3

Introdução a Algoritmos Distribuídos

Cláudio Geyer[1]
Alberto Egon Schaeffer Filho[2]
Fábio Reis Cecin[3]
Patrícia Kayser Vargas[4]

Resumo:

Este texto é uma introdução ao tema algoritmos distribuídos. Optou-se por uma abordagem de apresentação baseada em processos e troca de mensagens do tipo *send/receive*. Foram escolhidos alguns dos algoritmos mais citados e estudados em diversos livros sobre o tema e sobre programação distribuída. Os escolhidos são: difusão e coleta de informações em grafo, exclusão mútua, detecção de término de computação, eleição de líder, e obtenção de instantâneos.

[1] Professor do Instituto de Informática da UFRGS, doutor em Informática pela UJF (Grenoble, França), 1991. Email: geyer@inf.ufrgs.br.

[2] Mestrando em Ciência da Computação, PPGC, UFRGS, bacharel em Ciência da Computação pela UFRGS, 2002. Email: egon@inf.ufrgs.br.

[3] Mestrando em Ciência da Computação, PPGC, UFRGS, bacharel em Ciência da Computação pela UFRGS, 2001. Email: fcecin@inf.ufrgs.br.

[4] Professora do UniLaSalle-Canoas/RS, doutoranda em Ciência da Computação, COPPE/Sistemas-UFRJ, mestre em Ciência da Computação, pelo PPGC, UFRGS, 1998. Email: kayser@cos.ufrj.br.

3.1. Introdução

Este texto é uma introdução ao tema algoritmos distribuídos. O termo "algoritmos distribuídos" diz respeito a uma série de algoritmos concorrentes que podem ser projetados tanto para executar em processadores distribuídos (por exemplo, em uma rede local), quanto em multiprocessadores com memória compartilhada (ver [TAN 02] ou [LYN 96] para essas definições). Tais algoritmos são utilizados em uma ampla variedade de aplicações científicas, de controle de tempo real, de processamento distribuído de informações, de telecomunicações, de sistemas tolerantes a falhas, entre outros.

Foram escolhidos alguns dos algoritmos mais citados e estudados em diversos obras sobre o tema e sobre programação distribuída. Os escolhidos são: difusão e coleta de informações em grafo (seção 3.2), exclusão mútua (seção 3.3), detecção de término de computação (seção 3.4), eleição de líder (seção 3.5) e obtenção de instantâneos (seção 3.6).

Nesse texto optou-se pelo uso de uma abordagem de apresentação baseada em processos e troca de mensagens do tipo *send/receive*. A forma sintática usada na apresentação dos códigos é baseada na linguagem C acrescida dos comandos *send/receive* de troca de mensagens. Para mais detalhes sobre esses e outros algoritmos, sugere-se os seguintes livros: [AND 00], [BAR 96], [COU 01], [LYN 96], [TAN 02].

3.2. Difusão e coleta de informações em grafo

Em um ambiente de memória distribuída, um conjunto de processadores ou processos encontra-se interconectado de alguma forma por uma rede de comunicação. Nas próximas seções considera-se que os processos distribuídos que precisam enviar e receber informações não possuem um conhecimento global dessas interconexões. Note que este tipo de organização pode ser modelado como um grafo: os nós são os processos ou processadores envolvidos na computação e as arestas são os canais de ligação.

3.2.1. Difusão

O algoritmo de difusão tem por objetivo enviar uma informação, de um determinado nó, ou processo, a todos os outros nós de uma rede. Esta seria uma tarefa simples caso o nó emissor tivesse conhecimento do endereço (ou nome de processo) de todos os outros nós, como por exemplo se todos os nós receptores estivessem conectados diretamente ao nó emissor. Mas em muitas aplicações reais isto é raro. Freqüentemente, o nó emissor conhece somente alguns nós da rede, nomeados como vizinhos, os quais conhecem outros, formando uma topologia em grafo não totalmente conectado. Considera-se que todo nó participante do grafo tem ao menos um vizinho. Duas soluções serão apresentadas: considerando que a topologia possa ser transformada em uma *spanning tree*, e considerando o grafo com sua topologia normal ([AND 00]). Outros algoritmos relacionados, como para geração de uma *spanning tree* qualquer e para geração de uma *spanning tree* para pesquisa em largura (BFS), encontram-se em [LYN 96].

3.2.1.1. Solução via *spanning tree*

Primeiramente, faz-se necessária a definição de uma *spanning tree*, que consiste em uma árvore, construída a partir de um grafo, onde um certo nó é escolhido como raiz, e todos os nós deste grafo estão presentes. Eventuais ciclos, devido a diversos caminhos alternativos entre a raiz e um nó, são eliminados. Concretamente, isto implica a eliminação de certas arestas do grafo original. Mais formalmente, uma árvore é um grafo conexo, sem ciclos, com N nós e N-1 arestas, na qual há um e apenas um caminho entre dois nós. Uma *spanning tree* de um grafo G é um sub-grafo de G que é uma árvore que alcança (span) todos os nós.

A difusão com *spanning tree* consiste em inicialmente fazer o nó *i*, detentor da informação D, raiz desta *spanning tree*. Considera-se que esse nó *i* gerou ou obteve a *spanning tree* (ST) de alguma forma adicional ao algoritmo de difusão. Inicialmente o nó i envia D e a ST aos seus filhos na ST. Ao receberem a mensagem, os nós analisam quem são seus filhos na ST e repassam a mensagem completa (D e ST) a estes nós, e assim sucessivamente, até a mensagem ser recebida por todos os nós que não tem filhos na ST, causando o fim do algoritmo.

Observe-se que o nó *i* teria um código levemente diferente dos outros, por não receber nenhuma mensagem inicial. Para que o código-fonte de todos os nós seja simétrico, isto é, o mesmo programa sendo executado em todos os nós da rede, é adicionado um nó iniciador, que possui D e ST, e conhece o nó *i*. O nó iniciador envia ao nó *i* a primeira mensagem completa (D e ST). A Figura 3.1 exibe uma possível implementação do algoritmo em pseudo-código.

```
chan links[1..N]        /* canais globais, acessíveis a todos */

/* Código dos nós:  */
void node(int p)   {    /* p: índice (ID) do processo: 0, 1, 2, … */
   struct spanning_tree myTree;
   struct message myMsg;

   receive(links[p],myTree,myMsg); /* receive bloqueante */
   for(i=0;i<N;i++)  {
      if("i é filho de p na spanning tree")  {
         send(links[i],myTree,myMsg);
      }
   }
}

/* Código do iniciador:  */
void init(void)  {
   int i;                          /* nó i raiz */
   int top[N][N];                  /* topologia da rede */
   struct spanning_tree myTree;
   struct message myMsg;

   computeSpanningTree(i,&myTree); /* calcula ou obtem a ST */
   send(links[i],myTree,myMsg);
}
```

Figura 3.1: PseudoCódigo da difusão via spanning tree.

3.2.1.2. Solução via conjunto de vizinhos

A segunda abordagem para o problema da difusão faz uso somente do conhecimento que cada nó *p* tem de seus vizinhos no grafo original. Nenhuma *spanning tree* é calculada ou pressuposta. Suponha-se que cada nó conheça o endereço dos nós vizinhos (ou identificadores de processos) no grafo. O nó *i*, que desejar iniciar a difusão, envia a mensagem com a informação D a todos os seus vizinhos. Esses, ao receberem D, repassam-na a todos os seus vizinhos, e assim sucessivamente. Como não existe mais uma s*panning tree*, e sim um grafo, que pode ter ciclos, um nó *p* pode receber múltiplas cópias de D, que poderiam ser ignoradas. Contudo, considerando que o algoritmo de difusão está incluído em um problema maior e que os canais serão reutilizados na seqüência da execução, é necessário receber efetivamente essas mensagens, com o único objetivo de "limpar" os canais de comunicação. Um último detalhe precisa ser resolvido. Como o *receive* é bloqueante, é necessário saber o número exato de mensagens que serão enviadas a cada nó *p*. Para garantir um número fixo, ou determinístico, por nó, cada nó deve (re)enviar D também ao nó do qual inicialmente recebeu D. Cada nó enviará e receberá uma quantidade de mensagens igual ao seu número de vizinhos. Isto causará um maior número de mensagens, mas garantirá a inexistência de bloqueios. É fácil verificar que todos os nós e as arestas (canais) usados na primeira mensagem recebida por cada nó formam uma *spanning tree* dinâmica, isto é, construída durante a execução (mas não guardada).

Esta solução é apresentada em pseudo-código na Figura 3.2.

```
chan links[1..N]        /* Canais globais, acessíveis a todos */

/* Código dos nós: */
void node(int p)  {
   struct message myMsg;
   int myNeighs[N];
   int num;                    /* Número de vizinhos */

   receive(links[p],myMsg);    /* Receive bloqueante */
   for(i=0;i<N;i++)  {
      if("i é meu vizinho")  {
         send(links[i],myMsg);
      }
   }

   for(i=0;i<N;i++)  {
      if("i é meu vizinho")  {
         recvfrom(i,myMsg);
      }
   }
}

/* Código do iniciador: */
void init(void)  {
   int i;                      /* No que iniciara o algoritmo */
   struct message myMsg;

   send(links[i],myMsg)
}
```

Figura 3.2: Pseudo-código da difusão via vizinhos

3.2.1.3. Avaliação dos algoritmos

O objetivo dos algoritmos de coleta é difundir uma informação em uma rede. Esta rede normalmente é representada por um grafo, podendo este ser convertido em uma *spanning tree*, surgindo daí as variantes do algoritmo de difusão.

