MATEUS CORDEIRO INSSA

UM PROTOCOLO PARA DIFUSÃO CONFIÁVEL EM REDES DE TOPOLOGIA ARBITRÁRIA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre, Curso de Mestrado em Informática, Departamento de Informática, Universidade Federal do Paraná.

Orientador: Prof. Elias P. Duarte Jr.

CURITIBA

2002

Parecer de Aprovação de Dissertação de Mestrado

Eu, Elias P. Duarte Jr., venho por meio deste informar que o aluno Mateus Cordeiro Inssa entregou a versão final da sua Dissertação de Mestrado intitulada "Um Protocolo para Difusão Confiável em Redes de Topologia Arbitrária" devidamente corrigida.

Curitiba, 23 de agosto de 2002.

Prof. Elias P. Duarte Jr.
UFPR - Departamento de Informática
Caixa Postal 19018
CEP 81531-990 Curitiba PR
E-Mail: eliasinf.ufpr.br

Agradecimentos

À minha esposa Helga e a meus pais por todo o apoio dado.

Ao meu orientador Prof. Elias P. Duarte Jr. pela dedicação e pelo exemplo como orientador e como pesquisador.

Aos professores Renato Carmo, Roberto Hexsel e Alexandre Direne (da minha graduação) por me mostrarem os caminhos da pesquisa.

Aos diversos amigos e colegas que também contribuíram.

Resumo

Um protocolo de difusão confiável é utilizado por um processo de aplicação para transmitir uma mensagem de forma confiável a todos os nodos conectados em uma rede. Neste trabalho apresentamos um novo protocolo de difusão confiável que permite a ocorrência de múltiplos eventos de falha, tanto em nodos como em enlaces, durante a difusão. O protocolo prevê particionamentos da rede, completando nos componentes conectados nos quais pelo menos um nodo recebeu a mensagem antes do particionamento. Uma árvore de busca em largura distribuída é construída para a disseminação de mensagens, empregando o menor número de mensagens possível, bem como o menor intervalo de tempo possível, proporcional ao diâmetro da rede. Quando um nodo detecta um novo evento antes da difusão da mensagem ter sido completada, uma nova mensagem é criada, contendo tanto a mensagem anterior como informações sobre o evento detectado. Uma ferramenta prática que implementa o protocolo é apresentada, bem como resultados experimentais em um número de redes de topologias como hipercubo, anel e grafo aleatório.

Abstract

A reliable broadcast protocol is employed by an application process to send a message to every connected node in a given network. In this work we present a new reliable broadcast protocol that allows the occurrence of multiple concurrent fault events of both nodes and links. The procotol allows network partitions, completing in connected components in which at least one node has received the message. A distributed breadth-first tree is built in order to disseminate the message in the shortest possible time interval, employing the smallest possible number of messages. When a node detects a new event before the broadcast completes, a new dissemination tree is started, in which a message is broadcast that includes both the information in the previous message, and information about the new event. A practical tool that implements the protocol is presented, as well as experimental results on a number of network topologies, such as the hypercube, ring and random graph.

Conteúdo

1 Introdução 1

1.1 Diagnóstico em Redes de Topologia Arbitrária . . . 2

1.2 Difusão Confiável . . . 3

2 Difusão Confiável em Sistemas Distribuídos 6

2.1 O Algoritmo NBND . . . 7

2.2 Difusão Confiável . . . 8

2.2.1 Um Algoritmo Baseado em Inundação . . . 9

2.2.2 O Algoritmo de Schlicting . . . 11

3 Uma Nova Abordagem para Difusão Confiável 16

3.1 Descrição do Novo Algoritmo . . . 17

4 Implementação do Protocolo 22

4.1 Estados das Difusões Confiáveis . . . 23

4.2 O Protocolo de Difusão Confiável . . . 24

4.3 Comportamento na Presença Eventos . . . 26

4.4 Arquivos de Configuração . . . 27

4.4.1 /etc/srb/srb.conf . . . 28

4.4.2 /etc/srb/nodos.conf . . . 28

4.4.3 /etc/srb/rotas.conf 29
4.5 Comunicação Confiável entre Nodos 29
4.6 Protocolo de Comunicação entre Nodos 30
4.6.1 Transmissão de Mensagens 31
4.6.2 Monitoração de Nodo Filho 32
4.6.3 Detecção de Recuperação de Enlaces 34
4.7 Resultados Experimentais 35
4.7.1 Anel com 4 Nodos 35
4.7.2 Execução Com Ocorrência de Falhas 37
4.7.3 Hipercubo 39
4.7.4 Grafo Aleatório 44

5 Conclusões **49**

Referências **51**

A Código Fonte **53**

A.1 Makefile 53
A.2 srb.h 54
A.3 parser.h 54
A.4 rede.h 55
A.5 classes.h 55
A.6 classes.cpp 60
A.7 parser.cpp 85
A.8 rede.cpp 88
A.9 main.cpp 92
A.10 cliente.c 93

Capítulo 1

Introdução

A computação distribuída e paralela tem se tornado mais popular e, em alguns casos, indispensável. Sistemas podem ser desenvolvidos considerando este paradigma, visando melhorar a performance e a confiabilidade. Neste trabalho o foco é em tolerância a falhas: enlaces de comunicação e componentes do sistema podem falhar e os próprios componentes devem reconfigurar o sistema de forma a evitar um colapso completo.

A difusão confiável é uma forma de comunicação na qual todos os nodos (componentes) atingíveis pertencentes a um sistema devem receber todas as mensagens transmitidas. Pode-se considerar falhas nos nodos ou nos enlaces. Pode-se considerar ou não a recuperação dos nodos ou dos enlaces. O importante é que todos os nodos sem falha recebam as mensagens.

Este trabalho teve origem na necessidade de uma ferramenta de difusão

confiável para um algoritmo de diagnóstico de redes de topologia arbitrária, o algoritmo NBND *(Non-Broadcast Network Diagnosis)* [5], descrito na seção seguinte. Em seguida, são apresentados fundamentos de difusão confiável além do novo protocolo desenvolvido.

1.1 Diagnóstico em Redes de Topologia Arbitrária

O algoritmo NBND é uma estratégia distribuída de monitoramento de nodos e enlaces. Quando um nodo detecta um evento no sistema (nodo ou enlace falham, ou, nodo ou enlace se recuperam de uma falha), deve informar a todos os nodos não falhos a ocorrência deste evento. No algoritmo NBND com estratégia de teste baseada em bastão *(Token-Based Testing Strategy for NBND)* [6], o sistema é representado como uma árvore de busca em largura para a disseminação das mensagens contendo eventos. Importante notar que o algoritmo proposto em [6] não considera eventos durante sua difusão confiável.

1.2 Difusão Confiável

Um protocolo de difusão confiável é usado por um nodo para enviar uma mensagem a todos os nodos atingíveis de um sistema distribuído. No algoritmo proposto em [1], um nodo que desejar fazer uma difusão confiável, num sistema representável por um grafo completo, monta uma árvore para transmissão da mensagem e a transmite apenas para seus nodos filhos. Se o nodo filho não tiver filhos, responde ao nodo pai que terminou o seu trabalho. Se o nodo filho tiver filhos, então transmite a mensagem a todos os filhos. Quando seus filhos terminam, o nodo dá por encerrado o seu trabalho, avisando, por sua vez, seu nodo pai. O algoritmo considera falhas nos nodos e não nos enlaces. Não são considerados o particionamento da rede e a recuperação de nodos. O algoritmo assume o modelo *fail-stop* [3], no qual um componente falho simplesmente deixa de funcionar, não produzindo qualquer saída, e os componentes sem falha são capazes de determinar quais componentes estão falhos.

No protocolo aqui proposto para difusão confiável em sistemas de topologia arbitrária, um nodo que desejar fazer uma difusão confiável monta uma árvore de busca em largura e envia para seus nodos filhos a mensagem. Os nodos filhos montam a mesma árvore de busca em largura e transmitem a mensagem aos seus nodos filhos, e assim por diante. Quando um nodo não

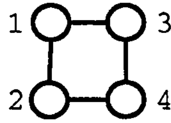

Figura 1.1: Sistema distribuído representado por um anel de 4 nodos.

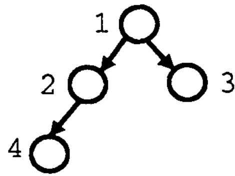

Figura 1.2: Árvore de busca em largura montada para iniciar a difusão confiável.

tem nodos filhos, encerra seu trabalho. Um nodo que tem nodos filhos encerra seu trabalho quando todos os nodos filhos (sem falhas) encerrarem. Os nodos pais monitoram os nodos filhos até que completem, permitindo, assim, detectarem falhas de comunicação com os nodos filhos. São consideradas falhas de enlaces e não de nodos.

A figura 1.1 representa um sistema distribuído em uma topologia de anel com 4 nodos. Para o nodo 1 iniciar uma difusáo confiável, monta uma árvore de busca em largura que resulta na figura 1.2. O nodo 1 tem os nodos 2 e 3 como filhos, portanto, envia a mensagem para ambos os nodos. O nodo 2 tem o nodo 4 como filho, portanto, envia a mensagem apenas para o nodo 4. Os nodos 3 e 4, por não terem filhos, não enviam a mensagem a nenhum

nodo.

Sempre que um nodo detecta um evento (falha de um enlace), inicia uma nova difusão confiável para notificar os demais nodos do sistema ou da mesma partição. Portanto, são considerados os casos em que uma falha pode particionar a rede. A própria difusão confiável faz o diagnóstico do sistema.

Há outros tipos de difusão confiável mais restritivos como a *difusão causal* e a *difusão atômica* [1, 2]. Na difusão causal, o sistema deverá garantir que mensagens geradas como consequência de uma mensagem anterior devam ser recebidas por todos os nodos após a mensagem "causa". Não há preocupação com mensagens independentes. Já na difusão atômica, o sistema deverá garantir que todos os nodos receberão todas as mensagens na mesma ordem mesmo que não haja correlação entre uma mensagem e outra.

O restante deste trabalho está organizado da seguinte maneira. O capítulo 2 apresenta algoritmos existentes para difusão confiável. O capítulo 3 contém a especificação do novo algoritmo. No capítulo 4, uma explicação detalhada sobre a implementação do algoritmo proposto. O capítulo 5 contém conclusões.

Capítulo 2

Difusão Confiável em Sistemas Distribuídos

A difusão confiável (*Reliable Broadcast*) é utilizada para compartilhar informações entre todos os nodos de um sistema distribuído. Cada sistema distribuído pode necessitar de formas diferentes de difusão. Os critérios para avaliar um algoritmo de difusão confiável incluemo custo no tráfego de pacotes, latência, segurança, complexidade de implementação, custo de processamento, garantia de entrega dos pacotes, conhecimento das falhas na transmissão, tolerância a falhas.

Neste capítulo é apresentado o algoritmo de diagnóstico NBND, origem do protocolo de difusão confiável apresentado no próximo capítulo. O algoritmo NBND precisa difundir mensagens sobre eventos (falhas ou recu-

perações de enlaces). Também são apresentados algoritmos publicados para difusão confiável.

2.1 O Algoritmo NBND

O algoritmo NBND *(Non-Broadcast Network Diagnosis)* [5] é uma estratégia distribuída de monitoramento de nodos e enlaces composto de três fases. Na primeira fase (monitoramento), cada nodo testa todos os seus nodos vizinhos. Quando um nodo detecta um evento no sistema (nodo ou link falham, ou, nodo ou link se recuperam de uma falha), o sistema passa à segunda fase (difusão), na qual o nodo notifica todos os nodos atingíveis deste evento. Terminada a difusão, o nodo inicia um algoritmo de conectividade para descobrir se houve ou não particionamento da rede.

O algoritmo NBND com estratégia de teste baseada em bastão *(Token-Based Testing Strategy for NBND)* [6], utiliza uma estratégia diferente na primeira fase (monitoramento). Os nodos são agrupados em pares e apenas um deles faz o monitoramento por vez (aquele que estiver com o bastão). Após a detecção de um evento, é iniciado um algoritmo de difusão que representa o sistema como uma árvore de busca em largura, na qual as mensagens notificando o evento são enviadas. A árvore de busca em largura tem a propriedade de ter o caminho mais curto possível para se chegar a qualquer nodo

(a partir do nodo raiz).

Como o objetivo principal da estratégia dos algoritmos NBND descritos anteriormente é o diagnóstico dos eventos em um sistema, não têm preocupação forte quanto à disseminação das notificações dos eventos. O algoritmo NBND não opera corretamente na presença de múltiplos eventos simultâneos pois a estratégia de difusão assume que não ocorrem eventos enquanto outro evento ainda está sendo diagnosticado. Esta asserção deve-se ao fato de que estes algoritmos não são tolerantes a falhas no momento da difusão das mensagens de notificação. Em outras palavras, a difusão não é confiável.

2.2 Difusão Confiável

A difusão confiável *(reliable broadcast)* é uma das técnicas utilizadas em tolerância de falhas para garantir que todos os nodos operacionais de um sistema distribuído sempre recebam todas as mensagens. Cada mensagem transmitida deve conter um identificador para permitir a detecção de mensagens repetidas. A difusão confiável deve garantir três propriedades [2]:

1. Todos os processos corretos concordam com o conjunto de mensagens que transmitem;

2. Todas as mensagens difundidas por processos corretos são transmitidas;

3. Nenhuma mensagem com erro será transmitida.

Nas seções a seguir apresentamos dois algoritmos publicados para a difusão confiável, um algoritmo baseado em inundação de mensagens e o algoritmo de Schlicting.

2.2.1 Um Algoritmo Baseado em Inundação

Em [2] é descrito um algoritmo simples para difusão confiável. Considera-se que as falhas não particionam a rede. Não se monta uma árvore para que seja feita a difusão da mensagem. Cada nodo envia a mensagem para todos os seus nodos vizinhos (inclusive a si mesmo). Se receber uma mensagem repetida, o nodo deve ignorá-la. Uma otimização óbvia citada seria evitar que, por exemplo, o nodo p envie a mensagem m para o nodo q se tinha recebido do próprio nodo q. Como o algoritmo faz a difusão da mensagem utilizando inundação, mesmo se houver falhas em enlaces (que não particionem a rede), todos os nodos sem falha recebem a mensagem. Como ilustra a Figura 2.1 na qual uma aresta com do nodo i para o nodo j representa o envio de uma mensagem do nodo i para o nodo j, mesmo que o enlace entre os nodos A e B esteja falho, a mensagem chega ao nodo B através do outro enlace que mantém o nodo B conectado à rede. Este algoritmo é bem simples e não contém falhas, porém, a quantidade de mensagens transmitidas pode

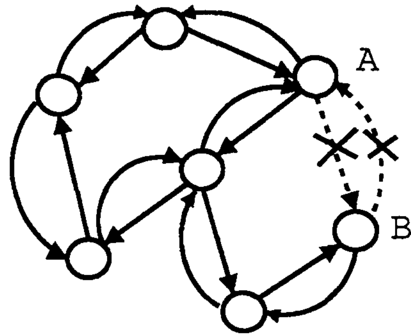

Figura 2.1: Inundação: o nodo `B` recebe a mensagem mesmo com a falha no enlace A<->B.

representar um alto tráfego na rede.

Segue o algoritmo em pseudo-código:

Chamada Inicial:

`ReliableBroadcast (nodo, identificador, mensagem)`

O procedimento executado no nodo *i* segue abaixo:

id é o identificador da difusão, *msg* é a mensagem a ser difundida.

`ReliableBroadcast(i,id,msg)`

- Se *id* já é conhecido, então a mensagem é repetida e deve ser ignorada;
- Caso contrário, para cada nodo vizinho *k*, enviar a mensagem;
- Enviar a mensagem para si mesmo;

2.2.2 O Algoritmo de Schlicting

No algoritmo apresentado em [3] considera-se que as falhas não particionam a rede, ou seja, não há casos em que nodos operacionais tornam-se inatingíveis. Este sistema distribuído é representado por um grafo completo onde cada nodo tem um enlace de comunicação com todos os demais nodos. O modelo de falhas assumido é conhecido como *fail-stop* [3], no qual um componente falho simplesmente deixa de funcionar, não produzindo qualquer saída, e os componentes sem falha são capazes de determinar quais componentes estão falhos.

Neste algoritmo, um nodo inicia uma difusão confiável caracterizando o sistema como uma árvore, sendo ele a raiz desta árvore. A árvore é definida estaticamente e conhecida por todos os nodos. A estrutura da árvore não precisa estar relacionada diretamente com a estrutura física da rede. O nodo então inicia a difusão da mensagem enviando-a a todos os nodos filhos. Os nodos filhos são responsáveis por reenviar a mensagem a seus próprios filhos e assim por diante. O nodo que não tiver filhos retorna uma mensagem ao seu nodo pai confirmando (enviando um *acknowledgement*) do recebimento da mensagem. Se o nodo tiver filhos, responderá ao nodo pai somente quando todos os seus nodos filhos confirmarem o recebimento da mensagem.

Quando um nodo falha, o nodo pai deverá assumir o seu lugar e reenviar

a mensagem aos nodos filhos daquele nodo falho. Se um nodo receber uma mesma mensagem mais de uma vez, confirma a recepção mas não realiza a difusão novamente. Há apenas um problema: e se o nodo raiz falhar ? Ele é o único nodo na árvore que não tem um nodo pai e, por causa disso, quem irá substituí-lo em caso de falha? Se ninguém assumir a posição de nodo raiz é possível que alguns nodos operacionais (sem falha) não recebam a mensagem, violando a definição e o objetivo da difusão confiável.

A Figura 2.2 ilustra este caso. O nodo A envia a mensagem para o nodo B e falha antes de enviá-la para o nodo D. O nodo B envia a mensagem para o nodo C, completando sua tarefa. O nodo D, apesar de não estar falho, nunca receberá a mensagem. Conseqüentemente, o nodo E também nunca receberá esta mensagem. A solução apresentada para este problema é a seguinte: os nodos filhos diretos da raiz ficam permanentemente monitorando o nodo raiz para saber se ainda está operacional ou entrou em falha até que a difusão esteja terminada. Se o nodo raiz estiver falho, pelo menos um de seus nodos filhos deverá assumirá o lugar do nodo raiz.

Se todos os nodos que possuem a mensagem ficam falhos, não há como continuar a difusão. Se assumirmos que um nodo não inicia uma difusão antes que a difusão anterior tenha sido completada, cada nodo pode assumir que a difusão anterior terminou quando receber uma nova mensagem com número de sequência maior.

