4. Sincronização em Sistemas Distribuídos

A sincronização entre processos é tão importante quanto a comunicação entre processos em sistemas distribuídos. Por exemplo, como as regiões críticas são implementadas em um sistema distribuído, e como estes recursos são alocados?

Em sistemas de uma única CPU, regiões críticas, exclusão mútua, e outros problemas de sincronização são geralmente resolvidos usando métodos como semáforos e monitores. Estes métodos não são recomendados para serem usados em sistemas distribuídos porque eles invariavelmente contam (implicitamente) com a existência de uma memória compartilhada. Por exemplo, dois processos que estão interagindo usando um semáforo, ambos devem ser capazes de acessar o semáforo. Se eles estão rodando na mesma máquina, eles podem compartilhar o semáforo tendo−o armazenado no *Kernel,* e executar chamadas de sistema (*system calls*) para acessá−lo. Se, entretanto, eles estiverem rodando em diferentes máquinas, este método não mais funcionará, e outras técnicas devem ser utilizadas. Mesmo parecendo problemas simples, como determinar se o evento *A* aconteceu antes ou depois do evento *B,* requer bastante cuidado.

Inicialmente será focalizado o tempo e a forma de medi−lo, devido ao fato de que o tempo é a principal parte de alguns métodos de sincronização. Em seguida será visto a exclusão mútua e os algoritmos de eleições. Depois será estudado uma técnica de sincronização de alto−nível chamada de transação atômica. Finalmente, será estudado o *deadlock.*

4.1 Sincronização de relógio

A sincronização em sistemas distribuídos é mais complicada do que em sistemas centralizados porque em sistemas distribuídos é necessário utilizar algoritmos distribuídos. Não é usualmente possível (ou desejável) coletar todas as informações sobre o sistema em um único lugar, e depois deixar que alguns processos examinem−na e tomem uma decisão como é feito no caso centralizado. Em geral, algoritmos distribuídos possuem as seguintes propriedades:

1− A informação relevante está espalhada em múltiplas máquinas

2− Processos tomam decisões baseadas somente nas informações locais

3− Um único ponto de falha no sistema deve ser evitado

4− Não existe um relógio em comum ou outro tipo preciso de tempo global

Os primeiros três pontos significa dizer que é inaceitável colecionar todas as informações em um único local para o processamento. Por exemplo, para fazer alocação de recursos, não é aceitável enviar todas as requisições para um único processo gerenciador, o qual examina as requisições e

permite ou nega as requisições baseadas nas informações contidas em suas tabelas. Em sistemas grandes, esta solução coloca uma carga pesada neste único processo.

Além do mais, tendo um único ponto de falha como este torna o sistema não confiável. Idealmente, um sistema distribuído deve ser mais confiável do que as máquinas individuais. Se uma máquina sair do ar, o restante deve ser capaz de continuar funcionando. Atingir a sincronização sem centralização requer fazer coisas de um modo diferente dos sistemas operacionais tradicionais.

O último ponto é também crucial. Nos sistemas centralizados, o tempo não é ambíguo. Quando um processo quer saber o tempo, ele faz uma chamada ao sistema e o Kernel responde. Se o processo *A* pergunta pelo tempo e então o próximo processo *B* pergunta pelo tempo, o valor que *B* possui será maior (ou possivelmente) igual ao que *A* possui. Ele certamente não será menor. Em um sistema distribuído, atingir a concordância do tempo não é trivial.

Imagine as implicações da falta de um tempo global no programa *make* do *UNIX,* como exemplo. Normalmente, no UNIX, grandes programas são separados em múltiplos arquivos fontes, logo uma mudança em um desses arquivos fonte, somente requer que um arquivo seja compilado, não todos.

Quando o programador termina as mudanças nos arquivos fontes, ele executa o *make,* o qual examina o tempo no qual todos os fontes e arquivos objetos (*object file*) tiveram sua última modificação. Se o arquivo fonte *input.c* possui o tempo 2151 e o correspondente arquivo objeto possui o tempo 2150, *make* sabe que *input.c* foi modificado após a sua criação, e logo *input.c* deve ser recompilado. Por outro lado, se *output.c* possui o tempo 2144 e *output.o* possui o tempo 2145, nenhuma compilação se faz necessária. Logo o *make* percorre todos os arquivos fontes para achar quais precisam ser compilados e chamar o compilador para recompilá−los.

