Universidade Federal Fluminense

VIVIANE ANTONIA CORRÊA THOMÉ

Comunicação Coletiva em Ambientes Distribuídos Dinâmicos

NITERÓI

2006

VIVIANE ANTONIA CORRÊA THOMÉ

Comunicação Coletiva em Ambientes Distribuídos Dinâmicos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Processamento Distribuído e Paralelo.

Orientadora:
Lúcia Maria de Assumpção Drummond

Universidade Federal Fluminense

NITERÓI

2006

Comunicação Coletiva em Ambientes Distribuídos Dinâmicos

Viviane Antonia Corrêa Thomé

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre.

Aprovada por:

Profa. D.Sc. Lúcia Maria de Assumpção Drummond / IC-UFF (Presidente)

Profa. D.Sc. Maria Clícia Stelling de Castro / UERJ

Profa. Ph.D. Inês de Castro Dutra / COPPE-UFRJ

Prof. Ph.D. Eugene Francis Vinod Rebello / IC-UFF

Prof. Ph.D. Wagner Meira Júnior / DCC-UFMG

Niterói, 17 de Agosto de 2006.

À minha mamãe.

Agradecimentos

Talvez este momento seja um dos momentos mais esperados, agradecer àquelas pessoas que foram essenciais na nossa conquista. São tantos a quem devo falar Muito Obrigada.

Inicio agradecendo à pessoa que sempre foi minha referência, exemplo de força e determinação, agradeço sempre por tê-la perto de mim, mesmo que um pouquinho distante, muito obrigada Mamãe. Agradeço aos meus sobrinhos: Núbia, Bianca, Luísa, Rafael, Thales e Arthur; aos meus irmãos: Vanda, Vander e Vanete, vocês também contribuíram muito para a minha formação através de suas atitudes. Admiro e Amo muito vocês!

À Professora Lúcia agradeço pela sua orientação. Suas observações, críticas e elogios foram essenciais para que eu obtivesse a confiança necessária para o desenvolvimento do trabalho. Fico feliz de ter sido orientada por você.

Outros professores também foram muito importantes para mim. É com muito carinho que mais uma vez agradeço ao professor Alexandre Plastino, por me despertar a vontade em fazer o Mestrado e aos professores Otton, Vinod e Cristina Boeres que me deram o apoio de que precisava nesta fase.

Fico feliz em ter compartilhado tantos momentos com a Cristiane, Daniela e Renatha. Está sendo muito bom dividir parte da minha vida com vocês na nossa casa, que por aí insistem em chamar de "Gaiola" ;-)

Agradeço ao SERPRO, onde entrei enquanto cursava o Mestrado, lugar onde tive o prazer em conhecer muitas pessoas bacanas. Muito obrigada pela torcida, vocês não imaginam o quanto me ajudaram. Faço um agradecimento especial à Jorge Franklin Martins que me permitiu conciliar o trabalho e o estudo.

Finalmente, agradeço a todos os meus amigos que contribuíram nesta conquista, que torceram por mim e que me apoiaram. São muitos da Pós, da Faculdade e do Trabalho. É uma tarefa difícil listar todos, mas vale uma tentativa: Laura, Renata, Izabela, Maria, Angela, Alex, Glauco, Ary, Bruno, Diego, Kennedy, Robson, Luciana, Jacques, Jonivan, Alexandre, Aline, Cristiano, Aletéia, Luciene, Idalmis, Janine, Luiz Merschmman, Ha-

roldo, Sandoval, Cristiane, Luciana Brugiollo, Daniela, Rodrigo, Stênio, Ivairton, Johnny, Renatha, Marcel, Raphael, Rafael (Guto), Maysa, Tiago (Facada), Tiago (Jovem), Aldenir, Fred Castro, Pedro Paulo, Franklin, Márcia, Ezequiel, Samuel, Eduardo, Sílvia, Fanny, Cláudia, Eduardo (Dudu), Humberto, Letícia, Renata (Bdn), Ricardo, Humberto e Carlos Frederico.

Tenham certeza que cada um teve sua parcela de ajuda, pelas conversas, pelo apoio, pelas brincadeiras, pelas saídas, pelas dicas usadas na Dissertação. Todos ajudaram muito para que eu mantivesse o meu equilíbrio e finalizasse o trabalho.

Faço um agradecimento especial ao Joni, que com tantos anos de amizade, na reta final do Mestrado conseguiu se mostrar ainda mais próximo. Pode ter certeza que sua presença foi fundamental!

Finalizando os meus agradecimentos, afirmo que mesmo com tanto tempo dentro desta universidade (entre Mestrado e Graduação), é difícil resumir minha satisfação em ter estudado em um lugar tão acolhedor e com professores tão competentes. A UFF - Computação - me possibilitou cultivar grandes amizades e me ajudou a crescer profissionalmente. Tenho orgulho em ter estudado aqui!

Resumo

Existem vários trabalhos que propõem a utilização de uma árvore geradora para a utilização de operações coletivas em ambientes distribuídos. Na maior parte deles, a criação desta topologia ocorre no início da execução da aplicação e não considera posteriores alterações no ambiente de execução. Entretanto, sistemas distribuídos apresentam características dinâmicas, tais como variações no desempenho, falhas e recuperações dos canais de comunicação. Este trabalho apresenta uma ferramenta que disponibiliza operações de comunicação coletivas para o MPI, considerando tais características dinâmicas. Para isso, além da construção inicial de uma árvore geradora de custo mínimo para a representação da topologia, a ferramenta também realiza sua constante adaptação, através de dados coletados pelo *Network Weather Service* - NWS. Tanto a geração como a adaptação da árvore geradora de custo mínimo são realizadas através de algoritmos distribuídos.

Palavras-chave: Árvore geradora mínima dinâmica, comunicação coletiva MPI, ferramenta de monitoração NWS.

Abstract

There are many works that consider the use of a spanning tree to perform collective operations in distributed environments. In most of them, the construction of that topology occurs at application startup, and posterior changes in the execution environment are not considered. However, distributed systems are dynamic, presenting performance variations, failures and recoveries of communication channels. This work presents a tool that provides collective operations for MPI that consider such dynamic behaviour. At first, the tool builds a minimum spanning tree representing the initial topology and then performs adaptations according to the data collected by the Network Weather Service (NWS). The tree construction and adaptation algorithms are executed in a distributed fashion.

Keywords: Minimum spanning tree, MPI collective communication, NWS monitoring tool.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A maior parte das ferramentas de programação paralela atuais disponibiliza operações de comunicação em grupo baseadas em estruturas estáticas pré-definidas, tais como: árvores binomiais, árvores que distinguem canais pertencentes ao mesmo *site* dos canais que interconectam *sites* distantes ou, também, árvores organizadas conforme uma hierarquia multinível. Tais operações, tipicamente, não consideram características dinâmicas usualmente apresentadas por sistemas distribuídos durante a execução de uma aplicação, como alteração no desempenho, falha e recuperação dos canais de comunicação e processadores.

Este trabalho propõe uma ferramenta para o ambiente MPI que fornece operações coletivas que consideram mudanças topológicas relacionadas a falhas, recuperações e mudanças significativas de desempenho em um ou mais canais de comunicação. A topologia usada para tais operações é representada por uma Árvore Geradora de custo Mínimo (AGM), construída e adaptada dinamicamente através de algoritmos distribuídos, o que dispensa a concentração de informações sobre a rede em um único nó central [Thomé e Drummond 2006]. Para que a AGM reflita, durante a execução da aplicação, as reais condições do ambiente é necessário monitorar a rede. Caso sejam verificadas mudanças, em relação às condições anteriores, a árvore é atualizada. Cada processo monitora os seus canais incidentes através da ferramenta Network Weather Service [Wolski et al. 1997, Wolski et al. 1999, Wolski 1998, Wolski 2003].

Este trabalho se diferencia dos demais que tratam de operações de comunicação coletiva [den Burger et al. 2002, Lacour 2001, Karonis et al. 2000, Kielmann et al. 1999, Husbands e Hoe 1998, Saito et al. 2005] em relação, principalmente, às seguintes propostas:

- Utilização de uma árvore geradora de custo mínimo, construída de forma distribuída

com base no algoritmo proposto em [Gallager et al. 1983];

- Adaptação da árvore geradora mínima durante a execução da aplicação para refletir eventuais mudanças ocorridas no ambiente. O algoritmo para a adaptação foi construído de acordo com as idéias propostas em [Cheng et al. 1988];

- Utilização de dados coletados pelo NWS para criação e adaptação da árvore geradora mínima.

Testes foram realizados para avaliar o desempenho das operações de comunicação coletiva da ferramenta proposta em relação às empregadas mais comumente nas implementações do padrão MPI. Constatamos que a ferramenta proposta permitiu reduzir o tempo de execução dessas operações significativamente, considerando especialmente ambientes com diversas variações de desempenho de canais de comunicação.

Existem vários trabalhos recentes que tratam de operações de comunicação coletiva. Saito, Taura e Chikayama [Saito et al. 2005] propõem um método para execução de operações coletivas em árvores criadas dinamicamente, em tempo de execução, de acordo com o conhecimento da topologia existente. Este trabalho apresenta as seguintes diferenças em relação ao aqui proposto: as árvores usadas não são de custo mínimo, a biblioteca usada para troca de mensagens neste trabalho é a *Phoenix* e a monitoração da rede é feita através de *pings*, que não fornece informação sobre o desempenho do canal de comunicação a nível da aplicação.

Burger et al. apresentam a ferramenta TopoMon em [den Burger et al. 2002], que utiliza um processo central para reunir informação sobre a rota entre todos os sítios de um ambiente de Grade e gerar a topologia a ser utilizada pelas aplicações do usuário e bibliotecas de comunicação. O nó central cria dois tipos de árvores: uma de latência mínima e outra de largura de banda máxima. Para previsão de latência também usa-se o NWS. Note que neste trabalho usa-se o algoritmo seqüencial de Dijkstra para determinação dos caminhos mínimos e toda informação sobre a topologia é concentrada em um único nó.

As implementações do padrão MPI também evoluíram em relação às operações coletivas. Tipicamente, o MPI monta uma árvore binomial para comunicações coletivas sem levar em consideração a localização dos processos. Mais recentemente alguns trabalhos foram propostos objetivando melhorar as estruturas usadas para as operações de comunicação coletiva [Husbands e Hoe 1998, Karonis et al. 2002a, Karonis et al. 2002b, Karonis et al. 2000, Kielmann et al. 1999, Lab 2004, Lacour 2001]. Nas implementações realizadas em MPI-StarT [Husbands e Hoe 1998], MagPIe [Kielmann et al. 1999] e MPI-

LAM [Lab 2004] a rede é vista em duas camadas. MPI-StarT distingue entre comunicação intra e interclusters, enquanto MagPIe e MPI-LAM diferenciam comunicação entre LAN e WAN.

Karonis et al [Karonis et al. 2000] apresentam uma implementação de operações coletivas no MPICH-G baseada na visão multicamada da rede obtida através de informações disponibilizadas pelo Globus. Este trabalho foi aperfeiçoado em [Karonis et al. 2002a, Karonis et al. 2002b, Lacour 2001] permitindo melhoria na eficiência da execução de operações coletivas em ambientes de Grade. Mais especificamente, na biblioteca MPICH-G2 [Lacour 2001], cada processo é classificado de acordo com a sua localização. Esta classificação é realizada no início da execução da aplicação através da construção de tabelas, chamadas de Tabelas de Cores e de Identificação de Clusters, e se mantêm constantes durante toda a execução da aplicação.

Todos estes trabalhos para MPI utilizam uma topologia estática.

O restante desta dissertação está organizada da seguinte forma. No Capítulo seguinte são descritos os algoritmos distribuídos utilizados para a construção, adaptação da AGM e os critérios de parada implementados. No Capítulo 3, é apresentada a proposta inicial de uma nova biblioteca MPI e sua integração com a ferramenta de monitoração *Network Weather Service - NWS*. Para analisar o desempenho da nossa proposta, foram realizados testes computacionais com bibliotecas de comunicação coletiva que utilizam outras topologias de árvores estáticas no ambiente MPI. Os resultados dos experimentos computacionais são mostrados no Capítulo 4. No Capítulo 5 estão as conclusões.

Capítulo 2

Algoritmos Distribuídos para Geração e Adaptação da Árvore Geradora de Custo Mínimo

Uma árvore geradora de uma rede representa uma estrutura conectada contendo todos os nós desta rede e, no contexto deste trabalho, é empregada em redes ponto-a-ponto para disseminação eficiente de mensagens.

Foram implementados dois algoritmos distribuídos para esta ferramenta: um para construção e outro para adaptação da árvore geradora mínima (AGM), as duas próximas subseções descrevem estes algoritmos, respectivamente.

2.1 Construção da Árvore Geradora Mínima

O algoritmo implementado para a criação da árvore geradora mínima inicial se baseia no GHS, trabalho apresentado em [Gallager et al. 1983].

O GHS é um algoritmo distribuído assíncrono que determina a árvore geradora mínima sobre um grafo completo. Cada nó do grafo é um processo que sabe inicialmente os pesos dos canais adjacentes. Os processos executam o mesmo algoritmo e trocam mensagens com seus vizinhos até que a árvore seja construída. Depois que cada processo termina seu algoritmo local, ele sabe quais canais adjacentes estão presentes na árvore.

No artigo não é informado explicitamente como deve ser tratada a fila de mensagens que contém as mensagens que não foram executadas imediatamente. Neste trabalho, sugerimos a implementação de uma lista para a inclusão destas mensagens que não foram executadas. Em relação à retirada destas mensagens da lista, elas acontecem quando as

condições que levaram à sua inclusão tiverem sido satisfeitas. Outra modificação realizada, sobre o algoritmo proposto, foi a integração do algoritmo com a ferramenta de monitoração de rede que obtém as informações de latências para a construção da árvore.

2.1.1 Definições e Propriedades

Árvore Geradora de Custo Mínimo. Sejam V um conjunto de vértices, E um conjunto de arestas e G = (V, E) um grafo conectado com custo associado a cada aresta, uma árvore geradora T de G é uma árvore que contém todos os vértices de G. O custo da árvore geradora T é o somatório dos custos de suas arestas.

$$custo(T) = \sum_{(u,v)\in T} custo(u, v)$$

Uma árvore geradora de custo mínimo (AGM) de G é a árvore geradora T de custo mínimo.

Um Fragmento é definido como uma sub-árvore da AGM, isto é, um subconjunto de nós e arestas conectadas da AGM. No algoritmo GHS cada nó, inicialmente, é um fragmento e, no decorrer do mesmo, eles se unem em fragmentos maiores até formar a Árvore Geradora de Custo Mínimo. A aresta de saída de um fragmento é definida como sendo a aresta que liga dois nós, onde um deles pertence ao próprio fragmento e o outro nó está fora do fragmento.

Propriedade 1. Dado um fragmento de uma AGM, seja e uma aresta de saída do fragmento de peso mínimo. A união de e a um nó adjacente não pertencente ao fragmento produz outro fragmento pertencente à AGM.

Prova. Suponha que a aresta incluída e não pertença à AGM. Então, existe um ciclo formado pela aresta e e um subconjunto de arestas das arestas da AGM. Ao menos uma aresta $x \neq e$ é também uma aresta de saída do fragmento, tal que $w(x) \geq w(e)$, onde w representa o custo da aresta. Então, removendo x da AGM e incluindo e uma nova árvore geradora é construída, que será mínima se a árvore original for mínima. O fragmento original com e incluído é um fragmento da nova AGM.

Propriedade 2. Se todas as arestas de um grafo conectado têm pesos distintos, então, a árvore geradora mínima construída é única.

Prova. Suponha, por contradição, que há duas AGM's diferentes. Seja e a aresta de

peso mínimo que está em apenas uma das árvores e, seja T o conjunto de arestas da AGM que possui e e T' o conjunto de arestas da outra AGM. O conjunto de arestas $\{e\} \bigcup T'$ precisa ter um ciclo e ao menos uma aresta deste ciclo chamada e', não estará presente em T (desde que T não possua ciclos). Se todas as arestas são diferentes e e' está presente em apenas uma das árvores, $w(e) < w(e')$. Então, $\{e\} \bigcup T' - \{e'\}$ é um conjunto de arestas da árvore geradora de peso menor que T', levando a uma contradição.

As Figuras 2.1, 2.2, 2.3, 2.4 e 2.5, mostram uma visão simplificada sobre as seqüências de passos para a construção de uma árvore geradora mínima. A Figura 2.1 apresenta cinco fragmentos desconectados quaisquer contendo apenas um nó cada e identificados pelos números *0*, *1*, *2*, *3* e *4*. Na Figura 2.2 é identificada a aresta de saída de menor custo de cada fragmento. As arestas de saída de custo mínimo estão representadas por traços mais fortes sobre as arestas. As Figuras 2.3 e 2.4 apresentam as combinações dos fragmentos iniciais em fragmentos maiores. Por último, na Figura 2.5, é apresentado o fragmento final gerado através da união dos fragmentos pelas arestas de custo mínimo. A forma como os fragmentos coordenam estas decisões será mostrada na subseção que descreve o algoritmo para a construção da árvore geradora mínima.

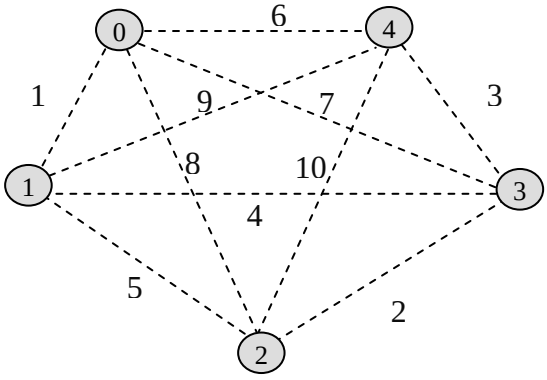


Figura 2.1: Fragmentos iniciais que irão compor a AGM.

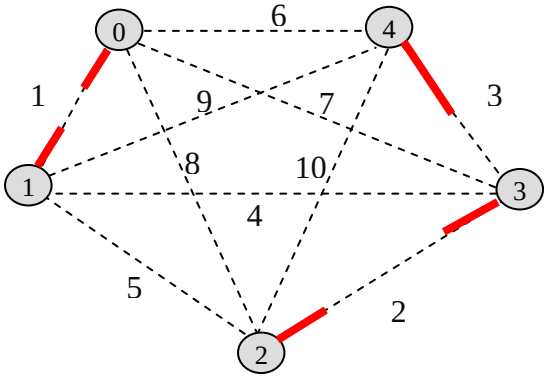


Figura 2.2: Aresta de saída de custo mínimo de cada fragmento.

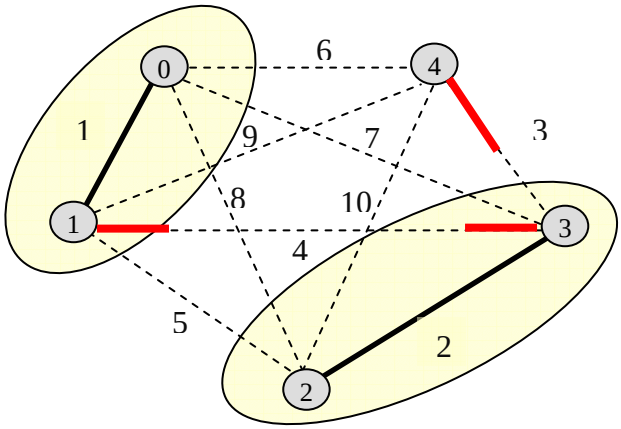


Figura 2.3: Representação dos novos fragmento gerados pela combinação dos fragmentos iniciais e as suas arestas de saída de custo mínimo.

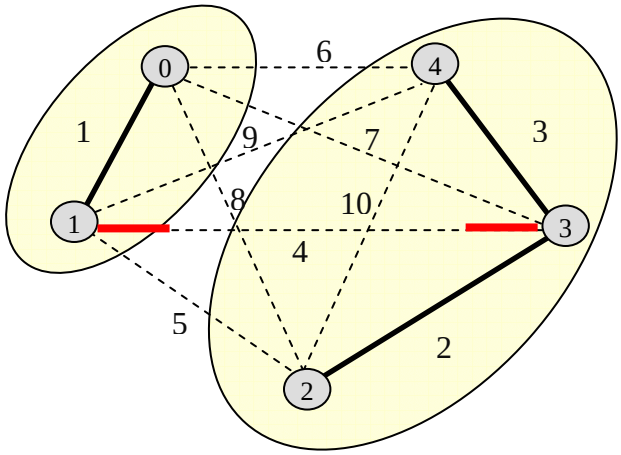


Figura 2.4: Representação dos dois fragmentos restantes e a aresta de custo mínimo entre eles.

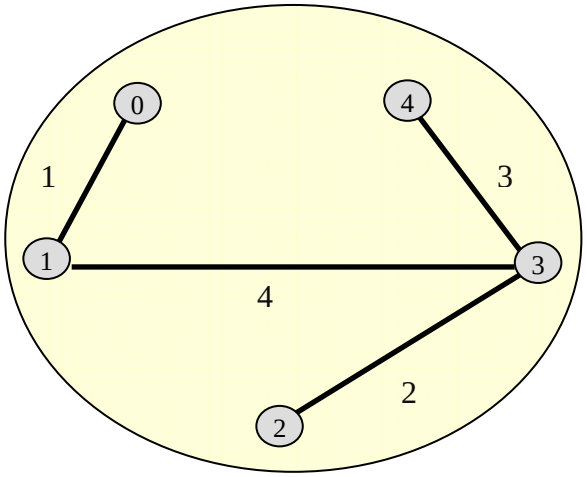


Figura 2.5: Representação do fragmento final gerado ou Árvore Geradora de custo Mínimo.

2.1.2 Descrição do Algoritmo

Todos os nós executam o mesmo programa que enviam mensagens pelas arestas adjacentes, aguardam o recebimento de mensagens e processam as mensagens recebidas. As mensagens trocadas são para a construção da árvore e podem ser transmitidas independentemente, em ambas as direções das arestas adjacentes. Depois que cada nó termina seu algoritmo local, ele sabe quais das suas arestas adjacentes estão presentes na árvore.

Cada nó do grafo está associado a um processo que mantém informações sobre o fragmento a que pertence, tais como:

- identificação do fragmento - *fid*;
- identificação da raiz do fragmento - *root*;
- estado do fragmento - *state*, que pode ser:
  Sleeping: nó não está participando do algoritmo;
  Find: nó está procurando a aresta de custo mínimo a ser incluída na árvore;
  Found: nó já terminou a rodada pela procura da aresta de custo mínimo.

Além dos atributos já citados, existem outros que serão explicados à medida que forem utilizados nas trocas de mensagens para a construção da árvore, são eles: *level*, *best-edge*, *best-wt*, *test-edge*, *find-count* e *in-branch*.

Em relação à classificação dos canais adjacentes a cada nó, eles podem estar em um dos três estados:

- *Branch*: aresta é um ramo no fragmento corrente, ou seja, ela já faz parte da árvore;
- *Rejected*: indica que a participação desta aresta está impedida, pois a sua inclusão irá acarretar na formação de ciclos;
- *Basic*: aresta não é *Branch* nem *Rejected*, portanto, está livre para ser usada.

Cada fragmento inicia sua participação no algoritmo através da execução de um procedimento chamado *wakeup*. A execução deste procedimento acarretará uma seqüência de trocas de mensagens entres os processos até que a árvore geradora mínima seja construída. As mensagens trocadas poderão ser: *Connect*, *Initiate*, *Test*, *Accept*, *Reject*, *Report*, *Change-core* e *Gosleep*. O recebimento de algumas mensagens poderá levar à execução de outros procedimentos, tais como: *test*, *report* e *Change-core*. A maneira como ocorrem as trocas de mensagens e a execução dos procedimentos é explicada mais adiante.

O algoritmo inicia com cada fragmento contendo apenas um nó e termina com apenas um único fragmento que é a própria AGM. A descrição do algoritmo será acompanhada por figuras que ilustram uma possível execução deste. O exemplo contém cinco fragmentos, todos com apenas um único nó no início da execução. A Tabela 2.1 apresenta os custos das arestas entre todos os nós utilizados no exemplo. Sendo que cada nó possui apenas as informações relativas às latências dos canais adjacentes a ele.

	0	***1***	***2***	***3***	***4***
0	0,00	10,00	80,00	70,00	40,00
1	10,00	0,00	50,00	20,00	60,00
2	80,00	50,00	0,00	5,00	90,00
3	70,00	20,00	5,00	0,00	35,00
4	40,00	60,00	90,00	35,00	0,00

Tabela 2.1: Custos de todas as arestas do grafo exemplo.

A seguir são explicados os procedimentos que podem ser executados e os tipos de mensagens trocadas entre os processos para a construção da árvore geradora mínima. Inicialmente, todos os nós encontram-se no estado *Sleeping* e, neste caso, cada nó é o próprio *Fragmento*. Além do atributo *state*, o atributo *level* ou nível do fragmento, poderá ser visto pela Figura 2.6. Este último é inicializado com o valor zero e sua função é, basicamente, controlar a união entre fragmentos, não permitindo que fragmentos de nível menor absorvam fragmentos de nível maior, como pode ser visto mais adiante. A informação *id* mostrada nesta figura, e nas demais que são apresentadas, possui a identificação de cada processo participante do fragmento.

Um nó, ou um conjunto de nós, pode iniciar sua participação no algoritmo espontaneamente ou a partir do recebimento de uma mensagem. A sua participação no algoritmo se inicia através da execução do procedimento *wakeup*. Na nossa implementação achamos mais interessante que todos os nós já executassem, no início da aplicação, o procedimento *wakeup* sem precisarem ser acordados através do recebimento de mensagem de outro nó.

Procedimento wakeup. Este procedimento determina o início da participação do nó na construção da árvore geradora mínima. Ao acordar, cada fragmento procura sua aresta de saída de custo mínimo, no estado *Basic*, em relação aos outros fragmentos, assincronamente.

Quando esta aresta é encontrada, ela é marcada como *Branch* e uma mensagem *Connect(level)* é enviada para o outro vértice desta aresta levando consigo o nível (*level*)

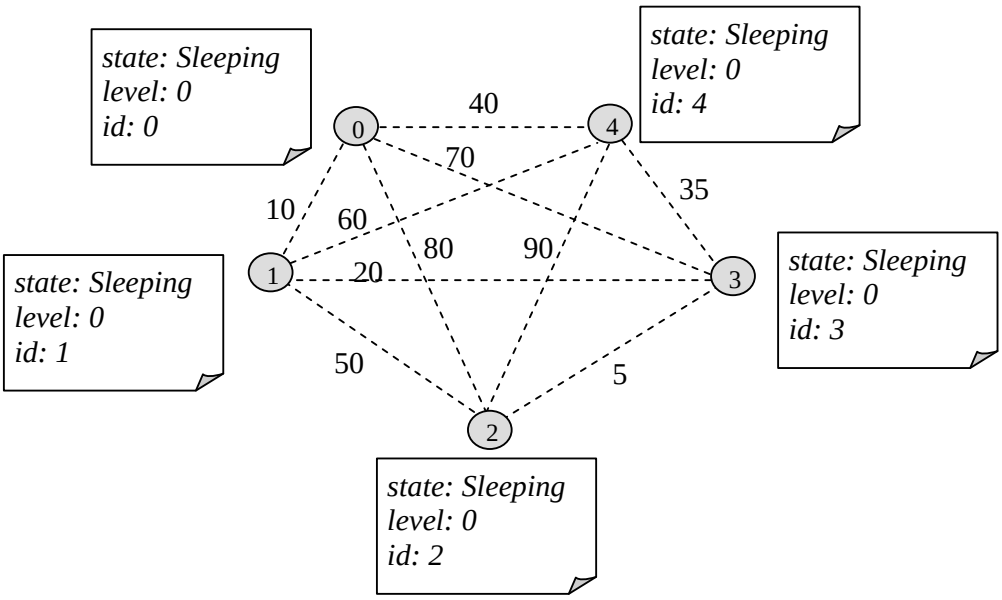


Figura 2.6: Estado dos processos no início do algoritmo.

do fragmento. A identificação do vértice destino é armazenada na variável *in-branch* que representa o pai do nó na árvore. O estado do fragmento, neste momento, é atualizado para *Found* e o atributo *fid* recebe a *identificação do processo (id)*. O envio da mensagem *Connect(level)* e as mudanças ocorridas durante a execução deste procedimento, podem ser vistas pela Figura 2.7.

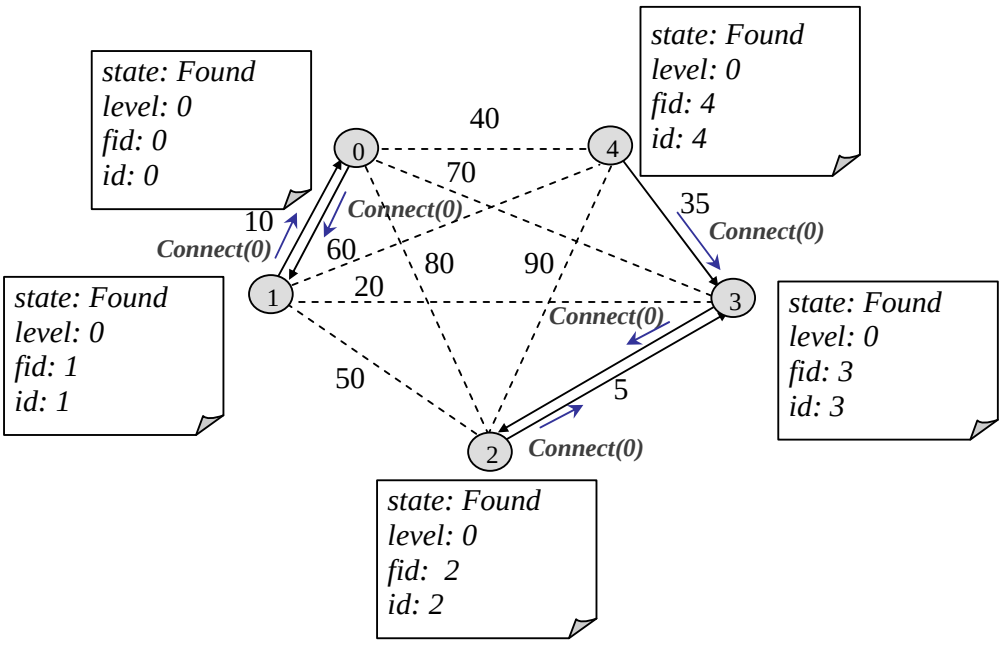


Figura 2.7: Envio da mensagem *Connect(level)* pela aresta adjacente de peso mínimo.

Mensagem Connect(L). Ao receber uma mensagem *Connect (L)*, um processo verifica se já está participando do algoritmo, caso não esteja, executa o procedimento *wa-*

keup, descrito anteriormente. Lembrando que na nossa implementação todos os processos acordam inicialmente, não sendo necessário que sejam acordados através do recebimento de mensagens. Os passos para tratar o recebimento da mensagem *Connect(L)* estão explicados a seguir:

Primeiro, comparam-se os níveis, o recebido pela mensagem *Connect* (L) e o nível armazenado no próprio fragmento (*level*). As possíveis combinações entre os níveis local (*level*) e recebido (L) possuem tratamentos específicos como segue:

1. Se $L < level$

   Significa que o fragmento que enviou a mensagem *Connect* pode ser absorvido imediatamente pelo fragmento que recebeu a mensagem. A Figura 2.8 mostra o envio desta mensagem e a absorção do fragmento pelo fragmento de nível maior. Os procedimentos adotados para a união desses fragmentos foram os seguintes:

   - A aresta que une os dois fragmentos é marcada como *Branch*;
   - Uma mensagem *Initiate (level, fid, state)* é enviada de volta para o processo que lhe enviou a mensagem *Connect(L)*, para que sejam atualizados os dados do fragmento que acabou de ser absorvido com as informações do fragmento que possui nível maior;
   - Se o fragmento de maior nível estiver no estado *Find* significa que está procurando por uma aresta de saída. Neste caso, o fragmento que acabou de ser absorvido também deve ajudar nesta busca. Para isto, o atributo *find-count* do processo que enviou *Inititate(level, fid, state)*, com valor zero no início, deverá ser incrementado. Este atributo é importante para indicar a quantidade de mensagens que precisa receber, com os custos das arestas de saída encontradas.

2. Se $L \geq level$

   - Se $L > level$

     A mensagem vem por uma aresta no estado *Basic* e não pode ser processada imediatamente. Um fragmento com nível menor não pode absorver um fragmento de nível maior, se isto ocorresse, este algoritmo permitiria a geração de ciclos. A solução, então, é colocar esta mensagem numa fila para que possa ser retirada no momento em que o nível de seu fragmento tenha aumentado o

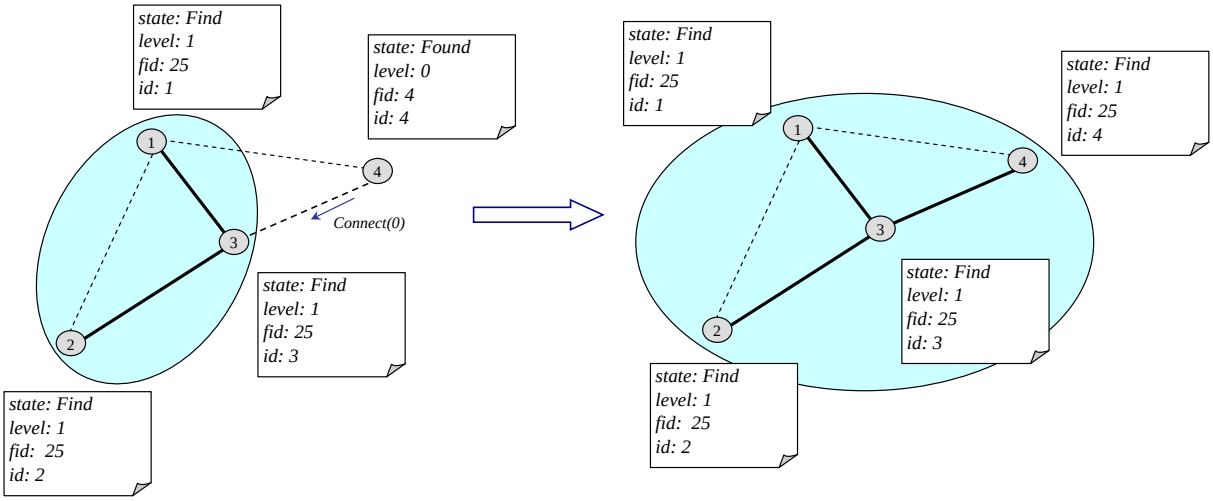


Figura 2.8: Recebimento mensagem *Connect(level)* e absorção de um fragmento com nível menor por outro de nível maior.

suficiente para que a absorção seja permitida, como apresentado na Figura 2.9. O tratamento da retirada de mensagens desta fila está descrito mais adiante.

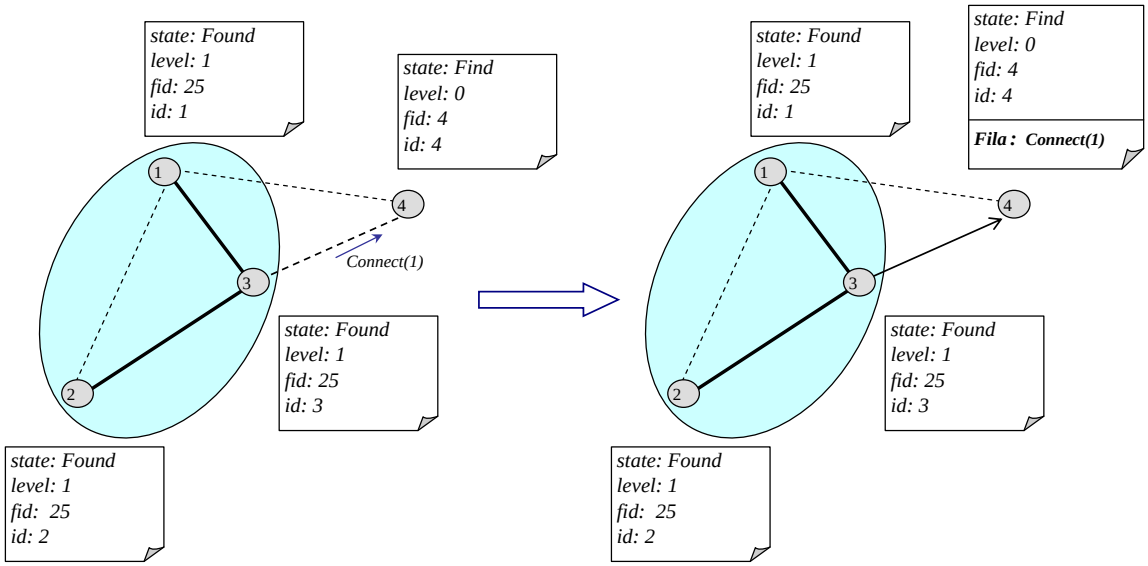


Figura 2.9: Recebimento mensagem *Connect(level)* de um fragmento com nível maior.

- Se $L = level$

  Neste caso, uma outra mensagem *Connect(level)* também foi enviada por esta mesma aresta, mas no sentido contrário. Isto quer dizer que os dois fragmentos concordaram ser esta a melhor aresta a ser incluída. Como os níveis dos dois fragmentos são iguais, um fragmento não é absorvido pelo outro, mas se combinam para formar um fragmento de nível maior.

  A aresta que unir estes dois fragmentos passa a ser a aresta coordenadora, ou

aresta *core*, do novo fragmento gerado e o custo desta aresta passa a ser a identificação do fragmento. Em seguida, uma mensagem *Initiate(level+1, w, Find)* é enviada para o outro nó que compõe a aresta *core*. As informações enviadas junto a esta mensagem e seus respectivos valores são: nível (*level+1*), identificação (*w* que é o peso da aresta *core*) e estado (*Find*) do fragmento. A identificação do novo fragmento através do custo da aresta *core* é possível porque garantimos que todas as arestas possuem pesos distintos, acrescentamos no valor das latências as identificações dos nós que a compõem. Cada nó, por sua vez, é identificado de forma única.

A aresta *core* é responsável pela disseminação da mensagem *Inititiate(level, fid, state)*, pela decisão de qual a melhor aresta a ser incluída na árvore e, também, pela identificação do término do algoritmo de criação da árvore geradora mínima.

Na Figura 2.10 é mostrado o envio da mensagem *Inititate(level, w, state)* como resposta ao recebimento da mensagem *Connect(L)*, quando os níveis dos fragmentos envolvidos na troca de mensagens *Connect* são iguais. Esta figura fornece a seqüência de execução do exemplo proposto inicialmente.