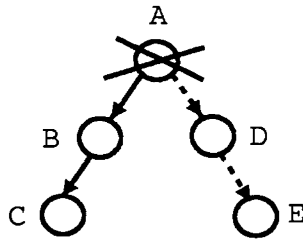

Figura 2.2: Nodo raiz falha antes de enviar mensagem a todos os nodos filhos.

Segue o algoritmo em pseudo-código:

Chamada Inicial:

`ReliableBroadcast(raiz,raiz,mensagem)`

O procedimento executado no nodo i segue abaixo,

onde *msg* se refere à mensagem em difusão

```
ReliableBroadcast (nodo i, raiz, msg)

• Constrói a árvore a partir da raiz;

• SE msg repetida, ignorar;

• Se nodo i não tem filhos então completa,
  notificando seu nodo pai;

• Senão - para cada filho k :
  ReliableBroadcast (nodo k, raiz, msg)

    – nodo i espera resposta de cada nodo k informando
      que completou.  Quando todos os nodos filhos sem
      falhas completam, o nodo i completa;
    – nodo i, se for filho do nodo raiz,
      monitora o nodo raiz.

• QUANDO ANTES DE COMPLETAR, O NODO i DETECTA
  UMA FALHA NO NODO K:
```

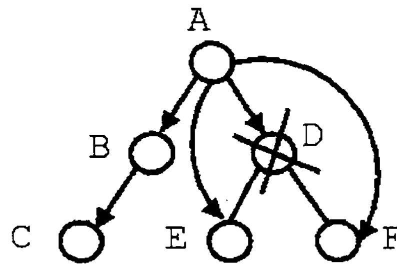

Figura 2.3: Nodo `A` assume a tarefa do nodo `D` falho.

– `Para cada filho j de k, ReliableBroadcast(j,raiz,msg).`

Importante notar que:

- São consideradas falhas de nodos;
- Uma vez detectada uma falha, o nodo pai do nodo falho na árvore de disseminação assume o trabalho do nodo falho. Como exemplo, observe a Figura 2.3: o nodo A envia a mensagem para o nodo B, mas ao tentar enviá-la para o nodo D, descobre que o nodo D está falho. O nodo A assume a função do nodo D na árvore e envia a mensagem para os nodos E e F que eram filhos do nodo D.
- Nodos falhos não se recuperam;
- Falhas de nodos nunca provocam o particionamento da rede;
- Em tempo finito, todos os nodos devem ter conhecimento das falhas em decorrência da asserção do modelo *fail-stop* [1]. Para implementar

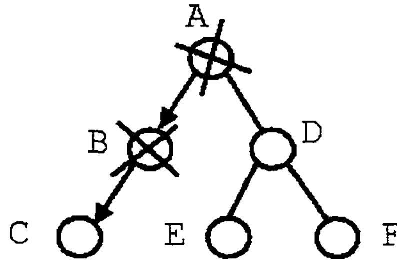

Figura 2.4: Nodo A raiz e nodo B em falha.

a asserção pode-se utilizar uma ferramenta de diagnóstico de falhas em conjunto com a difusão confiável [4];

- O algoritmo falha na circunstância representada pela Figura 2.4. Se o nodo A transmitiu a mensagem ao nodo B, que por sua vez transmitiu a mensagem para o nodo C e ambos os nodos A e B falham antes que o nodo A envia a mensagem para o nodo D, então toda a sub-árvore com raiz em D nunca receberá a mensagem. O nodo C não realiza qualquer monitoramento para testar se o nodo B ainda está sem falha e, conseqüentemente, não o substituirá. Como o nodo C não substitui o nodo B, nunca detectará que o nodo A (raiz) também está em falha e não irá substituí-lo também. Em resumo: o nodo C tem a mensagem mas não sabe que toda a sub-árvore iniciada no nodo D não a tem, e a difusão confiável não é concluída com sucesso.

Capítulo 3

Uma Nova Abordagem para Difusão Confiável

Neste capítulo é apresentado um novo algoritmo de difusão confiável que permite a ocorrência de múltiplos eventos durante a disseminação de informações. A motivação inicial foi de servir como ferramenta de difusão confiável para o algoritmo de diagnóstico NBND apresentado no capítulo 2. Com a descoberta de deficiências nos algoritmos de difusão confiável existentes, optou-se pelo desenvolvimento de um novo algoritmo.

O novo algoritmo desenvolvido cumpre as metas estabelecidas. Permite a ocorrência de múltiplos eventos durante a difusão de informações. Atua em redes de topologia arbitrária. Opera corretamente mesmo na ocorrência de particionamentos. Por se basear na árvore de busca em largura utilizada pelo

algoritmo de difusão usada pelo NBND, utiliza o mínimo possível de recursos de rede, evitando que nodos recebam mensagens redundantes, apresentando menor latência possível, proporcional ao diâmetro da árvore.

Uma das condições para uma difusão confiável é que a comunicação nodo a nodo seja confiável *(reliable unicast)*. A partir desta comunicação confiável nodo a nodo, é possível obter informações sobre falhas que ocorrem na rede. Portanto, durante a transmissão de mensagens, o novo algoritmo apresenta como funcionalidade o diagnóstico de eventos.

3.1 Descrição do Novo Algoritmo

O algoritmo aqui descrito funciona corretamente em redes de topologia arbitrária na presença de eventos que levam ao particionamento da rede. Em uma rede de topologia arbitrária consistindo de N nodos, cada par de nodos pode ou não estar conectado através de uma enlace de comunicação direto. Os eventos (falhas) considerados são em enlaces e não nos nodos, e, desta forma, é necessária a reconfiguração do sistema após cada evento detectado, pois um enlace falho não pode ser utilizado na difusão.

Todo nodo deve guardar, para cada difusão confiável, informações como o nodo raiz da difusão, seu nodo pai (se não for a própria raiz) na árvore de difusão, o identificador da mensagem (para controle de mensagens repetidas

e/ou relacionamento com futuras difusões), uma lista de eventos (falhas nos enlaces). Isto é necessário para garantir que a árvore de busca em largura montada em cada nodo a ser usada para disseminação das mensagens seja a mesma em todos os nodos.

Determinado nodo i inicia uma difusão confiável construindo uma árvore de busca em largura a partir dele mesmo, sendo o nodo i a raiz desta árvore. Se não tiver nodos filhos o nodo i completa. Se tiver nodos filhos, mas todos os enlaces entre o nodo i e seus nodos filhos estão falhos, então testa uma única vez cada enlace para detectar possíveis recuperações. Se nenhum enlace se recuperou, o nodo i completa seu trabalho. Se tiver nodos filhos acessíveis, para cada nodo k filho (acessível), envia a mensagem informando que a árvore de disseminação a ser montada pelo nodo k deverá considerar o nodo i como raiz. Quando todos os nodos filhos completarem, então toda a disseminação terá sido completada. Cada nodo k monta a árvore de disseminação considerando o nodo i como raiz e, da mesma forma que o nodo i, envia a mensagem aos seus nodos filhos e, se todos completam, este nodo k também completa.

Todo nodo monitora seus nodos filhos para saber se já completaram ou não. Enquanto a difusão não termina naquele nível da árvore, o nodo pai pode detectar falhas nos enlaces com os nodos filhos durante a monitoração.

Os nodos filhos também detectam a falha de enlace com o nodo pai se este não monitorá-los.

Se algum evento de falha ocorrer, o nodo desiste da difusão confiável corrente e inicia outra difusão sendo a raiz da árvore desta nova difusão. A mensagem enviada é composta pela lista de eventos mais a mensagem original. Objetivos: notificar a todos os nodos o(s) evento(s) ocorrido(s); garantir que todos os nodos ativos recebam aquela mensagem original, o princípio básico da difusão confiável. Quando os demais nodos recebem esta nova mensagem, descartam a difusão da mensagem original e fazem uma nova difusão com esta nova mensagem.

O algoritmo em pseudo-código é apresentado a seguir:

Chamada Inicial:

ReliableBroadcast(raiz,raiz,identificador,[],[],mensagem)

Os parâmetros são, respectivamente, o nodo que executará o algoritmo (inicialmente, a própria raiz), nodo raiz da difusão, identificador da mensagem, lista de identificadores de difusões anteriores (inicialmente vazia), lista de eventos (inicialmente vazia), a mensagem propriamente dita.

O procedimento executado no nodo i:

ReliableBroadcast (nodo i, raiz, id, [id anteriores], [eventos], msg)

Os parâmetros são, respectivamente, o nodo que executará o algoritmo (nodo i), nodo raiz da difusão, identificador da mensagem, lista de identificadores de difusões anteriores (se esta difusão é decorrente de eventos em difusões anteriores), lista de eventos (eventos conhecidos pelo nodo pai nesta árvore de difusão), a mensagem propriamente dita.

- SE o identificador id é repetido, ignorar mensagem
- SE a lista [id anteriores] não for vazia
  ENTÃO desiste da difusão de todos aqueles identificadores;
- Constrói a árvore de busca em largura a partir
  da raiz considerando os eventos;
- Se nodo i não tem filhos então completa;
- Senão
  - para cada filho k :
    ReliableBroadcast (**nodo k, raiz, [id anteriores], [eventos], msg**);
  - nodo i monitora cada nodo k até que complete. Quando todos os nodos filhos sem falhas completam, o nodo i completa;
  - nodo i monitora também seu nodo pai na árvore
- QUANDO ANTES DE COMPLETAR, O NODO i DETECTA
  UM NOVO EVENTO e:
  - Encerra o Difusão Confiável da mensagem corrente;
  - Cria novo identificador de difusão id';
  - Faz nova difusão confiável com identificador id', concatenando o identificador da difusão confiável corrente (id) na lista [id anteriores], e concatenando o evento e na lista [eventos]
    ReliableBroadcast(i,i,id',id | [id anteriores], e **| [eventos])**.

No capítulo seguinte é descrita a implementação de um protocolo que implementa o algoritmo descrito.

Capítulo 4

Implementação do Protocolo

O protocolo de difusão confiável em redes de topologia arbitrária foi implementado em C++ utilizando orientação a objetos e múltiplas *threads* (biblioteca pthreads [7, 8]), sendo pelo menos uma *thread* para cada difusão confiável em execução. No caso do sistema operacional Linux (plataforma onde foi desenvolvido), de uma forma bastante simplificada, pode-se considerar *threads* como processos independentes em que toda a memória é compartilhada. Conseqüentemente, operações críticas em memória exigem sincronização (semáforos), mas operações que podem bloquear um processo, como entrada e saída, bloqueiam apenas a *thread* que disparou a operação e não todo o sistema.

Nas seções a seguir, são descritos os estados que um nodo executando uma difusão confiável pode assumir, a implementação, o comportamento na

presença de eventos, a configuração para cada nodo, o protocolo de comunicação (comunicação confiável entre nodos, formato dos pacotes), e alguns exemplos de execução do algoritmo.

4.1 Estados das Difusões Confiáveis

Cada nodo pode estar participando de diversas difusões confiáveis simultaneamente, portanto, o estado não se refere ao nodo, mas sim, à cada difusão confiável. As difusões confiáveis, podem assumir os seguintes estados:

INICIO: Nodo contém a mensagem a ser transmitida e está montando a árvore para difundir a mensagem a seus nodos filhos;

EM DIFUSAO: Nodo está enviando a mensagem a seus nodos filhos;

DIFUNDIDO: Todos os nodos filhos já tem a mensagem. Este estado é bastante breve pois o nodo passa automaticamente para o estado seguinte;

MONITORANDO: Nodo está monitorando seus nodos filhos;

CONCLUIDO: Todos os nodos filhos completaram sua parte na difusão da mensagem, portanto, o nodo também completa;

ABORTADO: Nodo detectou um evento ou recebeu a mesma mensagem de novo contendo eventos e abortou a esta difusão confiável

A Figura 4.1 representa o diagrama de estados (detalhado) de uma difusão confiável.

4.2 O Protocolo de Difusão Confiável

Determinado nodo i vai iniciar uma difusão confiável. Para poder realizá-la, dispara uma nova *thread* com um identificador desta difusão confiável e constrói a árvore de busca em largura a partir da estrutura de enlaces corrente. A estrutura pode ter sido alterada em caso de eventos como falhas, sendo o próprio nodo i a raiz da árvore. O identificador da difusão confiável é composto pelo identificador deste nodo (não é permitido que dois nodos possuam a mesma identificação) e por um número sequencial gerado neste nodo. Semáforos são usados para evitar que duas threads alterem ao mesmo tempo os mesmos objetos, caso contrário, rapidamente ocorreriam acessos a dados e a endereços inválidos de memória provocando, respectivamente, o funcionamento incorreto do programa e seu término.

Se não tiver nodos filhos, o nodo i completa. Se tiver nodos filhos, mas há falhas de enlace para todos, então completa. Se tiver nodos filhos acessáveis, para cada nodo k filho, envia a mensagem (pacote da seção 4.6.1) informando

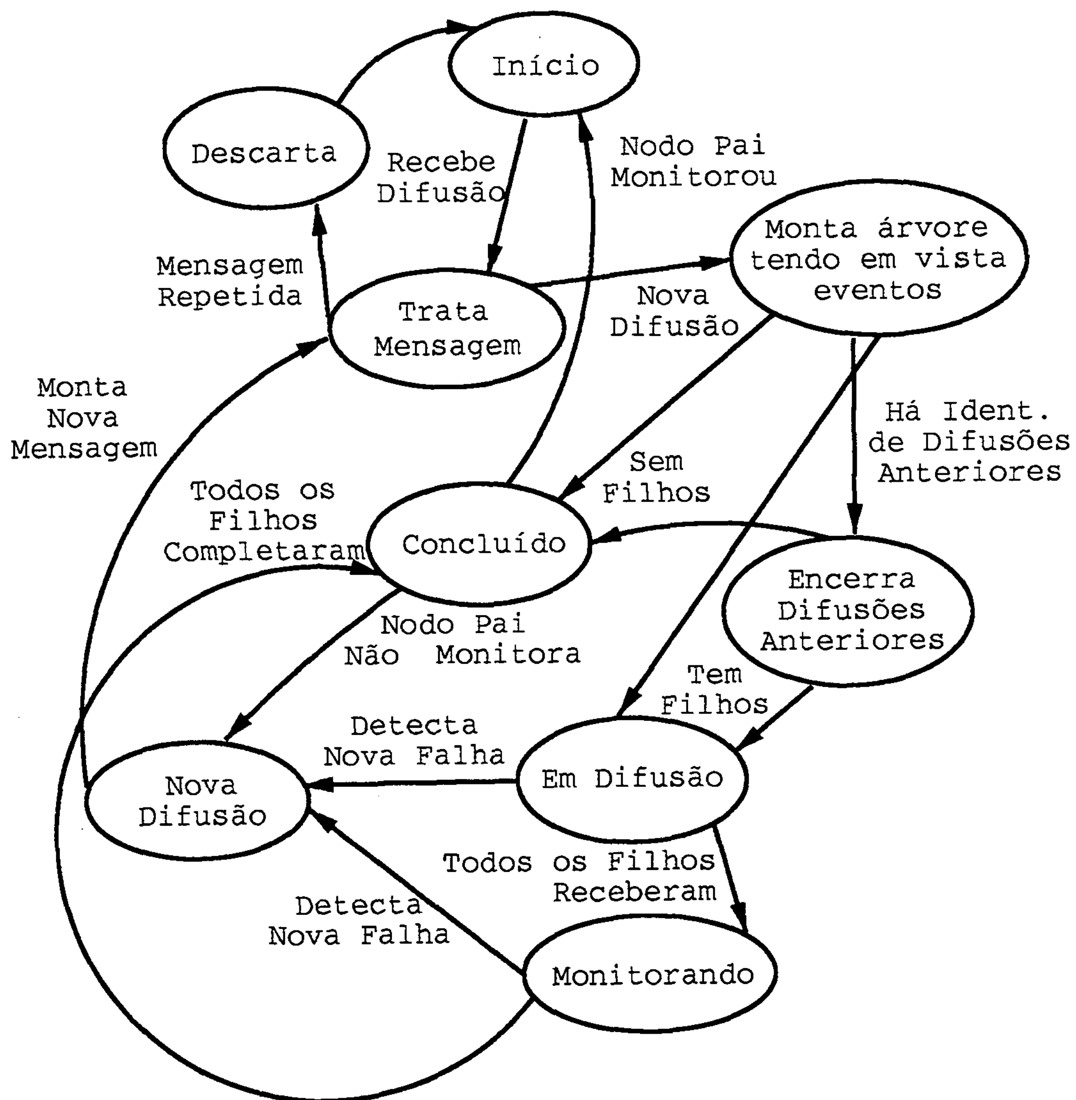

Figura 4.1: Diagrama de estados.

qual o identificador desta difusão confiável e que a árvore de disseminação montada pelo nodo k deverá considerar o nodo i como raiz. Quando todos os nodos filhos completarem, então toda a disseminação terá sido completada. Cada nodo k também dispara uma nova *thread* e monta a árvore de disseminação considerando o nodo i como raiz e, da mesma forma que o nodo i, envia a mensagem aos seus nodos filhos e, se todos completam, este nodo k também completa.

Todo nodo monitora (pacote da seção 4.6.2) seus nodos filhos para saber se já completaram ou não. Enquanto a difusão não termina naquele nível da árvore, o nodo pai pode detectar falhas nos enlaces com os nodos filhos caso não consiga, durante a monitoração, a informação sobre o estado da difusão em algum nodo filho . Os nodos filhos também detectam a falha de enlace com o nodo pai caso não sejam monitorados pelo nodo pai (pacote da seção 4.6.2). Um nodo filho só vai considerar sua "tarefa de filho" realmente completa se seu nodo pai monitorá-lo depois de ter completado sua difusão confiável.

4.3 Comportamento na Presença Eventos

Em caso dos seguintes eventos:

- falha de enlace com um nodo filho;

- falha de enlace com o nodo pai;
- recuperação de enlace

O nodo desiste da difusão confiável corrente e inicia outra com um novo identificador sendo a raiz da árvore. A mensagem enviada será composta pelo novo identificador, o(s) identificador(es) anterior(es), lista de eventos e a mensagem original. Objetivos: notificar a todos os nodos o(s) evento(s) ocorrido(s); garantir que todos os nodos ativos recebam aquela mensagem original (princípio básico da difusão confiável). A estrutura de enlaces da rede conhecida pelo nodo é modificada: enlaces recuperados voltam para a estrutura, enlaces falhos são retirados da estrutura, sendo guardados em uma lista de enlaces falhos.

