* pode escrever arquivos de rastro e inicializar o Nam para que se possa visualizar a simulação.

A Figura 3.2 reproduz um *script* de uma simulação simples (baseado no manual do NS [FAL2001]), que define uma topologia com quatro nodos e dois agentes: um agente UDP com um gerador de tráfego CBR (*Constant Bit Rate*) e um agente TCP com gerador FTP. A simulação é executada por 3 segundos e tem dois arquivos de rastro: out.tr e out.nam. Quando a simulação completa 3 segundos, é executado o Nam para a visualização da simulação.

```
#O símbolo # indica um comentário em Otcl
#Comando que cria um objeto simulador
set ns [new Simulator]                  ;#Inicializa a simulação

#Abertura de um arquivo para escrita que será usado para o rastro
set f [open out.tr w]                   ;#Abre o arquivo out.tr
$ns trace-all $f                        ;#Escreve todos os dados relevantes da simulação
                                        #no arquivo out.tr
#Abertura de um arquivo para escrita, usado para o rastro do Nam
set nf [open out.nam w]                 ;#Abre arquivo out.nam
$ns namtrace-all $nf                    ;#Escreve todos os dados relevantes para o nam
                                        #no arquivo out.nam

#Definição da topologia: quatro nodos (n0, n1, n2, n3)
#       n0
#         \
#      5Mb \
#      2ms  \
#            \
#             n2 --------- n3
#            /     1.5Mb
#      5Mb  /       10ms
#      2ms /
#         /
#       n1
#

#Novo objeto nodo é criado com o comando "$ns node". O código abaixo cria quatro nodos e
#os indica aos manipuladores n0, n1, n2 e n3
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#As linhas abaixo conectam os quatro nodos de acordo com o desenho acima. A sintaxe do
#comando é: $ns duplex-link <node1> <node2> <bw> <delay> <qtype>. Comando cria um enlace
#bi-direcional entre node1 e node2, com largura de banda <bw>, delay <delay> e tipo de
#fila <qtype>
$ns duplex-link $n0 $n2 5Mb 2ms DropTail
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 1.5Mb 10ms DropTail

#Criação de alguns agentes
set udp0 [new Agent/UDP]                ;#Agente UDP
$ns attach-agent $n0 $udp0              ;#anexado ao nodo n0
set cbr0 [new Application/Traffic/CBR]  ;#Gerador de tráfego CBR
$cbr0 attach-agent $udp0                ;#anexado ao agente UDP
$cbr0 set packetSize_ 500               ;#Tamanho do pacote = 500 bytes
$cbr0 set interval_ 0.005               ;#Pacotes serão enviados a cada 0.005 segundos
                                        #(isto é 200 pacotes por segundo)

set null0 [new Agent/Null]              ;#Agente que recebe o tráfego
$ns attach-agent $n3 $null0             ;#anexado ao nodo n3
```

FIGURA 3.2 - Exemplo de um *script* de simulação (continua)

```
$ns connect $udp0 $null0                ;#Conexão dos dois agentes criados acima
$ns at 1.0 "$cbr0 start"                ;#Inicia a geração do tráfego CBR em 1.0

puts [$cbr0 set packetSize_]            ;#Escreve na tela o tamanho do pacote
puts [$cbr0 set interval_]              ;#Escreve na tela o intervalo entre os pacotes

#Criação do agente TCP
set tcp [new Agent/TCP]
$ns attach-agent $n1 $tcp               ;#anexado ao nodo n1

#Criação do agente TCPSink que recebe o tráfego
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink              ;#anexado ao nodo n3

#TCP não gera seu próprio tráfego, criação de gerador FTP
set ftp [new Application/FTP]
$ftp attach-agent $tcp                  ;#ftp anexado ao agente tcp
$ns at 1.2 "$ftp start"                 ;#inicio da geração de tráfego ftp em 1.2

$ns connect $tcp $sink                  ;#Conecta agente tcp com o agente sink

#No tempo 1.35 os agentes tcp e sink são retirados dos seus respectivos nodos
$ns at 1.35 "$ns detach-agent $n1 $tcp ; $ns detach-agent $n3 $sink"

#A simulação é executada por três segundos
#A simulação chega ao fim quando o escalonador invoca o procedimento finish abaixo
#Este procedimento fecha todos os arquivos rastro e invoca o Nam em um dos arquivos
#rastro(out.nam)

$ns at 3.0 "finish"
proc finish {} {
        global ns f nf
        $ns flush-trace
        close $f
        close $nf

        puts "running nam..."
        exec nam out.nam &
        exit 0
}

#Simulação é iniciada com o comando abaixo
$ns run
```

FIGURA 3.2 - Exemplo de um script de simulação (continuação)

3.3 Descrição de classes

Nesta seção, serão apresentadas as classes que fazem a ligação de OTcl e C++, uma hierarquia parcial do NS, assim como a descrição de classes que têm maior relevância a esse trabalho. O texto desta seção baseia-se, principalmente, no manual do NS [FAL2001], sendo que serão apresentados apenas elementos essenciais para o entendimento da implementação de sistemas distribuídos em cenários com defeitos, relatada nos capítulos 4 e 5. Ao reproduzir aqui aspectos do manual, que é extenso e algumas vezes complicado de entender, pretende-se economizar trabalho do leitor em consultar diretamente esta fonte, e tornar a presente dissertação auto-contida.

3.3.1 Hierarquia de classes parcial do NS

A Figura 3.3 mostra uma hierarquia parcial do NS, para ilustrar sua hierarquia de classes. As classes: `Simulator`, `Node` e `Link`, por exemplo, são independentes das demais, por isso foram omitidas na figura.

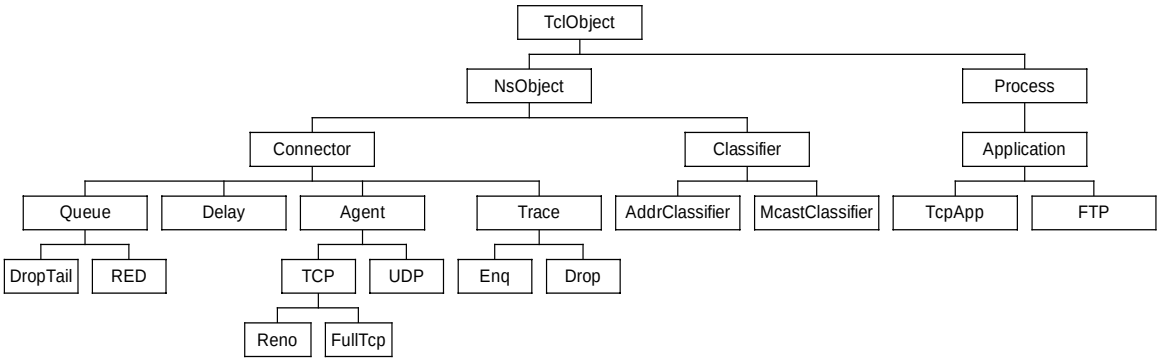


FIGURA 3.3 - Hierarquia parcial de classes do NS

3.3.2 Classes que fazem a ligação de OTcl e C++

O simulador suporta uma hierarquia de classes em C++ (também chamada de **hierarquia compilada**) e uma hierarquia de classes similar dentro do interpretador OTcl (também chamada de **hierarquia interpretada**). Essas duas hierarquias estão fortemente relacionadas: na perspectiva do usuário, existe uma correspondência biunívoca (de um para um) entre as classes da hierarquia compilada e da interpretada.

Pode-se criar novos objetos simuladores através do interpretador OTcl; esses objetos são instanciados dentro do interpretador e estão ligados a um objeto correspondente na hierarquia compilada. A hierarquia interpretada é automaticamente estabelecida através de métodos definidos na classe `TclClass`. Objetos instanciados pelo usuário são espelhados através de métodos definidos na classe `TclObject`.

Serão apresentadas a seguir classes que fazem a ligação entre OTcl e C++: a classe `Tcl` contém os métodos que o código C++ irá usar para acessar o interpretador OTcl; a classe `TclObject` é a classe base para todos os objetos do simulador que também serão espelhados na hierarquia compilada; a classe `TclClass` define a hierarquia de classes interpretada e os métodos que permitem ao usuário instanciar `TclObjects`; a classe `EmbeddedTcl` contém os métodos que fazem a inicialização do *script* do NS.

Classe Tcl

É a Classe C++ que encapsula a instância do interpretador OTcl e fornece métodos para acessar e se comunicar com este interpretador. Esses métodos podem executar as seguintes operações:

- obter uma referência para a instância OTcl;
- invocar procedimentos OTcl através do interpretador;
- retornar resultados para o interpretador;
- relatar ocorrência de erros;
- armazenar e consultar `TclObjects` (`TclObject` é um objeto particular que está ou na classe `TclObject` ou em uma de suas subclasses);
- obter acesso direto ao interpretador.

O programador deve **obter uma referência** para a instância da classe `Tcl` a fim de acessar os métodos (que serão apresentados a seguir) da classe `Tcl`. Para acessar essa instância é usado o comando: `Tcl& tcl = Tcl::instance()`.

Existem quatro métodos diferentes para **invocar um comando** OTcl através da instância `tcl`, que diferem basicamente nos seus argumentos:

- `tcl.eval(char* s)`: invoca o método `Tcl_GlobalEval()` para executar o *string* `s` através do interpretador;
- `tcl.evalc(const char* s)`: copia o *string* `s` para seu *buffer* interno e invoca `tcl.eval(char* s)`; isto faz com que o *string* `s` seja preservado;
- `tcl.eval()`: assume que o comando já esteja armazenado na variável interna `bp_`; este método invoca `tcl.eval(char bp_)`;
- `tcl.evalf(const char*s,...)`: método com sintaxe semelhante a função "`printf`".

Cada método passa um *string* para o interpretador, que o avalia em um contexto global. Esses métodos irão retornar para o método que os chamou se o interpretador retornar TCL_OK. Por outro lado, se o interpretador retornar TCL_ERROR, um método para tratamento de erros é chamado. A Figura 3.4 mostra o uso desses métodos.

```
Tcl& tcl = Tcl::instance();
char wrk[128];
strcpy(wrk, "Simulator set NumberInterfaces_ 1");
tcl.eval(wrk);

sprintf(tcl.buffer(), "Agent/SRM set requestFunction_ %s", "Fixed" );
tcl.eval();

tcl.evalc("puts stdout hello world");

tcl.evalf("%s request %d %d", name_, sender, msgid);
```

FIGURA 3.4 - Exemplos de métodos para invocar um comando OTcl

Quando o interpretador invoca um método C++, o **resultado é retornado** para a variável privada `tcl_->result`. Existem dois métodos que fazem esse retorno: `tcl.result(const char* s)` e `tcl.resultf(const char* s, ...)`. Esses dois métodos, que diferem basicamente em seus argumentos, passam o resultado do *string s* de volta para o interpretador. O método `tcl.error(const char* s)` **relata ocorrência de erros** no código compilado. Este método escreve o *string s* e `tcl_->result` em *stdout*.

O NS **armazena** uma referência para cada `TclObject` da hierarquia compilada em uma tabela *hash*, o que permite rápido acesso a esses objetos. Esta tabela é interna ao interpretador e o NS usa o nome do `TclObject` como sua chave. Os métodos usados para inserir, procurar e remover um determinado `TclObject` (exemplificado como "o") são:

- `tcl.enter(TclObject* o)` – insere um ponteiro para `TclObject` "o" na tabela *hash*;
- `tcl.lookup(char* s)` – retorna o objeto com o nome "s";
- `tcl.remove(TclObject* o)` – apaga referências para o `TclObject` "o" da tabela *hash*.

Classe TclObject

A classe `TclObject` é a superclasse da maioria das outras classes nas hierarquias compilada e interpretada. Para todo objeto da classe `TclObject`[1] criado pelo usuário de dentro do interpretador, um objeto "sombra" equivalente é criado na hierarquia compilada. Os dois objetos estão fortemente associados. A classe `TclClass`, que será descrita na próxima seção, contém os mecanismos para executar esse "sombreamento".

A Figura 3.5 mostra dois exemplos de configuração de um `TclObject`: um agente SRM (*Scalable Reliable Multicast*) da classe `Agent/SRM/Adaptative` desenvolvido no NS para implementar *multicast* confiável; e um agente TCP da classe `Agent/TCP/FullTcp`.

```
set srm [new Agent/SRM/Adaptative]          ;#criação do TclObject (srm)
$srm set packetSize_ 1024                   ;#configuração do limite de uma variável

$srm traffic-source $s0                     ;#objeto interpretado, invocando um método
                                            #C++ (traffic-source)como se ele fosse um
                                            #procedimento da instância


set tcpf [new Agent/TCP/FullTcp]           ;#criação do TclObject (tcpf)
$tcpf listen                               ;#objeto interpretado, invocando um método
                                           #C++ (listen) como se ele fosse um
                                           #procedimento da instância
```

FIGURA 3.5 - Exemplos de configuração de um TclObject

Por convenção, no NS, o símbolo "/" significa delimitador de hierarquia: a classe `Agent/TCP/FullTcp` é uma subclasse de `Agent/TCP`, que é subclasse de `Agent` que, por sua vez, é subclasse de `TclObject` na hierarquia interpretada. A hierarquia de classes compilada correspondente é `FullTcpAgent`, derivada de `TcpAgent`, derivada de `Agent`, que é derivada de `Tclobject`.

Como foi visto nos exemplos da Figura 3.5, usa-se `new` para criar um `TclObject`. O procedimento `new{}` pode ser usado para criar objetos de qualquer classe. Na criação de um `TclObject`, com o uso de `new{}`, o usuário cria um `TclObject` interpretado e o NS é responsável pela criação de automática do objeto compilado.

Embora `TclObject` seja a superclasse da maioria das classes do NS, classes como `Simulator`, `Node`, `Link` e `rtObject` não são derivadas de `TclObject`. Portanto, seus objetos não são `TclObjects`, apesar de também serem instanciados usando o procedimento `new`.

O NS tem cinco tipos de variáveis: inteiro (intvar), lógico (boolvar), real (realvar), variável de tempo (timevar) e variável de largura de banda (bwvar). É possível fazer uma **ligação** bidirecional entre as **variáveis** da hierarquia compilada e da interpretada, permitindo que ambas acessem os mesmos dados. Quando um valor de uma das variáveis é modificado, o valor da variável correspondente na outra hierarquia também é modificado. O exemplo da Figura 3.6 mostra o construtor da classe `ASRMAgent` (classe `Agent/SRM/Adaptive` na hierarquia interpretada) o qual contém métodos `bind()`, que fazem essa ligação entre as variáveis das duas hierarquias. Pode ser observado que os tipos real e inteiro são ligados com o método `bind` e os outros tipos usam `bind_<tipo>`.

[1] Nas novas versões do NS (observado a partir da versão 2.1b7), este objeto foi renomeado para SplitObject, mas no decorrer deste texto o termo TclObject continuará sendo usado.

```
ASRMAgent::ASRMAgent () {
     bind(“pdistance_”, &pdistance);      /* variável real */
     bind(“requestor_”, &requestor);      /* variável inteira */
     bind_time(“lastSent_”, & lastSent);  /* variável de tempo */
     bind_bw(“ctrlLimit_”, & ctrlLimit);  /* variável de largura de banda */
     bind_bool(“running_”, &running);     /* variável lógica */
}
```

FIGURA 3.6 - Ligação entre variáveis da hierarquia compilada e da interpretada

Todas as funções do exemplo da Figura 3.6 têm dois argumentos: o nome de uma variável OTcl e o endereço da variável compilada correspondente. Cada variável, que é ligada, é automaticamente inicializada com valores predefinidos quando o objeto é criado. Os valores predefinidos são especificados como variáveis da classe interpretada.

Para cada `TclObject` que é criado, o NS estabelece o procedimento OTcl `cmd{}` como um gancho para execução de métodos através do objeto compilado “sombra”. O procedimento `cmd{}` invoca o método C++ `command()` do objeto “sombra” automaticamente. O método `command()` faz a **ligação entre o método** OTcl e sua implementação em C++.

Pode-se invocar `cmd{}` de duas formas: explicitamente, especificando a operação desejada como primeiro argumento; ou implicitamente, como se houvesse um procedimento da instância com o mesmo nome da operação desejada. A maioria dos *scritps* usa a forma implícita, que será explicada primeiro.

Considerando-se que um cálculo de distância do agente SRM seja feito pelo objeto compilado e seja geralmente usado pelo objeto interpretado, o procedimento `distance?` no *script* é invocado da seguinte forma:

```
$srm distance? <agentAddress>
```

Se não existir um procedimento da instância chamado `distance?`, o interpretador irá invocar o procedimento `unknown{}` definido na superclasse `TclObject`. O procedimento `unknown{}`, então, invoca o procedimento `cmd{}` para executar a operação através do método `command()` do objeto compilado, da seguinte forma:

```
$srm cmd distance? <agentAddress>
```

Além desta forma, o usuário pode invocar **explicitamente** a operação, fazendo uma sobrecarga com o uso de um procedimento com o mesmo nome (nesse caso `distance?`), como na Figura 3.7.

```
Agent/SRM/Adaptive instproc distance? addr {
            $self instvar distanceCache_
            if ![info exists distanceCache_($addr)] {
                   set distanceCache_($addr) [$self cmd distance? addr}]
            }
            set distanceCache_($addr)
}
```

FIGURA 3.7 - Exemplo de invocação explícita do método `cmd{}`

A Figura 3.8 mostra um exemplo do método `command()` da classe `ASRMAgent`.

```
int ASRMAgent::command(int argc, const char*const*argv) {
     Tcl& tcl = Tcl::instance();
     if(argc ==3) {
            if (strcmp(argv[1], "distance?") ==0) {
                   int sender = atoi (argv[2]);
                   SRMinfo* sp = get_state(sender);
                   tcl.resultf("%f", sp->distance_);
                   return TCL_OK;

            }
     }
     return SRMAgent::command(argc, argv);
}
```

FIGURA 3.8 - Exemplo do método `command()`

O método `command()` é chamado com dois argumentos: `argc` e `argv`. O primeiro indica a quantidade de argumentos especificados na linha de comando do interpretador. O segundo (vetor `argv`) é composto de:

- `argv[0]` – contém o nome do procedimento – "`cmd`"
- `argv[1]` – especifica a operação desejada
- se o usuário especificar argumentos, estes serão colocados em `argv[2 ... (argc – 1)]`.

Esses argumentos são passados pelo interpretador como *strings* e devem ser convertidos para os tipos de dados apropriados. No exemplo da Figura 3.8, o `argv[2]` foi convertido para um inteiro pela função `atoi()`. Se a operação passada em `argv[1]` é encontrada dentro do método `command()`, deve-se retornar o resultado da sua operação. No caso do exemplo da Figura 3.8, a operação `distance?` está no método `command()` e retorna seu resultado através do método `tcl.result("%f", sp->distance_)`, que retorna para o interpretador o valor de `sp->distance` em um *string* criado internamente. O método `command()` deve retornar TCL_OK ou TCL_ERROR para indicar se houve sucesso ou algum erro, respectivamente. Se a operação não for encontrada no método desta classe, a classe deve invocar o método `command()` da sua superclasse e retornar o resultado correspondente. Todo esse processo é exemplificado na Figura 3.9.

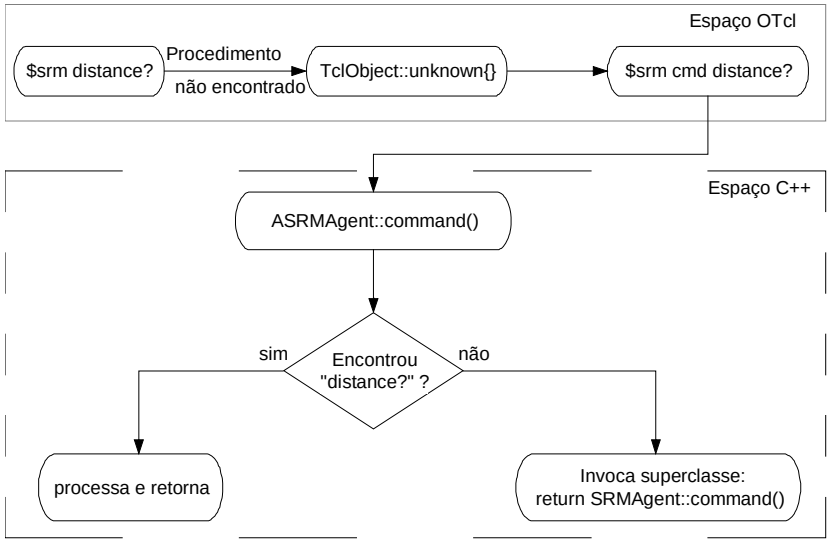


FIGURA 3.9 - Ligação de métodos OTcl e C++

Classe TclClass

As classes que são derivadas da superclasse `TclClass` têm duas funções: construir a hierarquia de classes interpretada para espelhar a hierarquia compilada; e fornecer métodos para instanciar novos `TclObjects`. Cada classe derivada da classe `TclClass` é associada a uma classe compilada particular e pode instanciar novos objetos da classe associada. Como exemplo (Figura 3.10), tem-se a classe `TcpClass` que é derivada da classe `TclClass` e está associada à classe `TcpAgent`.

```
static class TcpClass : public TclClass {
     Public:
     TcpClass() : TclClass("Agent/TCP") {}
     TclObject* create(int argc, const char*const* argv) {
            return (new TcpAgent());
     }
} class_tcp;
```

FIGURA 3.10 - Exemplo de como fazer o espelhamento entre as hierarquias do NS

A classe `TcpClass` define somente seu construtor e um método adicional `create` para criar instâncias do `TclObject` associado. O construtor especifica a classe interpretada explicitamente como, no exemplo, `Agent/TCP`. O construtor também especifica a hierarquia de classes interpretada implicitamente. No exemplo, o construtor cria duas classes, a classe `Agent/TCP` e sua superclasse `Agent`, que é subclasse de `TclObject`.

A classe `TcpClass` é associada à classe `TcpAgent` e cria novos objetos nesta classe associada. O método `TcpClass::create` retorna um `TclObject` da classe `TcpAgent`. Quando o usuário especifica `new Agent/TCP` no *script*, a rotina `TcpClass::create` é invocada. A Figura 3.11 mostra o espelhamento da classe TCP no NS.

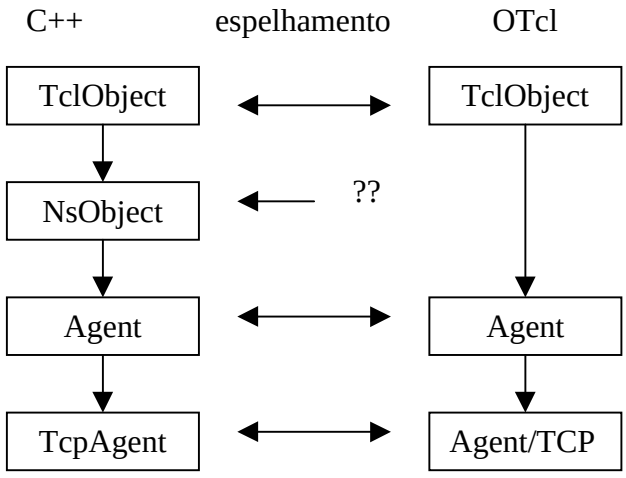


FIGURA 3.11 - Espelhamento da classe TCP

Classe EmbeddedTcl

O NS permite o desenvolvimento de funcionalidades tanto no código compilado quanto no código interpretado. O código interpretado é avaliado e carregado na inicialização do NS. A carga e a avaliação são feitos através de objetos da classe `EmbeddedTcl`.

A maneira mais simples de se estender o NS é adicionando código OTcl no diretório ~tclcl/tcl-object.tcl ou no diretório ~ns/tcl/lib. No último caso, o NS busca ~ns/tcl/lib/ns-lib.tcl automaticamente e portanto o programador deve adicionar algumas linhas nesse arquivo para que seu *script* também seja buscado automaticamente na inicialização do NS. No caso do arquivo ~ns/tcl/mcast/srm.tcl, que define alguns procedimentos para execução do agente SRM, o ~ns/tcl/lib/ns-lib.tcl tem a linha `source tcl/mcast/srm.tcl` que automaticamente busca srm.tcl na inicialização do NS.

Existem três pontos que devem ser considerados no código da `EmbeddedTcl`. Primeiro, se o código contém um erro que é detectado durante a avaliação, o NS não será executado. Segundo, o usuário pode explicitamente sobrescrever qualquer código *script*. Terceiro, depois que o usuário adicionar seus *scripts* no ~ns/tcl/lib/ns-lib, toda a vez que ele os modificar, terá que recompilar o NS para que suas mudanças tenham efeito.

3.3.3 Classe Simulator

A classe `Simulator` descreve todo o simulador, fornecendo um conjunto de interfaces para a configuração de uma simulação e para a escolha do tipo de escalonador de eventos que será usado. Um *script* de simulação geralmente começa com a criação de uma instância desta classe e com chamadas a vários métodos para: criação de nodos, topologias e configuração de outros aspectos da simulação. Quando um novo objeto dessa classe é criado, seu procedimento de inicialização cria um agente `Null`, que é utilizado para destino de pacotes que não são contados ou armazenados na simulação.

3.3.4 Classe Node

Cada simulação precisa de uma instância da classe `Simulator` para controlar e operar a simulação. Essa classe fornece procedimentos para criar e gerenciar a topologia e armazena referências para cada elemento desta topologia. Para a criação de um nodo, precisa-se usar procedimentos da classe `Simulator`. A primitiva básica para criação de um nodo é mostrada na Figura 3.12.

```
set ns [new Simulator]       ;#criação de uma instância da classe Simulator
$ns node                     ;#criação de um nodo
```

FIGURA 3.12 - Criação de um nodo

O número de identificação (*id*) de um nodo é automaticamente incrementado e atribuído a cada nodo que é criado através do procedimento `$ns node`. Esse procedimento constrói um nodo com objetos `classifier` (componentes de um nodo) simples. Cada nodo criado é adicionado ao vetor `Node_` indexado pelo *id* do nodo. A classe `Node` é uma classe OTcl independente, entretanto a maioria de seus componentes são `TclObjects`. A estrutura típica de um nodo *unicast* é mostrada na Figura 3.13.

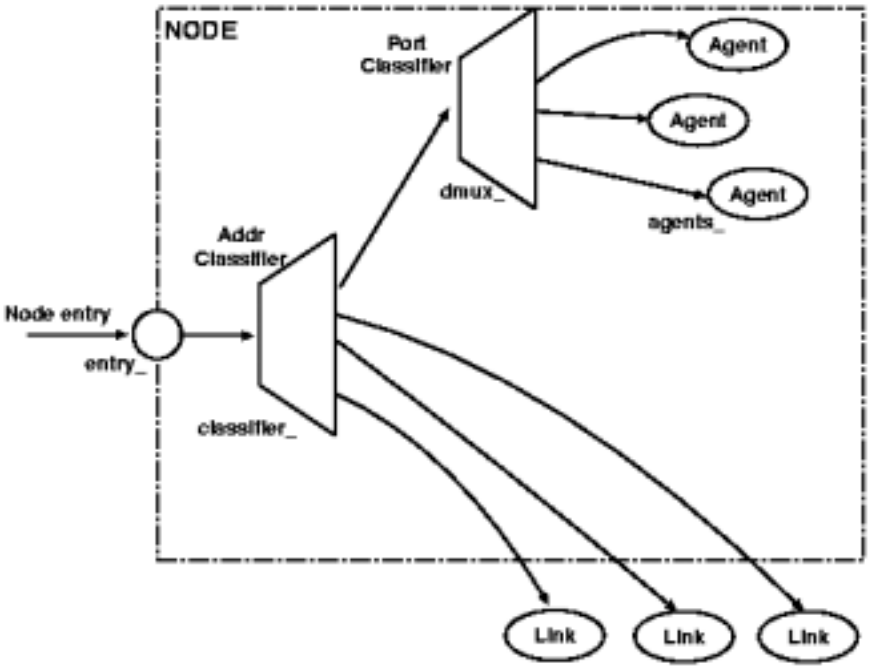


FIGURA 3.13 - Estrutura de um nodo *unicast* [FAL2001]

O nodo-padrão do NS é do tipo *unicast*; para criar nodos de simulação do tipo *multicast* necessita-se habilitar o *multicast* antes da criação dos nodos. Um exemplo da habilitação de *multicast* é na criação do objeto `Simulator`: `set ns [new Simulator -multicast on]`, assim todos os nodos da simulação passam a ser do tipo *multicast*. A estrutura de um nodo *multicast* é mostrada na Figura 3.14.

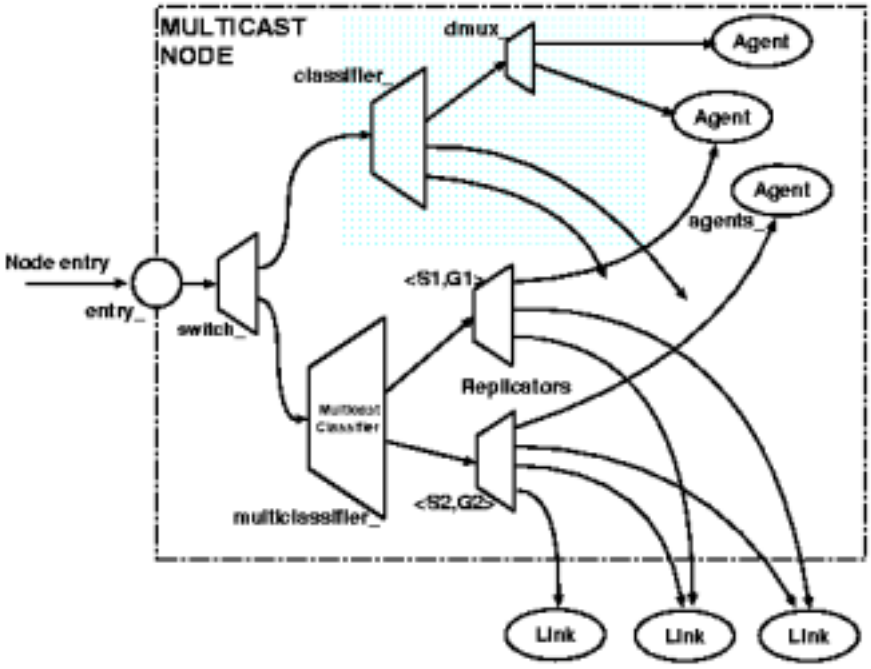


FIGURA 3.14 - Estrutura de um nodo *multicast* [FAL2001]

O primeiro elemento a manipular os pacotes que chegam até o nodo varia, dependendo do tipo do nodo. Em nodos *unicast*, o primeiro elemento é o classificador

de endereço (`Addr Classifier`), que analisa os bits mais significativos do endereço de destino do pacote. A variável `classifier_` (Figura 3.13) contém uma referência para esse classificador (*classifier*). Para nodos *multicast*, o ponto de entrada é referenciado por `switch_` (Figura 3.14) que analisa o primeiro bit do endereço e decide se deve passar o pacote para o classificador *unicast* ou para o *multicast*.

O classificador de endereço *unicast* analisa os bits do endereço de destino do pacote recebido, para saber se este pacote deve ser enviado para o `Port Classifier` (endereço de um agente que está no nodo) ou para o enlace (endereço de um agente de outro nodo).

O procedimento OTcl `attach{}` irá adicionar um agente a uma lista (`agents_`) de agentes do nodo. Esse procedimento também tem função de:

- atribuir ao agente um número de porta;
- estabelecer endereço de origem do agente;
- estabelecer o alvo do agente, que é a entrada `entry_` do nodo ao qual ele está anexado;
- adicionar um ponteiro do demultiplexador de porta (`dmux_`) para o agente no *slot* correspondente do classificador `dmux_`.

A função de um nodo, quando recebe um pacote, é examinar os seus campos, geralmente seu endereço de destino e, dependendo da ocasião, seu endereço de origem. No NS, essa tarefa é executada pelo objeto `classifier` simples. Objetos `classifier` múltiplos, cada um com a função de analisar uma porção específica do pacote, fazem com que o pacote ande através do nodo.

Um nodo, no NS, usa tipos diferentes de classificadores para funções diferentes. Cada classificador contém uma tabela de objetos de simulação indexada por número de *slot*. A função do `classifier` é determinar o número do *slot* associado com o pacote recebido e enviar este pacote para o objeto referenciado pelo *slot*. O classificador de endereço (`Addr Classifier`) é usado para dar suporte a envio de pacotes *unicast*.

O classificador *multicast* faz a classificação de pacotes de acordo com endereço de origem e de destino (grupo). Mantém uma tabela de mapeamento de pares origem/grupo para números de *slots*. Quando um pacote que chega tem o par origem/grupo desconhecido, o classificador invoca o procedimento `Node::new-group{}` e adiciona uma entrada na sua tabela.

3.3.5 Classe Link

Nessa seção, será feita a descrição da criação de enlaces entre nodos do tipo ponto-a-ponto. A classe `Link` é uma classe independente, inteiramente implementada em OTcl e tem poucas primitivas simples. A classe `Simulator` fornece o procedimento `simplex-link` que faz uma conexão unidirecional entre dois nodos como mostrado na Figura 3.15.

```
set ns [new Simulator]
$ns simplex-link <node0> <node1> <bandwidth> <delay> <queue_type>
```

FIGURA 3.15 - Conexão unidirecional entre dois nodos

O comando `simplex-link` cria um enlace do `node0` para o `node1`, com a largura de banda `<bandwidth>`, o atraso `<delay>` e o tipo de fila `<queue_type>`. O procedimento `duplex-link` constrói um enlace bidirecional a partir de dois `simplex-link`, um do `node0` para o `node1`, outro do `node1` para o `node0`. A sintaxe do `duplex-link` é a mesma de `simplex-link`. A classe `Link` fornece o procedimento `$link down` que possibilita simular a "queda" de um enlace. O procedimento `$link up` faz o enlace restabelecer-se.

3.3.6 Classe Agent

Agentes representam pontos finais onde pacotes da camada de rede são construídos ou consumidos. A classe `Agent` tem parte da implementação em OTcl e parte em C++. A classe C++ `Agent` inclui estados internos suficientes para determinar vários campos de um pacote simulado antes que este seja enviado. Estes estados incluem (conjunto incompleto):

- `agent_addr_` : endereço do agente que criou o pacote (endereço de origem dos pacotes);
- `agent_port_`: porta do agente que criou o pacote;
- `dst_addr_` : endereço de destino dos pacotes do agente;
- `dst_port_`: endereço da porta de destino dos pacotes do agente;
- `fid_`: identificador do fluxo IP;
- `prio_`: campo de prioridade de IP;
- `defttl_`: valor predefinido (*default*) de *ttl*;

Estes estados podem ser modificados por qualquer classe derivada de `Agent`. Nem todos eles são necessários para um agente particular.

A classe `Agent` dá suporte para geração e recepção de pacotes. A função `Packet* allocpkt()`, que aloca um novo pacote e atribui seus campos, é implementada por essa classe e geralmente não é sobrescrita por suas classes derivadas. As funções `void timeout(time number)` e `void recv(Packet*, Handler*)` geralmente são sobrescritas pelas classes que derivam de `Agent`.

O método `allocpkt()` é usado pelas classes derivadas de `Agent` para criar pacotes que serão enviados. O método `recv()` é o ponto de entrada principal para um agente que recebe pacotes.

Agentes podem ser criados em OTcl e seu estado interno pode ser modificado com o uso da função Tcl `set` ou por qualquer função Tcl que a classe `Agent` ou sua superclasse implemente. Alguns estados internos de um agente podem existir somente em OTcl e assim não são acessíveis diretamente de C++. O exemplo da Figura 3.16 ilustra a criação e a modificação de um agente em OTcl.

```
set newtcp [new Agent/TCP]              ;#cria um novo objeto (e sua sombra em C++)
$newtcp set window_ 20                  ;#estabelece janela do agente Tcp em 20
$newtcp target $dest                    ;#target é implementado na classe Connector
$newtcp set portID_ 1                   ;#portID existe somente em Otcl
```

FIGURA 3.16 - Criação e modificação de um agente em OTcl

A classe `Simulator` fornece o procedimento `simplex-connect` que faz uma conexão unidirecional entre dois agentes. A sintaxe desse procedimento é `$ns simplex-connect <src> <dst>`, no qual `<src>` é o agente de origem e `<dst>` é o agente destino.

O procedimento OTcl `simplex-connect{src dst}` atribui o `agent_addr_` (endereço do agente) do agente destino (`dst`) ao atributo `dst_addr_` (endereço de destino do agente) do agente de origem (`src`). Assim como atribui o `agent_port_` do objeto de destino ao `dst_port_` do objeto origem.

O procedimento `connect <src> <dst>` faz uma conexão bidirecional a partir de dois `simplex-connect`, um de `src` para `dst`, e outro de `dst` para `src`. A sintaxe do `connect` é a mesma de `simplex-connect`.

Criação de um novo Agente

Para criar um novo agente é necessário:

- decidir sua estrutura hierárquica e criar as definições de classe apropriadas;
- definir os métodos `recv()` e `timeout()`;
- definir funções que fazem a ligação com OTcl;
- escrever o código OTcl necessário para acessar o agente.