A abordagem de *spanning tree* gera uma mensagem para cada aresta da árvore, totalizando n-1 mensagens, onde n é o número de nós. Já a abordagem por conjunto de vizinhos gera duas mensagens para cada aresta da topologia. No caso mais simples (uma árvore com o nó iniciador como raiz), seriam 2(n-1) mensagens geradas. Contudo, no caso de um grafo totalmente conectado, seriam n(n-1) mensagens. No entanto, há de se lembrar que as mensagens da abordagem de *spanning tree* são bem maiores, uma vez que a árvore precisa ser transmitida a cada nó juntamente com a mensagem. Além disto, deve-se adicionar o custo de geração da *spanning tree*, o qual pode ser fatorado por diversas difusões sobre a mesma ST. Pressupondo que uma *spanning tree* para ser armazenada necessite de n^2 bits, o tamanho da rede deverá ser considerado na escolha do algoritmo. Por outro lado, a *spanning tree* pode ser armazenada de forma distribuída, isto é, cada nó teria a informação de quem são seus filhos na *spanning tree* ([LYN 96]), eliminando o custo de envio da ST a cada mensagem. Nesse caso, a difusão por *spanning tree* seria sempre vantajosa, desconsiderando-se o custo de geração da ST.

Vale lembrar ainda que nenhuma das duas abordagens suporta falhas (de rede, queda de nós, etc). Novos algoritmos (ou variantes dos acima), que sejam tolerantes a falhas, certamente implicam acréscimo de complexidade.

3.2.2. Coleta

Diferentemente da difusão, onde um nó (ou processo) simplesmente repassava uma informação a seus vizinhos, a coleta consiste em um algoritmo onde os nós recebem requisições de informação e têm como objetivo gerar requisições a seus filhos (se existirem), concentrar as informações recebidas de seus filhos e responder ao nó inquisidor. O primeiro nó a inquirir vai receber todas as informações, processá-las e eventualmente difundi-las a toda a rede. Uma aplicação clássica do algoritmo de coleta é a descoberta da topologia de uma rede, que será apresentada a seguir ([AND 00]).

3.2.2.1. Coleta em árvore

Seja uma rede constituída como uma árvore. Em um dado momento, o nó raiz *i* precisa descobrir a topologia da rede (ou coletar qualquer informação distribuída pela rede). Então, o nó *i* enviará uma mensagem do tipo *probe* (que conterá seu endereço ou ID) a seus filhos na árvore e ficará aguardando a mensagem de resposta (*echo*) de todos os filhos. Os filhos do nó iniciador, ao receberem a mensagem de *probe*, a reenviarão a seus filhos, aguardarão as mensagens *echo* e assim sucessivamente, até o momento que o *probe* chegue a um nó folha da árvore. Então, o nó folha, não tendo a quem repassar, enviará como *echo* a sua topologia local. O pai deste nó receberá o *echo* de todos os filhos, aglutinará a informação e a repassará a seu pai, e assim sucessivamente até o nó iniciador. Este, ao receber todos os *echo*s, os processará, podendo então processar ou difundir a informação global. No caso da informação ser a topologia da rede, o nó *i* pode usar a mesma para geração de uma *spanning tree*. No pseudo-código da Figura 3.3, é assumido que os canais são globais e que as mensagens têm rótulos que as identificam como *probe*s ou *echo*s.

```
const int source = i;         /* ID do processo iniciador */
chan probe-echo[1..N];        /* canais globais, acessíveis a todos */
chan finalEcho;         /* canal para o nó iniciador */

/* código dos nós p */
void node{int p}  {
   int links[N];               /* vizinhos deste nó */
   int localtop[N][N] = links;
   int newtop[N][N];
   int parent                  /* nó do qual o probe foi recebido */

   receive(probe-echo,parent); /* recebe um probe do nó pai */
   for(i=0;i<N;i++)  {
      if("i é vizinho e não é o pai")  {
         send(probe-echo,i);
      }
   }
   for(i=0;i<N;i++)  {
     if("i é vizinho e não é o pai")  {
        receive(probe-echo,newtop)
        localtop = localtop OR newtop;
      }
   }
   if(p == source)  {
      send(finalecho[p],newtop);   // este no eh a raiz
   }  else  {
      send(probe-echo[p],newtop);  // manda a seu pai
   }
}

/* código do processo iniciador */
void init(void)  {
   int top[N][N];

   send(probe-echo[i],source);
   receive(finalecho[i],top);

/*
      ...
      DIFUNDIR A INFORMAÇÃO COLETADA
      ...
*/
}
```

Figura 3.3: Pseudo-código da coleta via vizinhos sobre uma árvore.

3.2.2.2. Coleta em grafo

Caso a topologia da rede contenha ciclos, o algoritmo pode ser generalizado da seguinte forma: após receber um *probe*, um nó envia novos *probe*s aos seus vizinhos (exceto ao seu pai, remetente do *probe*) e aguarda um *echo* de cada um deles. Entretanto, devido aos ciclos e à execução concorrente dos nós, dois vizinhos podem vir a enviar um *probe* um ao outro, fazendo com que ocorra a recepção de um *probe* inesperado (deveriam chegar somente *echo*s). Esses *probes* adicionais podem ser respondidos com um *echo* vazio. Este *echo* vazio deve ser consistente com a operação de acumulação (zero no caso de OR ou soma) sobre as informações recebidas (em coleta).

A funcionalidade do algoritmo é garantida por uma série de fatores. Os bloqueios perpétuos (*deadlocks*) são evitados, uma vez que para cada *send* (*probe*) há um *receive* (*echo*), mesmo que vazio, e quando um nó termina sua execução é garantido que nenhum outro nó estará esperando mensagens oriundas do nó que está por terminar sua execução. Da mesma forma que na difusão, um nó iniciador pode ser considerado para tornar o algoritmo simétrico. Este nó iniciador será responsável depois por compilar as informações e difundi-las à rede. O algoritmo é apresentado na Figura 3.4.

```
const int source = i;        /* ID do processo iniciador */
chan probe-echo[1..N];       /* canais globais, acessíveis a todos */
chan finalEcho;        /* canal para o nó iniciador


/* código dos processos p */
void node{int p}  {
   int links[N];             /* vizinhos deste nó */
   int localtop[N][N] = links;
   int newtop[N][N];
   int first;                /* nó que enviou o primeiro probe */
   int k;                    /* tipo da mensagem (PROBE/ECHO) */
   int need_echo = N-1;

   receive(probe-echo[p],sender,newtop);
   first = sender;
   for(i=0;i<N;i++)  {
      if("i é vizinho")  {
         /* envia os probes */
         send(probe-echo[i],p,0);
      }
   }
   while(need_echo > 0)  {
      receive(probe-echo[p],k,sender,newtop);
      if(k == PROBE)  {
         send(probe-echo[sender],ECHO,p,0);
      }  else if(k == ECHO)  {
         localtop = localtop OR newtop;
         need_echo--;
      }
   }
   if(p == source)  {
      send(finalecho,localtop)
   }  else  {
      send(probe-echo[first],ECHO,p,localtop)
   }
}


/* código do processo iniciador */
void init(void)  {
   int top[N][N];

   send(probe-echo[i],PROBE,source,0);
   receive(finalecho,top);
/*
            ...
      PROCESSAR OU DIFUNDIR A INFORMAÇÃO COLETADA
      ...
*/
```

```
}
```

Figura 3.4: Pseudo-código da coleta via vizinhos sobre um grafo.

3.2.2.3. Avaliação dos algoritmos

O algoritmo de coleta com *probe*/*echo* requer um número menor de mensagens que o algoritmo Heartbeat (algoritmo normalmente usado em aplicações científicas paralales - [AND 00]) na aplicação de descoberta da topologia de uma rede, uma vez que, na coleta, duas mensagens são enviadas em cada link que liga os nós da *spanning tree* (que não tem links redundantes). Como a difusão final da topologia requer mais n mensagens, o algoritmo em árvore requer um total de $2n + n = 3n$ mensagens. Na topologia em grafo, somam-se mais quatro mensagens por link adicional (redundante) do grafo, totalizando $2n + n + 4l = 3n + 4l$, onde l é o número de links adicionais do grafo (links que seriam eliminados na conversão deste em uma *spanning tree*).

3.3. Exclusão mútua

O problema de exclusão mútua é um problema clássico em programação concorrente. A maioria dos livros sobre sistemas operacionais e programação concorrente apresentam soluções para esse problema ([TAN 02], [SIL 02], [AND 00]). Soluções para programação distribuída aparecem em [AND 00], [TAN01], [TAN 02], [COU 01], [LYN 96]. Existem diversas outras soluções (ou variações) em artigos científicos.

O problema da exclusão mútua, na sua forma elementar, pode ser expresso da seguinte forma:

- Dois ou mais processos devem executar um código que faz acesso a uma ou mais variáveis comuns;
- Ao menos um processo faz acesso em escrita sobre uma dessas variáveis;
- As operações de escrita e leitura sobre essas variáveis não são atômicas, isto é, durante a escrita, o valor da variável pode estar em um estado intermediário inconsistente.

Existem variações da forma acima. Por exemplo, pode ser definido que uma combinação de leituras e escritas sobre uma variável, por um único processo, deve ser uma operação atômica.