Quando os demais nodos recebem esta nova mensagem, descartam a todas as difusões (as respectivas *threads* são encerradas) cujos identificadores de mensagem estiverem listados como identificadores anteriores nesta nova mensagem. Em seguida, fazem uma nova difusão com esta nova mensagem.

4.4 Arquivos de Configuração

Considere uma máquina na Internet executando os protocolos *TCP/IP* (Transmission Control Protocol/Internet Protocol [9]). O protocolo para difusão

confiável se localiza na camada de aplicação, utilizando o protocolo da camada de transporte *UDP* (User Datagram Protocol). Para que esta máquina possa enviar pacotes para outras máquinas pertencentes ao sistema, e para que o programa possa executar corretamente a difusão confiável, é necessário o processamento de três arquivos de configuração, descritos a seguir.

4.4.1 /etc/srb/srb.conf

O arquivo `/etc/srb/srb.conf` contém a configuração do nodo. Os valores são apenas um exemplo:

Campo	Valor	Descrição
IDENTIFICATION	1	Número do nodo
MAX_MESSAGE_SIZE	2000	Tamanho máximo dos pacotes
MAX_NODES	100	Número máximo de nodos no sistema
BROADCAST_PORT	10011	Porta UDP para a difusão
PING_PORT	10012	Porta para teste de recuperação
SERVICE_PORT	10013	Porta UDP do sistema para a comunicação segura com um cliente local
CLIENT_PORT	4545	Porta UDP do cliente local para a comunicação segura com o sistema

4.4.2 /etc/srb/nodos.conf

O arquivo `/etc/srb/nodos.conf` contém a configuração de todos os nodos do sistema. Deve ser exatamente igual em todos os nodos. Exemplo:

Nodo	Endereço IP	Porta para Difusão	Porta para Ping
1	127.0.0.1	10011	10012
2	127.0.0.1	10021	10022
3	127.0.0.1	10031	10032
4	127.0.0.1	10041	10042

4.4.3 /etc/srb/rotas.conf

O arquivo `/etc/srb/rotas.conf` contém a relação de todos os enlaces do sistema, bem como a senha de comunicação de cada enlace (um nodo deve conhecer, pelo menos, as senhas dos seus enlaces). Exemplo:

Nodo	Nodo	Senha do Enlace
1	2	senha12
2	4	senha24
3	1	senha13
3	4	senha34

4.5 Comunicação Confiável entre Nodos

A comunicação confiável entre dois nodos é uma premissa para a difusão confiável. É usada quando um nodo envia uma mensagem para um outro nodo mas, principalmente, na monitoração de nodos.

A versão inicial do algoritmo foi implementada com o pacote *Scotty* [10] (uma extensão à linguagem *Tcl* [11]) utilizando *SNMP* (Simple Network Management Protocol [12]) para a comunicação entre os nodos. O protocolo SNMP garante a confiabilidade e a segurança na troca de mensagens entre os nodos *(reliable unicast)*. Como implementação seguinte é implementada em C++, a comunicação confiável entre dois nodos teve que ser implemen-

tada. A troca de mensagens entre nodos é feita utilizando o protocolo UDP, com senha para cada enlace de comunicação (na versão inicial baseada em SNMP, a senha (comunidade) refere-se ao nodo e não ao enlace). Se a senha estiver errada, o pacote será descartado, sem notificar o nodo que enviou a mensagem.

Para cada mensagem enviada, uma mensagem de confirmação deve ser recebida, e para cada mensagem recebida, uma mensagem de confirmação deve ser enviada. Mensagens de confirmação não são enviadas quando se recebe uma mensagem de confirmação. Caso o nodo que enviou a mensagem não receba a confirmação, a mensagem é retransmitida. Se atingir o limite de retransmissões, o enlace (e não o nodo) é considerado falho.

4.6 Protocolo de Comunicação entre Nodos

Foi desenvolvido um protocolo de comunicação entre os nodos. Como descrito na seção anterior, para cada mensagem deve haver uma mensagem de confirmação. Se um pacote é recebido fora do formato (ou a senha do enlace estiver errada), o nodo descarta o pacote sem enviar qualquer tipo de notificação. O pacote chamado de *Ping* [9] é definido na seção 4.6.3, mas só será utilizado na segunda versão do algoritmo onde são consideradas recuperações de nodos. A seguir, o formato dos pacotes é descrito byte a byte.

4.6.1 Transmissão de Mensagens

O pacote da Figura 4.2 é utilizado toda vez que um nodo precisar enviar um pacote para seus nodos filhos utilizando a difusão confiável. Os campos indicados são descritos a seguir.

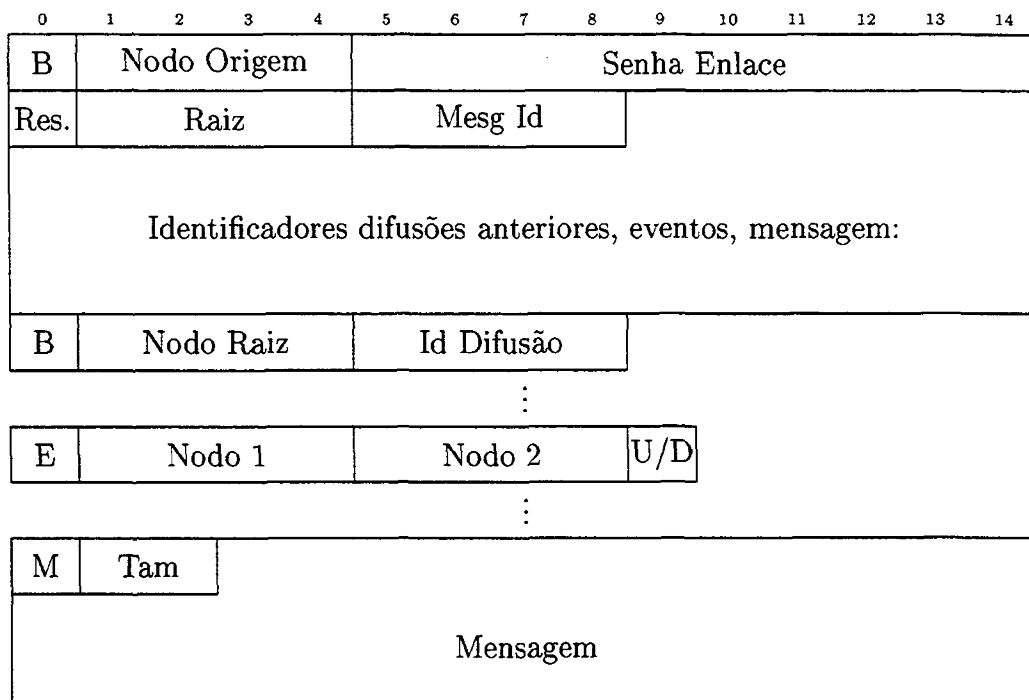

Figura 4.2: Transmissão de mensagens.

(0): B, indica difusão de mensagem (Broadcast);

(1-4): Identificador do nodo que enviou a mensagem;

(5-14): Senha do enlace;

(15): Reservado;

(16-19): Raiz da difusão;

(20-23): Identificador da mensagem;

(24-): Identificadores de difusões anteriores, eventos, mensagem

O pacote da Figura 4.3 é utilizado na confirmação do recebimento do pacote acima descrito.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>A</td><td colspan="4">Nodo Respondendo</td><td colspan="10">Senha Enlace</td></tr>
<tr><td>Res.</td><td colspan="4">Raiz</td><td colspan="4">Mesg Id</td></tr>
</table>

Figura 4.3: Confirmação do recebimento da mensagem.

(0): **A**, indica confirmação de mensagem (**A**ck);

(1-4): Identificador do nodo que está confirmando o recebimento da mensagem;

(5-14): Senha do enlace;

(15): Reservado;

(16-19): Raiz da difusão;

(20-23): Identificador da mensagem;

4.6.2 Monitoração de Nodo Filho

O pacote da Figura 4.4 utilizado por um nodo para consultar o estado de seus nodos filhos, em especial, para saber se já completaram sua difusão confiável. Os campos são descritos a seguir.

(0): **G**, Pedido de informações sobre o status desta difusão no nodo filho (**G**et Status);

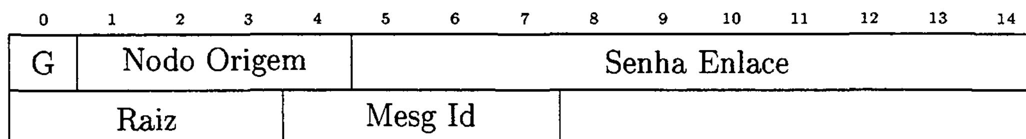

Figura 4.4: Consulta de estado de nodos filhos.

(1-4): Identificador do nodo que enviou a mensagem;

(5-14): Senha do enlace;

(15-18): Raiz da difusão;

(19-22): Identificador da mensagem;

O pacote da Figura 4.5 é utilizado para responder ao pedido de consulta de estado feita pelo nodo pai.

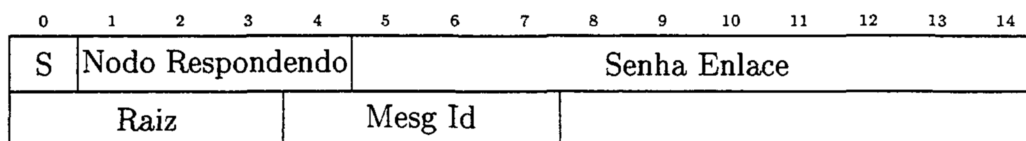

Figura 4.5: Resposta ao nodo pai sobre o estado.

(0): S, indica confirmação de mensagem (Status);

(1-4): Identificador do nodo que está informando o estado desta difusão confiável;

(5-14): Senha do enlace;

(15-18): Raiz da difusão;

(19-22): Identificador da mensagem;

4.6.3 Detecção de Recuperação de Enlaces

Conforme mencionado anteriormente, os pacotes desta seção só serão usados na segunda versão do algoritmo, o qual comtemplará a recuperação de enlaces que estavam falhos.

O pacote da Figura 4.6 é utilizado por um nodo que quer consultar se um determinado enlace falho se recuperou. É enviado para a porta de *ping* [9] do nodo cujo enlace está falho.

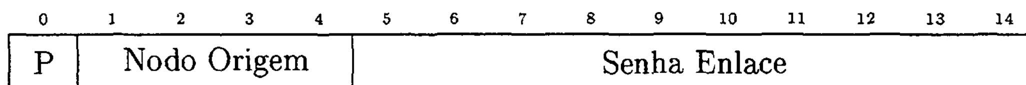

Figura 4.6: Pacote a ser usado na detecção de recuperação de enlaces.

(0): P, informa que é um teste de enlace (**Ping**);

(1-4): Identificador do nodo que enviou a mensagem;

(5-14): Senha do enlace;

O pacote da Figura 4.7 só será usado quando um enlace outrora falho se recuperar, como confirmação ao pacote da Figura 4.6.

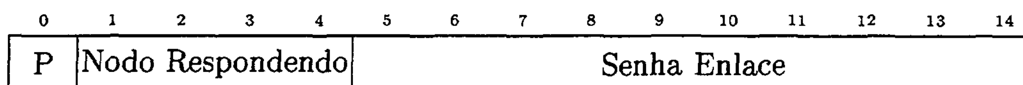

Figura 4.7: Confirmação de recebimento de pacote - enlace sem falhas.

(0): P, indica resposta ao teste (**Ping**);

(1-4): Identificador do nodo que está respondendo ao teste;

(5-14): Senha do enlace;

Esta resposta é enviada à porta (UDP) específica da difusão confiável, portanto, os nodos não confundem as duas mensagens.

4.7 Resultados Experimentais

Nesta seção são descritos resultados obtidos com a execução do algoritmo em redes de três topologias: anel, hypercubo e um grafo exemplo.

4.7.1 Anel com 4 Nodos

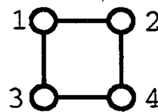

Figura 4.8: Um anel com 4 nodos.

Como primeiro exemplo, o sistema distribuído com a estrutura de enlaces da figura 4.8. O nodo 1 tem enlaces com os nodos 2 e 3, e ambos os nodos 2 e 3 tem enlaces com o nodo 4.

O nodo 1 precisa enviar uma mensagem. Como é o nodo que inicia a difusão, monta a árvore de busca em largura sendo sua raiz. A ávore

resultante é mostrada na figura 4.9: o nodo 1 com enlaces com os nodos 2 e 3, e o nodo 2 com enlace com o nodo 4.

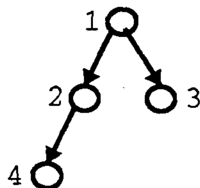

Figura 4.9: Árvore de disseminação.

Considerando o tempo inical T e que a execução do sistema em cada nodo é independente dos demais nodos, observe a tabela 4.1. Todos os passos foram executados na mesma unidade tempo.

Ação
nodo 1 monta a árvore de difusão (Figura 4.9)
nodo 1 envia mensagem para nodo 2
nodo 1 envia mensagem para nodo 3
nodo 2 envia confirmação de recebimento (Ack)
nodo 2 monta árvore de difusão com o nodo 1 como raiz. Será a mesma árvore da Figura 4.9
nodo 3 envia confirmação de recebimento (Ack)
nodo 3 monta árvore de difusão com o nodo 1 como raiz. Será a mesma árvore da Figura 4.9
nodo 2 tem apenas o nodo 4 como nodo filho. Envia mensagem para nodo 4
nodo 3 descobre que não tem nodos filhos, portanto, completa
nodo 4 envia confirmação de recebimento (Ack)
nodo 4 monta árvore de difusão com o nodo 1 como raiz. Será a mesma árvore da Figura 4.9
nodo 4 descobre que não tem nodos filhos, portanto, completa
nodo 2 consulta estado do nodo 4
nodo 4 responde estado: Concluído
nodo 2 completa pois todos os seus nodos filhos (4) completaram
nodo 1 consulta estado do nodo 2

nodo 1 consulta estado do nodo 3
nodo 2 responde estado: Concluído
nodo 3 responde estado: Concluído
nodo 1 completa pois todos os nodos filhos (2 e 3) completaram

Tabela 4.1: Execução da difusão confiável

4.7.2 Execução Com Ocorrência de Falhas

Considerando o mesmo sistema distribuído representado pela figura 4.8, o que aconteceria se o enlace entre os nodos 2 e 4 estivesse falho ? Segundo a árvore de disseminação da figura 4.9, nenhum outro nodo enviaria a mensagem para o nodo 4. Como o objetivo (e obrigação) de uma difusão confiável é de garantir que todos os nodos pertencentes a um sistema recebam a mensagem, os nodos executam os passos descritos na tabela 4.2. A coluna T mostra os tempos relativo ao início da execução ($T = 0$).

T	Raiz	Id	Ação
0	1	0	nodo 1 monta a árvore de difusão (Figura 4.9)
0	1	0	nodo 1 envia mensagem para nodo 2
0	1	0	nodo 1 envia mensagem para nodo 3
0	1	0	nodo 2 envia confirmação de recebimento (Ack)
0	1	0	nodo 2 monta árvore de difusão (Figura 4.9)
0	1	0	nodo 3 envia confirmação de recebimento (Ack)
0	1	0	nodo 3 monta árvore de difusão (Figura 4.9)
0	1	0	nodo 2 envia mensagem para nodo 4
0	1	0	nodo 3 descobre que não tem nodos filhos, portanto, completa
0	1	0	nodo 1 consulta estado do nodo 2
0	1	0	nodo 1 consulta estado do nodo 3
0	1	0	nodo 2 responde estado: Em Difusão

0	1	0	nodo 3 responde estado: Concluído
2	1	0	nodo 1 consulta estado do nodo 2
2	1	0	nodo 2 responde estado: Em Difusão
4	1	0	nodo 1 consulta estado do nodo 2
4	1	0	nodo 2 responde estado: Em Difusão
5	1	0	nodo 2 envia novamente a mensagem para nodo 4
6	1	0	nodo 1 consulta estado do nodo 2
6	1	0	nodo 2 responde estado: Em Difusão
8	1	0	nodo 1 consulta estado do nodo 2
8	1	0	nodo 2 responde estado: Em Difusão
10	1	0	nodo 2 envia novamente a mensagem para nodo 4
10	1	0	nodo 1 consulta estado do nodo 2
10	1	0	nodo 2 responde estado: Em Difusão
12	1	0	nodo 1 consulta estado do nodo 2
12	1	0	nodo 2 responde estado: Em Difusão
14	1	0	nodo 1 consulta estado do nodo 2
14	1	0	nodo 2 responde estado: Em Difusão
15	1	0	nodo 2 detecta falha no enlace com o nodo 4
15	2	0	nodo 2 monta a árvore de difusão da Figura 4.10
15	2	0	nodo 2 envia mensagem para nodo 1
			notificando a falha no enlace entre os nodos 2 e 4
15	2	0	nodo 1 envia confirmação de recebimento (Ack)
15	2	0	nodo 1 monta árvore de difusão (Figura 4.10)
15	2	0	nodo 1 envia mensagem para nodo 3
			notificando a falha no enlace entre os nodos 2 e 4
15	2	0	nodo 3 envia confirmação de recebimento (Ack)
15	2	0	nodo 3 monta árvore de difusão (Figura 4.10)
15	2	0	nodo 3 envia mensagem para nodo 4
			notificando a falha no enlace entre os nodos 2 e 4
15	2	0	nodo 4 envia confirmação de recebimento (Ack)
15	2	0	nodo 4 monta árvore de difusão (Figura 4.9)
15	2	0	nodo 4 descobre que não tem nodos filhos, portanto, completa
15	2	0	nodo 3 consulta estado do nodo 4
15	2	0	nodo 4 responde estado: Concluído
15	2	0	nodo 3 completa pois todos os seus nodos filhos (4) completaram
15	2	0	nodo 1 consulta estado do nodo 3
15	2	0	nodo 3 responde estado: Concluído
15	2	0	nodo 1 completa pois todos os seus nodos filhos (3) completaram
15	2	0	nodo 2 consulta estado do nodo 1
15	2	0	nodo 1 responde estado: Concluído
15	2	0	nodo 2 completa pois todos os nodos filhos (1) completaram

Tabela 4.2: Execução da difusão confiável com falhas.