A decisão da **estrutura hierárquica** é uma escolha pessoal, mas deve estar relacionada com a camada que o agente irá operar. Um tipo de agente mais simples, não orientado à conexão, pode usar a classe `Agent/UDP` como base; geradores de tráfego podem ser conectados a agentes UDP. Para criação de protocolos orientados à conexão, vários tipos de TCP podem ser usados como superclasse. Se um novo protocolo de transporte for desenvolvido, então pode ser melhor escolher `Agent` como superclasse. O exemplo da Figura 3.17 ilustra a criação de um agente que executa operações de *ping*. Para esse exemplo, utilizou-se `Agent` como superclasse.

```
Class PING_Timer;

Class Ping : public Agent {
    public:
           Ping();
           Int command(int argc, const char*const* argv);
    protected:
           void timeout();
           void sendit();
           double interval_;
           PING_Timer ping_timer_;
};

class PING_Timer : public TimerHandler {
   public:
          PING_Timer(Ping *a) : TimerHandler() { a_ = a; }
   protected:
           virtual void expire(Event *e);
           Ping *a_;
};
```

FIGURA 3.17 - Criação de um agente *ping*

O método `recv()` não está definido nessa classe porque esse agente tem função de requisição e geralmente não recebe eventos ou pacotes. Assim o método `recv()` da superclasse `Connector` é usado (`Connector::recv()`). O método `timeout()` é usado

para enviar pedidos de pacotes periodicamente. O método `timeout()` é mostrado na Figura 3.18, juntamente com o método auxiliar `sendit()`. Nesse caso, o método `timeout()` faz com que `sendit()` seja executado a cada `interval_` segundos. O `sendit()` cria um novo pacote com a maioria de seus cabeçalhos já estabelecidos pelo `allocpkt()`, com exceção do *timestamp* corrente. A chamada a `access()` fornece uma interface para os campos do cabeçalho do pacote e é usada para estabelecer o campo *timestamp*. Esse agente usa um novo tipo de cabeçalho `PINGHeader`. Para enviar o pacote ao próximo nodo é invocado `Connector::send()`.

```
void Ping::timeout()
{
       sendit();
       ping_timer_.resched(interval_);
}
void Ping::sendit()
{
      Packet* p = allocpkt();
      PINGHeader *eh = PINGHeader::access(p->bits());
      eh->timestamp() = Scheduler::instance().clock();
      send(p, 0);     // Connector::send()
}
void PING_Timer::expire(Event *e)
{
      a_->timeout();
}
```

FIGURA 3.18 - Métodos do agente *ping*

Os métodos para fazer a **ligação entre OTcl e C++** foram descritos na seção 3.3.2. Aqui, esses métodos serão mostrados de forma resumida. Existem três itens que devem ser seguidos para ligar o agente `Ping` com OTcl: fazer mapeamento do nome OTcl, ligar variáveis e ligar métodos.

Primeiro necessita-se estabelecer um **mapeamento** entre o nome OTcl da classe `Ping` com o objeto real criado quando a instanciação da classe é requisitada em OTcl. A forma de execução é mostrada na Figura 3.19. Nessa classe, o objeto estático `class_ping` é criado e seu construtor (chamado no momento que o simulador é executado) coloca o nome da classe `Agent/Ping` dentro do espaço de nomes de OTcl. Lembra-se que, na hierarquia interpretada, o caracter "/" é o delimitador de hierarquia. A definição do método `create()` especifica como um objeto sombra C++ deve ser criado, quando o interpretador OTcl é instruído a criar um objeto da classe `Agent/Ping`. Neste caso, é retornado um objeto alocado dinamicamente.

```
static class PINGClass : public TclClass {
  public:
       PINGClass() : TclClass("Agent/Ping") {}
       TclObject* create(int argc, const char*const* argv) { return (new Ping()); }
} class_ping;
```

FIGURA 3.19 - Mapeamento entre o nome OTcl e o objeto real

Para fazer a **ligação entre as variáveis** C++ e as variáveis correspondentes em OTcl, utiliza-se o método `bind()` que está no construtor da classe `Ping` (Figura 3.20). Nesse construtor, as variáveis `interval_` e `size_` são ligadas às variáveis OTcl `interval_` e `packetSize_` respectivamente. Operações de leitura ou modificação das variáveis OTcl irão acarretar o acesso correspondente a variáveis C++.

```
Ping::Ping() : Agent(PT_PING)
{
     bind_time("interval_", &interval_);
     bind("packetSize_", &size_);
}
```

FIGURA 3.20 - Construtor da classe `Ping`

Depois da criação do objeto e da ligação das variáveis, precisa-se **ligar os métodos** que são implementados em C++ e que podem ser invocados de OTcl. Nesse exemplo, a diretiva `start` inicia a requisição do *ping*. Esta implementação é mostrada na Figura 3.21. Ali, o comando `start` disponível em OTcl chama a função C++ `timeout()` que inicia a geração do primeiro pacote e escalona o próximo.

```
 int Ping::command(int argc, const char*const* argv)
{
      if (argc == 2) {
             if (strcmp(argv[1], "start") == 0) {
                /*
                 * <agent> start
                 */
                timeout();
                return (TCL_OK);
             }
      }
      return (Agent::command(argc, argv));
}
```

FIGURA 3.21 - Método `command()` da classe `Ping`

O agente que é criado tem que ser instanciado e anexado a um nodo. O **código em OTcl** da Figura 3.22 mostra como executar essas funções. Assume-se que os objetos nodo (`node`) e simulador (`simulator`) já foram criados.

```
set pingagent [new Agent/Ping]
$ns attach-agent $node $pingagent
```

FIGURA 3.22 - Criação do agente Ping em OTcl

Agentes UDP

Agentes UDP aceitam dados de tamanhos variáveis de uma aplicação e, se necessário, fazem a segmentação de dados. No NS, pacotes UDP contêm um número de seqüência crescente, apesar de pacotes UDP reais não conterem números de seqüência. Essa característica pode ser útil para a análise de arquivos de rastro, além de não ocasionar sobrecarga na simulação.

Agentes TCP

No NS, existem dois tipos principais de agentes TCP: agentes unidirecionais e agentes bidirecionais. Os agentes unidirecionais são divididos em emissores e receptores e os agentes bidirecionais representam tanto emissores quanto receptores.

O simulador contém muitos tipos de abstrações de emissores TCP que tentam reproduzir comportamentos de controle de erros e de congestionamento, mas não têm a intenção de serem réplicas de implementações de TCP do mundo real. Os agentes receptores, denominados *sinks*, são responsáveis pelo envio de confirmações (ACKs) para o TCP emissor e geram um ACK para cada pacote recebido.

Um agente TCP não gera dados de aplicação, sendo para isso necessário conectar um módulo de geração de tráfego ao agente TCP. Duas aplicações usadas com freqüência para gerar tráfego com TCP são: FTP, que representa transferência de dados de grande tamanho, e Telnet, que faz a escolha do tamanho dos dados aleatoriamente de uma biblioteca. Para executar uma simulação com TCP, é necessário criar e configurar o agente, anexar um gerador de tráfego e, então, iniciar este gerador.

Os agentes bidirecionais são diferentes dos outros agentes TCP já que podem simultaneamente atuar como emissores e receptores. Atualmente, o NS tem dois tipos de agentes TCP bidirecionais: `FullTcp` e `BayFullTcp`.

Nesse trabalho, será usado somente `FullTcp` na construção do modelo de sistemas distribuídos. A criação e execução de um agente `FullTcp` é similar a de outros agentes TCP, sendo que é necessário criar e configurar o agente, anexar a ele um gerador de tráfego e iniciar esse gerador de tráfego. Entretanto, o agente que será o receptor deve ser colocado em estado de escuta através do método `listen`.

4 Simulações de sistemas distribuídos e de defeitos no NS

Nesse capítulo, serão especificados o modelo de sistemas distribuídos e o modelo de defeitos assumidos no trabalho, de acordo com alguns conceitos apresentados no capítulo 2. Também será apresentada a forma como esses modelos podem ser implementados no NS. Além disso, serão exibidos arquivos de *script* de simulação que foram desenvolvidos para facilitar a criação de sistemas distribuídos em cenários com e sem defeitos no NS.

4.1 Especificação dos modelos de sistemas distribuídos e de defeitos

Os modelos de sistemas distribuídos empregados nesse trabalho são baseados nos conceitos apresentados por Jalote [JAL94] e Lynch [LYN96], comentados no capítulo 2. Assim, é assumido que o modelo físico é formado por nodos e pela rede de comunicação como componentes básicos. A rede não é confiável, entretanto a camada de transporte pode fornecer protocolos como TCP para garantir que os dados sejam entregues na ordem em que foram enviados sem perdas ou duplicação. Também é assumido que a troca de mensagens é o único meio de comunicação entre os nodos.

No modelo lógico, a rede é tratada como se estivesse totalmente conectada, o que significa que qualquer nodo pode se comunicar com qualquer outro nodo da rede. Nesse modelo, não é importante a topologia da rede física que será empregada, podendo ser utilizada uma topologia arbitrária como, por exemplo, qualquer uma das topologias ilustradas na seção 2.1.1 ou uma combinação das mesmas. Além disso, assume-se que a aplicação distribuída é formada por um conjunto de processos, sendo que cada nodo tem um único processo[1]. Nesse capítulo e no capítulo 5, será usado o termo aplicação como sinônimo de processo; em outras palavras, para fazer referência a um processo que está sendo executado em um nodo, pode-se chamá-lo de aplicação.

O modelo de sistema síncrono e assíncrono é definido do ponto de vista de programação de acordo com as definições de Lynch apresentadas na seção 2.1.2. No modelo assíncrono, componentes separados executam passos em uma ordem arbitrária com velocidades relativas arbitrárias. Já, no modelo síncrono, é assumido que todos os tempos de comunicação e processamento não são apenas limitados, mas também fixos e iguais para quaisquer enlaces e nodos. Além disso, componentes executam passos simultaneamente, ou seja, a execução e a troca de mensagens é feita em rodadas síncronas.

O modelo de defeitos emprega a classificação de Cristan (seção 2.2.1), sendo que em caso de defeito de colapso, um processo pode ser recuperado, podendo estar tanto em estado de amnésia, quanto em estado de amnésia parcial após seu reinício. Adicionalmente, é usada a classificação de defeitos de comunicação definida por Hadzilacos e Toueg (seção 2.2.3), já que Cristian não classifica este tipo de defeitos.

[1] Na verdade, esta restrição diz respeito apenas aos processos-alvo de uma aplicação em partitular e que está em análise: ela ignora todos os demais que “sustentam” o restante das operações dos nodos.

Uma das limitações para simular um sistema distribuído no NS está na forma em que os agentes são conectados entre si através de `connect` (seção 3.3.6). Quando se utiliza este procedimento para conectar um agente a vários outros agentes, o endereço destino deste agente é sempre o último agente conectado a ele, já que `connect` sobrescreve o valor do agente destino das conexões anteriores. Considere um cenário trivial com três clientes e um servidor. Tem-se quatro agentes A0, A1, A2, A3 localizados em quatro nodos diferentes, sendo que A0, A1 e A2 devem enviar mensagens para A3 e A3 deve responder a essas mensagens (Figura 4.1). Assim são feitas as seguintes conexões em OTcl: 1) `$ns connect $A0 $A3`; 2) `$ns connect $A1 $A3`; 3) `$ns connect $A2 $A3`.

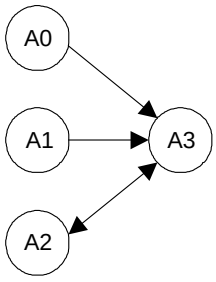


FIGURA 4.1 - Exemplo de conexões com a utilização de `connect`

O procedimento 1 fará com que o endereço de destino do agente A0 seja A3 e que o endereço de destino de A3 seja A0. Quando o procedimento 2 for chamado será atribuído A3 como endereço de destino de A1 e A1 como endereço de destino de A3, fazendo com que o endereço de destino anterior (A0) seja sobrescrito. O mesmo ocorre no procedimento 3. Depois dos três procedimentos, o endereço de destino dos agentes A0, A1 e A2 será o A3, mas o endereço destino de A3 será somente A2. Portanto A0, A1 e A2 enviam pacotes para A3, mas este irá responder somente para A2. A solução para esse problema, que inicialmente não foi encontrada por falta de clareza na documentação do NS, é apresentada na seção seguinte.

Neste trabalho, são apresentados dois modelos distintos através dos quais um sistema distribuído pode ser simulado [TRI2002], dependendo do protocolo de transporte utilizado. Um deles está baseado no protocolo TCP (confiável); o outro, no protocolo UDP (não confiável). Essas duas versões serão explicadas a seguir. Em ambas, cada nodo contém somente uma aplicação (processo que está na camada de aplicação).

4.1.1 Modelo baseado em TCP

Neste modelo, o transporte de dados é feito por agentes TCP. Cada nodo tem um agente TCP para cada conexão; sendo assim, um nodo pode ter até *n-1* agentes TCP, onde *n* é o número de nodos da rede, dependendo da conectividade lógica.

Cada agente TCP é associado a uma classe derivada da classe `Application`, chamada `TcpApp`, que é encarregada de fazer a emulação da transmissão de dados entre as aplicações (explicado na seção 4.2). Além disso, em cada instância da aplicação (`App`*n*), é mantida uma tabela de conexões, onde são armazenadas todas as aplicações às

quais ela está conectada, juntamente com a `TcpApp` encarregada de transmitir dados para a aplicação destino. Esta tabela é usada para solucionar a limitação do comando `connect`, explicada na seção anterior. A Figura 4.2 mostra a distribuição destes agentes e aplicações dentro de um nodo e suas conexões. Ali estão representados três nodos, sendo que cada nodo troca dados com os outros dois. `App`*n* representa um processo da aplicação distribuída, onde `TcpApp` é a classe responsável por enviar os dados e `Tcp` é o agente de transporte.

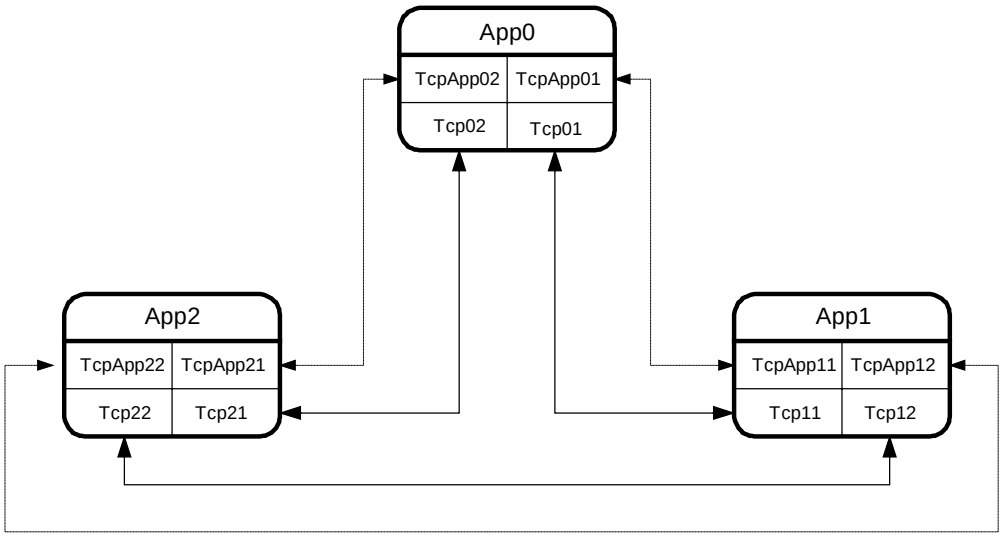


FIGURA 4.2 - Estrutura do modelo baseado em TCP, ilustrando uma topologia lógica em anel

Nesse modelo, a comunicação é bidirecional e feita por meio de *unicast*. O agente TCP usado é um agente bidirecional do NS chamado `FullTcp`. Para a transmissão de dados, estes percorrem o seguinte fluxo: a aplicação (por exemplo, `App0`) passa os dados à sua `TcpApp`, que os encaminha ao agente TCP. Este, por sua vez, os envia para o agente TCP da aplicação a qual está conectado. Quando um agente TCP recebe dados, ele os entrega à `TcpApp`, que os repassa à aplicação destinatária (`App1`).

4.1.2 Modelo baseado em UDP

Nesse modelo, o transporte de dados é feito através de agentes UDP e a comunicação pode ser feita tanto por meio de *unicast* quanto por *multicast*. A comunicação não é confiável, pois utiliza UDP, mas tem a vantagem de oferecer também *multicast* ao contrário do modelo baseado em TCP. Nesse trabalho, será utilizado UDP somente quando for necessário *multicast*, portanto um modelo com UDP e *unicast* não será enunciado.

No caso de *multicast*, cada agente UDP envia dados para o endereço de um grupo de destinatários, sendo que cada nodo tem um agente UDP para cada grupo. Se o nodo tem *n* grupos destino são necessários *n* agentes UDP. Na Figura 4.3, por exemplo, `App0` (através de seu agente `Udp0`) envia dados para um grupo que é composto por `Udp1` e `Udp2`. O fluxo de dados é o seguinte: a aplicação (`App0`) passa os dados para seu agente UDP (`Udp0`) que, por sua vez, os envia para os agentes UDP de seu grupo-destino. Quando um agente UDP recebe dados, repassa-os para sua aplicação.

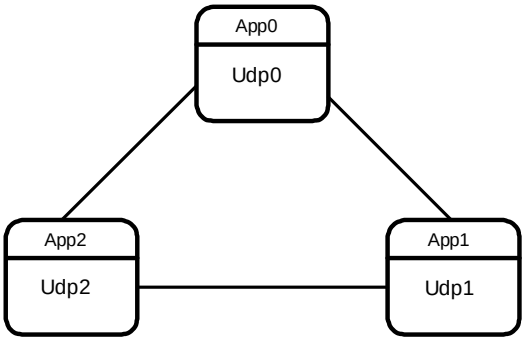


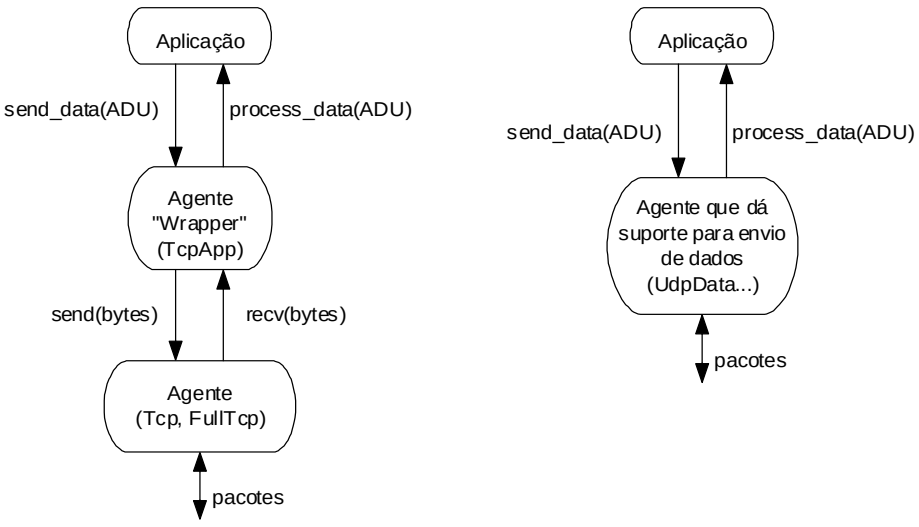
FIGURA 4.3 - Estrutura do modelo baseado em UDP

4.2 Implementação dos modelos de sistema distribuído no NS

No NS, aplicações ficam localizadas sobre os agentes de transporte como TCP e UDP. As aplicações básicas dividem-se em dois tipos: geradores de tráfego, como geradores CBR e aplicações simuladas, como FTP. Essas aplicações, na realidade, não transmitem dados; somente preocupam-se com o tamanho do dado e seu tempo de transmissão. Para transmitir dados reais entre aplicações no NS, é preciso utilizar uma estrutura uniforme para que se possa passar dados entre aplicações, bem como para passar dados entre aplicações e agentes de transporte. Esta estrutura é composta por três componentes:

- uma unidade de dados em nível de aplicação ADU (*aplication-level data unit*);
- uma interface comum para transmissão de dados entre aplicações e;
- dois mecanismos para passagem de dados entre aplicações e agentes de transporte: um para agentes UDP e outro para TCP.

A Figura 4.4 ilustra cada um desses componentes, que serão explicados a seguir.



a) com TCP b) com UDP

FIGURA 4.4 - Fluxo de dados da aplicação [FAL2001]

A função da **ADU** é “empacotar” os dados em nível de aplicação, sendo que, para transmitir dados, é preciso implementar uma classe derivada da classe abstrata base `AppData` (definida em ns-process.h), reescrevendo-se o construtor e os métodos `size()` e `copy()`. Então, deve-se adicionar o tipo de dado definido pelo usuário na classe derivada de `AppData` à enumeração `AppDataType` contida em ns-process.h. Exemplos de como implementar classes derivadas de `AppData` e de todo o processo para transmitir dados entre aplicações serão apresentados no capítulo 5.

A **interface** comum para transmitir dados entre aplicações é fornecida pela classe `Process` (definida em ns-process.h). Esta classe permite que aplicações passem e requisitem dados de outras aplicações. As funcionalidades básicas dessa classe são: processar dados recebidos, enviar dados para outra entidade e requisitar dados de outra entidade. Os métodos `send_data(ADU)` e `process_data(ADU)` da Figura 4.4 pertencem a esta classe e são responsáveis pelo envio e processamento de dados, respectivamente. A classe `Process` é a superclasse de todas as aplicações que enviam dados, por isso uma aplicação desenvolvida pelo usuário deve ser derivada dessa classe.

Para **transmissão de dados sobre UDP**, precisa-se implementar uma nova classe derivada de UDP, adicionando-se um novo método `send(int, AppData*)`– Figura 4.5, linha 35, e sobrescrevendo-se o método `recv(Packet*, Handler*)`– Figura 4.5, linha 21. O método `send` inclui os dados da aplicação (ADU) na área de dados do pacote e chama o método `send` da classe `Agent` (Figura 4.5, linha 43) que envia esses dados para o agente da aplicação destino. Já o método `recv` acessa os dados do pacote recebido por esse agente de transporte e os envia para a aplicação na qual está anexado. Para isso, cada agente tem um ponteiro para sua aplicação (`AsFloodAppUdp` – no exemplo da Figura 4.5) que chama o método `process_data()`– Figura 4.5, linha 29. Este método faz o processamento dos dados recebidos pela aplicação. Um exemplo de definição dessa nova classe, denominada `UdpDataAgent` (implementada de acordo com os procedimentos para criação de um novo agente, apresentados na seção 3.3.6), cujos métodos foram definidos na Figura 4.5, é apresentado na Figura 4.6.

```
1:   // Criação do novo tipo de pacote
2:   static class UdpDataHeaderClass : public PacketHeaderClass {
3:      public:
4:          UdpDataHeaderClass() : PacketHeaderClass("PacketHeader/UdpData",
5:                                                   sizeof(hdr_udata)) {
6:              bind_offset(&hdr_udata::offset_);
7:           }
8:   } class_udpdatahdr;

9:   static class UdpDataClass : public TclClass {
10:     public:
11:          UdpDataClass() : TclClass("Agent/UDP/UdpData") {}
12:          TclObject* create(int, const char*const*) {
13:               return (new UdpDataAgent());
14:          }
15:  } class_udpdata_agent;

     // Ligação entre código C++ e Otcl
16:  UdpDataAgent::UdpDataAgent() : UdpAgent(PT_UDPDATA)
17:  {
18:        bind("udata_hdr_size_", &udata_hdr_size_);
19:  }
```

FIGURA 4.5 - Exemplo de métodos da classe `UdpDataAgent` (continua)

```
20:  // Método que recebe os dados
21:  void UdpDataAgent::recv(Packet *pkt, Handler*)
22:  {
23:       hdr_ip *ip = hdr_ip::access(pkt);
24:       if ((ip->saddr() == addr()) && (ip->sport() == port()))
25:              return;
26:       if (app_ == 0)
27:              return;
28:       hdr_udata *ih = hdr_udata::access(pkt);
29:       ((AsFloodAppUdp*)app_)->process_data(ih->size(), pkt->userdata());
30:       Packet::free(pkt);
31:  }

32:  // Método que envia os dados
33:  // realsize: tamanho declarado pelo usuário
34:  // datasize: tamanho real do dado do usuário, usado para alocar o pacote
35:  void UdpDataAgent::send(int realsize, AppData* data)
36:  {
37:       Packet *pkt = allocpkt(data->size());
38:       hdr_udata *ih = hdr_udata::access(pkt);
39:       ih->size() = data->size();
40:       pkt->setdata(data);
41:       hdr_cmn *ch = hdr_cmn::access(pkt);
42:       ch->size() = udata_hdr_size_ + realsize;
43:       Agent::send(pkt, 0);
44:  }
```

FIGURA 4.5 - Exemplo de de métodos da classe `UdpDataAgent` (continuação)

```
struct hdr_udata {
       int size_;
       int& size() {
          return size_;
       }

       // Métodos de acesso ao cabeçalho do pacote
       static int offset_;
       inline static int& offset() { return offset_; }
       inline static hdr_udata* access(const Packet* p) {
              return (hdr_udata*) p->access(offset_);
       }
};

// Criação da classe UdpDataAgent derivada de UdpAgent
class UdpDataAgent : public UdpAgent {
public:
       UdpDataAgent();

       Virtual void recv(Packet *, Handler *);
       Virtual void send(int realsize, AppData* data);

protected:
       int udata_hdr_size_;
};
```

FIGURA 4.6 - Exemplo de definição da classe `UdpDataAgent` e de um novo tipo de pacote

Além disso, na classe que será implementada a aplicação distribuída, precisa-se incluir, dentro do método `command`[1]`(int argc, const char*const* argv)`, o método `attachApp()` para anexar a aplicação ao agente UDP. A seguir, serão resumidos os passos para a transmissão de dados sobre UDP:

- derivar uma classe de `AppData` (implementação da ADU) e reescrever o construtor e os métodos `size()` e `copy()`;

[1] O método `command`, que faz a ligação entre um método OTcl e sua implementação em C++, é chamado quando é executado um comando OTcl para esta classe, como: `$app0 attach-udp $udp0` (assumindo-se que `app0` seja uma instância da classe da aplicação e `udp0`, uma instância de UdpDataAgent). A seção 3.3.2 apresenta maiores detalhes sobre o método `command`.

- derivar uma classe de UDP, adicionar um novo método `send`, sobrescrever `recv` e criar um novo tipo de pacote;
- criar a classe da aplicação (onde será implementado o protocolo a ser testado) e implementar o método `command` e o método `process_data`. Dentro do método `command`, anexar a aplicação ao agente com `attachApp()`. O método `process_data` fará a leitura do conteúdo dos pacotes recebidos pela aplicação. Nessa classe, deve-se, também, chamar o método `send` de `UdpDataAgent` passando-se os dados da ADU como parâmetro. Exemplos de como implementar esses métodos serão mostrados no capítulo 5;
- acrescentar novo tipo de dado à enumeração `AppDataType` em ns-process.h e novo tipo de pacote a packet.h.

Para simular a **transmissão de dados sobre TCP** no NS, além de desenvolver uma classe derivada de `AppData` (explicada anteriormente) para implementação da ADU, é preciso utilizar a classe `Application/TcpApp` (definida em tcpapp.h) que é o invólucro do agente (*Wrapper* - Figura 4.4 a) e fornece as seguintes funcionalidades: no emissor, é implementado um *buffer* que guarda os dados da aplicação (ADU); no receptor, é feita a contagem de *bytes* recebidos. Após receber todos os *bytes* da transmissão, o receptor passa a obter os dados diretamente do emissor. Assim, para emular a transmissão de dados sobre TCP, através da classe `TcpApp` é enviado somente o tamanho dos dados; o receptor, após contar todos os *bytes* da transmissão, busca os dados diretamente do *buffer* do emissor.

A classe `TcpApp` tem um ponteiro para um agente de transporte TCP (`FullTcp`, no caso do modelo apresentado nesse trabalho) e contém os seguintes métodos:

- `connect(TcpApp *dst)` para fazer uma conexão bidirecional com outro `TcpApp`;
- `send(int nbytes, AppData *data)` para fazer o envio de dados, onde `nbytes` tem o tamanho nominal dos dados da aplicação e `data` abriga os dados, que são passados em forma de ADU;
- `recv` que, após receber todos *bytes* da transmissão, chama `process_data(int size, AppData* data)` que, por sua vez, faz o tratamento dos dados recebidos.

A seguir serão resumidos os passos para transmissão de dados sobre TCP:

- derivar uma classe de `AppData` e reescrever o construtor e os métodos `size()` e `copy()`;
- criar a classe da aplicação (onde será implementado o protocolo a ser testado) e implementar os métodos `process_data`, que fará a leitura do conteúdo dos pacotes, e `command`. Dentro do método `command`, implementar `connect`, que associa um agente `TcpApp` (que irá enviar dados) à aplicação que irá receber os dados. Na classe da aplicação, deve-se, também, chamar o método `send` de `TcpApp`, passando-se os dados da ADU como parâmetro. Exemplos de como implementar esses métodos serão apresentados no capítulo 5;
- Acrescentar novo tipo de ADU a ns-process.h.

4.3 Simulação de defeitos no NS

Nessa seção, serão apresentadas possibilidades para simulação de defeitos em enlaces e nodos no NS, focalizando-se a simulação de colapso em nodos. Portanto, serão mostradas tanto as alternativas de simulação de defeitos de colapso em nodos que foram descartadas e as razões para isso, quanto a solução que será usada para implementar este tipo defeito.

4.3.1 Simulação de defeitos em enlaces

O NS fornece um modelo de erros que introduz perdas de pacotes em uma simulação. Com isso, é possível simular defeitos de **omissão** em enlaces. Esse modelo simula perdas em enlaces marcando um *flag* de erro do pacote ou descartando o pacote em um agente que armazena pacotes perdidos. Nas simulações, perdas podem ser geradas por um modelo simples como a taxa de erro do pacote ou por um modelo estatístico mais complexo.

Esse modelo é implementado na classe `ErrorModel` que é derivada de `Connector`. Se for definido um agente para armazenamento de pacotes perdidos, ele o fará com relação aos pacotes descartados. Se este agente não for definido, a classe `ErrorModel` somente marca o *flag* `error_` do cabeçalho do pacote, permitindo aos agentes manipular a perda. A classe `ErrorModel` implementa somente modelos baseados em taxas de perdas. Modelos de perda mais sofisticados podem ser implementados em C++, derivando-se esta classe. A Figura 4.7 exemplifica a criação de um modelo de erros com taxa de perda de pacote de 1% (0.01).

```
#criação do modelo de perda loss_module com taxa de perda de 1%
set loss_module [new ErrorModel]
$loss_module set rate_ 0.01

#define o agente que armazena os pacotes perdidos
$loss_module drop_target [new Agent/Null]

#define que as perdas devem ocorrer no enlace entre o nodo 0 (n0) e o nodo 2 (n2)
$ns lossmodel $loss_module $n0 $n2
```

FIGURA 4.7 - Exemplo de simulação de omissão em um enlace

A simulação de defeito de **colapso** em um enlace é obtida pelo uso do comando `$ns rtmodel-at <time> <op> <args>`, no qual é aplicada a operação especificada em `<op>` que pode ser *down* (“rompimento” do enlace) ou *up* (restabelecimento do enlace) em um enlace especificado em `<args>` no tempo `<time>`. A utilização desse comando com a operação *down* simula o colapso de um enlace especificado em `<args>` (Figura 4.8, linha 1). Após, pode-se fazer o restabelecimento do enlace com a operação *up* (Figura 4.8, linha 2). Assim, pode-se simular tanto defeitos permanentes (não desfazendo o defeito pelo restabelecimento do enlace) quanto defeitos temporários (se o enlace for restabelecido posteriormente).

```
$ns rtmodel-at 1.0 down $n1 $n2  ;#“queda” do enlace entre nodos n1 e n2 no tempo 1.0
$ns rtmodel-at 3.0 up $n1 $n2    ;#restabelecimento do enlace entre n1 e n2 em 3.0
$ns rtmodel-at 2.0 down $n3      ;#defeito do nodo n3 – faz com que  todos os enlaces
                                 #desse nodo sejam “rompidos”
```

FIGURA 4.8 - Exemplo do comando `rtmodel-at <time> <op> <args>`

4.3.2 Simulação de defeito de colapso em nodos

Analisando-se o NS, verificou-se que este não oferece possibilidade direta para simulação de defeitos de colapso em nodos. A documentação do NS indica a possibilidade de simular este tipo defeito, utilizando procedimento similar ao empregado para defeito de enlace: apenas, dentre os parâmetros (em `<args>`), é listado um nodo em vez do enlace (Figura 4.8, linha 3). Nesse caso, a semântica do NS para defeito de colapso em um nodo não é correta, pois a implementação está baseada apenas no colapso de todos os enlaces do nodo. Assim, ao utilizar-se essa opção, o nodo e todos os seus componentes podem prosseguir na execução de suas atividades e, conseqüentemente, podem vir a mudar seu estado interno. Portanto, o nodo não pára seu funcionamento. Se for recuperado – o que corresponderia a reconectar o nodo – será difícil antecipar o estado em que ele se encontrará. É possível, inclusive, que ele tenha evoluído na computação, o que não está de acordo com a definição de defeitos de colapso tomada por base (Cristian, seção 2.2.1). Além disso, se a aplicação utiliza arquivos para salvar dados, com esta forma de simular defeito de colapso, os dados desses arquivos podem ficar inconsistentes, pois se a aplicação continuar seu processamento poderá continuar gravando dados. Por esse motivo, essa primeira alternativa de simular defeitos de colapso em um nodo foi descartada.

Outra opção analisada foi a utilização do comando do NS `$ns detach-agent <node> <agent>`, que faz com que um agente (`<agent>`) seja desanexado do nodo (`<node>`). Para utilizar essa forma de simulação de defeitos no modelo baseado em TCP (seção 4.1.1), é necessário executar o comando `detach-agent` para todos os agentes TCP (`FullTcp`) que estão anexados à aplicação. No caso do modelo baseado em UDP (seção 4.1.2), só é necessário usar esse comando para um agente UDP, pois existe somente um agente para cada aplicação (quando é usado *multicast* com somente um grupo de destino para cada aplicação). Entretanto, esta forma de simular defeitos tem o mesmo problema da situação anterior: não satisfaz a definição de defeitos de colapso tomada como base. Isoladamente, o procedimento de desanexação dos agentes de transporte não causa a pausa do processamento da aplicação e, provavelmente, quando recuperado (anexando-se os agentes novamente), o nodo tenha evoluído em sua computação. Além disso, a aplicação pode continuar gravando dados em arquivos.

Solução implementada

Visando a elaboração de uma proposta, buscou-se uma solução genérica que pudesse ser usada para simular defeitos em qualquer tipo de aplicação desenvolvida. Foi constatado que tal solução é difícil, já que, para simular defeito de colapso de uma aplicação, é necessário parar o seu processamento. Como cada aplicação é desenvolvida para propósitos específicos, ela possui dados e execuções particulares e, por isso, a implementação de uma simulação de defeitos de colapso deve ser própria para cada aplicação.

Portanto, a solução encontrada foi a inclusão de um método na classe da aplicação, no qual pode-se reiniciar variáveis desta aplicação. Com esse método (chamado `reset()`) pode-se simular colapso com amnésia total ou parcial. Para simular colapso com amnésia total basta reiniciar todas as variáveis da aplicação. Para simular defeito de colapso com amnésia parcial, pode-se fazer o reinício de parte das variáveis, e deixar as demais com os valores que tinham antes do colapso.

Com esse método, é simulado somente defeito na aplicação que está contida no nodo e não de todo o nodo. Portanto, juntamente com esse método, pode-se utilizar, ou o comando para desanexar todos os agentes de transporte ligados à aplicação - `$ns detach-agent <node> <agent>`, ou então o comando que faz o rompimento de todos os enlaces do nodo - `$ns rtmodel-at <time> down <args>`, dependendo da abordagem que se deseja utilizar. Com o primeiro comando só é simulado defeito da aplicação, já que, o nodo continua com função de roteamento, o que é relevante apenas quando o nodo não é um "nodo-fim" (no grafo que representa a rede, vértice com duas ou mais arestas incidentes). Com o segundo comando, pode-se simular defeito de todo o nodo, pois todos os seus enlaces são rompidos. Quando a aplicação for recuperada, precisa-se utilizar `$ns attach-agent <node> <agent>` para anexar os agentes de transporte novamente ao nodo, ou `$ns rtmodel-at <time> up <args>` para o restabelecimento dos enlaces. Exemplos de como implementar esse método serão apresentados na seção 5.3.