O conceito de exclusão mútua está diretamente relacionado ao de seção crítica, isto é, uma seção de código (ou uma coleção de operações) na qual apenas um processo pode executar de cada vez.

A solução elementar do problema da exclusão mútua implica o uso de dois protocolos:

- De entrada: executado sempre que um processo quer executar o código da seção crítica (“entrar” na seção crítica); esse protocolo deve fazer o processo esperar, caso outro processo já esteja executando a seção crítica;
- De saída: executado sempre que um processo termina de executar o código da seção crítica (“saída” da seção crítica); esse protocolo deve liberar a seção crítica para um outro processo que esteja eventualmente esperando por aquela.

O protocolo de entrada é freqüentemente chamado de *mutexbegin* ou *lock*, enquanto o de saída de *mutexend* ou *unlock*. Uma solução para exclusão mútua, tanto para memória compartilhada quanto para memória distribuída, deve apresentar uma série de propriedades. Entre elas, as principais são:

- Obviamente a de exclusão mútua é obrigatória, isto é, se um processo está ocupando a seção crítica, qualquer outro processo que queira entrar deve ficar bloqueado no *mutexbegin*;
- Não deve haver postergação indefinida (*starvation*), isto é, deve-se evitar que um processo em espera no *mutexbegin* ali permaneça indefinidamente; especificamente, se dois ou mais processos estão esperando a sua vez para ocupar a seção crítica, quando a mesma for liberada, a escolha de qual deve recebê-la deve ser uma escolha justa; o critério mais apropriado é, em geral, o de uma fila FIFO: escolhe-se o processo que há mais tempo está esperando pela seção crítica;
- A solução deve ser aplicável a qualquer quantidade de processos e independer da velocidade relativa desses processos;
- Deve-se evitar situações, mesmo que com baixa probabilidade de ocorrência, de bloqueio perpétuo (*deadlock*) entre dois ou mais processos na execução do *mutexbegin*; em princípio, a ocorrência de *deadlocks* seria simplesmente um erro de projeto da solução.

O problema da exclusão mútua é mais complexo de ser tratado em um sistema distribuído do que em um sistema que executa em um único computador. Em um sistema distribuído, não há uma memória física compartilhada, sendo necessário realizar troca de mensagens entre os processos para a realização dos protocolos de entrada e saída da seção crítica. Uma das soluções mais comumente empregadas e simples de implementar para sistemas distribuídos é a do *token* em anel. Uma segunda solução é a proposta por Ricart e Agrawala ([RIC 81], [TAN 02], [LYN 96]), a qual é baseada no conceito de *relógio lógico* proposto por Lamport ([LAM 78]).

3.3.1 Algoritmo de exclusão mútua em anel (*token*)

Dado um conjunto de *n* processos potencialmente interessados na seção crítica em questão (P_1, P_2, ... P_n), a idéia central do algoritmo de exclusão mútua em anel é organizar os processos logicamente (isto é, sem relação com a topologia "física" de rede utilizada) em um anel e fazer circular uma mensagem especial (o *token*) neste anel (Figura 3.5). A configuração do anel é realizada antes do início da execução do algoritmo, fornecendo-se para cada processo o endereço do próximo processo na seqüência do anel.

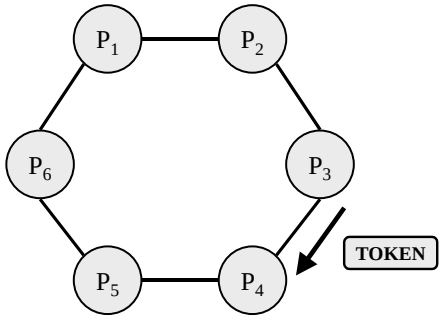

Figura 3.5: Um anel lógico com 6 processos (n = 6) passando um *token*.

Se um processo recebe o *token*, mas não está interessado em executar a seção crítica, repassa imediatamente o mesmo para o próximo processo. Um processo que quer executar a seção crítica deve esperar o recebimento do *token*. Após receber o *token*, o processo executa a seção crítica, e depois de encerrada a execução da mesma, deve repassar o *token* para o próximo processo. A Figura 3.6 exibe uma possível implementação do algoritmo.

É trivial provar que o algoritmo garante a exclusão mútua entre os processos, bastando observar que, em um determinado momento da execução, no máximo um processo possui o *token*. Também é trivial provar que não ocorre postergação indefinida na espera da seção crítica. Se a execução da seção crítica por um processo é concluída em tempo finito, o *token* será recebido repetidamente em todos os processos. *Deadlocks* não ocorrerão à medida que a implementação garantir que um processo não extraviará o *token*, e que o programador da aplicação não esquecerá de garantir a demarcação correta da seção crítica com invocações de *mutexbegin* e *mutexend* aos pares.

Com o emprego deste algoritmo, o número de mensagens necessário para se entrar na seção crítica varia entre 1 e (n-1). Porém, há circulação permanente de mensagens mesmo quando não há nenhum processo interessado na seção crítica. Adicionalmente, a ordem de entrada na seção crítica não ocorre necessariamente de uma forma justa. Utilizando o exemplo da Figura 3.5, considere a seguinte ordem de eventos:

- *Token* recebido por P5;
- P4 torna-se interessado em entrar na seção crítica;
- P3 torna-se interessado em entrar na seção crítica;
- *Token* faz a volta no anel e é recebido por P3.

Neste cenário, apesar de P4 se ter interessado no *token* antes de P3, será P3 o primeiro a entrar na seção crítica.

O algoritmo não é tolerante a falhas. Se um dos processos falhar, uma solução possível é remover o mesmo do anel, de forma que o seu predecessor conecte-se com o seu sucessor. Se o processo que falhou possuía o *token*, então é também necessário escolher um dos processos restantes para colocar um novo *token* em circulação no anel. Este problema de escolha é resolvido por algoritmos de eleição, que serão apresentados na Seção 3.5.

```
/* Variáveis */
int       n;     /* Ordem deste processo no anel: 1, 2, 3 ... n */
mensagem  token; /* Variável para armazenar a mensagem de token */
canal     pred;  /* Canal de comunicação com o predecessor no anel */
canal     succ;  /* Canal de comunicação com o sucessor no anel */
int       interesse; /* 1 se interessado no token; 0 caso contrário */
condicao  cvtoken;   /* Variável de condição para espera do token */

/* Inicialização */
void main() {
  n = ...; pred = ...; succ = ...; /* Inicializa anel */
  if (n == 1) {
    token = ...;        /* Primeiro processo cria o token */
    send(succ, token);  /* Primeiro processo injeta o token no anel */
  }
  token = null;
  interesse = 0;
}

/* Evento: Token disponibilizado pelo predecessor no anel */
void evTokenDisponivel() {
```

```
  token = receive(pred);
  if (interesse == 0) {
    send(succ, token); /* envia token para o sucessor imediatamente */
    token = null;
  } else
    signal(cvtoken);   /* retém o token e avisa a aplicação */
}

/* Protocolo de Entrada */
void mutexbegin() {
  interesse = 1;  /* Registra interesse para que o token não
                     seja repassado imediatamente ao chegar */
  /* Espera o token chegar */
  while (token == null) {
     wait(cvtoken);
  }
  interesse = 0;
}

/* Protocolo de Saída */
void mutexend() {
  send(succ, token);
  token = null;
}
```

Figura 3.6: Implementação do algoritmo de exclusão mútua com *token* em anel.

3.3.2 Relógio lógico

Para entender o algoritmo de exclusão mútua em grafo completo (que será apresentado na seção seguinte), torna-se necessário introduzir o conceito de *relógio lógico* ([LAM 78]).

Para fins de ordenamento de eventos gerados em uma única máquina, um relógio físico é suficiente: basta obter o valor atual do relógio da máquina e associá-lo ao evento. Posteriormente, para tomada de decisões baseado na ordem destes eventos, por exemplo, para decidir qual evento ocorreu primeiro, basta comparar os tempos associados (*timestamps*) a cada evento.

Em um ambiente distribuído, porém, não há um único relógio para consulta por todos os processos. Apesar da existência de protocolos para sincronização de relógios físicos, como o NTP (*Network Time Protocol*), geralmente não há um método disponível para garantir a sincronização constante dos relógios em nível de instrução de máquina ([COU 01]). Isto significa que não há como comparar os *timestamps* de dois eventos gerados em duas máquinas distintas, que utilizaram dois relógios físicos distintos para observar o tempo de ocorrência dos seus eventos.

Uma alternativa é a utilização de um *relógio lógico* para o ordenamento de eventos em um ambiente distribuído. Ao invés de associar valores retirados de relógios físicos aos eventos, serão associados valores obtidos de variáveis do tipo contador (o relógio lógico). O conceito central do relógio lógico é o de que certos eventos gerados localmente em um processo causarão o envio de uma mensagem para outro processo, e que o evento de recebimento desta mensagem ocorre depois do envio. Em outras palavras, o *timestamp* do evento de recebimento da mensagem será sempre maior do que o *timestamp* do evento que causou o envio da mensagem. O algoritmo de atualização do relógio lógico (RL) é o seguinte:

- Inicialmente, **RL** = 0 (opcional);

- Quando um evento é gerado localmente, **RL** = **RL** + 1, e o *timestamp* do evento é igual ao valor de **RL** após o incremento;
- Se o evento gerado localmente causa o envio de uma mensagem, esta mensagem leva o evento e o *timestamp* local associado ao evento;
- Quando um processo recebe uma mensagem **m** e seu *timestamp* **t**:
    1) **RL** = maior valor entre **RL** e **t**;
    2) **RL** = **RL** + 1;
    3) O evento de recebimento de **m** é gerado, com *timestamp* = **RL**.