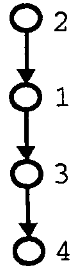

Figura 4.10: Nova Difusão com raiz no nodo 2

4.7.3 Hipercubo

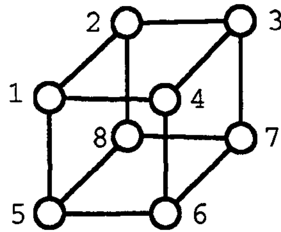

Figura 4.11: Hipercubo.

Neste exemplo de execução usaremos o sistema distribuído com a estrutura de enlaces da figura 4.11. O nodo 1 tem enlaces com os nodos 2, 4 e 5; o nodo 2 tem enlaces com os nodos 3 e 8; o nodo 3 tem enlaces com os nodos 4 e 7; o nodo 4 tem enlace com o nodo 6; o nodo 5 tem enlaces com os nodos 6 e 8; o nodo 6 tem enlace com o nodo 7; e o nodo 7 tem enlace com o nodo 8.

O nodo 4 precisa enviar uma mensagem. Como é o nodo que inicia a difusão, monta a árvore de busca em largura sendo sua raiz. A ávore resultante é mostrada na figura 4.12: o nodo 4 com enlaces com os nodos 1, 3 e 6; o nodo 1 com enlaces com os nodos 2 e 5, o nodo 2 com enlace com o nodo 8 e o nodo 3 com enlace com o nodo 7.

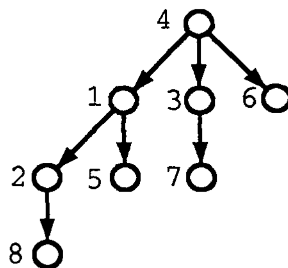

Figura 4.12: Árvore de disseminação com nodo 4 como raiz.

Os nodos executam os passos descritos na tabela 4.3, a seguir. considerando T como o tempo relativo ao início da execução ($T = 0$).

T	Raiz	Id	Ação
0	4	0	nodo 4 monta a árvore de difusão (figura 4.12)
0	4	0	nodo 4 envia mensagem para nodo 1
0	4	0	nodo 4 envia mensagem para nodo 3
0	4	0	nodo 4 envia mensagem para nodo 6
0	4	0	nodo 1 envia confirmação de recebimento (Ack)
0	4	0	nodo 1 monta árvore de difusão (figura 4.12)
0	4	0	nodo 3 envia confirmação de recebimento (Ack)
0	4	0	nodo 3 monta árvore de difusão (figura 4.12)
0	4	0	nodo 6 envia confirmação de recebimento (Ack)
0	4	0	nodo 6 monta árvore de difusão (figura 4.12)
0	4	0	nodo 6 completa por não ter filhos
0	4	0	nodo 1 envia mensagem para nodo 2
0	4	0	nodo 1 envia mensagem para nodo 5
0	4	0	nodo 3 envia mensagem para nodo 7

0	4	0	nodo 7 envia confirmação de recebimento (Ack)
0	4	0	nodo 7 monta árvore de difusão (figura 4.12)
0	4	0	nodo 7 completa por não ter filhos
0	4	0	nodo 3 consulta estado do nodo 7
0	4	0	nodo 7 responde estado: Concluído
0	4	0	nodo 3 completa pois todos seus nodos filhos (7) completaram
0	4	0	nodo 4 consulta estado do nodo 1
0	4	0	nodo 4 consulta estado do nodo 3
0	4	0	nodo 4 consulta estado do nodo 6
0	4	0	nodo 1 responde estado: Em Difusão
0	4	0	nodo 3 responde estado: Concluído
0	4	0	nodo 6 responde estado: Concluído
2	4	0	nodo 4 consulta estado do nodo 1
2	4	0	nodo 1 responde estado: Em Difusão
4	4	0	nodo 4 consulta estado do nodo 1
4	4	0	nodo 1 responde estado: Em Difusão
5	4	0	nodo 1 envia mensagem para nodo 2
5	4	0	nodo 1 envia mensagem para nodo 5
6	4	0	nodo 4 consulta estado do nodo 1
6	4	0	nodo 1 responde estado: Em Difusão
8	4	0	nodo 4 consulta estado do nodo 1
8	4	0	nodo 1 responde estado: Em Difusão
10	4	0	nodo 1 envia mensagem para nodo 2
10	4	0	nodo 1 envia mensagem para nodo 5
10	4	0	nodo 4 consulta estado do nodo 1
10	4	0	nodo 1 responde estado: Em Difusão
12	4	0	nodo 4 consulta estado do nodo 1
12	4	0	nodo 1 responde estado: Em Difusão
14	4	0	nodo 4 consulta estado do nodo 1
14	4	0	nodo 1 responde estado: Em Difusão
15	4	0	nodo 1 detecta falha nos enlaces com os nodos 2 e 5

Tabela 4.3: Execução da difusão confiável até a detectção das falhas.

No instante $T = 15$, o nodo 1 detecta falha em seus enlaces com os nodos 2 e 5. Portanto, interrompe a atual difusão confiável (raiz = 4, id = 0), criando uma nova difusão confiável sendo a própria raiz da árvore (figura

4.13). A execução do protocolo nos nodos, a partir da detecção destas falhas, é mostrada na tabela 4.4, a seguir.

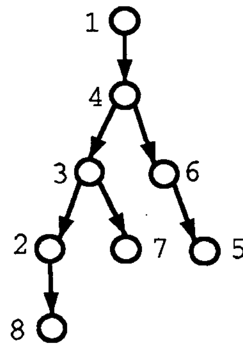

Figura 4.13: Árvore de disseminação com nodo 1 como raiz.

T	Raiz	Id	Ação
15	1	0	nodo 1 monta a árvore de difusão da Figura 4.13
15	1	0	nodo 1 envia mensagem para nodo 4
			notificando a falha nos enlaces $1 <-> 2$ e $1 <-> 5$
15	1	0	nodo 4 envia confirmação de recebimento (Ack)
15	1	0	nodo 4 cancela difusão anterior (raiz=4, id=0)
15	1	0	nodo 4 monta árvore de difusão (figura 4.13)
15	1	0	nodo 4 envia mensagem para nodo 1
15	1	0	nodo 4 envia mensagem para nodo 3
			notificando a falha nos enlaces $1 <-> 2$ e $1 <-> 5$
15	1	0	nodo 4 envia mensagem para nodo 6
			notificando a falha nos enlaces $1 <-> 2$ e $1 <-> 5$
15	1	0	nodo 3 envia confirmação de recebimento (Ack)
15	1	0	nodo 3 monta árvore de difusão (figura 4.13)
15	1	0	nodo 6 envia confirmação de recebimento (Ack)
15	1	0	nodo 6 monta árvore de difusão (figura 4.13)
15	1	0	nodo 3 envia mensagem para nodo 2
15	1	0	nodo 3 envia mensagem para nodo 7
15	1	0	nodo 6 envia mensagem para nodo 5
15	1	0	nodo 2 envia confirmação de recebimento (Ack)
15	1	0	nodo 2 monta árvore de difusão (figura 4.13)
15	1	0	nodo 7 envia confirmação de recebimento (Ack)
15	1	0	nodo 7 monta árvore de difusão (figura 4.13)

15	1	0	nodo 7 completa por não ter filhos
15	1	0	nodo 5 envia confirmação de recebimento (Ack)
15	1	0	nodo 5 monta árvore de difusão (figura 4.13)
15	1	0	nodo 5 completa por não ter filhos
15	1	0	nodo 2 envia mensagem para nodo 8
15	1	0	nodo 8 envia confirmação de recebimento (Ack)
15	1	0	nodo 8 monta árvore de difusão (figura 4.13)
15	1	0	nodo 8 completa por não ter filhos
15	1	0	nodo 2 consulta estado do nodo 8
15	1	0	nodo 8 responde estado: Concluído
15	1	0	nodo 2 completa pois todos seus nodos filhos (8) completaram
15	1	0	nodo 6 consulta estado do nodo 5
15	1	0	nodo 5 responde estado: Concluído
15	1	0	nodo 6 completa pois todos seus nodos filhos (5) completaram
15	1	0	nodo 3 consulta estado do nodo 2
15	1	0	nodo 3 consulta estado do nodo 7
15	1	0	nodo 2 responde estado: Em Difusão
15	1	0	nodo 7 responde estado: Concluído
15	1	0	nodo 4 consulta estado do nodo 3
15	1	0	nodo 4 consulta estado do nodo 6
15	1	0	nodo 3 responde estado: Em Difusão
15	1	0	nodo 6 responde estado: Concluído
15	1	0	nodo 1 consulta estado do nodo 4
15	1	0	nodo 4 responde estado: Em Difusão
17	1	0	nodo 3 consulta estado do nodo 2
17	1	0	nodo 2 responde estado: Concluído
17	1	0	nodo 3 completa pois todos seus nodos filhos (2 e 7) completaram
17	1	0	nodo 4 consulta estado do nodo 3
17	1	0	nodo 3 responde estado: Concluído
17	1	0	nodo 4 completa pois todos seus nodos filhos (3 e 6) completaram
17	1	0	nodo 1 consulta estado do nodo 4
17	1	0	nodo 4 responde estado: Concluído
17	1	0	nodo 1 completa pois todos seus nodos filhos (3) completaram

Tabela 4.4: Execução da difusão confiável após a detectção das falhas.

4.7.4 Grafo Aleatório

A maior parte dos algoritmos de difusão confiável assumem que o sistema distribuído possa ser representado por um grafo completo. Considere o caso prático de que os nodos deste sistema distribuído estão na Internet, podendo estar em *backbones* diferentes e com tecnologias de rede diferentes. Torna-se complicado (e caro) projetar a estrutura de enlaces físicos de forma a garantir essa representação do sistema como um grafo completo (enlace físico = enlace lógico).

O protocolo proposto neste trabalho executa corretamente em redes de topologia arbitrária. Este exemplo, baseado em um grafo aleatório (figura 4.14), demonstra que falhas em nodos só podem ser deduzidas quando todos os enlaces pertencentes aos nodos falhos forem testados. As falhas nos nodos não podem ser comprovadas. Portanto, se um nodo com N enlaces falha, então $(N + 1)$ difusões confiáveis são feitas, pois a partir da difusão inicial (1) ocorrem N detecções de falhas que disparam N novas difusões confiáveis. Uma otimização proposta é que, no projeto da topologia da rede para um sistema distribuído, evite-se criar nodos com muitos enlaces.

Na figura 4.14, O nodo 1 tem enlaces com os nodos 2, 3 e 4; o nodo 2 tem enlace com os nodos 5; o nodo 3 tem enlace com os nodos 5; e o nodo 4 tem enlace com o nodo 5.

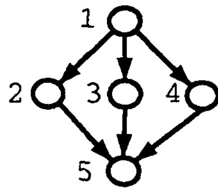

Figura 4.14: Grafo aleatório representando um sistema distribuído.

O nodo 1 precisa enviar uma mensagem. Como é o nodo que inicia a difusão, monta a árvore de busca em largura sendo sua raiz. A ávore resultante é mostrada na figura 4.15: o nodo 1 com enlaces com os nodos 2, 3 e 4, e o nodo 2 com enlaces com o nodo 5. Observe na tabela 4.5 o que acontece no caso do nodo 5 estar falho. Neste exemplo não será considerado o tempo de execução. Também não serão detalhados, novamente, todos os passos da difusão confiável.

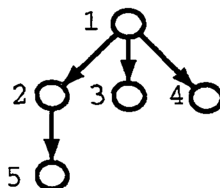

Figura 4.15: Árvore de disseminação com nodo 1 como raiz.

Raiz	Id	Ação
1	0	nodo 1 monta a árvore de difusão (figura 4.12)
1	0	nodo 1 envia mensagem para nodo 2
1	0	nodo 1 envia mensagem para nodo 3
1	0	nodo 1 envia mensagem para nodo 4
1	0	nodo 2 envia confirmação de recebimento (Ack)

1	0	nodo 2 monta árvore de difusão (figura 4.15)
1	0	nodo 3 envia confirmação de recebimento (Ack)
1	0	nodo 3 monta árvore de difusão (figura 4.15)
1	0	nodo 4 envia confirmação de recebimento (Ack)
1	0	nodo 4 monta árvore de difusão (figura 4.15)
1	0	nodo 3 completa por não ter filhos
1	0	nodo 4 completa por não ter filhos
1	0	nodo 2 envia mensagem para nodo 5
1	0	nodo 2 detecta falha no enlace com os nodos 2 e 5
2	0	nodo 2 monta a árvore de difusão (figura 4.16)
2	0	nodo 2 envia mensagem para nodo 1
		notificando da falha no enlace 2 < − > 5
2	0	nodo 1 envia confirmação de recebimento (Ack)
2	0	nodo 1 monta árvore de difusão (figura 4.16)
2	0	nodo 1 envia mensagem para nodo 3
		notificando da falha no enlace 2 < − > 5
2	0	nodo 1 envia mensagem para nodo 4
		notificando da falha no enlace 2 < − > 5
2	0	nodo 3 envia confirmação de recebimento (Ack)
2	0	nodo 3 monta árvore de difusão (figura 4.16)
2	0	nodo 4 envia confirmação de recebimento (Ack)
2	0	nodo 4 monta árvore de difusão (figura 4.16)
2	0	nodo 4 completa por não ter filhos
2	0	nodo 3 envia mensagem para nodo 5
		notificando da falha no enlace 2 < − > 5
2	0	nodo 3 detecta falha no enlace com os nodos 3 e 5
3	0	nodo 3 monta a árvore de difusão (figura 4.17)
3	0	nodo 3 envia mensagem para nodo 1
		notificando das falhas nos enlaces 2 < − > 5 e 3 < − > 5
3	0	nodo 1 envia confirmação de recebimento (Ack)
3	0	nodo 1 monta árvore de difusão (figura 4.17)
3	0	nodo 1 envia mensagem para nodo 2
		notificando das falhas nos enlaces 2 < − > 5 e 3 < − > 5
3	0	nodo 1 envia mensagem para nodo 4
		notificando das falhas nos enlaces 2 < − > 5 e 3 < − > 5
3	0	nodo 2 envia confirmação de recebimento (Ack)
3	0	nodo 2 monta árvore de difusão (figura 4.17)
3	0	nodo 4 envia confirmação de recebimento (Ack)
3	0	nodo 4 monta árvore de difusão (figura 4.17)
3	0	nodo 2 completa por não ter filhos
3	0	nodo 4 envia mensagem para nodo 5
		notificando das falhas nos enlaces 2 < − > 5 e 3 < − > 5
3	0	nodo 4 detecta falha no enlace com os nodos 4 e 5

4	0	nodo 4 monta a árvore de difusão (figura 4.18)
4	0	nodo 4 envia mensagem para nodo 1
		notificando das falhas nos enlaces 2 < − > 5, 3 < − > 5 e 4 < − > 5
4	0	nodo 1 envia confirmação de recebimento (Ack)
4	0	nodo 1 monta árvore de difusão (figura 4.18)
4	0	nodo 1 envia mensagem para nodo 2
		notificando das falhas nos enlaces 2 < − > 5, 3 < − > 5 e 4 < − > 5
4	0	nodo 1 envia mensagem para nodo 3
		notificando das falhas nos enlaces 2 < − > 5 e 3 < − > 5
4	0	nodo 2 envia confirmação de recebimento (Ack)
4	0	nodo 2 monta árvore de difusão (figura 4.17)
4	0	nodo 3 envia confirmação de recebimento (Ack)
4	0	nodo 3 monta árvore de difusão (figura 4.17)
4	0	nodo 2 completa por não ter filhos
4	0	nodo 3 completa por não ter filhos
4	0	nodo 1 completa pois todos seus nodos filhos (2 e 3) completaram
4	0	nodo 4 completa pois todos seus nodos filhos (1) completaram

Tabela 4.5: Execução da difusão confiável em um grafo aleatório.

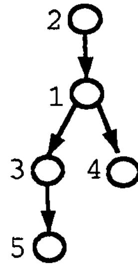

Figura 4.16: Árvore de disseminação com nodo 2 como raiz.

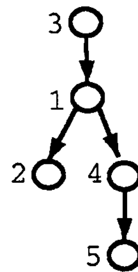

Figura 4.17: Árvore de disseminação com nodo 3 como raiz.

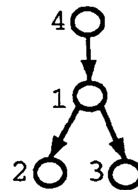

Figura 4.18: Árvore de disseminação com nodo 4 como raiz.

Capítulo 5

Conclusões

Neste trabalho foi apresentado um novo protocolo para difusão confiável em redes de topologia arbitrária que funciona corretamente na presença de múltiplos eventos, mesmo que estes eventos levem ao particionamento da rede. Foram definidas e implementadas todas as estruturas de dados e de controle, bem como o protocolo de comunicação entre nodos.

Este trabalho tem como principais contribuições:

- Solução de alguns problemas dos algoritmos anteriores, como atuar em redes de topologias arbitrárias e considerar particionamentos da rede;
- Difusão Confiável mesmo na presença de múltiplos eventos simultâneos;
- Consideração das falhas nos enlaces (invés de considerar nos nodos);
- Diagnóstico da rede durante as difusões confiáveis;

Entre os trabalhos futuros, encontra-se a transformação desta protocolo em um *Serviço de Difusão Confiável*, projetando, também, o protocolo de comunicação entre este serviço e seus aplicativos clientes.

Bibliografia

[1] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, 1994.

[2] S. Mullender, *Distributed Systems*, Addison-Wesley, 1993.

[3] F. B. Schneider, D. Gries, R. D. Schlicting, *Fault-Tolerant Broadcasts*, Science of Computer Programming, 4:1-15, 1984.

[4] E. P. Duarte Jr., T. Nanya, *A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm*, IEEE Transactions on Computers, Vol. 47, pp. 34-45,No. 1, Jan 1998.

[5] E.P. Duarte Jr., G. Mansfield, T. Nanya, and S. Noguchi, "Non-Broadcast Network Fault Monitoring Based on System-Level Diagnosis," *Proc. IEEE/IFIP IM'97*, pp.597-609, San Diego, May 1997.

[6] J. I. Siqueira, E. Fabris, E. P. Duarte Jr., "A Token Based Testing Strategy for Non-Broadcast Network Diagnosis", 1st IEEE Latin American Test Workshop, pp. 166-171, Rio de Janeiro, 2000.