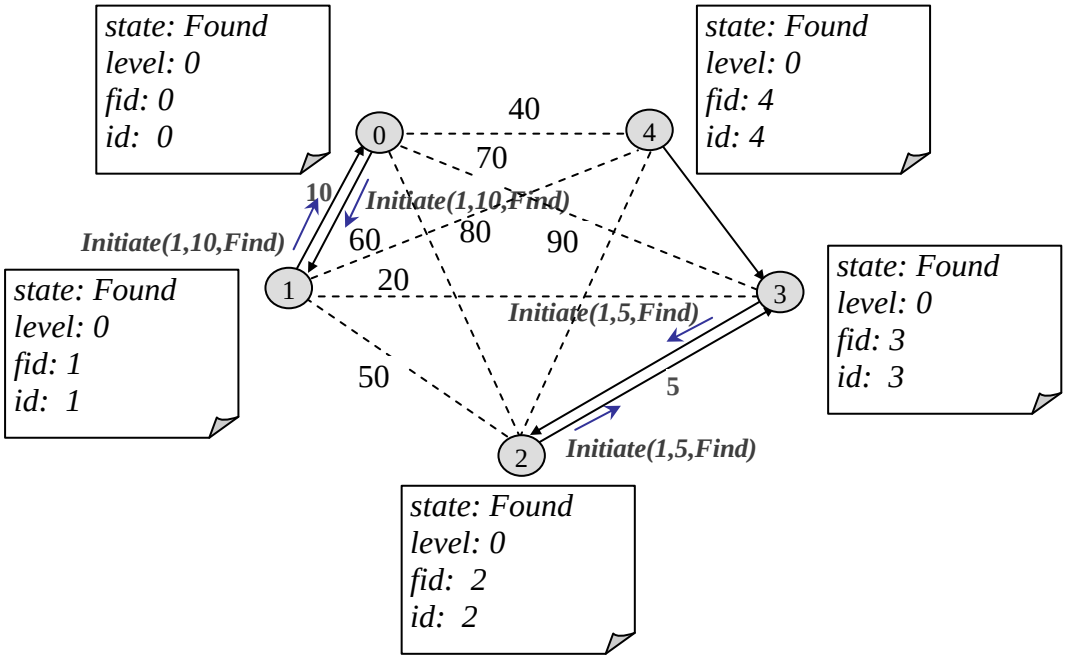


Figura 2.10: Troca de mensagens *Inititate(level, w, state)*.

Mensagem Initiate(L,F,S). O recebimento de uma mensagem *Initiate (L,F,S)* significa que, neste momento, o fragmento que a recebeu começa a fazer parte de um

outro fragmento. Ele pode simplesmente estar sendo absorvido por outro fragmento de maior nível ou pode estar ajudando a criar um novo fragmento de nível diferente.

Ao receber esta mensagem, este processo atualiza imediatamente seus dados, *level*, *fid* e *state*, com os respectivos valores L, F e S recebidos através dos paramêtros da mesma. Caso o parâmetro S seja igual a *Find*, indica que o processo que acabou de receber esta mensagem precisa ajudar na pesquisa pela aresta de saída para que a expansão do fragmento continue.

As variáveis, a seguir, também são atualizadas: *in-branch* recebe a identificação do processo que lhe enviou a mensagem *Initiate (L,F,S)* determinando quem é o seu pai na árvore; *best-edge* recebe *-1* e *best-wt* é marcada com o valor ∞, estas duas últimas variáveis indicam que a melhor aresta, para esta rodada, ainda não foi definida.

A mensagem *Initiate (L,F,S)* é propagada por todas as arestas deste fragmento que estejam no estado *Branch*, exceto por aquela aresta que recebeu a mensagem inicialmente, aresta que liga ao processo pai (*in-branch*). Se o estado do fragmento for igual a *Find*, a cada envio de mensagem *Initiate (L,F,S)* a variável *find-count* é incrementada de um e o procedimento *Test(level, fid)* deve ser executado, como pode ser visto pela Figura 2.11. Esta figura mostra o momento em que o processo *4* começa a sua participação no algoritmo através da execução do procedimento *wakeup* que envia a mensagem *Connect(0)*.

Continuando a execução, a Figura 2.12 apresenta a resposta à mensagem *Connect(0)* enviada pelo processo 4 que é o recebimento da mensagem *Initiate(1,5,Find)*. Ao receber esta mensagem o fragmento *4* é absorvido pelo fragmento *5*.

Após o fragmento *4* ser absorvido pelo fragmento *5*, o processo atualiza *state* com valor *Find* recebido. Após esta atualização, o processo envia mensagem *Test(level,fid)* para ajudar na busca pela melhor aresta, como pode ser acompanhado pela Figura 2.13.

Procedimeno Test. Este procedimento tem por objetivo enviar uma mensagem Test (*level,fid*) pela aresta de custo mínimo no estado *Basic*. No envio da mensagem *Test* são informados o nível e a identificação do fragmento. Neste momento, o atributo *test-edge* receber a identificação da aresta de custo mínimo encontrada, ou seja, armazena a identificação da aresta candidata à inclusão no fragmento. Se nenhuma aresta *Basic* for encontrada, *test-edge* recebe o valor -1 indicando que nenhuma aresta está sendo testada e o procedimento *Report* deve ser executado.

Mensagem Test(L,F). Um nó, ao receber uma mensagem *Test(L, F)*, compara se o valor do nível recebido pela mensagem (L) é maior que o nível de seu fragmento (*level*).

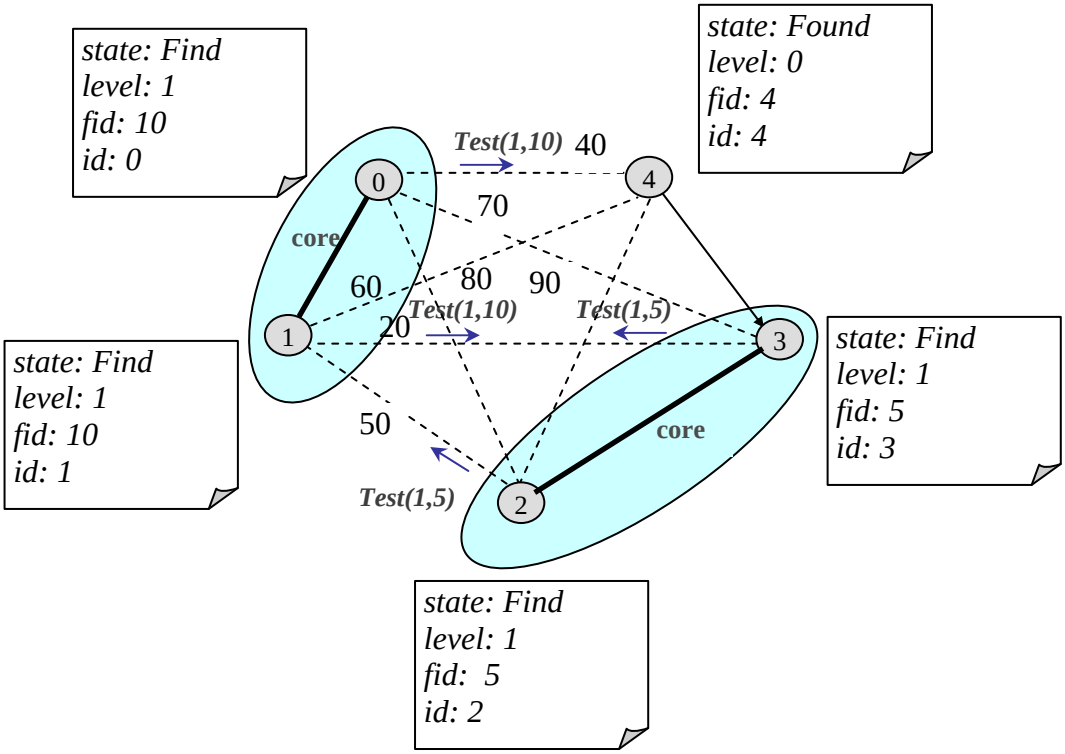


Figura 2.11: Envio da mensagem *Test(L,F)* pela aresta *Basic* de custo mínimo e execução do procedimento *wakeup* pelo processo *4*.

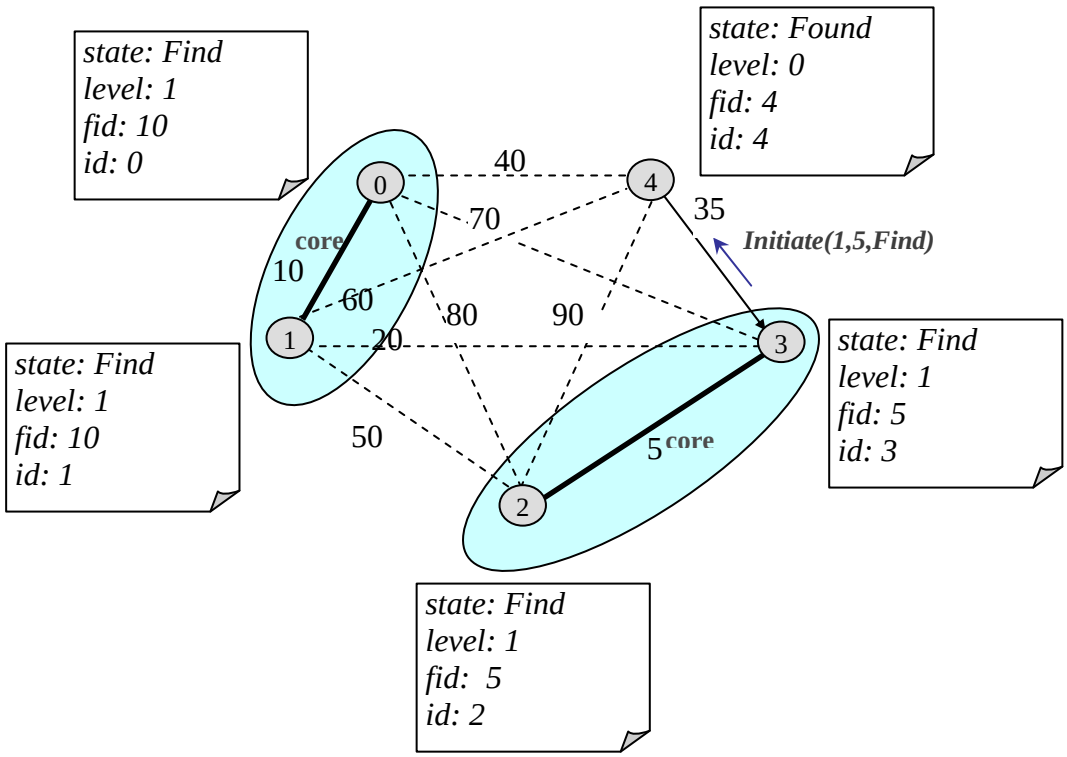


Figura 2.12: Envio da mensagem *Inititate(level, w, state)*, fragmento *4* absorvido.

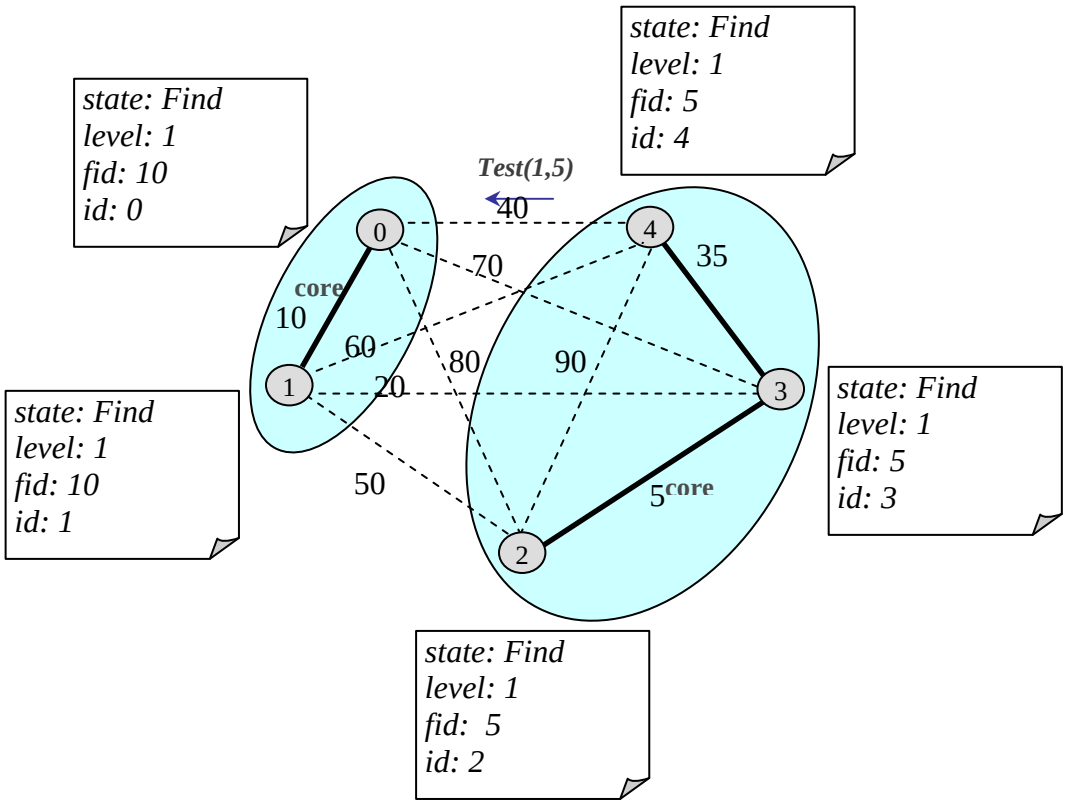


Figura 2.13: Envio da mensagem *Test(level,fid)* pelo processo *4*.

Se o nível recebido for menor ou igual, as identificações *(F)* e *(fid)* são comparadas. Caso sejam diferentes, uma mensagem *Accept* é enviada de volta informando que esta aresta é uma aresta candidata a unir os dois fragmentos. Se as identificações dos dois fragmentos forem iguais, significa que eles já pertencem ao mesmo fragmento, portanto, a união por esta aresta causa um ciclo. Neste caso, uma mensagem *Reject* deve ser enviada como resposta e a aresta testada é marcada como *Rejected*. Se a aresta rejeitada for a mesma que o nó escolheu para testar, ou seja, sua identificação está amazenada na variável *test-edge*, então o procedimento *Test* deve ser executado novamente para que uma nova mensagem *Test(level, fid)* seja enviada por outra aresta *Basic*.

Se o nível recebido pela mensagem *Test(L,F)* for maior, esta mensagem deve ser colocada numa lista de espera até que o nível do fragmento aumente o suficiente para ser retirada da lista. O nível de um fragmento, assim como a sua identificação, só muda quando o fragmento se unir a outro fragmento de mesmo nível ou quando for absorvido por um fragmento de nível maior. Estes atributos somente são atualizados após recebimento de mensagem *Initiate(L,F,S)*.

O problema de permitir o tratamento da mensagem *Test(L,F)* quando o nível que veio pela mensagem for maior, pode ser visto pela Figura 2.14, explicada a seguir:

Seguindo a ordem das mensagens recebidas, temos:

(1) Processo *2* envia mensagem *Connect(1)* para processo *7*;

Fragmento com identificação igual a *25* e nível igual a *1* verificou que sua aresta de custo mínimo é a que liga o processo *2* ao *7*, este último pertencente ao Fragmento *20* de nível *2*.

(2) Processo *7* verifica que pode absorver o Fragmento *25* e envia mensagem *Initiate(2,20,Find)* para processo *2*;

(3) Processo *2* recebe a mensagem *Initiate(2,20,Find)* e repassa para processo *3*.

A partir deste momento todos os processos pertencentes ao Fragmento *25* devem atualizar seus atributos, indicando que agora fazem parte do Fragmento *20*;

(4) Processo *4* envia mensagem *Test(2,20)* para processo *1*;

(5) Este passo é executado depois que o processo *1* receber a mensagem *Test(2,20)*.

Se processo *1* colocar esta mensagem na fila após comparar os níveis, não há nenhum problema. A mensagem *Test(2,20)* só pode ser liberada e tratada após recebimento da mensagem *Initiate(2,20,Find)*, quando terá seu nível, identificação e estado do fragmento atualizados. Neste caso, a mensagem *Test(2,20)* é rejeitada, pois o processo *1* após atualizar seus atributos pertence ao mesmo fragmento de quem lhe enviou a mensagem.

Caso o processo *1* não leve em consideração o nível, ele continua o processamento comparando as identificações do seu fragmento e do fragmento que enviou a mensagem *Test(2,20)*. Como as identificações desses fragmentos são diferentes, o processo *1* responde ao processo *4* com a mensagem *Accept*. Esta aresta é considerada uma candidata a ser incluída no fragmento e, caso seja a escolhida, um ciclo é criado na árvore.

Retornando ao nosso exemplo de execução, são apresentados os passos após recebimento da mensagem *Test(level, fid)* e envio da mensagem *Accept* como resposta, pela Figura 2.15. Neste momento, a mensagem *Accept* está sendo enviada pelo processo *4* um pouco antes do seu fragmento ser absorvido pelo fragmento *5*, isso pode ser percebido comparando os fragmentos apresentados nas Figuras 2.13 e 2.15.

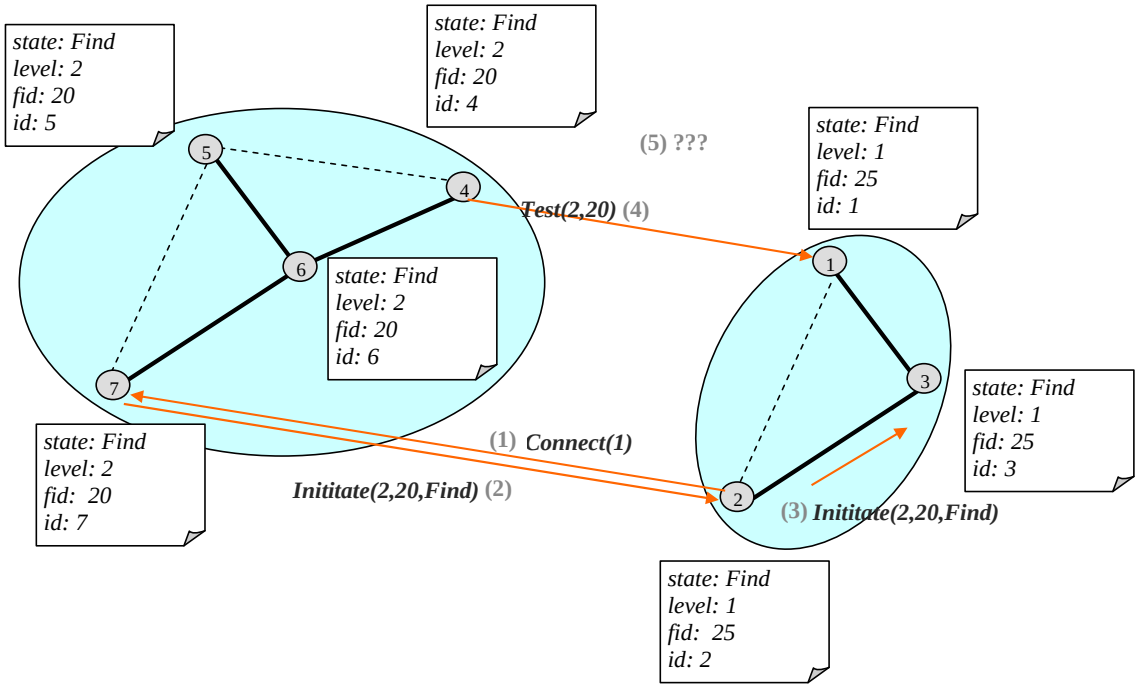


Figura 2.14: Recebimento mensagem *Test(level,fid)* de um fragmento com nível maior.

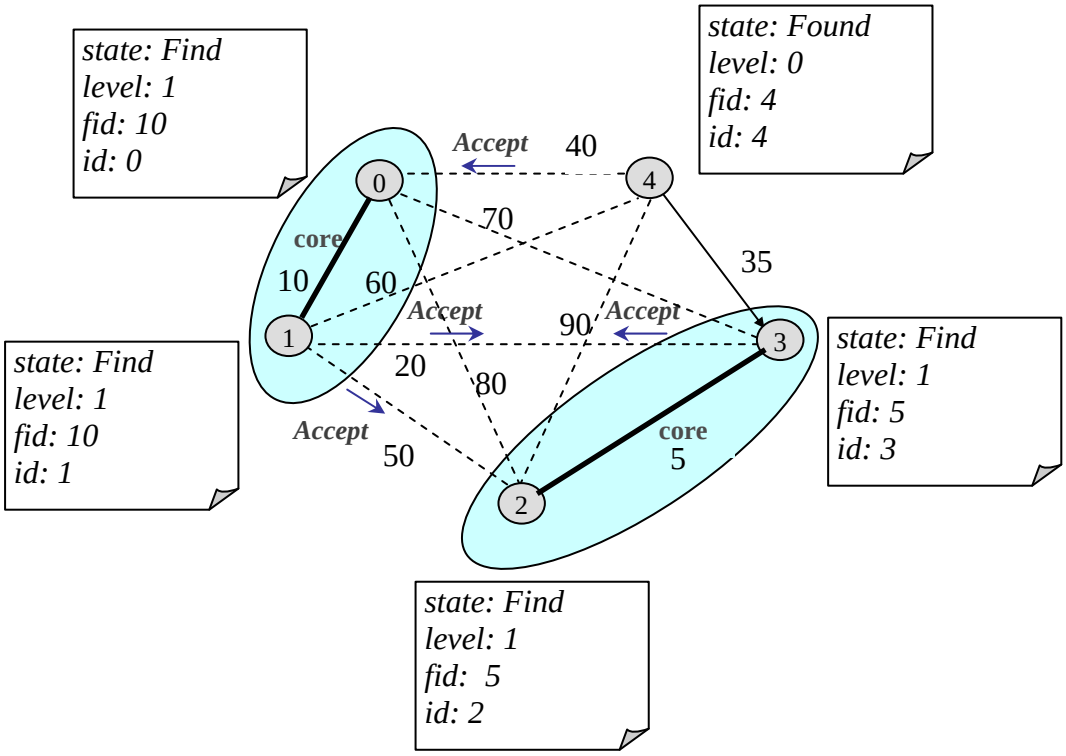


Figura 2.15: Envio da mensagem *Accept*.

A Figura 2.16 apresenta a resposta à mensagem *Test(L,F)* recebida quando o fragmento *4* foi absorvido. A mensagem de resposta foi um *Accept*, uma vez que os dois processos pertencem a fragmentos diferentes.

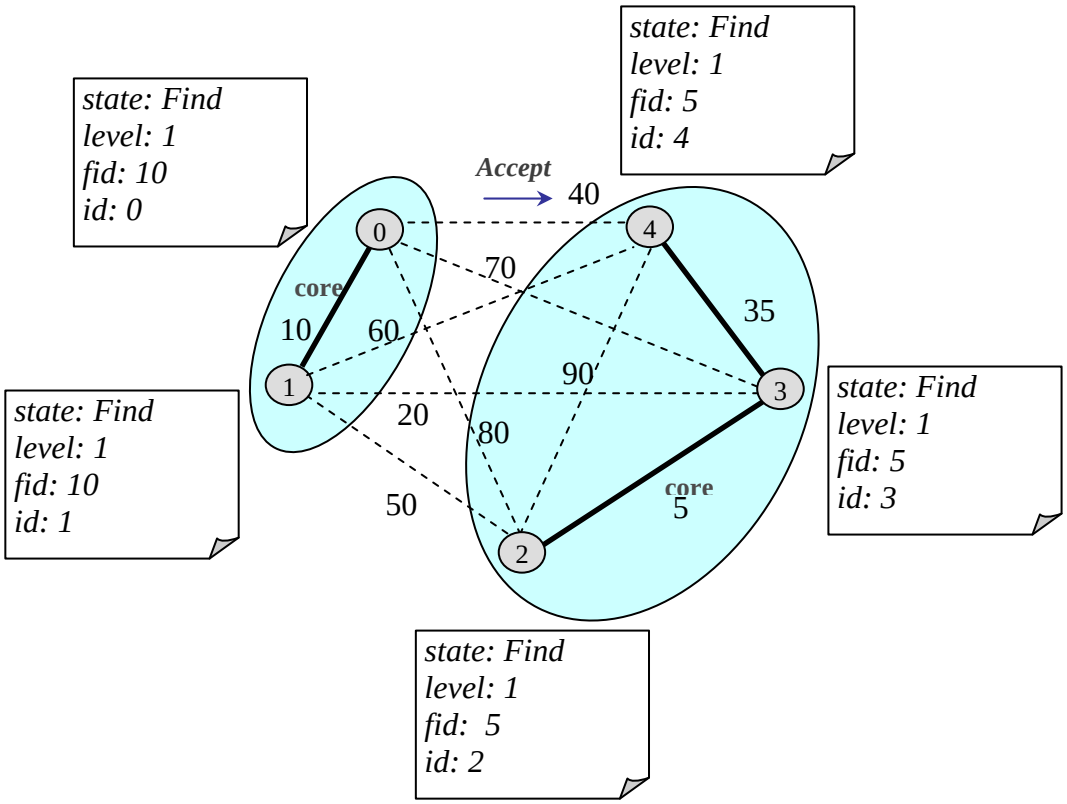


Figura 2.16: Envio da mensagem *Accept*.

Mensagem Accept. Ao receber uma mensagem *Accept*, o nó reinicializa a variável *test-edge*, com o valor *-1*, indicando que encerrou a busca da aresta de menor custo nesta rodada. O recebimento de uma mensagem *Accept* certifica que a aresta testada, através de envio da mensagem *Test(level, fid)*, é uma aresta candidata à aresta de peso mínimo de todo o fragmento.

O recebimento de uma mensagem *Accept* pode causar a atualização dos atributos *best-edge* e *best-wt* com a identificação e o peso da aresta pela qual recebeu a mensagem, respectivamente. Esta atualização só ocorre se o custo recebido pela mensagem *Accept* for menor que o custo armazenado no atributo *best-wt* do processo. No final do tratamento desta mensagem, o procedimento *Report* é executado.

Mensagem Reject. Ao receber uma mensagem *Reject*, o processo coloca a aresta testada no estado *Rejected*, sendo impedida a sua participação na árvore geradora mínima. Em seguida, o procedimento *Test(L, F)* deve ser executado.

Procedimento Report. Quando o processo executa este procedimento, ele verifica se já recebeu todas as mensagens *Report(w)* que esperava através da análise do atributo *find-count*. Se *find-count* for igual a 0 e não há aresta sendo testada no momento, o estado do processo é atualizado para *Found* e uma mensagem *Report(best-wt)* deve ser enviada para o processo pai (*in-branch*). Estes passos estão representados na Figura 2.17.

O objetivo do envio da mensagem *Report(best-wt)* é informar a identificação e o peso da melhor aresta candidata encontrada. Esta informação é repassada para o processo pai até alcançar a aresta *core* do fragmento, quando é decidida qual aresta é incluída para formar um novo fragmento.

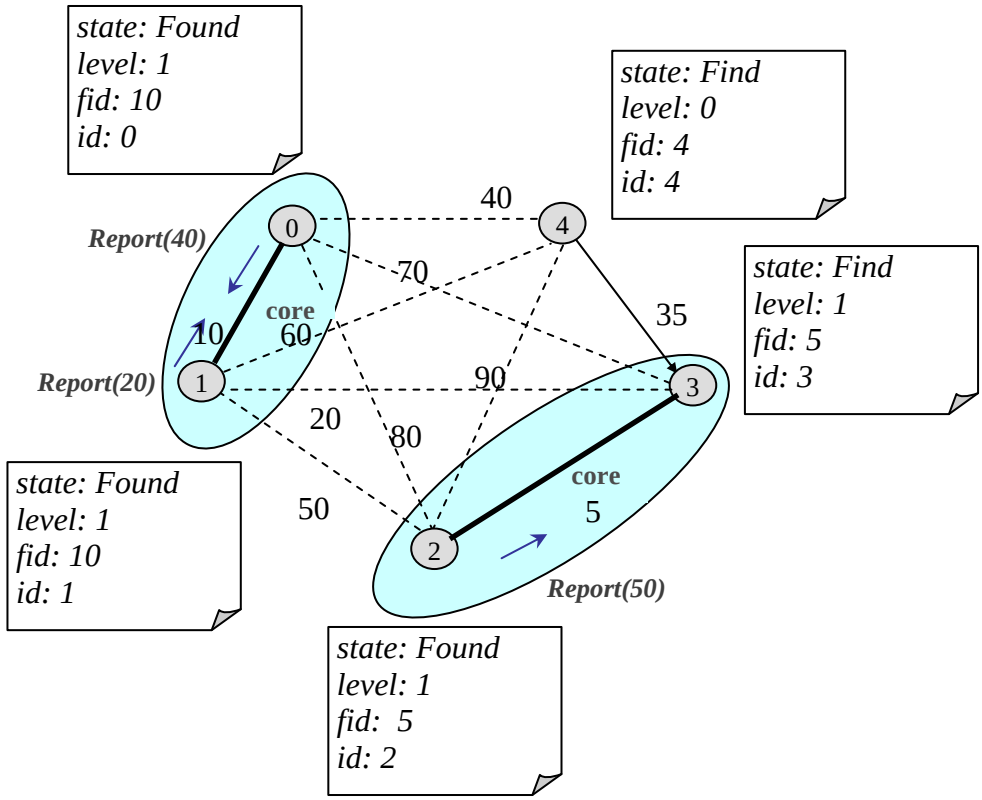


Figura 2.17: Após a execução do procedimento *Report* e envio da mensagem *Report (best-wt)* para o processo pai.

Mensagem Report(w). Quando um processo recebe uma mensagem *Report(w)*, deve-se verificar a origem da mensagem.

Caso a mensagem recebida tenha sido enviada por um processo que não seja o pai, ou seja, a identificação do processo que originou a mensagem é diferente de *in-branch*, os seguintes passos devem ser realizados:

1. Decrementar o valor da variável *find-count*, pois uma das mensagens *Report* esperada

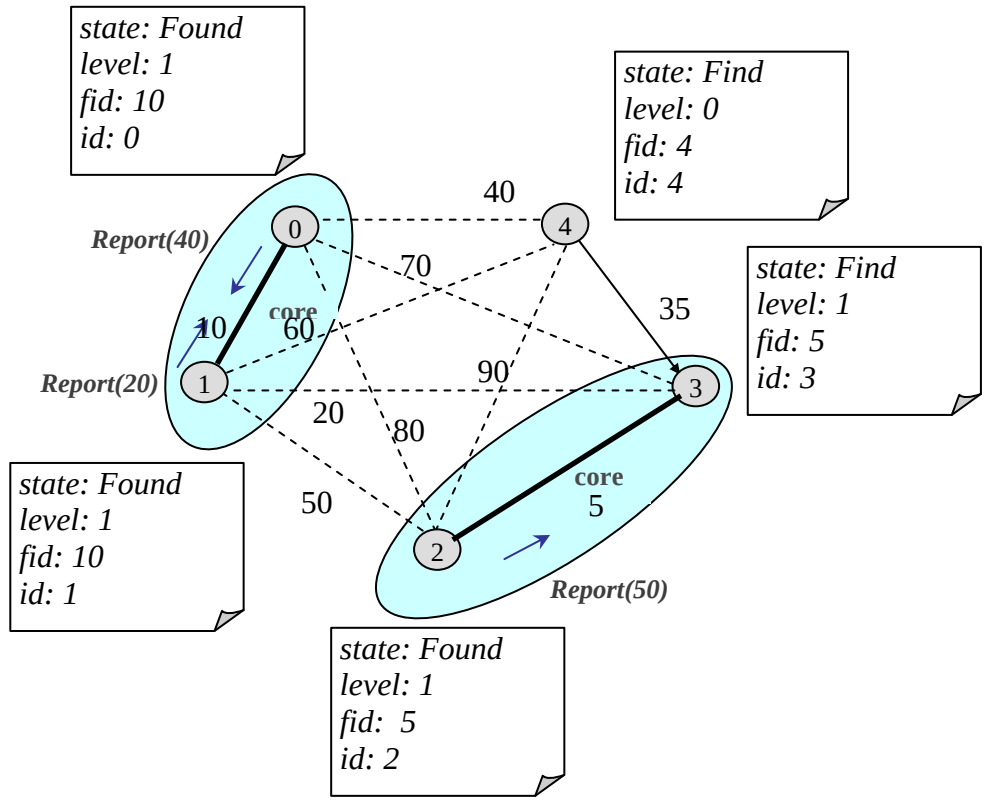


Figura 2.18: Após a execução do procedimento *Report* e envio da mensagem *Report (best-wt)* para o processo pai.

já foi recebida;

2. Comparar o valor, w, que veio junto à mensagem com o valor do atributo *best-wt* armazenado localmente. Caso o valor recebido seja menor, as variáveis *best-wt* e *best-edge* devem ser atualizadas, *best-wt* recebe o valor w e *best-edge* é atualizado com a identificação do processo que enviou a mensagem *Report(w)*. Este procedimento garante que a aresta incluída no fragmento sempre será a de menor custo e *best-edge* manterá o caminho para se alcançar esta aresta;

3. Por último, o procedimento *Report*, explicado anteriormente, deve ser executado.

Se a mensagem *Report(w)* foi enviada pelo processo pai, significa que a mensagem chegou através da aresta *core*, neste caso, os passos a seguir são:

1. Verificar o estado do processo, se for igual a *Find*, a mensagem é colocada na fila de mensagens para posterior processamento. Isso acontece porque o nó precisa esperar receber todas as mensagens *Report(w)* que necessita, antes de enviar mensagem *Report(best-wt)* pela aresta *core*;

2. Se o estado do processo for diferente de *Find*, o valor recebido pela mensagem e o armazenado localmente *best-wt* são comparados.

   Caso o valor recebido w seja igual a *best-wt* e igual a ∞, significa que não há mais arestas que possam ser incluídas nesta árvore. Neste momento, o processo de menor identificação que compõe a aresta *core*, é armazenado no atributo *root*, definindo assim a raiz da árvore gerada. O estado do processo passa a ser *Sleeping* indicando que já encerrou a sua participação no algoritmo e a terminação do algoritmo pode ser realizada através da disseminação da mensagem *Gosleep(root)* pelas arestas *Branch*.

   Senão, é verificado se o peso recebido w é maior que *best-wt*, neste caso o procedimento *change-core* deve ser executado. Note que se w for menor nada, deve ser feito, pois o outro nó que compõe a aresta *core* está cuidando deste tratamento, uma vez que estes dois processos trocaram mensagens *Report* e estão executando os mesmos passos.

No momento em que os dois nós que compõem a aresta *core* trocarem mensagems *Report(w)*, é decidido qual é a melhor aresta que fará parte da árvore.

Procedimento change-core. Este procedimento verifica se a aresta adjacente com a identificação armazenada pela variável *best-edge* já faz parte da árvore, ou seja, se está marcada como *Branch*. Caso já faça parte, a mensagem *Change-core* é enviada através dela. Se a aresta *best-edge* encontra-se no estado *Basic*, a mensagem a ser enviada através dela é *Connect(level)*. No momento do envio da mensagem *Connect(level)*, esta aresta é incluída, portanto seu estado passa a ser *Branch*.

Mensagem Change-core. Ao receber mensagem *Change-core*, o processo executa o procedimento *change-core*. Este procedimento é repetido, sucessivamente pelos outros processos, até alcançar a aresta a ser incluída.

Continuando a explicação da execução proposta, a Figura 2.19 mostra o envio da mensagem *Connect(level)* com a inclusão da aresta de custo mínimo e construção de um novo fragmento através do envio da mensagem *Initiate(level, id, state)*.

Após receber a mensagem *Connect(1)*, processos respondem com o envio da mensagem *Initiate(2, 20, Find)*, o que pode ser visto na Figura 2.20.

Ao receber a mensagem *Initiate(2, 20, Find)*, os processos atualzam seus atributos e repassam esta mensagem pelas suas arestas *Branch*, com exceção da aresta pela qual recebeu esta mensagem, como mostra a Figura 2.21.

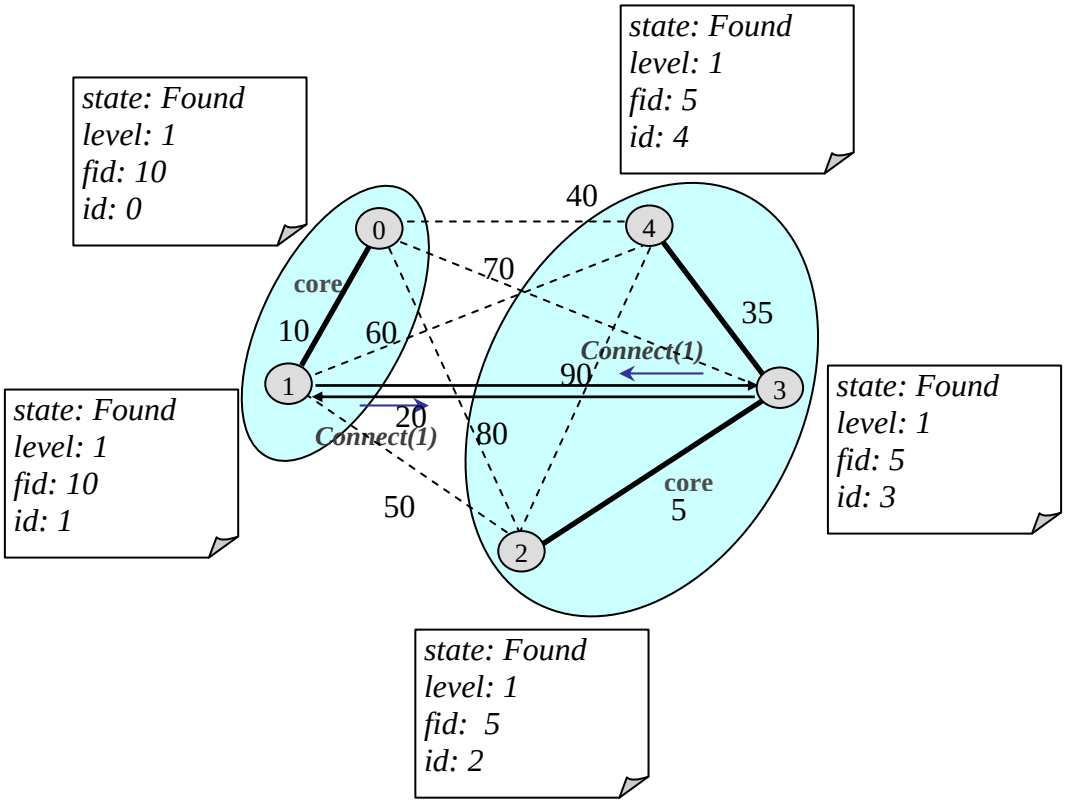


Figura 2.19: Envio da mensagem *Connect(level)*

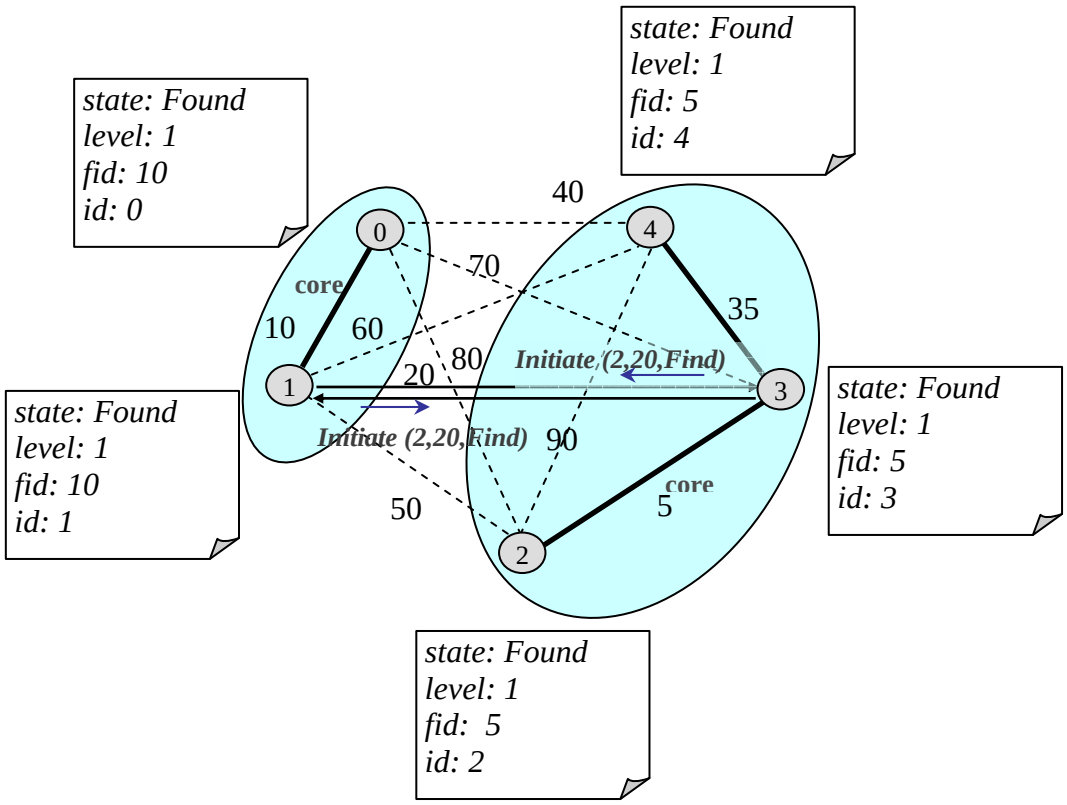


Figura 2.20: Envio da mensagem *Initiate(2, 20, Find)*.