A simulação de defeito de colapso com *halting* em um nodo é feita com a utilização do comando `$ns rtmodel-at <time> down <args>`, que faz com que todos os enlaces do nodo sejam rompidos. Como nesse tipo de defeito o nodo não é reiniciado, seu estado interno não é importante após a ocorrência do defeito. Portanto, com o uso desse comando sem utilizar, posteriormente, a opção que faz o restabelecimento dos enlaces (*up*), é possível simular defeito de colapso com *halting*.

4.3.3 Propostas de simulação de outros tipos de defeitos em nodos

A simulação de outros tipos de defeitos em nodos, de acordo com a classificação de Cristian, não foi implementada nesse trabalho. Entretanto, surgiram algumas idéias de como simulá-los no NS. Defeitos de omissão poderiam ser simulados utilizando-se os procedimentos de omissão ou colapso em enlaces, para simular a perda de uma mensagem, fazendo com que um componente não respondesse a uma solicitação. Defeitos de colapso com pausa poderiam ser simulados implementando-se um método, semelhante ao método `reset`, que teria a função de parar o processamento da aplicação e armazenar os valores contidos nas variáveis antes do colapso. Defeitos de temporização poderiam ser simulados agendando-se o evento do pacote para antes ou depois do tempo previamente estabelecido. Isto poderia ser feito cancelando-se o evento do pacote e, após, inserindo-o novamente, com um tempo de ocorrência diferente. Na classe do escalonador de eventos, existem dois métodos que poderiam ser usados para fazer esta operação: `cancel(Event *e)`, que cancela o evento apontado por `e`; `insert(Event *e)` que insere um evento apontado por `e`. Defeitos de resposta poderiam ser simulados incluindo-se um método na classe da aplicação (como o que é feito com o `reset`), que teria a função de alterar determinados dados da aplicação. Defeitos arbitrários poderiam ser simulados utilizando-se uma combinação das formas anteriores entre outras.

4.4 *Scripts* de simulação

Como o modelo físico dos sistemas distribuídos pode prever diferentes topologias de nodos, tais como a totalmente conectada, em estrela, em anel ou em

árvore, foi implementado um arquivo *script* (topo1.tcl) padrão que facilita a construção dessas topologias em aplicações síncronas ou assíncronas. Nesse *script* (Figura 4.9), foi implementado um procedimento que cria as conexões físicas de cada modelo de topologia. Todos os enlaces têm a mesma velocidade; portanto, para simular uma rede assíncrona, as aplicações devem ser iniciadas em tempos diferentes.

Para utilizar esse *script*, o usuário precisa adicionar o cabeçalho `source <caminho>/topo1.tcl` ao arquivo *script* de sua simulação e chamar o procedimento `create-topology <n_node> <topology> <bandwidth> <delay> <queue_type>`, cujos parâmetros significam: `<n_node>` é o número de nodos; `<topology>` é o tipo de topologia física, definida como *ring* para anel, *tree* para árvore binária, *fully* para totalmente conectada e *star* para estrela; `<bandwidth> <delay> <queue_type>` são a largura de banda, o atraso e tipo de fila dos enlaces, respectivamente. Portanto, com esse arquivo *script*, é preciso especificar a quantidade de nodos, tipo de topologia, a largura de banda, o atraso e o tipo de fila, para que os nodos e os enlaces sejam criados. Após chamar esse procedimento, pode-se referenciar os nodos utilizando-se instâncias da classe `Node` que foram nomeadas de `n(0)` a `n(n_node-1)`. Exemplos de utilização deste *script* serão apresentados no capítulo 5.

```
proc create-topology {nnode topo bw delay q_type} {
   global ns n
   #criação dos nodos
   for {set i 0} {$i < $nnode} {incr i} {
      set n($i) [$ns node]
   }
   #criação da topologia
   switch $topo {
    "fully" {
               for {set i 0} {$i < $nnode} {incr i} {
                    for {set j [expr $i+1]} {$j < $nnode} {incr j} {
                         $ns duplex-link $n($i) $n($j) $bw $delay $q_type
                    }
               }
    }
    "star" {
              for {set i 1} {$i < $nnode} {incr i} {
                   $ns duplex-link $n(0) $n($i) $bw $delay $q_type
              }
    }
    "ring" {
              for {set i 0} {$i < $nnode} {incr i} {
                $ns duplex-link $n($i) $n([expr ($i+1)%$nnode]) $bw $delay $q_type
              }
    }
    "tree" {
              set j 1
              for {set i 0} {$j < [expr $nnode]} {incr i} {
                   for {set x 0} {$x < 2 && $j < [expr $nnode]} {incr x} {
                       $ns duplex-link $n($i) $n($j) $bw $delay $q_type
                          incr j
                   }
              }
    }
     default {
              puts "Wrong parameter!"
              exit
            }
  }
}
```

FIGURA 4.9 - *Script* para criação de topologia

Além do *script* para criação da topologia, foi desenvolvido um *script* (tcp_sd.tcl – Figura 4.10) para facilitar a criação e o estabelecimento de conexões dos agentes e a criação de aplicações contidas nos nodos do sistema do modelo baseado em TCP. Para

utilizá-lo, também é preciso adicionar o cabeçalho `source <caminho>/tcp_sd.tcl` ao *script* de simulação. Este *script* contém os seguintes procedimentos:

- `create-tcpds <n_node>`, onde `<n_node>` é a quantidade de nodos. Dentro desse procedimento (Figura 4.10, linha 1), são criados e anexados os agentes `FullTcp` e `TcpApp` dos `<n_node>` nodos do sistema.
- `create-app <appl> <n_node>` (Figura 4.10, linha 13), onde `<n_node>` é a quantidade de nodos. Nesse procedimento, são criadas `<n_node>` aplicações do tipo definido em `<appl>`.
- `connect-all <n_node>` ou `connect-to <source> <dest>`. No procedimento `connect-all` (Figura 4.10, linha 23), são feitas conexões entre todos os agentes `FullTcp` e `TcpApp`; o parâmetro `<n_node>` é a quantidade de nodos. Esse procedimento é chamado quando se deseja que todas as aplicações sejam conectadas. O procedimento `connect-to` (Figura 4.10, linha 33) é usado quando são feitas algumas conexões. Os parâmetros `<source>` e `<dest>` identificam os nodos onde estão as aplicações de origem e destino respectivamente. Nesse procedimento, são feitas conexões bidirecionais entre os agentes `FullTcp` e `TcpApp` dos nodos identificados por `<source>` e `<dest>`.
- `detach-ag <n_node> <nodef>` (Figura 4.10, linha 49) e `attach-ag <n_node> <nodef>` (Figura 4.10, linha 56) fazem com que todos os agentes do nodo identificado por `<nodef>` sejam desanexados (`detach-ag`) ou anexados (`attach-ag`). Este procedimento pode ser usado, juntamente com o método `reset`, para simulação de defeitos como explicado na seção 4.3.2.

```
1:  proc create-tcpds {nnode} {
2:    global ns tcpap tcp n
3:    for {set i 0} {$i < $nnode} {incr i} { ;#Cria agentes FullTcp e os anexa os nodos
4:            for {set j 1} {$j < $nnode} {incr j} {
5:                    set tcp($i:$j) [new Agent/TCP/FullTcp]
6:                    $ns attach-agent $n($i) $tcp($i:$j)
7:                    $tcp($i:$j) set fid_ $i
8:                    set tcpap($i:$j) [new Application/TcpApp $tcp($i:$j)]
9:            }
10:     }
11:}

12: # Cria aplicações definidas em appl
13: proc create-app {appl nnode}     {
14:    global app n
15:    for {set i 0} {$i < $nnode} {incr i} {
16:            set app($i) [new $appl]
17:    }
18:    # Atribui id à aplicação
19:    for {set i 0} {$i < $nnode} {incr i} {
20:            $app($i) id [$n($i) id]
21:    }
22: }

23: proc connect-all {nnode} {
24:    global ns tcpap tcp
25:    # Conecta todos os agentes FullTcp e TcpApp
26:    for {set i 0} {$i < [expr $nnode-1]} {incr i} {
27:            for {set j [expr $i+1]} {$j < $nnode} {incr j} {
28:                    $ns connect $tcp($i:$j) $tcp($j:[expr $i+1])
29:                    $tcpap($i:$j) connect $tcpap($j:[expr $i+1])
30:            }
31:    }
32: }
```

FIGURA 4.10 - *Script* para criação e conexão de agentes do modelo baseado em TCP (continua)

```
33: proc connect-to {source dest} {
34:    global ns tcpap tcp app
35:    if {$source > $dest} {
36:            puts "source>dest."
37:                   $ns connect $tcp($source:[expr $dest+1]) $tcp($dest:$source)
38:                   $tcpap($source:[expr $dest+1]) connect $tcpap($dest:$source)
39:                   $app($source) connect $app($dest) $tcpap($source:[expr $dest+1])
40:                   $app($dest) connect $app($source) $tcpap($dest:$source)
41:    } else {
42:                   $ns connect $tcp($source:$dest) $tcp($dest:[expr $source+1])
43:                   $tcpap($source:$dest) connect $tcpap($dest:[expr $source+1])
44:                   $app($source) connect $app($dest) $tcpap($source:$dest)
45:                   $app($dest) connect $app($source) $tcpap($dest:[expr $source+1])
46:    }
47: }
48: # Desanexa todos os agentes FullTcp do nodo nodef
49: proc detach-ag {nnode nodef} {
50:    global tcp ns n
51:    for {set i 1} {$i < [expr $nnode]} {incr i} {
52:            $ns detach-agent $n($nodef) $tcp($nodef:$i)
53:    }
54: }

55: # Anexa todos os agentes FullTcp do nodo nodef
56: proc attach-ag {nnode nodef} {
57:    global tcp ns n
58:    for {set i 1} {$i < [expr $nnode]} {incr i} {
59:            $ns attach-agent $n($nodef) $tcp($nodef:$i)
60:    }
61: }
```

FIGURA 4.10 - *Script* para criação e conexão de agentes do modelo baseado em TCP (continuação)

5 Estudo de casos

Neste capítulo, as soluções propostas são exercitadas através de diversos estudos de casos, onde o simulador de redes NS é utilizado para simular sistemas distribuídos em cenários com e sem defeitos. Parte das implementações aqui desenvolvidas são inspiradas em algoritmos distribuídos do livro *Distributed Algorithms* de Nancy Lynch [LYN96]. A descrição dos algoritmos aponta somente os aspectos mais importantes; sugere-se consultar diretamente a fonte para uma descrição detalhada e estudo das provas. Também será apresentada uma implementação do protocolo de gerência de réplicas Primário-*Backup* [BUD93], [GUE97]. Todos os algoritmos foram implementados na versão ns-2.1b7 e testados nas versões ns-2.1b8 e ns-2.1b9 do NS para Linux. Nas Seções 5.1 e 5.2, serão relatadas implementações de algoritmos síncronos e assíncronos, respectivamente, baseados em [LYN96]. A seção 5.3 contém o protocolo Primário-*Backup*.

5.1 Algoritmos síncronos

Nas próximas seções, serão apresentados exemplos de implementações de algoritmos distribuídos em redes síncronas, que foram desenvolvidos utilizando-se o modelo baseado em TCP. Para simular um sistema síncrono, todos os processos iniciam a execução ao mesmo tempo e todas as conexões entre processos têm a mesma velocidade. Além disso, esses algoritmos não prevêem a ocorrência de defeitos.

5.1.1 Eleição de líder em anel - algoritmo LCR

Nessa seção, serão apresentados alguns dos principais aspectos da implementação do algoritmo LCR síncrono, um algoritmo para eleição de líder em um anel, que tem essa denominação dada por Lynch em homenagem a seus autores: Le Lann, Chang e Roberts. Começou-se por esse algoritmo, porque ele é simples e serve como base para explicação dos algoritmos seguintes. Primeiro, será enunciada a definição informal de LCR e, após, será explicada como foi feita sua implementação.

Este algoritmo utiliza somente comunicação unidirecional, não depende do conhecimento do tamanho do anel e usa a comparação entre identificadores dos processos, que estão em nodos diferentes do sistema, para determinar quem será o líder. Cada processo (aplicação) deve ser iniciado com um identificador único (UID) escolhido do conjunto de inteiros positivos; não existe nenhuma ordem entre os identificadores no anel, isto é, os números dos identificadores não precisam ser consecutivos. Além disso, somente um processo pode declarar-se líder.

Definição informal

Cada processo envia seu identificador para seu vizinho no sentido horário. Quando um processo recebe um identificador, compara-o com seu próprio. Existem três possibilidades: se o valor recebido for maior que o seu, o processo passa o identificador; se o valor recebido for menor que o seu, o processo descarta o pacote; por fim, se o identificador for igual ao seu, o processo declara-se líder.

Implementação

Para implementar o LCR, foi desenvolvida uma classe derivada de `AppData` (Figura 5.1) que define uma unidade de dados da aplicação (ADU), ou seja, as mensagens que serão trocadas entre os processos do algoritmo.

```
1:   class LcrData : public AppData {
2:    private:
3:     int cost_;
4:     int uid_;
5:     int id_;
6:    public:
7:     LcrData(int id, int cost, int uid) : AppData(LCR_DATA) {
8:             cost_ = cost;
9:             uid_ = uid;
10:            id_ = id;
11:    }
12:    LcrData(LcrData& d) : AppData(d) {
13:            cost_ = d.cost_;
14:            uid_=d.uid_;
15:            id_=d.id_;
16:    }
17:    virtual ~LcrData() {
18:    }
19:
20:    virtual int size() const {
21:            return (AppData::size()+3*sizeof(int));
22:    }
23:    virtual int cost() const { return cost_; }
24:    int uid() { return uid_; }
25:    int id() { return id_; }
26:    virtual AppData* copy() {
27:            return (new LcrData(*this));
28:    }
29:  };
```

FIGURA 5.1 - Classe `LcrData`

Como pode-se observar, na classe `LcrData` foi definido um construtor que passa o argumento `LCR_DATA` (tipo de dado ADU, definido em ns-process.h) para sua superclasse `AppData` (Figura 5.1, linha 7). Além disso, foram alterados os métodos `size()` e `copy()`- Figura 5.1, linhas 20 e 26 respectivamente.

Para o desenvolvimento da aplicação utilizando-se o modelo baseado em TCP, foi criada uma classe (`LcrApp`), na qual foi implementado o algoritmo LCR. A Figura 5.2 apresenta a definição dessa classe. Na Figura 5.3, são apresentados alguns de seus métodos, incluindo o método `command` onde estão contidas, entre outras, funções para envio de dados. Na Figura 5.7, está o método `process_data`, que contém funções para recepção de dados .

```
class LcrApp : public Process {
public:
        LcrApp();
        virtual ~LcrApp();

        virtual int command(int argc, const char*const* argv);
        void log(const char *fmt, ...);
        int id() const { return id_; }
        virtual void process_data(int size, AppData* d);

protected:
        int add_cnc(LcrApp *client, TcpApp *agt);
        void send();

        int myuid_, uid_ ;
        int bytes_;
        Tcl_HashTable *tpa_;  // Tabela hash TcpApp
        int id_;              // id do nodo
        Tcl_Channel log_;     // descritor do arquivo de log
};
```

FIGURA 5.2 - Definição da classe `LcrApp`

O método `command()` faz a ligação entre um método OTcl e sua implementação em C++. Este método é chamado quando um comando OTcl para a classe `LcrApp` é executado. Por exemplo, quando o comando `$app0 send <bytes>` (assumindo-se que "`app0`" é uma instância da classe `LcrApp`) é executado em OTcl, a função `command` de `LcrApp` é chamada fazendo-se uma busca pelo comando `send` (como explicado na seção 3.3.2). Se este não for encontrado, o comando com seus argumentos é passado para sua superclasse, nesse caso `return TclObject::command(argc, argv)`. O método `command` é mostrado na Figura 5.3 e, a seguir, serão explicados os comandos de maior relevância desse método.

```
1:   static class LcrAppClass : public TclClass { //espelhamento entre hierarquias do NS
2:     public:
3:        LcrAppClass() : TclClass("LcrApp") {}
4:        TclObject* create(int, const char*const*) {
5:             return (new LcrApp());
6:     }
7:   } class_lcr_app;

8:   LcrApp::LcrApp() : log_(0)
9:   {
10:      bind("myuid_", &myuid_);  //ligação de variáveis das duas hierarquias
11:      tpa_ = new Tcl_HashTable;
12:      Tcl_InitHashTable(tpa_, TCL_ONE_WORD_KEYS);
13:  }

14:  LcrApp::~LcrApp()
15:  {
16:      if (tpa_ != NULL) {
17:           Tcl_DeleteHashTable(tpa_);
18:           delete tpa_;
19:      }
20:  }

21:  int LcrApp::add_cnc(LcrApp* client, TcpApp *agt)
22:  {
23:      int newEntry = 1;
24:      Tcl_HashEntry *he = Tcl_CreateHashEntry(tpa_, (const char *)client->id(),
25:                                              &newEntry);
26:      if (he == NULL)
27:           return -1;
28:      if (newEntry)
29:           Tcl_SetHashValue(he, (ClientData)agt);
30:      return 0;
31:  }
```

FIGURA 5.3 - Métodos da classe `LcrApp` (continua)

```
32:  int LcrApp::command(int argc, const char*const* argv)
33:  {
34:       Tcl& tcl = Tcl::instance();

35:       if (argc == 3) {
36:            if (strcmp(argv[1], "id") == 0) {
37:                 /*
38:                  * <app> id <node_id>
39:                  */
40:                 id_ = atoi(argv[2]);
41:                 tcl.resultf("%d", id_);
42:                 return (TCL_OK);
43:            } else if (strcmp(argv[1], "log") == 0) {
44:                 /*
45:                  * <app> log <arq_log>
46:                  */
47:                 int mode;
48:                 log_ = Tcl_GetChannel(tcl.interp(), (char*)argv[2], &mode);
49:                 if (log_ == 0) {
50:                        tcl.resultf("%d: invalid log file handle %s\n",
51:                            id_, argv[2]);
52:                        return TCL_ERROR;
53:                 }
54:                 return (TCL_OK);
55:            }
56:            else if (strcmp(argv[1], "simplex-connect") == 0) {
57:                 /*
58:                  * <app> simplex-connect <tcpap>
59:                  * Faz com que a aplicação <app> seja o alvo
60:                  * quando <tcpap> receber dados
61:                  */
62:                 TcpApp *cnc = (TcpApp*)TclObject::lookup(argv[2]);
63:                 cnc->target() = (Process*)this;
64:                 return (TCL_OK);
65:           }
66:           else if (strcmp(argv[1], "send") == 0) {
67:                 /*
68:                  * <app> send <bytes>
69:                  * Envia dados de <app> para <appclient>
70:                  */
71:                 bytes_ = atoi(argv[2]);
72:                 uid_=myuid_;
73:                 send();
74:                 return (TCL_OK);
75:            }
76:       }else if (argc == 4) {
77:              if (strcmp(argv[1], "connect") == 0) {
78:                 /*
79:                  * <app> connect <appclient> <tcpap>
80:                  * Associa um agente TcpApp <tcpap> com a aplicação
81:                  * cliente <appclient>. O agente TCP é usado para
82:                  * enviar pacotes para a aplicação cliente.
83:                  */
84:                 LcrApp *client = (LcrApp *)TclObject::lookup(argv[2]);
85:                 TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
86:                 if (add_cnc(client, cnc)) {
87:                        tcl.resultf("%s: failed to connect to %s",
88:                                    name_, argv[2]);
89:                        return TCL_ERROR;
90:                 }
91:                 /*
92:                  *Estabelece o alvo de entrega dos dados,
93:                  *isto é, a própria aplicação
94:                 */
95:                 cnc->target() = (Process*)this;
96:                 return TCL_OK;
97:            }
98:      }
99:      return TclObject::command(argc, argv);
100:  }
```

FIGURA 5.3 - Métodos da `classe` LcrApp (continuação)

O comando `connect`, contido no método `command` (Figura 5.3, linha 77), tem a sintaxe em OTcl `<app> connect <appclient> <tcpap>`, na qual: `<app>` é a

aplicação que fará o envio dos dados; <appclient> é a aplicação que irá receber os dados; e <tcpap> é o agente TcpApp que se encarregará de enviar os dados para a aplicação <appclient>.

No connect, a linha 86 da Figura 5.3 chama o método add_cnc(client, cnc) (Figura 5.3, linha 21), que adiciona a uma tabela o par <appclient> e <tcpap>. Embora nessa implementação de LCR, tal tabela não seja necessária, pois só é utilizada para que a aplicação possa enviar dados para vários destinatários, ela foi empregada para manter a mesma estrutura que será usada na implementação dos outros algoritmos apresentados nesse trabalho. Nessa tabela, é armazenado um agente TcpApp para cada destinatário, assim, quando for preciso enviar dados para aplicações, é feita uma busca na tabela pelos agentes TcpApp ligados às aplicações destino. Na linha 95 da Figura 5.3, o comando cnc->target()=(Process*)this faz com que o agente TcpApp (nesse caso <tcpap>) seja ligado à sua aplicação (<app>), para que este agente TcpApp entregue os dados a esta aplicação quando recebê-los do agente FullTcp.

No comando simplex-connect (Figura 5.3, linha 56), só é feita a ligação entre o TcpApp e a aplicação (cnc->target()=(Process*)this). Nesse comando, a sintaxe é a seguinte: <app> simplex-connect <tcpap>, na qual <tcpap> é o agente TcpApp que só receberá dados (não enviará) e os passará para a aplicação <app>.

O comando simplex-connect é usado em LCR porque esse algoritmo é implementado em um anel unidirecional, sendo assim, existe um par de agentes TcpApp - FullTcp para envio (TcpApp_S e FullTcp_S, na Figura 5.4) e outro par para recepção de dados. Em outras palavras, um agente FullTcp é usado só para o envio e outro agente FullTcp só para recepção de dados. Em OTcl, é necessário usar o comando <tcp> listen para atribuir ao agente FullTcp <tcp>, que está ligado a TcpApp (<tcpap>), o estado de escuta (na Figura 5.4 os agentes FullTcp_R estão em estado de escuta), indicando que este está esperando os dados que serão enviados a ele. A Figura 5.4 mostra essas conexões.

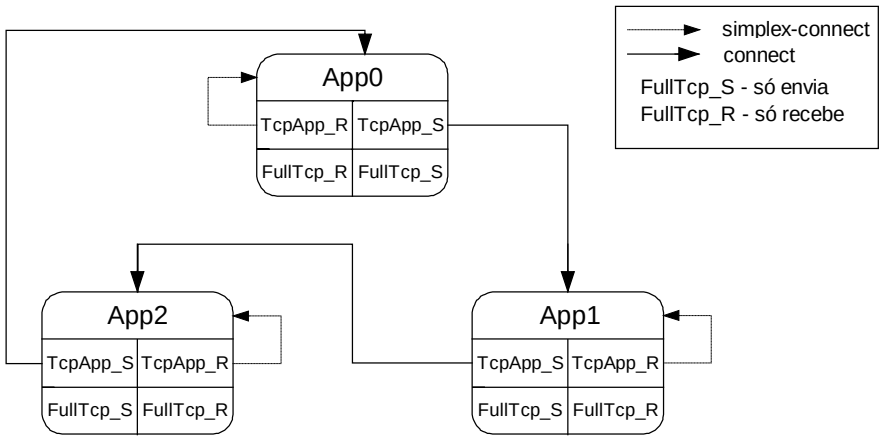


FIGURA 5.4 - Conexões de LCR

O comando send (<app> send <bytes> - Figura 5.3, linha 66) faz o envio de dados. Para isso, é feita uma chamada ao método send() (Figura 5.5), que faz uma busca na tabela, a partir do início, por todos os agentes TcpApp que estão associados às aplicações destino e os atribui à cnc (Figura 5.5, linha 10). Após, é feita a inclusão dos dados na ADU com LcrData *d = new LcrData(id_, bytes_, id_) – Figura 5.5, linha 12. Então os dados são enviados para todas as aplicações, que foram conectadas a

esta aplicação através do comando `connect`, chamando-se o método `send(bytes_, d)` da classe `TcpApp` (Figura 5.5, linha 14). No caso do algoritmo LCR, cada aplicação envia seu UID (identificador único) para um destinatário apenas, a aplicação ao seu lado no sentido horário.

```
1:   void LcrApp::send()
2:   {
3:     Tcl_HashEntry *he;
4:     Tcl_HashSearch hs;
5:     /* Laço que faz busca a partir do primeiro elemento da tabela
6:     * por todos os TcpApp que estão contidos nela
7:     */
8:     for (he = Tcl_FirstHashEntry(tpa_, &hs); he != NULL; he = Tcl_NextHashEntry(&hs))
9:     {
10:            TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
11:            // empacotamento de dados - ADU
12:            LcrData *d = new LcrData(id_, bytes_, uid_);
13:            log("UID %d  s \n", uid_);
14:            cnc->send(bytes_, d); //envio de dados
15:    }
16:}
```

FIGURA 5.5 - Método `send` da classe `LcrApp`

O método `log`[1] (Figura 5.6) gera um arquivo de *log* dessa aplicação, no qual podem ser colocadas informações julgadas importantes. Este arquivo é definido pelo comando da Figura 5.3, linha 43. A linha 13 da Figura 5.5 e as linhas 11 e 13 da Figura 5.7 contêm chamadas para esse método. Um exemplo de um arquivo de *log* gerado por essa aplicação é apresentado na Figura 5.10.

```
1: void LcrApp::log(const char* fmt, ...)
2: {
3:     if (log_ == 0)
4:             return;
5:     char buf[10240], *p;
6:     sprintf(buf, TIME_FORMAT" src %d ",
7:             Trace::round(Scheduler::instance().clock()), id_);
8:     p = &(buf[strlen(buf)]);
9:     va_list ap;
10:    va_start(ap, fmt);
11:    vsprintf(p, fmt, ap);
12:    Tcl_Write(log_, buf, strlen(buf));
13: }
```

FIGURA 5.6 - Método `log` da classe `LcrApp`

O método `process_data` (Figura 5.7) recebe os dados, faz sua interpretação e, após, se necessário, faz o envio de novos dados. Nesse método, inicialmente é verificado o tipo de dado recebido. Se o tipo é `LCR_DATA` (Figura 5.7, linha 5), então é verificado o valor do UID recebido (`uid_`); se este é maior que o UID da aplicação (`myuid_`), esta aplicação passa o UID recebido para seu vizinho, chamando o método `send` (linhas 8 e 9 da Figura 5.7). Se o UID recebido é menor que o da aplicação (linha 10 da Figura 5.7), nada é feito (dados são descartados). Se o UID recebido é igual ao seu, a aplicação declara-se líder (Figura 5.7, linhas 12 e 13).

[1] O método exibido na Figura 5.6 foi implementado na versão ns-2.1b7 do NS; nas versões seguintes, o nome da classe `Trace`, linha 7, foi modificado para `BaseTrace`

```
1:void LcrApp::process_data(int, AppData* data)
2:{
3:      if (data == NULL)
4:              return;
5:      if (data->type()==LCR_DATA) {
6:              LcrData *tmp = (LcrData*)data;
7:              uid_=tmp->uid();
8:              if (uid_ > myuid_) {//se uid recebido é maior que uid do processo(myuid_)
9:                      send();       //envia o uid recebido (uid_)
10:             } else if (uid_ < myuid_) { //se uid recebido é menor que uid do processo
11:                     log("Packet discarded! d\n");  //pacote é descartado
12:             } else if (uid_ == myuid_){ //se uid recebido é igual a uid do processo
13:                     log("Leader - UID %d\n", myuid_); //processo se declara líder
14:             }
15:     } else{
16:             fprintf(stderr, "Bad data type %d\n", data->type());
17:             abort();
18:     }
19:}
```

FIGURA 5.7 - Método process_data da classe `LcrApp`

Exemplo de simulação

A Figura 5.8 apresenta um diagrama de tempos com as trocas de mensagens entre aplicações de uma simulação do algoritmo LCR com três nodos. Cada nodo contém uma aplicação (`Appn`), sendo que `App0` tem UID igual a 5, `App1` tem UID igual a 15 e `App2` tem UID igual a 10. O *script* OTcl dessa simulação é mostrado na Figura 5.9, seu arquivo de *log* na Figura 5.10 e seu arquivo de rastro na Figura 5.11.

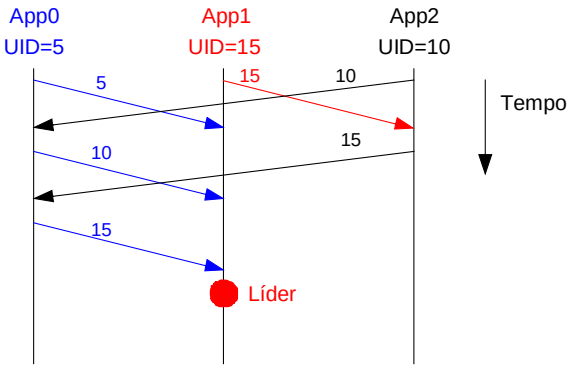


FIGURA 5.8 - Diagrama com trocas de mensagens entre aplicações LCR

```
source /home/renata/extcpap/topo1.tcl

#Cria o objeto Simulator
set ns [new Simulator]

#Abre arquivos
set f [open out.tr w]
$ns trace-all $f
set nf [open out1.nam w]
$ns namtrace-all $nf
set log [open lcr.log w]

#Procedimento finish
proc finish {} {
        global ns nf f
        $ns flush-trace
        close $nf
        close $f
        close $log
        exec nam out1.nam &
        exit 0
}
```

FIGURA 5.9 - Exemplo de um *script* de simulação de LCR (continua)

```
#Chamada o procedimento create-topology (contido em topo1.tcl) para criar a topologia
create-topology 3 ring 1Mb 10ms DropTail

#cria agentes FullTcp e os anexa seus respectivos nodos
set tcp01 [new Agent/TCP/FullTcp]
$ns attach-agent $n(0) $tcp01
$tcp01 set fid_ 40

set tcp02 [new Agent/TCP/FullTcp]
$ns attach-agent $n(0) $tcp02
$tcp02 set fid_ 43
$tcp02 listen

set tcp11 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp11
$tcp11 set fid_ 41
$tcp11 listen

set tcp12 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp12
$tcp12 set fid_ 44

set tcp21 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp21
$tcp21 set fid_ 42
$tcp21 listen

set tcp22 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp22
$tcp22 set fid_ 45

#Conecta os agentes FullTCP
$ns connect $tcp01 $tcp11
$ns connect $tcp12 $tcp21
$ns connect $tcp22 $tcp02

#Cria aplicações TcpApp e as anexa aos agentes FullTcp
set tcpap01 [new Application/TcpApp $tcp01]
set tcpap11 [new Application/TcpApp $tcp11]
set tcpap21 [new Application/TcpApp $tcp21]
set tcpap02 [new Application/TcpApp $tcp02]
set tcpap12 [new Application/TcpApp $tcp12]
set tcpap22 [new Application/TcpApp $tcp22]

#Cria aplicações
set app0 [new LcrApp]
set app1 [new LcrApp]
set app2 [new LcrApp]

#Atribui à id o id do nodo
$app0 id [$n(0) id]
$app1 id [$n(1) id]
$app2 id [$n(2) id]

$app0 log $log
$app1 log $log
$app2 log $log

#Atribui UID às aplicações
$app0 set myuid_ 5
$app1 set myuid_ 15
$app2 set myuid_ 10

#Conecta agentes TcpApp
$tcpap01 connect $tcpap11
$tcpap12 connect $tcpap21
$tcpap22 connect $tcpap02

$app0 connect $app1 $tcpap01
$app0 simplex-connect $tcpap02
$app1 simplex-connect $tcpap11
$app1 connect $app2 $tcpap12
$app2 simplex-connect $tcpap21
$app2 connect $app0 $tcpap22
```

FIGURA 5.9 - Exemplo de um script de simulação de LCR (continuação)

```
#Inicia eleição
$ns at 0.1 "$app0 send 40"
$ns at 0.1 "$app1 send 40"
$ns at 0.1 "$app2 send 40"

#Chama procedimento finish
$ns at 0.7 "finish"

#Executa a simulação
$ns run
```

FIGURA 5.9 - Exemplo de um script de simulação de LCR (continua)

```
0.1 src 0 UID 5  s
0.1 src 1 UID 15  s
0.1 src 2 UID 10  s
0.1316 src 1 Packet discarded! d
0.1316 src 2 UID 15  s
0.1316 src 0 UID 10  s
0.22096 src 1 Packet discarded! d
0.22096 src 0 UID 15  s
0.32096 src 1 Leader – UID 15
```

FIGURA 5.10 - Exemplo de arquivo de *log* gerado por uma aplicação `LcrApp`

```
+ 0.1 0 1 tcp 40 ------- 40 0.0 1.0 0 0
- 0.1 0 1 tcp 40 ------- 40 0.0 1.0 0 0
+ 0.1 1 2 tcp 40 ------- 44 1.1 2.0 0 1
- 0.1 1 2 tcp 40 ------- 44 1.1 2.0 0 1
+ 0.1 2 0 tcp 40 ------- 45 2.1 0.1 0 2
- 0.1 2 0 tcp 40 ------- 45 2.1 0.1 0 2
r 0.11032 0 1 tcp 40 ------- 40 0.0 1.0 0 0
+ 0.11032 1 0 ack 40 ------- 41 1.0 0.0 0 3
- 0.11032 1 0 ack 40 ------- 41 1.0 0.0 0 3
r 0.11032 1 2 tcp 40 ------- 44 1.1 2.0 0 1
+ 0.11032 2 1 ack 40 ------- 42 2.0 1.1 0 4
- 0.11032 2 1 ack 40 ------- 42 2.0 1.1 0 4
r 0.11032 2 0 tcp 40 ------- 45 2.1 0.1 0 2
+ 0.11032 0 2 ack 40 ------- 43 0.1 2.1 0 5
- 0.11032 0 2 ack 40 ------- 43 0.1 2.1 0 5
r 0.12064 1 0 ack 40 ------- 41 1.0 0.0 0 3
+ 0.12064 0 1 tcp 40 ------- 40 0.0 1.0 1 6
- 0.12064 0 1 tcp 40 ------- 40 0.0 1.0 1 6
+ 0.12064 0 1 tcp 80 ------- 40 0.0 1.0 1 7
r 0.12064 2 1 ack 40 ------- 42 2.0 1.1 0 4
+ 0.12064 1 2 tcp 40 ------- 44 1.1 2.0 1 8
- 0.12064 1 2 tcp 40 ------- 44 1.1 2.0 1 8
+ 0.12064 1 2 tcp 80 ------- 44 1.1 2.0 1 9
r 0.12064 0 2 ack 40 ------- 43 0.1 2.1 0 5
+ 0.12064 2 0 tcp 40 ------- 45 2.1 0.1 1 10
- 0.12064 2 0 tcp 40 ------- 45 2.1 0.1 1 10
+ 0.12064 2 0 tcp 80 ------- 45 2.1 0.1 1 11
- 0.12096 0 1 tcp 80 ------- 40 0.0 1.0 1 7
- 0.12096 1 2 tcp 80 ------- 44 1.1 2.0 1 9
- 0.12096 2 0 tcp 80 ------- 45 2.1 0.1 1 11
r 0.13096 0 1 tcp 40 ------- 40 0.0 1.0 1 6
r 0.13096 1 2 tcp 40 ------- 44 1.1 2.0 1 8
r 0.13096 2 0 tcp 40 ------- 45 2.1 0.1 1 10
r 0.1316 0 1 tcp 80 ------- 40 0.0 1.0 1 7
r 0.1316 1 2 tcp 80 ------- 44 1.1 2.0 1 9
r 0.1316 2 0 tcp 80 ------- 45 2.1 0.1 1 11
+ 0.2 1 0 ack 40 ------- 41 1.0 0.0 1 12
- 0.2 1 0 ack 40 ------- 41 1.0 0.0 1 12
+ 0.2 2 1 ack 40 ------- 42 2.0 1.1 1 13
- 0.2 2 1 ack 40 ------- 42 2.0 1.1 1 13
+ 0.2 0 2 ack 40 ------- 43 0.1 2.1 1 14
- 0.2 0 2 ack 40 ------- 43 0.1 2.1 1 14
r 0.21032 1 0 ack 40 ------- 41 1.0 0.0 1 12
+ 0.21032 0 1 tcp 80 ------- 40 0.0 1.0 41 15
- 0.21032 0 1 tcp 80 ------- 40 0.0 1.0 41 15
r 0.21032 2 1 ack 40 ------- 42 2.0 1.1 1 13
r 0.21032 0 2 ack 40 ------- 43 0.1 2.1 1 14
+ 0.21032 2 0 tcp 80 ------- 45 2.1 0.1 41 16
```

FIGURA 5.11 - Exemplo de arquivo de rastro gerado por uma aplicação `LcrApp` (continua)

```
- 0.21032 2 0 tcp 80 ------- 45 2.1 0.1 41 16
r 0.22096 0 1 tcp 80 ------- 40 0.0 1.0 41 15
r 0.22096 2 0 tcp 80 ------- 45 2.1 0.1 41 16
+ 0.3 1 0 ack 40 ------- 41 1.0 0.0 1 17
- 0.3 1 0 ack 40 ------- 41 1.0 0.0 1 17
+ 0.3 0 2 ack 40 ------- 43 0.1 2.1 1 18
- 0.3 0 2 ack 40 ------- 43 0.1 2.1 1 18
r 0.31032 1 0 ack 40 ------- 41 1.0 0.0 1 17
+ 0.31032 0 1 tcp 80 ------- 40 0.0 1.0 81 19
- 0.31032 0 1 tcp 80 ------- 40 0.0 1.0 81 19
r 0.31032 0 2 ack 40 ------- 43 0.1 2.1 1 18
r 0.32096 0 1 tcp 80 ------- 40 0.0 1.0 81 19
+ 0.4 1 0 ack 40 ------- 41 1.0 0.0 1 20
- 0.4 1 0 ack 40 ------- 41 1.0 0.0 1 20
r 0.41032 1 0 ack 40 ------- 41 1.0 0.0 1 20
```

FIGURA 5.11 - Exemplo de arquivo de rastro gerado por uma aplicação `LcrApp` (continuação)

5.1.2 Eleição de líder em rede arbitrária – algoritmo de inundação

Nessa seção, será apresentado um algoritmo de eleição de líder que pode ser implementado em qualquer tipo de rede, sendo enunciada sua definição informal e, após, sua implementação.