[7] I. Group, "Threads extension for portable operating systems", IEEE P1003.4a Working Group (1992).. Threads extension for portable operating systems (draft), 1992.

[8] Frank Mueller, "A library implementation of POSIX threads under Unix", Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition, pp. 29-41, 1993

[9] D. E. Comer, "Internetworking with TCP/IP", Prentice Hall, 1995.

[10] J. Schönwälder, L. Furman, K. Zufelde, "Tcl Extensions for Network Management", http://www.ibr.cs.tu-bs.de/projects/scotty .

[11] J. Ousterhout, "Tool Command Language (TCL)", 1998, http://tcl.sourceforge.net .

[12] J. Case, M. Fedor, M. Schoffstall, and C. Davin, "Simple Network Management Protocol (SNMP)", RFC 1157, 1990.

Apêndice A

Código Fonte

Neste capítulo é apresentado o código fonte que implementa o protocolo de difusão confiável em redes de topologia arbitrária.

A.1 Makefile

Arquivo Makefile para a contrução do projeto:

```
DEBUG= -Wall
LIBS= -lpthread
OPTIONS= -D_REENTRANT
OBJECTS= classes.o main.o parser.o rede.o

all: srbXXXXXXXXX cliente

srbXXXXXXXXX: classes.o main.o parser.o rede.o
	g++ -o srbXXXXXXXXX $(OBJECTS) $(DEBUG) $(LIBS) $(OPTIONS)

main.o: main.cpp classes.h rede.h srb.h
	g++ -c main.cpp $(DEBUG) $(OPTIONS)

classes.o: classes.cpp classes.h srb.h
	g++ -c classes.cpp $(DEBUG) $(OPTIONS)

parser.o: parser.cpp parser.h classes.h
	g++ -c parser.cpp $(DEBUG) $(OPTIONS)

rede.o: rede.cpp rede.h classes.h
	g++ -c rede.cpp $(DEBUG) $(OPTIONS)

cliente: cliente.o
	gcc -o cliente cliente.o $(DEBUG)

cliente.o: cliente.c
	gcc -c cliente.c $(DEBUG)

# Utils
########

clean:
	rm -f *.o *~ srbXXXXXXXXX cliente

strip:
	strip srbXXXXXXXXX; strip cliente
```


A.2 srb.h

Este arquivo tem as definições de constantes globais:

```
#ifndef __SRB_H__

#define __SRB_H__

/* Includes */

#include <sys/types.h>


/* Defines */

/* Gerais */

#define PROTOCOLO_DEFAULT AF_INET
#define CHAVES 3
#define FREQUENCIA_MONITOR_RBL 60
#define FREQUENCIA_CONSULTA 2
#define TEMPO_LIMPEZA 180

#endif
```

A.3 parser.h

Este arquivo contém as constantes necessárias à etapa de processamento dos arquivos de configuração (parser):

```
#ifndef __PARSER_H__

#define __PARSER_H__

/* Portas */

#define BROADCAST_PORT 7890
#define SERVICE_PORT   7891
#define PING_PORT      7892
#define CLIENT_PORT    7893

/* Gerais */

#define MAX_NODES 1024
#define MAX_MESSAGE_SIZE 1000
#define TIMEOUT          5

/* Arquivo de configuracao */

#define CONF "/etc/srb/srb.conf"
#define NODOS "/etc/srb/nodos.conf"
#define ROTAS "/etc/srb/rotas.conf"
#define SENHAS "/etc/srb/passwd"

int parser(int,char *[]);

#endif
```

A.4 rede.h

Este arquivo contém as definições necessárias para a configuração dos sockets UDP e para o atendimento das requisições (do arquivo rede.cpp, listado posteriormente):

```
#ifndef __REDE_H__
#define __REDE_H__
struct portas {
  int broadcast;
  int servico;
  int ping;
};

int inicializa_rede(void);
void atende_rede (void);
#endif
```

A.5 classes.h

Este arquivo contém as definições das classes necessárias ao projeto:

nodo: Dados específicos de cada nodo

nodos: Vetor de objetos da classe *nodo*

nodo_list_nodo: Dados de um nodo pertencendo a uma difusão confiável

nodo_list: Lista de objetos da classe *nodo_list_nodo*. Normalmente usada para guardar a lista de filhos (e seus estados) durante a difusão confiável de um nodo

links: Guarda uma estrutura de enlaces. Muito utilizada para guardar a árvore de busca em largura utilizada na difusão confiável

srb: Todos os dados pertinentes a uma difusão confiável

nodo_rb: Guarda informações para um nodo na lista de difusões confiáveis

rb_list: Lista de difusões confiáveis

globais: Guarda variáveis globais

log: Geração de logs

```
#ifndef __CLASSES_H__
#define __CLASSES_H__
/* Includes */
#include <pthread.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <fstream.h>
#include "srb.h"
/* Classes */
class nodo {
public:
  unsigned int nodoId;
  char name[40];
  int ipv4;
  struct in_addr ip;
  struct in6_addr ip6;
  unsigned short int bport;
  unsigned short int pport;
  int status;
  /* Construtores e Destrutores */
  nodo (unsigned int, char*, unsigned short int, unsigned short int);
  nodo (unsigned int, struct in_addr *, unsigned short int, unsigned short int);
  nodo (unsigned int, struct in6_addr *, unsigned short int, unsigned short int);
  nodo (nodo *);
  nodo (void);
  ~nodo (void);
  int vazio();
  void zerar();
  nodo & copia (nodo *);
};
```

```
class nodos {

public:

  nodo *node;
  unsigned int max_pos;
  unsigned int tamanho;

  nodos (void);
  nodos (unsigned int); // Tamanho
  nodos (nodos *);
  ~nodos(void);

  nodos& operator+ (nodo *);
  nodos& operator- (unsigned int);
  nodos& operator- (nodo *);
  void sem_down (void);
  void sem_up (void);
  void seta_semaforo(void);

private:
  pthread_mutex_t sem;
  int y_sem;
};

class nodo_list_nodo {

public:

  enum estado {
    NLN_QUALQUER,
    NLN_ZERO,
    NLN_INICIO,
    NLN_MONITORANDO,  // vou monitorar este nodo
    NLN_MONITORADO,  // consegui monitorar este nodo
    NLN_ENCERRADO
  };

  unsigned int nodoId;
  char password[11];
  int status;
  nodo_list_nodo *prox;

  nodo_list_nodo(void);
  nodo_list_nodo(unsigned int, char *);
  ~nodo_list_nodo(void);

};

class nodo_list {

public:

  nodo_list_nodo* inicio;
  nodo_list_nodo* fim;

  /* Construtores e Destrutores */

  nodo_list (void);
  nodo_list (int);
  nodo_list (nodo_list *);
  ~nodo_list(void);

  /* Funções */

  nodo_list& ins (unsigned int, char *);
  int del (unsigned int);
  nodo_list& copia (nodo_list *);
  int pega_senha (unsigned int, char *);
  int confere_senha (unsigned int, char *);
  int set_status (unsigned int, int);

  int nfilhos (int cond = nodo_list_nodo::NLN_QUALQUER);
  void sem_down();
  void sem_up();

private:
  pthread_mutex_t sem;
  int y_sem;
};
```



```
class links {

public:

  enum {
    DEL_ONE,
    DEL_BOTH
  };

  nodo_list *nodoList;
  unsigned int tamanho;
  unsigned int max_pos;

  links(int);
  links(int,unsigned int);
  links(int,links *);
  ~links(void);

  void ins (unsigned int, unsigned int, char *);
  void del (unsigned int, unsigned int, int ambos = links::DEL_ONE);
  void destroy_links (unsigned int, unsigned int);
  int pega_senha (unsigned int, unsigned int, char *);
  int confere_senha (unsigned int, unsigned int, char *);
  void limpa (void);

  void sem_down (void);
  void sem_up (void);
  unsigned int nlinks (unsigned int);
  int layout(char *);
  void monta_arvore (unsigned int);

  friend ostream& operator<< (ostream &o,links *);

 private:
  pthread_mutex_t sem;
  int y_sem;
};

#define BASE_LOGS "/etc/srb/logs/"

class srb {

public:

  enum {
    INICIO,
    EM_DIFUSAO,
    DIFUNDIDO,
    MONITORANDO,
    CONCLUIDO,
    FALHAS,
    ABORTADO,
    RECUPERACAO,
    ERRO
  };

  unsigned int raiz; /* Iniciador do Reliable Broadcast */
  unsigned int rbIId;  /* Id do RLB naquele nodoId */

  links *arvore;

  unsigned char *mensagem;
  unsigned int s_mensagem;
  unsigned char *dados;

  nodos *l_nodos;

  int status;
  char confirmacao;
  int sock;

  pthread_t thread;
  time_t inicio;
  time_t ultima_atualizacao;
  time_t ultima_consulta;

  /* Construtores e Destrutores */

  /* Atendendo Difusão */
  srb (unsigned int, unsigned int, links *,unsigned char *,unsigned int,
        struct sockaddr_in*,socklen_t);
```

```
  /* Iniciando Difusão */
  srb (links *,unsigned char *, unsigned int,struct sockaddr_in*,
        socklen_t, int conf=0);

  /* Nova Difusão depois de Eventos */
  srb (links *,srb *,nodo_list *,int);

  ~srb();

  /* Funções */

  srb& insert (nodo,nodo);

  void rependurar ();
  int responsavel (unsigned int, unsigned int);
  void ack(struct sockaddr_in *origem, socklen_t len);
  void nack(struct sockaddr_in *origem, socklen_t len);
  void difundir (void);
  void ficar_respondendo();
  int posso_limpar();
  void log(char *);

private:
  void monta_arvore (unsigned int);
  int processa_pacote(struct sockaddr_in *, socklen_t);

};

class nodo_rb {

public:
  srb *rb;
  nodo_rb *prox;

  nodo_rb(void);
  nodo_rb(srb *);
  ~nodo_rb(void);
};

class rb_list {

public:
  nodo_rb *inicio;
  nodo_rb *fim;
  unsigned int meuRBid;

  rb_list (void);
  ~rb_list (void);

  rb_list & ins (srb *);
  rb_list & del (srb *);
  rb_list & del (unsigned int,unsigned int);

  srb * responsavel (unsigned int, unsigned int, int);
  void limpeza();

  void sem_down();
  void sem_up();

  unsigned int next();
  unsigned int next2();

 private:

  pthread_mutex_t sem;
  int y_sem;
};

class globais {

public:
  unsigned int max_nodes;
  unsigned int max_message_size;

  unsigned int broadcast_port;
  unsigned int service_port;
  unsigned int ping_port;
  unsigned int client_port;
  unsigned int meuId;
  int timeout;

  globais (unsigned int, unsigned int, unsigned int, unsigned int, unsigned int,
```



```
        unsigned int,unsigned int, int);
  ~globais();

  void set (unsigned int, unsigned int, unsigned int, unsigned int,unsigned int,
        unsigned int, unsigned int, int);
};

class log {

 public:

  log ();
  ~log();

  log& form (char * ...);
  log& log::operator<< (links *);
  log& log::operator<< (char *);
  log& log::operator<< (int);

 private:
  ofstream *file;
  pthread_mutex_t sem;

};

#endif
```