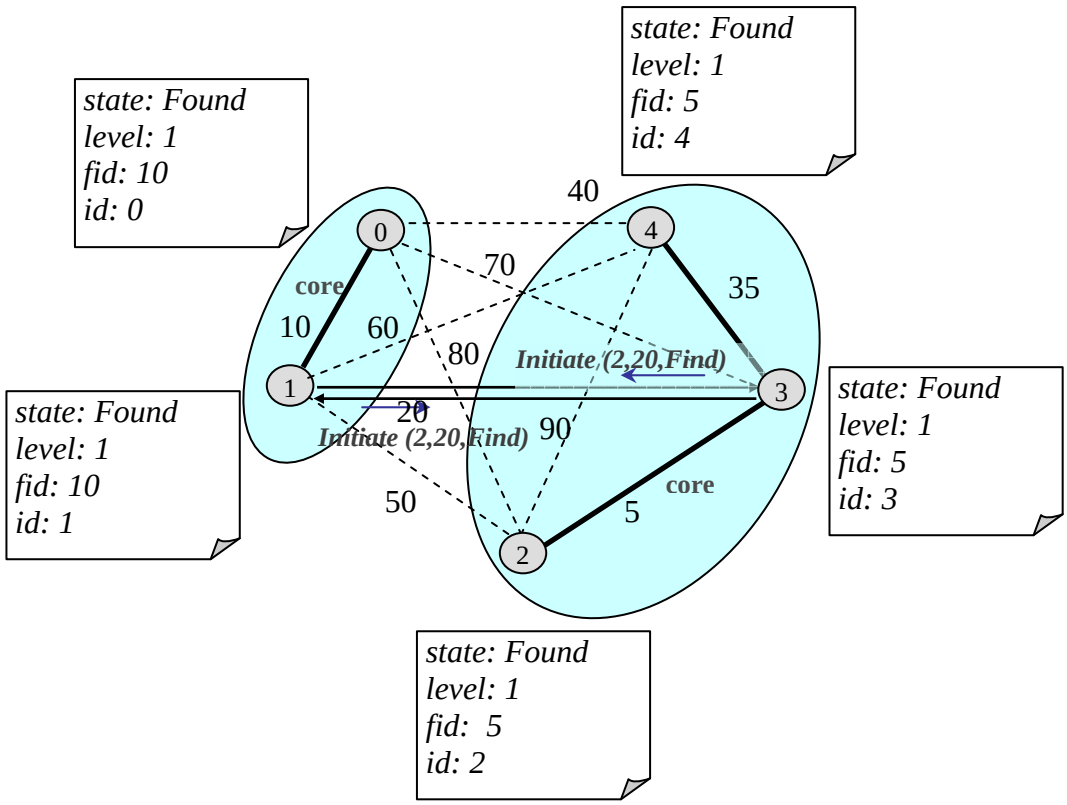


Figura 2.21: Repasse da mensagem *Initiate(2, 20, Find)*.

O recebimento desta mensagem *Initiate(2, 20, Find)* desencadea um novo ciclo de troca de mensagens, pois como cada processo atualiza seu estado para *Find*, novas mensagens *Test(level,fid)* são enviadas e, conseqüentemente, outras mensagens, *Reject* e *Report*, são trocadas, como podem ser vistos nos exemplos a seguir. Na Figura 2.22 pode ser visto o envio das mensagens *Test(level,fid)*.

Cada processo, ao receber a mensagem *Test(L,F)*, verifica a identificação do seu fragmento com a identificação de fragmento recebido pela mensagem. Em todos os casos as identificações são iguais, indicando que já pertencem ao mesmo fragmento. Portanto, estas arestas não podem fazer parte desta árvore e mensagens *Reject* são enviadas, como mostra a Figura 2.23.

Depois que todas as arestas foram testadas, o procedimento *Report* é executado. A mensagem *Report(best-wt)* é enviada, conforme mostra a Figura 2.24, onde o valor *best-wt* é igual a ∞ porque não há mais arestas *Basic* que possam ser incluídas. Neste momento a mensagem *Report(best-wt)* é enviada pela aresta *core*.

Quando processos da aresta *core* trocam mensagens *Report(∞)* significa que não há mais arestas a incluir na árvore e que o fragmento gerado é a árvore geradora de custo mínimo. A partir deste momento, o algoritmo informa o término da construção da árvore

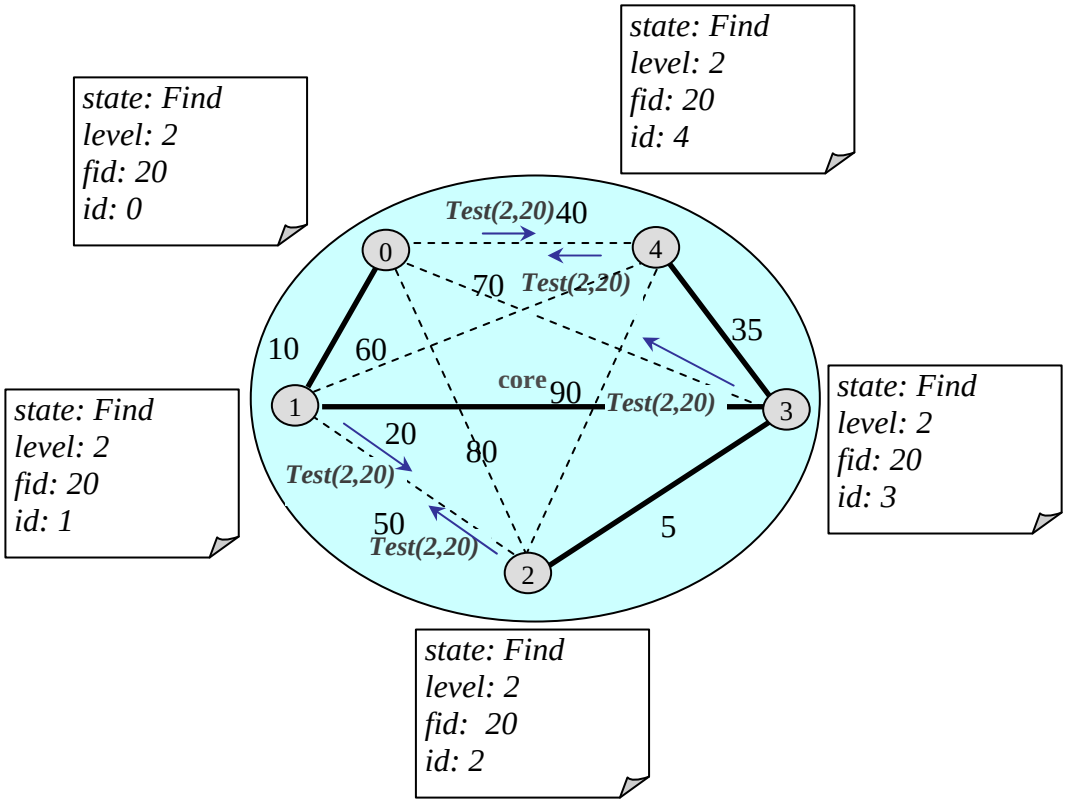


Figura 2.22: Envio da mensagem *Test(2, 20)*.

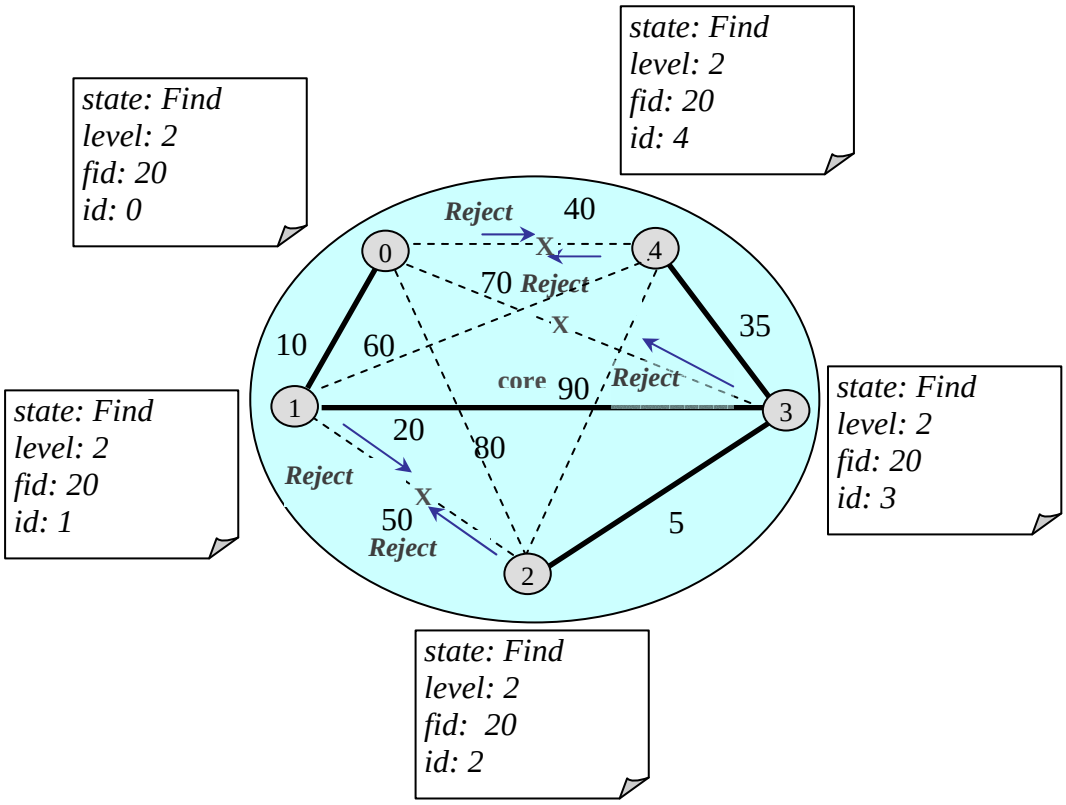


Figura 2.23: Envio da mensagem *Reject*.

através do envio da mensagem *Gosleep* da raiz para os processos filhos.

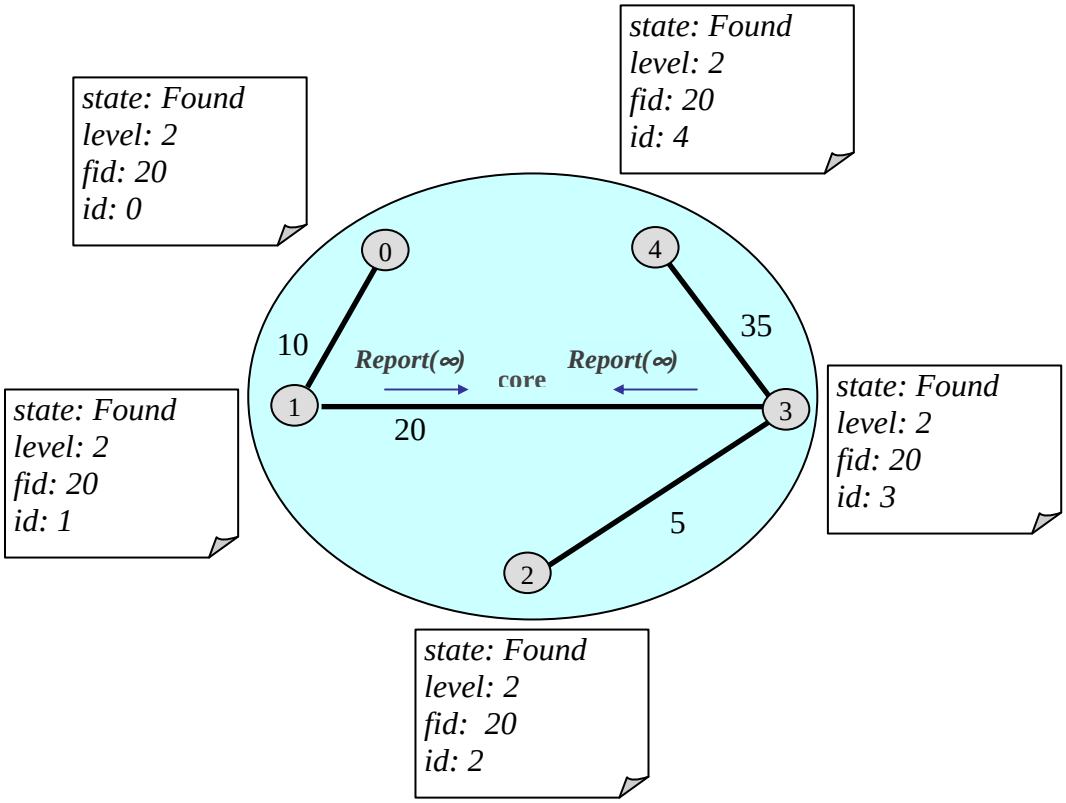


Figura 2.24: Troca de mensagens *Report* pela aresta *core*.

Mensagem Gosleep(R). Um processo, ao receber mensagem *Gosleep*, atualiza seu estado para *Sleeping*; armazena a informação R no atributo *root*, que diz quem é a raiz da árvore gerada e repassa esta mensagem para as demais arestas *Branch*. O algoritmo termina após todos os processos terem recebido a mensagem *Gosleep*. O procedimento de terminação da construção da AGM pode ser visualizado através da Figura 2.25.

2.2 Adaptação da Árvore Geradora Mínima

O desenvolvimento do algoritmo para adaptação da árvore geradora mínima distribuída baseou-se na proposta do artigo [Cheng et al. 1988]. O objetivo deste algoritmo é realizar a atualização da árvore geradora mínima em uma rede com mudanças topológicas sem precisar executar novamente o algoritmo GHS. Esse algoritmo pode responder a múltiplas falhas e recuperações de canais.

O artigo não apresentou detalhes sobre como deveriam ser realizadas as trocas de mensagens para a adaptação da árvore e nem como deveriam ser utilizados os protocolos fornecidos pelo algoritmo GHS. Em relação ao tratamento de recuperações utilizamos

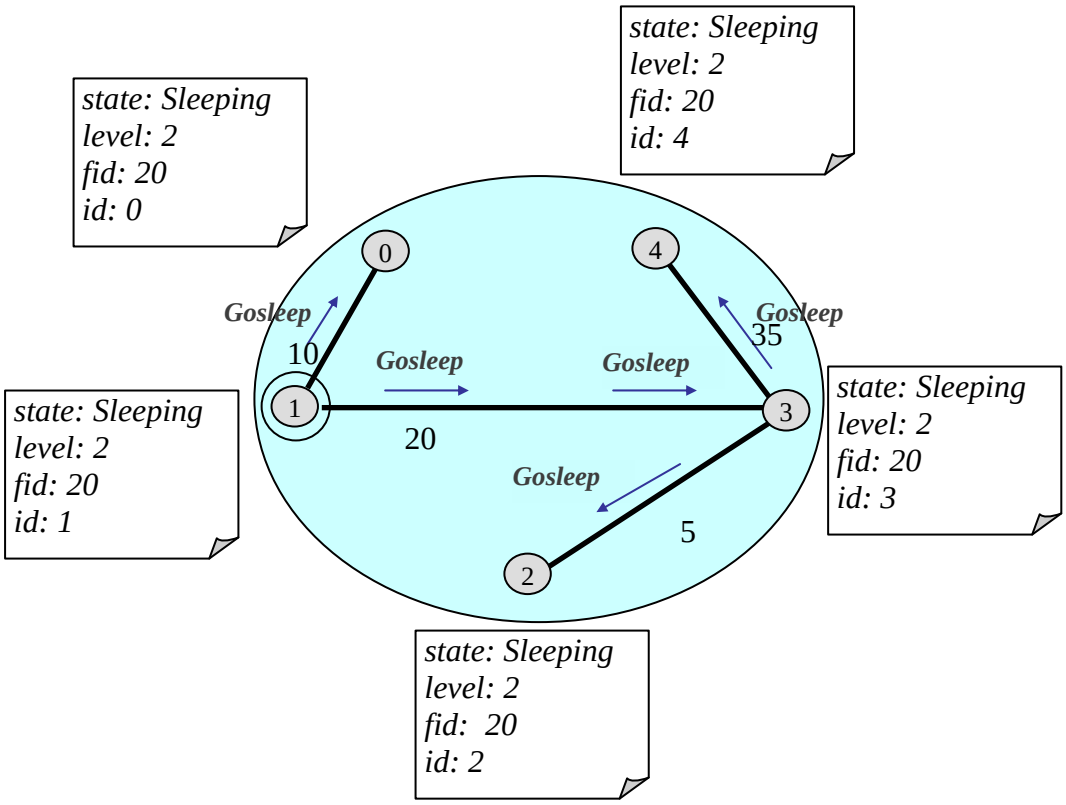


Figura 2.25: Resposta ao recebimento de uma mensagem *Gosleep*.

listas para armazenar as mensagens trocadas. Quanto ao tratamento de falhas não foi mencionado no artigo como deveria ser realizado o cálculo dos níveis dos fragmentos resultantes. Outra característica presente no algoritmo de adaptação é a sua integração com a ferramenta de monitoração de rede. Durante a explicação deste algoritmo podem ser vistos alguns detalhes da implementação.

2.2.1 Definições

A seguir são apresentados os possíveis estados, armazenados no atributo *state*, assumidos pelos nós do fragmento:

Sleeping: nó não está participando de nenhum tratamento de falha e nem de nenhuma recuperação;

Find: nó está procurando a aresta de custo mínimo a ser incluída na árvore durante o tratamento de uma falha;

Found: nó já achou a sua aresta de custo mínimo candidata, também durante o tratamento de uma falha;

Reiden: nó está disseminando a nova identificação do fragmento logo após a verificação da ocorrência de uma falha. Quando uma falha ocorre em um canal pertencente à árvore, dois fragmentos passam a existir, torna-se necessário reidentificá-los para que as trocas de mensagens possam ser executadas para unir os fragmentos novamente.

Recover: nó está executanto o tratamento de uma recuperação de canal que não fazia parte da árvore.

Da mesma forma que o algoritmo para a construção da árvore geradora mínima, as arestas adjacentes a cada nó podem assumir três estados:

***Branch*:** aresta já faz parte da árvore ou acabou de ser identificada a sua recuperação;

***Rejected*:** significa que a aresta foi retirada da árvore porque sofreu uma falha ou sua participação está impedida para evitar a formação de ciclos;

***Basic*:** aresta não é *Branch* nem *Rejected*, portanto está livre para ser usada.

Os tipos de mensagens trocadas durante o processo de adaptação da árvore são:

- Para falhas: *Failure*, *Reiden*, *Reiden-ack*, *Connect*, *Initiate*, *Findmoe*, *Test*, *Accept*, *Reject*, *Report* e *Change_ core*;
- Para recuperações: *Id-check*, *Recovery*, *Privilege* e *Replace*;
- Para terminação dos tratamentos de falhas, recuperações e do algoritmo de adaptação: *Not-fail*, *Not-fail-ack*, *Gosleep-fal*, *Not-rec*, *Not-rec-ack* e *Gosleep-rec*;

Os atributos *fid*, *root*, *state*, *level*, *best-edge*, *best-wt*, *test-edge* e *in-branch* utilizados por cada processo durante a construção da árvore, também são importantes para a adaptação e têm a mesma finalidade.

2.2.2 Descrição do Algoritmo

Neste trabalho consideramos como falha uma significativa piora no desempenho do canal e, reciprocamente, como recuperação uma sensível melhoria no desempenho do mesmo. Dado um número finito de mudanças topológicas ocorridas durante um período, o algoritmo encontra a árvore geradora mínima correspondente às últimas condições da rede.

Para executar a adaptação da árvore é preciso fazer a monitoração de todos os canais para coletar os novos custos. Após esta coleta, a classificação em falhas, recuperações e a atualização dos canais pode ser realizada. O aumento de custo de um canal é considerado como uma falha, quando este canal pertencer à árvore geradora mínima e, de forma similar, uma recuperação ocorre quando uma aresta não pertencente à árvore tiver o custo de seu canal diminuído. Quando um nó detecta o aumento de custo de qualquer canal adjacente que não faça parte da árvore, ou uma diminuição de custo de um canal que já faz parte da árvore, o algoritmo somente atualiza esses valores, não sendo necessário executar nenhum tipo de tratamento para falhas e/ou recuperações em relação a esses canais.

A seguir são apresentadas as mensagens trocadas para a realização da adaptação da árvore geradora mínima após detecção de falhas e recuperações e, também, para a determinação do fim da adaptação da árvore.

- Em relação à detecção de falha de um ou mais canais, os procedimentos a serem executados são:

    1. Cada processo atualiza o estado das suas arestas *Rejected* para *Basic*, permitindo que estas arestas possam ser inseridas no caso de constatação de falhas na árvore;

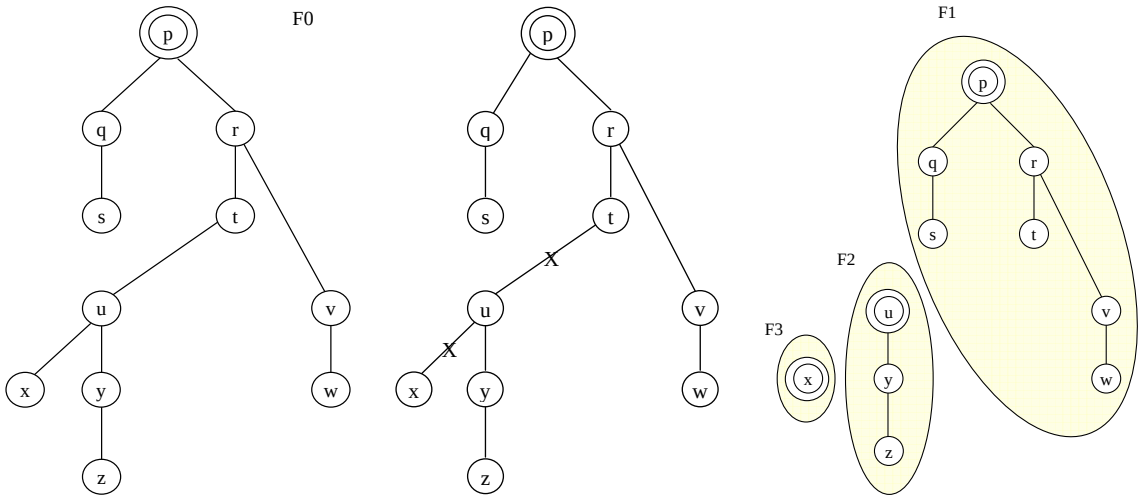
    2. Processo deve marcar a aresta que falhou como *Rejected*;

    3. Processos que identificaram a falha geram, imediatamente, uma nova identificação para os fragmentos resultantes. Cada fragmento gerado é identificado de forma única;

       Na Figura 2.26, em (a) pode ser vista uma árvore geradora mínima identificada como F_0, em (b) é apresentada a ocorrência de duas falhas nesta árvore, através do 'x' marcado sobre as arestas, e em (c) os três fragmentos resultantes (F_1, F_2 e F_3) após as falhas;

    4. Verificar se com a ocorrência da falha, um único nó ficou desconectado do resto da árvore;

       Caso isto ocorra, os seguintes passos precisam ser realizados: definir o nível (*level*) do seu fragmento como *0*; identificar a aresta de menor custo, marcá-la como *Branch* e enviar a mensagem *Connect(level)* através dela; atualizar o estado (*state*) do fragmento para *Found*. Estes passos podem ser vistos na



(a) Árvore inicial (b) Duas falhas (c) Fragmentos gerados

Figura 2.26: Árvore geradora mínima inicial e a ocorrência de duas falhas.

Figura 2.27 e são semelhantes ao que é realizado pelo procedimento *wakeup* no algoritmo para a construção da árvore.

5. Verificar se a ocorrência da falha desconectou o processo do seu pai na árvore;

   Quando isto acontece, o processo que identificou a falha torna-se raiz do fragmento, atualiza seu estado para *Reiden* e envia a mensagem *Reiden(level+1, fid)* pelas arestas *Branch* como mostrado na Figura 2.28. A mensagem *Reiden(level+1, fid)* precisa ser enviada pelos ramos do fragmento resultante para que todos os seus nós obtenham a nova identificação, ajudem na determinação do nível do fragmento e cooperem na reconstrução da árvore geradora mínima.

   Note que durante o envio da mensagem *Reiden(level+1, fid)*, além da nova identificação do fragmento (*fid*), o nível é passado com incremento de 1. Este procedimento faz parte da técnica adotada para a determinação do nível do fragmento durante a sua reidentificação. O cálculo da nova identificação do fragmento (*fid*) é realizado através da combinação da identificação do nó que verificou a ocorrência da falha com o custo da aresta que falhou, isto garante que a nova identificação do fragmento seja única.

6. Verificar se a falha desconectou o processo de algum ramo ou filho na árvore e uma mensagem *Failure(fid)* deve ser enviada para o processo *in-branch*, como mostra a Figura 2.29;

Os procedimentos apresentados referem-se à notificação de ocorrência de falhas nos

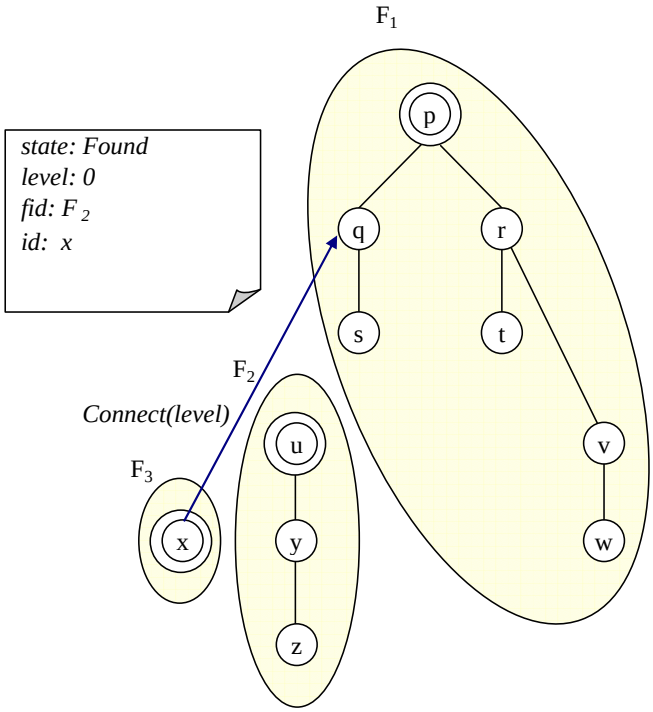


Figura 2.27: Envio da mensagem *Connect(level)*.

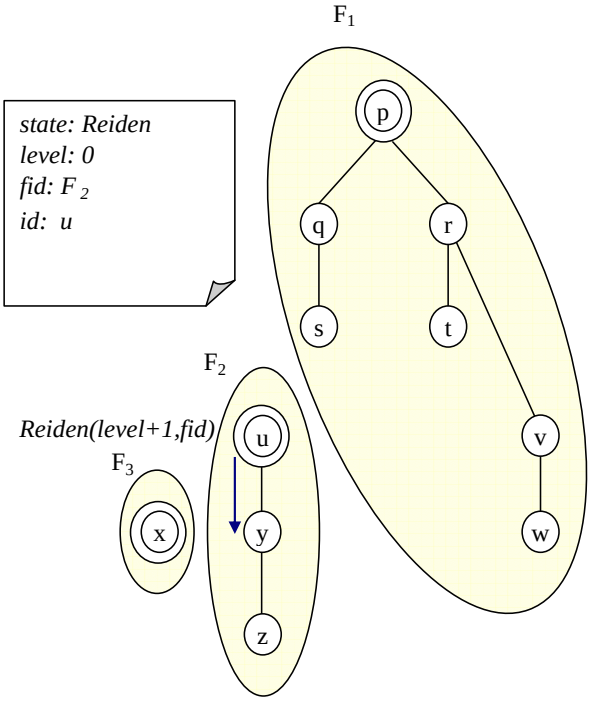


Figura 2.28: Envio da mensagem *Reiden (level+1, fid)* pela aresta *Branch*.

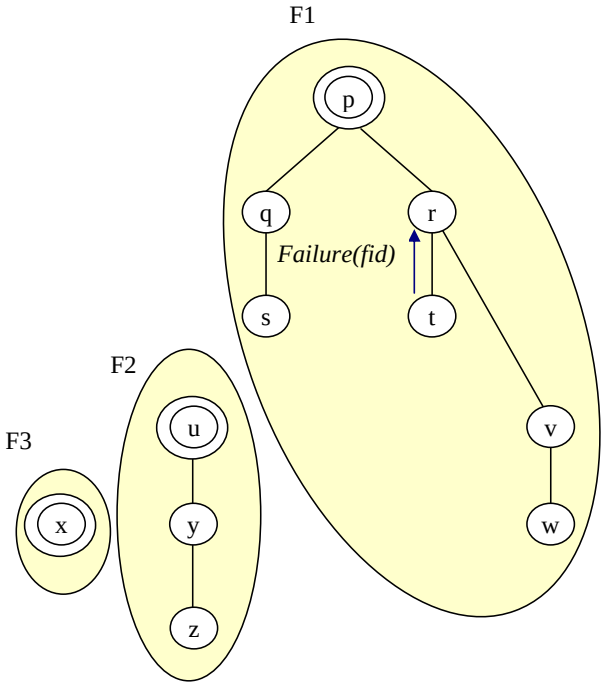


Figura 2.29: Envio da mensagem *Failure(fid)* do processo que identificou a falha para o processo pai.

canais após a realização da monitoração. Para que a reconstrução da árvore geradora mínima seja realizada com os fragmentos resultantes, utilizando praticamente os mesmos protocolos do algoritmo *GHS*, é necessário que cada processo conheça a identificação, o nível e o estado do seu novo fragmento. Os processos também devem saber a identificação do processo pai (*in-branch*) na árvore e da raiz do fragmento que pertence (*root*).

A seguir são explicadas as mensagens trocadas para que a árvore seja reconstituída após a ocorrência de uma ou mais falhas.

Mensagem Failure(F). Ao receber esta mensagem, o processo é informado que alguma falha ocorreu. Em seguida, este verifica se sua identificação é igual a *root*. Neste caso, alguns atributos do fragmento são atualizados, tais como: *level* que passa a ser zero e *state* que recebe *Reiden*. A mensagem *Reiden (level+1, fid, root)* é enviada para todos os processos filhos na árvore, como mostrado pela Figura 2.30. Se a identificação do processo for diferente de *root*, a mensagem *Failure(F)* recebida é repassada para o processo pai. Na Figura 2.31, o processo q recebeu mensagem *Failure(F)* e não é a raiz do seu fragmento. Neste caso, este processo repassa a mensagem recebida para o seu pai na árvore (p).

Mensagem Reiden (L,F,R). O objetivo desta mensagem é fazer com que o pro-

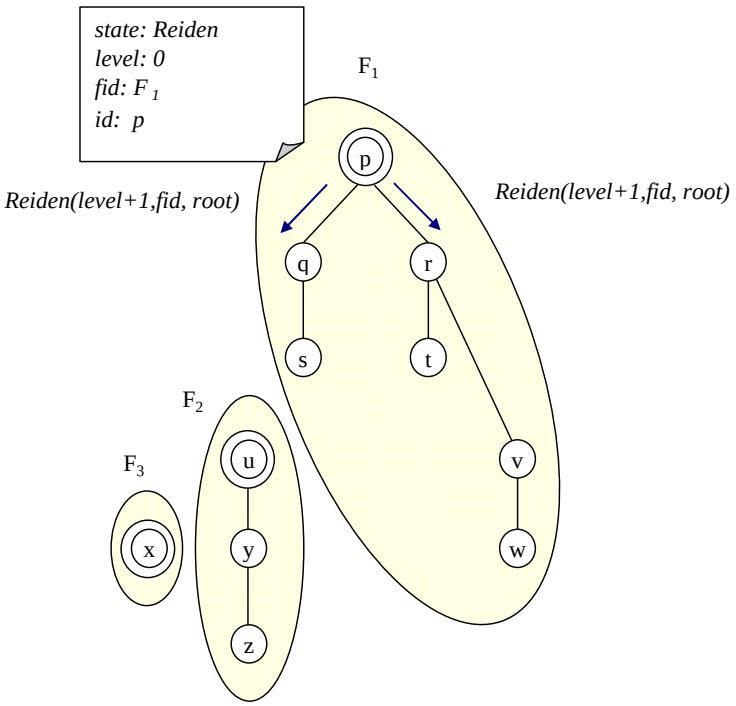


Figura 2.30: Envio da mensagem *Reiden(level+1,fid)*.

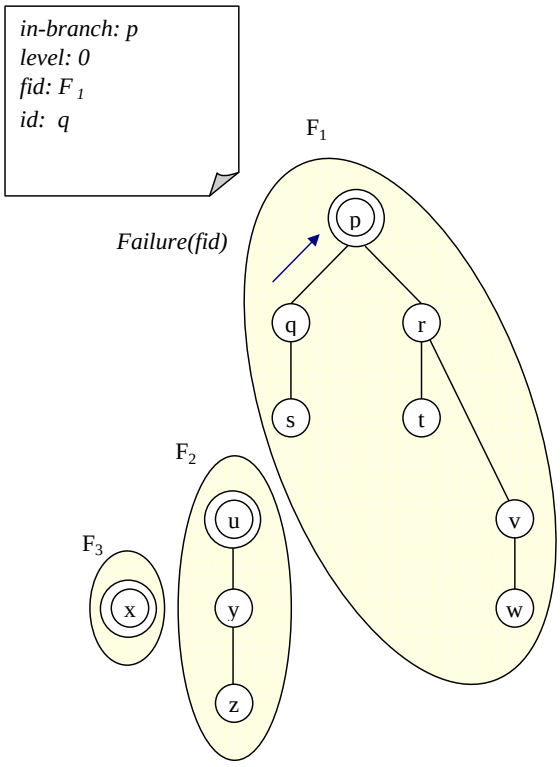


Figura 2.31: Envio da mensagem *Failure(fid)* para o processo pai.

cesso obtenha a nova identificação do fragmento.

Ao receber a mensagem *Reiden (L,F,R)*, o processo entra no estado *Reiden* e atualiza alguns atributos de acordo com aqueles recebidos pela mensagem - *level, fid e root*. Em seguida, o processo verifica se possui filhos na árvore e, neste caso, a mensagem *Reiden (level+1, fid, root)* é repassada para eles, como apresentado na Figura 2.32. Nota-se mais uma vez que o parâmetro com o nível do fragmento é repassado com o incremento de 1.

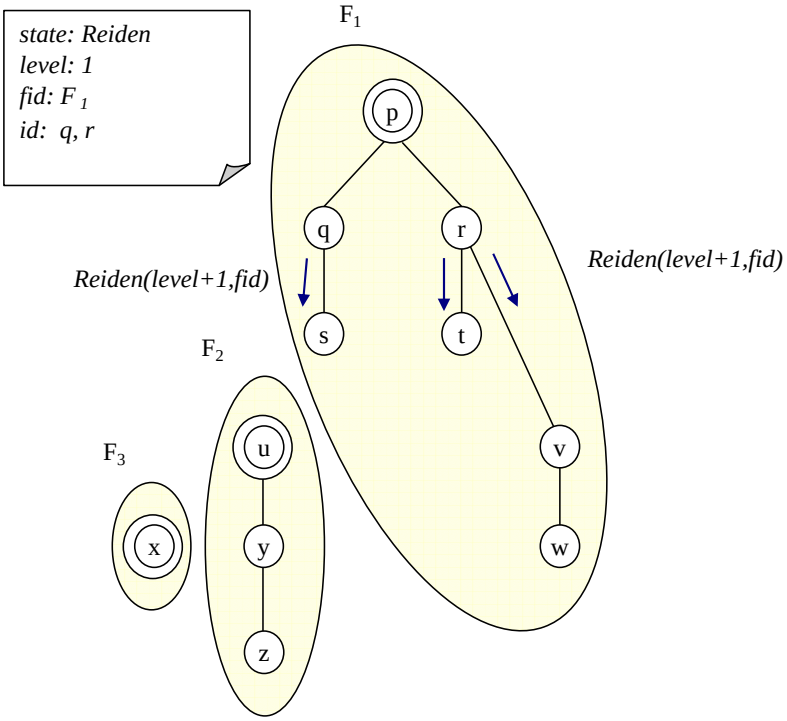


Figura 2.32: Processo repassa a mensagem *Reiden (level+1, fid, root)* para seus filhos.