A Figura 3.7 ilustra três processos gerando eventos ao longo do tempo. Os pontos denotam eventos, os *timestamps* dos eventos são indicados entre parênteses e as setas indicam envio e recebimento de mensagens. A primeira observação é a de que o evento de envio de uma mensagem sempre precede o evento correspondente de recebimento. Porém, observa-se que é também possível determinar a ordem de ocorrência de eventos não diretamente envolvidos em uma troca de mensagens. Por exemplo, o evento A ocorre antes do evento E. Isto é garantido pois A precede B (observado localmente pelo relógio físico de P1), B precede C (pela relação de envio e recebimento de mensagem), e assim sucessivamente.

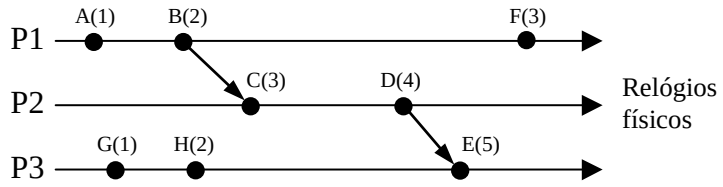

Figura 3.7: Eventos (A, B, C, ...) ocorrendo em três processos (P1, P2 e P3), e respectivos timestamps associados (1, 2, 3, ...) obtidos por relógio lógico.

Por outro lado, observa-se que o relógio lógico oferece um ordenamento parcial dos eventos do sistema: os eventos A e G possuem o mesmo *timestamp* (1), e o mesmo ocorre com os eventos C e F (3). Porém, é possível estender o algoritmo para se obter uma ordem total, bastando estender o *timestamp* dos eventos para incluir, por exemplo, o identificador do processo que gerou o evento. Em caso de empate nos valores obtidos do relógio lógico, e partindo-se do princípio que os identificadores dos processos são únicos, um desempate é feito comparando-se os identificadores. Portanto, na Figura 3.7, considerando que os eventos em P3 ocorrem após os de P2, e os de P2 após os de P1, A precede G, e F precede C.

De qualquer maneira, é importante observar que os *timestamps* gerados pelo relógio lógico apenas oferecem um critério consistente e uniforme para ordenamento dos eventos em um ambiente distribuído. Por um lado, pode-se garantir que B precede C em relação à passagem "real" do tempo, pois é n ecessário que uma mensagem seja enviada por um meio físico antes que ela seja recebida. Por outro lado, apesar do critério de ordenamento do relógio lógico afirmar que A precede H, nada pode ser afirmado em relação ao ordenamento destes eventos na escala de tempo real.

3.3.3 Algoritmo de exclusão mútua em grafo completo

Ao contrário do algoritmo em anel apresentado na Seção 3.3.1, o algoritmo apresentado nesta seção baseia-se em um grafo completo de vizinhança entre os processos, ou seja, todo processo conhece o endereço de todos os outros processos. O

algoritmo em grafo também é baseado na posse de um *token* associado à seção crítica, ou seja, apenas o processo que possui o *token* pode estar dentro da seção crítica. Porém, a transferência do *token* de um processo para outro requer a transferência de múltiplas mensagens.

Existem dois tipos de mensagens: a mensagem de requisição de *token* ("REQUEST"), e a mensagem de resposta à requisição ("REPLY"). Cada processo mantém um relógio lógico que, juntamente com o identificador dos processos, é utilizado para fornecer um *timestamp* com garantia de ordem total às mensagens do tipo REQUEST. Um processo pode estar em três estados distintos: sem o *token* e sem interesse no mesmo ("RELEASED"), sem o *token* e com interesse no mesmo ("WANTED") e com o *token* ("HELD").

A Figura 3.8 ilustra uma implementação possível do algoritmo. Inicialmente, todos os processos se encontram no estado RELEASED, isto é, nenhum está interessado na seção crítica e nenhum possui o direito de acessá-la. Quando um processo estiver interessado em obter o *token* da seção crítica (rotina *mutexbegin*), este modifica seu estado para WANTED e envia uma mensagem REQUEST (com o devido *timestamp* obtido a partir do relógio lógico) para todos os outros processos. Após o envio dos REQUESTs, o processo espera a obtenção de mensagem REPLY de todos os outros processos, e só procede para a seção crítica quando obter o REPLY correspondente ao REQUEST enviado a cada processo.

```
Int n, maxn;  /* n: n. do processo  max: n. total de processos */
canal proc[]; /* Canais de comunicação com os outros processos */
enum { REQUEST, REPLY }; /* Tipos de mensagens */
int state;    /* Estado atual do processo */
enum { RELEASED, WANTED, HELD }; /* Lista de estados possíveis */
int T;  /* timestamp do último pedido de token gerado localmente */
fila reqs;    /* Fila de nro. de processos que ainda não foram
                 respondidos */

/* Inicialização */
void main() {
  n = ...; maxn = ...; proc = ...; /* Inicializa o grafo */
  state = RELEASED; /* Inicialmente nenhum processo possui o token */
}

/* Evento: mensagem do tipo REQUEST disponibilizada pelo processo p */
void evRequestDisponivel(int p) {
  mensagem m = receive(proc[p], REQUEST);
  if ((state == HELD) || ((state == WANTED) && (T < m.T))
    fila.put(p); /* Registra requisição de p na fila e não responde */
  else {
    mensagem r = <REPLY>; /* Constrói mensagem REPLY */
    send(proc[p], r);     /* Envia REPLY para p imediatamente */
  }
}

/* Protocolo de Entrada */
void mutexbegin() {
  state = WANTED;
  T = ...;   /* Obtém timestamp para o evento de pedido de token */
  Mensagem m = <REQUEST, T>;   /* Pedido do processo n no tempo T */
  for (int i=0; i<maxn; i++)   /* Envia m para os outros processos */
    send(proc[i], m);

  /* O TOKEN só será possuído pelo processo quando este receber um
      REPLY de cada outro processo */
  for (int i=0; i<maxn; i++)   /* Coleta REPLY dos outros processos */
    m = receive(proc[i], REPLY);
  state = HELD;
}

/* Protocolo de Saída */
void mutexend() {
  /* Libera o TOKEN localmente */
  state = RELEASED;
  /* Envia REPLY para REQUESTs ainda não respondidos (enfileirados) */
  mensagem r = <REPLY>;
  for (int i=0; i<fila.size(); i++) {
    int p = fila.get(i); /* nro. de processo que não recebeu REPLY */
    send(proc[p], r);    /* Envia REPLY para este processo */
  }
}
```

Figura 3.8. Implementação do algoritmo de exclusão mútua em grafo completo.

Quando um processo *n* recebe um REQUEST de outro processo *p* (*evRequestDisponível*), duas ações são possíveis: (1) *n* pode enviar um REPLY imediatamente para *p* ou (2) ou pode registrar a requisição de *p* em uma fila e enviar o

REPLY mais tarde. A ação (2) é realizada quando *n* está em estado HELD, ou quando *n* está em estado WANTED e a requisição de *n* precede a requisição de *p* (de acordo com a comparação dos *timestamps* dos eventos). Isto significa que o REPLY não será enviado imediatamente para *p* se o processo *n* está dentro da seção crítica, ou se *n* também quer entrar na seção crítica e "pediu antes". A ação (1) executa no caso contrário, ou seja, se *n* está em estado RELEASED (sem interesse no *token*), ou se *n* está em estado WANTED, mas o *timestamp* do seu pedido é maior do que o *timestamp* de *p* (ou seja, "pediu depois"). Este mecanismo garante a resoluçã o de disputas no caso de requisições concorrentes ao *token*.

Para um melhor entendimento do algoritmo, considere-se uma rede com três processos (P1, P2 e P3) disputando uma seção crítica. A Figura 3.9 ilustra uma seqüência possível de eventos, no caso em que P1 e P2 requisitam o *token* de forma concorrente.

1. P1 e P2 chamam *mutexbegin* e se colocam em estado WANTED;
2. P1 envia REQUEST para P2 e P3 com T = 10;
3. P2 envia REQUEST para P1 e P3 com T = 20;
4. P3 recebe requisição de P1 e envia REPLY imediatamente;
5. P3 recebe requisição de P2 e envia REPLY imediatamente;
6. P1 recebe REQUEST de P2, mas enfileira o pedido, pois o estado de P1 é WANTED e T(10) < m.T (20);
7. P2 recebe REQUEST de P1 e envia REPLY imediatamente, pois o estado de P2 é WANTED, mas o teste T(20) < m.T(10) resulta em *falso*;
8. P1 recebe o REPLY de P2 e P3 na rotina *mutexbegin* e obtém o *token*;
9. P1 termina de executar a seção crítica e invoca *mutexend*. Ao percorrer a fila de REQUESTs não respondidos, envia o REPLY que faltava, para P2;
10. P2 recebe o REPLY de P1 e P3 na rotina *mutexbegin* e obtém o *token*...

Figura 3.9. Possível seqüência de eventos com três processos (P1, P2 e P3).