Nos sistemas distribuídos, onde não há uma concordância no tempo global, suponha que *output.o* tem o tempo de 2144 como acima, e pouco tempo depois *output.c* foi modificado mas o tempo rotulado é de 2143, porque o *clock* da máquina onde o arquivo foi modificado é um pouco atrasado em relação ao clock da máquina na qual o compilador está rodando. *Make* não chamará o compilador. Como resultado o programa binário executável conterá uma mistura de arquivos objetos de fontes antigas e novas. Provavelmente isto não funcionará.

1. Relógios Lógicos

Quase todos os computadores possuem um circuito para manter−se a par do tempo. O *timer* utilizado nos computadores é geralmente produzido a partir de cristal de quartzo. Quando mantido sob tensão, o cristal de quartzo oscila em uma freqüência bem−definida que depende do tipo de cristal,

como é cortado, e a quantidade de tensão. Associado a cada cristal existem dois registros, um *conter* e um *holding register*. Cada oscilação do cristal decrementa o *counter* por um. Quando o *counter* chega a zero, uma interrupção é gerada e o *counter* é recarregado pelo *holding register*. Deste modo, é possível programar o timer para gerar uma interrupção 60 vezes por segundo, ou qualquer outra freqüência desejada. Cada interrupção é chamada *clock tick.*

Quando o sistema é inicializado, ele usualmente pergunta ao operador para entrar com a data e hora, o qual é então convertido para o número de ticks depois da data conhecida e armazenada na memória. A todo clock tick, o serviço de interrupção adiciona uma unidade ao tempo armazenado na memória. Deste modo, o software clock é mantido atualizado.

Com um único computador e um único *clock*, não importa muito se este relógio estiver um pouco fora do tempo. Desde que todos os processos na máquina usem o mesmo relógio, eles ainda estarão internamente consistentes. Por exemplo, se o arquivo *input.c* tem o tempo 2151 e o arquivo *input.o* tem o tempo 2150, o *make* irá recompilar o arquivo fonte, mesmo se o relógio estiver fora do tempo correto por 2 unidades de tempo sendo o tempo correto igual a 2153 e 2154, respectivamente. Tudo o que realmente importa são os tempos relativos.

Assim que múltiplas CPUs são introduzidas, cada qual com seu próprio relógio, a situação muda. Embora a freqüência na qual o oscilador de cristal roda é usualmente estável, não é possível garantir que os cristais nos diferentes computadores rodarão na mesma freqüência. Na prática, quando um sistema possui n computadores, todos n cristais rodarão em taxas ligeiramente diferentes, levando os relógios a ficarem gradualmente fora de sincronismo. Esta diferença em valores de tempo é chamada *clock skew*. Como conseqüência deste *clock skew*, programas que esperam o tempo associado com um arquivo, objeto, processo, podem falhar.

Lamport apresentou um algoritmo que torna possível sincronizar todos os *clocks* para produzir um único, tempo padrão não ambíguo. Lamport afirmou que a sincronização dos relógios não precisa ser absoluta. Se dois processos não se interagem, não é necessário que os seus relógios sejam sincronizados porque a falta de sincronização não seria significativa e logo não causará problemas. Entretanto, ele afirmou o que usualmente importa não é se todos os processos estejam corretamente sincronizados, mas se eles concordam na ordem em que os eventos ocorrem. No exemplo do *make* acima, o que conta é se *input.c* é mais velho ou novo do que *input.o*, não seus tempos absolutos de criação.

Para muitos propósitos, é suficiente que todas as máquinas estejam no mesmo tempo. Não é essencial que este tempo concorde com o tempo real. Para rodar o *make*, por exemplo, é adequado que todas as máquinas concordem que são 10:00, mesmo se na realidade for 10:02. Então para certas classes de algoritmos, a consistência interna dos relógios é que importa, não se eles estão

particularmente próximos ao tempo real. Para estes algoritmos, o importante é a sincronização lógica dos relógios(*logical clocks*).