No caso de eleição de líder em uma rede arbitrária, é assumido que cada processo tem um identificador único (UID) diferente dos identificadores dos outros processos e não existe uma ordem predefinida de onde cada identificador deve estar localizado. Esse algoritmo exige que somente um processo se declare líder, que os demais processos determinem que não são líderes e que cada processo saiba a identidade do líder.

O algoritmo de inundação simples, ou Floodmax, como foi chamado por Lynch, faz com que tanto o processo líder quanto os demais processos (não-líderes) se identifiquem, isto é, faz com que cada um declare-se líder ou não-líder. Para isso, é necessário que os processos conheçam o diâmetro da rede (*diam*), que é a distância máxima entre quaisquer dois nodos. O algoritmo faz a inundação da rede com o maior UID conhecido até aquele momento.

Definição Informal

Cada processo armazena o máximo UID recebido, usando como valor inicial seu próprio UID. Em cada rodada[1], cada processo propaga o UID máximo para todos os seus vizinhos. Depois de um número de rodadas definido pelo diâmetro da rede (*diam*), se o maior valor visto pelo processo é seu próprio UID, ele se declara líder; caso contrário, declara-se não-líder.

Implementação

Para implementar esse algoritmo, foi criada uma classe derivada de `AppData` (como na implementação do LCR), que define a unidade de dados da aplicação (ADU). Esta classe, denominada `FloodData`, é apresentada na Figura 5.12.

[1] Segundo Lynch uma rodada é a combinação de dois passos: 1) Gerar mensagens (a partir do estado corrente) que serão enviadas a todos os vizinhos e colocar essas mensagens nos canais de comunicação apropriados; 2) Aplicar uma função de transição de estado a partir das mensagens recebidas e obter um novo estado. Remover todas as mensagens dos canais apropriados.

```
class FloodData : public AppData {
private:
        int cost_;
        int uid_;
        int id_;
public:
        FloodData(int id, int cost, int uid) :
                AppData(FLOOD_DATA) {
                cost_ = cost;
                uid_ = uid;
                id_ = id;
        }
        FloodData(FloodData& d) : AppData(d) {
                cost_ = d.cost_;
                uid_=d.uid_;
                id_=d.id_;
        }
        virtual ~FloodData() {
        }
        virtual int size() const {
                return (AppData::size()+3*sizeof(int));
        }
        virtual int cost() const { return cost_; }
        int uid() { return uid_; }
        int id() { return id_; }
        virtual AppData* copy() {
                return (new FloodData(*this));
        }
};
```

FIGURA 5.12 - Classe `FloodData`

Para o desenvolvimento da aplicação, foi criada a classe `FloodApp`, na qual foi implementado o algoritmo Floodmax. A Figura 5.13 apresenta a definição dessa classe. Na Figura 5.14, são apresentados seu construtor e seu destrutor. No construtor, as variáveis `diam_` e `neighbor_` são ligadas a variáveis OTcl pelo método `bind` (apresentado na seção 3.3.2). Na variável `diam_`, é atribuído o valor de *diam* especificado no *script*; na variável `neighbor_`, é atribuída a quantidade de vizinhos da aplicação.

```
class FloodApp : public Process {
public:
        FloodApp();
        Virtual ~FloodApp();

        virtual int command(int argc, const char*const* argv);
        void log(const char *fmt, ...);
        int id() const { return id_; }

        virtual void process_data(int size, AppData* d);
protected:
        int add_cnc(FloodApp *client, TcpApp *agt);
        void delete_cnc(FloodApp *client);
        TcpApp* lookup_cnc(FloodApp* client);
        void send();

        int myuid_;
        int rounds_;
        int diam_;
        int neighCount_;
        int maxid_;
        int neighbor_;
        int bytes_;
        Tcl_HashTable *tpa_;  // tabela hash
        int id_;              // id do nodo
        Tcl_Channel log_;     // descritor do arquivo de log
};
```

FIGURA 5.13 - Definição da classe `FloodApp`

```
1:  FloodApp::FloodApp() : rounds_(0), neighCount_(0), log_(0)
2:  {
3:      bind("neighbor_", &neighbor_);
4:      bind("diam_", &diam_);
5:      tpa_ = new Tcl_HashTable;
6:      Tcl_InitHashTable(tpa_, TCL_ONE_WORD_KEYS);
7:  }
8:  FloodApp::~FloodApp()
9:  {
10:     if (tpa_ != NULL) {
11:             Tcl_DeleteHashTable(tpa_);
12:             delete tpa_;
13:     }
14: }
```

FIGURA 5.14 - Construtor e destrutor da classe `FloodApp`

No método `command()` dessa classe, os principais comandos são: `uid`, `connect` e `send`. O comando `uid` (Figura 5.15, linha 5) atribui a uma variável (`myuid_`) o identificador único (UID) estabelecido no *script* OTcl, assim como atribui a `maxid_` o mesmo valor. O comando `connect` (Figura 5.15, linha 22) é feito da mesma forma que na implementação do algoritmo LCR, explicado na seção 5.1.1. O comando `send` chama o método `send()` (Figura 5.16) que será explicado a seguir. Os métodos `log` e `add_cnc` são implementados da mesma forma que no algoritmo LCR.

```
1: int FloodApp::command(int argc, const char*const* argv)
2: {
3:      Tcl& tcl = Tcl::instance();
4:      if (argc == 3) {
               .
               .
               .
5:             else if (strcmp(argv[1], "uid") == 0) {
6:                     /*
7:                      * <app> uid <UID>
8:                      */
9:                      myuid_=atoi(argv[2]);
10:                     maxid_=myuid_;
11:                     return TCL_OK;
12:            }
13:            else if (strcmp(argv[1], "send") == 0) {
14:                    /*
15:                     * <app> send <bytes_>
16:                     */
17:                    bytes_ = atoi(argv[2]);
18:                    send();
19:                    return TCL_OK;
20:            }
21:    }else if (argc == 4) {
22:            if (strcmp(argv[1], "connect") == 0) {
23:                    /*
24:                     * <app> connect <appclient> <tcpap>
25:                     * Associa um agente TcpApp <tcpap> com a aplicação
26:                     * cliente <appclient>. O agente TCP é usado para
27:                     * enviar pacotes para a aplicação cliente.
28:                     */
29:                    FloodApp *client = (FloodApp *)TclObject::lookup(argv[2]);
30:                    TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
31:                    if (add_cnc(client, cnc)) {
32:                           tcl.resultf("%s: failed to connect to %s",
33:                                       name_, argv[2]);
34:                           return TCL_ERROR;
35:                    }
36:                     cnc->target() = (Process*)this;
37:                     return TCL_OK;
38:            }
39:    }
40:    return TclObject::command(argc, argv);
41: }
```

FIGURA 5.15 - Método `command` da classe `FloodApp`

```
1: void FloodApp::send()
2: {
3:    Tcl_HashEntry *he;
4:    Tcl_HashSearch hs;
5:    for (he = Tcl_FirstHashEntry(tpa_, &hs);  he != NULL; he = Tcl_NextHashEntry(&hs))
6:    {
7:            TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
8:            FloodData *d = new FloodData(id_, bytes_, maxid_);  //ADU
9:            log("UID %d  s \n", maxid_);
10:           cnc->send(bytes_, d);
11:    }
12: }
```

FIGURA 5.16 - Método `send` da classe `FloodApp`

O método `send` faz uma busca, na tabela de conexões, por todos agentes `TcpApp` contidos nela. Em outras palavras, o laço `for` (Figura 5.16, linha 5) percorre toda a tabela fazendo uma busca pelos agentes `TcpApp` e chamando o método `send` (`cnc->send(bytes_, d)` - Figura 5.16, linha 10) de cada um desses agentes. Isto faz com que os dados sejam enviados para todas as aplicações, que foram conectadas a esta aplicação através do comando `connect` (Figura 5.15, linha 22).

No algoritmo Floodmax, cada aplicação envia o máximo UID conhecido para todos os seus vizinhos (`send` no método `command`). Após, cada aplicação espera receber os UIDs máximos de todos os seus vizinhos e os compara para verificar qual é o valor maior (Figura 5.17, linhas 8 a 12); então envia o máximo UID para todos os seus vizinhos novamente (Figura 5.17, linha 18). Esse processo é feito em rodadas e após *diam* rodadas, se o máximo UID conhecido pela aplicação for igual ao seu, esta se declara líder (Figura 5.17, linha 22), senão se declara não-líder (Figura 5.17, linha 24).

```
1: void FloodApp::process_data(int, AppData* data)
2: {
3:     if (data == NULL)
4:             return;
5:     if (data->type()== FLOOD_DATA) {
6:             FloodData *tmp = (FloodData*)data;
7:             int uid=tmp->uid();
8:             if (neighCount_<neighbor_){ //neighCount_ é o contador de vizinhos
9:                     if (maxid_ < uid) //compara se o valor do uid recebido é maior
10:                         maxid_=uid;   //que o uid máximo conhecido
11:                    neighCount_++;
12:            }
13:            if (neighCount_ == neighbor_) { //se neighCount é igual ao número de
14:                 rounds_++;  //vizinhos – recebeu de todos os vizinhos – incrementa
15:                 neighCount_=0;  //número da rodada (rounds_)
16:                 if (rounds_ < diam_) //se número da rodada é menor que diam
17:                 {
18:                      send();  //envia para todos os vizinhos
19:                 }
20:                 else if(rounds_== diam_) { // número de rodadas igual a diam
21:                     if(maxid_ == myuid_) //se UID máximo é igual ao UID da aplicação
22:                         log("Leader – UID %d\n", myuid_); //se declara líder
23:                     else
24:                         log("Not Leader - UID %d\n", myuid_); //se declara não líder
25:                   }
26:            }
27:    }
28:    else {
29:            fprintf(stderr, "Bad Floodmax data type %d\n", data->type());
30:            abort();
31:    }
32: }
```

FIGURA 5.17 - Método `process_data` da classe `FloodApp`

Exemplo de simulação

A Figura 5.18 apresenta uma tela do Nam correspondente a um exemplo de simulação do algoritmo Floodmax com dez nodos. Todas as aplicações começam a execução no tempo 0.1, enviando o máximo UID para todos os seus vizinhos. Depois de 4 rodadas, é determinado o líder: o nodo 6 com UID igual a 90. O *script* OTcl dessa simulação é mostrado na Figura 5.19 e seu arquivo de *log* na Figura 5.20.

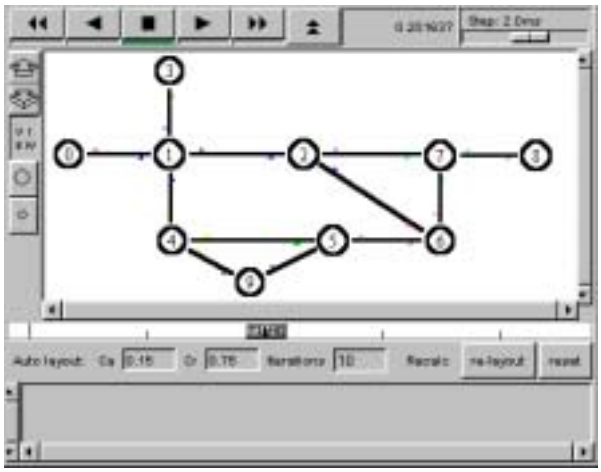


FIGURA 5.18 - Tela do Nam de um exemplo de simulação de Floodmax

```
#Criação do objeto Simulator
set ns [new Simulator]

#Abertura de arquivos de rastro e de log
set f [open flood10.tr w]
$ns trace-all $f
set nf [open flood10.nam w]
$ns namtrace-all $nf
set log [open flood10.log w]

$ns color 0 red
$ns color 1 blue
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow
$ns color 5 green
$ns color 6 HotPink
$ns color 7 cyan
$ns color 8 orange
$ns color 9 brown

 proc finish {} {
        global ns nf f
        $ns flush-trace
        close $nf
        close $f
        exec nam flood10.nam &
        exit 0
}

#Criação de 10 nodes
for {set i 0} {$i < 10} {incr i} {
        set n($i) [$ns node]
}

#Conexões entre os nodos
$ns duplex-link $n(0) $n(1) 1Mb 10ms DropTail
$ns duplex-link $n(1) $n(2) 1Mb 10ms DropTail
$ns duplex-link $n(1) $n(3) 1Mb 10ms DropTail
$ns duplex-link $n(1) $n(4) 1Mb 10ms DropTail
$ns duplex-link $n(4) $n(5) 1Mb 10ms DropTail
$ns duplex-link $n(4) $n(9) 1Mb 10ms DropTail
```

FIGURA 5.19 - Exemplo de *script* de simulação do algoritmo Floodmax (continua)

```
$ns duplex-link $n(2) $n(6) 1Mb 10ms DropTail
$ns duplex-link $n(5) $n(6) 1Mb 10ms DropTail
$ns duplex-link $n(5) $n(9) 1Mb 10ms DropTail
$ns duplex-link $n(2) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(6) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(7) $n(8) 1Mb 10ms DropTail

#Cria agentes FullTcp e os anexa seus respectivos nodos
set tcp01 [new Agent/TCP/FullTcp]
$ns attach-agent $n(0) $tcp01
$tcp01 set fid_ 0

set tcp11 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp11
$tcp11 set fid_ 1

set tcp12 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp12
$tcp12 set fid_ 1

set tcp13 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp13
$tcp13 set fid_ 1

set tcp14 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp14
$tcp14 set fid_ 1

set tcp21 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp21
$tcp21 set fid_ 2

set tcp22 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp22
$tcp22 set fid_ 2

set tcp23 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp23
$tcp23 set fid_ 2

set tcp31 [new Agent/TCP/FullTcp]
$ns attach-agent $n(3) $tcp31
$tcp31 set fid_ 3

set tcp41 [new Agent/TCP/FullTcp]
$ns attach-agent $n(4) $tcp41
$tcp41 set fid_ 4

set tcp42 [new Agent/TCP/FullTcp]
$ns attach-agent $n(4) $tcp42
$tcp42 set fid_ 4

set tcp43 [new Agent/TCP/FullTcp]
$ns attach-agent $n(4) $tcp43
$tcp43 set fid_ 4

set tcp51 [new Agent/TCP/FullTcp]
$ns attach-agent $n(5) $tcp51
$tcp51 set fid_ 5

set tcp52 [new Agent/TCP/FullTcp]
$ns attach-agent $n(5) $tcp52
$tcp52 set fid_ 5

set tcp53 [new Agent/TCP/FullTcp]
$ns attach-agent $n(5) $tcp53
$tcp53 set fid_ 5

set tcp61 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp61
$tcp61 set fid_ 6

set tcp62 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp62
$tcp62 set fid_ 6

set tcp63 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp63
$tcp63 set fid_ 6

set tcp71 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp71
$tcp71 set fid_ 7
```

FIGURA 5.19 - Exemplo de *script* de simulação do algoritmo Floodmax (continuação)

```
set tcp72 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp72
$tcp72 set fid_ 7

set tcp73 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp73
$tcp73 set fid_ 7

set tcp81 [new Agent/TCP/FullTcp]
$ns attach-agent $n(8) $tcp81
$tcp81 set fid_ 8

set tcp91 [new Agent/TCP/FullTcp]
$ns attach-agent $n(9) $tcp91
$tcp91 set fid_ 9

set tcp92 [new Agent/TCP/FullTcp]
$ns attach-agent $n(9) $tcp92
$tcp92 set fid_ 9

#Conecta agentes
$ns connect $tcp01 $tcp11
$ns connect $tcp12 $tcp21
$ns connect $tcp13 $tcp31
$ns connect $tcp14 $tcp41
$ns connect $tcp22 $tcp61
$ns connect $tcp23 $tcp71
$ns connect $tcp42 $tcp51
$ns connect $tcp43 $tcp91
$ns connect $tcp52 $tcp62
$ns connect $tcp53 $tcp92
$ns connect $tcp63 $tcp73
$ns connect $tcp81 $tcp72

set tcpap01 [new Application/TcpApp $tcp01]
set tcpap11 [new Application/TcpApp $tcp11]
set tcpap12 [new Application/TcpApp $tcp12]
set tcpap13 [new Application/TcpApp $tcp13]
set tcpap14 [new Application/TcpApp $tcp14]
set tcpap21 [new Application/TcpApp $tcp21]
set tcpap22 [new Application/TcpApp $tcp22]
set tcpap23 [new Application/TcpApp $tcp23]
set tcpap31 [new Application/TcpApp $tcp31]
set tcpap41 [new Application/TcpApp $tcp41]
set tcpap42 [new Application/TcpApp $tcp42]
set tcpap43 [new Application/TcpApp $tcp43]
set tcpap51 [new Application/TcpApp $tcp51]
set tcpap52 [new Application/TcpApp $tcp52]
set tcpap53 [new Application/TcpApp $tcp53]
set tcpap61 [new Application/TcpApp $tcp61]
set tcpap62 [new Application/TcpApp $tcp62]
set tcpap63 [new Application/TcpApp $tcp63]
set tcpap71 [new Application/TcpApp $tcp71]
set tcpap72 [new Application/TcpApp $tcp72]
set tcpap73 [new Application/TcpApp $tcp73]
set tcpap81 [new Application/TcpApp $tcp81]
set tcpap91 [new Application/TcpApp $tcp91]
set tcpap92 [new Application/TcpApp $tcp92]

for {set i 0} {$i < 10} {incr i} {
        set app($i) [new FloodApp]
        $app($i) id [$n($i) id]
        $app($i) log $log
        $app($i) set diam_ 4
}

#Atribui UID às aplicações
$app(0) uid 10
$app(1) uid 20
$app(2) uid 50
$app(3) uid 40
$app(4) uid 80
$app(5) uid 30
$app(6) uid 90
$app(7) uid 70
$app(8) uid 30
$app(9) uid 60
```

FIGURA 5.19 - Exemplo de *script* de simulação do algoritmo Floodmax (continuação)

```
#Define a quantidade de vizinhos de cada aplicação
$app(0) set neighbor_ 1
$app(1) set neighbor_ 4
$app(2) set neighbor_ 3
$app(3) set neighbor_ 1
$app(4) set neighbor_ 3
$app(5) set neighbor_ 3
$app(6) set neighbor_ 3
$app(7) set neighbor_ 3
$app(8) set neighbor_ 1
$app(9) set neighbor_ 2

$tcpap01 connect $tcpap11
$tcpap12 connect $tcpap21
$tcpap13 connect $tcpap31
$tcpap14 connect $tcpap41
$tcpap22 connect $tcpap61
$tcpap23 connect $tcpap71
$tcpap42 connect $tcpap51
$tcpap43 connect $tcpap91
$tcpap52 connect $tcpap62
$tcpap53 connect $tcpap92
$tcpap63 connect $tcpap73
$tcpap72 connect $tcpap81

#Conecta aplicações
$app(0) connect $app(1) $tcpap01
$app(1) connect $app(0) $tcpap11
$app(1) connect $app(2) $tcpap12
$app(1) connect $app(3) $tcpap13
$app(1) connect $app(4) $tcpap14
$app(2) connect $app(1) $tcpap21
$app(2) connect $app(6) $tcpap22
$app(2) connect $app(7) $tcpap23
$app(3) connect $app(1) $tcpap31
$app(4) connect $app(1) $tcpap41
$app(4) connect $app(5) $tcpap42
$app(4) connect $app(9) $tcpap43
$app(5) connect $app(4) $tcpap51
$app(5) connect $app(6) $tcpap52
$app(5) connect $app(9) $tcpap53
$app(6) connect $app(2) $tcpap61
$app(6) connect $app(5) $tcpap62
$app(6) connect $app(7) $tcpap63
$app(7) connect $app(2) $tcpap71
$app(7) connect $app(8) $tcpap72
$app(7) connect $app(6) $tcpap73
$app(8) connect $app(7) $tcpap81
$app(9) connect $app(4) $tcpap91
$app(9) connect $app(5) $tcpap92

#Inicia a eleição
$ns at 0.1 "$app(0) send  40"
$ns at 0.1 "$app(1) send  40"
$ns at 0.1 "$app(2) send  40"
$ns at 0.1 "$app(3) send  40"
$ns at 0.1 "$app(4) send  40"
$ns at 0.1 "$app(5) send  40"
$ns at 0.1 "$app(6) send  40"
$ns at 0.1 "$app(7) send  40"
$ns at 0.1 "$app(8) send  40"
$ns at 0.1 "$app(9) send  40"

#Chama procedimento finish
$ns at 0.8 "finish"

#Executa a simulação
$ns run
```

FIGURA 5.19 - Exemplo de *script* de simulação do algoritmo Floodmax (continuação)

```
0.1 src 0 UID 10  s
0.1 src 1 UID 20  s
0.1 src 1 UID 20  s
0.1 src 1 UID 20  s
0.1 src 1 UID 20  s
0.1 src 2 UID 50  s
0.1 src 2 UID 50  s
0.1 src 2 UID 50  s
0.1 src 3 UID 40  s
0.1 src 4 UID 80  s
0.1 src 4 UID 80  s
0.1 src 4 UID 80  s
0.1 src 5 UID 30  s
0.1 src 5 UID 30  s
0.1 src 5 UID 30  s
0.1 src 6 UID 90  s
0.1 src 6 UID 90  s
0.1 src 6 UID 90  s
0.1 src 7 UID 70  s
0.1 src 7 UID 70  s
0.1 src 7 UID 70  s
0.1 src 8 UID 0  s
0.1 src 9 UID 60  s
0.1 src 9 UID 60  s
0.13128 src 3 UID 40  s
0.13128 src 0 UID 20  s
0.13128 src 1 UID 80  s
0.13128 src 1 UID 80  s
0.13128 src 1 UID 80  s
0.13128 src 1 UID 80  s
0.13128 src 9 UID 80  s
0.13128 src 9 UID 80  s
0.13128 src 6 UID 90  s
0.13128 src 6 UID 90  s
0.13128 src 6 UID 90  s
0.13128 src 8 UID 70  s
0.13128 src 2 UID 90  s
0.13128 src 2 UID 90  s
0.13128 src 2 UID 90  s
0.13128 src 7 UID 90  s
0.13128 src 7 UID 90  s
0.13128 src 7 UID 90  s
0.13128 src 5 UID 90  s
0.13128 src 5 UID 90  s
0.13128 src 5 UID 90  s
0.13128 src 4 UID 80  s
0.13128 src 4 UID 80  s
0.13128 src 4 UID 80  s
0.21064 src 0 UID 80  s
0.21064 src 1 UID 90  s
0.21064 src 1 UID 90  s
0.21064 src 1 UID 90  s
0.21064 src 1 UID 90  s
0.21064 src 2 UID 90  s
0.21064 src 2 UID 90  s
0.21064 src 2 UID 90  s
0.21064 src 3 UID 80  s
0.21064 src 4 UID 90  s
0.21064 src 4 UID 90  s
0.21064 src 4 UID 90  s
0.21064 src 5 UID 90  s
0.21064 src 5 UID 90  s
0.21064 src 5 UID 90  s
0.21064 src 6 UID 90  s
0.21064 src 6 UID 90  s
0.21064 src 6 UID 90  s
0.21064 src 7 UID 90  s
0.21064 src 7 UID 90  s
0.21064 src 7 UID 90  s
0.21064 src 8 UID 90  s
0.21064 src 9 UID 90  s
0.21064 src 9 UID 90  s
0.31064 src 0 UID 90  s
0.31064 src 1 UID 90  s
0.31064 src 1 UID 90  s
```

FIGURA 5.20 - Exemplo de arquivo de *log* gerado por uma aplicação `FloodApp` (continua)

```
0.31064 src 1 UID 90  s
0.31064 src 1 UID 90  s
0.31064 src 2 UID 90  s
0.31064 src 2 UID 90  s
0.31064 src 2 UID 90  s
0.31064 src 3 UID 90  s
0.31064 src 4 UID 90  s
0.31064 src 4 UID 90  s
0.31064 src 4 UID 90  s
0.31064 src 5 UID 90  s
0.31064 src 5 UID 90  s
0.31064 src 5 UID 90  s
0.31064 src 6 UID 90  s
0.31064 src 6 UID 90  s
0.31064 src 6 UID 90  s
0.31064 src 7 UID 90  s
0.31064 src 7 UID 90  s
0.31064 src 7 UID 90  s
0.31064 src 8 UID 90  s
0.31064 src 9 UID 90  s
0.31064 src 9 UID 90  s
0.41064 src 0 Not Leader - UID 10
0.41064 src 1 Not Leader - UID 20
0.41064 src 2 Not Leader - UID 50
0.41064 src 3 Not Leader - UID 40
0.41064 src 4 Not Leader - UID 80
0.41064 src 5 Not Leader - UID 30
0.41064 src 6 Leader - UID 90
0.41064 src 7 Not Leader - UID 70
0.41064 src 8 Not Leader - UID 0
0.41064 src 9 Not Leader - UID 60
```

FIGURA 5.20 - Exemplo de arquivo de *log* gerado por uma aplicação `FloodApp` (continuação)

O número de nodos deste exemplo e a configuração da topologia foram escolhidos de forma arbitrária. Além deste, foram feitos outros experimentos com diferentes combinações de topologias e identificadores de processos e em todos eles obteve-se os resultados esperados: um processo declarou-se líder e os demais declararam que não eram líderes no número de rodadas estabelecido. Alguns desses exemplos, nos quais a quantidade máxima de nodos testados foi vinte, são apresentados no Anexo.

5.2 Algoritmo assíncrono para eleição de líder em rede arbitrária

Nessa seção, será apresentada a implementação do algoritmo de inundação em redes assíncronas. Este algoritmo foi desenvolvido utilizando-se tanto o modelo baseado em TCP-*unicast* quanto o modelo baseado em UDP-*multicast*. Primeiro, será enunciada sua definição e, após, será explicada como foi feita sua implementação com esses dois modelos. Esse algoritmo não é tolerante a falhas, isto é, sua implementação não trata a ocorrência de defeitos, portanto, não foram testados casos com ocorrência destes. Caso algum dos nodos apresente defeito, provavelmente, a eleição não será finalizada.

5.2.1 Definição do algoritmo de inundação assíncrono

O algoritmo de inundação assíncrono para redes arbitrárias difere um pouco do algoritmo de inundação síncrono (Floodmax) apresentado na seção 5.1.2,

principalmente, porque não existem rodadas em algoritmos assíncronos. Nessa seção, será relembrada a definição de Floodmax, após, será explicado como converter esse algoritmo para um algoritmo assíncrono e, em seguida, será apresentada a implementação do algoritmo de inundação assíncrono.

Para o algoritmo de inundação assíncrono, é assumido que a rede de comunicação é bidirecional e que todos os processos (aplicações) são idênticos só sendo diferenciados pelo seu UID. No algoritmo para redes síncronas (Floodmax), cada processo mantém armazenado o máximo UID conhecido e, em cada rodada síncrona, o processo envia o UID máximo para todos os seus vizinhos. O algoritmo termina quando é atingido o número de rodadas correspondente ao maior diâmetro (*diam*) da rede. Um único processo anuncia-se líder se, ao final das rodadas, seu UID é igual ao máximo UID conhecido por ele.

Como foi dito anteriormente, no Floodmax assíncrono, não existem rodadas, mas é possível simulá-las. Para isso, é preciso que cada processo envie o número da rodada *r* na mensagem. O processo receptor espera até receber as mensagens da rodada *r* de todos os seus vizinhos, antes de fazer a transição para a próxima rodada.

5.2.2 Implementação com TCP

Para implementar esse algoritmo, foi desenvolvida uma classe derivada de `AppData` (como nos outros algoritmos explicados anteriormente), que tem a função de definir a ADU. Serão inseridos na ADU tanto o valor de UID quanto o número da rodada a que a mensagem pertence. Esta classe, denominada `AsfloodData` (Figura 5.21), é usada tanto para TCP quanto para UDP.

```
class AsfloodData : public AppData {
   private:
        int cost_;
        int uid_;
        int r_;
        int id_;
   public:
        AsfloodData(int id, int cost, int uid, int r) :
                AppData(ASFLOOD_DATA) {
                cost_ = cost;
                uid_ = uid;
                r_=r;
                id_ = id;
        }
        AsfloodData(AsfloodData& d) : AppData(d) {
                cost_ = d.cost_;
                uid_=d.uid_;
                r_=d.r_;
                id_=d.id_;
        }
        virtual ~AsfloodData() {
        }
        virtual int size() const {
                return (AppData::size()+4*sizeof(int));
        }
        virtual int cost() const { return cost_; }
        int uid() { return uid_; }
        int id() { return id_; }
        int r() { return r_; }
        virtual AppData* copy() { return (new AsfloodData(*this)); }
};
```

FIGURA 5.21 - Classe `AsfloodData`

Para o desenvolvimento da aplicação com TCP, foi criada a classe `AsFloodApp`, onde foi implementado o algoritmo de inundação assíncrono. A seguir, serão apresentados os principais métodos dessa classe.

No método `command()` (Figura 5.22), os principais comandos são: `uid`, `diam`, `neighbor`, `connect` e `send`. O comando `uid` (Figura 5.22, linha 42) atribui a uma variável (`myuid_`) o identificador único (UID) estabelecido no *script* OTcl, assim como atribui a `maxid_` o mesmo valor. O comando `diam` (Figura 5.22, linha 27) atribui à variável `diam_` o valor de *diam* especificado no *script*. Já o comando `neighbor` (Figura 5.22, linha 34) atribui à variável `neigh_` a quantidade de vizinhos da aplicação. O comando `connect` (Figura 5.22, linha 62) é feito da mesma forma que na implementação do algoritmo LCR explicado na seção 5.1.1. O comando `send` chama o método `send()` (Figura 5.23), que é implementado da mesma forma que nos algoritmos LCR e Floodmax, tendo como diferença somente o tipo da unidade de dados (ADU – Figura 5.23, linha 8). Os métodos `log` e `add_cnc` são implementados da mesma forma que no algoritmo LCR.

```
1:   int AsFloodApp::command(int argc, const char*const* argv)
2:   {
3:        Tcl& tcl = Tcl::instance();
4:        if (argc == 3) {
5:             if (strcmp(argv[1], "id") == 0) {
6:                  /*
7:                   * <app> id <node_id>
8:                   */
9:                  id_ = atoi(argv[2]);
10:                 tcl.resultf("%d", id_);
11:                 return TCL_OK;
12:            }
13:            else if (strcmp(argv[1], "log") == 0) {
14:                 /*
15:                  * <app> log <log_file>
16:                  */
17:                 int mode;
18:                 log_ = Tcl_GetChannel(tcl.interp(),
19:                             (char*)argv[2], &mode);
20:                 if (log_ == 0) {
21:                        tcl.resultf("%d: invalid log file handle %s\n",
22:                            id_, argv[2]);
23:                        return TCL_ERROR;
24:                 }
25:                 return TCL_OK;
26:            }
27:            else if (strcmp(argv[1], "diam") == 0) {
28:                 /*
29:                  * <app> diam <diam>
30:                  */
31:                 diam_=atoi(argv[2]);
32:                 return (TCL_OK);
33:            }
34:            else if (strcmp(argv[1], "neighbor") == 0) {
35:                 /*
36:                  * <app> neighbor <n_neighbor>
37:                  */
38:                 neigh_=atoi(argv[2]);
39:                 memset(ro_, 0, 10);  //máximo 10 rodadas
40:                 return (TCL_OK);
41:            }
42:            else if (strcmp(argv[1], "uid") == 0) {
43:                 /*
44:                  * <app> uid <uid>
45:                  */
46:                 myuid_=atoi(argv[2]);
47:                 maxid_=myuid_;
48:                 return (TCL_OK);
49:            }
```

FIGURA 5.22 - Método `command` da classe `Asfloodapp` (continua)

```
50:             else if (strcmp(argv[1], "send") == 0) {
51:                     /*
52:                      * <app> send <bytes>
53:                      * Aplicação <app> chama função send que faz
54:                      * o envio de dados para todas as aplicações
55:                      * que <app> está conectada
56:                      */
57:                     bytes_ = atoi(argv[2]);
58:                     send();
59:                     return (TCL_OK);
60:             }
61:     }else if (argc == 4) {
62:             if (strcmp(argv[1], "connect") == 0) {
63:                     /*
64:                      * <app> connect <appclient> <ts>
65:                      *
66:                      * Associa um agente TcpApp <tcpap> com a aplicação
67:                      * cliente <appclient>. O agente TCP é usado para
68:                      * enviar pacotes para a aplicação cliente.
69:                      */
70:                     AsFloodApp *client =
71:                             (AsFloodApp *)TclObject::lookup(argv[2]);
72:
73:                     TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
74:                     if (add_cnc(client, cnc)) {
75:                             tcl.resultf("%s: failed to connect to %s",
76:                                     name_, argv[2]);
77:                             return TCL_ERROR;
78:                     }
79:                    /*
80:                     *Estabelece o alvo de entrega dos dados,
81:                     *isto é, a própria aplicação
82:                     */
83:                      cnc->target() = (Process*)this;
84:                     return TCL_OK;
85:             }
86:     }
87:     return TclObject::command(argc, argv);
88: }
```

FIGURA 5.22 - Método `command` da classe `Asfloodapp` (continuação)

```
1:  void AsFloodApp::send()
2:  {
3:      Tcl_HashEntry *he;
4:      Tcl_HashSearch hs;
5:    for (he = Tcl_FirstHashEntry(tpa_, &hs);  he != NULL; he = Tcl_NextHashEntry(&hs))
6:    {
7:       TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
8:       AsfloodData *d = new AsfloodData(id_, bytes_, maxid_, rounds_);
9:       log("UID %d  s \n", maxid_);
10:      cnc->send(bytes_, d);
11:   }
12: }
```

FIGURA 5.23 - Método send da classe `Asfloodapp`

O método `process_data` (Figura 5.24) recebe os dados, faz sua interpretação e, após, se necessário chama `send()` para envio de novos dados. Nesse método, inicialmente é feita a verificação do tipo de dado recebido; se este é `ASFLOOD_DATA` (Figura 5.24, linha 5) então são buscados os valores de UID e da rodada associada à mensagem. Após, é implementado um vetor, indexado pelo número da rodada, que armazena quantas mensagens foram recebidas em cada rodada (Figura 5.24, linhas 9 a 12). Quando esse valor for igual ao número de vizinhos (Figura 5.24, linha 13), significa que este processo recebeu as mensagens da rodada *r* de todos os seus vizinhos; só então ele pode ir para a próxima rodada.