Observa-se que todo processo que fizer a requisição certamente terá acesso ao *token*, pois toda mensagem REQUEST certamente será respondida com um REPLY, mesmo que isto não ocorra imediatamente. Neste contexto, mostra-se que a obtenção do *token* necessita do envio de 2(n – 1) mensagens, sendo (n – 1) mensagens REQUEST e (n – 1) mensagens REPLY correspondentes. Finalmente, apesar do algoritmo ser distribuído, é importante observar que o mesmo não pode prosseguir corretamente sua execução em caso de falha de algum processo. Por exemplo, se o processo que possui o *token* falhar, este será perdido e nenhum outro processo poderá entrar na seção crítica.

3.4. Detecção de término

Ao contrário do que acontece em um programa seqüencial ou em um programa concorrente sobre um único processador, determinar se um programa distribuído encerrou seu processamento não é uma tarefa trivial. Isso se deve principalmente a dois fatores: (1) o estado global da computação não é visível a nenhum dos processadores e, (2) mesmo que todos os processadores estivessem inativos, ainda poderia haver mensagens em trânsito no meio de comunicação.

Dessa forma, um algoritmo para detecção de término de um programa distribuído deve obter informações referentes a um estado global que leve em conta tanto o estado relativo de cada processo quanto também dos canais de comunicação que ligam tais processos. Se todos os processos estiverem em estado inativo e nenhuma mensagem estiver em trânsito nos canais de comunicação, diz-se que o algoritmo distribuído está terminado ([RAY 88]).

A seguir, serão apresentados dois algoritmos para a detecção de término em sistemas distribuídos: a primeira solução utiliza uma estratégia de passagem de *token* entre os processos supondo que a comunicação entre eles é realizada por meio de um anel; a segunda solução é uma generalização onde a comunicação se dá através de um grafo direcionado completo. Em [LYN 96], é apresentado um algoritmo para detecção de término especificamente em algoritmos de difusão através da construção e manutenção de uma *spanning tree* a partir do grafo de nós participantes. Além disso, [RAY 88] apresenta variações dos algoritmos de detecção de término discutidos nessa seção.

3.4.1. Detecção de término em um anel

Como já mencionado, primeiramente será apresentada uma solução para o problema de detecção de término em sistemas distribuídos supondo que a comunicação entre os processos é realizada por meio de um anel unidirecional ([AND 00]).

Consideremos P[1:n] o vetor de processos que compõem o programa distribuído e C[1:n] um conjunto de canais de comunicação entre esses processos. De acordo com a premissa de que a comunicação é realizada em um anel, podemos assumir que cada processo P[i] recebe mensagens unicamente através do seu canal C[i] e utiliza o canal seguinte para enviar mensagens para seu sucessor. Além disso, as mensagens são recebidas em ordem FIFO por cada um dos processos do anel.

Para realizar a detecção de término, o algoritmo propõe a utilização de um *token*, o qual não faz parte da computação propriamente dita, mas é passado entre os processos através dos mesmos canais de comunicação que formam o anel (Figura 3.10).

Os processos podem assumir dois estados possíveis: ativo ou inativo. Processos ativos são representados pela cor vermelha, ao passo que processos inativos são representados pela cor azul. Um processo é considerado ativo caso esteja realizando alguma computação. Por outro lado, um processo está em estado inativo caso tenha terminado sua computação ou esteja apenas aguardando por novas mensagens. O estado inicial de cada um dos processos é ativo. Sempre que um processo recebe o *token* de seu vizinho, ele torna-se inativo (quando também recebe a coloração azul) e então passa o *token* para o processo seguinte. Se após estar inativo o processo receber uma nova mensagem regular, sua cor é alterada novamente para vermelho.

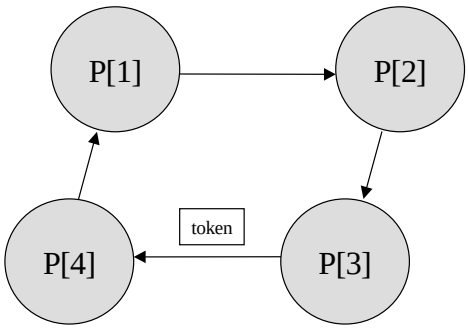

Figura 3.10: Processos organizados em um anel.

Ao receber um *token*, o processo sabe que seu vizinho anterior no anel está inativo e, ao enviar esse *token* para o vizinho seguinte, o próprio processo torna-se inativo (pelo menos até que uma nova mensagem regular de computação seja entregue, situação que fará com que o processo passe do estado inativo para o estado ativo).

Ao fazer o *token* dar uma volta completa no anel de comunicação, sabemos apenas que todos os processos estiveram inativos em algum momento, mas não se a computação completa realmente terminou. Para garantir que a computação terminou é necessário que o processo que iniciou o algoritmo de detecção de término, ao receber novamente o *token*, tenha permanecido inativo (azul) desde o momento em que o *token* foi passado pela primeira vez. Supondo que o processo P[1] tenha iniciado o algoritmo de detecção de término, a computação é considerada terminada se, após o *token* ter chegado de volta a P[1], este tenha permanecido continuamente em estado inativo desde o momento em que o *token* foi passado pela primeira vez. Dessa forma, o *token* tem a função de empurrar todas as mensagens regulares que estiverem a sua frente. Quando o *token* chegar novamente a P[1], é garantido que não existem mensagens regulares enfileiradas ou em trânsito devido às considerações iniciais de que o *token* circula nos mesmos canais de comunicação das mensagens regulares e que tais canais mantêm a ordenação das mensagens que neles trafegam.

O algoritmo associa um valor ao *token* indicando quantos canais estão vazios se o processo iniciador do algoritmo (nesse caso podemos considerar como sendo o processo P[1]) ainda estiver inativo. No momento em que P[1] tornar-se inativo, ele recebe a coloração azul, associa o valor zero ao *token* e o envia para o processo P[2]. A partir desse ponto, cada processo P[i] que receber o *token* será colorido de azul, incrementará o *token* e o passará para seu vizinho seguinte. Se, ao receber o *token* novamente, o processo P[1] ainda estiver colorido de azul, isso significa que nenhuma mensagem regular foi gerada, de forma que a computação pode ser considerada terminada. Resta apenas ao processo P[1] anunciar aos demais processos essa situação. O algoritmo para a passagem do *token* é listado na Figura 3.11.

O algoritmo garante que, se o processo P[1] estiver inativo (azul), ele não enviou mensagens após ter enviado o *token* e, portanto, não existem mensagens regulares nos canais até o ponto do anel onde o *token* está localizado, assim como todos os processos até esse ponto permanecem em estado inativo. Dessa forma, se P[1] estiver inativo quando o *token* voltar a ele, será possível afirmar que todos os canais estarão vazios e todos os processos que formam o anel estarão inativos.

```
/* Variáveis */
string cor[];           /* Array de cores para marcar os processos */
canal C[];              /* Canais de comunicação */
mensagem token;         /* Variável para armazenamento do token */
mensagem msgRegular;    /* Variável para mensagem regular */
int i;                  /* Índice do processo */
int n;                  /* Número de processos */

/* Detecção de inatividade feita por P[1] */
void detectouInatividade() {
    if (i == 1) {
        cor[1] = "azul";
        token = 0;
        send(C[2], token);
    }
}
```

```
/* Recebimento de mensagem regular */
void evMsgRegularDisponivel() {
    msgRegular = receive(C[i%n - 1]);
    cor[i] = "vermelho";
}

/* Recebimento do token */
void evTokenDisponivel() {
    token = receive(C[i%n - 1]);
    if (i == 1) {
        if (cor[1] == "azul") {
            anunciarTermino();
            exit();
        } else {
            cor[1] = "azul";
            token = 0;
            send(C[2], token);
        }
    } else {
        cor[i] = "azul";
        token++;
        send(C[i%n + 1], token);
    }
}
```

Figura 3.11: Implementação do algoritmo para detecção de término em anel.

Quanto a sua complexidade, a detecção de término em anel exige que N mensagens sejam enviadas ao redor do anel para que o processo que enviou o *token* pela primeira vez possa chegar à conclusão de que a computação realmente terminou. Além dessas, outras N mensagens serão enviadas para que todos os outros processos tenham conhecimento dessa situação, totalizando assim 2N mensagens geradas pelo algoritmo de detecção de término.

3.4.2. Detecção de término em um grafo

O algoritmo de detecção de término em um grafo ([AND 00]) assume uma estrutura de comunicação na forma de um grafo direcionado completo, onde os nós correspondem a processos e as arestas representam caminhos de comunicação. Um grafo é dito completo se em cada nó existir uma aresta para todos os outros nós. O algoritmo considera a existência de um vetor de processos P[1:n] e um vetor de canais de comunicação C[1:n]. Nesse contexto, cada processo P[i] recebe mensagens através de seu canal privado de entrada C[i], porém qualquer processo pode enviar mensagens para o canal C[i].

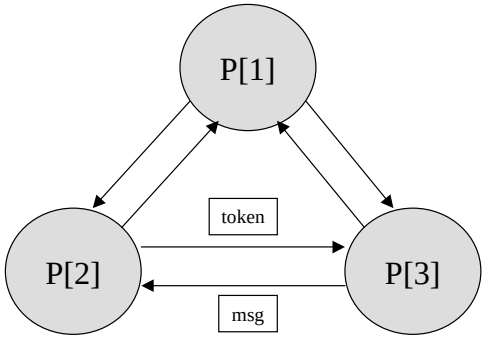

Figura 3.12: Processos em um grafo direcionado completo.