2. Relógio Físico

Embora o algoritmo de Lamport dê uma ordem não ambígua aos eventos, o valor do tempo associado aos eventos não é necessariamente próximo do tempo real no qual eles ocorrem. Em alguns sistemas (por exemplo, sistemas de tempo−real), o tempo real é importante. Para estes sistemas relógios físicos externos são necessários. Por razões de eficiência e redundância, múltiplos relógios físicos são geralmente considerados desejáveis, os quais trazem dois problemas: (1) Como será feito a sincronização deles com os relógios do mundo real, e (2) Como sincronizar os relógios entre si.

Não é tão simples medir o tempo, especialmente quando é necessária alta precisão. Desde a invenção dos relógios mecânicos no século 17, o tempo tem sido medido astronomicamente. Com a invenção do relógio atômico em 1948, tornou−se possível medir o tempo mais precisamente através da contagem de transições do átomo de césio 133. Os físicos definiram o segundo como sendo o tempo de que o átomo de césio 133 leva para fazer exatamente 9.192.631.770 transições. A escolha de 9.192.631.770 foi feita para fazer com que o segundo atômico fosse igual ao segundo solar no ano de sua introdução. Atualmente, cerca de 50 laboratórios em volta do mundo possuem o relógio de césio 133. Periodicamente, cada laboratório diz ao *Bureau intertional de I'Heure* (BIH) em Paris quantas vezes o seu relógio marcou. O BIH tira a média para produzir o Tempo Atômico Internacional, o qual recebe a sigla TAI.

Embora o TAI seja altamente estável, há um sério problema com ele; 86.400 segundos TAI é agora cerca de 3 mseg a menos do que um dia solar. O uso do TAI para manter o tempo poderia significar que através do decorrer dos anos, o meio−dia ficaria cada vez mais cedo, até que ocorreria nas primeiras horas da manhã.

BIH resolveu o problema através da introdução do *leap seconds* sempre que a discrepância entre TAI e o tempo solar chega a 800 mseg. Esta correção aumenta o tempo do sistema baseado nos segundos TAI, mas fica em fase com o aparente movimento do sol. Isto é chamado *Universal Coordinated Time*, mas é abreviado como UTC.

Para fornecer UTC para as pessoas que necessitam do tempo preciso, o *National Institute of Standard Time* (NIST) opera uma estação de rádio de ondas curtas com as letras WWV em Fort Collins, Colorado. WWV envia por *broadcasts* um pulso no começo de cada segundo UTC. A exatidão do WWV é cerca de ? 1mseg, mas devido a uma atmosfera randômica que pode afetar o distancia do caminho do sinal, na prática a precisão não é melhor do que ? 10 mseg. Na Inglaterra, a estação MSF,

operando de Rugby, Warwickshire, fornece um serviço similar, como é feito pelas estações em outros países.

Muitos satélites também oferecem um serviço UTC. O *Geostationary Environment Operational Satellite* pode fornecer o UTC com uma precisão de 0.5mseg, e alguns outros satélites fazem ainda melhor.

O uso de ondas curtas de rádio ou satélite requer um conhecimento preciso da posição relativa do transmissor ou receptor, de forma a compensar o tempo de propagação do sinal. Receptores de rádio de WWV, GEOS, e outras fontes UTC são comercialmente disponíveis. UTC pode também ser obtido através do telefone NIST em Fort Collins, mas aqui também, a correção deve ser feita pelo caminho do sinal e velocidade do modem.

BIH resolveu o problema através da introdução do *leap seconds* sempre que a discrepância entre TAI e o tempo solar chega a 800 mseg. Esta correção aumenta o tempo do sistema baseado nos segundos TAI, mas fica em fase com o aparente movimento do sol. Isto é chamado *Universal Coordinated Time*, mas é abreviado como UTC.

Para fornecer UTC para as pessoas que necessitam do tempo preciso, o *National Institute of Standard Time* (NIST) opera uma estação de rádio de ondas curtas com as letras WWV em Fort Collins, Colorado. WWV envia por *broadcasts* um pulso no começo de cada segundo UTC. A exatidão do WWV é cerca de ? 1mseg, mas devido a uma atmosfera randômica que pode afetar o distancia do caminho do sinal, na prática a precisão não é melhor do que ? 10 mseg. Na Inglaterra, a estação MSF, operando de Rugby, Warwickshire, fornece um serviço similar, como é feito pelas estações em outros países.