A.6 classes.cpp

Este arquivo implementa as classes necessárias ao projeto (descritas no arquivo *classes.h*):

```
#include <pthread.h>
#include <iostream.h>
#include <string.h>
#include <sys/types.h>
#include <netdb.h>
#include <errno.h>
#include <unistd.h>
#include <strstream.h>
#include <sys/time.h>
#include <fstream.h>
#include <stdio.h>
#include <stdarg.h>

#include "srb.h"
#include "classes.h"

extern nodo_list filhos_mortos;
extern globais *variaveis_globais;
extern links *rotas;
extern rb_list rbl;
extern nodos *descricao_nodos;
extern log f_log;

void * novo_broadcast (void *novo_rb) {

  srb *rb;

  rb = (srb *) novo_rb;

  rb->thread = pthread_self();

  rbl.ins(rb);

  rb->difundir();

  rb->ficar_respondendo();
```

```
  return 0;
}

/* Classe log */

log::log () {

  file = 0;
  pthread_mutex_init (&sem,0);

}

log::~log() {

  pthread_mutex_lock(&sem);

  file->close();
  delete file;

  pthread_mutex_unlock(&sem);
  pthread_mutex_destroy(&sem);

}

log & log::form (char *format ...) {

  // pthread_mutex_lock (&sem);

  va_list ap;

  va_start(ap,format);

  if ( !file ) {

    char nome[50];
    sprintf(nome,"%slogs-%d",BASE_LOGS,variaveis_globais->meuId);
    file = new ofstream (nome);
    file->form("%d: ****** Inicialização ******\n\n",time(NULL));
  }

  file->form("%d: ",time(NULL));
  file->vform(format,ap);
  file->flush();

  va_end(ap);
  // pthread_mutex_unlock(&sem);

  return *this;
}

log& log::operator<< (links *l) {

  char saida[5000];

  l->layout(saida);
  (*this) << saida;
  return *this;

}

log& log::operator<< (char *mesg) {

  //  pthread_mutex_lock(&sem);

  if ( !file ) {

    char nome[50];
    sprintf(nome,"%slogs-%d",BASE_LOGS,variaveis_globais->meuId);
    file = new ofstream (nome);
    file->form("%d: ****** Inicialização ******\n\n",time(NULL));
  }

  (*file) << mesg;
  file->flush();

  // pthread_mutex_unlock(&sem);
  return *this;

}

log& log::operator<< (int var) {

  pthread_mutex_lock(&sem);
```



```
  if ( !file ) {

    char nome[60];
    sprintf(nome,"%slogs-%d",BASE_LOGS,variaveis_globais->meuId);
    file = new ofstream (nome);
    file->form("%d: ****** Inicialização ******\n\n",time(NULL));
  }

  (*file) << var;
  file->flush();

  pthread_mutex_unlock(&sem);
  return *this;

}

/* Classe nodo */

nodo::nodo (unsigned int id, char* nam, unsigned short int bpor,unsigned short int ppor) {

  nodoId = id;
  struct hostent *h;

  strncpy((char *)name,(char *)nam,40);
  char dots[40];

  ipv4=1;
  dots[0]=0;

  h = gethostbyname2 ((char *)name,PROTOCOLO_DEFAULT);

  if ( !h) {

    if (PROTOCOLO_DEFAULT == AF_INET) {
      h = gethostbyname2((char *)name,AF_INET6);
      ipv4=0;
    } else
      h = gethostbyname2((char *)name,AF_INET);

    if (!h) {
      f_log.form("Maquina %s não encontrada !!!\n",nam);
      name[0]=0;
      ipv4=1;
      ip.s_addr=0;
      return;
    }
  }

  if (ipv4) {
    bcopy (h->h_addr,&ip,sizeof(struct in_addr));
    inet_ntop (AF_INET,h->h_addr,dots,40);
  } else {
    ipv4=0;
    bcopy (h->h_addr,&ip6,sizeof(struct in6_addr));
    inet_ntop (AF_INET6,h->h_addr,dots,40);
  }

  bport=bpor;
  pport=ppor;

  f_log << "Novo nodo: Id: " << nodoId << " Nome: " << name << " IPV4: " << ipv4 << " IP: "
     << (char *)dots << " BPorta: " << bport << " PPorta: " << pport << "\n";
}

nodo::nodo (unsigned int id, struct in_addr *ipaddr, unsigned short int bpor,
   unsigned short int ppor ) {

  nodoId = id;
  name[0]=0;
  ipv4=1;
  bcopy(ipaddr,&ip,sizeof(struct in_addr));
  bport=bpor;
  pport=ppor;

}

nodo::nodo (unsigned int id, struct in6_addr *ipaddr, unsigned short int bpor,
  unsigned short int ppor ) {

  nodoId = id;
  name[0]=0;
```

```
  ipv4=0;
  bcopy(ipaddr,&ip6,sizeof(struct in6_addr));
  bport=bpor;
  pport=ppor;
}
nodo::nodo (nodo* n_pai) {
  nodoId = n_pai->nodoId;
  strcpy((char *)name,(char *)n_pai->name);
  ip = n_pai->ip;
  ip6 = n_pai->ip6;
  ipv4 = n_pai->ipv4;
  bport = n_pai->bport;
  pport = n_pai->pport;
}
nodo::nodo (void) {
  zerar();
}
nodo::~nodo () {
}
int nodo::vazio () {
  return (nodoId == 0);
}
nodo & nodo::copia (nodo *n_pai) {
  nodoId = n_pai->nodoId;
  strcpy((char *)name,(char *)n_pai->name);
  ip = n_pai->ip;
  ip6 = n_pai->ip6;
  bport = n_pai->bport;
  pport = n_pai->pport;
  return *this;
}
void nodo::zerar () {
  nodoId=0;
  name[0]=0;
  ip.s_addr=0;
  ipv4=1;
  bport=0;
  pport=0;
}
/* Classe nodos */
nodos::nodos () {
  tamanho = variaveis_globais->max_nodes;
  max_pos = 0;
  node = new nodo[variaveis_globais->max_nodes];
  y_sem = 0;
}
nodos::nodos(unsigned int tam) {
  tamanho = tam+1;
  max_pos = 0;
  node = new nodo[tamanho];
  y_sem = 0;
}
nodos::nodos(nodos *n_pai) {
  unsigned int i,j;
  y_sem = 0;
  max_pos = 0;
  n_pai->sem_down();
```



```
  j = n_pai->max_pos;
  tamanho = j;
  node = new nodo[j+1];
  for (i=0; i<= j; i++ ) {
    if (!n_pai->node[i].vazio()) {
      //      f_log.form("Copiando nodo %d\n",i);
      (node+i)->copia(&(n_pai->node[i]));
    }
  }
  n_pai->sem_up();
}
nodos::~nodos () {
  if (y_sem != 0) {
    pthread_mutex_unlock(&sem);
    pthread_mutex_destroy(&sem);
  }
  delete[] node;
}
nodos& nodos::operator+ (nodo *n) {
  sem_down();
  node[n->nodoId] = n;
  if (n->nodoId > max_pos)
    max_pos = n->nodoId;
  sem_up();
  return *this;
}
nodos& nodos::operator- (nodo *n) {
  sem_down();
  node[n->nodoId].zerar();
  sem_up();
  return *this;
}
nodos& nodos::operator- (unsigned int id) {
  sem_down();
  node[id].zerar();
  sem_up();
  return *this;
}
void nodos::sem_down () {
  // f_log.form("sem_down nodos\n");
  if (y_sem != 0)
    pthread_mutex_lock(&sem);
}
void nodos::sem_up () {
  // f_log.form("sem_up nodos\n");
  if (y_sem != 0)
    pthread_mutex_unlock(&sem);
}
void nodos::seta_semaforo () {
  y_sem = 1;
  pthread_mutex_init (&sem,0);
}
/* Classe nodo_list_nodo */
nodo_list_nodo::nodo_list_nodo () {
```

```
  nodoId=0;
  password[0]=0;
  prox=0;
  status = nodo_list_nodo::NLN_INICIO;
}
nodo_list_nodo::nodo_list_nodo (unsigned int id, char *senha) {

  //  f_log.form("Entrou na criação de nodo\n");
  nodoId=id;
  if (senha[0] == 0) {
    // f_log.form("Ué...\n");
    password[0]=0;
  } else {
    // f_log.form("optou pelo strncpy\n");
    strncpy(password,senha,10);
  }
  prox=0;
  status = nodo_list_nodo::NLN_INICIO;
  // f_log.form("Saiu da criação de nodo\n");
}
nodo_list_nodo::~nodo_list_nodo () {
}


/* Classe nodo_list */

nodo_list::nodo_list () {

  // f_log.form("Chamando criacao vazia\n");
  inicio = fim = new nodo_list_nodo;
  y_sem = 0;
  //    y_sem = 1;
  //  pthread_mutex_init(&sem,0);
}
nodo_list::nodo_list (int semaforo) {

  inicio = fim = new nodo_list_nodo;

  if (semaforo) {
    y_sem = 1;
    pthread_mutex_init(&sem,0);
  } else {
    y_sem = 0;
  }
}


nodo_list::nodo_list (nodo_list *nl_pai) {

  nodo_list_nodo *nln, *aux;

  inicio = fim = new nodo_list_nodo;

  y_sem = 0;

  nl_pai->sem_down();

  for (aux=nl_pai->inicio->prox; !aux; aux=aux->prox) {

    nln = new nodo_list_nodo(aux->nodoId,aux->password);
    nln->status = nodo_list_nodo::NLN_INICIO;
    fim->prox = nln;
    fim = nln;
  }

  nl_pai->sem_up();

  fim->prox=0;
}
nodo_list::~nodo_list () {

    nodo_list_nodo *nln, *aux;

    //    cerr << "~nodo list\n";

    for (nln=inicio; nln->prox != 0; delete aux) {
      aux=nln;
```



```
      //cerr << "nodo: " << aux->nodoId << "\n";
      nln = nln->prox;
    }

    if (y_sem != 0 ) {
      pthread_mutex_unlock(&sem);
      pthread_mutex_destroy(&sem);
    }

    if (nln != 0) {
      //cerr << "nln != 0: " << nln->nodoId<< " \n";
      delete nln;
    }
}
int nodo_list::nfilhos (int cond) {

  nodo_list_nodo *nln;
  unsigned int i=0;

  sem_down();

  for (nln = inicio->prox; nln != 0; nln = nln->prox ) {
    if (nln->nodoId != 0)
      if (cond == nodo_list_nodo::NLN_QUALQUER || nln->status == cond )
i++;
  }

  sem_up();
  return i;
}
nodo_list & nodo_list::ins (unsigned int id, char *senha) {

  nodo_list_nodo *nln;

  if (id == 0) {
    return *this;
  }

  /* Primeiro, procura se este nodo já não está incluso */

  sem_down();

  for (nln = inicio; nln != 0; nln= nln->prox) {
    if (nln->nodoId == id) {
      sem_up();
      return *this;
    }
  }

  nln = new nodo_list_nodo(id,senha);

  fim->prox=nln;
  fim=nln;

  nln->prox = 0;
  sem_up();
  return *this;
}
int nodo_list::del (unsigned int id) {

  nodo_list_nodo *nln, *aux;

  if (id==0) {
    return 0;
  }

  sem_down();

  for (nln = inicio, aux=inicio; nln != 0; nln= nln->prox) {
    if (nln->nodoId == id) {
      aux->prox = nln->prox;

      if (fim == nln) {
fim = aux;
      }

      sem_up();
      delete nln;
      return 1;

    } else {
```

```
      aux=nln;
    }
  }
  sem_up();
  return 0;
}

int nodo_list::set_status (unsigned int id, int stat) {
  nodo_list_nodo *nln;

  if (id==0) {
    return 0;
  }

  sem_down();

  for (nln = inicio; nln != 0; nln= nln->prox) {
    if (nln->nodoId == id) {
      nln->status = stat;
      sem_up();
      return 1;
    }
  }

  sem_up();
  return 0;
}

nodo_list & nodo_list::copia (nodo_list *l_pai) {
  nodo_list_nodo *nln, *aux,*p;

  sem_down();

  //f_log.form("inicializacao\n");
  for (nln=inicio->prox; nln != 0; delete aux) {

    aux=nln;
    nln = nln->prox;
  }

  //f_log.form("Ok\n");

  inicio->prox = 0;
  fim = inicio;

  for (nln = inicio, aux=l_pai->inicio; aux->prox != 0; aux=aux->prox) {
    //f_log.form("no nodo pai: %d, %s, %p\n",aux->prox->nodoId,aux->prox->password,nln);
    p = new nodo_list_nodo (aux->prox->nodoId, aux->prox->password);
    //f_log.form("P. pegou a resposta\n");
    //f_log.form("P: %d\n",p->nodoId);
    nln->prox = p;
    //f_log.form("N: %d\n",nln->prox->nodoId);
    //f_log.form("passou\n");
    nln= nln->prox;
    nln->status = nodo_list_nodo::NLN_INICIO;
  }

  //  f_log.form("sai ileso do for\n");
  nln->prox = 0;
  sem_up();
  return *this;
}

void nodo_list::sem_down () {
  //f_log.form("sem_down nodo_list\n");

  if (y_sem != 0)
    pthread_mutex_lock(&sem);
}

void nodo_list::sem_up () {
```



```
  //f_log.form("sem_up nodo_list\n");
  if (y_sem != 0)
    pthread_mutex_unlock(&sem);
}

int nodo_list::pega_senha (unsigned int id, char *senha) {
  nodo_list_nodo *nln;

  sem_down();

  for (nln = inicio->prox; nln !=0; nln=nln->prox) {
    if (nln->nodoId == id) {
      nln->password[10] = 0;
      strcpy(senha,nln->password);
      //      f_log.form("Senha indo é '%s'\n",senha);
      sem_up();
      return 1;
    }
  }

  sem_up();

  return 0;
}

int nodo_list::confere_senha (unsigned int id, char *senha) {
  nodo_list_nodo *nln;
  int i;

  sem_down();

  for (nln = inicio->prox; nln !=0; nln=nln->prox) {
    if (nln->nodoId == id) {
      i= strncmp(senha,nln->password,10);
      sem_up();
      if (i)
return 0;
      else
return 1;
    }
  }

  sem_up();

  return 0;
}

/* Classe srb */

// Atendendo Broadcast
srb::srb (unsigned int root, unsigned int id, links *l_pai,unsigned char *mesg, unsigned int s_mesg,
   struct sockaddr_in *origem, socklen_t len) {

  raiz = root;
  rbId = id;

  status = srb::INICIO;

  log("Criando novo srb para novo broadcast que chegou");

  //f_log << "Pai\n" << l_pai;

  arvore = new links(1,l_pai);

  //  f_log.form("Enlaces\n")  << arvore << "\n";

  l_nodos = new nodos (descricao_nodos);

  s_mensagem = (variaveis_globais->max_message_size < s_mesg ) ?
               variaveis_globais->max_message_size : s_mesg;

  mensagem = new unsigned char[s_mensagem+1];
  bcopy (mesg,mensagem, s_mensagem);

  inicio = ultima_atualizacao = ultima_consulta = time(NULL);

  thread = pthread_self();

  sock = socket (AF_INET,SOCK_DGRAM,0);

  processa_pacote(origem,len);
```

```
  // log ("\nProcessou o pacote");
}
// Atendendo Serviço
srb::srb (links *l_pai,unsigned char *mesg, unsigned int s_mesg, struct sockaddr_in *origem,
  socklen_t len, int conf) {

  unsigned int nodo;
  unsigned int aux;
  unsigned short int sint;

  raiz = variaveis_globais->meuId;
  rblId = rbl.next2();
  confirmacao = conf;

  // f_log << "Pai\n"<< l_pai;

  arvore = new links(1,l_pai);

  //  f_log.form("Enlaces\n")  << arvore << "\n";

  status = srb::INICIO;

  l_nodos = new nodos (descricao_nodos);

  s_mensagem = (variaveis_globais->max_message_size < (s_mesg+24+3)) ?
         variaveis_globais->max_message_size : s_mesg + 24+3;

  mensagem = new unsigned char[s_mensagem+1];

  mensagem[15]=confirmacao;

  nodo = htonl(raiz);

  bcopy(&nodo,mensagem+16,4);

  aux = htonl(rblId);
  bcopy (&aux,mensagem+20,4);

  dados = mensagem + 24;

  mensagem[24]='M';
  sint = htons(s_mesg);
  bcopy(&sint,mensagem+25,2);
  bcopy (mesg,mensagem+27, s_mesg);

  inicio = ultima_atualizacao = ultima_consulta = time(NULL);

  thread = pthread_self();

  sock = socket (AF_INET,SOCK_DGRAM,0);

  mensagem[0]='A';
  sendto(sock,mensagem,1,0,(struct sockaddr *)origem,len);

  mensagem[0]='B';
}
srb::srb (links *l_pai,srb *s_pai,nodo_list *nl, int status_nodo) {

  int aux;
  nodo_list_nodo *nln;
  nodo *n;
  int cab_pai;
  char texto[200];
  int vizinhos[100];
  int nvizinhos;

  raiz = variaveis_globais->meuId;

  rblId = rbl.next2();

  confirmacao = s_pai->confirmacao;

  status = srb::INICIO;

  sock = socket (AF_INET,SOCK_DGRAM,0);

  inicio = ultima_atualizacao = ultima_consulta = time(NULL);

  //log ("depois das datas");
  mensagem = new unsigned char[variaveis_globais->max_message_size];
```



```
  cab_pai = ((s_pai->dados) - (s_pai->mensagem));

  //sprintf(texto,"Cab_pai: %d",cab_pai);
  //log(texto);
  bcopy (s_pai->mensagem, mensagem,cab_pai);

  aux = htonl(raiz);

  bcopy(&aux,mensagem+16,4);

  aux = htonl(rblId);
  bcopy (&aux,mensagem+20,4);

  //  log ("Antes da criaçao dos l_nodos");
  l_nodos = new nodos (descricao_nodos);

  dados = mensagem + cab_pai;

  /* Insere Lista de Broadcasts Anteriores */

  *dados++ = 'B';
  aux = htonl(s_pai->raiz);
  bcopy(&aux,dados,4);
  dados +=4;
  aux = htonl(s_pai->rblId);
  bcopy(&aux,dados,4);
  dados +=4;

  nvizinhos = 0;

  if ( nl->nfilhos(status_nodo) > 0 ) {

    sprintf(texto,"filhos em %d: %d",status_nodo,nl->nfilhos(status_nodo));
    log(texto);

    for (nln = nl->inicio->prox; nln != 0; nln = nln->prox ) {
      log("dentro do for");

      if (nln->nodoId != 0) {

if ( nln->status != status_nodo ) {
  sprintf(texto,"status veio %d inves de %d",nln->status,status_nodo);
  log(texto);
  continue;
}

//log("antes da atrib");
n = & (l_nodos->node[nln->nodoId]);

// log("antes antes");
sprintf(texto,"antes do nodoid %d",n->nodoId);
log(texto);
if (n->nodoId != 0) {

  sprintf(texto,"Falhou o link com o nodo: %d",n->nodoId);
  log(texto);

  vizinhos[nvizinhos] = n->nodoId;
  nvizinhos++;

  //log("consegui deletar");
  (*dados) = 'E';
  dados++;
  aux = htonl(raiz);
  bcopy (&aux,dados,4);
  //log("primeiro bcopy");
  dados += 4;

  aux = htonl(n->nodoId);
  bcopy (&aux,dados,4);
  dados += 4;
  //log("segundo bcopy");
  (*dados) = 'D';
  dados++;
  //if (nln->prox != 0 ) {
    //log("Tem proximo");
  //}
} else {
  //log("Nodo era vazio");
}
```

```
        } else {
//log("NODAO era vazio");
        }

        //log ("dando a volta");
      }

      //log("fora do for");

      for (aux = 0; aux < nvizinhos; aux++ ) {

        f_log.form("Deletando: %d -> %d\n",raiz,vizinhos[aux]);
        l_pai->del (raiz,vizinhos[aux],links::DEL_BOTH);

      }

      //log("la em baixo");
      bcopy (s_pai->dados, dados, (s_pai->s_mensagem) - cab_pai);
      s_mensagem = (dados - mensagem) + (s_pai->s_mensagem - cab_pai);

      // f_log << "Pai2\n"<< l_pai;

      //f_log.form("criando nova arvore\n") << l_pai;

      arvore = new links(1,l_pai);
      //log("Criando novos enlaces");
      f_log.form("Enlaces Novo Broadcast\n")  << arvore << "\n";

    } else {
      status = srb::ERRO;
    }

}
srb::~srb () {

  delete arvore;
  delete mensagem;
  delete l_nodos;
  close(sock);

}

void srb::monta_arvore (unsigned int root) {

  arvore->monta_arvore(root);

}

int srb::processa_pacote (struct sockaddr_in *origem, socklen_t len) {

  unsigned char *pos, *fim;
  unsigned int nodo1, nodo2, id;
  char senha[10];
  srb *rb;
  char texto[3000];
  unsigned short int aux;

  fim = mensagem + s_mensagem;

  confirmacao = mensagem[15];
  pos = mensagem + 24;

  while (pos != fim) {

    switch (*pos) {

    case 'B':
      if (!confirmacao) {
// abortar broadcasts

bcopy (pos+1,&nodo1,4);
nodo1 = ntohl (nodo1);

bcopy (pos+5,&id,4);
id = ntohl (id);

sprintf(texto,"Broadcast: raiz %d, id %d, pos: %d",nodo1,id,pos-mensagem);
log(texto);

if ( (rb = rbl.responsavel (nodo1,id,1)) != 0 ) {
  rbl.del(nodo1,id);
}

      }
```



```
        pos += 9;
        break;

      case 'E':
        f_log.form("E: pos: %d\n",pos-mensagem);
        pos++;
        bcopy(pos,&nodo1,4);
        nodo1 = ntohl (nodo1);
        pos+=4;
        bcopy(pos,&nodo2,4);
        nodo2 = ntohl (nodo2);
        pos+=4;
        if ( (*pos) == 'U' ) {

if ( nodo1 == variaveis_globais->meuId ) {
  filhos_mortos.pega_senha (nodo2,senha);
} else if ( nodo2 == variaveis_globais->meuId ) {
  filhos_mortos.pega_senha (nodo1,senha);
} else {
  senha[0]=0;
}
rotas->ins (nodo1,nodo2,senha);

        } else if ( (*pos) == 'D') {

rotas->pega_senha (nodo1,nodo2,senha);

if (nodo1 ==variaveis_globais->meuId ) {

  filhos_mortos.ins (nodo2,senha);

} else if (nodo2 ==variaveis_globais->meuId ) {

  filhos_mortos.ins (nodo1,senha);

}

rotas->del (nodo1,nodo2,links::DEL_BOTH);
arvore->del (nodo1,nodo2,links::DEL_BOTH);
        } else {
goto ERROR;
        }
        sprintf(texto,"Evento entre '%d' e '%d': %c",nodo1,nodo2,*pos);
        log(texto);
        pos++;
        break;

      case 'M':
        dados = pos;
        bcopy (pos+1,&aux,2);
        aux = ntohs(aux);
        *(dados + 3 + aux) = 0;
        sprintf(texto,"Dados recebidos: '%s', tamanho: %d, pos: %d",dados+3,aux,pos-mensagem);
        //f_log.form("Dados recebidos: %s\n",dados);
        log(texto);
        goto OK;
        break;

      default:
        sprintf(texto,"Ué, não poderia cair aqui: %c, %d, %d",*pos,*pos,pos-mensagem);
        log(texto);
        goto ERROR;
      }

  }

  OK:
  ack (origem,len);
  //  pthread_cleanup_push(morra_srb,this); CHAVES
  return 1;

  ERROR:
  status = srb::ERRO;
  nack (origem,len);
  return 0;

}

void srb::difundir () {

  unsigned int meuId, nodo1, nodo2;
```

```
  struct sockaddr_in destino;
  //  struct sockaddr_in6 destino6;
  socklen_t len;
  nodo_list_nodo *nln;
  nodo_list *nl;
  nodo *n;
  unsigned char pacote[50];
  fd_set resposta;
  struct timeval tv, *tmv;
  int ret, tam;
  time_t tempo;
  int status_nodo;
  int timeout;
  int tentativas=0;
  int max_tentativas = 3;
  char texto[1000];