Como já mencionado, a computação é considerada encerrada quando todos os processos estão inativos e não existem mensagens nos canais de comunicação. Da mesma forma que o algoritmo da seção anterior, um *token* é utilizado para empurrar as mensagens em trânsito entre os nós. Para isso, é necessário assegurar que o *token* percorra todas as arestas, visitando todos os nós múltiplas vezes. Como as mensagens podem chegar por qualquer um dos canais, a detecção de término torna-se mais difícil que aquela realizada em um anel. Tomando o grafo da Figura 3.12 como exemplo, podemos verificar que simplesmente fazer com que o *token* circule entre os processos P[1], P[2], P[3], e novamente para P[1] não irá garantir que ao chegar em P[1] a computação tenha terminado, mesmo que o processo P[1] tenha permanecido continuamente inativo. Isso se deve ao fato de que enquanto o *token* está circulando em um sentido, exatamente ao mesmo tempo uma mensagem regular pode estar circulando em sentido oposto (por exemplo, *token* sendo enviado de P[2] para P[3] e mensagem regular sendo enviada de P[3] para P[2]).

As cores vermelho e azul são utilizadas para representar os estados ativo e inativo (respectivamente) que um processo pode assumir. Ao receber uma mensagem regular, o processo torna-se ativo. Ao receber um *token*, o processo torna-se inativo e passa o *token* adiante. Se enquanto o processo estiver inativo (azul), este receber uma mensagem regular, sua coloração será alterada novamente para vermelho.

O algoritmo descrito nessa seção baseia-se no fato de que o grafo formado pelos processos contém um ciclo C (assim como qualquer outro grafo direcionado completo), de comprimento nc, que inclui todas as arestas desse grafo. Cada processo deve observar a ordem em que suas arestas de saída aparecem em C, de forma que ao receber o *token*, o processo possa enviá-lo para seu sucessor através da próxima aresta em C.

```
/* Variáveis */
string cor[];           /* Array de cores para marcar os processos */
canal C[];              /* Canais de comunicação */
mensagem token;         /* Variável para armazenamento do token */
mensagem msgRegular;    /* Variável para mensagem regular */
int i;                  /* Índice do processo */
int n;                  /* Número de processos */
int nc;                 /* Número total de arestas do grafo */
int succ;               /* Índice do processo sucessor no ciclo C */

/* Recebimento de mensagem regular */
void evMsgRegularDisponivel(int j) {
    msgRegular = receive(C[j]);
    cor[i] = "vermelho";
}

/* Recebimento do token */
void evTokenDisponivel(int j) {
    token = receive(C[j]);
    if (token == nc) {
        anunciarTermino();
        exit();
    } else if (cor[i] == "vermelho") {
        cor[i] = "azul";
        token = 0;
        send(C[succ], token);
    } else {
```

```
        token++;
        send(C[succ], token);
    }
}
```

Figura 3.13: Implementação do algoritmo para detecção de término em um grafo completo.

O valor carregado junto ao *token* expressa o número de vezes que este passou por processos inativos, sendo que seu valor inicial é zero. Quando qualquer processo tornar-se inativo, este assumirá a coloração azul e enviará o *token* através da primeira aresta contida em C. Ao receber um *token*, um processo deverá realizar os procedimentos descritos na Figura 3.13. Cada vez que um processo vermelho receber o *token*, ele deverá colorir-se de azul, reiniciar o contador do *token* e passá-lo adiante. Dessa forma, somente após o *token* ter visitado todos os nós pelo menos uma vez ele passará a ser efetivamente incrementado. Ao atingir o valor nc, saberemos que o *token* passou por todos os canais do ciclo após todos os processos estarem inativos, pois somente a partir daí o *token* deixou de ser reiniciado e é essa a condição que garante que todos os canais do grafo estão vazios. Nesse momento, portanto, a computação pode ser considerada terminada e o processo que verificar essa condição deverá avisar aos demais processos que a terminação da computação foi detectada.

Em relação à complexidade, o algoritmo exige que, caso a computação tenha realmente terminado, N mensagens sejam enviadas até que todos os processos fiquem coloridos de azul. A partir daí, outras nc mensagens devem ser propagadas por todos os canais do ciclo C do grafo de comunicação. Por último, é necessário que todos os N processos tomem conhecimento da detecção de término da computação, gerando-se assim pelo menos outras N mensagens. O total de mensagens geradas pelo algoritmo é, portanto, 2N + nc, caso o processo que suspeitou do término da computação esteja certo.

3.5. Eleição de líder

Diversos algoritmos distribuídos exigem a existência de um processo central que desempenhe o papel de coordenador ou realize alguma função especial durante a computação. Nessas ocasiões, é essencial a utilização de algoritmos de eleição de líder para determinar qual processo assumirá a função de coordenador, sendo que muitas vezes não importa qual processo seja o novo líder, desde que alguém seja eleito. Além disso, é necessário que todos os processos concordem com a escolha do novo líder e também que eleições subseqüentes possam ser realizadas para eleger novos substitutos no futuro.

Considerando um arranjo com N processos, até N eleições podem ser realizadas de forma concorrente, sendo que cada processo pode iniciar no máximo uma eleição por vez. Independente disso, um requisito importante é de que somente um líder seja eleito. A qualquer momento, um determinado processo pode ser considerado um participante ou um não-participante: no primeiro caso, significa que o processo está participando de uma execução do algoritmo de eleição, enquanto que no segundo caso o processo em questão não está participando da eleição.

Uma alternativa para guiar a escolha do novo líder é selecionar o processo que possuir o maior identificador dentre todos os processos que não estejam em estado de

falha (porém, existem algoritmos que utilizam outras abordagens, como por exemplo selecionar o processo com o menor identificador). Isso exige que os identificadores sejam únicos, pois do contrário não seria possível determinar a identidade do novo líder. O identificador de um processo pode ser uma combinação de dois ou mais atributos como, por exemplo, o endereço IP da máquina do processo e o PID (*process identificator*) desse processo.

Serão apresentados dois algoritmos para eleição de líder: o primeiro deles realiza o processo de eleição supondo que a comunicação entre os processos é realizada ao redor de um anel; o segundo não exige a disposição em anel, mas exige que cada processo conheça informações sobre os identificadores de outros processos. Além dos algoritmos para eleição de líder discutidos nessa seção, [LYN 96] apresenta uma série de outros algoritmos (tanto para processos organizados em anel, quanto também organizados em topologias arbitrárias) associados a detalhadas análises de suas complexidades relativas ao tempo e ao números de mensagens trocadas entre os processos.

3.5.1. Eleição de líder em anel

Essa solução para eleição de líder em sistemas distribuídos ([COU 01]) exige que os processos estejam organizados em um anel lógico, onde cada processo sabe quem é seu sucessor. As mensagens são enviadas no sentido horário através de um canal de comunicação que liga cada processo ao seu vizinho. O vetor de processos é representado por P[1:n], e C[1:n] corresponde ao conjunto de canais de comunicação entre esses processos. Além disso, o algoritmo parte do princípio que os processos não estão sujeitos a falhas durante a execução do algoritmo de eleição, e que o sistema é assíncrono (existem variações do algoritmo em [TAN 02] que permitem seu funcionamento mesmo com a ocorrência de falhas nos processos participantes do anel, mas nesse caso é exigido que os processos consigam se comunicar não somente com seu vizinho, mas também com os demais sucessores no anel).

Inicialmente, cada um dos processos é marcado como não-participante. Quando um processo identificar a ausência de um líder, ele inicia o processo de eleição primeiramente marcando-se como participante e em seguida enviando uma mensagem de eleição contendo seu identificador para seu vizinho no anel.

Ao receber uma mensagem de eleição, um processo deve comparar o identificador contido na mensagem com o seu próprio. Se o identificador recebido for maior, o processo deve passar adiante a mensagem para seu vizinho. Se por outro lado, o identificador contido na mensagem for menor que o identificador do processo em questão e o mesmo ainda não for um participante, então este deverá substituir o conteúdo da mensagem pelo seu próprio identificador antes de enviá-la para seu vizinho. Em ambos os casos, o receptor torna-se um participante. Porém, caso o identificador do receptor seja maior que o contido na mensagem e este já for um participante, nada deve ser feito, uma vez que seu identificador já está circulando no anel.

Caso o identificador contido na mensagem seja igual ao do processo receptor, significa que o identificador deu uma volta completa no anel, vencendo todos os demais, e sendo portanto o maior identificador dentre todos os processos. Resta a esse processo se anunciar como coordenador para os demais. Inicialmente ele marca-se como não-participante e envia para seu vizinho uma mensagem do tipo coordenador contendo seu identificador. Ao receber uma mensagem de coordenador, o receptor marca-se como não participante, guarda o identificador do novo coordenador e envia a mensagem para

seu vizinho. Ao completar uma volta no anel, a mensagem de coordenador é removida e os processos voltam a trabalhar normalmente, considerando o processo vencedor da eleição como o novo líder. A Figura 3.14 apresenta a implementação do algoritmo.

A variável de estado é responsável por indicar se um processo é ou não é participante durante a execução em um determinado momento, e desempenha um papel fundamental no algoritmo; ela permite que, se várias eleições forem iniciadas ao mesmo tempo, as mensagens adicionais sejam extintas logo que possível, e sempre antes do novo coordenador ser anunciado.