Se o processo não possui mais filhos, ou seja, é um nó folha na árvore, a mensagem a ser enviada será a *Reiden-ack (level, fid)* para o processo pai, como pode ser observado através da Figura 2.33.

Uma observação importante é que sempre que o processo envia uma mensagem *Reiden* por uma aresta *Branch*, ele incrementa o valor do seu atributo *reiden_ count* de 1. Este atributo determina em que momento a mensagem *Reiden-ack (level)* deverá ser enviada para o processo pai.

Reiden-ack (L, F). O objetivo desta mensagem é garantir que a nova identificação do fragmento obtida continua valendo para prosseguir no tratamento da falha. O processo ao receber esta mensagem verifica se a identificação de fragmento recebida através da mensagem é igual à identificação do seu próprio fragmento. Se for, o processo compara o nível recebido (L) com o nível armazenado (*level*), caso o

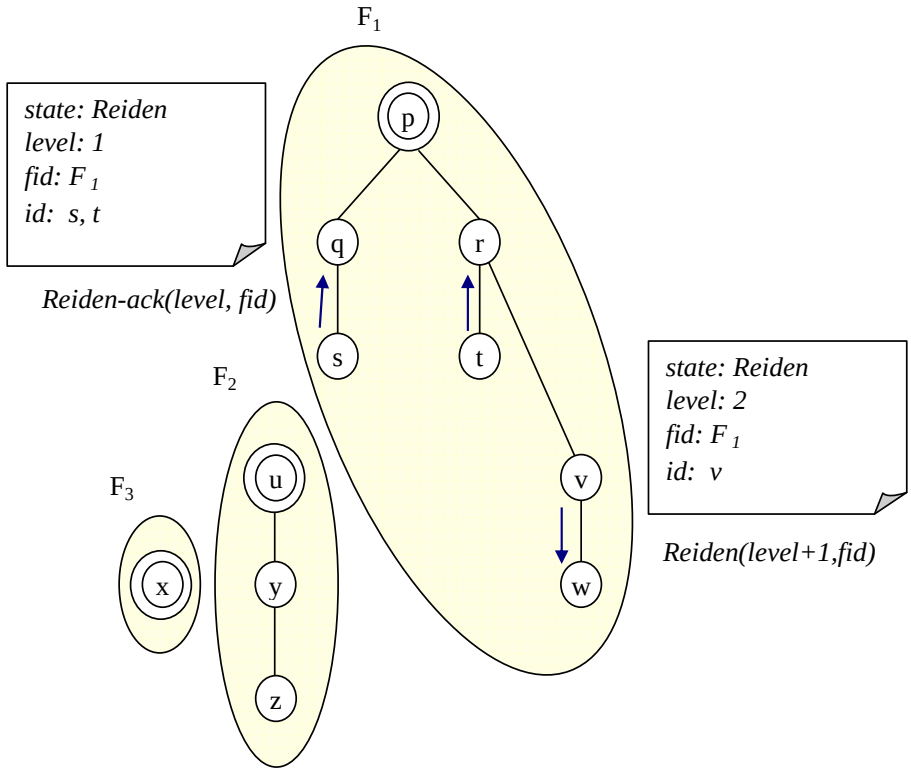


Figura 2.33: Envio da mensagem *Reiden-ack (level, fid)* .

nível recebido seja maior, *level* é atualizado com o valor de *L*. Se as identificações forem diferentes significa que outra falha foi detectada e o fragmento está sendo recebendo outra identificação novamente, portanto, nada deve ser feito em relação ao recebimento desta mensagem.

Se processo já recebeu todas as mensagens *Reiden-ack (L,F)* que esperava, através da análise do atributo *reiden_ count*, ele verifica se é a raiz do fragmento. Caso o processo não seja a raiz da árvore, a sua função é enviar a mensagem *Reiden-ack (level,fid)* para o processo pai que pode ser acompanhado pela Figura 2.34. Se for a raiz, os seguintes procedimentos são realizados pelo processo: atualização de seu estado para *Find*, atualização do seu nível (*level*) com o maior valor recebido para L, envio da mensagem *Findmoe (level, fid)* por todos os seus *ramos* e execução do procedimento *Test* que é o mesmo procedimento utilizado na construção da árvore geradora mínima. É importante que a raiz execute este procedimento para que também faça a busca pela aresta de custo mínimo do fragmento. A Figura 2.35 mostra o envio da mensagem *Findmoe(level,fid)*.

Mensagem Findmoe (L, F). Com o recebimento desta mensagem, o nó fica ciente que deve dar início à busca pela aresta de saída de custo mínimo para aumentar o tamanho do fragmento. Ao receber esta mensagem, o processo verifica se a iden-

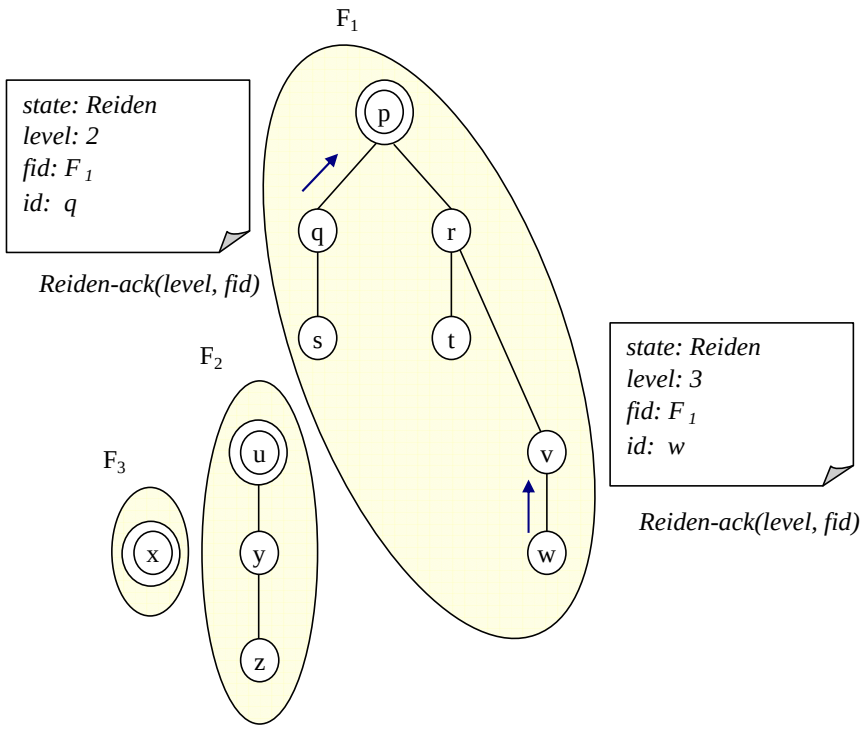


Figura 2.34: Envio da mensagem *Reiden-ack (level,fid)* para processo pai, após o recebimento da mesma.

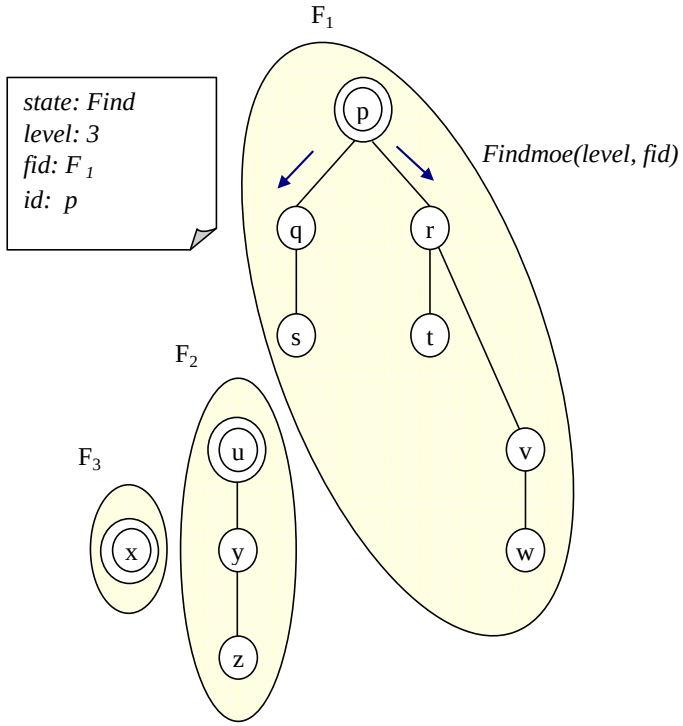


Figura 2.35: Envio da mensagem *Findmoe(level,fid)* por todas as arestas *Branch*.

tificação de fragmento recebida (F) através da mensagem é igual à identificação do seu próprio fragmento para dar continuidade a sua execução. Se as identificações forem diferentes, significa que uma nova falha foi detectada e o fragmento está sendo novamente reidentificado, nada deve ser feito neste momento.

Se as identificações forem iguais, o primeiro passo é atualizar o nível do seu fragmento de acordo com o valor (L) recebido pela mensagem, o estado do fragmento também deve ser atualizado para *Find*, indicando que a busca pela próxima aresta de custo mínimo foi iniciada. Após as atualizações desses atributos, a mensagem *Findmoe(level, fid)* é repassada pela arestas *Branch* adjacentes e o procedimento *Test* deve ser executado. Os passos descritos podem ser vistos na Figura 2.36.

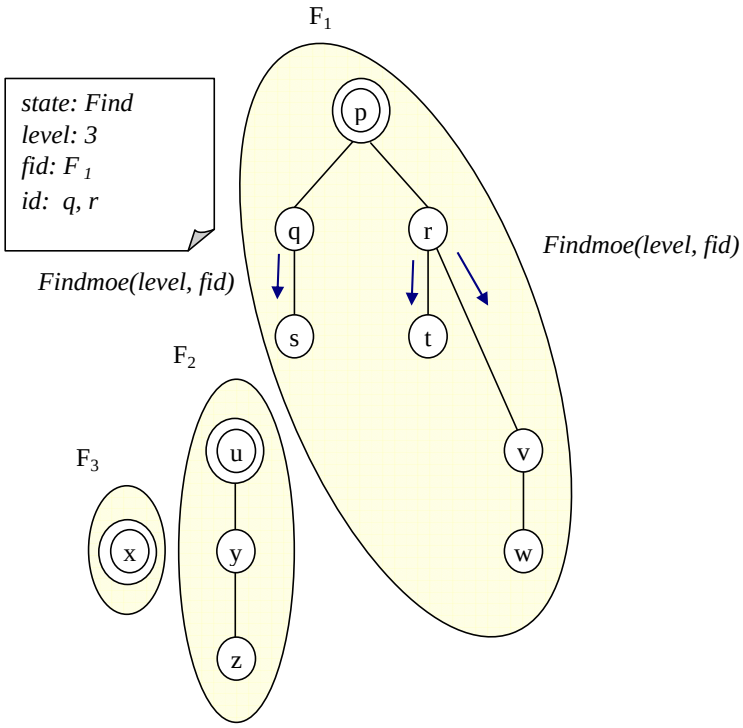


Figura 2.36: Atualização de atributos após receber mensagem *Findmoe (L,F)* e repasse desta mensagem para as demais arestas *Branch*.

Com a execução do procedimento *Test* devido ao recebimento da mensagem *Findmoe(L, F)*, mensagens *Test(level,fid)* são enviadas, como pode ser visto na Figura 2.37.

Como o procedimento *Test* é o mesmo utilizado na fase de construção da árvore geradora mínima, logo, sua função é enviar uma mensagem *Test(level,fid)* por uma aresta de saída. Da mesma forma, um processo que recebe tal mensagem, durante a fase de adaptação, pode responder com uma mensagem *Accept(fid)* ou *Reject(fid)* que indica se esta aresta pode ser considerada como uma candidata a ser incluída

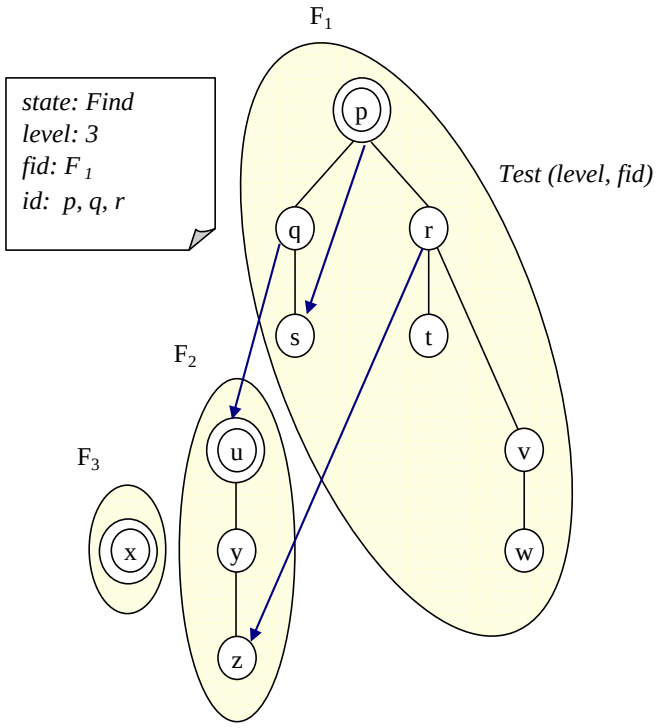


Figura 2.37: Envio de mensagem *Test* após recebimento da mensagem *Findmoe (L,F)*.

no fragmento ou não. A diferença apresentada aqui é que, neste momento, o *fid* enviado junto com a mensagem *Test(level, fid)* precisa ser devolvido nas mensagens *Accept(fid)* e *Reject(fid)* para garantir que a resposta recebida é de fato para o tratamento da mesma falha. Este cuidado é exigido porque o momento entre o envio e o recebimento de resposta da mensagem *Test(level, fid)*, uma nova falha pode ter sido detectada e, portanto, o fragmento pode estar recebendo nova identificação. Neste caso, a identificação presente na mensagem *Accept(fid)* ou *Reject(fid)* recebida é diferente daquela que foi enviada junto à mensagem *Test(level,fid)* e, portanto, deve ser desconsiderada.

Após responder a mensagem *Test*, os mesmos passos executados para a construção da árvore são realizados a partir de agora, ou seja, mensagens do tipo *Connect*, *Initiate*, *Report* são trocadas até gerar a árvore final que está representada na Figura 2.38. A adaptação termina quando cada processo verificar que não possui mais falhas a tratar e que não recebeu nenhuma mensagem que indique que ainda há falhas sendo tratadas. A partir deste momento, é iniciada a terminação do tratamento de falhas que está explicada na Seção 2.3.

- Em relação à detecção de recuperação de um ou mais canais, o processo executa os seguintes passos:

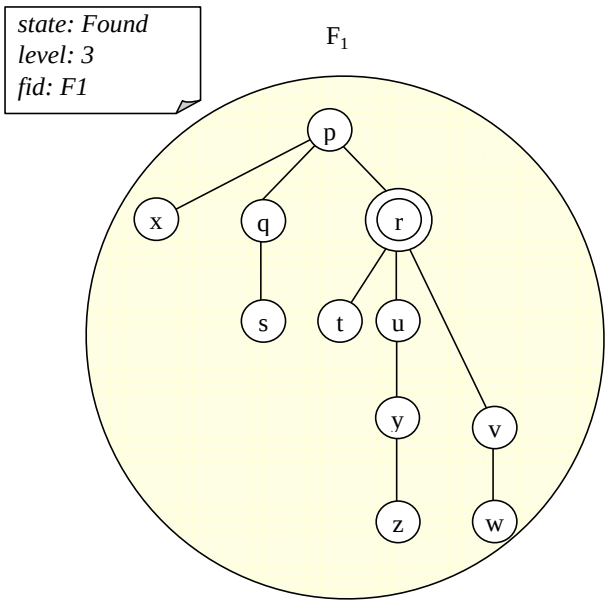


Figura 2.38: Árvore gerada após tratamento de falhas.

1. Verifica se já terminou a sua participação no tratamento de falhas. Caso ainda esteja finalizando sua participação neste tratamento, qualquer mensagem recebida em relação à recuperação de canal deve ser colocada numa fila. Esta mensagem só é avaliada após o término do tratamento de falhas.

2. Marca a aresta recuperada como *Branch*;

3. Após identificar a ocorrência de uma recuperação, o processo envia a mensagem *Id-check(fid)* para o outro nó que compõe a aresta recuperada.

Os procedimentos descritos anteriormente referem-se à notificação da ocorrência de recuperações. Em seguida, estão descritas as demais mensagens trocadas para tratá-las. O algoritmo de adaptação é capaz de detectar o ciclo formado após a inclusão do canal recuperado e decidir qual canal deve ser retirado. Naturalmente, o canal que deixa de participar da árvore é aquele que possuir o maior custo no ciclo.

A seguir, na Figura 2.39, é apresentado um exemplo de uma árvore que teve duas arestas recuperadas representadas por linhas tracejadas. Após a notificação da recuperação pelos nós adjacentes à aresta recuperada, mensagens *Id-check(fid)* serão trocadas como mostra a Figura 2.40.

Mensagem Id-check. O objetivo desta mensagem é informar ao processo que ele deve iniciar a sua participação no tratamento da recuperação de uma aresta adjacente a ele. Ao receber a mensagem *Id-check*, o processo verifica se já está par-

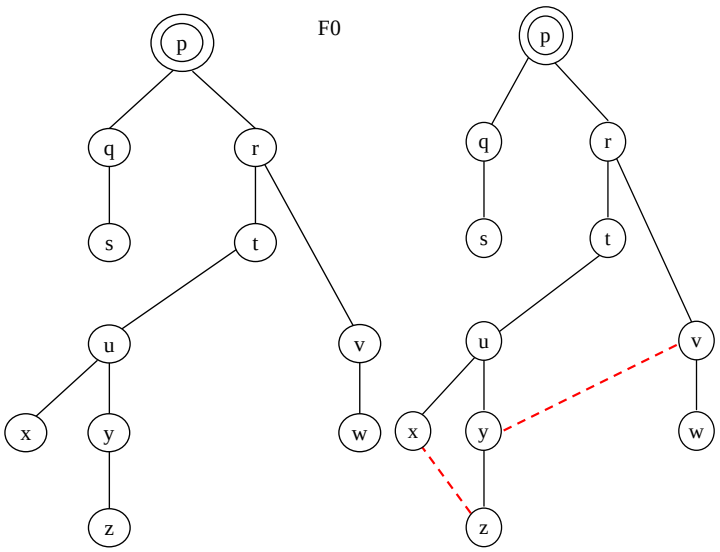


Figura 2.39: Árvore geradora mínima inicial e a ocorrência de duas recuperações.

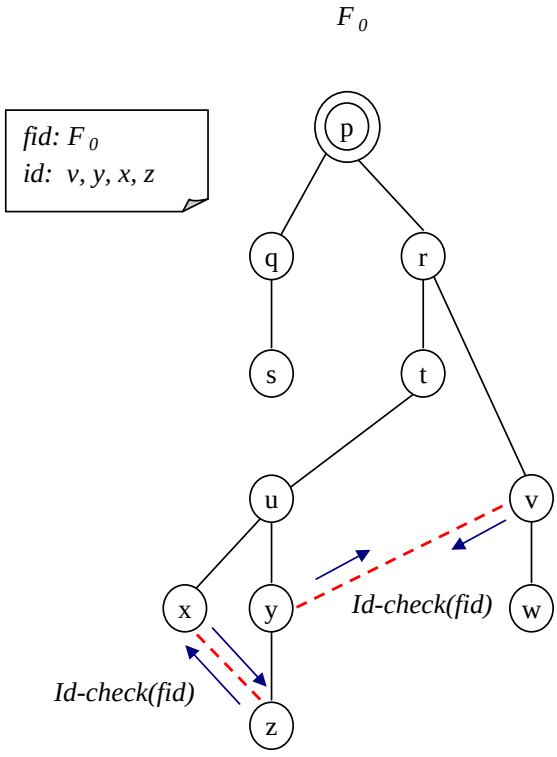


Figura 2.40: Envio de mensagens *Id-check(fid)*.

ticipando de algum tratamento de recuperação. Se isto for verdade, uma mensagem de recuperação é colocada em uma lista para posterior processamento.

Caso o processo não esteja tratando de nenhuma recuperação, ele atualiza seu estado para *Recover* e verifica se é a raiz do fragmento, comparando sua identificação com o atributo *root*. Se não for, o processo deve enviar a mensagem *Recovery(we, wp)* para o processo pai (*in-branch*). A mensagem *Recovery (we, wp)* leva as informações dos custos da aresta recuperada (*we*) e da aresta de maior custo no ciclo (*wp*) que, neste momento, recebe o maior valor entre os custos da aresta *we* e da aresta que liga ao processo pai. Na Figura 2.41 pode ser visto o envio desta mensagem *Recovery(we,wp)* e uma lista com as recuperações que este processo está tratando, a lista apresentada na figura mostra apenas a informação da aresta recuperada.

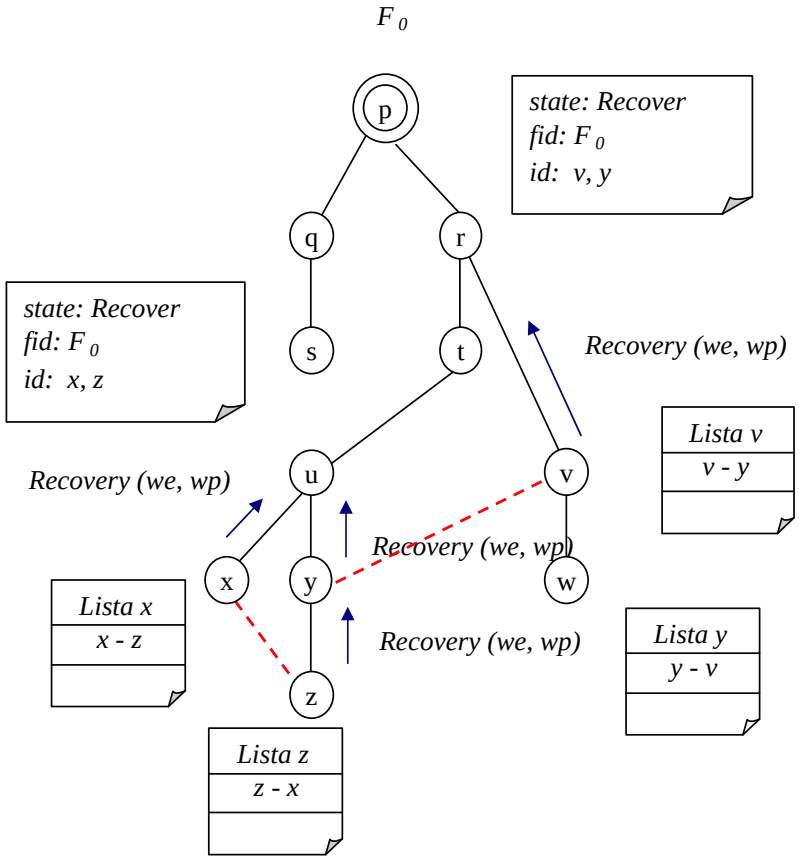


Figura 2.41: Envio da mensagem *Recovery (we,wp)* para o processo pai.

Caso o processo seja a raiz do fragmento, uma mensagem *Privilege(we,wp)* é gerada e somente a raiz mantém uma fila adicional para este tipo de mensagem, que é responsável pela serialização do tratamento de recuperações. Quando a raiz recebe a mensagem *Id-check* e não está participando de nenhuma recuperação, além de colocar uma mensagem *Recovery(we,wp)* na sua lista de recuperações e uma mensagem *Privilege(we,wp)* na fila, ela envia a mensagem *Privilege(we,wp)* para o processo que

lhe mandou a mensagem *Id-check*, iniciando imediatamente o tratamento desta recuperação. Se já estiver tratando de outra recuperacão, a mensagem *Privilege(we,wp)* fica na fila aguardando sua liberação.

Mensagem Recovery (W, W'). Ao receber esta mensagem, o processo sabe que precisa participar do tratamento da recuperação da aresta de custo W.

O primeiro passo, ao receber esta mensagem, é armazená-la na sua lista de recuperações. Em seguida, deve verficar se o processo é a raiz do fragmento.

Caso o processo não seja a raiz do fragmento e já esteja participando de algum tratamento de recuperação, ele deve segurar a mensagem recebida, colocando-a na sua lista de recuperações para posterior processamento. Isto pode ser visto pela Figura 2.42, onde os processos u e y não repassam as mensagens de recuperações recebidas, por estarem tratando de outra recuperação. No caso, u já iniciou o tratamento da aresta (x,z) e y o tratamento da aresta (y,v).

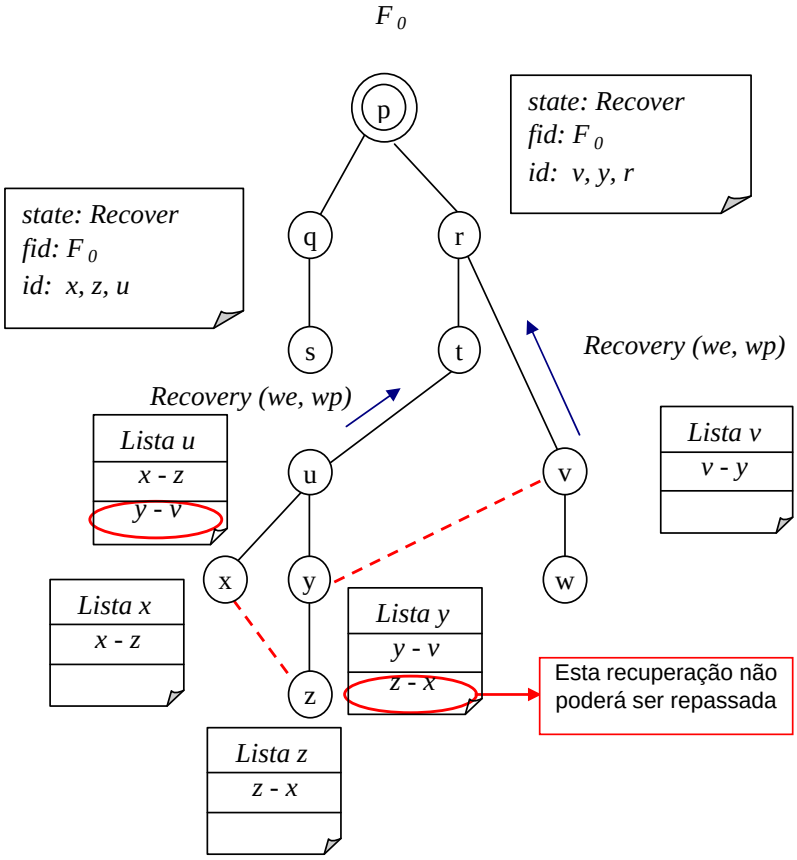


Figura 2.42: Mensagem *Recovery (we,wp)* não repassada.

Se o processo não está tratando de nenhuma recuperação, o seu estado deve ser atualizado para *Recover* e a mensagem *Recovery(we,wp)* que chegou primeiro deve ser repassada. Caso o custo da aresta que liga ao processo pai seja maior que

o custo de *wp*, então *wp* é atualizado com este valor. O repasse da mensagem *Recovery(we,wp)* pode ser visto na Figura 2.43, quando o processo t recebe de u a informação sobre a recuperação (x,z) e o processo r recebe de v a informação sobre a recuperação (v,y).

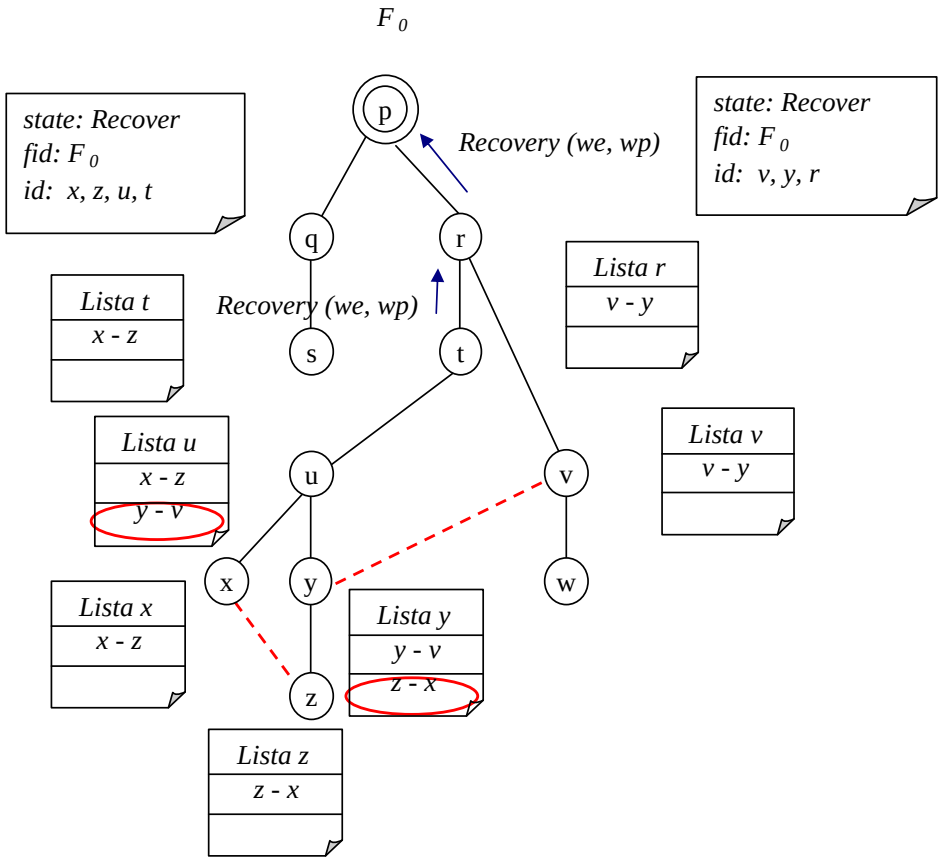


Figura 2.43: Repasse da mensagem *Recovery (we,wp)* para o processo pai.

Caso o processo seja a raiz do fragmento, a mensagem *Privilege(we,wp)* deve ser criada e colocada na fila de mensagens *Privilege* mantida pela raiz, como já foi visto anteriormente. O próximo passo é verificar se o processo já está participando de algum tratamento de recuperação, neste caso, nenhuma mensagem é enviada. Caso contrário, a mensagem *Privilege (we,wp)* deve ser enviada para o processo que lhe enviou a mensagem *Recovery (W,W')*, onde *we* terá o custo da aresta recuperada e *wp* o custo de maior valor encontrado até o momento. A Figura 2.44 apresenta a realização desses passos.

Mensagem Privilege (W, W'). O objetivo desta mensagem é serializar o tratamento de recuperações para evitar possíveis situações de *deadlock*.

Quando várias recuperações ocorrem simultaneamente em um fragmento e os ciclos formados são disjuntos, o tratamento das recuperações pode ser executado de

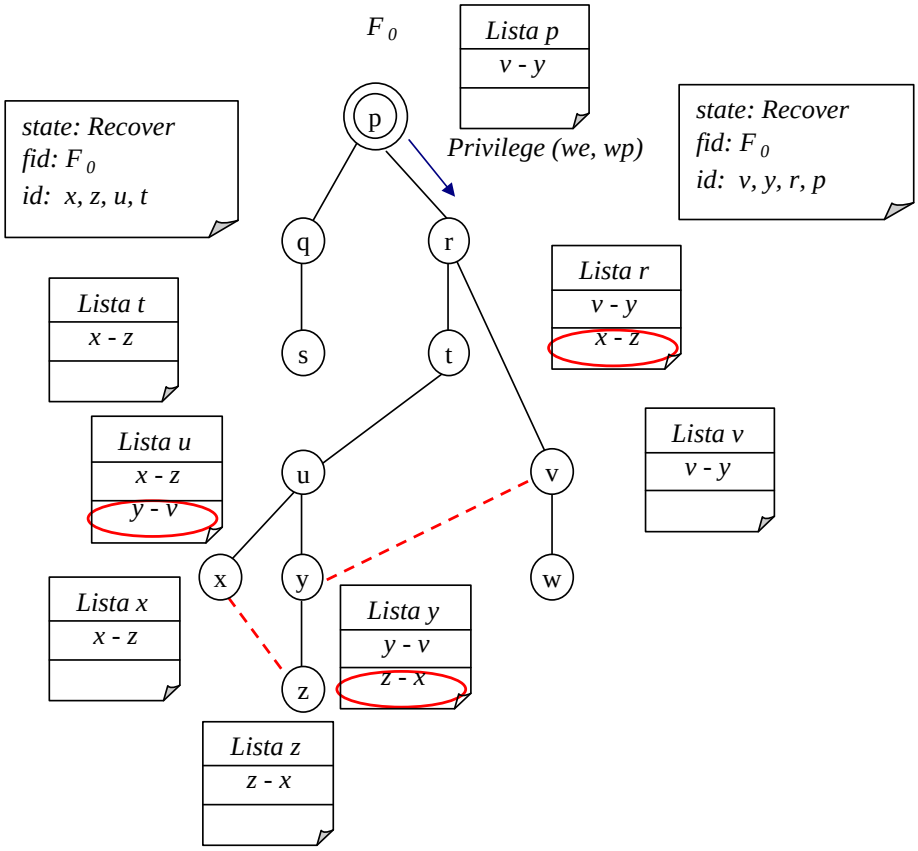


Figura 2.44: Envio da mensagem *Privilege (we,wp)* pela raiz do fragmento após recebimento da mensagem *Recovery*.

forma independente. Entretanto, quando os ciclos se sobrepõem, uma recuperação deve aguardar o término da recuperação anterior para iniciar o seu tratamento, como já pôde ser observado na Figura 2.42. Se as mensagens ficarem simplesmente aguardando o término do tratamento da recuperação anterior, este procedimento levará à situação de *deadlock*, pois, mensagens de recuperação ficarão aguardando indefinidamente.

Como cada processo envolvido na recuperação envia pelo menos uma mensagem de recuperação, a raiz do fragmento recebe no mínimo uma mensagem de recuperação. A proposta, então, é serializar o tratamento das recuperações pela ordem de chegada das mensagens *Recovery(W,W')* na raiz. Logo, a primeira mensagem de recuperação que alcança a raiz é a primeira recuperação a ser tratada e isto é garantido através da criação e do envio das mensagens *Privilege*.

A mensagem *Privilege(we,wp)* referente a uma recuperação, ao ser enviada pela raiz, percorre todo o caminho por onde a mensagem de recuperação passou. Caso esta mensagem alcance um processo que esteja com a mensagem *Recovery(we,wp)* aguardando sua liberação devido à ocorrência de uma outra recuperação, esta mensagem é liberada permitindo que o tratamento desta recuperação prossiga. A outra recuperação só será tratada quando a mensagem *Privilege(we,wp)*, referente a ela, for liberada pela raiz. Uma nova mensagem *Privilege(we,wp)* só é enviada pela raiz quando a anterior alcançar a raiz e for retirada da fila.

Quando uma mensagem *Privilege* alcança um processo, ele verifica se possui apenas uma mensagem referente a esta recuperação e se ela já está sendo tratada, neste caso, a mensagem *Privilege(we,wp)* é simplesmente repassada. Caso a mensagem de recuperação não tenha sido repassada porque chegou depois de outra mensagem de recuperação, o processo a libera, forçando que seu tratamento seja realizado imediatamente. Estes casos podem ser vistos pela Figura 2.45, onde o primeiro acontece quando a mensagem *Privilege(we,wp)*, referente à recuperação da aresta (v,y), alcança os processos v, y e o segundo caso quando alcança o processo u, momento que a mensagem de recuperação referente à aresta (v,y) é liberada.

Com a liberação da mensagem *Recovery(we,wp)*, esta mensagem é repassada ao processo pai, seguida do envio da mensagem *Privilege (we,wp)*. A sequência de envio destas mensagens pode ser vista na Figura 2.46.

Quando a mensagem *Privilege(we,wp)* alcançar o processo que possui as duas mensagens *Recovery(we,wp)* para a mesma recuperação, ou seja, o ancestral é comum

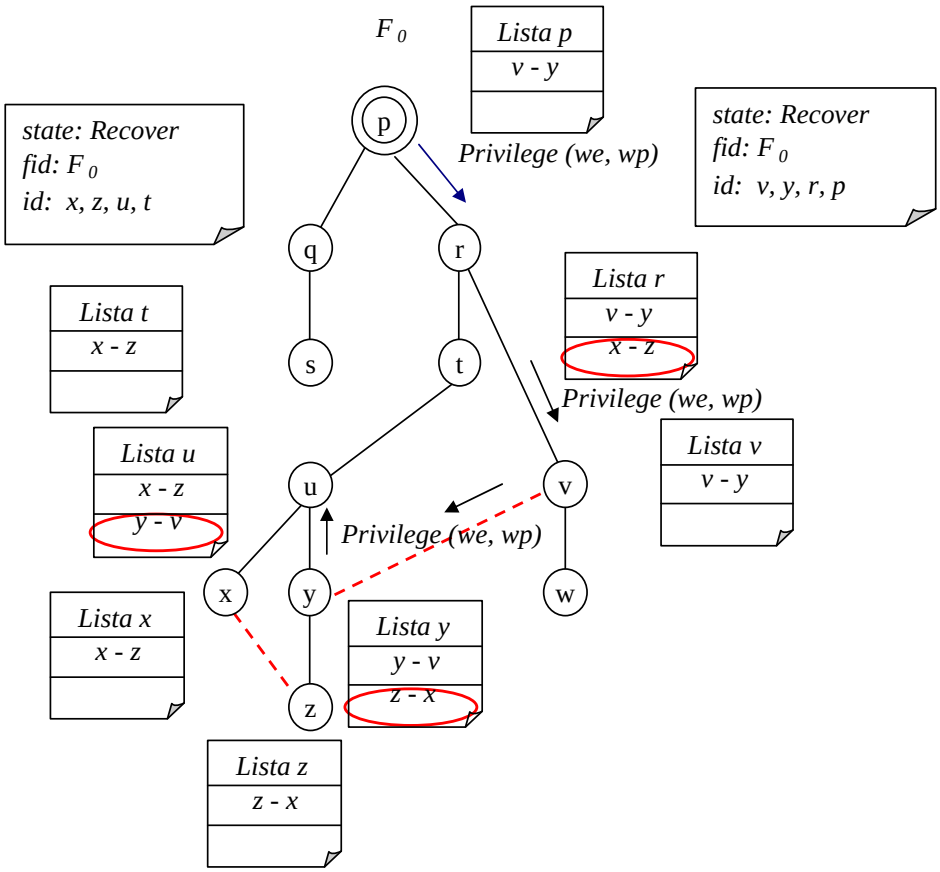


Figura 2.45: Envio da mensagem *Privilege (we,wp)* para nó que enviou mensagem *Recovery*.