Após receber as mensagens de todos os seus vizinhos, se o número de rodadas (`rounds_`) for menor que `diam_` (Figura 5.24, linha 15), o processo envia o UID máximo encontrado para todos os seus vizinhos (método `send`). Quando o número de rodadas for igual ao valor de `diam_`, é feita a comparação entre o maior UID conhecido pela aplicação e o seu UID: se os valores forem iguais, a aplicação declara-se líder, senão se declara não-líder (Figura 5.24, linhas 17 a 30).

```
1:   void Asfloodapp::process_data(int, AppData* data)
2:   {
3:       if (data == NULL)
4:             return;

5:       if (data->type() == ASFLOOD_DATA) {
6:           AsfloodData *tmp = (AsfloodData*)data;
7:           int uid=tmp->uid();
8:           int r=tmp->r();
9:           if (ro_[r]<neigh_){ //espera até receber mensagens da rodada r de todos
10:              ro_[r]++;   //os vizinhos
11:              if (maxid_ < uid) maxid_=uid;//se uid recebido é maior que o uid
12:          }                                    //maximo conhecido maxid=uid)
13:          if (ro_[r] == neigh_) {//se recebeu mensagens da rodada r de todos os vizin
14:              rounds_++;    //faz transição para próxima rodada
15:              if (rounds_ < diam_) //se o número de rodadas for menor que diam
16:                  send();  //envia máximo uid para todos os vizinhos
17:              if(rounds_== diam_)  {  //se o número de rodadas é igual a diam
18:                  Tcl& tcl = Tcl::instance();
19:                  if(maxid_ == myuid_) { //se uid do processo é igual ao maxid
20:                      log("Leader – UID %d\n", myuid_); //se declara não líder
21:                      tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1 circle",
22:                             id_, id_); //insere círculo ao redor do nodo líder no Nam
23                       tcl.evalf("$n(%d) label Leader", id_); //insere rótulo “Leader”
24:                      tcl.evalf("$ns trace-annotate \"Leader - UID %d\"", myuid_);
25:                  } else {
26:                      log("Not Leader - UID %d\n", myuid_); //se declara líder
27:                      tcl.evalf("$ns trace-annotate \"Not Leader - UID %d\"", myuid_);
28:                    }
29:              }
30:         }
31:      }
32:      else {
33:              fprintf(stderr, "Bad data type %d\n", data->type());
34:              abort();
35:      }
36:  }
```

FIGURA 5.24 - Método `process_data` da classe `Asfloodapp`

Exemplo de simulação

A Figura 5.25 apresenta a tela inicial do Nam de um exemplo de simulação do algoritmo Floodmax assíncrono com 8 nodos. A topologia desse exemplo foi escolhida de forma arbitrária. As aplicações iniciam a execução em tempos diferentes para simular um sistema assíncrono. Como o diâmetro da rede é quatro, depois de quatro rodadas, o processo 5 (UID igual a 50) declara-se líder e cada um dos demais declara-se não-líder. O *script* OTcl dessa simulação é mostrado na Figura 5.26 e seu arquivo de *log* na Figura 5.27.

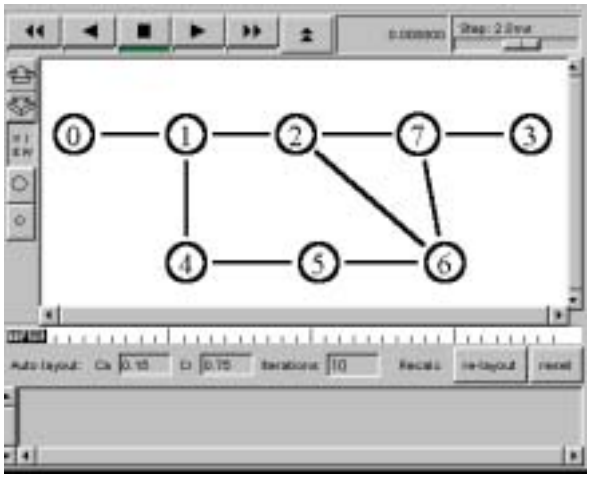


FIGURA 5.25 - Tela do Nam de um exemplo de Floodmax assíncrono

```
#Create a simulator object
set ns [new Simulator]

#Open a trace file
set f [open out.tr w]
$ns trace-all $f
set nf [open asflood8.nam w]
$ns namtrace-all $nf
set log [open asflood8.log w]

$ns color 1 blue
$ns color 0 red
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow
$ns color 5 green
$ns color 6 HotPink
$ns color 5 green
$ns color 7 cyan

#Define procedimento finish
proc finish {} {
        global ns nf f log
        $ns flush-trace
        close $nf
        close $f
        close $log
        exec nam out1.nam &
        exit 0
}

#Cria 8 nodos
for {set i 0} {$i < 8} {incr i} {
        set n($i) [$ns node]
}

#Conecta os nodos
$ns duplex-link $n(0) $n(1) 1Mb 10ms DropTail
$ns duplex-link $n(1) $n(2) 1Mb 10ms DropTail
$ns duplex-link $n(1) $n(4) 1Mb 10ms DropTail
$ns duplex-link $n(4) $n(5) 1Mb 10ms DropTail
$ns duplex-link $n(2) $n(6) 1Mb 10ms DropTail
$ns duplex-link $n(5) $n(6) 1Mb 10ms DropTail
$ns duplex-link $n(2) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(6) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(3) $n(7) 1Mb 10ms DropTail

#Cria agentes FullTcp e os anexa aos nodos
set tcp01 [new Agent/TCP/FullTcp]
$ns attach-agent $n(0) $tcp01
$tcp01 set fid_ 0

set tcp11 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp11
$tcp11 set fid_ 1
set tcp12 [new Agent/TCP/FullTcp]
```

FIGURA 5.26 - Exemplo de *script* de simulação de Floodmax assíncrono com TCP (continua)

```
$ns attach-agent $n(1) $tcp12
$tcp12 set fid_ 1

set tcp13 [new Agent/TCP/FullTcp]
$ns attach-agent $n(1) $tcp13
$tcp13 set fid_ 1

set tcp21 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp21
$tcp21 set fid_ 2

set tcp22 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp22
$tcp22 set fid_ 2

set tcp23 [new Agent/TCP/FullTcp]
$ns attach-agent $n(2) $tcp23
$tcp23 set fid_ 2

set tcp31 [new Agent/TCP/FullTcp]
$ns attach-agent $n(3) $tcp31
$tcp31 set fid_ 3

set tcp41 [new Agent/TCP/FullTcp]
$ns attach-agent $n(4) $tcp41
$tcp41 set fid_ 4

set tcp42 [new Agent/TCP/FullTcp]
$ns attach-agent $n(4) $tcp42
$tcp42 set fid_ 4

set tcp51 [new Agent/TCP/FullTcp]
$ns attach-agent $n(5) $tcp51
$tcp51 set fid_ 5

set tcp52 [new Agent/TCP/FullTcp]
$ns attach-agent $n(5) $tcp52
$tcp52 set fid_ 5

set tcp61 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp61
$tcp61 set fid_ 6

set tcp62 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp62
$tcp62 set fid_ 6

set tcp63 [new Agent/TCP/FullTcp]
$ns attach-agent $n(6) $tcp63
$tcp63 set fid_ 6

set tcp71 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp71
$tcp71 set fid_ 7

set tcp72 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp72
$tcp72 set fid_ 7

set tcp73 [new Agent/TCP/FullTcp]
$ns attach-agent $n(7) $tcp73
$tcp73 set fid_ 7

#Conecata agentes tcp
$ns connect $tcp01 $tcp11
$ns connect $tcp12 $tcp21
$ns connect $tcp13 $tcp41
$ns connect $tcp22 $tcp61
$ns connect $tcp23 $tcp71
$ns connect $tcp31 $tcp72
$ns connect $tcp42 $tcp51
$ns connect $tcp52 $tcp62
$ns connect $tcp63 $tcp73
set tcpap01 [new Application/TcpApp $tcp01]
set tcpap11 [new Application/TcpApp $tcp11]
set tcpap12 [new Application/TcpApp $tcp12]
set tcpap13 [new Application/TcpApp $tcp13]
set tcpap21 [new Application/TcpApp $tcp21]
set tcpap22 [new Application/TcpApp $tcp22]
set tcpap23 [new Application/TcpApp $tcp23]
set tcpap31 [new Application/TcpApp $tcp31]
```

FIGURA 5.26 - Exemplo de *script* de simulação de Floodmax assíncrono com TCP (continuação)

```
set tcpap41 [new Application/TcpApp $tcp41]
set tcpap42 [new Application/TcpApp $tcp42]
set tcpap51 [new Application/TcpApp $tcp51]
set tcpap52 [new Application/TcpApp $tcp52]
set tcpap61 [new Application/TcpApp $tcp61]
set tcpap62 [new Application/TcpApp $tcp62]
set tcpap63 [new Application/TcpApp $tcp63]
set tcpap71 [new Application/TcpApp $tcp71]
set tcpap72 [new Application/TcpApp $tcp72]
set tcpap73 [new Application/TcpApp $tcp73]
for {set i 0} {$i < 8} {incr i} {
        set app($i) [new AsFloodApp]
        $app($i) id [$n($i) id]
        $app($i) log $log
        $app($i) diam 4
}
#Atribui UID de cada aplicação
$app(0) uid 10
$app(1) uid 20
$app(2) uid 45
$app(3) uid 15
$app(4) uid 35
$app(5) uid 50
$app(6) uid 25
$app(7) uid 40
#Atribui a quantidade de vizinhos de cada aplicação
$app(0) neighbor 1
$app(1) neighbor 3
$app(2) neighbor 3
$app(3) neighbor 1
$app(4) neighbor 2
$app(5) neighbor 2
$app(6) neighbor 3
$app(7) neighbor 3

$tcpap01 connect $tcpap11
$tcpap12 connect $tcpap21
$tcpap13 connect $tcpap41
$tcpap22 connect $tcpap61
$tcpap23 connect $tcpap71
$tcpap31 connect $tcpap72
$tcpap42 connect $tcpap51
$tcpap52 connect $tcpap62
$tcpap63 connect $tcpap73

$app(0) connect $app(1) $tcpap01
$app(1) connect $app(0) $tcpap11
$app(1) connect $app(2) $tcpap12
$app(1) connect $app(4) $tcpap13
$app(2) connect $app(1) $tcpap21
$app(2) connect $app(6) $tcpap22
$app(2) connect $app(7) $tcpap23
$app(3) connect $app(7) $tcpap31
$app(4) connect $app(1) $tcpap41
$app(4) connect $app(5) $tcpap42
$app(5) connect $app(4) $tcpap51
$app(5) connect $app(6) $tcpap52
$app(6) connect $app(2) $tcpap61
$app(6) connect $app(5) $tcpap62
$app(6) connect $app(7) $tcpap63
$app(7) connect $app(2) $tcpap71
$app(7) connect $app(3) $tcpap72
$app(7) connect $app(6) $tcpap73

#Inicia eleição
$ns at 1.2 "$app(0) send  40"
$ns at 0.1 "$app(1) send  40"
$ns at 0.5 "$app(2) send  40"
$ns at 1.7 "$app(3) send  40"
$ns at 2.6 "$app(4) send  40"
$ns at 2.0 "$app(5) send  40"
$ns at 1.1 "$app(6) send  40"
$ns at 3.3 "$app(7) send  40"

$ns at 4.0 "finish"
$ns run
```

FIGURA 5.26 - Exemplo de *script* de simulação de Floodmax assíncrono com TCP (continuação)

```
0.1 src 1 UID 20  s
0.1 src 1 UID 20  s
0.1 src 1 UID 20  s
0.5 src 2 UID 45  s
0.5 src 2 UID 45  s
0.5 src 2 UID 45  s
1.1 src 6 UID 25  s
1.1 src 6 UID 25  s
1.1 src 6 UID 25  s
1.2 src 0 UID 10  s
1.2416 src 0 UID 20  s
1.7 src 3 UID 15  s
2 src 5 UID 50  s
2 src 5 UID 50  s
2.6 src 4 UID 35  s
2.6 src 4 UID 35  s
2.6316 src 5 UID 50  s
2.6316 src 5 UID 50  s
2.6316 src 1 UID 45  s
2.6316 src 1 UID 45  s
2.6316 src 1 UID 45  s
2.6416 src 4 UID 50  s
2.6416 src 4 UID 50  s
2.64224 src 0 UID 45  s
2.71064 src 4 UID 50  s
2.71064 src 4 UID 50  s
3.3 src 7 UID 40  s
3.3 src 7 UID 40  s
3.3 src 7 UID 40  s
3.3316 src 6 UID 50  s
3.3316 src 6 UID 50  s
3.3316 src 6 UID 50  s
3.3316 src 3 UID 40  s
3.3316 src 2 UID 45  s
3.3316 src 2 UID 45  s
3.3316 src 2 UID 45  s
3.3416 src 7 UID 45  s
3.3416 src 7 UID 45  s
3.3416 src 7 UID 45  s
3.34224 src 5 UID 50  s
3.34224 src 5 UID 50  s
3.34224 src 1 UID 50  s
3.34224 src 1 UID 50  s
3.34224 src 1 UID 50  s
3.35288 src 4 UID 50  s
3.35288 src 4 UID 50  s
3.35288 src 0 UID 50  s
3.41064 src 7 UID 50  s
3.41064 src 7 UID 50  s
3.41064 src 7 UID 50  s
3.41064 src 6 UID 50  s
3.41064 src 6 UID 50  s
3.41064 src 6 UID 50  s
3.41064 src 3 UID 45  s
3.41064 src 2 UID 50  s
3.41064 src 2 UID 50  s
3.41064 src 2 UID 50  s
3.42128 src 5 UID 50  s
3.42128 src 5 UID 50  s
3.42128 src 1 UID 50  s
3.42128 src 1 UID 50  s
3.42128 src 1 UID 50  s
3.43192 src 4 Not Leader - UID 35
3.43192 src 0 Not Leader - UID 10
3.51064 src 7 UID 50  s
3.51064 src 7 UID 50  s
3.51064 src 7 UID 50  s
3.51064 src 6 UID 50  s
3.51064 src 6 UID 50  s
3.51064 src 6 UID 50  s
3.51064 src 3 UID 50  s
3.51064 src 2 UID 50  s
3.51064 src 2 UID 50  s
3.51064 src 2 UID 50  s
```

FIGURA 5.27 - Exemplo de arquivo de *log* gerado por uma aplicação `AsFloodApp` (continua)

```
3.52128 src 5 Leader - UID 50
3.52128 src 1 Not Leader - UID 20
3.61064 src 7 Not Leader - UID 40
3.61064 src 6 Not Leader - UID 25
3.61064 src 3 Not Leader - UID 15
3.61064 src 2 Not Leader - UID 45
```

FIGURA 5.27 - Exemplo de arquivo de *log* gerado por uma aplicação `AsFloodApp` (continuação)

5.2.3 Implementação com UDP

Para a implementação desse algoritmo com UDP, utiliza-se, como com TCP, a classe auxiliar `AsfloodData` (Figura 5.21) onde é definida a unidade de dados - ADU. Além dessa classe, foi desenvolvida uma classe derivada de UDP chamada `UdpDataAgent` (seção 4.2), na qual foi criado um novo tipo de pacote `UDPDATA`, o método `send` foi adicionado e o método `recv` foi sobrescrito. Esses métodos são mostrados na Figura 5.28 e na Figura 5.29, respectivamente.

```
1:  // realsize: tamanho declarado pelo usuário
2:  // datasize: tamanho real do dado do usuário, usado para alocar o pacote
3:  void UdpDataAgent::send(int realsize, AppData* data)
4:  {
5:      Packet *pkt = allocpkt(data->size());
6:      hdr_udata *ih = hdr_udata::access(pkt);
7:      ih->size() = data->size();
8:      pkt->setdata(data); // coloca dados no pacote

9:     // Estabelece o tamanho do pacote proporcional a quantidade de dados udpdata
10:    hdr_cmn *ch = hdr_cmn::access(pkt);
11:    ch->size() = udata_hdr_size_ + realsize;
12:    Agent::send(pkt, 0);
13: }
```

FIGURA 5.28 - Método `send` da classe `UdpDataAgent`

O método `send` faz o envio dos dados para o agente, que os envia para o agente da aplicação destino. Para isso, os dados do usuário são colocados em um pacote (linhas 5 a 8 na Figura 5.28) e, após, o pacote é enviado para o agente - `Agent::send(pkt, 0)` (linha 12 na Figura 5.28).

```
1:  void UdpDataAgent::recv(Packet *pkt, Handler*)
2:  {
3:      hdr_ip *ip = hdr_ip::access(pkt);
4:      if ((ip->saddr() == addr()) && (ip->sport() == port()))
5:              return;
6:      if (app_ == 0)
7:              return;
8:      hdr_udata *ih = hdr_udata::access(pkt);
9:      ((AsFloodAppUdp*)app_)->process_data(ih->size(), pkt->userdata());
10:     Packet::free(pkt);
11:  }
```

FIGURA 5.29 - Método `recv` da classe `UdpDataAgent`

O método `recv()` faz com que os dados sejam passados do agente de transporte (UDP) para a aplicação. Nesse caso, é feita a chamada do método `process_data` da aplicação com o comando: `((AsFloodAppUdp*)app_)->process_data(ih->size(), pkt-> userdata())` – Figura 5.29, linha 9. Assim, é possível fazer a leitura dos dados recebidos na aplicação.

A seguir, foi desenvolvida a aplicação (algoritmo Floodmax assíncrono) na classe denominada `AsFloodAppUdp`. Esta classe difere em poucas coisas da classe `AsFloodApp` da implementação com TCP. Uma das modificações está no método `command`, no qual o comando `connect` foi substituído por `attach_udp` (linha 5, Figura 5.30). Outra diferença está na implementação do método `send`. A parte do método `command` que difere da implementação com TCP e o método `send` desta classe são apresentados na Figura 5.30.

```
1:   int AsFloodAppUdp::command(int argc, const char*const* argv)
2:   {
3:         Tcl& tcl = Tcl::instance();
           .
           .
           . // indicam que a implementação dessa classe não é apresentada completa

4:         if (argc == 3) {
5:           if (strcmp(argv[1], "attach_udp") == 0) {
6:                   /*
7:                    * <app> attach_udp <udp>
8:                    */
9:                   UdpDataAgent *tmp =
10:                          (UdpDataAgent *)TclObject::lookup(argv[2]);
11:                  // Anexa agente UDP (tmp) a esta aplicação
12:                  tmp->attachApp((Application *)this);
13:                  int lenudpat = strlen(argv[2]) + 1;
14:                  if (lenudpat > 0) {
15:                     udpat_ = new char[lenudpat];
16:                     strcpy(udpat_, argv[2]);
17:                  } else
18:                          udpat_ = NULL;
19:                  return (TCL_OK);
20:          }
21:      }
22:    return TclObject::command(argc, argv);
24:  }

25:  void AsFloodAppUdp::send()
26:  {
27:    //procura objeto udpat_ que é o agente de UDP anexado à aplicação
28:    UdpDataAgent *cnc = (UdpDataAgent *)TclObject::lookup(udpat_);
29:    // insere os dados na ADU
30:    AsfloodData *d = new AsfloodData(id_, bytes_, maxid_, rounds_);
31:    log("UID %d  s \n", maxid_);
32:    cnc->send(bytes_, d);  //chama o método send do agente UDP
33: }
```

FIGURA 5.30 - Métodos `command` e `send` da classe `AsFloodAppUdp`

Dentro do método `command`, o comando `attach_udp` (Figura 5.30, linha 5), que é definido no *script* OTcl como `<app> attach_udp <udp>`, tem a função de anexar a aplicação (`<app>`) ao agente UDP (`<udp>`). A variável `udpat_` armazena o agente UDP que foi anexado à aplicação (Figura 5.30, linha 12). Já o comando `send` chama o método `send()`(Figura 5.30, linhas 25 a 33), que insere os dados na ADU e os envia para o agente de transporte, que se encarrega de enviá-los ao agente de transporte da(s) aplicação(ões) destino. Como, nesse caso, o transporte é feito por *multicast*, o destino do agente de transporte (UDP) é um grupo. Por isso, a aplicação UDP não precisa armazenar seus clientes em uma tabela de aplicações clientes e agentes `TcpApp`, como no caso do TCP. O método `process_data`, que recebe os dados na aplicação, é idêntico ao método implementado com TCP.

Exemplo de simulação

A Figura 5.31 apresenta uma tela do Nam, que mostra o final da eleição de um exemplo de simulação do algoritmo Floodmax assíncrono com UDP, no qual a topologia foi escolhida de forma arbitrária. Neste exemplo, são criados treze nodos, sendo que em dois deles (nodo 0 e nodo 5) não foram incluídos o agente de transporte e a aplicação. Portanto, esses nodos não processam os dados, só os enviam para as aplicações às quais eles estão destinados. O diâmetro (*diam*) dessa rede é seis, portanto, depois de seis rodadas, o processo do nodo 6, que tem UID igual a 80, declara-se líder e cada um dos demais declara-se não-líder. O *script* OTcl dessa simulação é mostrado na Figura 5.32 e seu arquivo de *log* na Figura 5.33.

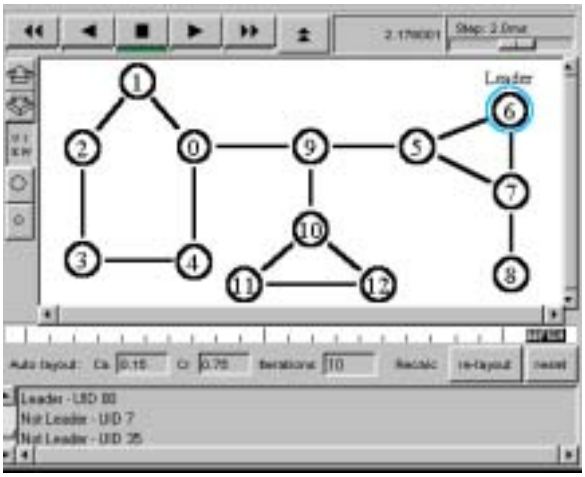


FIGURA 5.31 - Tela do Nam de um exemplo de simulação de Floodmax assíncrono com UDP

```
source /root/asfloodb/topo1.tcl

#Criação do objeto Simulator
set ns [new Simulator -multicast on]

#Abertura de arquivos
set f [open asfloodu2.tr w]    ;# Arquivo de rastro
$ns trace-all $f
set nf [open asfloodu2.nam w]  ;# Arquivo do Nam
$ns namtrace-all $nf
set log [open asfloodu2.log w] ;# Arquivo de log

$ns color 1 blue
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow
$ns color 5 green
$ns color 6 HotPink
$ns color 7 cyan
$ns color 8 gold1
$ns color 9 OrangeRed1
$ns color 10 DarkOrchid3
$ns color 11 brown
$ns color 12 HotPink
$ns color 13 red

#Definição do procedimento 'finish'
proc finish {} {
        global ns nf f log
        $ns flush-trace
        close $nf
        close $f
         close $log
        exec nam asfloodu2.nam &
        exit 0
}
```

FIGURA 5.32 - Exemplo de *script* de simulação de Floodmax assíncrono com UDP (continua)

```
#Criação dos nodos
create-topology 5 ring 1Mb 10ms DropTail
for {set i 5} {$i < 13} {incr i} {
        set n($i) [$ns node]
}
#Conexão dos nodos
$ns duplex-link $n(0) $n(9) 1Mb 10ms DropTail
$ns duplex-link $n(9) $n(10) 1Mb 10ms DropTail
$ns duplex-link $n(9) $n(5) 1Mb 10ms DropTail
$ns duplex-link $n(10) $n(11) 1Mb 10ms DropTail
$ns duplex-link $n(10) $n(12) 1Mb 10ms DropTail
$ns duplex-link $n(11) $n(12) 1Mb 10ms DropTail
$ns duplex-link $n(5) $n(6) 1Mb 10ms DropTail
$ns duplex-link $n(5) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(6) $n(7) 1Mb 10ms DropTail
$ns duplex-link $n(7) $n(8) 1Mb 10ms DropTail

set mproto DM
set mrthandle [$ns mrtproto $mproto {}]

#Cria agentes UdpData e os anexa aos nodos
for {set i 1} {$i < 13} {incr i} {
   if {$i != 5} {
        set udp($i) [new Agent/UDP/UdpData]
        $ns attach-agent $n($i) $udp($i)
        $udp($i) set fid_ $i
  }
}

for {set i 1} {$i < 13} {incr i} {
   if {$i != 5} {
        set group($i) [Node allocaddr]   ;# Cria grupo
        $udp($i) set dst_addr_ $group($i);# Atribui o grupo ao endereço destino do agente
        set app($i) [new AsFloodAppUdp]  ;# Cria aplicação
        $app($i) id [$n($i) id]          ;# Atribui ao id da aplicação o id do nodo
        $app($i) log $log
        $app($i) diam 6
        $app($i) attach_udp $udp($i)     ;# Anexa agente udp a aplicação
  }
}
#Insere agentes udp nos determinados grupos
$n(2) join-group $udp(2) $group(1)
$n(4) join-group $udp(4) $group(1)
$n(9) join-group $udp(9) $group(1)
$n(1) join-group $udp(1) $group(2)
$n(3) join-group $udp(3) $group(2)
$n(2) join-group $udp(2) $group(3)
$n(4) join-group $udp(4) $group(3)
$n(1) join-group $udp(1) $group(4)
$n(3) join-group $udp(3) $group(4)
$n(9) join-group $udp(9) $group(4)
$n(7) join-group $udp(7) $group(6)
$n(9) join-group $udp(9) $group(6)
$n(6) join-group $udp(6) $group(7)
$n(9) join-group $udp(9) $group(7)
$n(8) join-group $udp(8) $group(7)
$n(7) join-group $udp(7) $group(8)
$n(1) join-group $udp(1) $group(9)
$n(4) join-group $udp(4) $group(9)
$n(6) join-group $udp(6) $group(9)
$n(7) join-group $udp(7) $group(9)
$n(10) join-group $udp(10) $group(9)
$n(9) join-group $udp(9) $group(10)
$n(11) join-group $udp(11) $group(10)
$n(12) join-group $udp(12) $group(10)
$n(10) join-group $udp(10) $group(11)
$n(12) join-group $udp(12) $group(11)
$n(10) join-group $udp(10) $group(12)
$n(11) join-group $udp(11) $group(12)
#Atribui UIDs de cada aplicação
$app(1) uid 20
$app(2) uid 5
$app(3) uid 40
$app(4) uid 35
```

FIGURA 5.32 - Exemplo de *script* de simulação de Floodmax assíncrono com UDP (continuação)

```
$app(6) uid 80
$app(7) uid 7
$app(8) uid 70
$app(9) uid 10
$app(10) uid 22
$app(11) uid 30
$app(12) uid 44

#Determina a quantidade de vizinhos de cada aplicação
$app(1) neighbor 3
$app(2) neighbor 2
$app(3) neighbor 2
$app(4) neighbor 3
$app(6) neighbor 2
$app(7) neighbor 3
$app(8) neighbor 1
$app(9) neighbor 5
$app(10) neighbor 3
$app(11) neighbor 2
$app(12) neighbor 2

#Inicia a eleição
$ns at 0.1 "$app(1) send  40"
$ns at 0.5 "$app(2) send  40"
$ns at 1.7 "$app(3) send  40"
$ns at 1.2 "$app(4) send  40"
$ns at 1.1 "$app(6) send  40"
$ns at 2.0 "$app(7) send  40"
$ns at 1.6 "$app(8) send  40"
$ns at 1.0 "$app(9) send  40"
$ns at 0.8 "$app(10) send  40"
$ns at 0.3 "$app(11) send  40"
$ns at 0.3 "$app(12) send  40"

$ns at 3.5 "finish"
#Executa a simulação
$ns run
```

FIGURA 5.32 - Exemplo de *script* de simulação de Floodmax assíncrono com UDP (continuação)

```
0.1 src 1 UID 20  s
0.3 src 11 UID 30  s
0.3 src 12 UID 44  s
0.5 src 2 UID 20  s
0.8 src 10 UID 44  s
0.81064 src 11 UID 44  s
0.81064 src 12 UID 44  s
1 src 9 UID 44  s
1.01064 src 10 UID 44  s
1.02128 src 11 UID 44  s
1.02128 src 12 UID 44  s
1.1 src 6 UID 80  s
1.2 src 4 UID 44  s
1.21064 src 3 UID 44  s
1.22128 src 1 UID 44  s
1.23192 src 2 UID 44  s
1.6 src 8 UID 70  s
1.61064 src 7 UID 80  s
1.62096 src 8 UID 80  s
1.7 src 3 UID 44  s
1.71064 src 2 UID 44  s
1.71064 src 4 UID 44  s
1.72128 src 3 UID 44  s
2 src 7 UID 80  s
2.01064 src 6 UID 80  s
2.01064 src 8 UID 80  s
2.02128 src 9 UID 80  s
2.03192 src 10 UID 80  s
2.03192 src 9 UID 80  s
2.04256 src 4 UID 44  s
2.04256 src 1 UID 80  s
2.04256 src 11 UID 80  s
2.04256 src 12 UID 80  s
```

FIGURA 5.33 - Exemplo de arquivo de *log* gerado por uma aplicação `AsFloodAppUdp` (continua)

```
2.04256 src 6 UID 80  s
2.04256 src 7 UID 80  s
2.04256 src 10 UID 80  s
2.0532 src 3 UID 44  s
2.0532 src 2 UID 80  s
2.0532 src 8 UID 80  s
2.0532 src 11 UID 80  s
2.0532 src 12 UID 80  s
2.0532 src 6 UID 80  s
2.0532 src 7 UID 80  s
2.06384 src 1 UID 80  s
2.06384 src 4 UID 80  s
2.06384 src 8 UID 80  s
2.06448 src 9 UID 80  s
2.07448 src 2 UID 80  s
2.07448 src 3 UID 80  s
2.0764 src 10 UID 80  s
2.0864 src 9 UID 80  s
2.08768 src 11 UID 80  s
2.08768 src 12 UID 80  s
2.08768 src 6 UID 80  s
2.08768 src 7 UID 80  s
2.08832 src 1 UID 80  s
2.08832 src 4 UID 80  s
2.09704 src 10 UID 80  s
2.09832 src 8 UID 80  s
2.09896 src 2 UID 80  s
2.09896 src 3 UID 80  s
2.10768 src 11 Not Leader - UID 30
2.10768 src 12 Not Leader - UID 44
2.10768 src 6 UID 80  s
2.10768 src 7 UID 80  s
2.1096 src 4 UID 80  s
2.1096 src 1 UID 80  s
2.11024 src 9 UID 80  s
2.11832 src 8 Not Leader - UID 70
2.12024 src 3 Not Leader - UID 40
2.12024 src 2 Not Leader - UID 5
2.12088 src 10 Not Leader - UID 22
2.13216 src 6 Leader - UID 80
2.13216 src 7 Not Leader - UID 7
2.13216 src 4 Not Leader - UID 35
2.13216 src 1 Not Leader - UID 20
2.13216 src 9 Not Leader - UID 10
```

FIGURA 5.33: Exemplo de arquivo de *log* gerado por uma aplicação `AsFloodAppUdp` (continuação)

Tanto na implementação com TCP quanto na implementação com UDP deste algoritmo, foram desenvolvidos outros exemplos com diferentes topologias e combinações de identificadores de processos; em todos eles, foram obtidos os resultados esperados: um processo declarou-se líder e os demais determinaram que não estavam na condição de líderes. O número máximo de nodos usado nos testes foi de vinte. Alguns desses exemplos são apresentados no Anexo.

5.3 Protocolo Primário-*Backup*

Nessa seção, será apresentada uma implementação da técnica de replicação Primário-*Backup*, tomando-se como base as definições de Guerraoui e Schiper [GUE97] e de Budhiraja *et al.* [BUD93]. Primeiramente, será feita uma descrição desse protocolo; a seguir, será apresentada sua implementação e, após, alguns exemplos de experimentos feitos com esse algoritmo.

5.3.1 Definição

O protocolo Primário-*Backup* consiste de um conjunto de servidores replicados, no qual, um servidor é o primário e os restantes são *backups*. Este primário recebe invocações do cliente, executa as operações necessárias e envia a resposta ao cliente. Já os *backups* são réplicas passivas e interagem somente com o primário.

Nesse exemplo de Primário-*Backup*, assume-se que toda a comunicação é ponto-a-ponto e que somente o primário pode apresentar defeito[1]; nesse caso, um dos *backups* deve ser escolhido como novo primário. Assim, existe um servidor primário p_1 e n servidores *backups* p_2 a p_n que são conectados por canais de comunicação. Quando o primário p_1 recebe uma requisição do cliente c (mensagem 1 na Figura 5.34), ele executa os seguintes passos:

- processa as operações e atualiza seu estado;
- envia para os *backups* uma mensagem de atualização de estado (mensagens 2 na Figura 5.34);
- espera a confirmação dos *backups* ao recebimento da atualização (mensagens 3 na Figura 5.34);
- envia a resposta ao cliente (mensagem 4 na Figura 5.34).

Os *backups* atualizam seu estado quando recebem a mensagem de atualização do primário. Além disso, no exemplo desse trabalho, foi escolhido utilizar mensagens sem conteúdo (mensagens *I am alive*) para detecção de defeitos. Essas mensagens são enviadas pelo primário para os *backups* a cada t segundos. Se algum dos *backups* não receber esta mensagem por $t + d$ segundos, sendo que d é o atraso máximo possível na transmissão da mensagem, então este *backup* convoca uma eleição (mensagens 5 na Figura 5.34) para escolha do novo primário. Uma vez que o novo primário é eleito, este informa ao cliente (mensagem 6 na Figura 5.34) sua nova condição e começa a processar as próximas requisições. O servidor que apresentou defeito (antigo primário) pode voltar ao cenário como um *backup*.

No exemplo apresentado nesse trabalho, quando o novo primário recebe requisições do cliente, ele testa o identificador das mensagens. Se este identificador for maior que o identificador armazenado pelo primário, o que significa que é uma nova requisição do cliente, o primário executa os passos explicados acima e ilustrados na Figura 5.34. Se o identificador da mensagem recebida for igual ao identificador armazenado pelo primário, significa que o cliente não recebeu a resposta da requisição anterior. Nesse caso, o primário envia mensagens de atualização para os *backups*, espera as confirmações pelo recebimento dessas mensagens e envia a resposta ao cliente.

[1] Defeito no primário é o caso de interesse. Nessa implementação de Primário-*Backup*, não foi tratada a ocorrência de defeito em um *backup*. Se for necessário simular este caso, é preciso fazer algumas alterações na implementação.

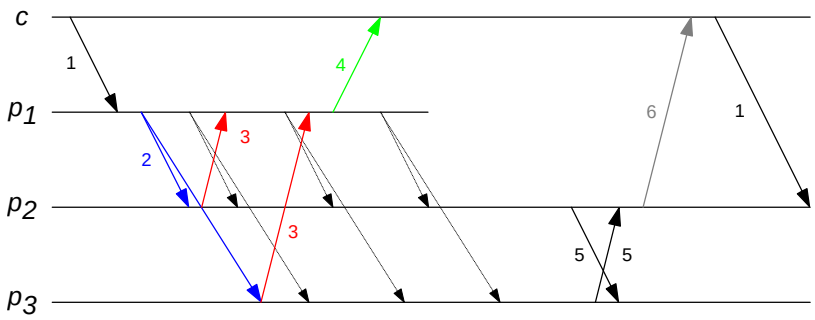


FIGURA 5.34 - Troca de mensagens do Primário-*Backup*

5.3.2 Implementação

Como nas outras implementações explicadas nesse capítulo, foram desenvolvidas classes derivadas de `AppData` (Figura 5.35), que têm a função de empacotar os dados que serão enviados entre as aplicações. Nessas classes, foram criadas duas formas de unidade de dados da aplicação (ADU): `PB_GENERAL` (Figura 5.35, linhas 31 e 32) e `PB_NEWPRIMARY` (Figura 5.35, linhas 78 e 79). Na primeira, são incluídos o identificador nodo da aplicação – `id`, o identificador da mensagem – `invid`, o valor do dado que está sendo transmitido – `value`, a aplicação que enviou a mensagem – `source` e o tipo da mensagem – `msgtype` (um dos tipos apresentados na enumeração da Figura 5.35 linhas 1 a 9). A segunda contém o identificador do nodo – `id`, o identificador da mensagem – `invid` e a indicação de quem é o novo primário – `primary`. Mensagens com essa ADU são enviadas ao cliente quando um *backup* assume como novo primário.

```
1:  enum PbMsgType {  // tipos de mensagens
2:     PB_REQUEST,
3:     PB_RESPONSE,
4:     PB_UPDATE,
5:     PB_ACK,
6:     PB_IAMALIVE,
7:     PB_RECOVERY,
8:     PB_ELECTION
9:  };

10:  const int PBDATA_COST           = 8;

11:  class PbData : public AppData {
12:  private:
13:     int id_;       // ID of the sender
14:  public:
15:     PbData() : AppData(PB_DATA) {}
16:     PbData(AppDataType t, int d) : AppData(t) { id_ = d; }
17:     PbData(PbData& d) : AppData(d) { id_ = d.id_; }
18:     inline int& id() { return id_; }
19:     virtual int size() const { return sizeof(PbData); }
20:     virtual int cost() const { return PBDATA_COST; }
21:     virtual AppData* copy() { return (new PbData(*this)); }
22:  };

23:  class PbGeneralData : public PbData {
24:  private:
25:     int value_;
26:     int msgtype_;
27:     int invID_;
28:     int lensource_;
29:     char *source_;
```

FIGURA 5.35 - Classes auxiliares derivadas de AppData (continua)

```
30:  public:
31:    PbGeneralData(int id, int invID, int value, const char *source, int msgtype) :
32:            PbData(PB_GENERAL, id) {
33:            value_ = value;
34:            msgtype_ = msgtype;
35:            invID_= invID;
36:            if ((source == NULL) || (*source == 0))
37:                    lensource_ = 0;
38:            else
39:                    lensource_ = strlen(source) + 1;
40:            if (lensource_ > 0) {
41:                    source_ = new char[lensource_];
42:                    strcpy(source_, source);
43:            } else
44:                    source_ = NULL;
45:    }

46:    PbGeneralData(PbGeneralData& d) : PbData(d) {
47:            value_ = d.value_;
48:            msgtype_ = d.msgtype_;
49:            invID_= d.invID_;
50:            if (lensource_ > 0) {
51:                    source_ = new char[lensource_];
52:                    strcpy(source_, d.source_);
53:            } else
54:                    source_ = NULL;
55:    }
56:    virtual ~PbGeneralData() {
57:            if (source_ != NULL)
58:                    delete [] source_;
59:    }
60:
61:    virtual int size() const {
62:            return (sizeof(PbData)+3*sizeof(int)+lensource_);
63:    }
64:    int value() { return value_; }
65:    int invID() {  return invID_; }
66:    int msgtype() { return msgtype_; }
67:    char* source() { return source_; }
68:    virtual AppData* copy() {
69:            return (new PbGeneralData(*this));
70:    }
71:  };

72:  class PbNewPrimaryData : public PbData {
73:  private:
74:    ServerAppR *primary_;
75:    int invID_;
76:
77:  public:
78:    PbNewPrimaryData(int id, int invID, ServerAppR *primary) :
79:            PbData(PB_NEWPRIMARY, id) {
80:            invID_= invID;
81:            primary_ = primary;
82:    }
83:    PbNewPrimaryData(PbNewPrimaryData& d) : PbData(d) {
84:            primary_ = primary_;
85:            invID_= d.invID_;
86:    }
87:    virtual ~PbNewPrimaryData() {
88:    }
89:
90:    virtual int size() const {
91:            return (sizeof(PbData)+sizeof(int));
92:    }
93:    int invID() {  return invID_; }
94:    ServerAppR* primary() {
95:            return primary_;
96:    }
97:    virtual AppData* copy() {
98:            return (new PbNewPrimaryData(*this));
99:    }
100: };
```

FIGURA 5.35 - Classes auxiliares derivadas de AppData (continuação)

Para implementação do protocolo, foram desenvolvidas mais duas classes: uma que desempenha a função do cliente (`ClientApp`) e outra do servidor (`ServerAppR`). Como nos outros algoritmos apresentados nesse capítulo, essas classes contêm os métodos `command`, `log` e `process_data`, entre outros, que serão apresentados e explicados a seguir, começando-se pela classe cliente.