Muitos satélites também oferecem um serviço UTC. O *Geostationary Environment Operational Satellite* pode fornecer o UTC com uma precisão de 0.5mseg, e alguns outros satélites fazem ainda melhor.

O uso de ondas curtas de rádio ou satélite requer um conhecimento preciso da posição relativa do transmissor ou receptor, de forma a compensar o tempo de propagação do sinal. Receptores de rádio de WWV, GEOS, e outras fontes UTC são comercialmente disponíveis. UTC pode também ser obtido através do telefone NIST em Fort Collins, mas aqui também, a correção deve ser feita pelo caminho do sinal e velocidade do modem.

3. Relógios sincronizados

Está havendo, recentemente, uma fácil disponibilidade de hardware e software para sincronização de *clocks* em larga escala (Ex. sob a Internet). Novos e interessantes algoritmos que

fazem uso da sincronização são cada vez mais freqüentes. Um exemplo deste tipo de algoritmo é o algoritmo de consistência de *cache* com base em *clock.*

Na maioria dos sistemas existentes é desejado que cada usuário tenha uma cópia local de arquivos por questões de desempenho. Entretanto, o *cache* introduz problemas de inconsistência se dois clientes modificam o mesmo arquivo ao mesmo tempo. A solução mais adequada é distinguir entre cache de arquivos para leitura e *cache* de arquivos para escrita. A desvantagem deste esquema é que se um cliente tem um arquivo "cacheado" para leitura, antes de conceder uma cópia de escrita para um outro cliente, o servidor precisa primeiro dizer ao cliente detentor da cópia de leitura para anulá−la. Este *overhead* extra pode ser eliminado usando relógios sincronizados.

3.1 Sincronização física de Relógios

Cada computador possui seu próprio relógio, os quais oscilam em freqüências diferentes uns dos outros. Isto não é um problema quando não é necessário saber o tempo nos demais computadores. Para uma aplicação local o que vale é a seqüência em que os eventos ocorrem, como citado no algoritmo de Lamport. Porém, em certas aplicações é fundamental que os relógios do sistema trabalhem de uma forma sincronizada e marcando tempos bem próximos. A sincronização física de relógios não é uma tarefa trivial, principalmente quando se deseja uma discrepância mínima entre os relógios.

Esta sincronização pode resultar em dois problemas principais [TANEMBAU −97]. Primeiro, um relógio não pode voltar atrás(se estiver marcando 300, o relógio não poderá ser corrigido para 299, tendo em vista que várias aplicações dependem desta semântica). Desta forma, se um computador necessita sincronizar−se com um outro computador cujo relógio está atrasado, ele terá que reduzir a freqüência de seu *clock* de modo que trabalhe um pouco mais lento e possa se igualar ao *clock* do computador com o qual está se sincronizando.

Segundo, em sistemas distribuídos, não se pode determinar com precisão o *delay* de transmissão da rede. Por exemplo: Um cliente desejando sincronizar−se com um servidor envia−lhe uma requisição. O servidor responde devolvendo o tempo atual de seu *clock*. O problema é que quando o cliente recebe de volta a mensagem, o tempo retornado já está desatualizado.

Alguns algoritmos foram propostos para tentar amenizar os problemas de sincronização, como o algoritmo de Cristian e o algoritmo de Berkeley.

4.3 Exclusão Mútua

Sistemas envolvendo múltiplos processos são mais facilmente programáveis usando regiões

críticas. Quando um processo tem que ler ou atualizar determinadas estruturas de dados compartilhados, ele primeiro entra na região crítica para atingir a exclusão mútua e garantir que nenhum processo irá usar as estruturas de dados compartilhadas no mesmo tempo. Em sistemas de um único processador, regiões críticas são protegidas usando semáforos, monitores, e construções similares. Serão apresentados alguns exemplos de como regiões críticas e exclusões mútuas podem ser implementadas em sistemas distribuídos.