  log("Iniciando difusão");

  meuId = variaveis_globais->meuId;

  nl = &(arvore->nodoList[meuId]);

  monta_arvore(raiz);

  //  f_log.form("Arvore montada\n") << arvore << "\n";

  log("Arvore montada");
  f_log << arvore << "\n";

  //ret = arvore->layout(texto);
  //log (texto);

  /* Checa filhos mortos */

  len = sizeof(struct sockaddr_in);

  tmv = &tv;

  status = srb::EM_DIFUSAO;
  status_nodo = nodo_list_nodo::NLN_INICIO;

  ultima_atualizacao = time(NULL);
  timeout = variaveis_globais->timeout;

  do {

    tentativas++;

    // Agora transmite pra todos os filhos vivos

    if ( nl->nfilhos() == 0 ) {
      status = srb::CONCLUIDO;
      log("Concluido, sem filhos");
      break;
    }

    sprintf(texto,"Filhos em %d: %d",status_nodo,nl->nfilhos(status_nodo));
    log(texto);
    if ( nl->nfilhos(status_nodo) > 0 ) {

      for (nln = nl->inicio->prox; nln != 0; nln = nln->prox ) {

if (nln->nodoId != 0) {

  if ( nln->status != status_nodo )
    continue;

  n = & (l_nodos->node[nln->nodoId]);

  if (n->nodoId != 0) {

    if (n->ipv4 )  {

      nodo1 = htonl (meuId);
      nodo2 = htonl (n->nodoId);

      sprintf(texto,"Enviando pacote para nodo %d",n->nodoId);
      log(texto);
      arvore->pega_senha (meuId,nln->nodoId,(char *)pacote);

      bcopy (&nodo1,mensagem+1,4);
      bcopy (pacote,mensagem+5,10);
```



```
      destino.sin_family = AF_INET;
      destino.sin_port = htons(n->bport);
      destino.sin_addr = n->ip;

      //sprintf(texto,"e porta %d",n->bport);
      //log(texto);
      switch (nln->status) {

      case nodo_list_nodo::NLN_INICIO:
mensagem[0]='B';

if (sendto (sock,mensagem,s_mensagem,0,(struct sockaddr *) &destino,len) < 0) {
  sprintf(texto,"Inicio %s\n",strerror(errno));
  log(texto);
}
break;

      case nodo_list_nodo::NLN_MONITORADO:
nl->set_status(n->nodoId,nodo_list_nodo::NLN_MONITORANDO);

      case nodo_list_nodo::NLN_MONITORANDO:
mensagem[0]='G';

if ( sendto (sock,mensagem,15,0,(struct sockaddr *) &destino,len) < 0) {
  sprintf(texto,"Monitorando: %s\n",strerror(errno));
  log(texto);
}
break;
      }

    } else {
      cerr << "Nao implementado IPV6\n";

    }

  }
}
      }
    }

    FD_ZERO(&resposta);

    tempo = time(NULL);

    do {

      if ( nl->nfilhos(status_nodo) == 0 ) {
//log("Não vou enviar nada pois não há filhos");
break;
      }

      FD_SET(sock,&resposta);

      if (tmv != 0) {
tmv->tv_sec = timeout - (time(NULL) - tempo);

if (tmv->tv_sec <= 0)
  tmv->tv_sec = 1;

tmv->tv_usec = 0;
      }

      if ( (ret = select (sock+1,&resposta,NULL,NULL,tmv)) < 0) {
cerr << "Erro no select: " << strerror (errno) << '\n';
continue;
      }

      if (ret != 0) {
len = sizeof(destino);

tam = recvfrom(sock,pacote,49,0,(struct sockaddr *) &destino, &len);

if (tam < 5 )
  continue;

bcopy (pacote+1,&nodo2,4);
nodo2 = ntohl(nodo2);

switch (pacote[0]) {

  /*
  ** Checagem de filhos mortos suspensa por enquanto
case 'P':
```

```
   // link com nodo2 voltou
   if (filhos_mortos.confere_senha (nodo2,(char *)pacote+5)) {
     filhos_mortos.del(nodo2);
     nl->ins(nodo2,(char *)pacote+5);
     status = srb::RECUPERACAO;
     // precisa fazer novo broadcast
     goto RECUP;
   }
   break;
   */
 case 'A':
   sprintf(texto,"Ack de nodo %d",nodo2);
   log(texto);

   nl->set_status(nodo2,nodo_list_nodo::NLN_MONITORANDO);
   n = & (l_nodos->node[nodo2]);
   n->ipv4=1;
   n->ip = destino.sin_addr;
   n->bport = ntohs(destino.sin_port);
   break;
 case 'G':
   if (arvore->confere_senha(meuId,nodo2,(char *) pacote+5))  {

     sprintf(texto,"Get status de nodo %d",nodo2);
     log(texto);

     pacote[0]='S';
     nodo1 = htonl(meuId);
     bcopy (&nodo1,pacote+1,4);
     pacote[15]=status;
     ultima_consulta = time(NULL);

     sendto (sock,pacote,16,0,(struct sockaddr *) &destino,len);

   } else {
     sprintf(texto,"Senha inválida no 'G': meuid: %d, nodo2: %d,%s",meuId,nodo2,pacote+5);
     log(texto);
   }

   break;
 case 'S':
   if (arvore->confere_senha(meuId,nodo2,(char *) pacote+5))  {

     if (pacote[15] == srb::CONCLUIDO || pacote[15] == srb::ERRO ||
                  pacote[15] == srb::ABORTADO || pacote[15] == srb::FALHAS) {
       nl->set_status(nodo2,nodo_list_nodo::NLN_ENCERRADO);
       sprintf(texto,"Status de '%d': Concluido",nodo2);

     } else {
       nl->set_status(nodo2,nodo_list_nodo::NLN_MONITORADO);
       sprintf(texto,"Status de '%d'",nodo2);

     }
     log(texto);
   } else {
     log("Senha inválida no 'S'");
   }

   break;
 default:
   cerr << variaveis_globais->meuId << ": Pacote veio com codigo " << pacote[0] << '\n';
 }

       }

     } while ( (time(NULL) - tempo) < (timeout) );

     if (tentativas >= max_tentativas && status == srb::MONITORANDO) {
       if (nl->nfilhos(nodo_list_nodo::NLN_MONITORANDO) == 0) {
```



```
// Todos os filhos responderam, então
for (nln = nl->inicio->prox; nln != 0; nln = nln->prox ) {
  if (nln->nodoId != 0) {
    if ( nln->status != nodo_list_nodo::NLN_MONITORADO )
      continue;

    n = & (l_nodos->node[nln->nodoId]);

    if (n->nodoId != 0) {

      nl->set_status(n->nodoId,nodo_list_nodo::NLN_MONITORANDO);
    }

  }
}
tentativas = 0;
sleep(FREQUENCIA_CONSULTA);
      } else {
log("Abortando...");
status = srb::FALHAS;
      }

    }

    if ( status == srb::EM_DIFUSAO && nl->nfilhos(nodo_list_nodo::NLN_INICIO) == 0 ) {
      status = srb::MONITORANDO;
      //      timeout=1;
      tentativas=0;
      ultima_atualizacao = time(NULL);
      status_nodo = nodo_list_nodo::NLN_MONITORANDO;
      timeout=variaveis_globais->timeout;
    }

    if ( status == srb::MONITORANDO && nl->nfilhos(nodo_list_nodo::NLN_ENCERRADO) == nl->nfilhos() ) {
      status = srb::CONCLUIDO;
      ultima_atualizacao = time(NULL);
      break;
    }


  }
  while ( status != srb::CONCLUIDO && status != srb::ABORTADO &&
          status != srb::FALHAS && tentativas < max_tentativas);

  // Já que parei tudo, posso sair da lista
  rbl.del(this);

  if ( status == srb::CONCLUIDO ) {
    log("Concluido!!!");
    return;
  }

  if ( status == srb::FALHAS || status == srb::EM_DIFUSAO) {
    // ué, teve filho que nao respondeu
    log("Teve filho que não respondeu");

    status = srb::CONCLUIDO;
    srb *novo_rb;

    // Para evitar que a limpeza ocorra enquanto estou gerando novo broadcast
    rbl.sem_down();

    novo_rb = new srb(rotas, this, nl,status_nodo);
    if ( novo_rb->status == srb::ERRO ) {
      log ("Erro na criação do novo broadcast");
      delete novo_rb;
    } else {
      pthread_t th;

      log("Disparando thread");
      pthread_create(&th,0,novo_broadcast,novo_rb);
      pthread_detach(th);

    }

    // Para evitar que a limpeza ocorra enquanto estou gerando novo broadcast
```

```
    rbl.sem_up();
  }
  if ( status == srb::ABORTADO )
    log ("Fui abortado");

  return;
}
void srb::ficar_respondendo() {

  int tam, nodo1,nodo2;
  socklen_t len;
  char pacote[200];
  struct sockaddr_in destino;
  fd_set fd;
  struct timeval tv;

  /* if (status == srb::ABORTADO ) {

    log("Não vou ficar respondendo pois estou abortando");
    return;
  }
  */

  // status = srb::CONCLUIDO;
  log("Ficar respondendo");

  FD_ZERO(&fd);

  while (1) {
    FD_SET (sock,&fd);
    tv.tv_sec = 30;
    tv.tv_usec = 0;

    len = sizeof(destino);

    if ( select (sock+1,&fd,NULL,NULL,&tv) == 0 ) {

      if ( (ultima_consulta - ultima_atualizacao) > 30  ) {
log ("Pai já sabe que tô bem => Caindo fora");
return;
      }

      if ( (time(NULL) - ultima_consulta) > 300) {
log ("Pai morreu!!!!!!!!!!!");
return;
      }
    }

    tam = recvfrom(sock,pacote,49,0,(struct sockaddr *) &destino, &len);

    log("Recebi pergunta de Status");
    if (tam < 5 )
      continue;

    bcopy (pacote+1,&nodo2,4);
    nodo2 = ntohl(nodo2);

    if (pacote[0] == 'G') {

      if (arvore->confere_senha(variaveis_globais->meuId,nodo2,(char *) pacote+5))  {

pacote[0]='S';
nodo1 = htonl(variaveis_globais->meuId);
bcopy (&nodo1,pacote+1,4);
pacote[15]=status;
ultima_consulta = time(NULL);

sendto (sock,pacote,16,0,(struct sockaddr *) &destino,len);

sprintf(pacote+60,"Respondendo status para nodo: %d",nodo2);
log(pacote+60);
      } else {
log ("Ué, senha não conferiu para responder status");
      }
    }

  }
}

int srb::posso_limpar() {

  if ( status == srb::ABORTADO )
```



```
    return 1;

  return ( (status == srb::CONCLUIDO || status == srb::ABORTADO ) &&
           ( (time(NULL) - ultima_atualizacao) > TEMPO_LIMPEZA));
}

void srb::log (char *mesg) {

  /*  char nome[50];

  sprintf(nome,"%slogs-%d",BASE_LOGS,variaveis_globais->meuId);

  ofstream of(nome,ios::app);
  */

  f_log.form ("Status: %d, Raiz: %d, Id: %d => %s\n", status, raiz, rblId, mesg);

}
int srb::responsavel (unsigned int root, unsigned int id) {

  return (root==raiz && rblId == id);

}

void srb::ack(struct sockaddr_in *origem, socklen_t len) {

  unsigned int nodoid;
  char texto[200];

  nodoid = htonl (variaveis_globais->meuId);
  mensagem[0] = 'A';
  bcopy(&nodoid,mensagem+1,4);

  log("Enviando ack");

  if (sendto (sock,mensagem,24,0,(struct sockaddr *) origem,len) < 0) {
    sprintf(texto,"Enviando ack: %s",strerror(errno));
    log(texto);
  }

}
void srb::nack(struct sockaddr_in *origem, socklen_t len) {

  unsigned int nodoid;
  char texto[200];

  nodoid = htonl (variaveis_globais->meuId);
  mensagem[0] = 'N';
  bcopy(&nodoid,mensagem+1,4);
  sprintf(texto,"Enviando ack: %d",sendto (sock,mensagem,24,0,(struct sockaddr *) origem,len));
  log(texto);

}

/* Classe links */


links::links (int usar_semaforo) {

  nodoList = new nodo_list[variaveis_globais->max_nodes];
  max_pos = 0;
  tamanho = variaveis_globais->max_nodes;

  // aqui   if (usar_semaforo >= 0) {
  if (1) {
    y_sem = 1;
    pthread_mutex_init(&sem,0);
  }
  else {
    y_sem = 0;
  }
}

links::links (int usar_semaforo, unsigned int tam) {

  nodoList = new nodo_list[tam];
  tamanho = tam;
  max_pos = 0;

  // aqui  if (usar_semaforo >= 0) {
  if (1) {
    y_sem = 1;
```

```
    pthread_mutex_init(&sem,0);
  }
  else {
    y_sem = 0;
  }
}
links::links (int usar_semaforo, links *l_pai) {

  nodo_list *nl;
  unsigned int i;

  // aqui if (usar_semaforo >= 0) {
  if (1) {
    y_sem = 1;
    pthread_mutex_init(&sem,0);
  }
  else {
    y_sem = 0;
  }

  max_pos = 0;

  l_pai->sem_down();

  //f_log.form ("criando nodo list\n");
  nodoList = new nodo_list[l_pai->max_pos+1];

  nl = nodoList;
  nl++;

  // f_log.form ("antes do for\n");
  for (i=1; i<= l_pai->max_pos; i++) {

    //    f_log.form ("Rodada %d\n",i);
    nl->copia(&l_pai->nodoList[i]);
    nl++;
  }

  //f_log.form ("limpo\n");
  tamanho = l_pai->tamanho;
  max_pos = l_pai->max_pos;

  l_pai->sem_up();
}
links::~links () {

  if (y_sem != 0) {
    pthread_mutex_unlock(&sem);
    pthread_mutex_destroy(&sem);
  }

  delete[] nodoList;
}
ostream& operator<< (ostream &s, links *l) {

  //nodo_list *nl;
  //nodo_list_nodo *nln;
  //  unsigned int i;
  char saida[5000];

  /*  l->sem_down();
      for (i=1; i <= l->max_pos; i++ ) {
      nl = (l->nodoList) + i;

      s << "Nodo: " << i << ": ";

      for (nln = nl->inicio->prox; nln != 0; nln = nln->prox) {
      s << nln->nodoId << " ";
      }

      s << '\n';

      }

      l->sem_up();
  */
```



```
  l->layout(saida);
  s << saida;
  return s;
}
int links::layout (char *saida) {

  nodo_list *nl;
  nodo_list_nodo *nln;
  unsigned int i;
  char *pos;

  pos = saida;

  pos += sprintf(pos,"\n");
  sem_down();

  for (i=1; i <= max_pos; i++ ) {
    nl = nodoList + i;

    pos += sprintf(pos,"\tNodo %d: ",i);
    //    s << "Nodo " << i << ": ";

    for (nln = nl->inicio->prox; nln != 0; nln = nln->prox) {
      pos += sprintf (pos,"%d ",nln->nodoId);
      //      s << nln->nodoId << " ";
    }

    pos += sprintf(pos,"\n");
    //    s << '\n';
  }

  sem_up();

  return (pos - saida);
}
void links::ins (unsigned int nodo1, unsigned int nodo2, char *senha) {

  sem_down();

  nodoList[nodo1].ins(nodo2,senha);
  nodoList[nodo2].ins(nodo1,senha);

  if (nodo1 > max_pos)
    max_pos = nodo1;

  if (nodo2 > max_pos)
    max_pos = nodo2;

  sem_up();
}
void links::del (unsigned int nodo1, unsigned int nodo2, int ambos) {

  if (nodo1 == 0 || nodo2 == 0) {
    f_log.form("Limpando coisa que não devia\n");
    return;
  }

  sem_down();

  nodoList[nodo1].del(nodo2);

  if (ambos == links::DEL_BOTH)
    nodoList[nodo2].del(nodo1);

  sem_up();
}
void links::destroy_links (unsigned int nodo1, unsigned int nodo2) {

  unsigned int i;

  sem_down();

  // Limpa link retorno
  //f_log.form("Limpando: %d -> %d\n",nodo1,nodo2);
  nodoList[nodo1].del(nodo2);

  // Limpa todos os links associados ao nodo1 (já que ele já foi pego)
  for (i = 1; i <= max_pos; i++) {
```

```
    if (i != nodo1 && i != nodo2) {
      // f_log.form("Removendo: %d -> %d\n",i,nodo1);
      nodoList[i].del(nodo1);
    }
  }

  sem_up();
}
void links::limpa () {
  sem_down();
  delete nodoList;
  nodoList = new nodo_list[variaveis_globais->max_nodes];
  max_pos = 0;
  sem_up();
}
void links::sem_down () {
  if (y_sem != 0) {
    //f_log.form ("links sem down\n");
    pthread_mutex_lock(&sem);
  }
}
void links::sem_up () {
  if (y_sem != 0 ) {
    //f_log.form ("links sem up\n");
    pthread_mutex_unlock(&sem);
  }
}
unsigned int links::nlinks (unsigned int id) {
  unsigned int i;
  sem_down();
  i = nodoList[id].nfilhos();
  sem_up();
  return i;
}
int links::pega_senha (unsigned int nodo1, unsigned int nodo2, char *senha) {
  int i;
  if (nodo1 > max_pos || nodo2 > max_pos ) {
    return 0;
  }
  sem_down();
  i= nodoList[nodo1].pega_senha(nodo2, senha);
  sem_up();
  return i;
}
int links::confere_senha (unsigned int nodo1, unsigned int nodo2, char *senha) {
  char pass[20];
  if (!pega_senha (nodo1,nodo2,pass) ) {
    if (!pega_senha (nodo2,nodo1,pass) ) {
      f_log.form("Nao achei senha pra esse link: %d %d\n",nodo1,nodo2);
      return 0;
    }
  }
  // f_log.form("Comparando senhas '%s' e '%s'\n",pass,senha);
  if (strcmp(pass,senha))
    return 0;
  else
    return 1;
```



```
}
void links::monta_arvore (unsigned int root) {
  nodo_list *nl;
  nodo_list_nodo *nln;
  unsigned int filhos[2][this->max_pos+1];
  int nfilhos[2], rodada;
  int i;

  rodada = 0;
  nfilhos[rodada] = 1;
  nfilhos[!rodada] = 0;
  filhos[rodada][0]=root;

  while (nfilhos[rodada] > 0) {
    for (i = 0; i < nfilhos[rodada]; i++ ) {
      // f_log.form("Pendurando em %d\n", filhos[rodada][i] );
      nl = &(this->nodoList[filhos[rodada][i]]);
      for (nln = nl->inicio->prox; nln != 0; nln = nln->prox ) {
//f_log.form("Peguei %d -> %d\n",filhos[rodada][i],nln->nodoId);
filhos[!rodada][nfilhos[!rodada]] = nln->nodoId;
nfilhos[!rodada]++;
this->destroy_links(nln->nodoId,filhos[rodada][i]);
      }
    } /* for */
    nfilhos[rodada] = 0;
    rodada = !rodada;
  } /* while */
}
/* Classe nodo_rb */
nodo_rb::nodo_rb () {
  rb=0;
  prox=0;
}
nodo_rb::nodo_rb (srb *s) {
  // f_log.form("To aqui dentro\n");
  rb=s;
  prox=0;
}
nodo_rb::~nodo_rb () {
  // pthread_cancel(rb->thread);
  // pthread_join(rb->thread,NULL);
  // delete rb;
}
/* Classe rb_list */
rb_list::rb_list () {
  inicio = fim = new nodo_rb;
  y_sem = 1;
  pthread_mutex_init(&sem,0);
  meuRBid = 0;
}
rb_list::~rb_list () {
  do {
    fim = inicio->prox;
    delete inicio;
    inicio = fim;
  } while (fim != 0);
  if (y_sem != 0) {
    pthread_mutex_unlock(&sem);
```

```
    pthread_mutex_destroy(&sem);
  }
}

rb_list & rb_list::ins (srb *s) {

  // f_log.form ("Estou sendo inserido na arvore: %d %d",fim->prox, fim->rb);
  // if (fim != inicio)
  // fim->rb->log("Antes");
  fim->prox = new nodo_rb(s);
  // s->log("depois da criacao");
  fim = fim->prox;

  return *this;
}

unsigned int rb_list::next () {

  sem_down();

  unsigned int x = meuRBid;

  meuRBid++;

  sem_up();
  return x;

}

unsigned int rb_list::next2 () {

  // sem_down();

  unsigned int x = meuRBid;

  meuRBid++;

  // sem_up();
  return x;

}

rb_list & rb_list::del (srb *s) {

  nodo_rb *p, *q;

  sem_down();
  for (p=inicio; p->prox != 0; p=p->prox) {
    if ( p->prox->rb == s ) {
      q = p->prox;
      p->prox = q->prox;
      delete q;
      if ( p->prox == 0 ) {
// f_log.form("Ahaaaaaaaaaaaaaa\n");
fim = p;
      }
      goto FIM;
    }
  }

 FIM:
  sem_up();
  return *this;
}

rb_list & rb_list::del (unsigned int r,unsigned int id) {

  nodo_rb *p, *q;

  sem_down();

  f_log.form("Matando broadcast que for raiz=%d, id=%d\n",r,id);
  for (p=inicio; p->prox != 0; p=p->prox) {
    if ( p->prox->rb->raiz == r && p->prox->rb->rblId == id ) {
      q = p->prox;
      p->prox = q->prox;

      // Como esta função só é usada quando um novo broadcast deve
      // interromper outro, então o status do rb que está sendo removido
      // deve ser alterado para ABORTADO
      //       f_log.form("Ahaaa\n");
      q->rb->status = srb::ABORTADO;
```



```
      delete q;
      if ( p->prox == 0 ) {
//f_log.form("Ahaaaaaaaaaaaaaa2\n");
fim = p;
      }

      goto FIM;
    }
  }

 FIM:
  sem_up();
  return *this;
}

srb * rb_list::responsavel (unsigned int raiz, unsigned int id, int abortar = 0) {

  nodo_rb *nr;

  // sem_down();
  for (nr = inicio->prox; nr != 0; nr = nr->prox ) {

    if ( nr->rb->responsavel (raiz,id) ) {
      if (abortar) {
nr->rb->status = srb::ABORTADO;
nr->rb->log("Broadcast descartado");
      }
      sem_up();
      return nr->rb;
    }
  }

  // sem_up();
  return 0;

}

void rb_list::limpeza () {

  nodo_rb *p, *q;

  sem_down();
  for (p=inicio; p->prox != 0; ) {
    if ( p->prox->rb->posso_limpar()) {
      //f_log.form("Achei alguém pra limpar\n");
      q = p->prox;
      p->prox = q->prox;
      q->rb->log("Limpando");
      delete q;
    } else {
      p=p->prox;
    }
  }

  sem_up();
}
void rb_list::sem_down () {

  // f_log.form("Lock no rbl\n");
  pthread_mutex_lock(&sem);

}

void rb_list::sem_up () {

  // f_log.form("UNLock no rbl\n");
  pthread_mutex_unlock(&sem);

}

/* Classe globais */

globais::globais (unsigned int nodes, unsigned int mesg_size, unsigned int bport,
  unsigned int sport, unsigned int pport,unsigned int id, unsigned int cport, int tm) {

  max_nodes = nodes;
  max_message_size = mesg_size;
  broadcast_port = bport;
  service_port = sport;
  ping_port = pport;
  meuId = id;
  client_port = cport;
```

```
  timeout = tm;
};

void globais::set (unsigned int nodes, unsigned int mesg_size, unsigned int bport,
  unsigned int sport, unsigned int pport,unsigned int id, unsigned int cport, int tm) {

  max_nodes = nodes;
  max_message_size = mesg_size;
  broadcast_port = bport;
  service_port = sport;
  ping_port = pport;
  meuId = id;
  client_port = cport;
  timeout = tm;

};
```