O método `command` de `ClientApp` (Figura 5.36) tem como comandos principais: `primary-is`, `start-request` e `connect`. O comando `primary-is` (Figura 5.36, linha 28) tem a seguinte sintaxe em OTcl: `<clientapp> primary-is <serverapp>`, na qual `<clientapp>` é a aplicação cliente e `<serverapp>` é o servidor primário. Este comando atribui à variável `prim_` o servidor que é estabelecido como primário (Figura 5.36, linha 32). O comando `start-request` (Figura 5.36, linha 35) tem sintaxe em OTcl `<clientapp> start-request <initial_value>`, onde `<clientapp>` é a aplicação cliente e `<initial_value>` é qualquer valor inteiro a ser calculado pelo servidor. Nesse comando, primeiro é atribuído à variável `value_` o valor inicial, e após esse valor é empacotado e enviado para o servidor primário, juntamente com: o identificador do nodo (`id_`), a identificação da mensagem (`invid_` que é igual a zero), a referência de origem da mensagem (`this->name()`) e o tipo da mensagem que, nesse caso, é `PB_REQUEST` (mensagem de requisição). O comando `connect` (Figura 5.36, linha 51) é feito da mesma forma que na implementação do algoritmo LCR, explicado na seção 5.2.1. Nesse caso, o cliente é conectado a todos os servidores.

```
1:  int ClientApp::command(int argc, const char*const* argv)
2:  {
3:      Tcl& tcl = Tcl::instance();
4:
5:      if (argc == 3) {
6:              if (strcmp(argv[1], "id") == 0) {
7:                      /*
8:                       * <clientapp> id <node_id>
9:                       */
10:                     id_ = atoi(argv[2]);
11:                     tcl.resultf("%d", id_);
12:                     return TCL_OK;
13:             }
14:             else if (strcmp(argv[1], "log") == 0) {
15:                     /*
16:                      * <clientapp> log <log_file>
17:                      */
18:                     int mode;
19:                     log_ = Tcl_GetChannel(tcl.interp(),
20:                                     (char*)argv[2], &mode);
21:                     if (log_ == 0) {
22:                             tcl.resultf("%d: invalid log file handle %s\n",
23:                                 id_, argv[2]);
24:                             return TCL_ERROR;
25:                     }
26:                     return TCL_OK;
27:             }
28:             else if (strcmp(argv[1], "primary-is") == 0) {
29:                     /*
30:                      * <clientapp> primary-is <serverapp>
31:                      */
32:                     prim_ = (ServerAppR *)TclObject::lookup(argv[2]);
33:                     return TCL_OK;
34:             }
35:             else if (strcmp(argv[1], "start-request") == 0) {
36:                     /*
37:                      * <clientapp> start-request <initial_value>
38:                      */
39:                     initvalue_ = atoi (argv[2]);
40:                     value_ = initvalue_;
```

FIGURA 5.36 - Método `command` da classe `ClientApp` (continua)

```
41:                     int bytes = 10;
42:                     invid_ = 0;
43:                     PbGeneralData *d = new PbGeneralData(id_, invid_,
44:                                                 value_,this->name(),PB_REQUEST);
45:                     TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);
46:                     log("------ mID %d REQUEST \n", invid_);
47:                     cnc->send(bytes, d);
48:                     return TCL_OK;
49:             }
50:     }else if (argc == 4) {
51:             if (strcmp(argv[1], "connect") == 0) {
52:                     /*
53:                      * <clientapp> connect <server> <tcpap>
54:                      * Associa um agente TcpApp <tcpap> com o servidor
55:                      * <server>. O agente TcpApp é encarregado de
56:                      * enviar pacotes para o servidor <server>.
57:                      */
58:                     ServerAppR *server = (ServerAppR *)TclObject::lookup(argv[2]);
59:                     TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
60:                     if (add_cnc(server, cnc)) {
61:                             tcl.resultf("%s: failed to connect to %s",
62:                                             name_, argv[2]);
63:                             return TCL_ERROR;
64:                     }
65:                     /*
66:                      *Estabelece o alvo de entrega dos dados,
67:                      *isto é, a própria aplicação */
68:                     cnc->target() = (Process*)this;
69:                     return TCL_OK;
70:             }
71:     }
72:     return TclObject::command(argc, argv);
73:  }
```

FIGURA 5.36 - Método `command` da classe `ClientApp` (continuação)

O método `log` da classe `ClientApp` (Anexo) é implementado da mesma forma que nos outros algoritmos explicados nesse capítulo. O método `process_data`, que recebe os dados enviados pelo primário, pode receber duas formas de ADU: `PB_GENERAL` (Figura 5.37, linha 6) e `PB_NEWPRIMARY` (Figura 5.37, linha 25). Quando recebe `PB_GENERAL`, o cliente verifica o tipo da mensagem recebida; se for `PB_RESPONSE` (Figura 5.37, linha 8), que é a resposta do primário à sua requisição, o cliente confere se o identificador da mensagem recebida (`tmp->invID()`) é igual ao identificador da última mensagem de requisição enviada (`invid_` - Figura 5.37, linha 9). Se for, ele incrementa seu identificador de mensagem (`invid_`), prepara outra requisição e a envia ao primário (Figura 5.37, linhas 10 a 21). A recepção de uma mensagem do tipo `PB_NEWPRIMARY` significa que um novo primário foi eleito e enviou esta informação ao cliente. Ao receber essa mensagem, o cliente atribui à variável `prim_` o novo servidor primário. Após, o cliente verifica se o novo primário recebeu sua última requisição, conferindo o identificador de mensagem (`tmp->invID()`) enviada pelo novo primário (Figura 5.37, linha 28): se este valor é igual ao identificador de mensagem do cliente (`invid_`) então ele recebeu a última requisição, mas a resposta não foi enviada ao cliente; se o valor for menor que o `invid_`, então o novo primário não recebeu a última requisição. Nos dois casos, o cliente envia a requisição novamente.

```
1:  void ClientApp::process_data(int, AppData* data)
2:  {
3:      if (data == NULL)
4:              return;
5:      switch (data->type()) {
6:         case PB_GENERAL: {
7:              PbGeneralData *tmp = (PbGeneralData*)data;
8:                 if (tmp->msgtype() == PB_RESPONSE) { // se recebeu a resposta
```

FIGURA 5.37 - Método `process_data` da classe `ClientApp` (continua)

```
9:                      if (tmp->invID() == invid_) {   // se o invid_ enviado é igual ao
10:                             value_ = tmp->value();  // recebido
11:                             if (value_ > 1000000000)
12:                                     value_ = initvalue_;
13:                             int bytes = 10;
14:                             invid_ ++;   // incrementa identificador de mensagem
15:                             value_ = value_ + 1;
16:                             PbGeneralData *d = new PbGeneralData(id_, invid_, value_,
17:                                                             this->name(), PB_REQUEST);
18:                             TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);
19:                             log("------ mID %d REQUEST \n", invid_);
20                              cnc->send(bytes, d); // envia nova requisição ao primário
21:                     }
22:             }
23:             break;
24:         }
25:         case PB_NEWPRIMARY: {
26:             PbNewPrimaryData *tmp = (PbNewPrimaryData*)data;
27:             prim_ = tmp->primary();
28:             if (invid_ >= tmp->invID()) { //se id da última requisição é maior ou
29:                     int bytes = 10;              // igual a id recebido
30:                     PbGeneralData *d = new PbGeneralData(id_, invid_, value_,
31:                                                             this->name(), PB_REQUEST);
32:                     TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);//procura TcpApp na tab.
33:                     log("------ mID %d REQUEST \n", invid_);
34:                     cnc->send(bytes, d);  // envia a requisição novamente
35:             }
36:             break;
37:     }
38:     default:
39:             fprintf(stderr, "Bad primary backup data type %d\n", data->type());
40:             abort();
41:             break;
42:     }
43:  }
```

FIGURA 5.37 - Método `process_data` da classe `ClientApp` (continuação)

No construtor da classe `ServerAppR` (Figura 5.38), que implementa os servidores (primário e *backups*), são iniciadas variáveis (linhas 1 a 3) e feitas ligações entre as variáveis OTcl e C++ (linhas 5 a 7), permitindo que ambas acessem os mesmos dados.

```
1:  ServerAppR::ServerAppR() : log_(0), value_(-1), rounds_(0), Iam_primary_(false),
2:          invid_or_r_(-1), invidprev_(-1), n_of_backups_(0), ack_received_(0),
3:          detectfail_(0), fname_(NULL), itimer_(this), rtimer_(this)
4: {
5:      bind("diam_", &diam_);  // valor default=1 – definido em ns-default.tcl
6:      bind("prim_timeout_", &prim_timeout_);  // timeout para enviar I am alive
7:      bind("timeout_", &timeout_); // timeout para detectar defeito do primário
8:      tpa_ = new Tcl_HashTable;
9:      Tcl_InitHashTable(tpa_, TCL_ONE_WORD_KEYS);
10:  }
```

FIGURA 5.38 - Construtor da classe `ServerAppR`

No método `command` da classe `ServerAppR`, os principais comandos são: `primary`, `recovery`, `reset`, `connect`, `connect-to-client`, `start-Iamalive`. Esses comandos são apresentados na Figura 5.39 e explicados a seguir.

```
1:  int ServerAppR::command(int argc, const char*const* argv)
2:  {
3:      Tcl& tcl = Tcl::instance();
4:      if (argc == 2) {
5:              if (strcmp(argv[1], "primary") == 0) {
6:                      Iam_primary_ = true;
7:                      tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1 circle", id_, id_);
8:                      tcl.evalf("$n(%d) label Primary", id_);
```

FIGURA 5.39- Método `command` da classe `ServerAppR` (continua)

```
9:                      return TCL_OK;
10:             }
11:             else if (strcmp(argv[1], "start-Iamalive") == 0) {
12:                     set_timer();
13:                     return TCL_OK;
14:             }
15:             else if (strcmp(argv[1], "recovery") == 0) {
16:                     bytes_ = 10;
17:                     msg_t_= PB_RECOVERY;
18:                     sendto_servers();
19:                     return TCL_OK;
20:             }
21:             else if (strcmp(argv[1], "reset") == 0) {
22:                     reset();
23:                     return TCL_OK;
24:             }
25:     }
26:     else if (argc == 3) {
27:             if (strcmp(argv[1], "id") == 0) {
28:                     /*
29:                      * <serverapp> id <node_id>
30:                      */
31:                     id_ = atoi(argv[2]);
32:                     minid_ = id_;
33:                     memset(ro_, 0, 30); // limited in 30 rounds
34:                     tcl.resultf("%d", id_);
35:                     return TCL_OK;
36:             }
37:             else if (strcmp(argv[1], "log") == 0) {
38:                     /*
39:                      * <serverapp> log <log_file>
40:                      */
41:                     int mode;
42:                     log_ = Tcl_GetChannel(tcl.interp(),
43:                                     (char*)argv[2], &mode);
44:                     if (log_ == 0) {
45:                             tcl.resultf("%d: invalid log file handle %s\n",
46:                                 id_, argv[2]);
47:                             return TCL_ERROR;
48:                     }
49:                     return TCL_OK;
50:             }
51:             else if (strcmp(argv[1], "backups-number") == 0) { // para amnésia total
52:                     /*
53:                      * <serverapp> backups-number <n_of_servers>
54:                      */
55:                     n_of_backups_ = atoi(argv[2])-1;
56:                     return TCL_OK;
57:             }
58:             else if (strcmp(argv[1], "file") == 0) {
59:                     int lenfile = strlen(argv[2]) + 1;
60:                     fname_ = new char[lenfile];
61:                     strcpy(fname_, argv[2]);
62:                     if((file_=fopen(fname_, "w")) == NULL) {
63:                         fprintf(stderr, "Could not open file\n");
64:                          abort();
65:                     }
66:                     return TCL_OK;
67:             }
68:     }else if (argc == 4) {
69:             if (strcmp(argv[1], "connect") == 0) {
70:                     /*
71:                      * <serverapp> connect <server> <tcpap>
72:                      *
73:                      * Associate a TcpApp agent with the given server.
74:                      * <tcpap> is the agent used to send packets out.
75:                      * We assume two-way TCP connection, therefore we
76:                      * only need one agent.
77:                      */
78:                     n_of_backups_++;
79:                     ServerAppR *server = (ServerAppR *)TclObject::lookup(argv[2]);
80:                     TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
81:                     if (add_cnc(server, cnc)) {
82:                             tcl.resultf("%s: failed to connect to %s",
```

FIGURA 5.39 - Método `command` da classe `ServerAppR` (continuação)

```
83:                                     name_, argv[2]);
84:                             return TCL_ERROR;
85:                     }
86:                     cnc->target() = (Process*)this;
87:                     return TCL_OK;
88:             }
89:             else if (strcmp(argv[1], "connect-to-client") == 0) {
90:                     /*
91:                      * <serverapp> connect-to-client <client> <tcpap>
92:                      *
93:                      * Associate a TcpApp agent with the given client <client>.
94:                      * <tcpap> is the agent used to send packets out.
95:                      * We assume two-way TCP connection, therefore we
96:                      * only need one agent.
97:                      */
98:                     cnc_client_ = (TcpApp *)TclObject::lookup(argv[3]);
99:                     // Set data delivery target - this aplication
100:                    cnc_client_->target() = (Process*)this;
101:                    return TCL_OK;
102:            }
103:    }
104:    return TclObject::command(argc, argv);
105: }
```

FIGURA 5.39 - Método `command` da classe `ServerAppR` (continuação)

O comando `primary` (Figura 5.39, linha 5), que tem sintaxe em OTcl `<serverapp> primary`, atribui à variável lógica `Iam_primary` (iniciada com `false`) o valor `true`: isto significa que o servidor identificado em `<serverapp>` é o primário. Além disso, são feitas duas chamadas a `tcl.evalf` (explicado na seção 3.3.2), que invocam comandos OTcl através da instância `tcl`. Na primeira chamada (linha 7), é invocado o comando `add-mark` que adiciona uma marca, em forma de círculo, a um nodo na animação do Nam. A segunda chamada (linha 8) invoca `label` que adiciona um rótulo a este mesmo nodo. Estes dois comandos não são essenciais à implementação do algoritmo, entretanto foram utilizados para facilitar a visualização de seu funcionamento. Exemplos de telas do Nam, com utilização desses comandos, são apresentados na seção 5.3.3.

O comando `recovery` (`<serverapp> recovery` – Figura 5.39, linha 15) é chamado quando o primário, que apresentou defeito, é recuperado como *backup*. Nesse comando, o antigo primário recuperado envia mensagens do tipo `PB_RECOVERY` para os outros servidores (linhas 17 e 18, Figura 5.39).

O comando `reset` (Figura 5.39, linha 21) é chamado para simular defeito de colapso no primário. Esse comando invoca o método `reset()` que reinicia as variáveis da aplicação. Pode-se reiniciar todas as variáveis para simular colapso com amnésia total ou somente algumas para amnésia parcial. Detalhes do método `reset` e exemplos de como simular esses dois tipos de colapso são apresentados a seguir.

O comando `file` (Figura 5.39, linha 58) é usado quando se deseja armazenar os dados, que serão processados pelo primário, em um arquivo. Pode-se atribuir um arquivo para cada servidor, incluindo primário e *backups*.

O comando `connect` (Figura 5.39, linha 69), que faz as conexões entre os servidores, tem implementação semelhante a dos outros algoritmos descritos nesse capítulo. A diferença está na variável `n_of_backups` (linha 78) que é incrementada a cada conexão. Este valor é usado pelo primário para verificar se a quantidade de confirmações recebidas, depois que este enviou uma atualização para os *backups*, é igual ao número de *backups* existentes.

O comando `connect-to-client` (Figura 5.39, linha 89) faz a conexão do servidor com o cliente. Este comando deve ser feito não só pelo primário, mas também

pelos *backups*, já que, quando o primário apresentar defeito, o *backup*, que assumir como novo primário, deve estar conectado ao cliente.

O método `set_timer()` – Figura 5.40, linha 1- é chamado pelo comando `start-Iamalive` (Figura 5.39, linha 11), que só é feito pelo primário. Este método chama `resched` (Figura 5.40, linha 3) que faz um temporizador expirar no tempo atribuído em `prim_timeout_`, assim, é chamado o método `expire` (Figura 5.40, linha 20) que invoca `iam_alive()`. Dentro desse método (Figura 5.40, linhas 5 a 19) são enviadas, a todos os *backups*, mensagens do tipo `PB_IAMALIVE`. Após, o método `set_timer()` é chamado para que o temporizador comece a contar o tempo novamente. Todo esse processo é utilizado para que, de tempos em tempos, o primário envie mensagens *I am alive* para os *backups*. Mais detalhes sobre a implementação de temporizadores podem ser obtidos no manual do NS [FAL2001].

```
1:  void ServerAppR::set_timer()
2:  {
3:      itimer_.resched(prim_timeout_);
4:  }

5:  void ServerAppR::iam_alive ()
6:  {
7:      bytes_ = 2;
8:      Tcl_HashEntry *he;
9:      Tcl_HashSearch hs;
10:     for(he = Tcl_FirstHashEntry(tpa_, &hs);  he != NULL; he = Tcl_NextHashEntry(&hs))
11:     {
12:             TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
13:             PbGeneralData *d = new PbGeneralData(id_, 0, 0,
14:                                     this->name(), PB_IAMALIVE);
15:             log("------ IAMALIVE\n");
16:             cnc->send(bytes_, d);
17:     }
18:     set_timer();
19: }

20: void IamAliveTimer::expire(Event*)
21: {
22:     a_->iam_alive();
23: }
```

FIGURA 5.40 - Métodos para envio de mensagens *I am alive*

O método `process_data`, que faz o tratamento das mensagens recebidas, somente recebe ADUs do tipo `PB_GENERAL`. Nesse método, é testado se o servidor que recebeu os dados é o primário (Figura 5.41, linha 9), que pode receber os seguintes tipos de mensagens:

- `PB_REQUEST` (Figura 5.41, linha 10): mensagem de requisição enviada pelo cliente. Nesse caso, primeiro é testado se o valor do identificador da mensagem recebida (`tmp->invID()`) é maior que o identificador da mensagem anterior (`invidprev_`), o que significa que é uma nova requisição do cliente. Nesse caso, a operação é processada e gravada em um arquivo (se este foi definido no *script* pelo comando `file`). Se o identificador da mensagem recebida for igual ao identificador anterior, significa que o cliente não recebeu a resposta da mensagem anterior. Então são enviadas mensagens de atualização para os *backups*.
- `PB_ACK` (Figura 5.41, linha 34): enviada pelos *backups*, é a confirmação pelo recebimento da mensagem de atualização. Nesse caso, primeiro é testado se o identificador da mensagem recebida é igual ao identificador da mensagem de atualização enviada (Figura 5.41, linha 35); se for, então o contador de confirmação (`ack_received_`) é incrementado. Após receber as confirmações de todos os

backups, o primário envia a mensagem de resposta ao cliente (Figura 5.41, linhas 37 a 44).

Se o receptor da mensagem é um *backup* (Figura 5.41, linha 48) são verificados os seguintes tipos de mensagens:

- `PB_UPDATE` (Figura 5.41, linha 49): mensagem de atualização recebida do primário. Nesse caso, o valor recebido é armazenado em `value_` e em um arquivo (se este foi definido no *script*). Então é enviada a mensagem de confirmação `PB_ACK` para o primário (linha 63).
- `PB_IAMALIVE` (Figura 5.41, linha 65): mensagem *I am alive*, enviada pelo primário. Chama `set_rcvr_timer` (Figura 5.42, linha 1), que começa a contar o tempo (atribuído em `timeout_` no *script* de simulação) que o *backup* deve esperar após receber uma mensagem desse tipo. Quando esse tempo terminar (Figura 5.42, linha 31), se outra mensagem *I am alive* não houver sido recebida, este *backup* convoca uma eleição (Figura 5.42, linhas 20 a 29).
- `PB_ELECTION` (Figura 5.41, linha 68): significa que algum dos outros *backups* detectou defeito do primário e enviou mensagem de eleição. O algoritmo de eleição utilizado é o Floodmax assíncrono apresentado na seção 5.2.1. A diferença entre a eleição do Floodmax assíncrono e a implementada no Primário-*Backup* é que, em vez de eleger a aplicação de maior UID, é eleita a de menor UID. O *backup* que tiver o menor UID se declara novo primário (Figura 5.41, linha 88) e envia uma mensagem `PB_NEWPRIMARY` para o cliente (Figura 5.41, linha 97). O valor predefinido de `diam_` é 1 – 5.38, linha 5 (para redes totalmente conectadas, a maior distância entre dois nodos é sempre 1).

```
1: void ServerAppR::process_data(int, AppData* data)
2: {
3:     if (data == NULL)
4:             return;
5:     PbGeneralData *tmp = (PbGeneralData*)data;
6:     if (tmp->msgtype() == PB_RECOVERY) {
7:             n_of_backups_++;
8:     }
9:     if (Iam_primary_) { // se o servidor é o primário
10:         if (tmp->msgtype() == PB_REQUEST) {  // se a mensagem é uma requisição
11:             if(tmp->invID() > invidprev_) {  // testa se o invID recebido é maior
12:                 value_ = tmp->value() * 2;  //processa o dado
13:                 if(fname_ != NULL)  // se arquivo foi definido no script
14:                         fprintf(file_, "%d\n", value_); // armazena o valor
15:             }
16:             else if(tmp->invID() == invidprev_)
17:                 value_ = tmp->value() * 2;
18:             invid_or_r_ = tmp->invID();
19:              invidprev_ =  invid_or_r_;
20:             if (n_of_backups_ != 0) { // se tiver backups, não for somente primário
21:                 bytes_ = 10;
22:                 msg_t_ = PB_UPDATE;
23:                 sendto_servers(); // envia mensagem de atualização aos backups
24:             }
25:             else {  // se número de backups for igual a zero (não tem backup)
26:                bytes_ = 10;
27:                ack_received_ = 0;
28:                PbGeneralData *d = new PbGeneralData(id_, invid_or_r_,
29:                                                    value_, this->name(), PB_RESPONSE);
30:                log("------ mID %d RESPONSE %d\n", invid_or_r_, value_);
31:                cnc_client_->send(bytes_, d);  // envia resposta ao cliente
32:             }
33:         }
34:         else if (tmp->msgtype() == PB_ACK) {  // se recebeu uma confirmação
35:               if (tmp->invID() == invid_or_r_) { // com invID igual ao da msg que
36:                   ack_received_++;             //enviou incrementa contador de acks
```

FIGURA 5.41 - Método `process_data` da classe `ServerAppR` (continua)

```
37:                     if (ack_received_ == n_of_backups_) { //recebeu acks de todos
38:                         bytes_ = 10;
39:                         ack_received_ = 0;
40:                         PbGeneralData *d = new PbGeneralData(id_, invid_or_r_,
41:                                                 value_, this->name(), PB_RESPONSE);
42:                         log("------ mID %d RESPONSE %d\n", invid_or_r_, value_);
43:                         cnc_client_->send(bytes_, d);  // envia resposta ao cliente
44:                     }
45:                 }
46:         }
47:     }
48:     else { // se é backup
49:          if (tmp->msgtype() == PB_UPDATE) { //se mensagem é de atualização
50:                 value_ = tmp->value();
51:                 if(tmp->invID() > invidprev_) {  // testa se o invID recebido é maior
52:                    if (fname_ != NULL)          // se arquivo foi definido no script
53:                       fprintf(file_, "%d\n", value_); //armazena o valor
54:                 }
55:                 invid_or_r_ = tmp->invID();
56:                 invidprev_ =  invid_or_r_;
57:                 bytes_ = 10;
58:                 ServerAppR *primary = (ServerAppR *)TclObject::lookup(tmp->source());
59:                 TcpApp *cnc = (TcpApp *)lookup_cnc(primary);
60:                  PbGeneralData *d = new PbGeneralData(id_, invid_or_r_,
61:                                                       value_, this->name(), PB_ACK);
62:                 log("------ mID %d ACK\n", invid_or_r_);
63:                 cnc->send(bytes_, d); //envia confirmação para o primário
64:         }
65:         else if (tmp->msgtype() == PB_IAMALIVE) {  // se mensagem é I am alive
66:            set_rcvr_timer(timeout_); // inicia temporizador
67:         }
68:         else if (tmp->msgtype() == PB_ELECTION) { // se recebeu mensagem de eleição
69:              int uid=tmp->value();
70:              int r=tmp->invID();
71:              if (ro_[r] < (n_of_backups_ - 1)){
72:                     ro_[r]++;
73:                     if (uid < minid_) minid_=uid; // se uid recebido é menor
74:              }
75:              if (ro_[r] == (n_of_backups_ - 1)) {
76:                   if(detectfail_==1){
77:                       rounds_++;
78:                       if (rounds_ < diam_)
79:                       {
80:                          invid_or_r_ = rounds_;
81:                          value_ = minid_;
82:                          msg_t_ = PB_ELECTION;
83:                          sendto_servers();
84:                       }
85:                       if(rounds_ == diam_) {
86:                          if(minid_ == id_)  { //backup com menor UID – novo primário
87:                                 n_of_backups_--;
88:                                 Iam_primary_ = true;
89:                                 Bytes_ = 10;
90:                                 PbNewPrimaryData *d= new PbNewPrimaryData(id_,
91:                                                          invid_or_r_, this);
92:                                 log("------ NEWPRIMARY \n");
93:                                 Tcl& tcl = Tcl::instance();
94:                                 tcl.evalf("$n(%d) add-mark m(%d)
95:                                             DeepSkyBlue1 circle", id_, id_);
96:                                 tcl.evalf("$n(%d) label Primary", id_);
97:                                 cnc_client_->send(bytes_, d); //envia mensagem
98:                                 set_timer(); // começa a enviar I am alive
99:                          }
100:                         else  n_of_backups_--;
101:                    }
102:                }
103:            }
104:        }
105:     }
106: }
```

FIGURA 5.41 - Método `process_data` da classe `ServerAppR` (continuação)

```
1:  void ServerAppR::set_rcvr_timer(double timeout_rcvr)
2:  {
3:      rtimer_.resched(timeout_rcvr);
4:  }

5:  // Ocorreu timeout, backups detectam defeito do primário
6:  void ServerAppR::on_rcvr_timeout()
7:  {
8:      if(n_of_backups_ == 1)  { //somente primario–sem backups (antes existia somente
9:              n_of_backups_--;  // outro servidor), então só restou esse servidor
10:             Iam_primary_ = true;  // que se declara primário
11:             bytes_ = 10;
12:             PbNewPrimaryData *d= new PbNewPrimaryData(id_, invid_or_r_, this);
13:             log("------ NEWPRIMARY \n");  // se declara primário porque não tem
14:             Tcl& tcl = Tcl::instance();   // mais backups
15:             tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1 circle", id_, id_);
16:             tcl.evalf("$n(%d) label Primary", id_);
17:             cnc_client_->send(bytes_, d);
18:             set_timer();
19:     }
20:     else {
21:         bytes_ = 10;
22:         rounds_ = 0;
23:         detectfail_=1;
24:         memset(ro_, 0, 30);
25:         invid_or_r_ = rounds_;
26:         value_ = minid_;
27:         msg_t_ = PB_ELECTION; // convoca eleição,
28:         sendto_servers();     // enviando mensagem PB_ELECTION para os outros backups
29:     }
30: }

31: void ReceiverTimer::expire(Event*)
32: {
33:     a_->on_rcvr_timeout();
34: }
```

FIGURA 5.42 - Métodos para envio de convocação de eleição (continua)

O método `reset()` - Figura 5.43 é utilizado para simular defeito de colapso em um servidor primário. Como conseqüência, por execução desse método, se o servidor for o primário, é cancelado o temporizador que controla o envio de mensagens *I am alive* (linha 4). Se o arquivo para armazenar os dados foi definido no *script*, ele é esvaziado, fechado e, então, aberto novamente (linhas 6 a 10). Este arquivo é aberto no método `reset` porque, nesse caso, é simulado colapso com amnésia parcial. Na implementação com amnésia total (Anexo), este arquivo seria aberto com a invocação do comando `file` (Figura 5.39, linha 58) no *script* OTcl, pois, no caso de amnésia total, deve-se executar os comandos OTcl necessários para a aplicação servidora voltar ao seu estado inicial. Na Figura 5.43 – linhas 14 a 19, algumas variáveis são reiniciadas, como é simulado colapso com amnésia parcial, variáveis como `n_of_backups` e `diam_` não são reiniciadas.

Um método como esse pode ser implementado em qualquer tipo de aplicação, fazendo com que o estado da aplicação após o defeito possa ser determinado. Se esse método não for utilizado e, para simular defeito de colapso, for empregado somente o comando que rompe todos os enlaces de um nodo ou o comando que desanexa os agentes de transporte, podem ocorrer as seguintes situações:

1) o temporizador (Figura 5.43, linha 4) não ser cancelado;
2) a variável `Iam_primary_` não ser reiniciada, se o primário com defeito for recuperado como *backup* (com o restabelecimento dos links ou anexando-se os agentes de transporte);
3) arquivos não serem esvaziados.

O problema da primeira situação é que o primário continua tentado enviar mensagens *I am alive* para os *backups*, que não recebem essas mensagens por falta do meio de envio (enlaces ou agentes de transporte). Neste caso, o processamento interno da aplicação continua.

Já na segunda situação, onde a variável `Iam_primary_` permanece com valor `true`, o servidor recuperado continua supondo que é o primário e não desempenha as funções de *backup*. Um dos problemas dessas duas situações ocorre quando o servidor que assumiu como novo primário apresenta defeito. Como o antigo primário está com estado inconsistente, ele continua notificando os outros servidores, através de mensagens *I am alive*, que o primário não está com defeito. Assim, estes servidores não detectam o defeito no primário. Como resultado, nenhum outro primário é eleito e o cliente fica sem resposta para suas requisições.

Na terceira situação, o conteúdo dos arquivos fica diferente. Com o uso do método `reset`, o conteúdo do arquivo é esvaziado e, após a recuperação do primário como *backup*, só serão gravados os dados recebidos após o defeito. Sem o uso desse método, o conteúdo do arquivo é mantido e seus dados podem ficar inconsistentes, principalmente se o servidor for recuperado.

Estes problemas mostram claramente que não são equivalentes defeitos onde a aplicação perde a comunicação com as outras aplicações do sistema; e defeitos nos quais a aplicação pára seu processamento e pode ser recuperada.

```
1:  void ServerAppR::reset()
2:  {
3:      if (Iam_primary_)
4:              itimer_.cancel();
5:
6:      if (fname_!=NULL) {
7:          fflush(file_);
8:          fclose(file_);
9:          file_=fopen(fname_, "w");
10:     }
11:     Tcl& tcl = Tcl::instance();
12:     tcl.evalf("$n(%d) delete-mark m(%d)", id_, id_);
13:     tcl.evalf("$n(%d) label \" \" ", id_);

14:     Iam_primary_ = false;
15:     ack_received_ = 0;
16:     invid_or_r_ = -1;
17:     value_ = -1;
18:     rounds_=0;
19:     detectfail_=0;
20:  }
```

FIGURA 5.43 - Método `reset` da classe `ServerAppR`

Esta implementação foi desenvolvida com o modelo baseado em TCP. O uso de TCP como agente de transporte faz com que sejam retransmitidas as mensagens enviadas para o servidor que apresentou defeito e foram perdidas, já que seus enlaces foram rompidos ou agentes foram desanexados. Como tanto o cliente quanto os servidores têm o controle da identificação das mensagens, as que não correspondem aos valores esperados não são processadas. Portanto, o TCP pode ser utilizado quando se deseja ter confiabilidade nos canais de comunicação.

5.3.3 Exemplo de simulação

Nessa seção, será apresentado um exemplo de simulação do protocolo Primário-*Backup*, exibindo-se: o *script* OTcl, o arquivo de *log* gerado por essa simulação e telas do Nam com momentos da execução. Este exemplo simula defeitos de colapso com amnésia parcial. Outros exemplos de simulações desse protocolo são apresentados no Anexo, incluindo exemplo de colapso com amnésia total.