− Algoritmos Centralizados

O caminho mais rápido para atingir exclusão mútua em sistemas distribuídos é similar ao feito em sistema com um único processador. Um processo é eleito como o coordenador (por exemplo, um rodando na máquina com o maior endereço da rede). Sempre que um processo quiser entrar na região crítica, ele envia uma requisição de mensagem para o coordenador declarando qual região ele quer entrar e requisitando permissão. Se nenhum outro processo estiver atualmente naquela região crítica, o coordenador envia uma resposta de volta dando permissão. Quando a resposta chega, o processo de requisição entra na região crítica.

Quando o processo 1 sair da região crítica, ele envia uma mensagem para o coordenador liberando o acesso exclusivo. O coordenador toma a requisições da fila de requisições, e envia ao processo uma mensagem de permissão. Se o processo ainda estiver bloqueado (por exemplo, é a primeira mensagem para ele), ele é desbloqueado e entra na região crítica.

Este algoritmo garante exclusão mútua: o coordenador somente deixa um processo, por tempo, entrar na região crítica. É também justo, desde que as concessões as requisições são feitas na ordem na qual elas são recebidas. Nenhum processo espera para sempre. O esquema é fácil de implementar, também, e requer somente três mensagens para uso da região crítica (requisição, concessão, liberação). Pode também ser usado para alocação mais geral dos recursos ao invés de somente gerenciar regiões críticas.

A abordagem centralizada também possui problemas. O coordenador é um único ponto de falha, logo se ela quebrar, todo o sistema pode vir abaixo. Se os processos normalmente bloquearem depois de fazer uma requisição, eles não poderão distinguir um coordenador "morto" de uma "permissão negada" desde que em ambos nenhuma mensagem retorna. Em adição, em sistemas grandes, um único coordenador pode tornar um gargalo da performance.

− Algoritmos Distribuídos

Tendo um único ponto de falhar é inaceitável, logo pesquisadores tem estudado algoritmo para exclusão mútua distribuída.

O algoritmo de Ricard e Agrawala requer que haja uma ordem total de todos os eventos no sistema. Ou seja, para qualquer par de eventos, como mensagens, não deve haver ambigüidade na qual o primeiro acontece. O algoritmo de Lamport apresentado antes, é um modo de alcançar esta ordenação e pode ser usada para fornecer *time−stamp* para a exclusão mútua distribuída.

O algoritmo funciona como segue. Quando um processo que entrar na região crítica, ele constrói uma mensagem contendo o nome da região crítica que está interessado, o número do processo, e o tempo corrente. Ele envia a mensagem para todos os outros processos, conceitualmente incluindo−o. Um grupo confiável de comunicação se disponível, pode ser usado ao invés de mensagens individuais.

Quando um processo recebe uma requisição de outro processo, a ação que toma depende do seu estado com relação a região crítica nomeada na mensagem. Três casos devem ser distinguidos:

1 − Se o receptor não é a região crítica e não quer entrar na região crítica, ele retorna um *OK* para o transmissor.

2 − Se o receptor estiver na região crítica, ele não responderá. Ao invés, ele enfileira a requisição.

3 − Se o receptor quiser entrar na região crítica mas não o fez ainda, ele compara o *timestamp* da mensagem com o contendo na mensagem que ele enviou para todos. A menor ganha. Se a mensagem que chegar tiver menor prioridade, o receptor envia uma mensagem de *OK*. Se sua própria mensagem tiver um *timestamp* inferior, o receptor enfileira a mensagem e não envia nada.

Depois de enviar as requisições pedindo autorização para entrar na região crítica, um processo espera até que todos tenham dado permissão. Assim que todas as permissões cheguem, ele poderá entrar na região crítica. Quando ele sai da região crítica, ele envia um *OK* para todos os processos da sua fila e deleta−os de sua fila. Se não há um conflito, ele funcionará corretamente.

− Algoritmo Token Ring

Uma abordagem totalmente diferente para obter a exclusão mútua em um sistema distribuído é apresentado um barramento (por exemplo, Ethernet), sem ordenação dos processos. Por software, um anel circular é construído no qual cada processo é associado a uma posição no anel. As posições no anel podem ser alocadas em uma ordem numérica dos endereços da rede ou de outra forma. Não importa qual é a ordem. O que importa é que cada processo saiba que é o próximo no anel.