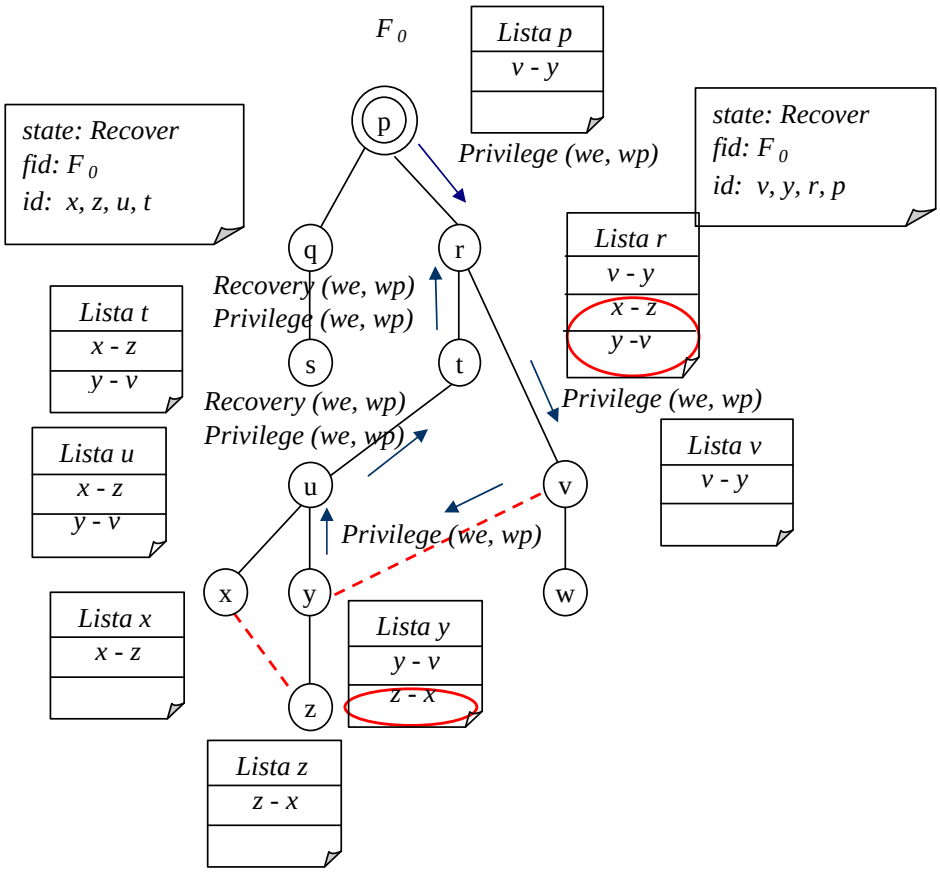


Figura 2.46: Envio da mensagem *Privilege (we,wp)* para nó que enviou mensagem *Recovery*.

aos dois nós que constataram a mesma recuperação, é decidido qual aresta deverá ser retirada do ciclo. Os seguintes procedimentos são executados: a mensagem *Privilege(we,wp)* deve ser repassada para o processo pai indicando que o tratamento desta recuperação está sendo finalizado e a mensagem *Replace(we,wp)* é enviada pelo ciclo como mostrado na Figura 2.47. Neste exemplo, o ancestral comum é o processo r e a mensagem *Privilege* enviada pela aresta $(r,\ p)$ tem a função de indicar o término do tratamento desta falha. Cada processo que receber esta mensagem retira a mensagem referente à esta recuperação da lista. Após a retirada da mensagem de recuperação, caso exista alguma outra mensagem aguardando, ela deve ser liberada. Quando a raiz do fragmento receber a mensagem *Privilege*, ela verifica se existe outra mensagem *Privilege* esperando para iniciar o tratamento da próxima recuperação.

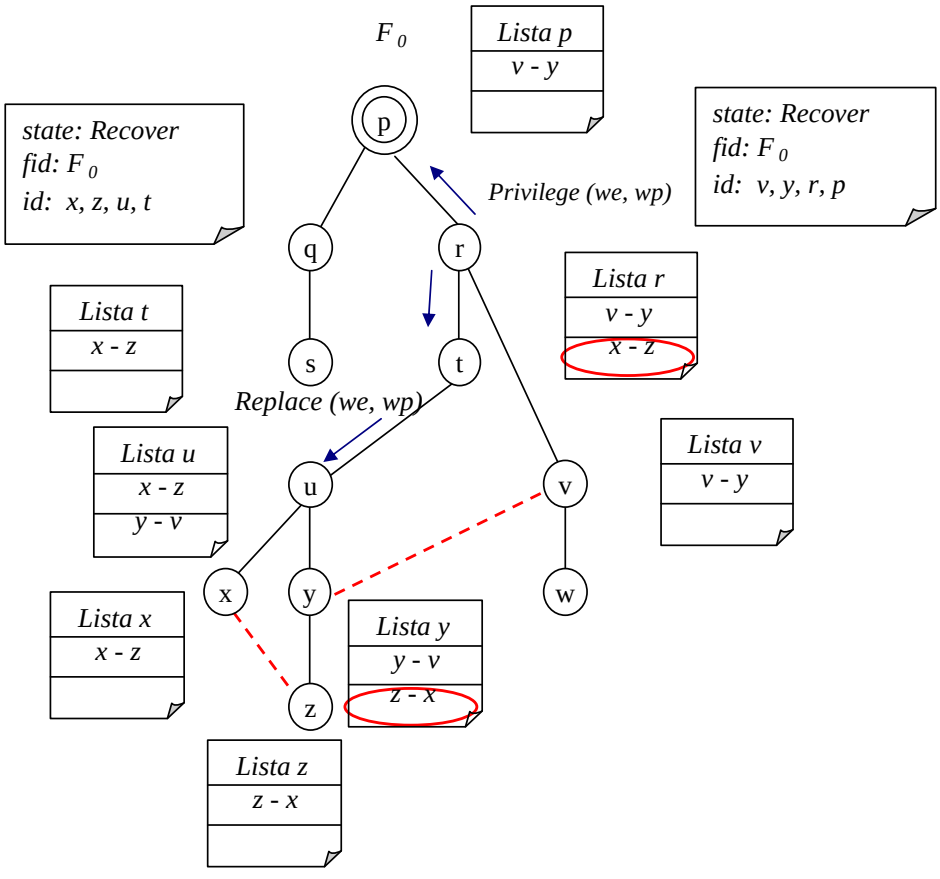


Figura 2.47: Envio da mensagem *Replace (we,wp)* para a aresta que enviou mensagem *Privilege*.

Mensagem Replace (W, W'). O objetivo desta mensagem é definir qual aresta deve ser retirada do ciclo para que seja mantida uma árvore de custo mínimo.

Ao receber a mensagem *Replace (W,W')*, o processo verifica se a mensagem foi

recebida pelo canal de maior ou menor peso no ciclo. Nestes casos ele deve ser retirado ou incluído, respectivamente. Caso a mensagem não seja proveniente de um canal em uma dessas duas situações, a mensagem é apenas repassada pelo ciclo. Antes de repassar a mensagem, o processo faz a mesma análise sobre a aresta destino, uma vez que ela também pode ser incluída ou retirada.

Sempre que receber uma mensagem *Replace (W,W')*, após feita a análise sobre ela, o processo retira as mensagens em relação a esta recuperação da lista e libera a próxima mensagem de recuperação que esteja aguardando para realizar o seu tratamento.

Na Figura 2.48 é apresentado o repasse da mensagem *Replace(we,wp)* pelo ciclo. Se o nó enviar a mensagem *Replace(we,wp)* pelo canal de peso máximo, então, ele marca este canal como *Rejected* e repassa a mensagem *Replace(we,wp)*, isto pode ser visto no envio da mensagem pela aresta $(u,\ y)$. Por outro lado, ao enviar a mensagem pelo nó que deve ser incluído, este canal é marcado como *Branch*, como é o caso de $(y,\ v)$. O mesmo procedimento deve ser adotado quando o nó receber a mensagem pelo canal de maior custo e pelo canal a ser incluído.

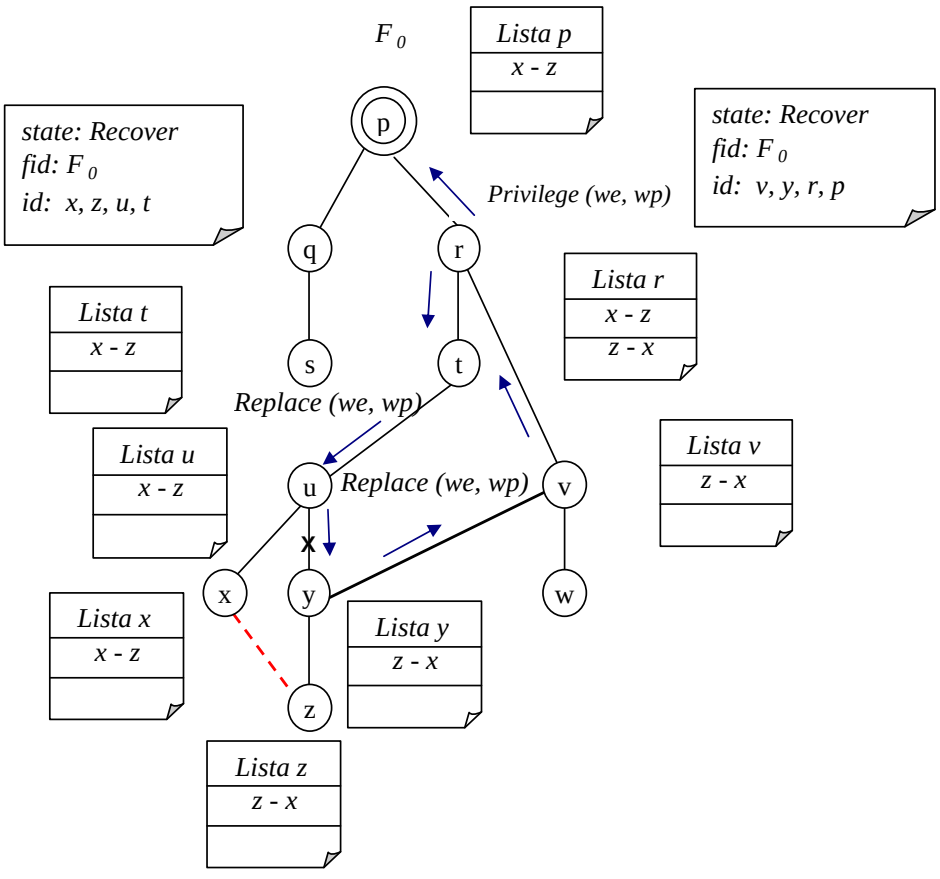


Figura 2.48: Envio da mensagem *Replace (we,wp)* pelo ciclo.

Certos cuidados são necessários na passagem da mensagem *Replace (we,wp)* pelo ciclo, pois é preciso avaliar se ao retirar e incluir canais, a relação pai-filho em todo ciclo deverá ser modificada. Este controle foi feito através de variáveis enviadas junto com a mensagem *Replace (we,wp)*, indicando se canal de maior peso já foi retirado e se a aresta recuperada já foi incluída. Na Figura 2.49, após o término do tratamento da primeira recuperação, pode ser visto que o novo pai do processo y passou a ser o processo v ($in\text{-}branch = v$) para onde foi repassada a mensagem sobre a próxima recuperação *(z, x)* e não para o processo u que era o pai antes do recebimento da mensagem *Replace*. Em seguida, na Figura 2.50 pode ser vista a árvore geradora mínima após o tratamento das duas recuperações identificadas.

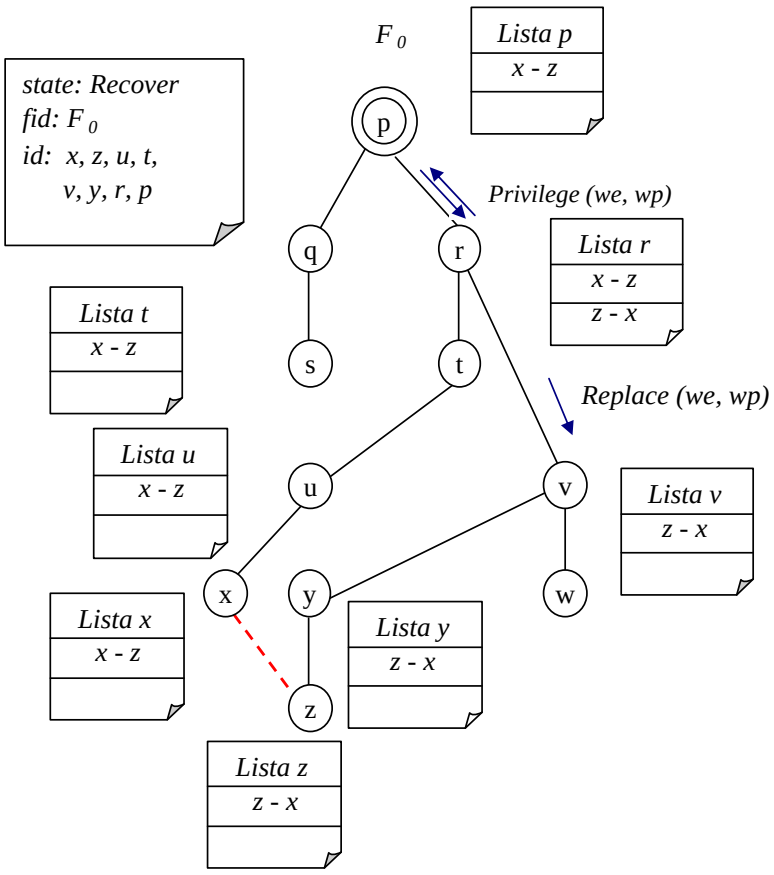


Figura 2.49: Envio da mensagem *Replace (we,wp)* pelo ciclo.

Depois que a mensagem *Recovery(we,wp)* for retirada da lista, se não existir mais nenhuma mensagem de recuperação, o processo muda seu estado para *Sleeping* e começa a enviar mensagens para indicar a sua suspeita de terminação no tratamento de recuperações. Este pode ser o início de uma possível terminação do algoritmo de adaptação da árvore geradora mínima. A seguir na Seção 2.3 são apresentados os procedimentos para a terminação do tratamento de falhas e recuperações.

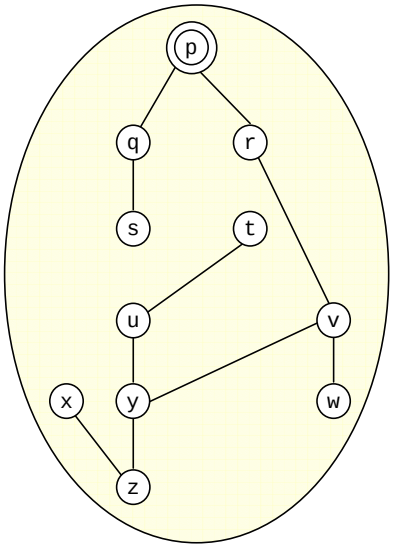


Figura 2.50: Árvore após o tratamento das duas recuperações.

2.3 Procedimentos de Terminação

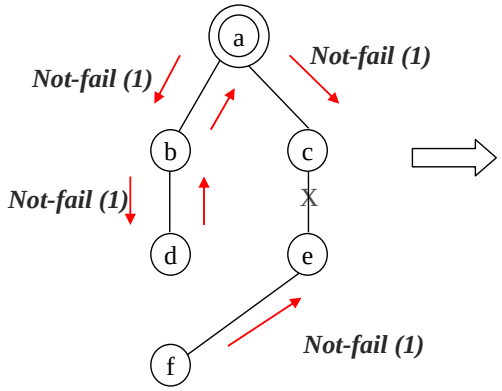
Como contribuição do trabalho aqui proposto, foram incluídos no algoritmo para a adaptação da árvore dois procedimentos de terminação. O primeiro refere-se à detecção de terminação do tratamento de falhas, para que o algoritmo passe a tratar as recuperações. O segundo procedimento trata da detecção de terminação das recuperações e, conseqüentemente, a identificação do fim do processo de adaptação da AGM. Estes procedimentos são necessários para garantir que uma rodada de ajustes tenha terminado antes de se iniciar a outra e a execução da aplicação somente continue depois que todos os ajustes tenham sido concluídos.

Os processos podem suspeitar diversas vezes da terminação de ajustes. Localmente, isto ocorre em duas situações: (i) quando nenhuma aresta adjacente falhou ou foi recuperada; (ii) quando o tratamento da aresta adjacente que falhou ou foi recuperada tiver terminado localmente. Para cada suspeita de terminação local, o processo dispara o que chamamos de detecção de terminação.

A seguir são detalhadas as mensagens e procedimentos para detecção do fim do tratamento de falhas e recuperações, respectivamente.

Cada processo mantém um atributo *tag* e um vetor com uma entrada para os outros processos. O atributo *tag* e cada posição do vetor são inicializados com valores zeros. Eles são utilizados para contabilizar em que rodada do tratamento de terminação cada processo pertencente à árvore se encontra. Esta informação garante que a terminação só seja concretizada após todos os processos terminarem de tratar suas falhas.

São trocados três tipos de mensagens: *Not-fail*, *Not-fail-ack* e *Gosleep-fal*. Se um processo, durante o procedimento de adaptação da árvore, na fase de tratamento de falhas, identificar que não possui falhas a tratar, incrementa a variável *tag* e envia para todos os seus vizinhos, na árvore AGM, a mensagem *Not-fail (tag)* como pode ser visto na Figura 2.51. Esta figura apresenta também uma tabela. Na primeira coluna da tabela está a identificação de cada processo da árvore e, nas colunas seguintes são mostradas as identificações dos processos vizinhos, seguidos dos valores das rodadas em que se encontram no tratamento da terminação. A última coluna mostra o valor da *tag* do próprio processo. Para a terminação ser efetivada, todos os processos devem estar na mesma rodada e sem mais nenhuma falha a tratar.



Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	d	0	--	--	1
c	e	0	--	--	0
d	--	--	--	--	1
e	f	0	--	--	0
f	--	--	--	--	1

Figura 2.51: Envio de mensagens *Not-fail(tag)* que indicam a suspeita de terminação do tratamento de falhas.

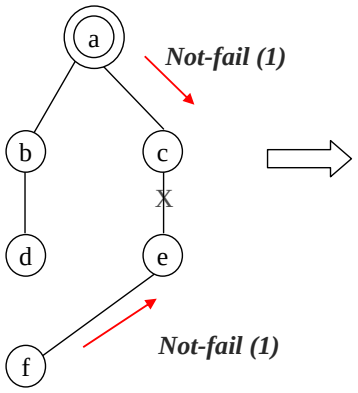
Mensagem Not-fail(T). Um processo ao receber a mensagem *Not-fail (tag)* compara o valor da *tag* recebida com a sua variável local. Caso o valor da *tag* seja maior, significa que uma nova suspeita de terminação de falhas está se iniciando, logo, o processo armazena na sua variável local o valor da *tag* recebida. Se o processo não estiver participando de nenhum tratamento de falhas, o próximo passo é repassar a mensagem *Not-fail(tag)* para os demais vizinhos na árvore. Caso o processo esteja participando de algum tratamento de falha, ele não envia mensagens. Este é o caso dos processos '*c*' e '*e*', mostrados na Figura 2.52, que somente atualizam os valores de suas *tag's* locais após recebimento da mensagem *Not-fail(T)*.

Se o processo que recebeu a mensagem *Not-fail(tag)* for um processo folha, a mensagem *Not-fail-ack (tag)* é enviada ao processo pai, como apresentado na Figura 2.53.

A atualização da *tag* local por um processo que ainda está tratando uma falha é importante, pois, este processo ao terminar de tratar as suas falhas, deve incrementar o valor da sua *tag* e enviar a mensagem *Not-fail(tag)* para seus vizinhos, indicando o início

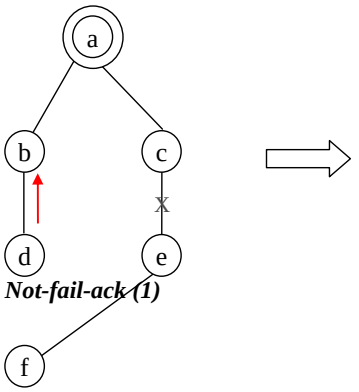
de uma nova rodada de suspeita de terminação, conforme mostra a Figura 2.54.

O processo ao receber nova mensagem *Not-fail(tag)* com valor de *tag* maior, caso não esteja participando do tratamento de falhas, deve confirmar a suspeita de terminação com o envio da mensagem *Not-fail-ack(tag)* com o valor da *tag* incrementada, conforme mostra a Figura 2.55.



Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	d	0	--	--	1
c	e	0	--	--	1
d	--	--	--	--	1
e	f	0	--	--	1
f	--	--	--	--	1

Figura 2.52: Atualização da *tag* após recebimento da mensagem *Not-fail(tag)* por processos que estão tratando falhas.

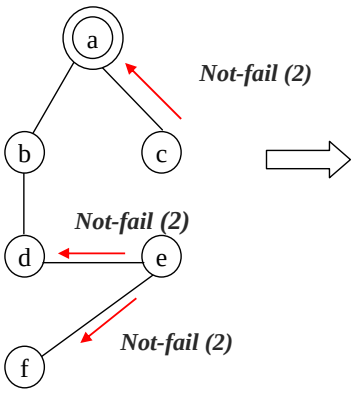


Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	d	0	--	--	1
c	e	0	--	--	1
d	--	--	--	--	1
e	f	0	--	--	1
f	--	--	--	--	1

Figura 2.53: Envio de mensagem *Not-fail-ack(tag)*.

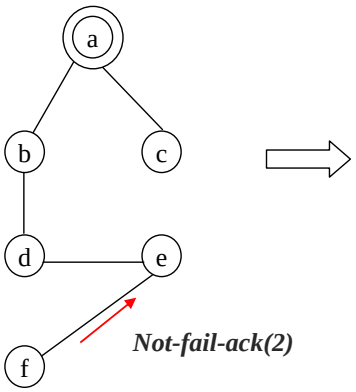
Mensagem Not-fail-ack(T). Quando um processo recebe a mensagem *Not-fail-ack (tag)*, ele verifica se a *tag* recebida é igual a sua variável local. Se os valores forem diferentes a mensagem é ignorada, senão, o processo armazena no seu vetor, na posição correspondente ao vizinho, o valor da *tag* recebida, indicando que estão na mesma rodada do tratamento de terminação falhas.

Após receber esta mensagem, o processo analisa as informações armazenadas no seu vetor para verificar se todos os seus processos filhos já lhe enviaram a mensagem *Not-fail-ack (tag)*. Se isto for verdade, o processo analisa se é a raiz da árvore, neste caso, a mensagem de fim de tratamento de falhas *Gosleep-fal* é enviada para todos os processos



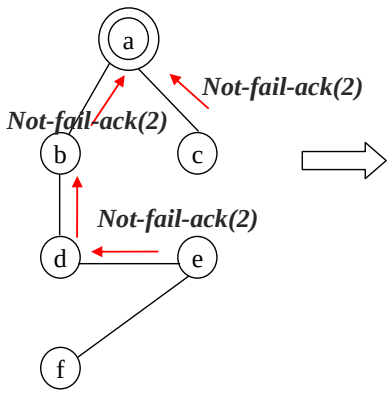
Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	d	0	--	--	1
c	--	--	--	--	2
d	e	0	--	--	2
e	f	0	--	--	2
f	--	--	--	--	2

Figura 2.54: Envio da mensagem *Not-fail(tag)* após o término do tratamento de falhas.



Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	2
b	d	0	--	--	2
c	--	-	--	--	2
d	e	0	--	--	2
e	f	0	--	--	2
f	--	--	--	--	2

Figura 2.55: Envio de mensagem *Not-fail-ack(tag)* .



Processos	Filhos				Tag
	Id	Valor	Id	Valor	
a	b	2	c	0	2
b	d	2	--	--	2
c	--	--	--	--	2
d	e	2	--	--	2
e	f	2	--	--	2
f	--	--	--	--	2

Figura 2.56: Envio da mensagem *Not-fail-ack(tag)* após recebimento das mensagens *Not-fail(tag)*.

filhos, caso contrário, a mensagem *Not-fail-ack (tag)* é repassada para o processo pai, como mostra a Figura 2.56.

Mensagem Gosleep-fal. Ao receber esta mensagem, o processo a repassa para os seus processos filhos como mostra a Figura 2.57. Cada processo ao receber *Gosleep-fal* atualiza seu estado para *Sleeping* e identifica que pode ser iniciado o tratamento das recuperações, pois, o seu tratamento de falhas já se encerrou.

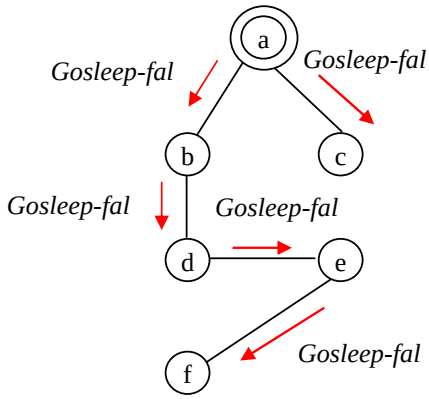
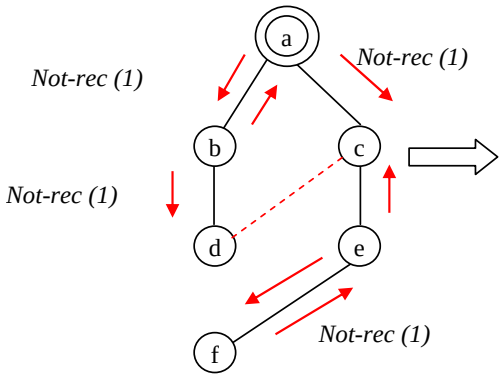


Figura 2.57: Envio da mensagem *Gosleep-fal*.

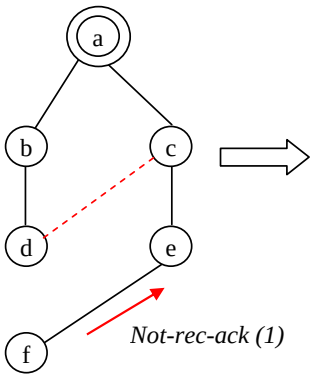
Em relação ao tratamento de terminação das recuperações, são trocadas mensagens do tipo *Not-rec(tag)*, *Not-rec-ack(tag)* e *Gosleep-rec*. Novamente cada processo mantém uma variável local *tag* que indica a rodada da terminação do tratamento de recuperações em que o processo está participando, assim como um vetor para todas as arestas que estão no estado *Branch*.

Mensagem Not-rec(tag). Um processo ao suspeitar do término de tratamento de recuperações envia mensagens *Not-rec(tag)* para os seus vizinhos, como no exemplo da Figura 2.58. No caso de ser processo folha, envia a mensagem *Not-rec-ack (tag)* para o processo pai, como mostra a Figura 2.59.



Processos	Vizinhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	a	0	d	0	1
c	a	0	e	0	0
d	b	0	--	--	0
e	c	0	f	0	1
f	e	0	--	--	1

Figura 2.58: Envio de mensagens *Not-rec(tag)* que indicam a suspeita de terminação do tratamento de recuperações.

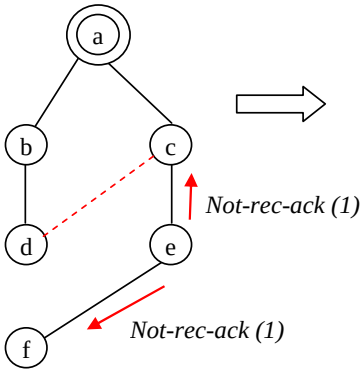


Processos	Vizinhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	a	0	d	0	1
c	a	0	e	0	1
d	b	0	--	--	1
e	c	0	f	1	1
f	e	0	--	--	1

Figura 2.59: Envio de mensagem *Not-rec-ack(tag)* que indica a confirmação da suspeita de terminação do tratamento de recuperações pelo processo.

O processo que está participando de um tratamento de recuperação e recebeu a mensagem *Not-rec(tag)* verifica se o valor da *tag* recebida é maior, se for, atualiza o valor da sua *tag* local. É importante sempre manter o valor da rodada atualizado, pois quando este processo verificar que terminou de tratar suas recuperações, deve enviar mensagens *Not-rec(tag)* com um valor de *tag* superior, para que as rodadas não se confundam.

Mensagem Not-rec-ack(tag). Processo, ao receber esta mensagem, executa um procedimento semelhante ao adotado na terminação das falhas, a diferença é que aqui o processo, caso não seja a raiz da árvore, repassa para todos os seus vizinhos a mensagem *Not-rec-ack (tag)*, ao invés de enviar somente para o processo pai atual. Isto foi necessário porque durante o tratamento de recuperações a relação pai-filho pode ter sido invertida. Portanto, um processo que era filho e passou a ser o pai pode ficar esperando indefinidamente pelo recebimento da mensagem *Not-rec-ack (tag)* para finalizar o tratamento de recuperações. O envio dessas mensagens pode ser visualizado pela Figura 2.60.

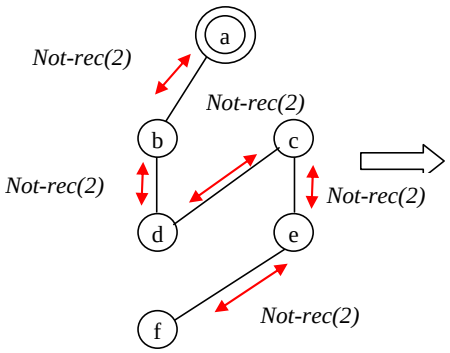


Processos	Vizinhos				Tag
	Id	Valor	Id	Valor	
a	b	0	c	0	1
b	a	0	d	0	1
c	a	0	e	1	1
d	b	0	--	--	1
e	c	0	f	1	1
f	e	1	--	--	1

Figura 2.60: Envio da mensagem *Not-rec-ack(tag)* após receber as mensagens *Not-rec-ack(tag)* que esperava.

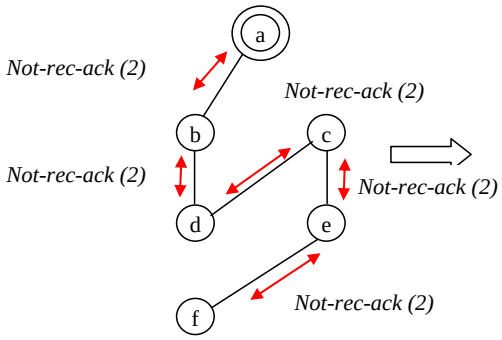
O processo que recebeu a mensagem *Not-rec-ack(tag)*, mas ainda está participando de um tratamento de recuperação simplesmente ignora o recebimento desta mensagem.

Quando o tratamento de todas as recuperações forem concluídas uma nova rodada de envio de mensagens *Not-rec(tag)* e *Not-rec-ack(tag)* é realizada, conforme mostram as Figuras 2.61 e 2.62.



Processos	Vizinhos				Tag
	Id	Valor	Id	Valor	
a	b	0	--	--	2
b	a	0	d	0	2
c	d	0	e	1	2
d	b	0	c	0	2
e	c	0	f	1	2
f	e	1	--	--	2

Figura 2.61: Envio da mensagem *Not-rec(tag)* após tratar todas as recuperações.



Processos	Vizinhos				Tag
	Id	Valor	Id	Valor	
a	b	2	--	--	2
b	a	2	d	2	2
c	d	2	e	2	2
d	b	2	c	2	2
e	c	2	f	2	2
f	e	2	--	--	2

Figura 2.62: Envio da mensagem *Not-rec-ack(tag)* por todos os processos.

Observe que no momento em que o processo raiz receber *Not-rec-ack(tag)* de todos os seus filhos que possuam valores de *tag* iguais ao seu, pode enviar pela árvore a mensagem *Gosleep-rec*. Esta verificação pode ser feita comparando os valores da *tag* de cada vizinho armazenados no vetor e o valor da sua própria *tag*. Se todos os valores forem iguais significa que todos os ajustes na árvore foram concluídos e, portanto, a execução da aplicação pode prosseguir sem problemas.

Mensagem Gosleep-rec. Ao receber esta mensagem o processo a repassa para os seus processos filhos como mostra a Figura 2.63. Cada processo, ao receber *Gosleep-rec*, atualiza seu estado para *Sleeping* e identifica que o tratamento de recuperações se encerrou, assim como o algoritmo de adaptação da árvore. A execução do algoritmo pode dar continuidade ao seu processamento utilizando a nova estrutura adaptada.

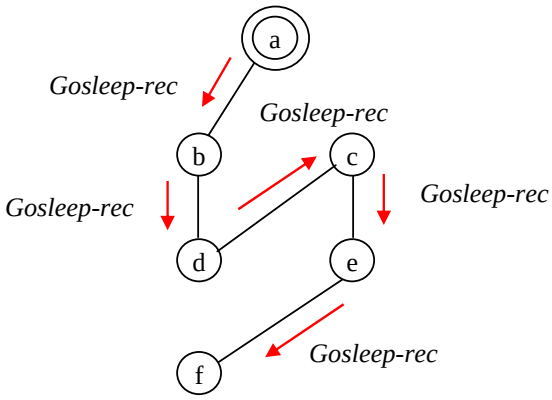


Figura 2.63: Envia mensagem *Gosleep-rec* pela árvore.

Capítulo 3

Ferramenta para Operações Coletivas Dinâmicas para MPI

Este trabalho propõe a construção inicial de uma nova biblioteca para operações coletivas utilizando o MPI. Estas operações consideram as mudanças ocorridas no ambiente quando forem executadas. A ferramenta proposta, inicialmente, disponibiliza as seguintes operações MPI: *MPI_ Init*, *MPI_ Init_ thread*, *MPI_ Bcast* e *MPI_ Finalize*. Futuramente, implementações para outras operações coletivas poderão ser incluídas.

Além das operações MPI citadas, as seguintes funcionalidades estão presentes na ferramenta: monitoração de rede, construção da árvore geradora mínima, adaptação da árvore geradora mínima e algoritmos para detecção do término da adaptação da árvore. Para um melhor entendimento destas funcionalidades, elas foram separadas em módulos.

A Figura 3.1 apresenta os módulos que compõem a ferramenta e a integração entre eles. O quadro à esquerda representa a aplicação do usuário que pode acessar o módulo "Operações MPI" da ferramenta de acordo com o tipo de operação que está sendo realizada em seu programa. À direita estão os módulos presentes na ferramenta, o primeiro módulo é "Operações MPI" responsável pela implementação das operações MPI que levam em consideração o ambiente de execução. Em seguida, está o Módulo "Monitoração da Rede" que obtém as latências entre os processadores através de consultas utilizando a ferramenta *Network Weather Service*. O Módulo "Construção da Árvore" é responsável pela criação da árvore geradora mínima utilizada para disseminação da informação de acordo com as latências coletadas pelo Módulo "Monitoração da Rede". O Módulo "Adaptação da Árvore", após obter os últimos valores das latências, realiza as adaptações necessárias na árvore para que ela continue sendo uma árvore de custo mínimo. Por último, o Módulo "Terminação da Adaptação" será acessado quando a árvore geradora mínima terminar a

sua adaptação ou quando, após monitoração da rede, for constatada que nenhuma variação nos valores de latência ocorreu antes de ser realizada uma operação coletiva. Nas seções a seguir são apresentadas as funcionalidades de cada módulo e suas características.

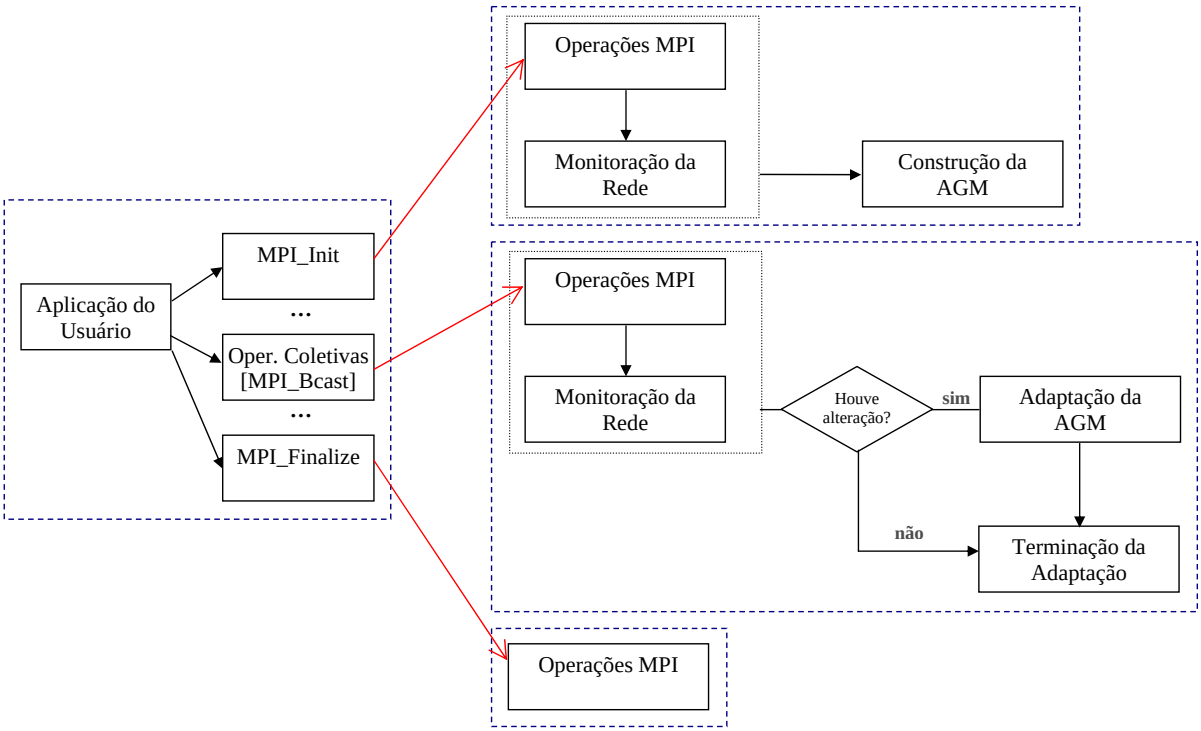


Figura 3.1: Módulos da Ferramenta.

3.1 Operações MPI

O Módulo para Operações MPI é composto pelas operações MPI reescritas que estão disponíveis na ferramenta. Inicialmente, são elas: *MPI_Init*, *MPI_Init_thread*, *MPI_Finalize* e *MPI_Bcast*. Futuramente outras operações para disseminação da informação poderão ser reescritas e incluídas neste módulo, tais como: *MPI_Reduce*, *MPI_Gather* e *MPI_Scatter*.

As funções MPI implementadas podem ser chamadas pelo programador sem a necessidade de mudança na assinatura de nenhuma delas. Isto acontece porque caso a aplicação do usuário utilize alguma operação que esteja presente no módulo que fornece as operações MPI, a operação original será imediatamente substituída pela implementação presente na ferramenta de forma transparente ao usuário. Caso não exista a operação chamada na ferramenta, a operação MPI executada será aquela fornecida pela biblioteca MPI original.

A seguir são apresentadas as modificações realizadas nas operações MPI para se ade-

quar às necessidades da nossa ferramenta que realiza a disseminação de através de uma AGM.