O *script* da Figura 5.44 é um exemplo de simulação com quatro nodos, sendo um cliente e três servidores. O servidor com menor identificador é o primário; nesse caso, a aplicação contida no nodo 1 inicia como servidor primário. A partir do tempo `0.0s` (linha 88), o primário inicia o envio de mensagens *I am alive*, que serão enviadas a cada `0.2s` (linha 82). O atraso na transmissão da mensagem (representado por *d* - seção 5.3.1) é `0.04s` (linha 86). Em `1.0s`, o cliente começa a fazer requisições ao primário (linha 89). O tempo despendido entre a requisição e entrega da mensagem ao cliente é de aproximadamente `0.03s`. O primário 1 apresenta defeito no tempo `1.5s` (linhas 90 a 95) e é recuperado como *backup* em `1.9s` (linhas 96 a 102). No primário 2, ocorre defeito em `2.6s` e recuperação em `2.9s`.

```
1:  #Arquivos com criação da topologia(topo1.tcl) e de agentes e conexões(tcp_sd.tcl)
2:  source /home/renata/extcpap/topo1.tcl
3:  source /home/renata/extcpap/tcp_sd.tcl

4:  global n_node
5:  set n_node 4     ;# Quantidade de nodos
6:  set topol fully  ;# Tipo da topologia

7:  #Criação do objeto Simulator
8:  set ns [new Simulator]

9:  #Abertura de arquivos
10: set f [open out.tr w]    ;# Arquivo de rastro
11: $ns trace-all $f
12: set nf [open out1.nam w] ;# Arquivo do Nam
13: $ns namtrace-all $nf
14: set log [open pb1.log w] ;# Arquivo de log

15: $ns color 0 blue
16: $ns color 1 red
17: $ns color 2 gold1
18: $ns color 3 SpringGreen3
19: $ns color 4 OrangeRed1
20: $ns color 5 DarkOrchid3
21: $ns color 6 HotPink
22: $ns color 5 yellow
23: $ns color 7 cyan

24: #Definição o procedimento 'finish'
25: proc finish {} {
26:        global ns nf f log
27:        $ns flush-trace
28:        close $nf
29:        close $f
30:         close $log
31:        exec nam out1.nam &
32:        exit 0
33: }

34: #Criação dos nodos e da topologia, através de chamada a create-topology (topo1.tcl)
35: create-topology $n_node $topol 1Mb 10ms DropTail

36: #Criação dos agentes FullTcp e TcpApp chamando-se create-tcpds (tcp_sd.tcl)
37: create-tcpds $n_node

38: #Criação das aplicações
39: set app(0) [new ClientApp]
40: set app(1) [new ServerAppR]
41: set app(2) [new ServerAppR]
```

FIGURA 5.44 - Exemplo de *script* de simulação do Primário-*Backup* (continua)

```
42: set app(3) [new ServerAppR]

43: $n(0) color "blue"    ;# Atribui cor azul ao nodo que contém o cliente (Nam)
44: $n(0) shape "hexagon" ;# Atribui forma hexagonal ao nodo cliente
45: $n(0) label "Client"  ;# Coloca o rótulo Client em n(0)

46: #Atribui aos ids das aplicações os ids dos nodos
47: $app(0) id [$n(0) id]
48: $app(1) id [$n(1) id]
49: $app(2) id [$n(2) id]
50: $app(3) id [$n(3) id]

51: #Atribui a log o arquivo de log
52: $app(0) log $log
53: $app(1) log $log
54: $app(2) log $log
55: $app(3) log $log
56: #Definição dos arquivos que armazenam dados da aplicação
57: $app(1) file pb1_test1
58: $app(2) file pb1_test2
59: $app(3) file pb1_test3
60: #Conexão de todos os agentes FullTcp e TcpApp, chamando-se connect-all (tcp_sd.tcl)
61: connect-all $n_node

62: $app(0) connect $app(1) $tcpap(0:1) ;#Conexão de cliente com servidor – app(1)
63: $app(0) connect $app(2) $tcpap(0:2) ;#Conexão de cliente com servidor - app(2)
64: $app(0) connect $app(3) $tcpap(0:3) ;#Conexão de cliente com servidor - app(3)

65: $app(1) connect-to-client $app(0) $tcpap(1:1) ;#conexão de servidor com o cliente
66: $app(1) connect $app(2) $tcpap(1:2) ;#conexão do servidor app(1) com app(2)
67: $app(1) connect $app(3) $tcpap(1:3) ;#conexão do servidor app(1) com app(3)

68: $app(2) connect-to-client $app(0) $tcpap(2:1) ;#conexão de servidor com o cliente
69: $app(2) connect $app(1) $tcpap(2:2) ;#conexão do servidor app(2) com app(1)
70: $app(2) connect $app(3) $tcpap(2:3) ;#conexão do servidor app(2) com app(3)

71: $app(3) connect-to-client $app(0) $tcpap(3:1) ;#conexão de servidor com o cliente
72: $app(3) connect $app(1) $tcpap(3:2) ;#conexão do servidor app(3) com app(1)
73: $app(3) connect $app(2) $tcpap(3:3) ;#conexão do servidor app(3) com app(2)

74: $app(0) primary-is $app(1) ;#Comando que diz ao cliente que servidor é o primário
75: $app(1) primary            ;# Indica que servidor é o primário

# Atribuição a agentes FullTcp o estado de espera
76: $tcp(0:2) listen ;# Cliente sempre recebe do servidor 2 antes de enviar
77: $tcp(0:3) listen ;# Cliente sempre recebe do servidor 3 antes de enviar

78: $tcp(1:1) listen ;# Servidor 1 sempre espera requisição do cliente antes de enviar
                      # qualquer dado a ele
79: $tcp(2:2) listen ;# Servidor 2 sempre espera receber dados do primário (1) antes de
                      # enviar qualquer dado a ele
80: $tcp(3:2) listen ;# Servidor 3 sempre espera receber dados do primário (1) antes de
                      # enviar Qualquer dado a ele

81: # Atribuição de tempo de envio de mensagens I am alive
82: $app(1) set prim_timeout_ 0.2
83: $app(2) set prim_timeout_ 0.2
84: $app(3) set prim_timeout_ 0.2

85: # Atribuição de tempo de espera por mensagens I am alive
86: $app(2) set timeout_ 0.24
87: $app(3) set timeout_ 0.24

88: $ns at 0.0 "$app(1) start-Iamalive"  ;# Inicio do envio de mensagens I am alive
89: $ns at 1.0 "$app(0) start-request 2" ;# Cliente inicia requisição

90: $ns rtmodel-at 1.5 down $n(1)  ;# Quebra de links do nodo n(1)
91: $ns at 1.5 {         ;# Defeito no primário
92:     $app(1) reset     ;# Chamada ao método reset
93:     $n(1) color red   ;# Muda a cor do nodo n(1) para vermelha (Nam)
94:     $ns trace-annotate "Primary 1 crash failure" ;# Anotações nos arquivos de rastro
95: }

96: $ns rtmodel-at 1.9 up $n(1) ;# Restauração de links do nodo n(1)
97: $ns at 1.9 {          ;# Recuperação do nodo 1 (como backup)
98:     $app(1) recovery   ;# Chama recovery
99:     $n(1) color black  ;# Muda a cor do nodo n(1) para preta (Nam)
100:    $ns trace-annotate "1 recovery as backup" ;# Anotações nos arquivos de rastro
101:    $app(1) set timeout_ 0.24  ;# Tempo de espera por mensagem I am alive
102:}
```

FIGURA 5.44 - Exemplo de *script* de simulação do Primário-*Backup* (continuação)

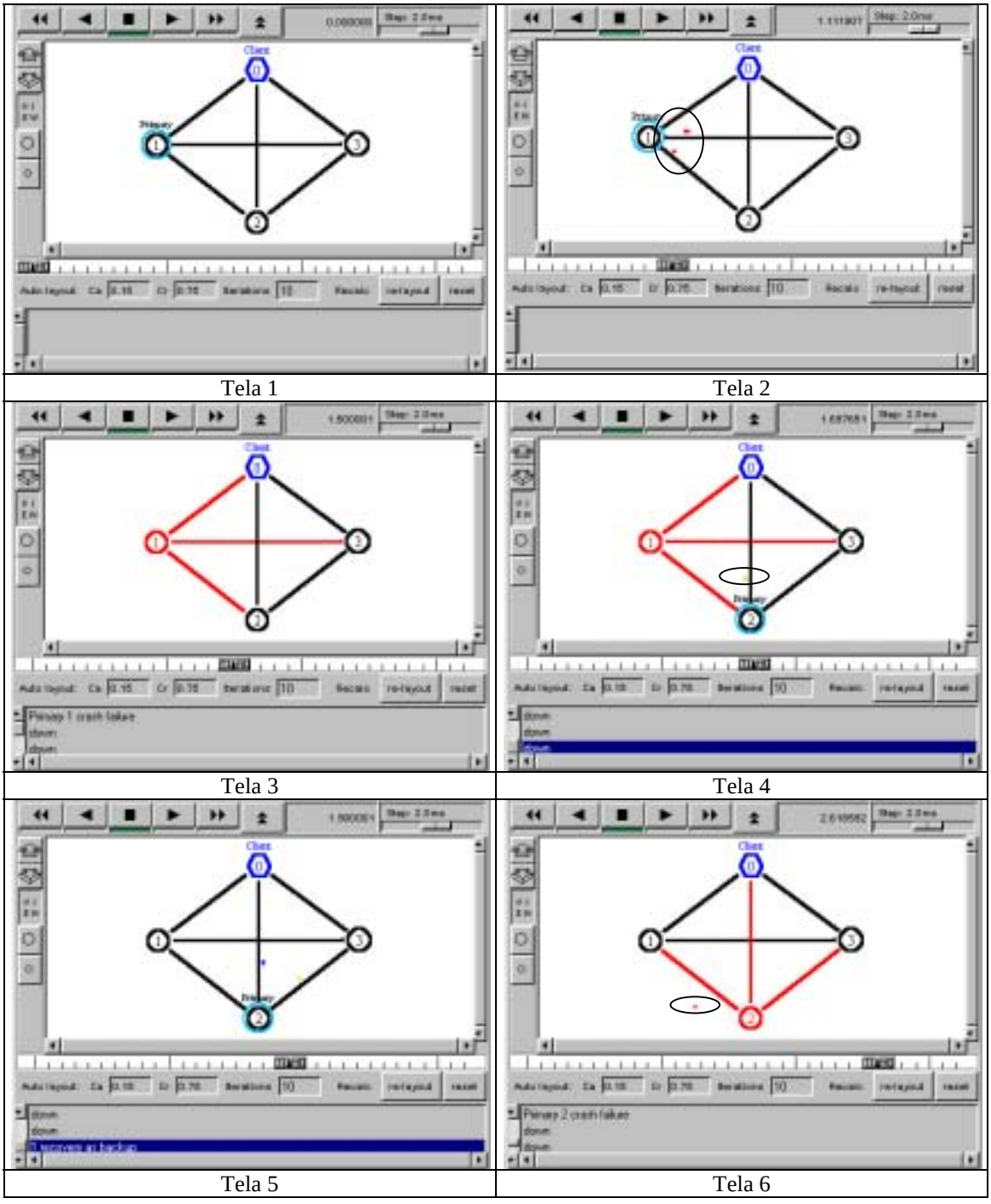
```
103:$ns rtmodel-at 2.6 down $n(2) ;# Quebra de links do nodo n(2)
104: $ns at 2.6 {       ;# Defeito no primário
105:    $app(2) reset    ;# Chamada ao método reset
106:    $n(2) color red  ;# Muda a cor do nodo n(2) para vermelha (Nam)
107:    $ns trace-annotate "Primary 2 crash failure"  ;# Anotações nos arquivos de rastro
108:}
109:$ns rtmodel-at 2.9 up $n(2) ;# Restauração de links do nodo n(2)
110:$ns at 2.9 {         ;# Recuperação do nodo 2 (como backup)
111:    $app(2) recovery  ;# Chama recovery
112:    $n(2) color black ;# Muda a cor do nodo n(2) para preta (Nam)
113:    $ns trace-annotate "2 recovery as backup"    ;# Anotações nos arquivos de rastro
114:    $app(2) set timeout_ 0.24  ;# Tempo de espera por mensagem I am alive
115:}
116:$ns at 3.2 "finish"  ;# Término da simulação
117:$ns run   ;#Execução da simulação
```

FIGURA 5.44 - Exemplo de *script* de simulação do Primário-*Backup* (continuação)

O método `listen` (Figura 5.44, linhas 78 a 80) é utilizado pelo `FullTcp` para determinar quais agentes estão a espera de mensagens. Sempre que um agente `FullTcp` for receber dados antes de enviá-los, ele deve chamar `listen`. Se isto não for feito, este agente não processará os dados recebidos, pois a conexão não será estabelecida.

Nessa implementação de Primário-*Backup*, assume-se que somente um servidor pode estar com defeito em um determinado momento. Isto significa que para simular defeito de outro primário, o anterior tem que ser recuperado. No exemplo da Figura 5.44, foi simulado exatamente este caso, no qual um antigo primário é recuperado antes do defeito do primário atual. Entretanto, foram testados alguns casos (Anexo), nos quais mais de um servidor tem defeito ao mesmo tempo e foi constatado que o algoritmo continuou apresentando comportamento correto. Contudo, não se pode afirmar que, em todos os casos que mais de um servidor apresente defeito, o algoritmo continue com funcionamento correto.

A Figura 5.45 exibe telas do Nam com alguns momentos da simulação. A Tela 1 é o início da simulação (tempo 0.0), na qual se pode observar que o nodo 1 tem um o rótulo “Primary” e um círculo ao seu redor (Figura 5.39, linhas 7 e 8) recursos usados para destacar que o nodo 1 é o servidor primário. O nodo 0 tem formato de hexágono e o rótulo “Client” para identificá-lo como cliente (Figura 5.44, linhas 43 a 45). A Tela 2 mostra o primário enviando mensagens de atualização para os *backups* (mensagens estão destacadas por uma elipse, que não faz parte do Nam). Na Tela 3, é exibido o momento (1.5s) em que ocorre defeito no primário, como se pode notar o rótulo é retirado e é inserida uma anotação que indica o defeito (Figura 5.44, linha 94). A Tela 4 mostra que o nodo 2 assume como primário; nesse momento, o rótulo e o círculo, que destacam o primário, são inseridos nesse nodo. Além disso, é enviada uma mensagem ao cliente, notificando-o da mudança de primário (mensagem destacada pela elipse). A Tela 5 exibe o momento em que o nodo 1 é recuperado como *backup*, o que é indicado pela anotação “1 recovery as backup”. Na Tela 6, ocorre defeito do primário (nodo 2), pode-se notar que uma mensagem é perdida quando os enlaces desse nodo são rompidos (pacote aparece caindo - marcado pela elipse). Na Tela 7, o nodo 1 assume novamente como primário e, na Tela 8, o nodo 2 é recuperado.

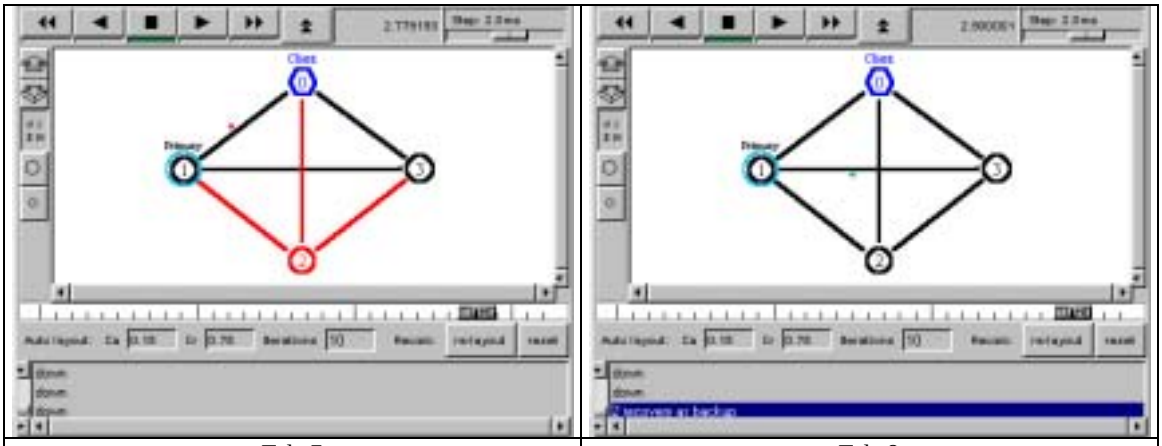


Tela 1 | Tela 2

Tela 3 | Tela 4

Tela 5 | Tela 6

FIGURA 5.45 - Telas do Nam (continua)



Tela 7	Tela 8

FIGURA 5.45 - Telas do Nam (continuação)

Com esses exemplos de telas do Nam, observa-se que esta ferramenta pode ser bastante útil para verificar o funcionamento de protocolos: com a utilização de cores, formas e rótulos para identificar diferentes nodos, enlaces e pacotes. Também, pode-se utilizar anotações para verificar onde ocorreram determinados eventos. Uma anotação definida no *script* de simulação (Figura 5.44, linhas 94, 100, 107 e 113) aparece tanto no Nam quanto no arquivo de rastro, no qual são detalhadas as trocas de mensagens em nível de transporte.

Esse exemplo de simulação gera o arquivo de *log* mostrado na Figura 5.46. Com o uso desse arquivo, juntamente com o Nam e o arquivo de rastro, pode-se conferir detalhes da execução da simulação.

```
0.2 s 1 ------ IAMALIVE
0.2 s 1 ------ IAMALIVE
0.4 s 1 ------ IAMALIVE
0.4 s 1 ------ IAMALIVE
0.6 s 1 ------ IAMALIVE
0.6 s 1 ------ IAMALIVE
0.8 s 1 ------ IAMALIVE
0.8 s 1 ------ IAMALIVE
1 c 0 ------ mID 0 REQUEST 2
1 s 1 ------ IAMALIVE
1 s 1 ------ IAMALIVE
1.031344 s 1 ------ mID 0 UPDATE
1.031344 s 1 ------ mID 0 UPDATE
1.120704 s 3 ------ mID 0 ACK
1.120704 s 2 ------ mID 0 ACK
1.131088 s 1 ------ mID 0 RESPONSE 4
1.141472 c 0 ------ mID 1 REQUEST 5
1.151856 s 1 ------ mID 1 UPDATE
1.151856 s 1 ------ mID 1 UPDATE
1.16224 s 3 ------ mID 1 ACK
1.16224 s 2 ------ mID 1 ACK
1.172624 s 1 ------ mID 1 RESPONSE 10
1.183008 c 0 ------ mID 2 REQUEST 11
1.193392 s 1 ------ mID 2 UPDATE
1.193392 s 1 ------ mID 2 UPDATE
1.2 s 1 ------ IAMALIVE
1.2 s 1 ------ IAMALIVE
1.203776 s 3 ------ mID 2 ACK
1.203776 s 2 ------ mID 2 ACK
1.21416 s 1 ------ mID 2 RESPONSE 22
1.224544 c 0 ------ mID 3 REQUEST 23
1.234928 s 1 ------ mID 3 UPDATE
1.234928 s 1 ------ mID 3 UPDATE
1.320704 s 3 ------ mID 3 ACK
```

FIGURA 5.46 - Arquivo de *log* de Primário-*Backup* (continua)

```
1.320704 s 2 ------ mID 3 ACK
1.331088 s 1 ------ mID 3 RESPONSE 46
1.341472 c 0 ------ mID 4 REQUEST 47
1.351856 s 1 ------ mID 4 UPDATE
1.351856 s 1 ------ mID 4 UPDATE
1.36224 s 3 ------ mID 4 ACK
1.36224 s 2 ------ mID 4 ACK
1.372624 s 1 ------ mID 4 RESPONSE 94
1.383008 c 0 ------ mID 5 REQUEST 95
1.393392 s 1 ------ mID 5 UPDATE
1.393392 s 1 ------ mID 5 UPDATE
1.4 s 1 ------ IAMALIVE
1.4 s 1 ------ IAMALIVE
1.403776 s 3 ------ mID 5 ACK
1.403776 s 2 ------ mID 5 ACK
1.41416 s 1 ------ mID 5 RESPONSE 190
1.424544 c 0 ------ mID 6 REQUEST 191
1.434928 s 1 ------ mID 6 UPDATE
1.434928 s 1 ------ mID 6 UPDATE
1.664496 s 3 ------ ELECTION
1.664496 s 3 ------ ELECTION
1.664496 s 2 ------ ELECTION
1.664496 s 2 ------ ELECTION
1.69552 s 2 ------ NEWPRIMARY
1.726864 c 0 ------ mID 6 REQUEST 191
1.737248 s 2 ------ mID 6 UPDATE
1.737248 s 2 ------ mID 6 UPDATE
1.747632 s 3 ------ mID 6 ACK
1.758016 s 2 ------ mID 6 RESPONSE 382
1.7684 c 0 ------ mID 7 REQUEST 383
1.778784 s 2 ------ mID 7 UPDATE
1.778784 s 2 ------ mID 7 UPDATE
1.789168 s 3 ------ mID 7 ACK
1.799552 s 2 ------ mID 7 RESPONSE 766
1.809936 c 0 ------ mID 8 REQUEST 767
1.82032 s 2 ------ mID 8 UPDATE
1.82032 s 2 ------ mID 8 UPDATE
1.830704 s 3 ------ mID 8 ACK
1.841088 s 2 ------ mID 8 RESPONSE 1534
1.851472 c 0 ------ mID 9 REQUEST 1535
1.861856 s 2 ------ mID 9 UPDATE
1.861856 s 2 ------ mID 9 UPDATE
1.87224 s 3 ------ mID 9 ACK
1.882624 s 2 ------ mID 9 RESPONSE 3070
1.893008 c 0 ------ mID 10 REQUEST 3071
1.89552 s 2 ------ IAMALIVE
1.89552 s 2 ------ IAMALIVE
1.9 s 1 ------ RECOVERY
1.9 s 1 ------ RECOVERY
1.903392 s 2 ------ mID 10 UPDATE
1.903392 s 2 ------ mID 10 UPDATE
1.975216 s 1 ------ mID 6 ACK
1.975216 s 1 ------ mID 7 ACK
1.975216 s 1 ------ mID 8 ACK
1.975216 s 1 ------ mID 9 ACK
1.975216 s 1 ------ mID 10 ACK
2.020704 s 3 ------ mID 10 ACK
2.031088 s 2 ------ mID 10 RESPONSE 6142
2.041472 c 0 ------ mID 11 REQUEST 6143
2.051856 s 2 ------ mID 11 UPDATE
2.051856 s 2 ------ mID 11 UPDATE
2.06224 s 3 ------ mID 11 ACK
2.06224 s 1 ------ mID 11 ACK
2.072624 s 2 ------ mID 11 RESPONSE 12286
2.083008 c 0 ------ mID 12 REQUEST 12287
2.093392 s 2 ------ mID 12 UPDATE
2.093392 s 2 ------ mID 12 UPDATE
2.09552 s 2 ------ IAMALIVE
2.09552 s 2 ------ IAMALIVE
2.103776 s 3 ------ mID 12 ACK
2.11416 s 2 ------ mID 12 RESPONSE 24574
2.120784 s 1 ------ mID 12 ACK
2.124544 c 0 ------ mID 13 REQUEST 24575
2.134928 s 2 ------ mID 13 UPDATE
2.134928 s 2 ------ mID 13 UPDATE
```

FIGURA 5.46 - Arquivo de *log* de Primário-*Backup* (continuação)

```
2.220704 s 3 ------ mID 13 ACK
2.220768 s 1 ------ mID 13 ACK
2.231088 s 2 ------ mID 13 RESPONSE 49150
2.241472 c 0 ------ mID 14 REQUEST 49151
2.251856 s 2 ------ mID 14 UPDATE
2.251856 s 2 ------ mID 14 UPDATE
2.26224 s 3 ------ mID 14 ACK
2.272624 s 2 ------ mID 14 RESPONSE 98302
2.283008 c 0 ------ mID 15 REQUEST 98303
2.293392 s 2 ------ mID 15 UPDATE
2.293392 s 2 ------ mID 15 UPDATE
2.29552 s 2 ------ IAMALIVE
2.29552 s 2 ------ IAMALIVE
2.303776 s 3 ------ mID 15 ACK
2.31416 s 2 ------ mID 15 RESPONSE 196606
2.320848 s 1 ------ mID 14 ACK
2.320848 s 1 ------ mID 15 ACK
2.324544 c 0 ------ mID 16 REQUEST 196607
2.334928 s 2 ------ mID 16 UPDATE
2.334928 s 2 ------ mID 16 UPDATE
2.420704 s 3 ------ mID 16 ACK
2.420832 s 1 ------ mID 16 ACK
2.431088 s 2 ------ mID 16 RESPONSE 393214
2.441472 c 0 ------ mID 17 REQUEST 393215
2.451856 s 2 ------ mID 17 UPDATE
2.451856 s 2 ------ mID 17 UPDATE
2.46224 s 3 ------ mID 17 ACK
2.472624 s 2 ------ mID 17 RESPONSE 786430
2.483008 c 0 ------ mID 18 REQUEST 786431
2.493392 s 2 ------ mID 18 UPDATE
2.493392 s 2 ------ mID 18 UPDATE
2.49552 s 2 ------ IAMALIVE
2.49552 s 2 ------ IAMALIVE
2.503776 s 3 ------ mID 18 ACK
2.51416 s 2 ------ mID 18 RESPONSE 1572862
2.520848 s 1 ------ mID 17 ACK
2.520848 s 1 ------ mID 18 ACK
2.524544 c 0 ------ mID 19 REQUEST 1572863
2.534928 s 2 ------ mID 19 UPDATE
2.534928 s 2 ------ mID 19 UPDATE
2.585328 s 3 ------ mID 6 ACK
2.59688 s 1 ------ mID 19 ACK
2.760848 s 1 ------ ELECTION
2.760848 s 1 ------ ELECTION
2.764496 s 3 ------ ELECTION
2.764496 s 3 ------ ELECTION
2.77488 s 1 ------ NEWPRIMARY
2.785264 c 0 ------ mID 19 REQUEST 1572863
2.795648 s 1 ------ mID 19 UPDATE
2.795648 s 1 ------ mID 19 UPDATE
2.810384 s 3 ------ mID 19 ACK
2.820768 s 1 ------ mID 19 RESPONSE 3145726
2.831152 c 0 ------ mID 20 REQUEST 3145727
2.841536 s 1 ------ mID 20 UPDATE
2.841536 s 1 ------ mID 20 UPDATE
2.85192 s 3 ------ mID 20 ACK
2.862304 s 1 ------ mID 20 RESPONSE 6291454
2.872688 c 0 ------ mID 21 REQUEST 6291455
2.883072 s 1 ------ mID 21 UPDATE
2.883072 s 1 ------ mID 21 UPDATE
2.893456 s 3 ------ mID 21 ACK
2.9 s 2 ------ RECOVERY
2.9 s 2 ------ RECOVERY
2.90384 s 1 ------ mID 21 RESPONSE 12582910
2.914224 c 0 ------ mID 22 REQUEST 12582911
2.924608 s 1 ------ mID 22 UPDATE
2.924608 s 1 ------ mID 22 UPDATE
2.934992 s 3 ------ mID 22 ACK
2.945376 s 1 ------ mID 22 RESPONSE 25165822
2.95576 c 0 ------ mID 23 REQUEST 25165823
2.966144 s 1 ------ mID 23 UPDATE
2.966144 s 1 ------ mID 23 UPDATE
2.97488 s 1 ------ IAMALIVE
2.97488 s 1 ------ IAMALIVE
2.976528 s 3 ------ mID 23 ACK
```

FIGURA 5.46 - Arquivo de *log* de Primário-*Backup* (continuação)

```
2.986912 s 1 ------ mID 23 RESPONSE 50331646
2.997296 c 0 ------ mID 24 REQUEST 50331647
3.007664 s 2 ------ mID 19 ACK
3.007664 s 2 ------ mID 20 ACK
3.007664 s 2 ------ mID 21 ACK
3.007664 s 2 ------ mID 22 ACK
3.007664 s 2 ------ mID 23 ACK
3.00768 s 1 ------ mID 24 UPDATE
3.00768 s 1 ------ mID 24 UPDATE
3.020704 s 3 ------ mID 24 ACK
3.028496 s 2 ------ mID 24 ACK
3.039136 s 1 ------ mID 24 RESPONSE 100663294
3.04952 c 0 ------ mID 25 REQUEST 100663295
3.059904 s 1 ------ mID 25 UPDATE
3.059904 s 1 ------ mID 25 UPDATE
3.070288 s 3 ------ mID 25 ACK
3.070288 s 2 ------ mID 25 ACK
3.080672 s 1 ------ mID 25 RESPONSE 201326590
3.091056 c 0 ------ mID 26 REQUEST 201326591
3.10144 s 1 ------ mID 26 UPDATE
3.10144 s 1 ------ mID 26 UPDATE
3.111824 s 3 ------ mID 26 ACK
3.111824 s 2 ------ mID 26 ACK
3.122208 s 1 ------ mID 26 RESPONSE 402653182
3.132592 c 0 ------ mID 27 REQUEST 402653183
3.142976 s 1 ------ mID 27 UPDATE
3.142976 s 1 ------ mID 27 UPDATE
3.15336 s 3 ------ mID 27 ACK
3.15336 s 2 ------ mID 27 ACK
3.163744 s 1 ------ mID 27 RESPONSE 805306366
3.174128 c 0 ------ mID 28 REQUEST 805306367
3.17488 s 1 ------ IAMALIVE
3.17488 s 1 ------ IAMALIVE
3.184512 s 1 ------ mID 28 UPDATE
3.184512 s 1 ------ mID 28 UPDATE
```

FIGURA 5.46 - Arquivo de *log* de Primário-*Backup* (continuação)

6 Conclusões

Desenvolver algoritmos e protocolos distribuídos é uma tarefa complexa, devido às várias fontes de indeterminismo de um sistema distribuído. Além disso, o projeto e avaliação dos mecanismos para mascarar defeitos representam grande parte deste desenvolvimento. Portanto, são necessárias técnicas que auxiliem na investigação das implementações desses algoritmos e protocolos para verificar os seus resultados, principalmente em ambientes com defeitos. Simulação é uma técnica possível para essa tarefa. No entanto, não foram encontradas ferramentas de simulação adequadas à investigação de protocolos distribuídos em configurações de rede típicas da Internet. O presente trabalho teve como objetivo investigar a adequação de um simulador de redes existente no auxílio de desenvolvimento de protocolos. Para isso, foi escolhida uma ferramenta de cunho acadêmico (*software* livre) e que possibilita o desenvolvimento de extensões.

A partir de estudo sobre a utilização no NS como instrumento de simulação de sistemas distribuídos (e não redes, como é o seu propósito inicial), foram apresentados, nesse trabalho, dois modelos de sistemas que podem ser desenvolvidos no NS. Assim, foram estudados vários métodos para simulação de defeitos nesses sistemas, buscando-se uma solução genérica que pudesse ser utilizada para simular defeitos em qualquer tipo de aplicação. Não houve sucesso no cumprimento exato desse objetivo, mas foi proposto o desenvolvimento de um método em cada aplicação que, para simular defeito de colapso, deve ser utilizado em conjunto com comandos do NS que simulem defeitos em enlaces ou agentes. Além disso, foram apresentadas algumas idéias de como simular outros tipos de defeitos no NS. Após o desenvolvimento dos modelos de sistemas distribuídos e de defeitos, foram implementados algoritmos para exemplificar casos de simulação desses modelos em sistemas síncronos, assíncronos, com e sem a ocorrência de defeitos.

Um dos desafios encontrados na execução do trabalho foi o estudo do NS, principalmente devido às deficiências de documentação: o manual [FAL2001], por exemplo, é disperso, incompleto e freqüentemente inconsistente com a implementação. Outra dificuldade foi causada pelo uso de duas linguagens, C++ e OTcl, e pelo enorme tamanho do código fonte (aproximadamente metade em C++ e metade em OTcl). Esse modelo de programação dual faz com que o código do NS se torne bastante complexo.

Baseando-se nos resultados obtidos, concluiu-se que NS pode ser de grande ajuda no desenvolvimento de sistemas distribuídos com simulação de defeitos. Tomando-se como modelo os exemplos apresentados nesse trabalho, é possível utilizá-lo para testar vários protocolos e algoritmos. Além disso, apesar da complexidade do NS, com o tempo torna-se mais fácil desenvolver aplicações em função da organização em classes, pois muitas delas não precisam ser estudadas, concentrando-se o trabalho nos aspectos relevantes do projeto que está sendo desenvolvido.

Portanto, para quem deseja utilizar o NS para simulação de sistemas distribuídos com ou sem a simulação de defeitos, sugere-se que sejam utilizados os exemplos apresentados no capítulo 5 como ponto inicial. Acredita-se que, a partir destes exemplos e tomando como referência os capítulos 3 e 4 quando necessário, o trabalho para simulação desses sistemas seja facilitado.

Além disso, ferramentas adicionais como o visualizador Nam auxiliam a verificação do funcionamento correto de algoritmos e protocolos, já que é possível fazer

uso de vários recursos como troca de cores, rótulos e anotações. Também, arquivos de *log* e de rastro da simulação ajudam no controle do funcionamento, pois no arquivo de *log* pode-se inserir informações julgadas pertinentes. No arquivo de rastro, além da visualização detalhada da troca de mensagens em nível de transporte, pode-se inserir anotações.

O prosseguimento deste trabalho inclui a expansão do modelo de defeitos suportado pelo simulador. Algumas idéias de como simular outros tipos de defeitos foram apresentadas na seção 4.3.3. Entretanto, estes casos precisam ser melhor investigados, além da necessidade de esquematizar como tal simulação seria adotada.

Este trabalho também pode ser expandido através da simulação de outros algoritmos e protocolos, para determinar de forma genérica que tipos de experimentos podem ser realizados, se existem, por exemplo, restrições na quantidade de nodos e conexões.

Anexo Códigos fonte

Primário-Backup

pb_auxr.h

```
/// Auxiliary classes for primary backup – ADUs implementation
#ifndef ns_pb_aux_h
#define ns_pb_aux_h
#include <tclcl.h>
#include "random.h"
#include "ns-process.h"
#include "pb_serverapp.h"

enum PbMsgType {
        PB_REQUEST,
        PB_RESPONSE,
        PB_UPDATE,
        PB_ACK,
        PB_IAMALIVE,
        PB_RECOVERY,
        PB_ELECTION,
};

// ADU defined in ns-process.h
const int PBDATA_COST         = 8;

// User-level packets
class PbData : public AppData {
private:
        int id_;       // ID of the sender

public:
        PbData() : AppData(PB_DATA) {}
        PbData(AppDataType t, int d) : AppData(t) { id_ = d; }
        PbData(PbData& d) : AppData(d) { id_ = d.id_; }

        inline int& id() { return id_; }
        virtual int size() const { return sizeof(PbData); }
        virtual int cost() const { return PBDATA_COST; }
        virtual AppData* copy() { return (new PbData(*this)); }
};

class PbGeneralData : public PbData {
private:
        int value_;
        int msgtype_;
        int invID_;
        int lensource_;
        char *source_;

public:
        PbGeneralData(int id, int invID, int value, const char *source, int msgtype) :
                PbData(PB_GENERAL, id) {
                value_ = value;
                msgtype_ = msgtype;
                invID_= invID;

                if ((source == NULL) || (*source == 0))
                        lensource_ = 0;
                else
                        lensource_ = strlen(source) + 1;

                if (lensource_ > 0) {
                        source_ = new char[lensource_];
                        strcpy(source_, source);
                } else
                        source_ = NULL;
        }
        PbGeneralData(PbGeneralData& d) : PbData(d) {
                value_ = d.value_;
```

```
                msgtype_ = d.msgtype_;
                invID_= d.invID_;
                if (lensource_ > 0) {
                        source_ = new char[lensource_];
                        strcpy(source_, d.source_);
                } else
                        source_ = NULL;
        }
        virtual ~PbGeneralData() {
                if (source_ != NULL)
                        delete [] source_;
        }

        virtual int size() const {
                return (sizeof(PbData)+3*sizeof(int)+lensource_);
        }
        int value() const { return value_; }
        int invID() {
                return invID_;
        }
        int msgtype() {
                return msgtype_;
        }
        char* source() {
        return source_;
        }
        virtual AppData* copy() {
                return (new PbGeneralData(*this));
        }
};

class PbNewPrimaryData : public PbData {
private:

        ServerAppR *primary_;
        int invID_;

public:
        PbNewPrimaryData(int id, int invID, ServerAppR *primary) :
                PbData(PB_NEWPRIMARY, id) {
                invID_= invID;
                primary_ = primary;
        }
        PbNewPrimaryData(PbNewPrimaryData& d) : PbData(d) {
                primary_ = primary_;
                invID_= d.invID_;
        }
        virtual ~PbNewPrimaryData() {
        }

        virtual int size() const {
                return (sizeof(PbData)+sizeof(int));
        }
        int invID() {
                return invID_;
        }
        ServerAppR* primary() {
                return primary_;
        }
        virtual AppData* copy() {
                return (new PbNewPrimaryData(*this));
        }
};
#endif // ns_pb_aux_h
```

pb_clientapp.h

```
#ifndef ns_client_h
#define ns_client_h

#include <stdlib.h>
#include <tcl.h>
#include "config.h"
#include "agent.h"
#include "ns-process.h"
#include "app.h"
```

```
#include "tcpapp.h"
#include "pb-auxr.h"

class ClientApp : public Process {
public:
        ClientApp();
        virtual ~ClientApp();

        virtual int command(int argc, const char*const* argv);
        void log(const char *fmt, ...);
        int id() const { return id_; }
        virtual void process_data(int size, AppData* d);

protected:
        int add_cnc(ServerAppR *server, TcpApp *agt);
        void delete_cnc(ServerAppR *server);
        TcpApp* lookup_cnc(ServerAppR *server);

        Tcl_HashTable *tpa_;  // TcpApp hash table
        int id_;              // Node id
        Tcl_Channel log_;     // Log file descriptor

        int initvalue_;
        int value_;
        ServerAppR *prim_;
        int invid_;           // message id
};
#endif // ns_client_h
```