Em relação à complexidade do algoritmo, o pior caso de execução acontece quando o processo que inicia a eleição é o vizinho seguinte (no sentido horário) do processo com o maior identificador. Nesse caso, serão necessárias N – 1 mensagens até atingir o processo que realmente será o futuro líder, o qual só então colocará seu identificador na mensagem que será enviada N vezes até completar o circuito. Por fim, a mensagem de coordenador será enviada N vezes para informar o resultado da eleição aos demais processos do anel, totalizando 3N - 1 mensagens.

```
/* Variáveis */
canal C[];              /* Canais de comunicação */
mensagem msg;           /* Variável para armazenamento da mensagem */
int i;                  /* Índice do processo */
int n;                  /* Número de processos */
boolean participante;   /* Indica se processo participa de eleição */
id idProprio;           /* Identificador único do processo */
id coordenador;         /* Identificador do processo coordenador */

/* Recebimento de mensagem do tipo eleição */
void evMsgEleicaoDisponivel(int j) {
    msg = receive(C[j]);
    if (msg.idMsg > idProprio) {
        participante = true;
        send(C[i%n + 1], msg);
    } else if (msg.idMsg < idProprio) {
        if (!participante) {
            msg.idMsg = idProprio;
            participante = true;
            send(C[i%n + 1], msg);
        } else {
            /* não faz nada */
        }
    } else {            /* msg fez a volta no anel */
        participante = false;
        msg.tipo = "coordenador";
        send(C[i%n + 1], msg);
    }
}

/* Recebimento de mensagem do tipo coordenador */
void evMsgCoordenadorDisponivel(int j) {
    msg = receive(C[j]);
    if (msg.idMsg <> idProprio) {
        participante = false;
        coordenador = msg.idMsg;
        send(C[i%n + 1], msg);
```

```
    } else {
        /* não faz nada: eleição encerrada */
    }
}
```

Figura 3.14: Implementação do algoritmo para eleição de líder em anel.

No melhor caso, o processo com o maior identificador inicia a execução, de forma que seu identificador seja enviado N vezes até completar uma volta no anel. Outras N mensagens serão necessárias para propagar o resultado da eleição, totalizando 2N mensagens.

3.5.2. Algoritmo "valentão" (Bully algorithm)

O algoritmo "valentão" para eleição de líd er ([COU 01], [TAN 02]) diferencia-se da solução anterior pela sua capacidade de tolerar falhas nos processos durante a execução da eleição (porém, a comunicação em rede deve ser confiável). Além disso, esse algoritmo faz uso de *timeouts* para detectar possíveis defeitos nos processos participantes, exigindo, portanto, que o sistema seja síncrono.

Outra diferença entre os dois algoritmos refere-se ao fato de que, na eleição de líder em anel, é requisitado que cada processo saiba apenas como se comunicar com seu vizinho. Essa solução, por outro lado, exige que cada processo saiba quais são os processos que possuem identificador maior que o seu e também como se comunicar com eles. O conjunto de processos é representado por P[1:n], sendo que cada um dos processos utiliza zero ou mais canais de comunicação do conjunto de canais C[1:n] para comunicar-se com cada um dos processos que possuírem identificador maior que o seu próprio identificador.

Essa solução utiliza três tipos de mensagens diferentes: a mensagem de eleição é utilizada por um processo para indicar um processo de eleição; a mensagem ok é utilizada para responder a uma mensagem de eleição; por último, uma mensagem do tipo coordenador é utilizada para que um processo se anuncie como o novo líder.

Uma eleição é iniciada quando um processo verifica que o atual coordenador não está mais respondendo (através da utilização de *timeouts*). Diversos processos podem perceber, ao mesmo tempo, que o coordenador falhou, iniciando assim uma ou mais eleições concorrentes.

Se o processo que iniciar a eleição apresentar o maior identificador dentre todos os processos, este pode simplesmente se anunciar para os outros como sendo o novo líder. Por outro lado, se o processo não possuir o maior identificador, este deve enviar uma mensagem de eleição para todos os outros que possuírem identificador maior que o seu e aguardar pelas mensagens de resposta. Caso o processo em questão não receba nenhuma resposta dentro de um determinado intervalo de tempo, este vence a eleição e pode considerar-se o novo líder, bastando apenas enviar mensagens de coordenador para todos os demais processos que possuírem identificador menor que o seu próprio identificador. De outra forma, se algum processo com identificador superior responder a mensagem de eleição com uma mensagem de ok, o processo que iniciou a eleição pode se retirar da disputa pela liderança, pois a tarefa será completada por algum processo com identificador superior.

Ao receber uma mensagem de eleição, o receptor responde com uma mensagem de ok, informando ao processo de identificador mais baixo a existência de pelo menos um superior. Além disso, cabe a esse processo de identificador mais alto iniciar uma

nova eleição (se ainda não iniciou), considerando apenas os processos que possuam identificador maior que o seu.

Quando um processo recebe uma mensagem de coordenador, o procedimento a ser tomado é apenas guardar o identificador recebido na mensagem e tratar o processo correspondente como o novo líder. A implementação do algoritmo é apresentada na Figura 3.15.

Um processo que é incluído (ou reincluído) ao conjunto de processos ativos é responsável por iniciar uma nova eleição. Caso esse processo possua o maior de todos os identificadores, ele vencerá a eleição, anunciará seu identificador para todos os demais processos e assumirá o papel de líder. Essa situação ilustra claramente que um novo líder pode ser eleito mesmo que o atual ainda esteja em funcionamento, justificando o nome dado ao algoritmo ("valentão").

Em relação à complexidade, o pior caso de execução corresponde à situação na qual o processo com o identificador mais baixo detectar a ausência de um líder e, a partir daí, N – 1 eleições serão iniciadas, sempre com os processos com identificadores mais baixos enviando mensagens para os processos com identificadores mais altos. Nesse caso, o algoritmo exige $O(N^2)$ mensagens.

```
/* Variáveis */
canal C[];              /* Canais de comunicação */
processo P[];           /* Processos que participam da escolha */
mensagem msg;           /* Variável para armazenamento da mensagem */
int i;                  /* Índice do processo */
int n;                  /* Número de processos */
boolean dispensado;     /* Indica se processo recebeu resposta */
id idProprio;           /* Identificador único do processo */
int timeout;            /* Timeout para recebimento de respostas */

/* Disparo de eleição (ao verificar que o líder não está ativo) */
void iniciarEleicao() {
    dispensado = false;
    for (k = 0; k < n; k++) {
        if (P[k].idProcesso > idProprio) {
            msg.tipo = "eleição";
            send(C[k], msg);
            if ((msg = receiveAssync(C[k])).tipo == "ok") {
                dispensado = true;
            }
        }
    }

    sleep(timeout);

    if (!dispensado) {
        for (k = 0; k < n; k++) {
            if (P[k].idProcesso < idProprio) {
                msg.tipo = "coordenador";
                send(C[k], msg);
            }
        }
    } else {
        /* não faz nada, pois outro processo assumirá a eleição */
    }
}

/* Recebimento de mensagem de eleição */
void evMsgEleicaoDisponivel(int j) {
```

```
    msg = receive(C[j]);
    msg.tipo = "ok";
    send(C[j], msg);
    iniciarEleicao();
}

/* Recebimento de mensagem do tipo coordenador */
void evMsgCoordenadorDisponivel(int j) {
    msg = receive(C[j]);
    coordenador = msg.idMsg;
}
```

Figura 3.15: Implementação do algoritmo "valentão" para realizar a escolha de líder.

O melhor caso de execução acontece quando o processo com o segundo identificador mais alto detectar que o líder atual não está mais respondendo. Caberá a esse processo eleger-se o novo líder e enviar um total de N – 2 mensagens para todos os demais processos que possuírem identificador menor que o seu para informar o resultado da eleição.

3.6. Obtenção de Instantâneos

Em um sistema distribuído, cada processo executa em sua memória local sem possuir uma visão global do sistema. Em diferentes situações pode ser desejável obter instantâneos do sistema (*consistent global state*, *global state* ou *snapshot*), isto é, obter estados globais durante a execução de um algoritmo assíncrono. Este é um problema bem conhecido na literatura [LYN 96] [BAR 96], e algumas das principais aplicações para este algoritmo são a depuração distribuída, a detecção de propriedades estáveis e a detecção de terminação.

Informalmente, podemos dizer que um instantâneo é "consistente" se ele parecer aos processos como se ele fosse obtido no mesmo instante em todos os pontos do sistema. Considere os exemplos da Figura 3.16. Em cada processador ou nó n_i, estados locais e_i são determinados pelo processamento local e pelo recebimento de mensagens de outros nós. O estado global representado pelo corte (a) é consistente uma vez que existe um ordenamento parcial coerente dos eventos. No entanto, o representado pelo corte (b) é inconsistente uma vez que existe um "evento futuro" no nó 2 em relação ao 1 (uma mensagem ainda não enviada por n_1 foi recebida por n_2). A seção 3.1 de Barbosa [BAR 96] pode ser consultada para uma definição formal deste problema.

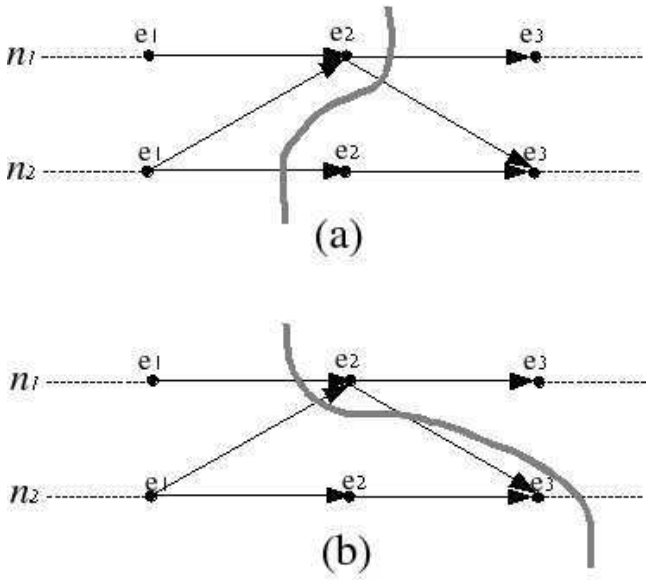

Figura 3.16: Exemplos de estados globais consistente (a) e inconsistente (b).

O algoritmo de obtenção de instantâneos pode ser expresso da seguinte forma:

- existe um algoritmo A que é um algoritmo de rede arbitrário que utiliza comandos do tipo *send/receive*;
- um instantâneo é obtido através de um algoritmo de monitoração B(A), que também é um algoritmo de rede com *send/receive*: qualquer execução correta de B(A) "contém" uma execução correta de A;
- cada processo $B(A)_i$ do algoritmo de monitoração B(A) é definido em termos do correspondente processo A_i;
- os vários estados reportados por $B(A)_i$ constituem um estado global de A;
- seja α uma execução de A, então deve existir uma execução α' de A, tal que as seguintes condições sejam observadas:
  - α' é indistinguível de α para cada processo A_i;
  - α' inicia com o prefixo α_1 de α ocorrendo antes do primeiro evento *snap* em uma dada execução de B(A);
  - α' termina com o sufixo α_2 de α ocorrendo após o último evento *report* em uma dada execução de B(A);
  - o estado retornado é exatamente o estado global após o prefixo de α' que inclui todos os α_1 e nenhum dos α_2.

3.6.1. Algoritmo LogicalTimeSnapshot

O ponto central deste algoritmo é uma subrotina que utiliza um tempo lógico pré-determinado $t \in T$, que se assume que seja conhecido por todos os processos. Assim, no tempo lógico t cada processo guarda localmente seu estado o que garante um estado global consistente.

Assim, assumindo que t seja conhecido, temos os seguinte passos [LYN 96]:

1. determinar o estado de cada processo de A após todos os eventos com tempo lógico menor ou igual que t e antes de todos os eventos com tempo lógico maior que t;
2. para cada canal de comunicação, determinar a seqüência de mensagens enviadas com tempo lógico menor ou igual a t mas recebidas no tempo lógico estritamente maior que t (mensagens em trânsito).

Nesse algoritmo, é utilizado o conceito de relógio lógico tal como definido na seção 3.3.2.

Na Figura 3.17 é apresentado um pseudo-código descrevendo a lógica desse algoritmo, através de dois procedimentos, a *recebeMensagens()* (que realiza o passo 1) e a *processaMensagem()* (que realiza o passo 2).

```
/* Variáveis */
int    tempoLogico; /* tempo lógico local*/
```

```
                    /* atualizado segundo algoritmo de Lamport */
canal  out[];  /* Canais de comunicação para envio de msgs*/
canal  in[];   /* Canais de comunicação recebimento de msgs */

/* Recebimento de mensagens */
void recebeMensagens() {
  for (int j = 0; j< MAX_IN; j++)
      if probe(in[j]) {
        mensagem dado = receive(in[j]);
        atualizaRelogioLogico();
        if ((dado.tempoLogicoEnvio <=t) && (tempoLogico > t))
           guardaMensagemEmTransito(dado);
        processaMensagem(dado);
      }
}

/* Rotina de armazenamento de estado */
void processaMensagem(mensagem dado) {
  if (tempoLogico == t)
     guardaEstadoLocal();
  ...
}
```

Figura 3.17: Pseudo-código do instantâneo com RL.

3.6.2. Algoritmo de Chandy-Lamport

Este algoritmo é similar ao anterior, porém não existe um tempo lógico explícito. Ao invés do tempo lógico, utilizam-se mensagens com marcadores (*marker*) para indicar pontos de divisão entre as mensagens enviadas no antes do tempo *t* e as enviadas após o tempo *t*. A seguir apresentamos a descrição informal tal como apresentada em [LYN96].

Quando um processo *ChandyLamport(A)*$_i$ que não tenha previamente sido envolvido no algoritmo de instantâneo recebe uma entrada snap$_i$, ele registra o estado corrente de A_i.

Então ele imediatamente envia uma mensagem *marker* para cada um dos seus canais de saída; este *marker* indica uma delimitação entre as mensagens que foram enviadas antes do estado local ser registrado e as mensagens que serão enviadas posteriormente. Neste momento, inicia-se o armazenamento das mensagens de entrada em cada canal de entrada a fim de obter o estado do canal; este armazenamento de mensagens prossegue até que seja encontrada uma mensagem *marker*. Neste ponto, *ChandyLamport(A)*$_i$ armazenou todas as mensagens enviadas pelo canal antes do vizinho na outra ponta do canal ter registrado seu estado local.

Caso o processo *ChandyLamport(A)*$_i$ receba uma mensagem *marker* antes de ter registrado o estado local de A_i, imediatamente após receber esta mensagem, *ChandyLamport(A)*$_i$ armazena o estado corrente de A_i, envia a mensagem *marker* e inicia o registro das mensagens. Observe-se que o canal no qual acabou de ser recebida a mensagem *marker* é armazenado como vazio.

```
/* Variáveis */
boolean   registroAtivo = false; /*inicialmente não registra estado*/
canal     out[];  /* Canais de comunicação para envio de msgs*/
```

```
canal     in[];   /* Canais de comunicação recebimento de msgs */

/* Recebimento de mensagens */
void recebeMensagens() {
  for (int j = 0; j< MAX_IN; j++)
      if probe(in[j]) {
        mensagem dado = receive(in[j]);
        ChandyLamport(dado,j)
      }
}

/* Rotina de armazenamento de estado */
void ChandyLamport(mensagem dado, int nroDoCanal) {
  if (dado == "marker") {
     if (!registroAtivo ) {
         /* inicia processo de armazenamento de estado local */
         registroAtivo  = true;
         encerrarRegistroDeMsgNoCanal(nroDoCanal);
         armazenaEstadoLocal();
         /* envia marker para todos os canais de saída */
        for (int j = 0; j < MAX_OUT; j++)
           send(out[j], "marker");
     } else {  /* registroAtivo  == true */
         encerrarRegistroDeMsgNoCanal(nroDoCanal);
     }
   } else { /* dado != "marker" */
     if (registroAtivo ) {
        armazenarMsgEstadoLocal(dado,nroDoCanal);
     }
     /* todas as mensagens diferentes de marker */
     /* devem ser processadas pelo sistema      */
     processarMensagemNormalmente();
   }
}
```

Figura 3.18: Pseudo-código do instantâneo Chandy-Lamport.

3.7. Considerações Finais

Esse texto apresentou uma breve introdução ao tema algoritmos distribuídos através da apresentação de problemas clássicos e da solução através de algoritmos conhecidos na literatura. O estudo de algoritmos distribuídos vem se tornando cada vez mais importante para a formação de uma base sólida em ciência da computação. Esta base servirá no futuro como um importante diferencial para profissionais que cada vez mais atuam em ambientes impregnados de tecnologia distribuída tais como redes locais, Internet, *web services,* servidores replicados, entre outros.

É importante notar que existem na literatura várias propostas de algoritmos mais eficientes para tratar os problemas aqui apresentados. As escolhas recaíram sobre aqueles apresentados em livros didáticos e que servem de base para grande parte dos novos algoritmos propostos. Aos que se sentirem motivados a continuar os estudos nesta área recomenda-se consultar as referências bibliográficas constantes neste texto, bem como buscar artigos em conferências de sistemas distribuídos e em revistas da ACM e IEEE.

Agradecimentos: agradecemos ao aluno Bruno Bohrer Cozer (ECP-UFRGS) pela ajuda na elaboração desse trabalho.

3.8. Bibliografia

[AND 00] ANDREWS, G. R. **Foundations of Multithreaded, Parallel, and Distributed Programming.** Addison-Wesley; 1st edition, 2000. 664p.

[BAR 96] BARBOSA, V. C. **An Introduction to Distributed Algorithms.** London, England: The MIT PRess, 1996. 365p.

[COU 01] COULOURIS, G., DOLLIMORE, J., KINDBERG, T**. Distributed Systems: Concepts and Design.** Addison-Wesley; 3rd edition, 2001. 772p.

[LAM 78] LAMPORT, L. **Time, clocks, and the ordering of events in a distributed system.** Communications of the ACM, v.21, n.7 p.558—565, July 1978.

[LYN 96] LYNCH, N. A. **Distributed Algorithms.** Morgan Kaufmann Publishers, Inc., 1996. 872p.

[RAY 88] RAYNAL, M. **Distributed Algorithms and Protocols.** John Wiley & Sons, 1988. 163p.

[RIC 81] RICART, G. and AGRAWALA, A. K. **An optimal algorithm for mutual exclusion in computer networks.** Communications of the ACM, vol. 24, no. 1, pp. 9-17.

[SIL 02] SILBERSCHATZ, A. et al. **Operating System Concepts.** John Wiley & Sons, 6th edition, 2002. 976p.

[TAN 02] TANENBAUM, A. S. et al. **Distributed Systems**. Prentice Hall, 2002. 803p.