***MPI_ Init*:** A função *MPI_ Init* da ferramenta possui as mesmas funcionalidades presentes na operação *MPI_ Init* original. A diferença, em relação à nossa implementação, está na alocação de estruturas de dados para o armazenamento da árvore geradora mínima que é construída neste momento e utilizada durante a realização de uma operação coletiva. Portanto, esta operação, além do que a operação *MPI_ Init* original realiza, executa os seguintes passos para que a árvore de custo mínimo seja construída:

- alocação de memória;

  Após ter sido feita a alocação, os valores das latências entre os processos podem ser obtidos e armazenados.

- acesso ao módulo "Monitoração de Rede";

  Este módulo é responsável pela obtenção das latências entre todos os processos e é explicado na próxima seção.

- acesso ao módulo "Construção da AGM";

  Após todas as latências terem sido coletadas, a construção da árvore geradora pode ser iniciada conforme o algoritmo distribuído descrito na Seção 2.1.

***MPI_ Init_ thread*:** Todas as considerações feitas na implementação da operação *MPI _Init* são válidas para a implementação desta operação.

***MPI_ Bcast*:** Sempre que uma operação *MPI_ Bcast* é chamada pela aplicação do usuário, a ferramenta executa os seguintes procedimentos:

- acessa o módulo "Monitoração de Rede";

  A proposta da operação coletiva na nossa ferramenta é sempre realizá-la sobre a árvore de custo mínimo. Para isto, na sua implementação foi preciso fazer o acesso ao módulo "Monitoração de Rede". Desta forma, sempre que a operação *MPI_ Bcast* for executada, ela primeiro obtém os últimos valores de latência entre os processos através do NWS.

- identificação de mudanças nos valores das latências;

  Após coletar os últimos valores para as latências, eles devem ser comparados com os valores das latências anteriores para verificar se sofreram alterações. Quando são identificadas mudanças nas latências, com o aumento ou a diminuição de seus valores, os canais envolvidos são classificados como tendo sofrido uma falha ou recuperação, respectivamente;

- acesso ao módulo "Adaptação Árvore";

  Caso tenham sido detectadas falhas e/ou recuperações nos canais, a adaptação da árvore deve ser realizada. Esta adaptação é feita através do algoritmo distribuído disponibilizado por esta ferramenta, conforme descrito na Seção 2.2;

- Após terem sido feitas as alterações necessárias na árvore, finalmente a operação *MPI_Bcast* pode ser executada. A disseminação da informação pela árvore é realizada através das operações ponto-a-ponto *MPI_Send* e *MPI_Recv*;

***MPI_Finalize*:** Esta foi a última operação MPI implementada que, da mesma forma que a operação *MPI_Init*, realiza as mesmas atividades apresentadas na operação original. A implementação desta operação possui como atividade adicional a liberação das estruturas de dados utilizadas para armazenamento das informações da árvore geradora mínima utilizada na execução das operações coletivas.

3.2 Monitoração da Rede

Este módulo é acessado durante toda a execução da aplicação. Sua presença é fundamental, pois, através dele são fornecidos os valores das latências para a construção da árvore inicial, identificação das ocorrências de falhas e recuperações dos canais pela análise dos valores coletados e, também, a necessidade da adaptação da árvore. Este módulo utiliza a ferramenta de monitoração de rede NWS para a consulta dos valores das latências instantânea entre cada par de processos.

3.2.1 Network Weather Service - NWS

O NWS é uma ferramenta que executa monitoração de forma distribuída para produzir previsões dinâmicas de desempenho dos recursos, levando em consideração as medidas

armazenadas. Para armazenar estes dados, a ferramenta utiliza um conjunto distribuído de sensores que reúne informações instantâneas sobre os recursos que estão sendo monitorados [Wolski et al. 1999, Wolski 2003, Wolski et al. 1997]. Os dados coletados, para cada recurso monitorado, são armazenados em arquivos de sistemas com as informações do tempo (*time-stamp*) e da medida obtida.

A ferramenta *NWS* é composta por três processos, *nws_sensor*, *nws_memory* e *nws_server*, descritos a seguir:

nws_sensor: coleta medidas de desempenho de um recurso específico. Para que um ou mais recursos de uma máquina sejam monitorados pelo NWS, é preciso que o sensor esteja executando nela. É possível monitorar os seguintes recursos através do NWS:

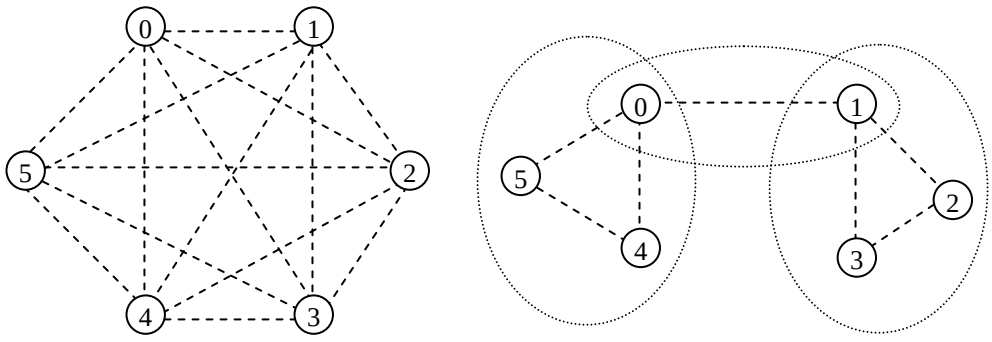
- a fração de CPU disponível para os processos;
- a quantidade de espaço disponível em disco;
- a quantidade de memória livre disponível na máquina;
- o tempo requerido para estabelecer uma conexão TCP;
- a latência e banda TCP entre pares de processos.

Somente um sensor é necessário para monitorar múltiplos recursos de uma mesma máquina. Alguns recursos requerem no mínimo duas máquinas para operar corretamente, este é o caso dos sensores de rede.

O sensor de rede fornece medidas de comprimento de banda e latência, sendo que cada medida coletada é nomeada e armazenada separadamente. Durante a monitoração de rede, a medição de latência, por exemplo, é obtida pelo NWS através do RTT (*Round-Trip Time*) de um pacote. O RTT é o tempo gasto (tempo de ida e volta), por um pacote pequeno, para viajar do cliente ao servidor. A latência poderia ser medida pelo próprio *ping*, mas poderia ocorrer filtragem de pacotes.

O NWS tenta medir desempenho de rede fim-a-fim entre todos os possíveis pares de sensores que estejam executando a monitoração de rede. Neste caso, a comunicação entre todos os sensores pode consumir uma quantidade considerável de recursos. Logo, uma característica importante presente no NWS para evitar contenção, mas ainda fornecer todas as medidas entre todos os sensores, é organizar os sensores de rede em cliques.

A clique é composta por um conjunto de sensores que realizam a monitoração da rede entre si. Cada sensor participante de uma clique só pode realizar a monitoração com os outros membros desta mesma clique. Os sensores também podem participar de várias cliques simultaneamente permitindo a definição de uma hierarquia na comunicação, como pode ser visto na Figura 3.2. Nesta figura, a linha tracejada refere-se aos experimentos realizados, que são as medições de RTT, entre os pares de sensores e as elipses representam as cliques formadas. Observa-se que a quantidade de experimentos realizados é muito inferior com a utilização de cliques, sendo apenas monitoradas as seguintes latências, neste exemplo: (0,4), (0,5) e (4,5) para a clique representada à esquerda, (1,2), (1,3) e (2,3) para a clique representada à direita e (0,1) para a clique do meio. Enquanto, sem a utilização das cliques, cada processo realiza experimentos com todos os demais.



(a) Sensores de rede (b) Sensores organizados em *cliques*

Figura 3.2: Representação dos sensores de rede e a sua organização em *cliques*.

Outro dado importante é que o *nws_sensor* não tenta relacionar as características de um recurso que monitora.

nws_memory: fornece armazenamento de dados persistente para as medidas monitoradas através do nws_sensor. As informações dos dados coletados pelo NWS sobre cada recurso são armazenadas em arquivos de sistemas chamados séries (*time series*). Logo, para que se obtenha informação de um dado recurso, é preciso fazer a consulta à série correspondente.

O NWS mantém nas séries um conjunto limitado de registros sobre as informações coletadas durante a monitoração de um dado recurso. Essas informações são armazenadas utilizando uma fila circular, portanto, à medida que novas monitorações forem armazenadas nas séries, as informações mais antigas tendem a desaparecer. A quantidade de registros que são armazenadas pode ser definido quando for iniciado o serviço *nws_memory*.

O NWS mantém um conjunto de métodos que são utilizados para a geração de previsão sobre um dado recurso. Cada método possui sua própria parametrização e realiza as suas previsões, para os recursos monitorados, de acordo com as informações armazenadas em suas séries.

nws_nameserver: é o servidor de nomes que armazena as informações de todos os componentes presentes no NWS, tais como: o registro das máquinas, dos sensores (*nws_sensor*), das memórias (*nws_memory*) e séries utilizadas para armazenar as informações sobre a monitoração dos recursos. Todos estes componentes precisam se registrar no *nws_server* para informar sobre a sua existência e suas capacidades.

O endereço do processo *nws_nameserver* é o único conhecido por todos os processos do sistema que registram seus nomes e localização nele. As demais informações dos dados quanto dos serviços podem ficar distribuídos.

O nws_memory e o nws_sensor são *daemons* que ficam executando continuamente, somente o nws_memory necessita de armazenamento em disco. A comunicação destes *daemons* é realizada através de *sockets* TCP.

O papel do NWS, neste trabalho, é fornecer as medidas das latências instantâneas entre todos os processos, para que se possa construir e manter atualizada uma árvore geradora mínima que é utilizada nas operações coletivas.

3.3 Construção da AGM

Sempre que uma aplicação MPI faz a chamada à operação *MPI_Init*, a ferramenta substitui esta operação pela nossa implementação. Nesta implementação, o primeiro passo é a obtenção dos últimos valores das latências entre todos os processos e, em seguida, é a construção da árvore geradora mínima. O passo-a-passo de funcionamento do algoritmo para a construção da árvore inicial foi explicado na Seção 2.1.

3.4 Adaptação da AGM

Quando uma operação coletiva é executada, todos os processadores verificam se ocorreram alterações nos valores das latências adjacentes, no caso de mudanças de latências, a adaptação da árvore geradora mínima é realizada. Na Seção 2.2 foi apresentado o

passo-a-passo da execução deste algoritmo. Este algoritmo sempre é executado quando for verificado que um ou mais canais tiveram suas latências alteradas.

3.5 Terminação da Adaptação

Neste módulo são encontrados os algoritmos desenvolvidos para a detecção do término do tratamento de falhas e do término do tratamento de recuperações dos canais, explicados na Seção 2.3. Estes tratamentos tornaram-se necessários para que após ter sido feito o acesso ao módulo "Monitoração de Rede" e a verificação da necessidade de realizar a adaptação na AGM, o processo seja informado em que momento pode dar continuidade à execução da aplicação, com a garantia de que nenhum outro processo esteja envolvido na adaptação da árvore geradora mínima.

Capítulo 4

Resultados Computacionais

Para a realização dos testes, foram utilizadas 24 máquinas com processadores Pentium IV, 2.6 GHz de freqüência, 512 MB de memória RAM, sistema operacional GNU/Linux versão 2.6.8-1.521, biblioteca MPI-LAM-7.0.6 e versão i386-redhat-linux/3.3.3 do compilador gcc. As máquinas, pertencentes a uma mesma rede, encontravam-se dedicadas exclusivamente aos testes. Simulamos um ambiente heterogêneo da seguinte forma: definimos uma topologia constituída de 6 *sites* com 4 máquinas cada, onde cada *site* representa uma localidade geográfica distinta e todos os processadores se comunicam por canais com valores de latências variados.

Com o objetivo de obter valores mais próximos à realidade, as latências entre os *sites* foram estimadas com base em *pings* realizados a universidades do Brasil, China e Estados Unidos. Assim, os *sites* foram identificados pelos nomes S0, S1, S2, S3, S4 e S5 e a distribuição foi realizada da seguinte forma: consideramos três *sites* localizados em uma mesma região (S0, S3 e S5), dois em outra (S1 e S4) e um único *site* isolado (S2).

A definição da topologia descrita foi utilizada nos testes realizados e pode ser vista na Tabela 4.1, onde também são apresentadas as identificações dos processos que compõem cada *site* e os valores das latências iniciais em *milisegundos*. As latências entre processos do mesmo *site* foram consideradas com valores muito pequenos, próximos a zero. Todas as latências possuem valores únicos, pois estes são compostos dos valores obtidos pela monitoração e da combinação da identificação dos processadores que compõem o canal. Como cada processador possui uma identificação única, o valor da latência torna-se único também. A cada execução do algoritmo AGM, para os mesmos valores de latência, a mesma árvore será construída.

A eficiência da proposta apresentada foi avaliada comparando o desempenho dos al-

	S0 **00-03**	**S1** **04-07**	**S2** **08-11**	**S3** **12-15**	**S4** **16-19**	**S5** **20-23**
S0	0.0	485.4	698.9	14.9	332.8	61.4
S1	485.4	0.00	364.1	583.8	13.5	490.5
S2	698.9	364.1	0.0	701.2	371.7	722.9
S3	14.9	583.8	701.2	0.0	331.0	35.1
S4	332.8	13.5	371.7	331.0	0.0	355.9
S5	61.4	490.5	722.9	35.1	355.9	0.0

Tabela 4.1: Latências entre os sites.

goritmos implementados *MPICH-like*, *MagPIe-like* e *AGM*. A comparação entre os algoritmos foi feita através de uma aplicação sintética que realiza *broadcasts* consecutivos. Os algoritmos *MPICH-like* e *MagPIe-like* referem-se às implementações de comunicação coletiva conforme proposto em *MPICH* e *MagPIe*, que não se encontravam disponíveis no ambiente utilizado. No *AGM*, a análise foi feita através de execuções com e sem a atualização da árvore gerada inicialmente. Como observamos pequenas variações de tempo em execuções sucessivas de uma mesma aplicação, cada um dos testes foi executado três vezes em um ambiente exclusivo com uma variância média de 0,35%. As figuras e tabelas apresentadas neste capítulo mostram estas médias.

4.1 Comparação entre *MPICH-like* e *MagPIe-like*

A primeira seqüência de testes comparou o desempenho dos algoritmos *MPICH-like* e *MagPIe-like*, onde foi possível verificar a vantagem em se utilizar a árvore gerada pela versão *MagPIe-like* que detém o conhecimento da topologia em duas camadas. *MagPIe-like* consegue identificar quais processos pertencem ao seu *site* e quais estão mais distantes fisicamente, enquanto o algoritmo *MPICH-like* constrói uma árvore binomial sem a distinção da localização dos processos. O algoritmo *MagPIe-like* tenta minimizar a quantidade de comunicação entre processos de *sites* diferentes da raiz do *broadcast* até o seu destino.

As Figuras 4.1 e 4.2 representam, respectivamente, as árvores geradas pelos algoritmos *MPICH-like* e *MagPIe-like*, que possuem como raiz do *broadcast* o processo 12. Além disso, também mostram os *sites* a que os nós pertencem. A Tabela 4.2 mostra o tempo total, em segundos, da aplicação com a variação da quantidade de MPI_Bcast's e a porcentagem de melhoria do *MagPIe-like* em relação ao *MPICH-like*.

Neste teste, já é possível observar que a utilização da árvore gerada pelo algoritmo *MagPIe-like* para a realização do *broadcast*, tende a ser melhor. Pode-se observar que o

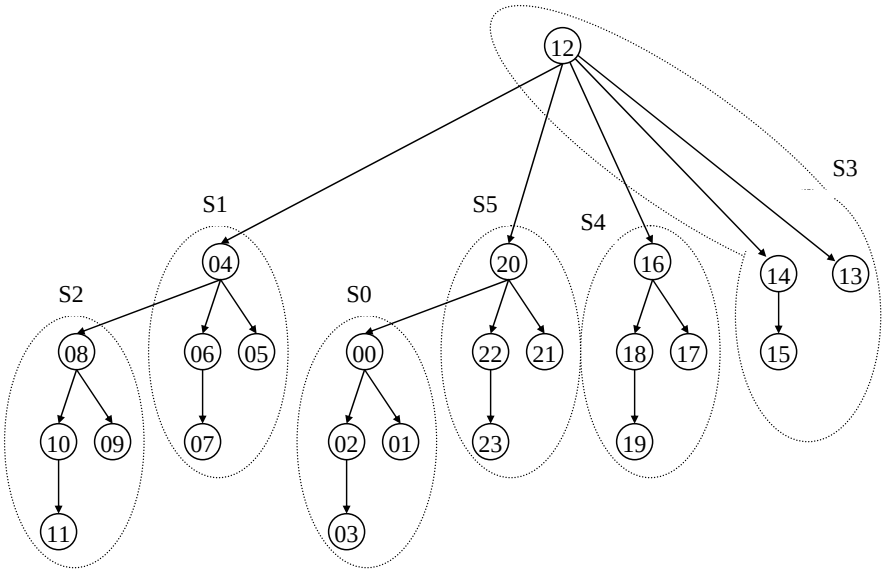


Figura 4.1: Árvore gerada pelo algoritmo *MPICH-like*.

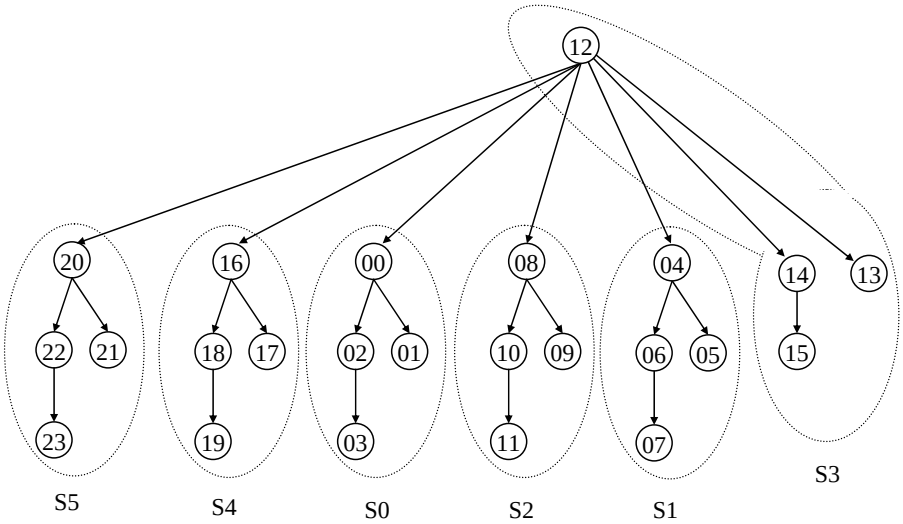


Figura 4.2: Árvore gerada pelo algoritmo *MagPIe-like*.

No de bcast's	**1**	**4**	**16**
MPICH-like	4,93	6,95	15,88
MagPIe-like	3,80	5,40	13,08
Melhoria	22,92%	22,30%	17,63%

Tabela 4.2: Execuções *MPICH-like* e *MagPIe-like* com tempos em segundos.

tempo de espera, considerando as latências dos canais, para o recebimento da mensagem pelos processos (0) e (8) tende a diminuir com o *MagPIe-like*, uma vez que pelo algoritmo *MPICH-like*, estes processos deveriam esperar duas comunicações entre *sites*. Para uma mensagem alcançar o processo (0), através do *MPICH-like*, primeiro a raiz do *broadcast* (12) deve enviar a mensagem para processo (20) e este, em seguida, para o processo (0), o que leva aproximadamente 96,5 *ms*. Com o *MagPIe-like* o processo (0) deve aguardar 14,9 *ms* para o recebimento desta mensagem, um ganho equivalente a 18% em relação ao tempo de espera.

4.2 Comparação entre *MagPIe-like* e *AGM*

O algoritmo *MagPIe-like* foi escolhido para comparações nos testes apresentados, devido ao seu melhor desempenho. Na Figura 4.3, pode ser vista a árvore criada pelo algoritmo *AGM* e na Figura 4.4 são apresentados os gráficos para testes realizados com a execução de 4, 8 e 16 *broadcasts* consecutivos com mensagens de tamanhos iguais a 24 *bytes*. Nestes gráficos são comparadas as médias obtidas em três execuções dos tempos de geração da árvore, tempo total da aplicação e a diferença entre os tempos gastos em toda a aplicação e na geração da árvore.

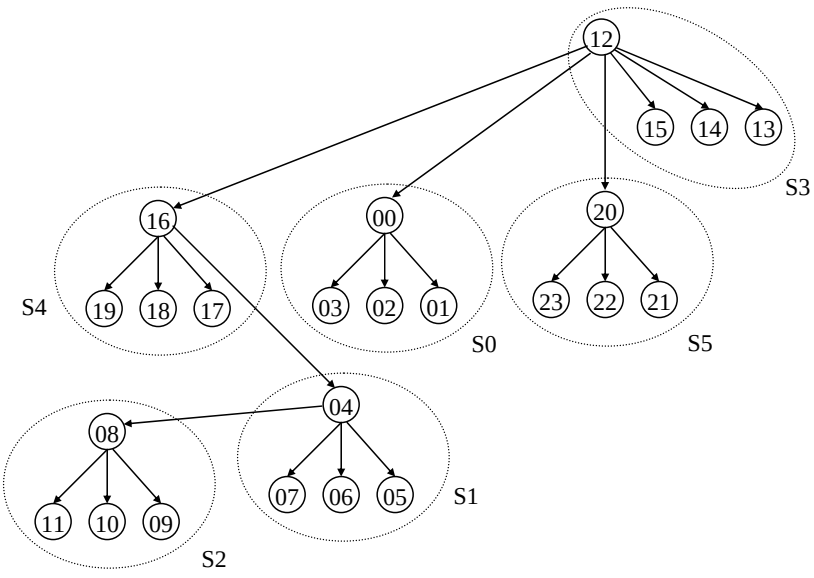


Figura 4.3: Árvore gerada pelo algoritmo *AGM*.

O algoritmo *AGM*, nestas execuções, não apresentou atualização da árvore e nenhuma latência sofreu modificação no seu valor. O objetivo destes testes foi mostrar o custo associado à criação desta estrutura para disseminar a informação e apontar a partir de

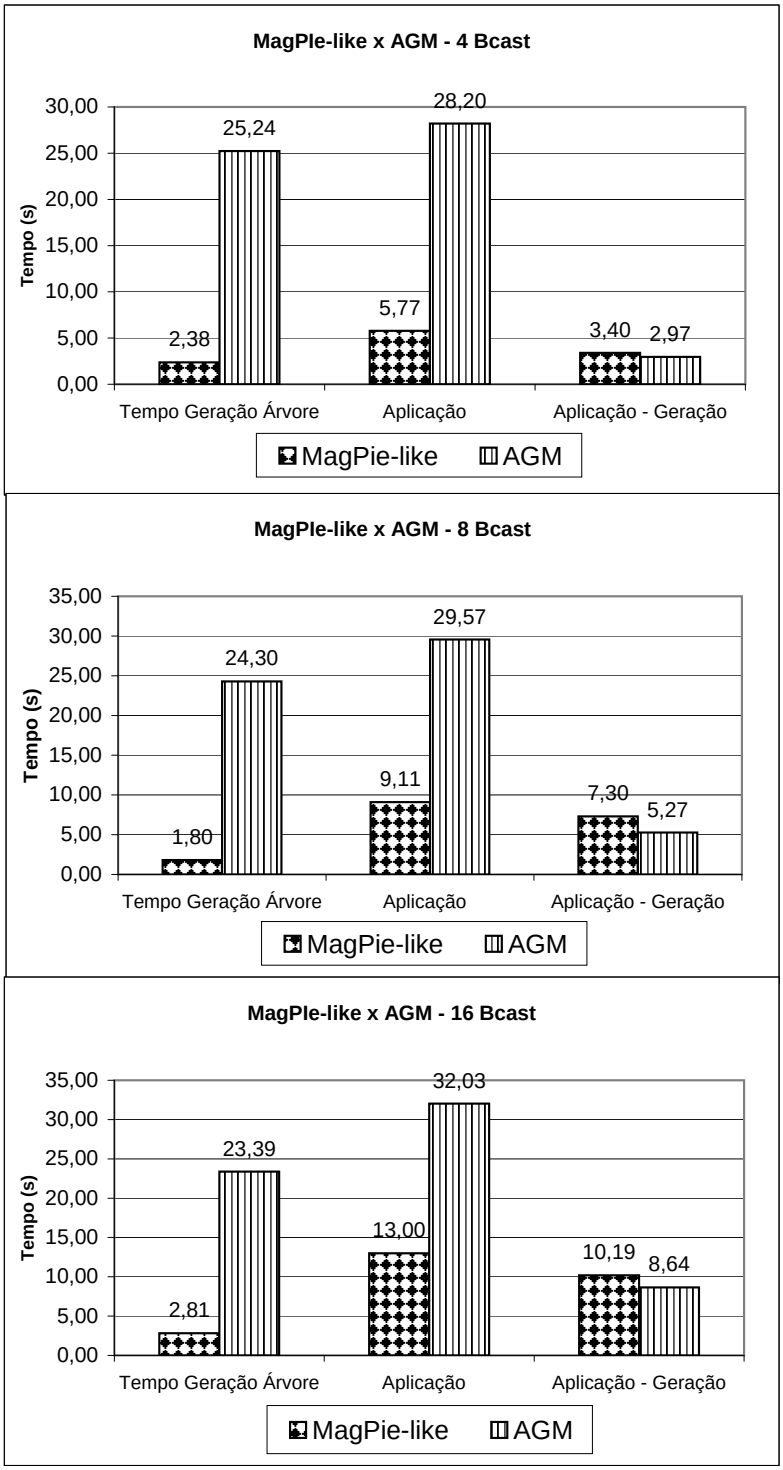


Figura 4.4: Comparação entre os algoritmos *MagPIe-like* e *AGM*.

que momento a sua utilização torna-se vantajosa.

Analisando os gráficos da Figura 4.4, é fácil ver que a versão *MagPIe-like* mostrou-se mais rápida, em relação ao tempo total da aplicação, em todas as execuções. No entanto, comparando os tempos de criação da árvore em relação ao tempo total da aplicação verificou-se o seguinte: o algoritmo *MagPIe-like* utilizou 41%, 20% e 22% do tempo total da aplicação para a criação da árvore para as execuções com 4, 8 e 16 *broadcasts*, nesta ordem, sendo o restante do tempo usado para a disseminação da informação. Por outro lado, a *AGM* possui um custo maior na construção da árvore: 90%, 82% e 73% para 4, 8 e 16 *broadcasts*.

É importante notar que tanto na Figura 4.4, como nas próximas, para cada uma das três execuções o tempo de cada etapa é o maior valor obtido dentre todos os processos que compõem a aplicação. O valor final apresentado nas figuras é a média destes três valores máximos.

O terceiro conjunto de barras dos gráficos, "Aplicação - Geração", representa o tempo destinado à disseminação da informação através das operações MPI_Bcast, observa-se que, para todos os três casos, a *AGM* teve um desempenho melhor que *MagPIe-like*. Para as execuções com 4, 8 e 16 *broadcasts* consecutivos a *AGM* obteve uma melhoria de 13%, 28% e 15% em seus tempos, respectivamente. Portanto, pela análise deste conjunto de barras, é possível ver que a árvore utilizada pelo algoritmo *AGM* para a realização do *broadcast* se mostra mais eficiente.

A utilização da *AGM* torna-se mais vantajosa à medida que as latências entre os processos das árvores estáticas, geradas pelo algoritmo *MagPIe-like*, sofrerem variações mais significativas e a quantidade de MPI_Bcast's for incrementada na aplicação. Esta observação é confirmada nos testes apresentados mais adiante.

Outra observação importante é que *AGM* pode ter mais de uma comunicação entre *sites* durante a difusão das mensagens até alcançar um destino, como ocorre com o *MPICH-like*. Porém, é importante lembrar que isto ocorre somente quando o custo total é minimizado, o que não é garantido pelo *MPICH-like*.

Na Tabela 4.3 é feito um comparativo sobre a quantidade média de mensagens trocadas entre os processos durante a geração da árvore e a execução da operação coletiva para a execução dos algoritmos *MagPIe-like* e *AGM* com *4*, *8* e *16 Bcasts*. A primeira coluna descreve os tipos de mensagens trocadas, na segunda e terceira colunas são apresentados os algoritmos *MagPIe-like* e *AGM* com as suas quantidades de mensagens, respectivamente.

	4 Bcasts		8 Bcasts		16 Bcasts	
	MagPIe-like	***AGM***	***MagPIe-like***	***AGM***	***MagPIe-like***	***AGM***
Bcast	92,0	92,0	184,0	184,0	368,0	368,0
Initiate	0,0	72,0	0,0	72,0	0,0	72,0
Test	0,0	394,0	0,0	406,7	0,0	390,7
Accept	0,0	48,0	0,0	48,0	0,0	48,0
Connect	0,0	32,0	0,0	32,0	0,0	32,0
Reject	0,0	174,0	0,0	160,0	0,0	177,3
Report	0,0	72,0	0,0	72,0	0,0	72,0
Gosleep	0,0	23,0	0,0	23,0	0,0	23,0
Total	92,0	907,0	184,0	997,7	368,0	1183,0

Tabela 4.3: Total de Mensagens trocadas nas execuções dos algoritmos *MagPIe-like* e *AGM*.

Especificamente na terceira linha da tabela (*Bcast*), são apresentados os totais de mensagens trocadas durante a execução da operação coletiva *MPI_Bcast* na aplicação, note que somente esta mensagem é enviada pelo algoritmo *MagPIe-like*, uma vez que os demais tipos são responsáveis pela construção da árvore *AGM*.

É possível perceber que o algoritmo *AGM* apresenta uma quantidade de mensagens bastante superior que o *MagPIe-like* como pode ser observado pela última linha da tabela. O total de mensagens trocadas entre todos os processos pelo algoritmo *AGM* é igual a *907*, *997* e *1183* e o algoritmo *MagPIe-like* *92*, *184* e *368*, os valores apresentados por estes algoritmos referem-se às execuções com *4*, *8* e *16* *broadcasts*, respectivamente. A quantidade de mensagens enviadas pelo *AGM* é aproximadamente 10, 5 e 3 vezes superior ao *MagPIe-like* para as execuções com 4, 8 e 16 *broadcasts*, ou seja o impacto da construção da *AGM* reduz com o aumento do número de *broadcasts*.

4.3 Comparação entre *MagPIe-like*, *AGM* e *AGM-adap*

O algoritmo distribuído que constrói a *AGM* pode ser executado de duas formas: com e sem adaptação da árvore geradora mínima após a definição inicial da árvore. Caso a opção de execução seja com adaptação da árvore, a cada execução da operação MPI_Bcast, é avaliado se há necessidade de atualização da árvore geradora mínima. Nos testes, esses algoritmos foram identificados como *AGM* e *AGM-adap*, o primeiro somente constrói a árvore e o segundo, além da construção, realiza a manutenção da árvore durante toda a aplicação.

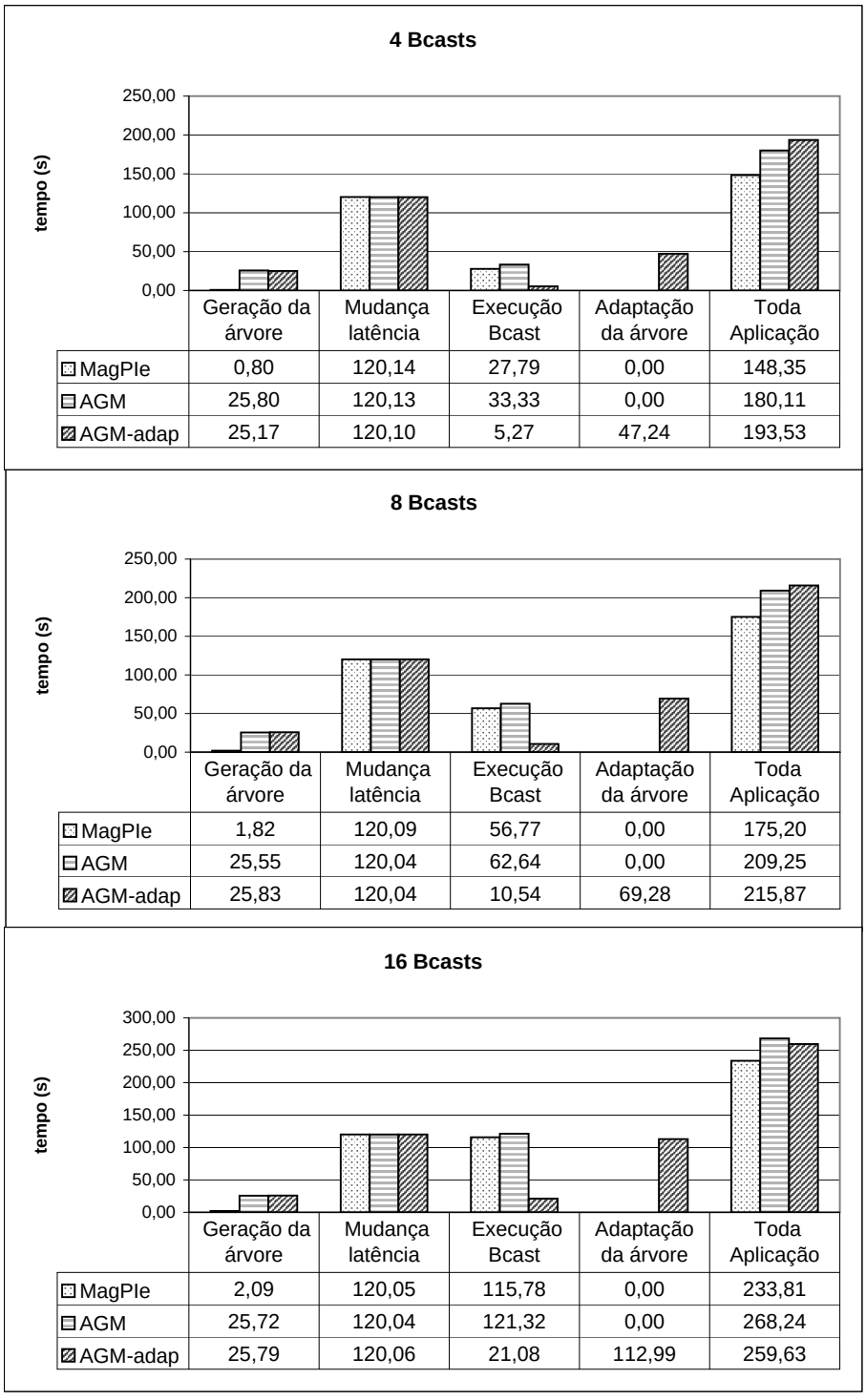
A Figura 4.5 apresenta os resultados dos algoritmos *MagPIe-like*, *AGM* e *AGM-adap* com execuções para 4, 8 e 16 *broadcasts* com ocorrência de duas falhas no início da aplicação, antes da execução do primeiro *broadcast*. As falhas ocorreram nos canais (4,6) e (12,16) que passaram a ter latências iguais a *4000,00* e *7331,12* milissegundos, respectivamente. A escolha dos canais que sofreram modificações nos seus valores ocorreu de forma aleatória.

Neste exemplo pode ser observado na primeira coluna na tabela do gráfico os tempos referentes à geração da árvore. Em todos os casos o tempo para a construção da árvore mostrou-se superior no caso do algoritmo *AGM*, como já foi observado na explicação do teste anterior. Mais uma vez este tempo maior refere-se à necessidade de troca de mensagens nos algoritmos para a construção inicial da árvore geradora mínima. O algoritmo *MagPIe-like* não precisa trocar mensagens para a construção da sua árvore, só necessita saber quem é a raiz do *broadcast*.

Neste exemplo, e nos demais que são apresentados, incluímos o tempo medido referente à "Mudança de Latência". Trata-se de um atraso imposto pela nossa aplicação, após a inclusão de novos valores de latências. Este artifício foi utilizado para garantir que no momento de uma nova consulta realizada pelo *NWS*, ele obtenha as informações atuais, evitando acessar dados da *cache*. Este atraso não inclui nenhum processamento no algoritmo, apenas aguarda *120* segundos após a inserção de novo valor de latência para garantir que todos os processos estão trabalhando com os dados atuais. Isto foi feito porque, em testes iniciais, foi percebido que o NWS não reconhecia imediatamente a mudança do valor da latência inserida pela aplicação.

A terceira coluna apresenta os tempos gastos para a execução de todos os *broadcasts* da aplicação. É importante observar que nas três execuções para *4*, *8* e *16* *broadcasts*, o tempo gasto pelo algoritmo *AGM* mostrou-se superior em relação ao *MagPIe-like*. Apesar de inicialmente o algoritmo *AGM* ser a árvore de menor custo, com a ocorrência de duas falhas seu desempenho mostrou-se pior. Com uma rápida análise das Figuras 4.2 e 4.3, que descrevem as topologias dos algoritmos *MagPIe-like* e *AGM*, respectivamente, já é possível notar que a árvore do AGM tem que aguardar um tempo maior para completar a disseminação de suas mensagens. Isto acontece, porque o aumento dos valores da latência ocorreu em canais da AGM que repassam a mensagem recebida para dar prosseguimento na disseminação das mensagens, no caso do algoritmo *MagPIe-like* o caminho, onde houve a mudança de latência, não é crítico como no AGM.

De acordo com as observações realizadas sobre os tempos obtidos pelo algoritmo



	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	0,80	120,14	27,79	0,00	148,35
AGM	25,80	120,13	33,33	0,00	180,11
AGM-adap	25,17	120,10	5,27	47,24	193,53

	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	1,82	120,09	56,77	0,00	175,20
AGM	25,55	120,04	62,64	0,00	209,25
AGM-adap	25,83	120,04	10,54	69,28	215,87

	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	2,09	120,05	115,78	0,00	233,81
AGM	25,72	120,04	121,32	0,00	268,24
AGM-adap	25,79	120,06	21,08	112,99	259,63

Figura 4.5: Execução para 4, 8 e 16 Bcasts com falhas para as versões *MagPIe-like*, *AGM* e *AGM-adap*.

AGM, nota-se a necessidade da realização de uma adaptação na árvore, antes do envio de novas mensagens, pois a utilização de estruturas estáticas nem sempre são vantajosas. A terceira linha com os tempos de execução para a versão do algoritmo que também cria uma árvore geradora mínima inicialmente, comprova o benefício em se adaptar a estrutura da árvore antes de realizar uma operação coletiva. Observe que nas três execuções o tempo médio de execução da operação coletiva, no caso de adaptação da árvore, mostrou-se aproximadamente *6* vezes mais rápido quando comparada com a versão que não executa a adaptação.

A última coluna apresenta os tempos gastos em toda a aplicação pelos três algoritmos. A observação a ser feita é que o tempo gasto para a construção e adaptação da árvore comprometem praticamente todo o tempo da aplicação para a versão *AGM-adap*. À medida que a quantidade de *broadcasts* vai aumentando o tempo total da aplicação vai se aproximando dos tempos obtidos pelos outros algoritmos, pois neste momento ele começa a utilizar freqüentemente a árvore de custo mínimo para disseminar mensagens por todos os processos.