A.7 parser.cpp

Este arquivo implementa o processamento dos arquivos de configuração (parser):

```
#include <string.h>
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>
#include <assert.h>
#include <netdb.h>
#include <stdio.h>
#include <fstream.h>

#include "classes.h"
#include "srb.h"
#include "parser.h"

extern globais *variaveis_globais;
extern links *rotas;
extern nodos *descricao_nodos;
extern log f_log;

static int parser_conf (char *arq, char *argv) {

  unsigned int bport = BROADCAST_PORT;
  unsigned int sport = SERVICE_PORT;
  unsigned int pport = PING_PORT;
  unsigned int cport = CLIENT_PORT;
  unsigned int nodes = MAX_NODES;
  unsigned int mesg_size = MAX_MESSAGE_SIZE;
  int tm = TIMEOUT;

  unsigned int id = 0;

  char param[100];

  ifstream conf(arq);

  if (conf.bad()) {
    cerr << "Nao achei arquivo de configuracao " << CONF << '\n';
    return 0;
  }

  while ( ! conf.eof()) {

    conf >> param;

    if (!strcmp (param,"BROADCAST_PORT") ) {

      conf >> bport;
```



```
    } else if (!strcmp (param,"SERVICE_PORT") ) {

      conf >> sport;

    } else if (!strcmp (param,"PING_PORT") ) {

      conf >> pport;

    } else if (!strcmp (param,"MAX_NODES") ) {

      conf >> nodes;

    } else if (!strcmp (param,"MAX_MESSAGE_SIZE") ) {

      conf >> mesg_size;

    } else if (!strcmp (param,"IDENTIFICATION") ) {

      conf >> id;
      char *p = strstr(argv,"srb");
      sprintf(p,"srb%9d",id);

    } else if (!strcmp (param,"TIMEOUT") ) {

      conf >> tm;

    } else if (!strcmp (param,"CLIENT_PORT") ) {

      conf >> cport;
    }

  }

  assert(id!=0);

  if (variaveis_globais == 0) {

    variaveis_globais = new globais (nodes,mesg_size,bport,sport,pport,id,cport,tm);

  } else {

    variaveis_globais->set(nodes,mesg_size,bport,sport,pport,id,cport,tm);

  }

  f_log << "Identification: " << id << "\nGlobais: BPort " << bport << " SPort: " << sport << " PPort: "
        << pport << " MaxNodes: " << nodes << " MaxMessageSize: " << mesg_size << "\n\n";

  return 1;
}

static int parser_nodos (char *arq) {

  unsigned int id;
  unsigned char nome[100];
  unsigned short int bporta;
  unsigned short int pporta;
  nodo *n;

  ifstream conf(arq);

  if (conf.bad()) {
    cerr << "Nao achei arquivo de descricao de nodos " << NODOS << '\n';
    return 0;
  }

  if (!descricao_nodos) {
    descricao_nodos = new nodos();
    descricao_nodos->seta_semaforo();
  } else {
  //    descricao_nodos->limpa();
  }

  int x;

  while ( ! conf.eof()) {

    conf >> id >> nome >> bporta >> pporta;

    x = conf.rdstate();
    if (x & ios::failbit) {
    //  cerr << "Arquivo deve ser acabado \n";
      break;
    }
```

```
    n = new nodo(id,(char *)nome,bporta,pporta);

    if ( n->name[0] == 0) {
      delete n;
    } else {
      (*descricao_nodos) + n;
    }
  }

  f_log << "\n";
  return 1;
}

static int parser_rotas (char *arq) {

  unsigned int n1, n2;
  char senha[201];
  unsigned char linha[200];

  ifstream conf(arq);

  if (conf.bad()) {
    cerr << "Nao achei arquivo de roteamento " << ROTAS << '\n';
    return 0;
  }

  if (!rotas) {
    rotas = new links(1);
  } else {
    rotas->limpa();
  }

  istrstream *i;

  while ( ! conf.eof()) {

    conf.getline(linha,200);

    if (conf.fail())
      break;

    i = new istrstream((const char*)linha,200);

    (*i) >> n1 >> n2;

    //    conf >> n1 >> n2;

    if ( i->fail() ) {
      delete i;
      break;
    }

    (*i) >> senha;

    if (i->fail()) {
      senha[0]=0;
    }

    delete i;


    rotas->ins(n1,n2,senha);
    f_log << "Adicionando rotas entre " << n1 << " e " << n2 << " Senha: " << senha << "\n" ;
  }

  f_log << "\n";
  return 1;
}

int parser (int argc, char *argv[]) {

  int i=0;

  if (argc > 1) {

    if (parser_conf(argv[1],argv[0]) ) {
      i =1;
    }

  } else {
    if (parser_conf(CONF,argv[0]) ) {
      i =1;
    }
```



```
  }

  if (argc > 2) {

    if (parser_nodos(argv[2]) ) {
      i |= 2;
    }

  } else {
    if (parser_nodos(NODOS) ) {
      i |= 2;
    }
  }

  if (argc > 3) {

    if (parser_rotas(argv[3]) ) {
      i |= 4;
    }

  } else {
    if (parser_rotas(ROTAS) ) {
      i |= 4;
    }
  }

  return i;
}
```

A.8 rede.cpp

Este arquivo implementa a configuração de rede (sockets UDP) e é responsável, também, por todo o atendimento de pacotes vindos de outros nodos:

```
#include <pthread.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <iostream.h>
#include <sys/times.h>
#include <unistd.h>
#include <errno.h>
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

#include "rede.h"
#include "classes.h"

extern globais *variaveis_globais;
extern struct portas socks;
extern rb_list rbl;
extern links *rotas;
extern log f_log;

int inicializa_rede () {

  struct sockaddr_in dados;

  socks.broadcast = socket (AF_INET,SOCK_DGRAM,0);
  socks.servico = socket (AF_INET,SOCK_DGRAM,0);
  socks.ping = socket (AF_INET,SOCK_DGRAM,0);

  dados.sin_family = AF_INET;
```

```
  dados.sin_port = htons(variaveis_globais->broadcast_port);
  dados.sin_addr.s_addr = htonl(INADDR_ANY);

  if ( bind (socks.broadcast, (struct sockaddr *) &dados, sizeof(dados)) < 0) {
    cerr << "Falha no bind no socket do broadcast\n";
    return 0;
  }

  dados.sin_family = AF_INET;
  dados.sin_port = htons(variaveis_globais->service_port);
  dados.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

  if ( bind (socks.servico, (struct sockaddr *) &dados, sizeof(dados)) < 0) {
    cerr << "Falha no bind no socket do servico\n";
    return 0;
  }

  dados.sin_family = AF_INET;
  dados.sin_port = htons(variaveis_globais->ping_port);
  dados.sin_addr.s_addr = htonl(INADDR_ANY);

  if ( bind (socks.ping, (struct sockaddr *) &dados, sizeof(dados)) < 0) {
    cerr << "Falha no bind no socket do ping\n";
    return 0;
  }

  return 1;

}
/****************************************************************
Formato dos pacotes:
--------------------

Ping:

P----+++++++
  id  senha

****************************************************************/

static void *atende_broadcast (void *sock) {

  unsigned char pacote[variaveis_globais->max_message_size +1];
  struct sockaddr_in origem;
  socklen_t len;
  int tam;
  unsigned int remetente;
  char senha[11];
  unsigned int raiz, id;
  srb *rb;

  //  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);

  len = sizeof(struct sockaddr_in);

  tam = recvfrom(*((int *)sock),pacote,variaveis_globais->max_message_size,0,
      (struct sockaddr *) &origem, &len);

  if (tam < 0) {
    f_log.form("Atende_broadcast: %s\n", strerror(errno));
    pthread_exit(0);
  }

  pacote[tam]=0;

  if (pacote[0] != 'B' || tam < 27) {
    f_log.form("Atende_broadcast: protocolo errado\n");
    pthread_exit(0);
  }

  bcopy (pacote+1,&remetente,4);
  remetente = ntohl(remetente);
  bcopy (pacote+5,senha,10);
  senha[10]=0;

  if ( ! rotas->confere_senha(remetente,variaveis_globais->meuId,senha) ) {
    f_log.form("Senha errada: '%s'\n",senha);
    pthread_exit(0);
  }
```



```
  bcopy(pacote+16,&raiz,4);
  raiz = ntohl(raiz);

  bcopy(pacote+20,&id,4);
  id = ntohl(id);

  rbl.sem_down();

  if ( (rb=rbl.responsavel(raiz,id,0)) != 0) {
    // pacote repetido
    // apenas manda ack

    f_log.form("Pacote repetido\n");
    rbl.sem_up();
    rb->ack(&origem,len);
    pthread_exit(0);
  }

  f_log.form("Atendendo nova Difusão Confiável: raiz: %d, id: %d\n",raiz,id);

  rb = new srb(raiz,id, rotas, pacote,tam,&origem,len);

  rbl.sem_up();

  f_log.form("Criei novo srb: %d\n",rb->raiz);

  if ( rb->status == srb::ERRO ) {
    f_log.form("Pacote mal formado\n");
    delete rb;
    pthread_exit(0);
  }

  //  f_log.form("inserindo na arvore\n");

  rbl.sem_down();
  rbl.ins(rb);
  rbl.sem_up();

  f_log.form("iniciar difusao\n");
  rb->difundir();

  rb->ficar_respondendo();

  f_log.form("Vou deletar rb\n");
  delete rb;

  return 0;
}

static void *atende_servico (void *sock) {

  unsigned char pacote[variaveis_globais->max_message_size +1];
  struct sockaddr_in origem;
  socklen_t len;
  int tam;
  srb *rb;

  //  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);

  len = sizeof(struct sockaddr_in);

  tam = recvfrom(*((int *)sock),pacote,variaveis_globais->max_message_size,0,
        (struct sockaddr *) &origem, &len);

  if (tam < 0 ) {
    f_log.form("Atende_servico: %s\n",strerror(errno));
    pthread_exit(0);
  }

  if (ntohs(origem.sin_port) != variaveis_globais->client_port) {
    f_log.form("Atende_servico: porta do cliente nao esta batendo: %d\n",ntohs(origem.sin_port));
    pthread_exit(0);
  }

  pacote[tam]=0;

  f_log.form("Atendendo novo pedido de envio de mensagem\n");

  rbl.sem_down();
  rb = new srb(rotas,pacote,tam,&origem,len,0);

  rbl.ins(rb);
```

```
  rbl.sem_up();

  rb->difundir();
  //  rb->ficar_respondendo();

  f_log.form("Serviço completo! Vou deletar rb\n");
  delete rb;
  return 0;
}

static void atende_ping (int sock) {

  struct sockaddr_in origem;
  socklen_t len;
  int tam;

  char pacote[50];

  len = sizeof(struct sockaddr_in);

  tam = recvfrom(sock,pacote,49,0,(struct sockaddr *) &origem, &len);

  if (tam < 0) {
    f_log.form("Atende_ping: %s\n",strerror(errno));
    return;
  }

  pacote[tam]=0;

  if (pacote[0] != 'P') {
    f_log.form("Fora do padrao: \n");
    return;
  }

  sendto (sock,pacote,tam,0,(struct sockaddr *) &origem, len);

  return;
}


void atende_rede () {

  fd_set portas, info;
  int broadcast, servico,ping;
  pthread_t pt;
  pthread_attr_t attr;
  int atender;

  broadcast = socks.broadcast;
  servico = socks.servico;
  ping = socks.ping;

  FD_ZERO(&portas);
  FD_ZERO(&info);

  FD_SET (broadcast,&portas);
  FD_SET (servico,&portas);
  FD_SET (ping,&portas);

  atender = broadcast;

  if (servico > atender)
    atender = servico;

  if ( ping > atender )
    atender = ping;

  atender++;

  while (1) {
    info = portas;

    pthread_attr_init (&attr);
    pthread_attr_setdetachstate (&attr,PTHREAD_CREATE_DETACHED);

    if ( select (atender,&info,NULL,NULL,NULL) < 0) {

      if (errno == EINTR)
continue;

      assert(errno==EINTR);
    }
```



```
    if (FD_ISSET(broadcast,&info)) {
      pthread_create (&pt,&attr,atende_broadcast,(void *)&broadcast);
    }

    if (FD_ISSET(servico,&info)) {
      pthread_create (&pt,&attr,atende_servico,(void *)&servico);
    }

    if (FD_ISSET(ping,&info)) {
      atende_ping (ping);
    }
  }

}
```

A.9 main.cpp

Este arquivo implementa a função *main* responsável pelo início do programa, bem como coordenação da execução:

```
#include <pthread.h>
#include <iostream.h>
#include <unistd.h>
#include <stdio.h>

#include "classes.h"
#include "srb.h"
#include "parser.h"
#include "rede.h"

unsigned int meuId;
nodo_list filhos_mortos(1);
globais *variaveis_globais;
links *rotas;
nodos *descricao_nodos;
rb_list rbl;

log f_log;

struct portas socks;

int main (int argc, char *argv[]) {

  variaveis_globais = 0;
  rotas = 0;
  descricao_nodos = 0;

  parser(argc,argv);

  inicializa_rede();

  (f_log << "Enlaces:\n" << rotas << "\n").form(" ****** Fim da Inicialização ******\n\n");

  atende_rede();

  return 0;
}
```

A.10 cliente.c

Este arquivo simula algum sistema externo enviando uma mensagem para o sistema de difusão confiável difundir para os demais nodos:

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/times.h>
#include <unistd.h>
#include <errno.h>
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {

  int sock;
  struct sockaddr_in addr;

  if (argc != 3) {
    fprintf(stderr,"\nSintaxe: \n\tcliente <porta servidor> <mensagem>\n\n");
    exit(1);
  }

  sock = socket(AF_INET,SOCK_DGRAM,0);

  addr.sin_family = AF_INET;
  addr.sin_port = htons( (unsigned short) 4545);
  addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

  bind (sock, (struct sockaddr *) &addr, sizeof(addr));

  addr.sin_family = AF_INET;
  addr.sin_port = htons( (unsigned short) atoi(argv[1]));
  addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

  sendto (sock,argv[2],strlen(argv[2]),0,(struct sockaddr *) &addr, sizeof(addr));

  return 0;
}
```