pb_clientapp.cc

```
//pb_clientapp.cc Client (primary backup)

#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "tclcl.h"
#include "agent.h"
#include "app.h"
#include "pb_clientapp.h"
#include "trace.h"
#include "tcpapp.h"

//--------------------------------------------------------------------
// Client Application
// -------------------------------------------------------------------
static class ClientAppClass : public TclClass {
public:
        ClientAppClass() : TclClass("ClientApp") {}
        TclObject* create(int, const char*const*) {
                return (new ClientApp());
        }
} class_client_app;

ClientApp::ClientApp() : log_(0)
{
        tpa_ = new Tcl_HashTable;
        Tcl_InitHashTable(tpa_, TCL_ONE_WORD_KEYS);
}

ClientApp::~ClientApp()
{
        if (tpa_ != NULL) {
                Tcl_DeleteHashTable(tpa_);
                delete tpa_;
        }
}

int ClientApp::add_cnc(ServerAppR* server, TcpApp *agt)
{
        int newEntry = 1;
        Tcl_HashEntry *he = Tcl_CreateHashEntry(tpa_,
                                                (const char *)server->id(),
                                                &newEntry);
        if (he == NULL)
                return -1;
```

```
        if (newEntry)
                Tcl_SetHashValue(he, (ClientData)agt);
        return 0;
}
void ClientApp::delete_cnc(ServerAppR* server)
{
        Tcl_HashEntry *he = Tcl_FindHashEntry(tpa_,(const char *)server->id());
        if (he != NULL) {
                TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
                Tcl_DeleteHashEntry(he);
                delete cnc;
        }
}

TcpApp* ClientApp::lookup_cnc(ServerAppR* server)
{
        Tcl_HashEntry *he =
                Tcl_FindHashEntry(tpa_, (const char *)server->id());

        if (he == NULL)
                return NULL;
        return (TcpApp *)Tcl_GetHashValue(he);
}

int ClientApp::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();

        if (argc == 3) {
                if (strcmp(argv[1], "id") == 0) {
                        /*
                         * <clientapp> id <node_id>
                         */
                        id_ = atoi(argv[2]);
                        tcl.resultf("%d", id_);
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "log") == 0) {
                        /*
                         * <clientapp> log <log_file>
                         */
                        int mode;
                        log_ = Tcl_GetChannel(tcl.interp(),
                                        (char*)argv[2], &mode);
                        if (log_ == 0) {
                                tcl.resultf("%d: invalid log file handle %s\n",
                                    id_, argv[2]);
                                return TCL_ERROR;
                        }
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "primary-is") == 0) {
                        /*
                         * <clientapp> primary-is <serverapp>
                         */
                        prim_ = (ServerAppR *)TclObject::lookup(argv[2]);
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "start-request") == 0) {
                        /*
                         * <clientapp> start-request <initial_value>
                         */
                        initvalue_ = atoi (argv[2]);
                        value_ = initvalue_;
                        int bytes = 10;
                        invid_ = 0;
                        PbGeneralData *d = new PbGeneralData(id_, invid_, value_, this-
>name(), PB_REQUEST);
                        TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);
                        log("------ mID %d REQUEST %d\n", invid_, value_);
                        cnc->send(bytes, d);
                        return TCL_OK;
                }
        }else if (argc == 4) {
                if (strcmp(argv[1], "connect") == 0) {
                        /*
                         * <clientapp> connect <server> <tcpap>
```

```
                         *
                         * Associate a TcpApp agent with the given server.
                         * <tcpap> is the agent used to send packets out.
                         * We assume two-way TCP connection, therefore we
                         * only need one agent.
                         */
                        ServerAppR *server = (ServerAppR *)TclObject::lookup(argv[2]);
                        TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);

                        if (add_cnc(server, cnc)) {
                                tcl.resultf("%s: failed to connect to %s",
                                            name_, argv[2]);
                                return TCL_ERROR;
                        }
                        // Set data delivery target
                        cnc->target() = (Process*)this;
                        return TCL_OK;
                }

        }

        return TclObject::command(argc, argv);
}

void ClientApp::log(const char* fmt, ...)
{
        // Don't do anything if we don't have a log file.
        if (log_ == 0)
                return;

        char buf[10240], *p;
        sprintf(buf, TIME_FORMAT" c %d ",
                Trace::round(Scheduler::instance().clock()), id_);
        p = &(buf[strlen(buf)]);
        va_list ap;
        va_start(ap, fmt);
        vsprintf(p, fmt, ap);
        Tcl_Write(log_, buf, strlen(buf));
}

void ClientApp::process_data(int, AppData* data)
{
        if (data == NULL)
                return;
        switch (data->type()) {
           case PB_GENERAL: {
                PbGeneralData *tmp = (PbGeneralData*)data;
                 if (tmp->msgtype() == PB_RESPONSE) { // if received response
                        if (tmp->invID() == invid_) {
                                value_ = tmp->value();
                                if (value_ > 1000000000)
                                        value_ = initvalue_;

                                int bytes = 10;
                                invid_ ++;
                                value_ = value_ + 1;
                                PbGeneralData *d = new PbGeneralData(id_, invid_, value_,
this->name(), PB_REQUEST);
                                TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);
                                log("------ mID %d REQUEST %d\n", invid_, value_);
                                cnc->send(bytes, d); //new request
                        }
                }
                break;
           }
           case PB_NEWPRIMARY: {
                PbNewPrimaryData *tmp = (PbNewPrimaryData*)data;
                prim_ = tmp->primary();
                if (invid_ >= tmp->invID()) { //if the new primary do not receive the
last message (invid_ > tmp->invID())
                        int bytes = 10;             //or received but do not send the
response to the client (if invid_=tmp->invID())
                        PbGeneralData *d = new PbGeneralData(id_, invid_, value_, this-
>name(), PB_REQUEST);
                        TcpApp *cnc = (TcpApp *)lookup_cnc(prim_);
                        log("------ mID %d REQUEST %d\n", invid_, value_);
                        cnc->send(bytes, d);  // send de message again
```

```
                }
                break;
        }
        default:
                fprintf(stderr, "Bad primary backup data type %d\n",
                        data->type());
                abort();
                break;
        }
}
```

pb_serverapp.h

```
#ifndef ns_server_r_h
#define ns_server_r_h

#include <stdlib.h>
#include <tcl.h>
#include <stdio.h>
#include "config.h"
#include "agent.h"
#include "ns-process.h"
#include "app.h"
#include "tcpapp.h"

class ServerAppR;

class IamAliveTimer : public TimerHandler {
public:
        IamAliveTimer(ServerAppR *a) : TimerHandler() { a_ = a; }
protected:
        virtual void expire(Event *e);
        ServerAppR *a_;
};

class ReceiverTimer : public TimerHandler {
public:
        ReceiverTimer(ServerAppR *a) : TimerHandler() { a_ = a; }
protected:
        virtual void expire(Event *e);
        ServerAppR *a_;
};

class ServerAppR : public Process {
public:
        ServerAppR();
        virtual ~ServerAppR();

        virtual int command(int argc, const char*const* argv);
        void log(const char *fmt, ...);
        int id() const { return id_; }

        virtual void process_data(int size, AppData* d);
        virtual void iam_alive();
        virtual void on_rcvr_timeout();

protected:
        int add_cnc(ServerAppR *server, TcpApp *agt);
        void delete_cnc(ServerAppR* server);
        TcpApp* lookup_cnc(ServerAppR* server);

        void reset();
        virtual void set_timer();
        virtual void set_rcvr_timer(double timeout_rcvr);
        void sendto_servers();

        Tcl_HashTable *tpa_;  // TcpApp hash table
        int id_;              // Node id
        Tcl_Channel log_;     // Log file descriptor
        int value_;
        int rounds_;
        int minid_;
        int ro_[30];
        int diam_;
        bool Iam_primary_;
        int invid_or_r_;
```

```
        int invidprev_;
        int n_of_backups_;
        int ack_received_;
        int msg_t_;
        int bytes_;
        int detectfail_;
        double timeout_;
        double prim_timeout_;
        char *fname_;
        FILE *file_;
        TcpApp *cnc_client_;
        IamAliveTimer itimer_;
        ReceiverTimer rtimer_;
};
#endif // ns_server_h
```

pb_serverappt.cc – amnésia total

```
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "tclcl.h"
#include "agent.h"
#include "app.h"
#include "pb_clientapp.h"
#include "pb_serverapp.h"
#include "pb-auxr.h"
#include "trace.h"
#include "tcpapp.h"

//----------------------------------------------------------------------
// Server Application – total amnesia crash
// ---------------------------------------------------------------------
static class ServerAppRClass : public TclClass {
public:
        ServerAppRClass() : TclClass("ServerAppR") {}
        TclObject* create(int, const char*const*) {
               return (new ServerAppR());
        }
} class_server_appr;


ServerAppR::ServerAppR() : log_(0), value_(-1), rounds_(0), diam_(1),
Iam_primary_(false), invid_or_r_(-1), invidprev_(-1),
                      n_of_backups_(0), ack_received_(0), detectfail_(0), fname_(NULL),
itimer_(this), rtimer_(this)
{
        bind("diam_", &diam_);
        bind("prim_timeout_", &prim_timeout_);
        bind("timeout_", &timeout_);
        tpa_ = new Tcl_HashTable;
        Tcl_InitHashTable(tpa_, TCL_ONE_WORD_KEYS);
}

ServerAppR::~ServerAppR()
{
        if (Iam_primary_)
               itimer_.cancel();
        if (tpa_ != NULL) {
               Tcl_DeleteHashTable(tpa_);
               delete tpa_;
        }
        if(fname_!=NULL)
           fclose(file_);
}

void ServerAppR::reset()
{

        if (Iam_primary_)
               itimer_.cancel();

        if (fname_!=NULL) {
            fflush(file_);
            fclose(file_);
        }
```

```
        Tcl& tcl = Tcl::instance();
         tcl.evalf("$n(%d) delete-mark m(%d)", id_, id_);
        tcl.evalf("$n(%d) label \" \" ", id_);
        value_ = -1;
        rounds_=0;
        diam_ = 1;
        Iam_primary_ = false;
        invid_or_r_ = -1;
        invidprev_ = -1;
        n_of_backups_ = 0;
        ack_received_ = 0;
        detectfail_ = 0;
        fname_ = NULL;
        itimer_ = this;
        rtimer_ = this;
}

int ServerAppR::add_cnc(ServerAppR* server, TcpApp *agt)
{
        int newEntry = 1;
        Tcl_HashEntry *he = Tcl_CreateHashEntry(tpa_,
                                                (const char *)server->id(),
                                                &newEntry);
        if (he == NULL)
                return -1;
        if (newEntry)
                Tcl_SetHashValue(he, (ClientData)agt);
        return 0;
}

void ServerAppR::delete_cnc(ServerAppR* server)
{
        Tcl_HashEntry *he = Tcl_FindHashEntry(tpa_,(const char *)server->id());
        if (he != NULL) {
                TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
                Tcl_DeleteHashEntry(he);
                delete cnc;
        }
}

TcpApp* ServerAppR::lookup_cnc(ServerAppR* server)
{
        Tcl_HashEntry *he =
                Tcl_FindHashEntry(tpa_, (const char *)server->id());

        if (he == NULL)
                return NULL;
        return (TcpApp *)Tcl_GetHashValue(he);
}

int ServerAppR::command(int argc, const char*const* argv)
{
        Tcl& tcl = Tcl::instance();

        if (argc == 2) {
                if (strcmp(argv[1], "primary") == 0) {
                        Iam_primary_ = true;
                        tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1 circle", id_, id_);
                        tcl.evalf("$n(%d) label Primary", id_);
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "recovery-backup") == 0) {
                        n_of_backups_++;
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "start-Iamalive") == 0) {
                        set_timer();
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "recovery") == 0) {
                        bytes_ = 10;
                        msg_t_= PB_RECOVERY;
                        sendto_servers();
                        return TCL_OK;
                }
                else if (strcmp(argv[1], "reset") == 0) {
```

```
                reset();
                return TCL_OK;
        }
}
else if (argc == 3) {
        if (strcmp(argv[1], "id") == 0) {
                /*
                 * <serverapp> id <node_id>
                 */
                id_ = atoi(argv[2]);
                minid_ = id_;
                memset(ro_, 0, 30); // limited in 30 rounds
                tcl.resultf("%d", id_);
                return TCL_OK;
        }
        else if (strcmp(argv[1], "log") == 0) {
                /*
                 * <serverapp> log <log_file>
                 */
                int mode;
                log_ = Tcl_GetChannel(tcl.interp(),
                                (char*)argv[2], &mode);
                if (log_ == 0) {
                        tcl.resultf("%d: invalid log file handle %s\n",
                            id_, argv[2]);
                        return TCL_ERROR;
                }
                return TCL_OK;
        }
        else if (strcmp(argv[1], "backups-number") == 0) { // used to not
                n_of_backups_ = atoi(argv[2])-1; //connect all aplications
                return TCL_OK;                    //again, because backups
        }                                       //number is defined in connect
        else if (strcmp(argv[1], "file") == 0) {
                int lenfile = strlen(argv[2]) + 1;
                fname_ = new char[lenfile];
                strcpy(fname_, argv[2]);
                if((file_=fopen(fname_, "w")) == NULL) {
                    fprintf(stderr, "Could not open file\n");
                     abort();
                }
                return TCL_OK;
        }

}else if (argc == 4) {
        if (strcmp(argv[1], "connect") == 0) {
                /*
                 * <serverapp> connect <server> <tcpap>
                 *
                 * Associate a TcpApp agent with the given server.
                 * <tcpap> is the agent used to send packets out.
                 * We assume two-way TCP connection, therefore we
                 * only need one agent.
                 */
                n_of_backups_++;
                ServerAppR *server = (ServerAppR *)TclObject::lookup(argv[2]);
                TcpApp *cnc = (TcpApp *)TclObject::lookup(argv[3]);
                if (add_cnc(server, cnc)) {
                        tcl.resultf("%s: failed to connect to %s",
                                    name_, argv[2]);
                        return TCL_ERROR;
                }
                cnc->target() = (Process*)this;
                return TCL_OK;
        }
        else if (strcmp(argv[1], "connect-to-client") == 0) {
                /*
                 * <serverapp> connect-to-client <client> <tcpap>
                 *
                 * Associate a TcpApp agent with the given client <client>.
                 * <tcpap> is the agent used to send packets out.
                 * We assume two-way TCP connection, therefore we
                 * only need one agent.
                 */
                cnc_client_ = (TcpApp *)TclObject::lookup(argv[3]);
                // Set data delivery target - this aplication
                cnc_client_->target() = (Process*)this;
```

```
                        return TCL_OK;
                }

        }
        return TclObject::command(argc, argv);
}

void ServerAppR::log(const char* fmt, ...)
{
        // Don't do anything if we don't have a log file.
        if (log_ == 0)
                return;

        char buf[10240], *p;
        sprintf(buf, TIME_FORMAT" s %d ",
                Trace::round(Scheduler::instance().clock()), id_);
        p = &(buf[strlen(buf)]);
        va_list ap;
        va_start(ap, fmt);
        vsprintf(p, fmt, ap);
        Tcl_Write(log_, buf, strlen(buf));
}

void ServerAppR::sendto_servers()
{
        Tcl_HashEntry *he;
        Tcl_HashSearch hs;
        for (he = Tcl_FirstHashEntry(tpa_, &hs);  he != NULL; he =
Tcl_NextHashEntry(&hs))
        {
                TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
                PbGeneralData *d = new PbGeneralData(id_, invid_or_r_, value_, this-
>name(), msg_t_);
                switch (msg_t_) {
                        case 2:
                                log("------ mID %d UPDATE \n", invid_or_r_);
                                break;
                        case 5:
                                log("------ RECOVERY \n");
                                break;
                        case 6:
                                log("------ ELECTION \n");
                                break;
                        default :
                                log("------msgtype %d \n", msg_t_);
                                break;
                }
                cnc->send(bytes_, d);
        }
}

void ServerAppR::set_timer()
{
        itimer_.resched(prim_timeout_);
}

void ServerAppR::iam_alive ()
{
        bytes_ = 2;
        Tcl_HashEntry *he;
        Tcl_HashSearch hs;
        for (he = Tcl_FirstHashEntry(tpa_, &hs);  he != NULL; he =
Tcl_NextHashEntry(&hs))
        {
                TcpApp *cnc = (TcpApp *)Tcl_GetHashValue(he);
                PbGeneralData *d = new PbGeneralData(id_, 0, 0, this->name(),
PB_IAMALIVE);
                log("------ IAMALIVE\n");
                cnc->send(bytes_, d);
        }
        set_timer();
}

void IamAliveTimer::expire(Event*)
{
        a_->iam_alive();
}
```

```
void ServerAppR::set_rcvr_timer(double timeout_rcvr)
{
       rtimer_.resched(timeout_rcvr);
}

/*****************************************************/
/* When the backups detect primary failure             */
/*****************************************************/
void ServerAppR::on_rcvr_timeout()
{

       if(n_of_backups_ == 1)  { // just the primary - none backup
              n_of_backups_--;
              Iam_primary_ = true;
              bytes_ = 10;
              PbNewPrimaryData *d= new PbNewPrimaryData(id_, invid_or_r_, this);
              log("------ NEWPRIMARY \n");
              Tcl& tcl = Tcl::instance();
              tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1 circle", id_, id_);
              tcl.evalf("$n(%d) label Primary", id_);
              cnc_client_->send(bytes_, d);
              set_timer();
       }
       else {
           bytes_ = 10;
           rounds_ = 0;
           detectfail_=1;
           memset(ro_, 0, 30);
           invid_or_r_ = rounds_;
           value_ = minid_;
           msg_t_ = PB_ELECTION;
           sendto_servers();
       }
}

void ReceiverTimer::expire(Event*)
{
       a_->on_rcvr_timeout();
}

void ServerAppR::process_data(int, AppData* data)
{

       if (data == NULL)
              return;
        PbGeneralData *tmp = (PbGeneralData*)data;
       if (tmp->msgtype() == PB_RECOVERY) {
              n_of_backups_++;
       }
       if (Iam_primary_) { // if this aplication server is the primary
            if (tmp->msgtype() == PB_REQUEST) {  // if the message received is a request
                if(tmp->invID() > invidprev_) {  // test if the invID received is
higher than current invID
                     value_ = tmp->value() * 2;
                     if(fname_ != NULL)  // if a file was set
                            fprintf(file_, "%d\n", value_);
                }
                else if(tmp->invID() == invidprev_)
                     value_ = tmp->value() * 2;
                invid_or_r_ = tmp->invID();
                 invidprev_ =  invid_or_r_;
                if (n_of_backups_ != 0) { // for only one server - this if is necessary
                     bytes_ = 10;
                     msg_t_ = PB_UPDATE;
                     sendto_servers(); // send update message to backups
                }
                else {  // number the backups equal to zero - just the primary
                   bytes_ = 10;
                   ack_received_ = 0;
                   PbGeneralData *d = new PbGeneralData(id_, invid_or_r_, value_, this-
>name(), PB_RESPONSE);
                   log("------ mID %d RESPONSE %d\n", invid_or_r_, value_);
                   cnc_client_->send(bytes_, d);  // send response to client
                }
           }
           else if (tmp->msgtype() == PB_ACK) {  // if received an ack
```

```
                if (tmp->invID() == invid_or_r_) {
                    ack_received_++;
                    if (ack_received_ == n_of_backups_) { //if received acks from all
backups
                        bytes_ = 10;
                        ack_received_ = 0;
                        PbGeneralData *d = new PbGeneralData(id_, invid_or_r_, value_,
this->name(), PB_RESPONSE);
                        log("------ mID %d RESPONSE %d\n", invid_or_r_, value_);
                        cnc_client_->send(bytes_, d);  // send response to client
                    }
                }
          }
       }
       else { //server is a bachup
            if (tmp->msgtype() == PB_UPDATE) { //if received an update
                  value_ = tmp->value();
                  if(tmp->invID() > invidprev_) {  // test if the invID received is
higher than current invID
                     if (fname_ != NULL)
                        fprintf(file_, "%d\n", value_);
                  }
                  invid_or_r_ = tmp->invID();
                  invidprev_ =  invid_or_r_;
                  bytes_ = 10;
                  ServerAppR *primary = (ServerAppR *)TclObject::lookup(tmp->source());
//answer to the update sender
                  TcpApp *cnc = (TcpApp *)lookup_cnc(primary);
                   PbGeneralData *d = new PbGeneralData(id_, invid_or_r_, value_, this-
>name(), PB_ACK);
                  log("------ mID %d ACK\n", invid_or_r_);
                  cnc->send(bytes_, d);
            }
            else if (tmp->msgtype() == PB_IAMALIVE) {  // if the message received is an I
am alive
               set_rcvr_timer(timeout_ + (timeout_/10 * 2)); // start timer

            }
            else if (tmp->msgtype() == PB_ELECTION) { // if received a election message
                 int uid=tmp->value();
                 int r=tmp->invID();
                 if (ro_[r] < (n_of_backups_ - 1)){
                        ro_[r]++;
                        if (uid < minid_) minid_=uid;
                 }
                 if (ro_[r] == (n_of_backups_ - 1)) {
                      if(detectfail_==1){
                          rounds_++;
                          if (rounds_ < diam_)
                          {
                              invid_or_r_ = rounds_;
                              value_ = minid_;
                              msg_t_ = PB_ELECTION;
                              sendto_servers();
                          }
                          if(rounds_ == diam_) {
                              if(minid_ == id_)  {
                                     n_of_backups_--;
                                     Iam_primary_ = true;
                                     bytes_ = 10;
                                     PbNewPrimaryData *d= new PbNewPrimaryData(id_,
invid_or_r_, this);
                                     log("------ NEWPRIMARY \n");
                                     Tcl& tcl = Tcl::instance();
                                     tcl.evalf("$n(%d) add-mark m(%d) DeepSkyBlue1
circle", id_, id_);
                                     tcl.evalf("$n(%d) label Primary", id_);
                                     cnc_client_->send(bytes_, d);
                                     set_timer();
                              }
                              else  n_of_backups_--;
                          }
                      }
                 }
            }
     } //else - is not a primary
}
```

Exemplo de script – amnésia total pbt1.tcl

```
source /home/renata/extcpap/topo1.tcl
source /home/renata/extcpap/tcp_sd.tcl

global n_node
set n_node 4
set topol fully

#Create a simulator object
set ns [new Simulator]

#Open files
set f [open out.tr w]
$ns trace-all $f
set nf [open out1.nam w]
$ns namtrace-all $nf
set log [open pbt1.log w]

$ns color 1 blue
$ns color 0 red
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow

#Define a 'finish' procedure
proc finish {} {
        global ns nf f log
        $ns flush-trace
        close $nf
        close $f
         close $log
        exec nam out1.nam &
        exit 0
}

#Create nodes
create-topology $n_node $topol 1Mb 10ms DropTail

create-tcpds $n_node

$n(0) color "blue"
$n(0) shape "hexagon"
$n(0) label "Client"

set app(0) [new ClientApp]
set app(1) [new ServerAppR]
set app(2) [new ServerAppR]
set app(3) [new ServerAppR]

$app(0) id [$n(0) id]
$app(1) id [$n(1) id]
$app(2) id [$n(2) id]
$app(3) id [$n(3) id]

$app(0) log $log
$app(1) log $log
$app(2) log $log
$app(3) log $log

$app(1) file pbt1_test1
$app(2) file pbt1_test2
$app(3) file pbt1_test3

connect-all $n_node

$app(0) connect $app(1) $tcpap(0:1)
$app(0) connect $app(2) $tcpap(0:2)
$app(0) connect $app(3) $tcpap(0:3)

$app(1) connect-to-client $app(0) $tcpap(1:1)
$app(1) connect $app(2) $tcpap(1:2)
$app(1) connect $app(3) $tcpap(1:3)

$app(2) connect-to-client $app(0) $tcpap(2:1)
$app(2) connect $app(1) $tcpap(2:2)
```

```
$app(2) connect $app(3) $tcpap(2:3)

$app(3) connect-to-client $app(0) $tcpap(3:1)
$app(3) connect $app(1) $tcpap(3:2)
$app(3) connect $app(2) $tcpap(3:3)

$app(0) primary-is $app(1)
$app(1) primary

$tcp(0:2) listen
$tcp(0:3) listen
$tcp(1:1) listen
$tcp(2:2) listen
$tcp(3:2) listen

$app(1) set prim_timeout_ 0.2
$app(2) set prim_timeout_ 0.2
$app(3) set prim_timeout_ 0.2

$app(2) set timeout_ 0.2
$app(3) set timeout_ 0.2

$ns at 0.0 "$app(1) start-Iamalive"
$ns at 1.0 "$app(0) start-request 2"

$ns rtmodel-at 1.53 down $n(1)
$ns at 1.53 {
       $app(1) reset
       $n(1) color red
       $ns trace-annotate "Primary 1 crash failure"
}

$ns rtmodel-at 1.8 up $n(1)
$ns at 1.8 {
       $app(1) backups-number 3
       $app(1) file pbt1_test1
       $app(1) recovery
       $n(1) color black
       $ns trace-annotate "1 recovery as backup"
       $app(1) set prim_timeout_ 0.2
       $app(1) set timeout_ 0.2
}

$ns rtmodel-at 2.0 down $n(2)
$ns at 2.0 {
       $app(2) reset
       $n(2) color red
        $ns trace-annotate "Primary 2 crash failure"
}

$ns rtmodel-at 2.5 up $n(2)
$ns at 2.5 {

       $app(2) backups-number 3
       $app(2) file pbt1_test2
       $app(2) recovery
       $n(2) color black
       $ns trace-annotate "2 recovery as backup"
       $app(2) set prim_timeout_ 0.2
       $app(2) set timeout_ 0.2
}

$ns at 3.3 "finish"

#Run the simulation
$ns run
```

Exemplos de script – amnésia parcial

```
#pb3.tcl – com uso de detach em vez de down para simular defeitos
source /home/renata/extcpap/topo1.tcl
source /home/renata/extcpap/tcp_sd.tcl

global n_node
set n_node 4
set topol fully
```

```
#Create a simulator object
set ns [new Simulator]

#Open files
set f [open out.tr w]
$ns trace-all $f
set nf [open out1.nam w]
$ns namtrace-all $nf
set log [open pb3.log w]

$ns color 1 blue
$ns color 0 red
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow
$ns color 5 green
$ns color 6 HotPink
$ns color 5 green
$ns color 7 cyan

#Define a 'finish' procedure
proc finish {} {
        global ns nf f log
        $ns flush-trace
        close $nf
        close $f
       close $log
        exec nam out1.nam &
        exit 0
}

#Create nodes
create-topology $n_node $topol 1Mb 10ms DropTail

create-tcpds $n_node

set app(0) [new ClientApp]
set app(1) [new ServerAppR]
set app(2) [new ServerAppR]
set app(3) [new ServerAppR]

$app(0) id [$n(0) id]
$app(1) id [$n(1) id]
$app(2) id [$n(2) id]
$app(3) id [$n(3) id]

$app(0) log $log
$app(1) log $log
$app(2) log $log
$app(3) log $log

connect-all $n_node

$app(0) connect $app(1) $tcpap(0:1)
$app(0) connect $app(2) $tcpap(0:2)
$app(0) connect $app(3) $tcpap(0:3)

$app(1) connect-to-client $app(0) $tcpap(1:1)
$app(1) connect $app(2) $tcpap(1:2)
$app(1) connect $app(3) $tcpap(1:3)

$app(2) connect-to-client $app(0) $tcpap(2:1)
$app(2) connect $app(1) $tcpap(2:2)
$app(2) connect $app(3) $tcpap(2:3)

$app(3) connect-to-client $app(0) $tcpap(3:1)
$app(3) connect $app(1) $tcpap(3:2)
$app(3) connect $app(2) $tcpap(3:3)

$app(0) primary-is $app(1)
$app(1) primary

$tcp(0:2) listen
$tcp(1:1) listen
$tcp(2:2) listen
$tcp(3:2) listen
```

```
$app(1) set prim_timeout_ 0.2"
$app(2) set prim_timeout_ 0.2"
$app(3) set prim_timeout_ 0.2"
$ns at 0.0 "$app(1) start-Iamalive"
$app(2) set timeout_ 0.24"
$app(3) set timeout_ 0.24"
$ns at 1.0 "$app(0) start-request 2"

$ns at 1.53 {
       detach-ag $n_node 1
       $app(1) reset
}

$ns at 1.8 {
       attach-ag $n_node 1
       $app(1) recovery
       $app(1) set timeout_ 0.24
}

$ns at 2.0 {
       detach-ag $n_node 2
       $app(2) reset
}

$ns at 2.5 {
       attach-ag $n_node 2
       $app(2) recovery
       $app(2) set timeout_ 0.24
}

$ns at 3.3 "finish"

#Run the simulation
$ns run
```

#pb9.tcl – mais de um servidor com defeito ao mesmo tempo

```
source /home/renata/extcpap/topo1.tcl
source /home/renata/extcpap/tcp_sd.tcl

global n_node
set n_node 4
set topol fully

#Create a simulator object
set ns [new Simulator]

#Open files
set f [open out.tr w]
$ns trace-all $f
set nf [open out1.nam w]
$ns namtrace-all $nf
set log [open pbrl9b.log w]

$ns color 1 blue
$ns color 0 red
$ns color 2 purple
$ns color 3 chocolate
$ns color 4 yellow
$ns color 5 green
$ns color 6 HotPink
$ns color 5 green
$ns color 7 cyan

#Define a 'finish' procedure
proc finish {} {
        global ns nf f log
        $ns flush-trace
        close $nf
       close $log
        close $f
        exec nam out1.nam &
        exit 0
}

#Create nodes
```

```
create-topology $n_node $topol 1Mb 10ms DropTail

create-tcpds $n_node

set app(0) [new ClientApp]
set app(1) [new ServerAppR]
set app(2) [new ServerAppR]
set app(3) [new ServerAppR]

$app(0) id [$n(0) id]
$app(1) id [$n(1) id]
$app(2) id [$n(2) id]
$app(3) id [$n(3) id]

$app(0) log $log
$app(1) log $log
$app(2) log $log
$app(3) log $log

$app(1) file pbrl9b_test1
$app(2) file pbrl9b_test2
$app(3) file pbrl9b_test3

connect-all $n_node

$app(0) connect $app(1) $tcpap(0:1)
$app(0) connect $app(2) $tcpap(0:2)
$app(0) connect $app(3) $tcpap(0:3)

$app(1) connect-to-client $app(0) $tcpap(1:1)
$app(1) connect $app(2) $tcpap(1:2)
$app(1) connect $app(3) $tcpap(1:3)

$app(2) connect-to-client $app(0) $tcpap(2:1)
$app(2) connect $app(1) $tcpap(2:2)
$app(2) connect $app(3) $tcpap(2:3)

$app(3) connect-to-client $app(0) $tcpap(3:1)
$app(3) connect $app(1) $tcpap(3:2)
$app(3) connect $app(2) $tcpap(3:3)

$app(0) primary-is $app(1)
$app(1) primary

$tcp(0:2) listen
$tcp(1:1) listen
$tcp(2:2) listen
$tcp(3:2) listen
$tcp(0:3) listen

$app(1) set prim_timeout_ 0.2
$app(2) set prim_timeout_ 0.2
$app(3) set prim_timeout_ 0.2
$app(2) set timeout_ 0.24
$app(3) set timeout_ 0.24

$ns at 0.0 "$app(1) start-Iamalive"
$ns at 1.0 "$app(0) start-request 2"

$ns rtmodel-at 1.5 down $n(1)
$ns at 1.5 "$app(1) reset"

$ns rtmodel-at 2.0 down $n(2)
$ns at 2.0 "$app(2) reset"

$ns rtmodel-at 3.0 up $n(1)
$ns at 3.0 {
       $app(1) recovery
       $app(1) set prim_timeout_ 0.2
       $app(1) set timeout_ 0.24
}

$ns at 4.2 "finish"

#Run the simulation
$ns run
```

Referências

[ALV2000] ALVAREZ, Guilhermo; CRISTIAN, Flaviu. Simulation-based Testing of Communication Protocols for Dependable Embedded Systems. **Journal of Supercomputing**, Athens, v.16, n.1-2, p.93-116, May 2000.

[ALV97] ALVAREZ, Guilhermo; CRISTIAN, Flaviu. Centralized Failure Injection for Distributed, Fault-tolerant Protocol Testing. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 17., 1997, Baltimore, Maryland. **Proceedings…** Baltimore: IEEE, 1997.

[ALV97a] ALVAREZ, Guilhermo; CRISTIAN, Flaviu. Simulation-based Test of Fault-tolerant Group Membership Services. In: ANNUAL CONFERENCE ON COMPUTER ASSURANCE, 12., 1997, Gaithersburg, Maryland. **Proceedings…** Gaithersburg: IEEE, 1997.

[ALV9?] ALVAREZ, Guilhermo; CRISTIAN, Flaviu. **Cesium Programmer's Reference Guide, V1.0 Beta**. [199?]. Disponível em: <galvarez@hpl.hp.com>. Acesso em: 25 jul. 2001.

[BIR96] BIRMAN, Kenneth P. **Building Secure and Reliable Network Applications**. Greenwich: Manning, 1996.

[BRE2000] BRESLAU, Lee et al. Advances in Network Simulations. **Computer**, Los Alamitos, v.33, n.5, p.59-66, May 2000.

[BUD93] BUDHIRAJA, Navin et al. The Primary-Backup Approach In: MULLENDER, S. **Distributed Systems**. New York: Addison-Wesley, 1993. chap.8, p.199-215.

[CES2002] CESIUM Home Page. Disponível em: <http://www.hpl.hp.com/personal/Guillermo_Alvarez/cesium.html>. Acesso em: 21 mar. 2002.

[CIA94] CIAFELLA, P. et al. The Totem Protocol Development Environment. In: INTERNATIONAL CONFERENCE ON NETWORK PROTOCOLS, 1994, Boston, MA. **Proceedings…** Boston: IEEE, 1994.

[CRI91] CRISTIAN, Flaviu. Understanding Fault Tolerant Distributed Systems. **Comunications of the ACM**, New York, v.34, n.2, p.57-78, Feb. 1991.

[EST2000] ESTRIN, Deborah. et al. Network Visualization with the Nam VINT Network Animator. **Computer**, Los Alamitos, v.33, n.11, p.63-68, Nov. 2000.

[FAL2001] FALL, Kevin; VARADHAN, Kannan. **The NS Manual.** Nov. 2001. Disponível em: <http://www.isi.edu/nsnam/ns/ns-documentation.html>. Acesso em: 14 dez. 2001.

[GUE97] GUERRAOUI, R.; SCHIPER, A. Software-Based Replication for Fault Tolerance. **Computer**, Los Alamitos, v.40, n.4, p.68-74, Apr. 1997.

[GRE2000] GREIS, Marc. **Tutorial for the Network Simulator "ns"**. Disponível em: <http://www.isi.edu/nsnam/ns/tutorial/index.html>. Acesso em: 7 ago. 2000.

[HAD93] HADZILACOS, Vassos. TOUEG, Sam. Fault-Tolerant Broadcasts and Related Problems. In: MULLENDER, S. **Distributed Systems**. New York: Addison-Wesley, 1993. chap.5, p.99-102.

[HOL97] HOLZMANN, G. The Model Checker SPIN. **IEEE Transactions on Software Engineering**, New York, v.23, n.5, p.279-295, May 1997.

[JAL94] JALOTE, Pankaj. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice Hall, 1994.

[JAN2002] JANSCH-PÔRTO, Ingrid. **Reflexões sobre a Classificação de Defeitos em Sistemas Distribuídos**. Disponível em: <http://www.inf.ufrgs.br/~ingrid>. Acesso em: 27 dez. 2002.

[LEG2002] LEGOUT, Arnaud; BIERSACK, Ernest W. Revisiting the Fair Queuing Paradigm for End-to-End Congestion Control. **IEEE Network**, New York, v.16, n.5, Sept. 2002.

[LAP98] LAPRIE, Jean-Claude. Dependability of Computer Systems: from Concepts to Limits. In: INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, 1998, Johannesburg, South Africa. **Proceedings…** Johannesburg: University of the Witwatersrand, 1998.

[LIN94] LINDBLAD, C.; WETHERALL,; D. TENNENHOUSE, D. The VuSystem: A Programming System for Visual Processing of Digital Video. In: ACM MULTIMEDIA, 1994, San Francisco, CA. **Proceedings...** San Francisco: [s.n.], 1994.

[LYN96] LYNCH, Nancy A. **Distributed Algorithms**. San Francisco: Morgan Kaufmann Publishers, 1996.

[NET2001] NETWORK Simulator – NS. Disponível em: <http://www.isi.edu/nsnam/ns>. Acesso em 17: out. 2001.

[NSU2002] NS-USERS: Lista de discussão do NS. Disponível em: <ns-users@isi.edu>. Acesso em: 10 abr. 2002.

[NUN2000] NUNES, Raul C.; JANSCH-PÔRTO, Ingrid. Aplicações na Internet *versus* Comunicação em Grupo. In: SIMPÓSIO DE INFORMÁTICA DO PLANALTO MÉDIO, 2., 2000, Passo Fundo. **Anais...** Passo Fundo: UPF, 2000.

[OUS94] OUSTERHOUT, John K. **Tcl and Tk Toolkit.** Reading: Addison-Wesley Publishing Company, 1994

[RES2001] RESEARCH About NS Itself. Disponível em: <http://www.isi.edu/nsnam/ns/ns-research.html> . Acesso em 17: out. 2001.

[SCH93] SCHNEIDER, Fred B. What Good are Models and What Models are Good? In: MULLENDER, S. **Distributed Systems**. New York: Addison-Wesley, 1993. chap.2, p.20-24.

[TRI2002] TRINDADE, Renata; BARCELLOS, Marinho; JANSCH-PÔRTO, Ingrid. Simulação de Sistemas Distribuídos em Cenários com Defeitos. In: WORKSHOP DE TESTES & TOLERÂNCIA A FALHAS, 3., 2002, Búzios. **Anais...** Rio de Janeiro: UFRJ, 2002.

[VIN2000] VINT – Virtual InterNetwork Testbed. Disponível em: <http://www.isi.edu/nsnam/vint/index.html>. Acesso em: 11 set. 2000.

[WET95] WETHERALL, David. **OTcl – MIT Object Tcl:** The FAQ & Manual. Disponível em: <ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html>. Acesso em: 13 set. 2000.

[WET95a] WETHERALL, David. LINDBLAD, Christopher J. Extending Tcl for Dynamic Object-Oriented Programming. In: TCL/TK WORKSHOP, 1995, Toronto, Ontario. **Proceedings…** Toronto: [s.n.], 1995.

[YUK2000] YUKSEL, M. et al. Workload Generation for ns Simulations of Wide Area Networks and the Internet. In: COMMUNICATION NETWORKS AND DISTRIBUTED SYSTEMS MODELING AND SIMULATION CONFERENCE, 2000, San Diego, CA. **Proceedings…** San Diego: [s.n.], 2000.