Na Figura 4.6 é apresentada a árvore final gerada após as adaptações realizadas pelo algoritmo *AGM-adap*.

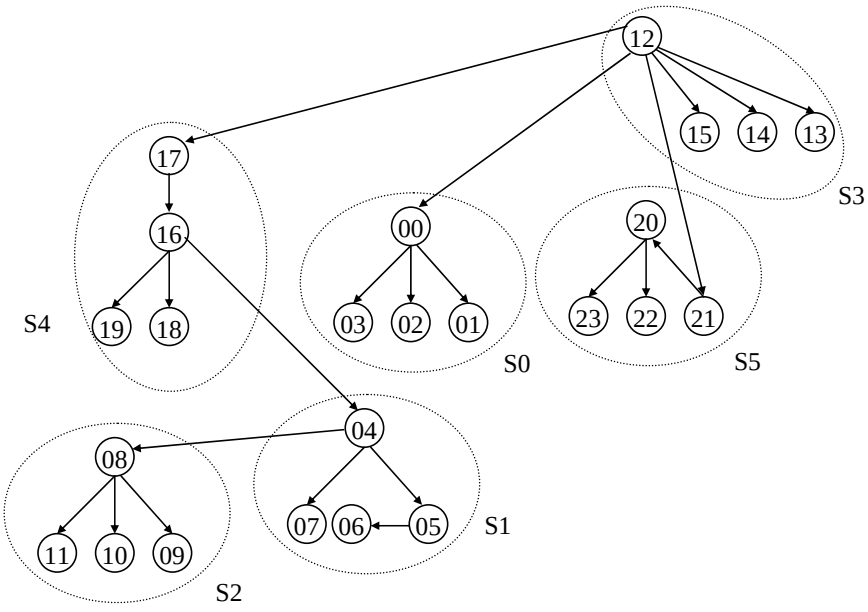


Figura 4.6: Árvore gerada pelo algoritmo *AGM-adap* após a realização da adaptação.

Outro teste realizado e apresentado pela Figura 4.7, mostra os resultados dos algoritmos *MagPIe-like*, *AGM* e *AGM-adap* com execuções para 4, 8 e 16 *broadcasts*. Nestes testes foram identificadas ocorrências de falhas e recuperações.

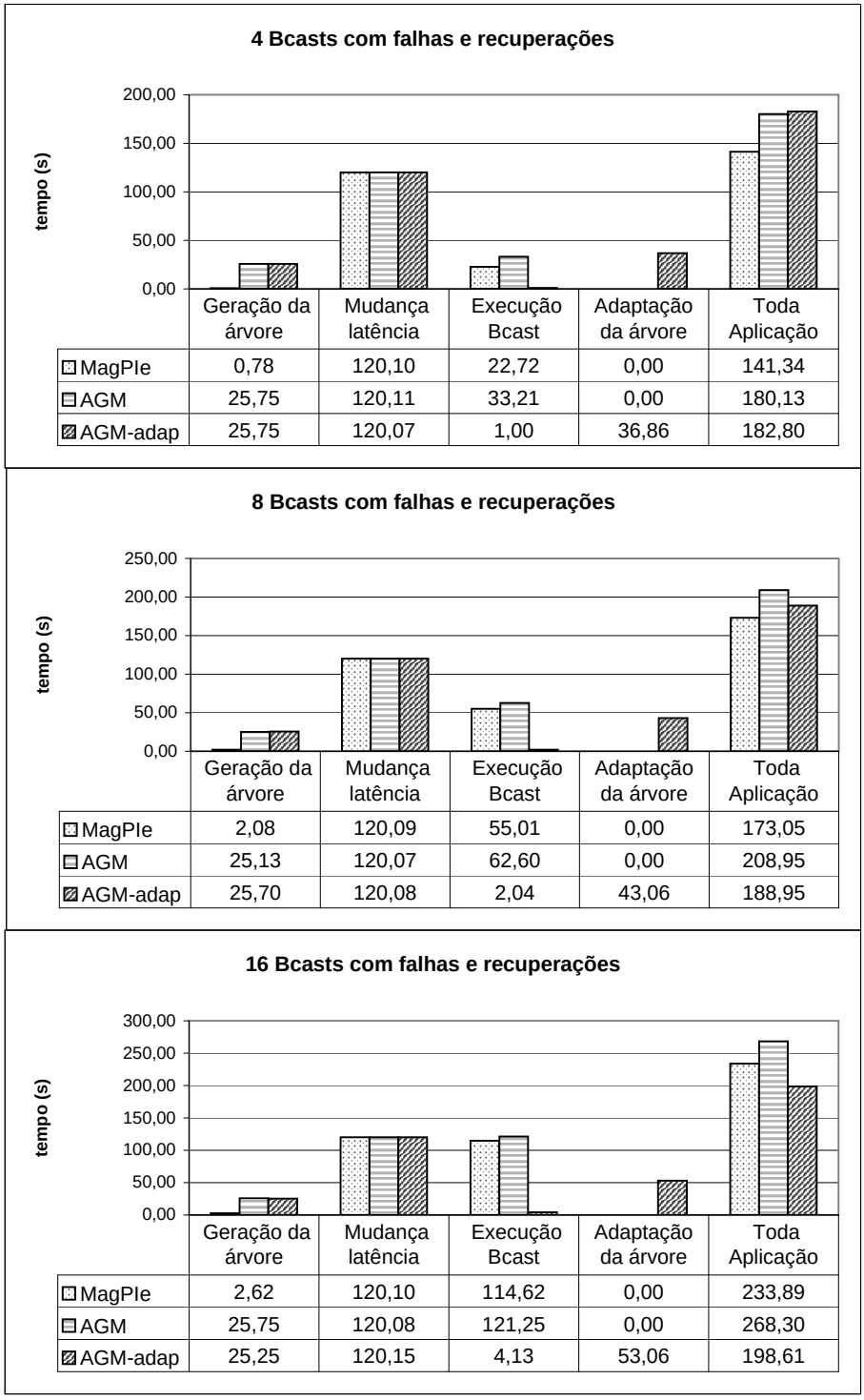


Figura 4.7: Execução para 4, 8 e 16 Bcasts com falhas e recuperações para as versões *MagPIe-like*, *AGM* e *AGM-adap*.

Durante a aplicação foram atualizados os valores de latências para os canais (4,6) e (12,16) que tiveram aumento no valor de suas latências para *4000,00* e *7331,12* milissegundos e os canais (0,21), (3,9), (6,12) (13,14) que tiveram suas latências diminuídas para *51,021*; *99,039*; *84,061* e *0,00* milissegundos, respectivamente. Os canais e os seus novos valores foram escolhidos de forma aleatória. A variação de latência do canal (13,14) foi mínima, uma vez que seu valor já era considerado próximo a zero. Outra observação sobre este canal é que não será realizado nenhum tratamento de recuperação para ele, pois já faz parte da árvore. Portanto, nestas execuções das seis mudanças nos valores de latências apresentadas, apenas cinco foram realmente tratadas: duas falhas e três recuperações.

As seguintes observações podem ser feitas em relação a estas execuções, a primeira coluna apresenta os tempos para a geração das árvores. Como já era esperado o tempo total da criação da árvore para o algoritmo *MagPIe-like* é muito inferior ao tempo gasto pelas versões que criam a árvore geradora mínima. Em relação aos tempos de *AGM* e *AGM-adap*, estas versões utilizam o mesmo algoritmo para a criação da árvore inicial, justificando o fato de que os tempos obtidos por estes dois algoritmos serem praticamente os mesmos. A segunda coluna refere-se ao tempo imposto pela aplicação para a garantia de obtenção dos valores mais recentes das latências, já explicado anteriormente. A terceira coluna apresenta os tempos utilizados para a execução da operação coletiva *MPI_Bcast*, nas três execuções para *4*, *8* e *16 broadcasts*, o comportamento entre os algoritmos estáticos foram semelhantes aos apresentados no teste anterior, onde o algoritmo *AGM* teve desempenho muito ruim com a ocorrência das falhas em relação ao algortimo *MagPIe-like*. Em contrapartida, o desempenho do algoritmo *AGM-adap* mostrou-se muito superior em relação aos outros dois algoritmos. Tanto para *4*, *8* e *16 broadcasts*, o desempenho mostrado do *AGM-adap* em relação ao *MagPIe-like* foi acima de vinte vezes mais rápido, enquanto que em relação à versão *AGM* seu desempenho ainda se mostrou melhor, aproximadamente, trinta vezes mais rápido.

Em relação à quarta coluna, que descreve o tempo gasto com a atualização da árvore, somente a versão *AGM-adap* mostrou tempos de execução para ela. Os tempos referentes à atualização da árvore equivalem a 20%, 23% e 26% do tempo total da aplicação para as execuções com *4*, *8* e *16 broadcasts*, respectivamente.

A última coluna mostra o tempo total da aplicação, em todos os testes o algoritmo *AGM* teve o pior desempenho e o algoritmo *AGM-adap* à medida que teve incrementado o número de *broadcasts* foi se aproximando do tempo de execução total do algoritmo *MagPIe-like*, até que na execução com *16 broadcasts* teve um desempenho melhor. Com

isto pode-se concluir que quando temos a necessidade de se executar um número elevado de *broadcasts* a versão *AGM-adap* tende a ser a mais apropriada para realização desta tarefa.

Na Figura 4.8 é apresentada a árvore final gerada após as adaptações realizadas pelo algoritmo *AGM-adap*.

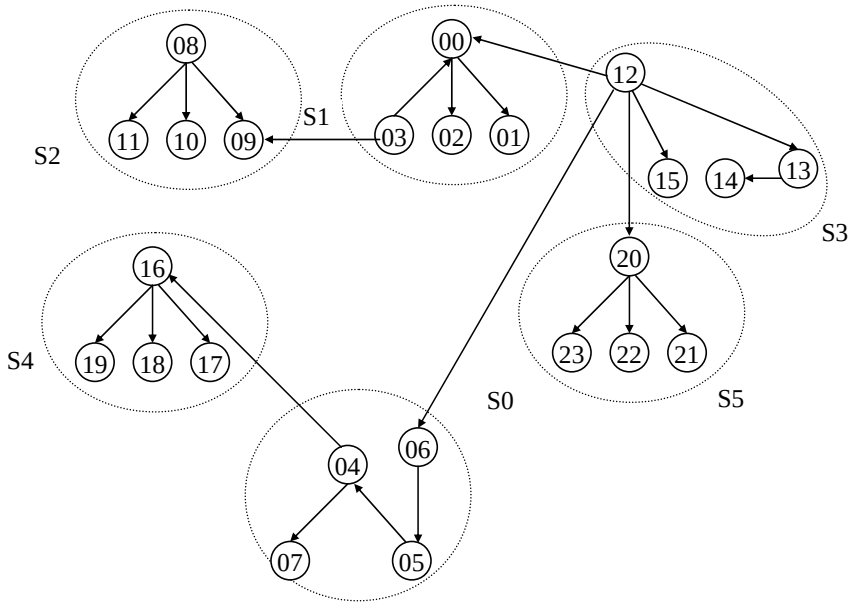


Figura 4.8: Árvore gerada pelo algoritmo *AGM-adap* após a realização da adaptação.

4.4 Comparação entre *MagPIe-like*, *AGM*, *AGM-adap* e *AGM-adapII*

Um fator importante, na versão *AGM-adap*, é que antes de cada execução da operação *MPI_ Bcast* sempre são verificadas se mudanças nos canais ocorreram. Este procedimento é feito para identificar possíveis falhas e recuperações, ele sempre será realizado após consulta, através do NWS, aos últimos valores coletados das latências.

Uma desvantagem neste procedimento de verificação dos valores das latências é que ele pode executar desnecessariamente, quando não ocorrem mudanças nas latências dos canais, adicionando um custo na aplicação, mesmo sem a ocorrência da adaptação na árvore. Embora o tempo para a verificação seja relativamente pequeno, 0.57 segundos em média, disponibilizamos uma versão que fornece uma opção que define o intervalo da quantidade de *broadcasts* em que é feita a análise dos valores das latências. As execuções que possuem esta opção são identificadas por *AGM-adapII* e podem ser vistas na Seção 4.4.1.

Outra desvantagem percebida é que sempre que ocorrem variações nos valores das latências, estas podem ser classificadas como uma falha ou como uma recuperação, o que provoca a atualização da árvore. Nem sempre este procedimento é vantajoso, principalmente, se a variação de latência for muito pequena, pois o procedimento para atualização da árvore pode se mostrar mais custoso do que utilizá-la sem adaptação em certos casos. Para minimizar este esforço de atualização desnecessária foi incluída uma opção de execução, que permite definir a partir de qual porcentagem da variação do valor da latência será realizada a atualização da árvore. Execuções com esta característica são apresentadas na Seção 4.4.2.

4.4.1 Execuções com a versão AGM-adapII com a definição do intervalo para análise dos valores das latências

A seguir são apresentadas execuções com a versão *AGM-adapII*. Os testes mostrados nesta seção referem-se a aplicações que executam *8*, *16* e *20 broadcasts*. Todos os testes foram realizados sobre a árvore original formada pelo algoritmo *AGM* que foi apresentada pela Figura 4.3.

Os testes serão apresentados conforme os seguintes grupos de execução:

- Execução para *8 broadcasts* com a ocorrência de uma, três e quatro falhas;
- Execução para *16 broadcasts* com a ocorrência de uma, três e quatro falhas;
- Execução para *8 broadcasts* com a ocorrência de uma, três e quatro recuperações;
- Execução para *20 broadcasts* com a ocorrência de duas falhas e duas recuperações;

Os dois primeiros conjuntos de testes, execuções com 8 e 16 *broadcasts*, referem-se a falhas ocorridas sobre os canais presentes na árvore, que ocorreram nas seguintes situações:

- Identificação de uma falha no canal *(4,6)* que passou a ter latência igual a *6001,00ms*;
- Ocorrência de três falhas nos canais *(4,6)*, *(12,16)* e *(12,20)* que passaram a ter latências iguais a *6001,00 ms*, *35120,12 ms* e *29669,16 ms*, respectivamente;
- Quando quatro falhas foram identificadas, os canais *(4,6)*, *(12,16)*, *(12,20)* e *(0,2)* passaram a ter latências iguais a *6001,00 ms*, *35120,12 ms*, *29669,16 ms* e *12001,02 ms*, respectivamente.

O conjunto de testes seguintes, para as execuções com 8 *broadcasts*, refere-se à recuperação de canais. Cada canal recuperado passa a ter o mesmo valor que apresentava na árvore *AGM* inicial, apresentada na Figura 4.3.

- Identificação da recuperação do canal *(4,6)*;
- Identificação das recuperações dos canais *(4,6)*, *(12,16)* e *(12,20)*;
- Identificação das recuperações dos canais *(4,6)*, *(12,16)*, *(12,20)* e *(0,2)*.

4.4.1.1 Execução para *8 broadcasts* com a ocorrência de uma, três e quatro falhas

A Figura 4.9 mostra os tempos de execução obtidos por cada algoritmo durante a realização da adaptação da árvore e a execução consecutiva de 8 *bcasts* para a ocorrência de uma, três e quatro falhas, respectivamente.

No primeiro gráfico nota-se que o tempo de execução para a geração da árvore do algoritmo *MagPie-like* demostrou ser o mais rápido em todos os casos. Para os demais algoritmos, os tempos de geração da árvore foram semelhantes, uma vez que em todos eles as falhas foram detectadas após a geração da árvore inicial e as três versões utilizam o mesmo algoritmo para a criação da árvore geradora mínima. A segunda coluna apresentada nos gráficos referem-se ao tempo médio esperado por todos os processos para garantir que a próxima consulta realizada no momento da execução da operação *MPI_Bcast*, todos eles tenham a certeza que consultaram os valores coletados mais recentes.

A partir da terceira coluna começam a ser percebidas as diferenças entre os testes realizados. A primeira observação é que a execução da operação coletiva melhora muito com a utilização do algoritmo adaptativo, à medida que o número de falhas aumenta, o tempo de execução da operação coletiva aumenta, principalmente nos casos dos algoritmos estáticos *MagPIe-like* e *AGM*. A diferença de tempo de execução desta operação varia, aproximadamente, de 18 a 70 vezes entre as execuções dos algoritmos estáticos (*MagPIe-like* e *AGM*) e os dinâmicos (*AGM-adap* e *AGM-adapII*) para as execuções com 1, 3 e 4 falhas.

Na quarta coluna de cada teste, observa-se que o tempo de adaptação da árvore diminui bastante quando utilizada a versão *AGM-adapII*. Isto ocorre porque imediatamente antes da execução da operação *MPI_Bcast*, o módulo de monitoração de rede é acessado para obter as mais recentes latências coletadas. Com estes dados verifica-se se há necessidade de adaptação da árvore. Quando nenhuma modificação do valor de canal

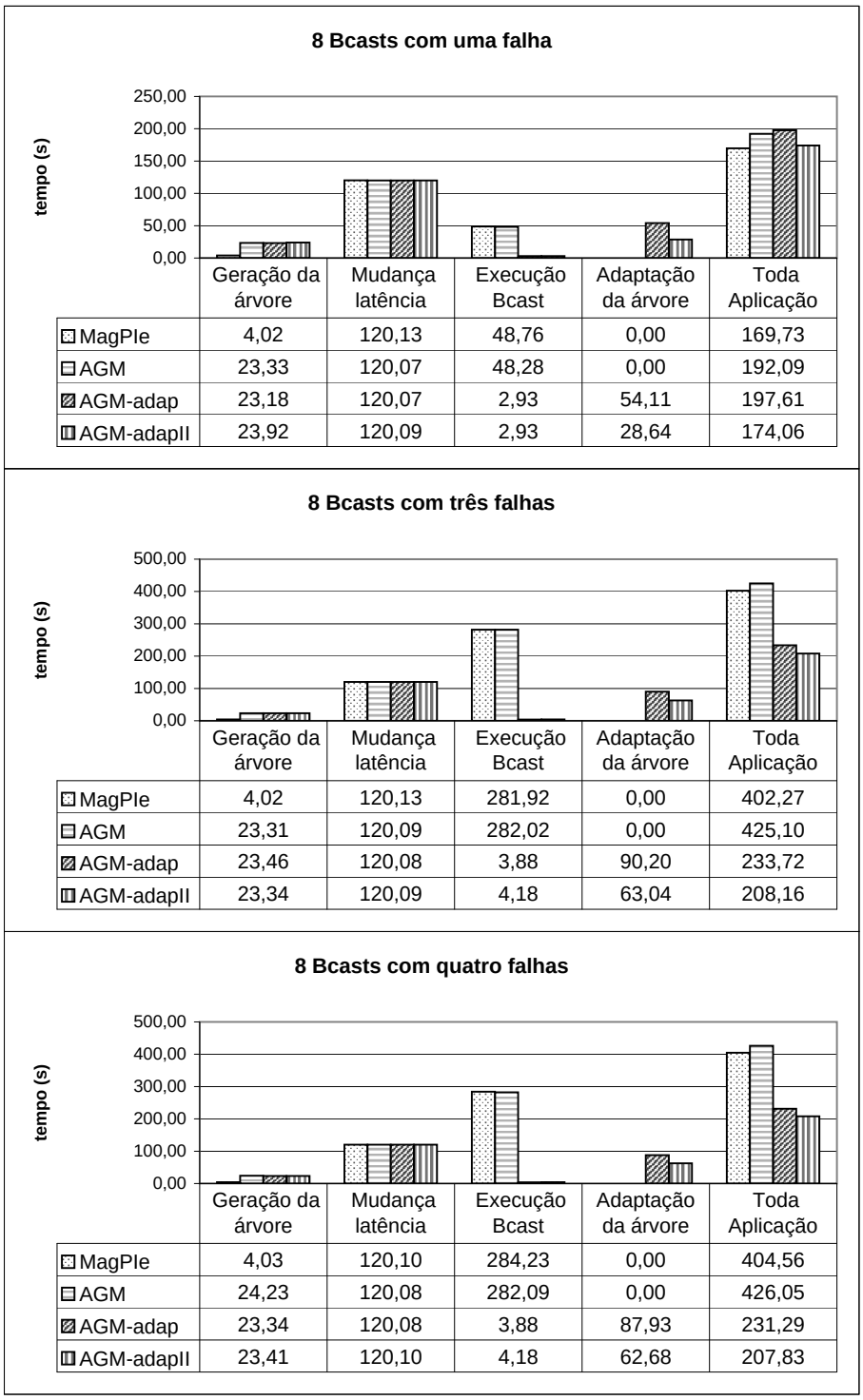


Figura 4.9: Tempos de execução para as quatro versões de algoritmos com uma, três e quatro falhas para oito *bcasts*.

é identificada, o procedimento de Terminação da Adaptação é executado, conforme foi mostrado na Figura 3.1 do Capítulo 3. Este procedimento gasta um certo tempo de processamento mesmo quando nenhuma adaptação na árvore é necessária, pois, de qualquer maneira, os processos precisam trocar mensagens informando sobre a não ocorrência de falhas e recuperações.

Como na versão *AGM-adapII* a verificação de mudanças de latências somente ocorre em certos intervalos de *broadcasts*, que é um parâmetro configurável, o procedimento de Terminação da Adaptação também só é executado nestes intervalos, diminuindo o tempo de processamento da aplicação. Nas execuções destes testes, a verificação de mudança de latência para o algoritmo *AGM-adapII* ocorre a cada 4 *bcasts* executados, enquanto que a versão *AGM-adap* faz esta verificação, a cada execução da operação *MPI_Bcast*.

A última coluna informa o tempo médio total gasto por cada algoritmo para executar toda a aplicação. Nota-se que quando o número cresce de uma para três falhas, o tempo total do algoritmo fica cerca de duas vezes maior para os algoritmos estáticos e no caso dos algoritmos dinâmicos o aumento é de aproximadamente 16%. Quando comparamos os tempos finais dos algoritmos dinâmicos em relação aos estáticos, percebe-se que os algoritmos dinâmicos passam a ter um desempenho melhor à medida que mais variações de latências ocorrem.

4.4.1.2 Execução para *16 broadcasts* com a ocorrência de uma, três e quatro falhas

Na Figura 4.10 são apresentados os tempos de execução obtidos para cada algoritmo durante a realização da adaptação da árvore e execução consecutiva de 16 *bcasts* para a ocorrência de uma, três e quatro falhas, respectivamente.

As observações feitas para as execuções com *8 broadcasts* também se aplicam nos testes realizados com *16 broadcasts*. Deve-se ressaltar, que o tempo gasto, por toda a aplicação, para a execução de *8 broadcasts* em relação à ocorrência de uma e três falhas, para os algoritmos estáticos *MagPIe-like* e *AGM*, cresceu em média 23% e 21%. Em relação aos algoritmos dinâmicos *AGM-adap* e *AGM-adapII*, o aumento de tempo foi de 16% e 6%, respectivamente. Quanto às execuções com *16 broadcasts*, os algoritmos *MagPIe-like* e *AGM* apresentaram aumento de 40% do tempo de execução, aproximadamente, enquanto que os algoritmos *AGM-adap* e *AGM-adapII* tiveram acréscimo de tempo próximos a 14% e 4%.

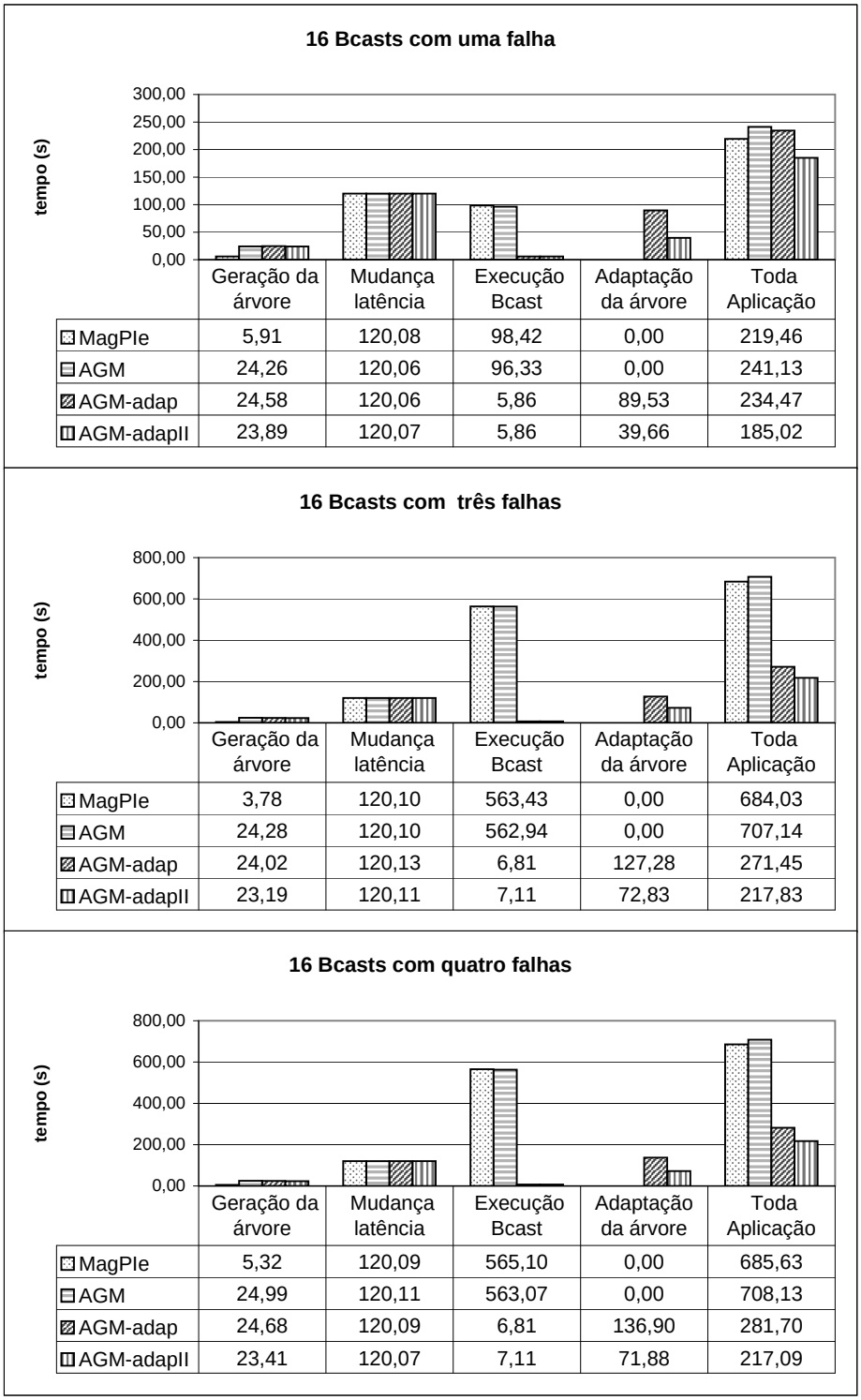


Figura 4.10: Tempos de execução para as quatro versões de algoritmos com uma, três e quatro falhas para dezesseis *bcasts*.

4.4.1.3 Execução para *8 broadcasts* com a ocorrência de uma, três e quatro recuperações

É apresentado na Figura 4.11 os tempos de execução obtidos para cada algoritmo durante a execução com 8 *broadcasts* onde ocorreram uma, três e quatro recuperações, respectivamente.

Os tempos de execução da operação coletiva foram melhores em todas as execuções para os algoritmos que realizam as adaptações da árvore. Como neste exemplo, os canais recuperados já fazem parte da topologia mantida pelo algoritmo *MagPIe-like*, este algoritmo se beneficiou dessas mudanças de latências, que se refletiu no tempo total da aplicação. Este algoritmo apresentou o melhor tempo de execução da aplicação em todos os casos. Observa-se que a maior parte do tempo da aplicação gasto pelos algoritmos dinâmicos referem-se à geração e manutenção da árvore. Em todos os casos, a versão *AGM-adapII* demonstrou melhor desempenho em relação à execução do algoritmo *AGM-adap*.

4.4.1.4 Execução para *20 broadcasts* com duas falhas e duas recuperações

Outro experimento realizado foi para a execução de 20 *broadcasts* consecutivos, como mostra a Figura 4.12, onde falhas e recuperações ocorreram em momentos distintos. Aqui a porcentagem de variação foi igual a 0%, ou seja, qualquer variação, mesmo que pequena, foi tratada como uma falha ou como uma recuperação.

A necessidade de atualização da árvore, neste teste, ocorreu durante a execução do 4º, 8º, 12º e 16º *broadcast*, quando foram identificadas uma falha no canal *(4,6)*, uma recuperação no canal *(0,2)*, outra falha no canal *(12,20)* e outra recuperação apresentada no canal *(12,16)*, respectivamente. Pode-se observar que a execução da operação MPI_Bcast mostrou um desempenho muito superior, para as versões *AGM*, quando comparadas com a *MagPIe-like*. O tempo total da aplicação para a versão *AGM-adap* ficou um pouco maior que o *MagPIe-like*, devido, principalmente, ao custo da manutenção da árvore. Em relação à versão *AGM-adapII*, o tempo da aplicação mostrou-se melhor com a diminuição do tempo na análise de latências e adaptação da árvore.

A Figura 4.13 ilustra a situação antes e depois da ocorrência das mudanças dos valores nos canais. A parte superior desta figura mostra o momento em que os processos percebem as recuperações de seus canais adjacentes, identificados por linhas tracejadas e as falhas nos canais, que foram marcados com "X". A parte inferior mostra a árvore

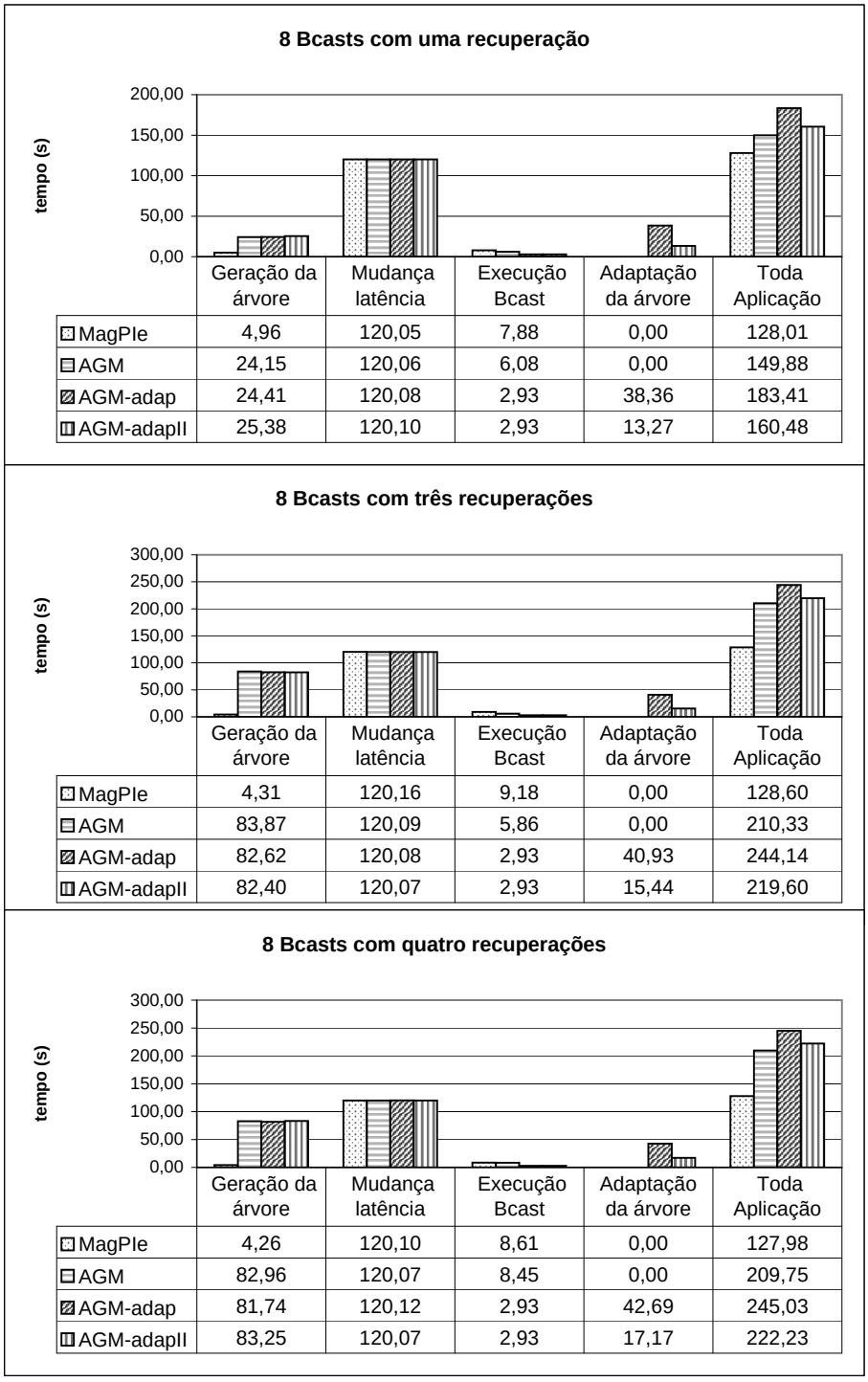
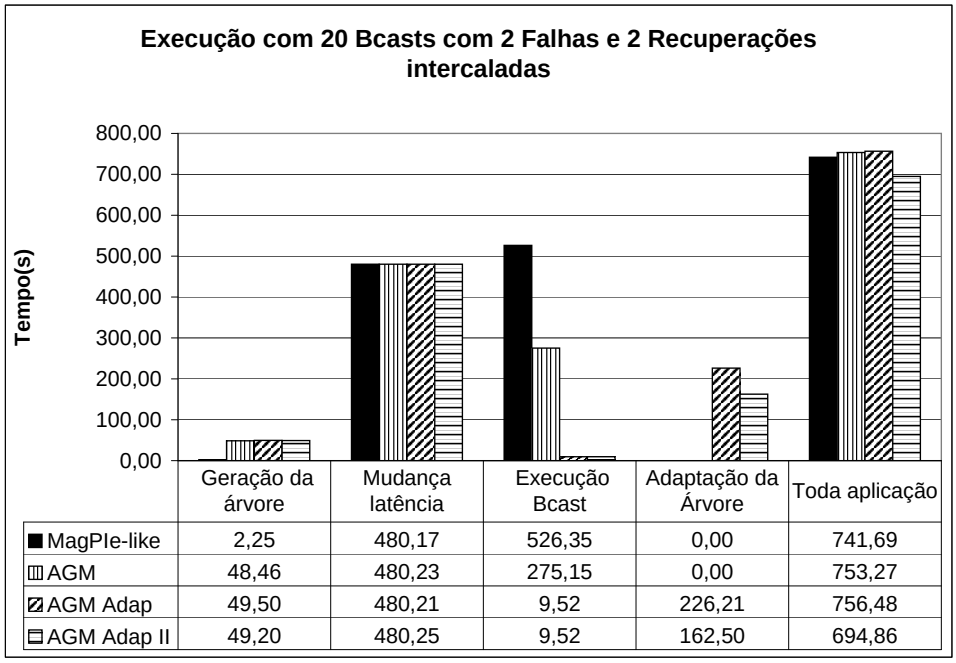


Figura 4.11: Tempos de execução para as quatro versões de algoritmos com uma, três e quatro recuperação para oito *bcasts*.



	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da Árvore	Toda aplicação
MagPIe-like	2,25	480,17	526,35	0,00	741,69
AGM	48,46	480,23	275,15	0,00	753,27
AGM Adap	49,50	480,21	9,52	226,21	756,48
AGM Adap II	49,20	480,25	9,52	162,50	694,86

Figura 4.12: Comparação dos algoritmos para execução com 20 bcast's

geradora mínima após passar pelo processo de adaptação.

Na Tabela 4.4 são apresentados os tipos e as quantidades de mensagens trocadas para as versões *MagPIe-like*, *AGM* que cria somente a árvore inicialmente e a versão *AGM-adap* que realiza a adaptação da árvore geradora mínima. O objetivo desta tabela é mostrar o aumento expressivo da quantidade de mensagens trocadas à medida que utilizamos algoritmos para a criação da árvore geradora mínima.

4.4.2 Execução com porcentagem de variação da latência dos canais

4.4.2.1 Execução para *16 broadcasts*

Nas Figuras 4.14 e 4.15 foram realizadas execuções com 16 *broadcasts* consecutivos, tendo como parâmetros os valores 10% e 70% para a realização da adaptação da árvore. Novos valores nas latências foram inseridos após a geração da árvore inicial e, imediatamente, antes da execução da primeira operação MPI_Bcast. A inserção destes valores, artificialmente, provocou um acréscimo no tempo de execução da aplicação, que foi referenciado nestas figuras como "Mudança latência".