Quando o anel é inicializado, processo o recebe um *token.* O *token* circula pelo anel. Passa pelo

processo K para o processo K + 1 em mensagens ponto−a−ponto. Quando um processo adquire o *token* do processo vizinho, ele checa para ver se precisa entrar na região crítica. Se precisar, o processo entra na região crítica, efetua o trabalho a ser feito, e libera a região. Depois de sair, ele passa o token para o anel. Não é permitido entrar numa segunda região crítica usando o mesmo *token.*

Se o processo recebe o token de seu vizinho e não está interessado em entrar na região crítica, ele somente passa o *token* para frente. Como conseqüência, quando nenhum processo quer entrar em nenhuma região crítica, o *token* somente circula em alta velocidade ao longo do anel.

Deste modo somente um processo tem o *token* em qualquer instante, logo somente um processo pode estar na região crítica. Visto que o *token* circula pelos processos em uma ordem bem−definida, *starvation* não poderá ocorrer. Uma vez que um processo decide entrar na região crítica, no pior caso ele terá que esperar que todos os outros processos entrem na região crítica e liberem−na.

Como sempre, este algoritmo também possui problemas. Se o *token* for perdido, ele deve ser gerado novamente. De fato, detectar que o *token* foi perdido é difícil, visto que o tempo entre sucessivas aparições do *token* da rede não é limitado. O fato de que o *token* não foi avistado por uma hora não significa que ele foi perdido; alguém pode ainda estar utilizando−o.

O algoritmo também tem problemas se um processo falha, mas a recuperação é mais fácil do que os outros casos.Um processo ao passar o *token* para seu visinho pode perceber, se for o caso, que ele está fora do ar. Neste ponto, o processo fora do ar pode ser removido do grupo, e o processo que possui o *token* enviará para o próximo processo, no anel. Para isso, é necessário que todos mantenham a configuração atual do anel.

− Comparação dos três algoritmos

O algoritmo centralizado é o mais simples e também o mais eficiente. Ele necessita somente de três mensagens para entrar e liberar a região crítica: uma requisição e a autorização, e a liberação para sair. O algoritmo distribuído requer n−1 requisições, uma para cada processo, e um adicional de n−1 autorizações, para um total de 2(n−1). Com o algoritmo do anel de *token,* o número é variável. Se todos processos constantemente quiser entrar na região crítica, então cada *token* passado resultará em uma entrada e saída, para uma média de uma mensagem por entrada na região crítica. Em outro extremo, o *token* poderia algumas vezes circular por horas sem ninguém estar interessado nele. Neste caso, o número de mensagens por entrada na região crítica não é limitado.

O tempo que o processo aguarda para entrar na região crítica até a entrada varia nos três algoritmos. Quando regiões críticas são pequenas e raramente usadas, o fator dominante na espera é o mecanismo corrente para entrar na região crítica. Quando são longas e freqüentemente usadas, o fator

dominante é esperar que todos tenham sua vez.

Finalmente, todos os três algoritmos reagem mal a eventos de quebra. Medidas especiais e complexidade adicional devem ser introduzidas para impedir que uma quebra faça com que todo o sistema saia do ar. É um pouco irônico que os algoritmos distribuídos são mais sensíveis a falhas do que os centralizados. Em um sistema tolerante a falhas, nenhum deles seria indicado, mas se as falhas não são muito freqüentes, eles são aceitáveis.

−Algoritmo de Eleição

Em muitos algoritmos de sistemas distribuídos existe a necessidade de que um processo desempenhe funções especiais tais como coordenar, inicializar, seqüenciar, etc. Semelhante ao já visto nos sistemas centralizados em algoritmos de exclusão mútua. Em geral não importa qual o processo toma para si a responsabilidade, mas alguém tem que desempenhar este papel. Nesta seção serão mostrados algoritmos para a eleição de um coordenador.

Um pergunta que surge, mas que é facilmente respondida, é como distinguir um processo eleito dos outros já que nenhum dele possui uma característica distinta? Para responder esta dúvida [TANEMBAU−1995] foi adotado que cada processo possui uma identificação única, por exemplo, o endereço de rede. Em geral, para eleger o coordenador procura−se pelo processo que tenha o nr de rede mais alto. Existem diferentes maneiras para fazer está localização.

Além disso, será assumido que todo processo conhece o número dos outros processos e que os processos não sabem quais deles estão correntemente ativos ou não. A meta da eleição é assegura que após o início da eleição os processos concordem entre si quem é o novo coordenador. Abaixo serão mostrados alguns algoritmos que tratam este assunto.

− Algoritmo de Bully

Este algoritmo, (Tanembaum *apud* Garcia−Molina (1982)), quando verifica que o coordenador está morto, inicia uma eleição da seguinte forma:

1. P envia uma mensagem de ELEIÇÃO para todos os processos.
2. Se nenhum processo responde então ele se torna o coordenador
3. Se alguém que responder tiver o número maior do que o de P então P’ será tomado sobre ele.

Em qualquer momento um processo de menor número pode enviar uma mensagem de ELEIÇÃO. Quando isto acontece o processo coordenador de maior número responde ao *sender* com uma mensagem de OK indicando que ele está ativo e assim o processo *sender* se cala. Entretanto se o processo que enviou a mensagem de ELEIÇÃO for um número maior do que o coordenador. O processo *sender* então será eleito como o novo coordenador. Desta forma garante que se um processo

de maior número ficar inativo ou *crasher* temporariamente ele poderá voltar a ser o novo coordenador. Assim sempre ficará no topo o processo de maior número por isso chamado "Algoritmo do Valentão".

− Algoritmo do Anel

Um outro modelo de eleição do coordenador é utilizando um anel mas sem a presença do token. Os processos estão ordenados logicamente ou fisicamente, desta forma cada processo sabe quem é o seu sucessor. Quando um processo percebe que o seu coordenador está desativo ele constrói uma mensagem ELEICÃO com o seu identificador e a passa para o seu sucessor, se o seu sucessor estiver inativo a mensagem passará para o próximo assim por diante até que a mensagem volte ao processo que enviou. Durante este processo cada processo coloca o nr do seu identificador. Após a mensagem circular por todos os processos, o processo que enviou a mensagem (isto porque ele verifica o seu próprio nr na mensagem) altera a mensagem para o tipo COORDENADOR e coloca novamente para circular para que todos reconheçam quem é o coordenador e quais os novos membros do anel. Assim que a mensagem retornar novamente ao processo transmissor, ela é retirada do anel e todos voltam a trabalho novamente.

4.4Deadlock

Um deadlock é causado pela situação onde um conjunto de processos está bloqueado permanentemente, isto é, não consegue prosseguir a execução, esperando um evento que somente outro processo do conjunto pode causar.

– Condições necessárias para ocorrer o Deadlock no sistema:

- Exclusão mútua;
- segura e espera;
- não preempção;
- espera circular.

– Estratégias mais utilizadas para trabalhar o Deadlock

- algoritmo do avestruz (ignorar o problema);
- detecção (permite que o deadlock ocorra, detecta e tenta recuperar);
- prevenção (estaticamente faz com que o deadlock não ocorra);
- espera circular (evita deadlock alocando recursos cuidadosamente).

Algoritmo de detecção distribuída

O algoritmo de detecção funciona da seguinte maneira: O algoritmo é executado quando um processo tem que esperar por algum recurso em função de um outro processo esta utilizando o mesmo. Quando esta situação ocorre, uma mensagem especial é enviada (probe message) e enviada para o processo que esta utilizando o recurso, sendo composta de três partes: informações sobre o número do processo que está bloqueando, o número do processo que esta enviando a mensagem e o número do processo para quem a mensagem está sendo enviada. Quando a mensagem chega a um processo, o processo verifica se está esperando por algumrecurso que esta em uso por outro processo. Se estiver, então a mensagem é atualizada, mantendo−se o primeiro campo, mas trocando o segundo campo por seu número de processo e o terceiro pelo número do processo que está esperando desbloqueamento do recurso. Se está bloqueado devido a diversos processos, então a mensagem é enviada para todos os processos que detem o recurso que o processo necessita. Se a mensagem dá toda a volta e chega no processo que iniciou a mensagem, isto é o processo cujo identificado está no primeiro campo da mensagem, existe um ciclo, logo o sistema está em deadlock.