Analisando o gráfico da Figura 4.14, verifica-se que o tempo de geração da árvore inicial, como era de se esperar, na versão *MagPIe-like* mostrou-se menor, enquanto nas versões *AGM*, os tempos foram superiores ao *MagPIe-like*, devido à necessidade de troca

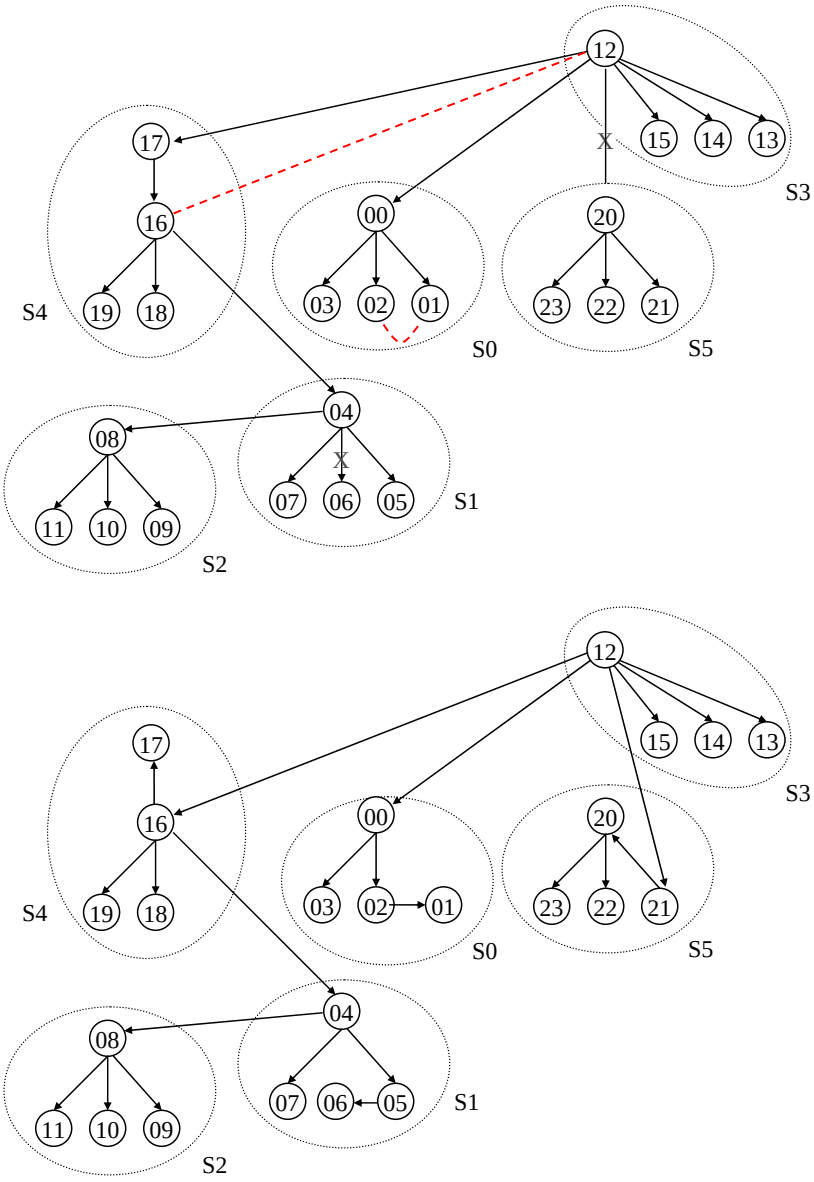
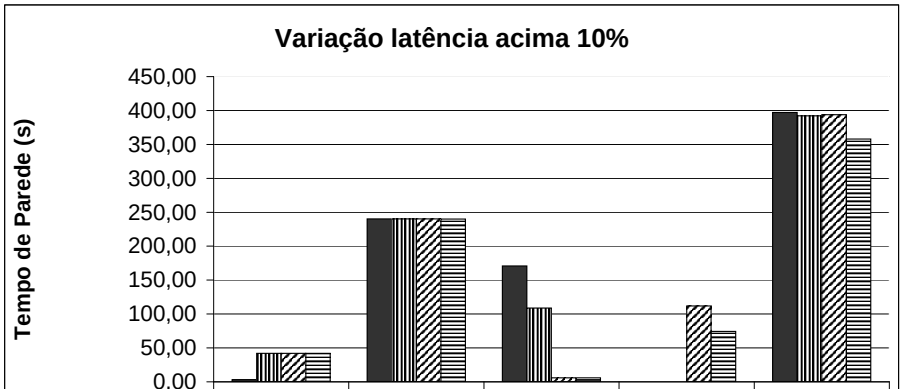


Figura 4.13: Representação das mudanças ocorridas nos canais e a adaptação da árvore.

	20 Bcasts		
	MagPIe-like	***AGM***	***AGM-adap***
Bcast	460	460,0	460,0
Initiate	0,0	72,0	77,0
Test	0,0	384,7	1171,3
Accept	0,0	48,0	52,0
Connect	0,0	32,0	34,0
Reject	0,0	185,3	412,7
Report	0,0	72,0	120,0
Change-root	0,0	1,0	1,0
Gosleep	0,0	23,0	23,0
Failure	0,0	0,0	3,0
Reiden	0,0	0,0	44,0
Reiden-ack	0,0	0,0	44,0
Findmoe	0,0	0,0	44,0
Id-Check	0,0	0,0	4,0
Recovery	0,0	0,0	5,0
Privilege	0,0	0,0	5,0
Replace	0,0	0,0	6,0
Not-rec	0,0	0,0	1057,0
Not-rec-ack	0,0	0,0	1741,0
Gosleep-rec	0,0	0,0	460,0
Not-fail	0,0	0,0	904,0
Not-fail-ack	0,0	0,0	451,0
Gosleep-fal	0,0	0,0	462,0
Total	460,0	1278,0	7583,0

Tabela 4.4: Total de Mensagens trocadas nas execuções dos algoritmos *MagPIe-like*, *AGM* e *AGM-adap* para 20 *Bcasts*.

de mensagens para a construção das suas árvores. A segunda coluna, mostra o atraso imposto, já citado anteriormente, para a inclusão do novo valor de latência. Para os tempos de execução da operação MPI_Bcast, nota-se que as versões *AGM-adap* e *AGM-adapII* obtiveram tempos muito inferiores em relação aos tempos gastos pela *AGM* e *MagPIe-like*. As duas primeiras versões foram, aproximadamente, 21 vezes mais rápidas que a *AGM* e 85 vezes mais rápidas do que a *MagPIe-like*.



	Geração da árvore	Mudança latência	Execução bcast	Adaptação da árvore	Toda aplicação
MaPIe-like	3,01	240,16	170,79	0,00	397,20
AGM	41,99	240,13	108,86	0,00	392,16
AGM - Adap	41,93	240,12	5,86	111,89	393,83
AGM - Adap II	42,03	240,09	5,86	74,31	357,94

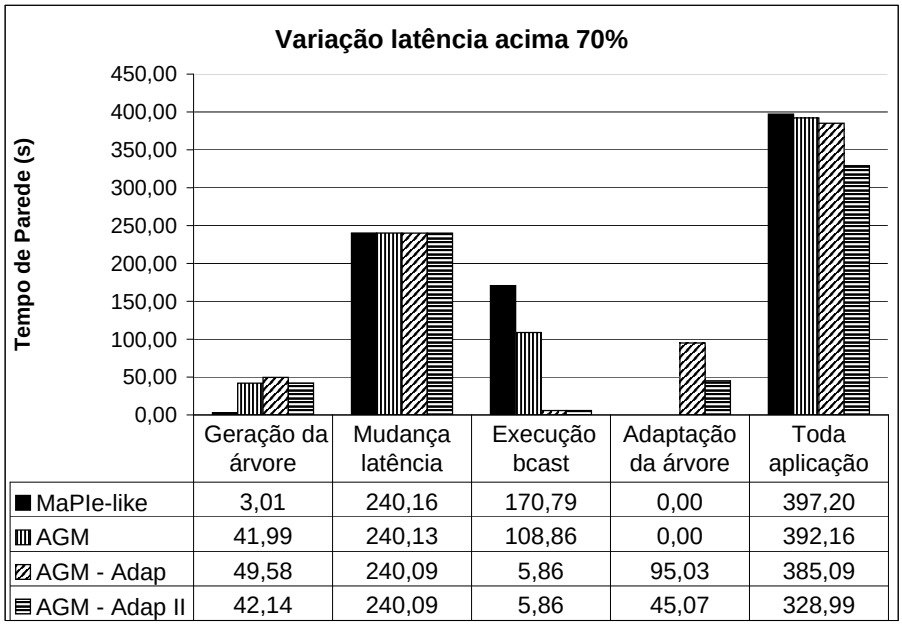
Figura 4.14: Atualização da árvore com variação no valor das latências superior a 10%.

Durante a execução destes testes, os seguintes canais tiveram seus valores alterados: (04,06): 1,00 → 9999,00 e (12,20): 35,00 → 45,00 com aumento das latências; (00,02): 120001,00 → 6000 e (12, 16): 30000,00 → 21,00 tiveram os valores das latências reduzidos.

A quarta coluna apresenta os tempos gastos para a análise das latências e adaptação da árvore, lembrando que a versão *AGM-adap* executa a cada *bcast* a análise de alteração das latências, enquanto a versão *AGM-adapII*, nestes testes, fez esta análise a cada quatro *broadcasts* executados. Isto justifica o seu melhor desempenho nos tempos gastos para adaptação da árvore e, conseqüentemente, em toda a aplicação. A última coluna informa o tempo total de execução da aplicação, o que leva a concluir que, mesmo com a inclusão da análise das latências e da adaptação da árvore, as versões *AGM-adap* e *AGM-adapII* demonstram ter um desempenho melhor em relação ao *MagPIe-like*.

Na Figura 4.15, as mesmas observações feitas para o teste anterior se aplicam aqui, acrescentando apenas o fato da variação das latências ter de ser igual ou superior a 70% do valor original da latência, para que seja realizada a atualização da árvore. Desta forma, o canal (12,20) continua pertencendo à árvore, pois o aumento do seu valor foi próximo a

28,5%. Como uma adaptação deixa de ser realizada, em relação ao teste anterior, observa-se uma melhoria nos tempos de adaptação da árvore e de toda a aplicação para as versões *AGM-adap* e *AGM-adapII*.



	Geração da árvore	Mudança latência	Execução bcast	Adaptação da árvore	Toda aplicação
MaPIe-like	3,01	240,16	170,79	0,00	397,20
AGM	41,99	240,13	108,86	0,00	392,16
AGM - Adap	49,58	240,09	5,86	95,03	385,09
AGM - Adap II	42,14	240,09	5,86	45,07	328,99

Figura 4.15: Atualização da árvore com variação no valor das latências superior a 70%.

4.5 Execução com vários *broadcasts* com ocorrência de falhas e recuperações.

Nesta seção são apresentados vários testes realizados para os algoritmos *MagPIe-like*, *AGM*, *AGM-adap* e *AGM-adapII*.

4.5.1 Execução para *40 broadcasts* com uma falha e duas recuperações

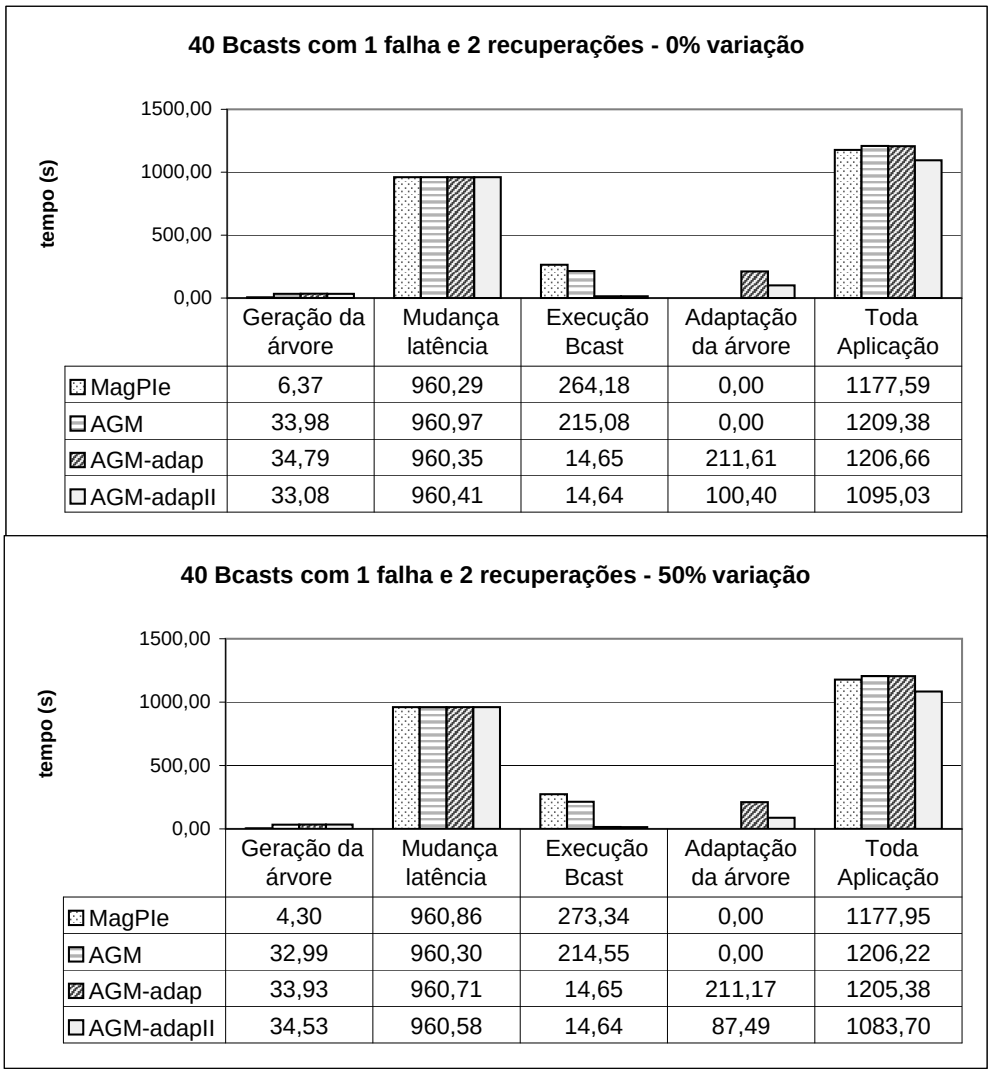
Nestes testes a seguir, analisamos a adaptação da árvore utilizando o parâmetro de porcentagem que indica a tolerância na variação da latência permitida, sem que seja considerada como a ocorrência de uma falha ou de recuperação. Nestes casos, é dispensada a atualização da árvore.

Foram realizadas duas execuções, onde na primeira foi utilizado o parâmetro de variação igual a 0%, isto significa que qualquer mudança de latência é vista como uma falha ou recuperação. O segundo teste apresenta o parâmetro de variação igual a 50%, assim sendo, se uma variação ocorrer até o limite de 50% do seu valor atual, não é realizada

atualização da árvore devido a mudança de valor desta latência.

Os canais que tiveram os valores de suas latências alterados foram: *(4,6)*, *(0,2)* e *(12,16)*. Uma falha ocorreu no canal *(4,6)* que, inicialmente, tinha valor igual a *1.00 ms*, passou a ter valor igual a *6000,00 ms* a partir do oitavo *bcast*. Duas recuperações ocorreram, uma no canal *(0,2)* que possuía latência igual a *10698,00 ms* e, a partir do décimo segundo *bcast*, seu valor mudou para *1,00 ms*. A segunda recuperação ocorreu no vigésimo *bcast*, quando o canal *(12,16)* com valor de latência igual *450.12 ms* passou a ter *330,12 ms*. Os demais valores de canais seguem as medidas presentes na Tabela 4.1 apresentada inicialmente.

A Figura 4.16 apresenta os tempos de execução para os algoritmos *MagPIe-like*, *AGM*, *AGM-adap* e *AGM-adapII* para execução de 40 *bcasts* para a variação de 0% e 50%.



	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	6,37	960,29	264,18	0,00	1177,59
AGM	33,98	960,97	215,08	0,00	1209,38
AGM-adap	34,79	960,35	14,65	211,61	1206,66
AGM-adapII	33,08	960,41	14,64	100,40	1095,03

	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	4,30	960,86	273,34	0,00	1177,95
AGM	32,99	960,30	214,55	0,00	1206,22
AGM-adap	33,93	960,71	14,65	211,17	1205,38
AGM-adapII	34,53	960,58	14,64	87,49	1083,70

Figura 4.16: Tempos de execução para as quatro versões de algoritmo com uma falhas e duas recuperações intercaladas para 40 *bcasts* para variação de 0% e 50%.

No primeiro gráfico são apresentados os tempos obtidos na ocorrência de uma falha e duas recuperações intercaladas com a procentagem de variação igual a 0%, logo, qualquer alteração de latência neste teste provocou atualizações na árvore. Já no segundo gráfico, desta mesma figura, há uma tolerância na mudança do valor de latência de acordo com a porcentagem de 50%. Somente duas mudanças de latências acarretaram atualizações na árvore neste teste. Assim, o canal (*12,16*) teve o valor de sua latência reduzido, mas isto não causou a adaptação da árvore para este canal, pois a sua redução não ultrapassou o limite de 50% definido.

Analisando os tempos, para a adaptação da árvore, verifica-se que o tempo obtido pelo *AGM-adapII* com o parâmetro igual a 50% foi melhor do que quando executado com o parâmetro de variação igual a 0%. Na execução com parâmetro de variação igual a 0%, uma recuperação foi tratada a mais.

4.5.2 Execução para *20 broadcasts* com a ocorrência de três falhas

A Figura 4.17 apresenta os tempos de execução para os algoritmos *MagPIe-like*, *AGM*, *AGM-adap* e *AGM-adapII* para execução de 20 *Bcasts*.

No primeiro gráfico são apresentados os tempos obtidos na ocorrência de três falhas. As falhas ocorreram após a metade da execução da aplicação, ou seja, a partir da execução do décimo *broadcast* foram modificados os valores das latências. Os canais que falharam foram (*4,6*), (*12,16*) e (*12,20*) que passaram a ter as seguintes latências: *30000,01 ms*; *35120,00 ms* e *6001,00 ms*, respectivamente.

Observa-se mais uma vez que a execução da operação coletiva para as versões que adaptam a árvore demonstraram um desempenho bastante superior, no mínimo 40 vezes. Outro comportamento interessante apresentado por esta execução foi o desempenho do algoritmo *AGM* mostrar-se inferior, aproximadamente 33%, ao do algoritmo *MagPIe-like*. Isto ocorreu devido às falhas ocorridas nos canais (12,16) e (4,6) que são pontos de ligação, nesta topologia, a outros *sites*. Portanto, os processos que precisam receber a mensagem passando por estes canais levam um tempo maior para processar a operação coletiva. A demora no repasse da mensagem por estes canais prejudica o tempo total da execução da operação coletiva. A falha desses canais não é tão ruim quando utilizado o algoritmo *MagPIe-like*.

Mesmo com tempos de execução para a adaptação elevada, para as versões *AGM-adap* e *AGM-adapII*, o tempo total de execução da aplicação é aproximadamente a metade do

tempo gasto pelas execuções que utilizam árvores estáticas. Note ainda que o tempo de adaptação para a versão *AGM-adapII* é praticamente a metade do tempo gasto pelo algoritmo *AGM-adap*.

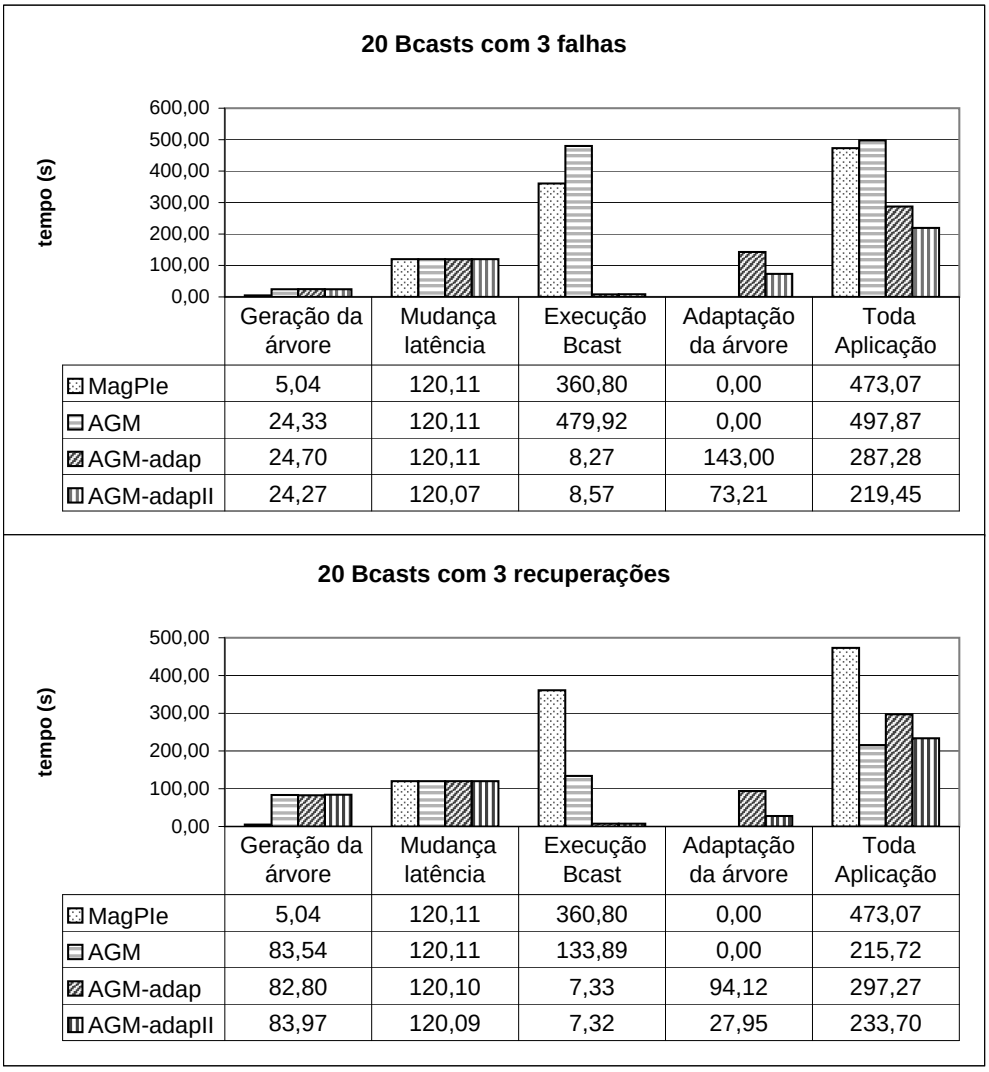


Figura 4.17: Tempos de execução para as quatro versões de algoritmo com três falhas e três recuperação para vinte *bcasts*.

4.5.3 Execução para *20 broadcasts* com a ocorrência de três recuperações

Veja o segundo gráfico da Figura 4.17, onde são apresentados os tempos obtidos na ocorrência de três recuperações.

Os tratamentos realizados, nesta execução, referem-se às recuperações dos canais $(4,6)$, $(12,16)$ e $(12,20)$ que voltam a ter os valores de latências iniciais, antes da ocorrência das falhas descritas anteriomente. As recuperações ocorrem imediatamente antes

da execução do décimo primeiro *broadcast* da aplicação.

Note que o tempo obtido na execução do algoritmo *MagPIe-like* é o mesmo apresentado no teste anterior, isto foi possível porque nesta execução o algoritmo inicia sua disseminação de mensagens utilizando os canais que sofreram aumento de latência na execução anterior. Como neste exemplo esses mesmos canais foram recuperados e passaram a ter os mesmos valores que possuíam antes das falhas, esses dois testes realizam o mesmo processamento. A diferença é que, no exemplo anterior, nas dez primeiras operações de *broadcasts* a mensagem é difundida antes da ocorrência de falhas pelos mesmos canais com latências menores, enquanto no exemplo atual isto ocorre a partir do décimo primeiro *broadcast*, após a mudança da latência destes canais.

O desempenho do algoritmo *AGM*, neste exemplo, mostrou-se muito superior ao do algoritmo *MagPIe-like*, ao contrário do que aconteceu anteriormente. Isto ocorreu porque, apesar do algoritmo *AGM* manter uma árvore estática durante toda a execução da aplicação, inicialmente ele constrói uma árvore de custo mínimo, descartando aqueles canais com latências elevadas que estão presentes na topologia construída pelo algoritmo *MagPIe-like*.

Mais uma vez, o desempenho do algoritmo *AGM-adapII* mostrou-se mais eficiente que o algoritmo *AGM-adap*. Isto se deve ao menor tempo gasto durante a execução do procedimento que analisa a necessidade de execução de adaptações.

4.5.4 Execuções para *100 broadcasts* com 4 falhas e com 4 recuperações separadamente

A Figura 4.18 apresenta os tempos de execução para os algoritmos *MagPIe-like*, *AGM* e *AGM-adap* para execução de 100 *Bcasts*.

As alterações de latências foram incluídas a partir da execução do quinquagésimo *bcast*. No primeiro gráfico são apresentados os tempos obtidos na ocorrência de quatro falhas e no segundo são apresentados os tempos para a recuperação de quatro canais.

As falhas ocorreram nos canais (*4,6*), (*1,0*), (*12,16*) e (*8,9*), com mudanças dos valores para (*4000,0 ms*), (*10,0 ms*), (*7331,0 ms*) e (*60,0 ms*).

As recuperações referem-se à execução apresentada no segundo gráfico, onde foram realizadas as recuperações dos canais (*6,12*), (*0,21*), (*13,14*) e (*3,9*), que passaram a ter valores de latências iguais a (*84,0 ms*), (*1,0 ms*), (*0,1 ms*) - que continua tendo um valor próximo a zero e (*19,0 ms*).

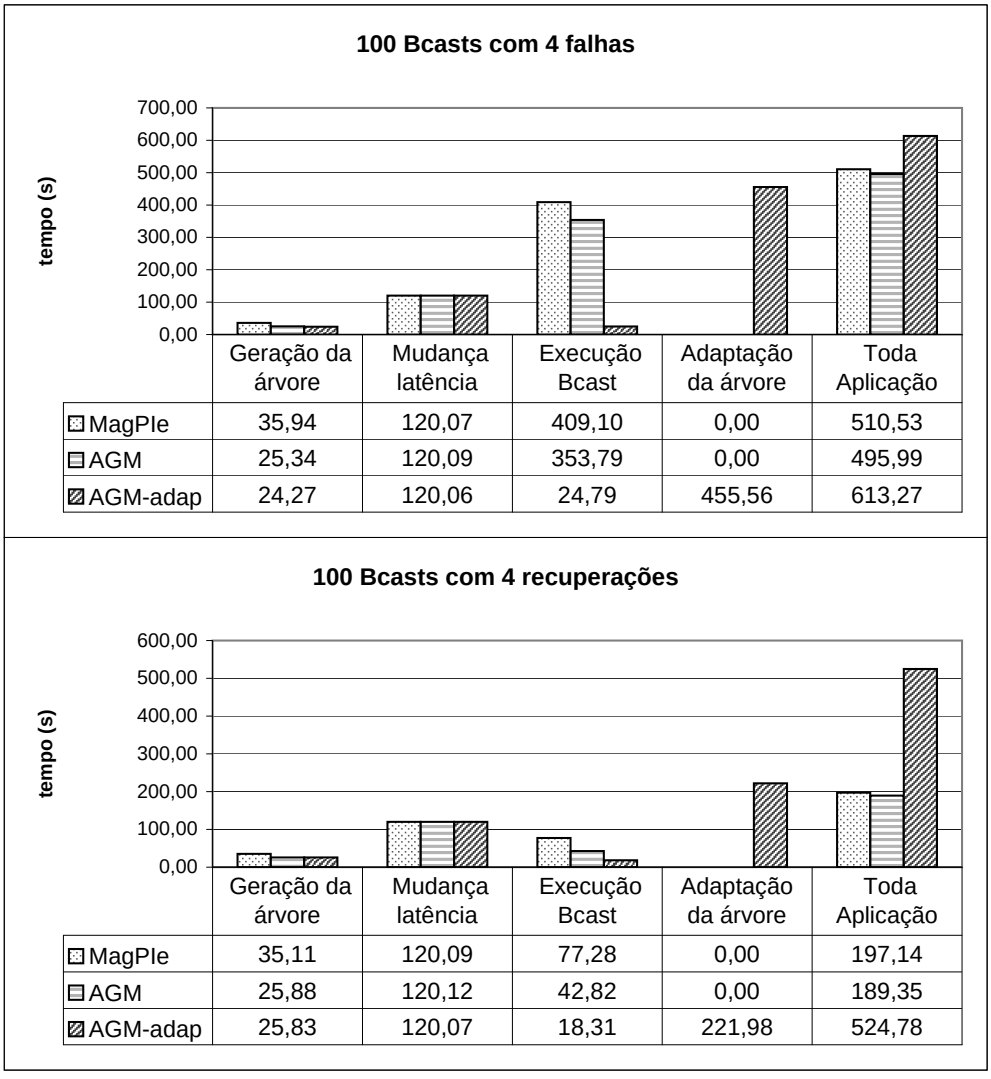
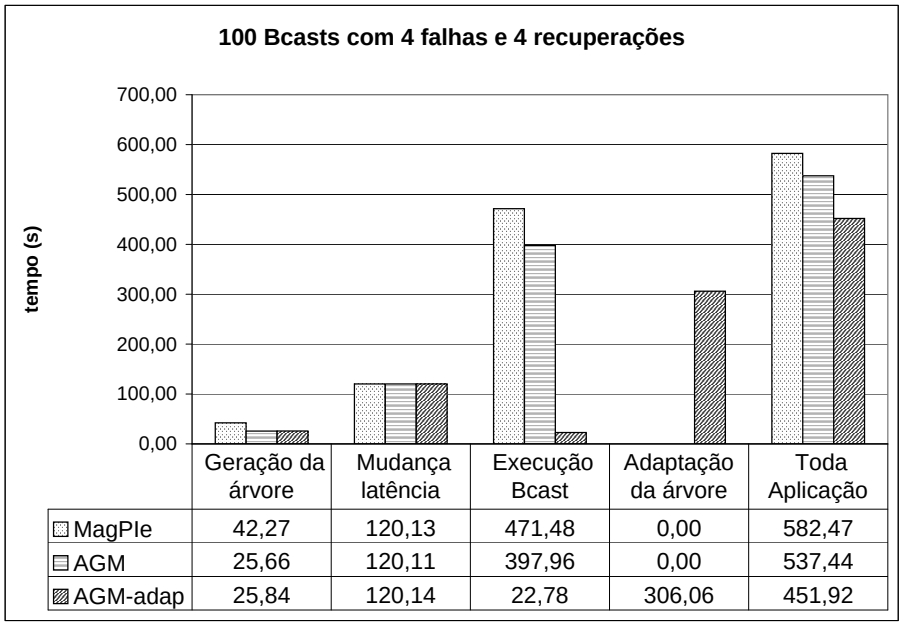


Figura 4.18: Tempos de execução para as quatro versões de algoritmo com quatro falhas e quatro recuperação para 100 *bcasts*.

Nestas duas execuções nota-se um bom desempenho para a execução da operação coletiva utilizando a árvore que foi adaptada. Outra observação é o tempo gasto no processamento da adaptação da árvore, sendo boa parte referente ao procedimento de verificação já explicado, onde mensagens de terminação são trocadas entre os processos, para que todos saibam que não há falhas nem recuperações a serem tratadas.

4.5.5 Execuções para *100 broadcasts* com 4 falhas e com 4 recuperações, simultaneamente

A Figura 4.19 apresenta os tempos de execução para os algoritmos *MagPIe-like*,*AGM* e *AGM-adap* para execução de 100 *Bcasts*. As falhas ocorreram para os canais (*4,6*), (*1,0*), (*12,16*) e (*8,9*), com mudanças dos valores para (*4000,0 ms*), (*10,0 ms*), (*7331,0 ms*) e (*60,0 ms*). E as recuperações foram realizadas nos canais (*6,12*), (*0,21*), (*13,14*) e (*3,9*), que passaram a valores de latências iguais a (*84,0 ms*), (*1,0 ms*), (*0,1 ms*) e (*19,0 ms*).



	Geração da árvore	Mudança latência	Execução Bcast	Adaptação da árvore	Toda Aplicação
MagPIe	42,27	120,13	471,48	0,00	582,47
AGM	25,66	120,11	397,96	0,00	537,44
AGM-adap	25,84	120,14	22,78	306,06	451,92

Figura 4.19: Tempos de execução para as três versões de algoritmo com quatro falhas e quatro recuperações intercaladas para 100 *bcasts*.

Todas as mudanças ocorreram após executar o quinquagésimo *bcast*. Observa-se que o algoritmo *AGM-adap* mostrou um melhor desempenho na execução da operação coletiva, seu tempo de execução foi, aproximadamente, 20 vezes mais rápido que o dos algoritmos *MagPIe-like* e *AGM*. A adaptação da árvore consumiu bastante tempo da aplicação, praticamente 67% de todo o tempo de execução da aplicação. Mesmo com tanto tempo gasto na adaptação, o algoritmo *AGM-adap* mostrou um tempo menor de execução em

relação aos algoritmos *MagPIe-like* e *AGM*.

Capítulo 5

Conclusões

Como contribuição, neste trabalho, propomos a construção inicial de uma biblioteca MPI para disponibilizar operações coletivas levando em consideração os valores das latências dos canais adjacentes. A topologia adotada para a disseminação da informação tem uma característica totalmente dinâmica, distinguindo-se, desta forma, dos trabalhos que mantêm estruturas estáticas para a disseminação das mensagens. Este é o caso das implementações dos algoritmos para a execução de operações *MPI_Bcast* fornecidos pelas bibliotecas *MPICH*, *MPI-LAM* e *MPICH-G2*.

Para a construção de uma topologia inicial foi preciso utilizar uma ferramenta de monitoração de rede para obtenção das latências dos canais adjacentes. Após ter todas as informações de latência de que necessita à disposição, o processo pôde dar início à construção da topologia, a ser utilizada para a disseminação da informação, através de um algoritmo distribuído.

Testes comprovaram a eficiência em se utilizar as árvores construídas e adaptadas dinamicamente, com base nas latências, para a disseminação da informação. Porém, somente a utilização da informação sobre a latência nem sempre é suficiente, principalmente se os tamanhos das mensagens tenderem a ser maiores. Em casos em que se esteja trabalhando com o envio de mensagens grandes poderia ser interessante utilizar o caminho que apresentasse maior largura de banda. Neste caso, o algoritmo distribuído para a construção da árvore irá considerar a largura de banda máxima do canal e não a sua latência.

Em todas as execuções realizadas, trabalhamos com os valores das latências momentâneas. Uma proposta interessante é utilizar de forma mais eficiente a ferramenta NWS, trabalhando com outras facilidades que ela oferece, tais como o fornecimento de previsões

sobre a disponibilidade de recursos. Assim, seria possível construir uma topologia que poderia ser útil por mais tempo, ou seja, com a probabilidade de adaptação imediata diminuída, uma vez que sua topologia tenha sido criada conforme a previsão do melhor caminho a ser utilizado em um dado espaço de tempo.

Na implementação apresentada, sempre que uma variação de latência ocorre, os canais podem ser classificados como tendo sofrido uma falha ou uma recuperação. Note que a falha ocorre se algum canal pertencente à arvore tiver sua latência incrementada, enquanto que uma recuperação é determinada se um canal fora da árvore tiver o seu tempo de latência diminuído. Um trabalho futuro seria não apenas continuar a tratar falhas e recuperações da maneira como está sendo feito atualmente, mas também fornecer mecanismos para que uma adaptação seja realizada quando uma falha de verdade ocorrer em um determinado canal. O NWS consegue identificar canais que deixaram de ser monitorados, uma investigação maior sobre este assunto possibilitaria o tratamento da ocorrências de falhas e recuperações verdadeiras sobre os canais. Este é um ponto de interesse para trabalhos futuros.

Um outro ponto a ser abordado é a investigação de técnicas que contribuam na melhoria de desempenho do algoritmo que realiza a terminação da adaptação da árvore. Seria importante minimizar o tempo gasto, por este algoritmo, quando nenhuma mudança significativa ocorre e não há necessidade de realizar a atualização da árvore. Através dos testes foram identificados que boa parte do tempo de execução do algoritmo de adaptação é gasto neste momento. Note que nestes casos, um processo localmente não possui a informação global de que a árvore não necessita de adaptação e, por isso, é preciso executar o algoritmo de terminação também, incrementando o tempo de execução da aplicação inutilmente.

Foi observado um ganho de desempenho significativo para a disseminação da informação quando utilizamos a ferramenta proposta, que considera as variações nos valores das latências nos canais. De acordo com o que foi analisado pelos testes, torna-se interessante a implementação de outras operações coletivas para a ferramenta proposta, mostrando-se um tópico de interesse, a construção de uma biblioteca de comunicação coletiva completa a ser integrada nesta ferramenta.

Referências

[den Burger et al. 2002]den Burger, M.; Kielmann, T. e Bal, H. E. **TOPOMON: A Monitoring Tool for Grid Network Topology**. In: *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, London, UKSpringer-Verlag, p. 558–567, 2002. ISBN 3-540-43593-X.

[Cheng et al. 1988] Cheng, C.; Cimet, I. e Kumar, S. **A protocol to maintain a minimum spanning tree in a dynamic topology**. *SIGCOMM Comput. Commun. Rev.*, ACM Press, New York, NY, USA, v. 18, n. 4, p. 330–337, 1988. ISSN 0146-4833.

[Gallager et al. 1983] Gallager, R. G.; Humblet, P. A. e Spira, P. M. **A Distributed Algorithm for Minimum-Weight Spanning Trees**. *ACM Trans. Program. Lang. Syst.*, ACM Press, New York, NY, USA, v. 5, n. 1, p. 66–77, 1983. ISSN 0164-0925.

[Husbands e Hoe 1998] Husbands, P. J. e Hoe, J. C. **MPI-StarT: Delivering Network Performance to Numerical Applications**. In: *SC'98, Nov*, 1998.

[Karonis et al. 2002a] Karonis, N.; Toonen, B. e Foster, I. **MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface**. *ArXiv Computer Science e-prints*, jun. 2002.

[Karonis et al. 2002b] Karonis, N. T.; de Supinski, B.; Foster, I.; Gropp, W. e Lusk, E. **A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids**. *ArXiv Computer Science e-prints*, jun. 2002.

[Karonis et al. 2000] Karonis, N. T.;de Supinski, B. R. d.; Foster, I.; Gropp, W.; Lusk, E. e Bresnahan, J. **Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance**. p. 377–386, 2000.

[Kielmann et al. 1999] Kielmann, T.; Hofman, R. F. H.; Bal, H. E.; Plaat, A. e Bhoedjang, R. A. F. **MAGPIE: MPI's collective communication operations for clustered wide area systems**. *ACM SIGPLAN Notices*, v. 34, n. 8, p. 131–140, ago. 1999.

[Lab 2004] Lab, T. L. M. O. **Lam/mpi user's guide - version 7.1.1**. 2004.

[Lacour 2001] Lacour, S. **MPICH-G2 Collective Operations: Performance Evaluation, Optimizations**. Mathematics and Computer Science Division, Argonne National Laboratory, USA, September 2001.

[Saito et al. 2005] Saito, H.; Taura, K. e Chikayama, T. **Collective Operations for Wide-Area Message Passing Systems Using Adaptive Spanning Trees**. *The 6th IEEE/ACM International Workshop*, p. 40–48, 2005.

[Thomé e Drummond 2006] Thomé, V. e Drummond, L. **Biblioteca de Comunicação Coletiva para Ambientes Distribuídos Dinâmicos**. *A ser publicado no VII Workshop em Sistemas Computacionais de Alto Desempenho*, Ouro Preto, MG, Brasil, 2006.

[Wolski 1998] Wolski, R. **Dynamically forecasting network performance using the Network Weather Service**. *Cluster Computing*, Kluwer Academic Publishers, Hingham, MA, USA, v. 1, n. 1, p. 119–132, 1998. ISSN 1386-7857.

[Wolski 2003] Wolski, R. **Experiences with predicting resource performance online in computational grid settings**. *SIGMETRICS Perform. Eval. Rev.*, ACM Press, New York, NY, USA, v. 30, n. 4, p. 41–49, 2003. ISSN 0163-5999.

[Wolski et al. 1999] Wolski, R.; Spring, N. e Hayes, J. **The network weather service: a distributed resource performance forecasting service for metacomputing**. *Future Generation Computer Systems*, v. 15, p. 757–768, 1999.

[Wolski et al. 1997] Wolski, R.; Spring, N. e Peterson, C. **Implementing a performance forecasting system for metacomputing: the Network Weather Service**. In: *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, p. 1–19, 1997. ISBN 0-89791-985-8.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo