Detectores Perfeitos em Sistemas Distribuídos Não Síncronos

Raimundo José de Araújo Macêdo[1], **Sérgio Gorender**[1]

[1]Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computacao
Universidade Federal da Bahia
Campus de Ondina - Salvador - BA - Brasil

{macedo,gorender}@ufba.br

Abstract. *In this paper we show that is possible to implement a perfect failure detector P (one that suspect all faulty processes if and only if those processes failed) in a system weaker than the synchronous system (contradicting a common believe). To realize that, we introduce the partitioned synchronous system (Spa) that is strictly weaker than the conventional synchronous system (it is not always possible to implement global synchronous computations in Spa such as internal clock synchronization). From some properties we introduce (such as strong partitioned synchrony), we show how to implement P over Spa. Moreover, we show that even if strong partitioned synchrony cannot be sustained, we still are able to take advantage of the existing synchronous partitions to improve the robustness of applications, by introducing a partially perfect detector named xP. The necessary properties and algorithms to implement P and xP are presented in the paper, as well as the related correctness proofs.*

Resumo. *No presente artigo mostramos que um detector perfeito de defeitos P(que suspeita todos os processos que falharam se e somente os mesmos falharam) pode ser implementado num sistema mais fraco que o sistema distribuído síncrono (contrariando uma crença estabelecida). Nesse sentido, introduzimos o sistema síncrono particionado (Spa) que é estritamente mais fraco que o sistema síncrono (em Spa não é sempre possível implementar ações síncronas globais como sincronização interna de relógios). Através da propriedade que definimos como sincronia particionada forte, mostramos como implementar P em Spa. Melhor ainda, mostramos que mesmo que Sincronia Particionada Forte não possa ser garantida, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância da falhas, através de um detector parcialmente perfeito, denominado por nós de xP. As propriedades e algoritmos necessários para implementar P e xP são introduzidos no artigo, assim como as provas de correção relacionadas.*

1. Introdução

A capacidade de resolver certos problemas de tolerância a falhas em sistemas distribuídos está intimamente ligada à existência de um modelo de sistema adequado. Nesse sentido, há algumas décadas, pesquisadores vêm propondo uma variedade de modelos de resolução de problemas, onde os modelos assíncronos (ou livres de tempo) e síncronos (baseados no tempo) têm dominado o centro das atenções, por serem considerados modelos extremos em termos de

resolução de problemas de tolerância a falhas. Por exemplo, o problema de difusão confiável - na presença de canais confiáveis e falhas silenciosas de processos - é solúvel em ambos os modelos [Lynch 1996, Mullender 1993]. Contudo, o problema de consenso distribuído é solúvel no modelo síncrono, mas não no modelo assíncrono [Fisher et al. 1985]. A origem da não solubilidade do consenso em modelos de sistemas assíncronos está relacionada à dificuldade em distinguir-se, nesse modelo, entre uma falha de um processo p (exemplo, de parada) e o atraso no recebimento de uma mensagem de p. Essa crença, levou pesquisadores a acreditarem que o poder de um modelo de sistema assíncrono equipado com um detector perfeito de falhas - que identifica todos os processos que falharam, se e somente se os mesmos falharam - se igualaria ao poder de um modelo de sistema síncrono.

Essa crença foi adequadamente contestada por Charron-Bost, Guerraoui e Schiper [Charron-Bost et al. 2000], que mostraram que o modelo de sistema síncrono (denominado por eles Ss) é estritamente mais forte que o modelo de sistema assíncrono equipado com um detector perfeito (denominado por eles como Sp). No referido artigo, eles demonstraram que alguns problemas solúveis em Ss não o são em Sp, mesmo quando tais problemas não envolvam especificações temporais (nesses casos, por definição, impossíveis de serem resolvidos nos modelos assíncronos). Alem disso, eles provaram que a solução do problema de consenso distribuído uniforme - um problema fundamental de tolerância a falhas - é feita com mais eficiência em Ss, quando comparado a Sp, usando como métrica o grau de latência, que é baseada no conceito de número de rodadas [Schiper 1997].

Por outro lado, ao longo dos anos, a implementação de detectores perfeitos, que tornaria o problema de consenso solúvel, tem sido considerada impossível em sistemas assíncronos. Esse fato levou a Chandra e Toueg [Chandra and Toueg 1996] a proporem um modelo de computabilidade para tolerância a falhas em sistemas assíncronos baseado em propriedades axiomáticas de várias classes de detectores de defeitos, identificando aquelas para as quais existem soluções com os menores requisitos de implementação (no caso, a classe $\diamond W$). A partir daí, vários autores passaram a propor soluções de implementação para essas classes [Chen et al. 2000, Macêdo 2000, Nunes and Jansch-Pôrto 2004, Falai and Bondavalli 2005, Lima and Macêdo 2005] cuja viabilidade de funcionamento dependeria de certas condições de estabilidade do ambiente de execução assíncrono (exemplo, a propriedade Global-Stabilization-Time [Dwork et al. 1988]). A classe mais forte apresentada por Chandra e Toueg foi justamente a do detector perfeito, denominado por eles de P e possuindo as propriedades axiomáticas de *strong completeness* (P suspeita todos os processos que falharam) e *strong accuracy* (P somente suspeitam processos que realmente falharam). No mesmo artigo, Chandra e Toueg argumentam que seria possível implementar P no modelo síncrono de sistemas distribuídos, utilizando-se de *timeouts*.

Baseados na idéia de que *timeouts* ou hipóteses temporais podem ser utilizados nos sistemas Ss para implementar detectores de falhas perfeitos, Charron-Bost, Guerraoui and Schiper prosseguem em seu artigo [Charron-Bost et al. 2000] para concluir que a comparação dos dois modelos se daria via a identificação das propriedades do modelo Ss que se perderiam na transformação axiomática proposta por Chandra e Toueg. Apesar de elucidativas, tais discussões relativas ao artigo de Charron-Bost, Guerraoui e Schiper incorrem em uma imprecisão: a crença subjacente de que o ambiente de implementação de ambos os modelos, Ss e Sp, seria necessariamente o sistema síncrono[1] onde existem limites temporais

[1]Neste artigo, *sistema* refere-se ao ambiente de execução e *modelo* às propriedades que se pode inferir de

superiores conhecidos para processamento de informação e entrega de mensagens nos canais de comunicação (observem que tal comportamento casa perfeitamente com o que se assume para modelo síncrono de sistema distribuído). Tal crença subjacente de fato justificaria a comparação vantajosa do modelo Ss em relação ao modelo Sp, como colocada no artigo. Tal imprecisão, de outro lado, reforça outra crença: a de que detectores perfeitos não seriam implementáveis em sistemas mais fracos que os síncronos.

No presente artigo nós desmontamos essa última crença mostrando que P pode ser implementado num sistema mais fraco que Ss, denominado por nós de Spa (síncrono particionado), o que torna injusta a comparação direta do poder de resolução dos dois modelos Ss e Sp, uma vez que Ss requer mais recursos para sua implementação comparada ao modelo Sp (quando implementado sobre Spa). Vale salientar, como veremos adiante, que o modelo Sp implementado sobre Spa não requer a existência de um *wormhole* síncrono [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona [Gorender and Macêdo 2002], onde seria possível implementar ações síncronas globais em todos os processos, como sincronização interna de relógios. No sistema Spa, que propomos, componentes do ambiente distribuído necessitam ser síncronos, mas os mesmos não precisam estar conectados entre si via canais síncronos, o que torna impossível a execução de ações síncronas distribuídas em todos os processos do sistema. Mesmo assim, mostramos ser possível implementar P em Spa. Melhor ainda, mostramos que mesmo que parte dos processos não esteja em qualquer das componentes síncronas, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas, através de um detector parcialmente perfeito, denominado por nós de xP.

O resto deste artigo se estrutura da seguinte forma. Na Seção a seguir são apresentados trabalhos relacionados. Na Seção 3 é apresentado o modelo Spa, suas características e propriedades. As Subseções 3.1, 3.2 e 3.3 descrevem uma implementação de um detector de defeitos P no sistema Spa, e apresentam suas propriedades e provas formais, e na Subseção 3.4 é apresentada a classe de detectores de defeitos xP, com suas propriedades, além de uma implementação e provas formais. A Seção 4 apresenta as conclusões ao trabalho.

2. Trabalhos Relacionados: Modelos de Sistemas Distribuídos

O desenvolvimento de serviços tolerantes a falhas, como consenso distribuído, depende sobremaneira da existência de tempos conhecidos máximos para escalonamento de processos e transmissão de mensagens (isto é, canais e processos isócronos). Tais limites superiores temporais - para todos os processos e canais de comunicação - somente podem ser continuamente garantidos nos sistemas síncronos. Portanto, problemas como o consenso distribuído são solúveis em sistemas distribuídos síncronos [Lamport et al. 1982]. De outro lado, serviços de consenso distribuído podem também ser garantidos em ambientes assíncronos ou parcialmente síncronos desde que o "comportamento síncrono"se estabeleça durante períodos de tempo suficientemente longos para a execução do consenso [Dwork et al. 1988]. Assim sendo, tais ambientes considerados parcialmente síncronos podem apresentar comportamento alternado, entre síncrono com garantias temporais e assíncrono com nenhuma garantia temporal. Conseqüentemente, podemos considerar tais sistemas como híbridos na dimensão temporal. Por exemplo, o sistema assíncrono temporizado depende de períodos de estabilidade síncrona suficientemente longos para prover serviços, e o sistema pode alternar

determinado *sistema*

entre estável e não estável (i.e., síncrono e assíncrono) [Cristian and Fetzer 1999]. A hipótese GST - Global Stabilization Time - é necessária para garantir que os protocolos de consenso baseados no detector de defeitos $\diamond\mathcal{S}$ trabalhem de forma adequada [Chandra and Toueg 1996] (i.e., para trabalhar de forma correta o sistema deve apresentar comportamento "síncrono"a partir de algum momento).

Há outros modelos que consideram a natureza híbrida do ambiente não somente na dimensão temporal (como os anteriormente referenciados), mas também na dimensão espacial, onde o sistema é particionado em componentes síncronas e assíncronas que funcionam de forma concomitante. Logo, nós consideramos esses modelos como híbridos na dimensão espacial. Esse é o caso do modelo TCB que se utiliza de um wormhole síncrono (i.e., uma componente síncrona que inclui todos os processos do sistema) para implementar serviços tolerantes a falhas [Veríssimo and Casimiro 2002], e tal caráter híbrido persiste durante toda a vida do sistema.

No caso dos sistemas síncronos particionados (Spa) assumimos que o sistema computacional distribuído é capaz de prover qualidades de serviços fim-a-fim distintas tanto para execução dos processos quanto para os canais de comunicação. Nos sistemas operacionais, tais características podem ser implementadas através de sistemas operacionais de tempo real capazes de lidar ao mesmo tempo com tarefas críticas (com deadlines garantidos) e não críticas (*best-effort*). Nas redes de computadores, o mesmo efeito pode ser conseguido quando determinado nó está ligado a duas redes de QoS distintas (por exemplo, de um lado uma rede de tempo real e do outro uma rede TCP/IP). Também se pode conseguir tal característica através de multiplexação, de modo que um determinado canal físico possa ser capaz de transmitir fluxos de comunicação relacionados a diversos canais virtuais com qualidades de serviço (QoS) distintas (por exemplo, um canal isócrono e dois não isócronos implementados no mesmo canal físico). Tal efeito pode ser implementado através de redes de tempo reais híbridas ou arquiteturas de qualidade de serviços (e.g., diffserv [Aurrecoechea et al. 1998]).

Dado que canais e processos podem ser isócronos e não isócronos no mesmo sistema e ao mesmo tempo, consideramos nosso sistema híbrido no espaço. Não obstante, os canais não isócronos podem apresentar comportamento síncrono em certos períodos de estabilidade do sistema. Logo, nosso sistema Spa se diferencia dos demais modelos apresentados acima não somente porque considera o caráter híbrido nas dimensões temporal e espacial (nesse sentido, similar ao TCB [Veríssimo and Casimiro 2002]), mas também porque tal caráter híbrido pode mudar com o tempo, com períodos com poucas ou mesmo nenhuma partição síncrona (nesse caso, distinto do TCB). Em outras palavras, o sistema pode se tornar totalmente assíncrono ou não ter todos os seus processos em componentes síncronas, dado que em algumas circunstâncias a qualidade de serviço de canais ou processos pode degradar (devido a falhas, por exemplo).

Resumidamente, modelos de sincronia parcial como o GST e o assíncrono temporizado não consideram o caráter híbrido espacial. O modelo TCB é híbrido em espaço e tempo, mas sua natureza híbrida permanece durante a vida do sistema. Nosso modelo Spa considera um ambiente computacional subjacente que pode também ser híbrido no tempo e espaço. Contudo, a parte síncrona pode se deteriorar com o tempo. O preço que se paga para dispor de tal flexibilidade no modelo é a necessidade de implementação de mecanismos de gerenciamento de qualidade de serviço (provisão e monitoria) ambos nos sistemas operacionais e nas redes, capazes de acompanhar continuamente o estado de QoS dos processos e canais. Tal mecanismo é, em certo sentido, semelhante ao serviço *fail-awareness*

implementado no sistema assíncrono temporizado, onde a ocorrência de certo número de falhas de desempenho ativa um sinal indicando que o sistema não mais pode satisfazer ao comportamento "síncrono"[Fetzer and Cristian 1996]. A principal diferença reside no fato de que no sistema assíncrono temporizado a sinalização não é baseada na gestão de QoS e sim na violação de limites de tempo para envio de recebimento de mensagens (*round-trip times*).

3. O Modelo Síncrono Particionado Spa

Modelo de sistema e hipóteses assumidas Um sistema é formado por um conjunto Π de processos $p_1, p_2, ..., p_n$, e um conjunto χ de canais de comunicação $c_1, c_2, ..., c_m$. Cada canal c_i de χ liga somente dois processos distintos de Π e cada processo de Π está ligado a todos os outros processos de Π. Portanto, nosso sistema forma um grafo simples completo não direcionado $DS(\Pi, \chi)$ com $(n \times (n-1))/2$ arestas. Tanto processos em Π quanto canais em χ podem ser isócronos ou não isócronos. Um dado processo é isócrono se existe valor máximo conhecido (digamos, ϕ) para a execução de passos de computação em p_i. Da mesma forma, um canal c_i é isócrono se uma mensagem é transmitida em c_i dentro de um limite de tempo limitado e conhecido, digamos δ. Do contrário, processos e canais são não isócronos. δ e ϕ são parâmetros do sistema de execução e garantidos por mecanismos operacionais de sistemas e redes de tempo real. Assumimos também que processos e canais podem mudar sua qualidade de serviço de forma que possam alternar entre isócronos e não isócronos.

Assumimos que canais são confiáveis, ou seja, não alteram ou perdem mensagens enviadas. Também assumimos que processos falham de forma silenciosa, ou seja, apenas param prematuramente de funcionar, sem produzir quaisquer efeitos adicionais. Processos considerados corretos, não falham durante um intervalo de tempo de interesse (por exemplo, durante a execução da aplicação).

Dados $\Pi' \subseteq \Pi$, $\Pi' \neq \emptyset$ e $\chi' \subseteq \chi$, $\chi' \neq \emptyset$, um componente ou sub-grafo $C(\Pi', \chi') \subseteq DS(\Pi, \chi)$ conexo é síncrono se $\forall p_i \in \Pi'$ e $\forall c_j \in \chi'$, p_i e c_j são isócronos. Utilizamos a notação Cs para denotar um componente C síncrona e Ca para um componente não síncrono.

A Figura 1 ilustra a representação no grafo DS para duas redes locais com propriedades síncronas (processos e canais de comunicação), interligadas por um canal assíncrono (por exemplo, um canal TCP/IP). No exemplo, cada nó da rede (númerados de 1 a 6) hospeda apenas um processo. Observa-se em DS, por exemplo, que os sub-grafos $C1(\{1,2,3\}, \{(1,2),(1,3),(2,3)\})$, $C2(\{4,5\}, \{(4,5)\})$ e $C3(\{5,6\}, \{(5,6)\})$ formam componentes síncronos, enquanto que $C4(\{3,4,5\}, \{(3,4),(4,5),(3,5)\})$ forma um componente não síncrono.

No sistema distribuído Spa, assumimos inicialmente a seguinte propriedade necessária para a implementação de P.

- *Sincronia Particionada Forte*: $\forall p_i \in \Pi$, $\exists$ um $Cs \subset DS$ tal que $p_i \in Cs$.

Na exemplo da Figura 1, *Sincronia Particionada Forte* se verifica pois $C1$, $C2$, e $C3$ incluem todos os processos em DS. A hipótese de *Sincronia Particionada Forte* será relaxada mais tarde no texto quando introduzirmos o detector xP, onde processos podem pertencer a componentes não síncronos (Ca). No entanto, é importante observar que para um dado sistema distribuído, *Sincronia Particionada Forte* não necessariamente implica que exista um componente Cs onde $\Pi = \Pi'$, o que caracterizaria um *wormhole* síncrono na terminologia

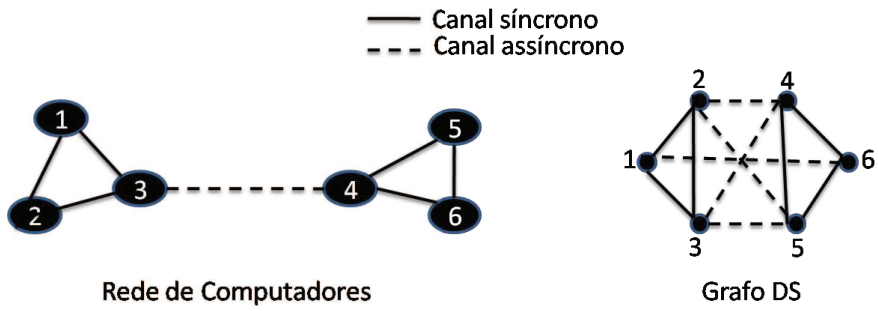

Figura 1. Exemplo de uma Rede e sua representação no Grafo DS

em [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona (no exemplo da Figura 1 não há qualquer sub-grafo Cs que inclua todos os vértices de DS).

Assumimos, também, que seja possível, a partir do ambiente de execução, identificar localmente qual a qualidade de serviço de um dado processo ou canal de comunicação.

3.1. Detector Perfeito em Spa

Descrevemos a seguir a implementação de um detector de defeitos que satisfaz as propriedades *strong completeness* e *strong accuracy*, para detectores de defeitos da classe P, quando executando em Spa com a propriedade *Sincronia Particionada Forte*. Esta implementação foi inicialmente apresentada em [Macêdo et al. 2005, Gorender et al. 2007].

Informações com relação ao estado dos processos, e à possibilidade de se efetuar detecções confiáveis, é fornecida ao detector de defeitos por dois mecanismos adicionais, chamados Detector de Estados dos Processos e QoS Provider [Gorender 2005, Macêdo et al. 2005]. A partir das informações providas por estes mecanismos o detector de defeitos verifica as condições mínimas para assumir um comportamento de um detector Perfeito.

O QoS Provider (QoSP) funciona como uma interface entre o ambiente de execução e os Detectores de Defeitos e de Estados, e basicamente informa a estes detectores quais canais de comunicação são isócronos e quais não são. Esta informação é fornecida através da execução da função $QoS(p_i, p_j)$, onde p_i, p_j identifica o canal de comunicação ligando os processos p_i e p_j. A função identifica o canal como T, quando o canal é isócrono, e U, quando o canal não é isócrono. Uma descrição detalhada do QoSP pode ser encontrada em [Macêdo et al. 2005].

O Detector de Estados dos Processos executa distribuído em módulos, um para cada processo do sistema, e detecta o estado de cada processo como sendo *live* ou *uncertain*, inserindo a identificação destes processos nos conjuntos $live_i$ e $uncertain_i$, fornecidos a cada processo p_i do sistema. Processos que fazem parte de uma Componente Síncrona (Cs), são processos isócronos que possuem ao menos um canal de comunicação isócrono, e são identificados no estado *live*. Este mecanismo utiliza informações do QoS Provider, sobre os canais de comunicação. O Detector de Estados satisfaz as propriedades *strong completeness* (toda transição no estado de um processo entre $live$ e $uncertain$ será detectada por todos os processos corretos do sistema, em um tempo finito) e *strong accuracy* (nenhuma detecção de uma transição de estado entre $live$ e $uncertain$ para um processo, efetuada por qualquer processo correto do sistema, será incorreta.). A descrição detalhada deste mecanismo, além

das provas das propriedades, pode ser encontrada em [Gorender 2005, Macêdo et al. 2005, Gorender et al. 2007].

Como em Spa com a propriedade **Sincronia Particionada Forte** todos os processos pertencem a alguma Componente Síncrona, temos que o Detector de Estados dos Processos identifica todos os processos como possuindo o estado *live*: $\forall p_i \in \Pi, live_i = \Pi$.

3.2. Algoritmo do Detector de Defeitos

O detector de defeitos é executado em módulos, um para cada processo do sistema. Os módulos do detector de defeitos enviam periodicamente mensagens *are-you-alive* para todos os demais módulos do detector de defeitos, solicitando destes, em resposta, o envio de mensagens do tipo *I-am-alive*, através dos canais de comunicação, segundo o modelo *pull*. O tempo esperado para as respostas dos processos (*timeout*), em cada canal de comunicação $c_{x/y}$, é definido pela função $Delay(p_x, p_y)$ do QoS Provider. No caso de canais de comunicação isócronos a função $Delay(p_x, p_y)$ informa limites de tempo conhecidos para a transferência de mensagens. Portanto, para estes canais, o módulo do detector de defeitos provê notificações de falhas de processos que de fato tenham falhado. Para canais não isócronos, os módulos do detector de defeitos produzem suspeitas de falhas, ou seja, detecções não confiáveis de falhas de processos. Detecções confiáveis são registradas com a inserção da identificação do processo detectado no conjunto $down$, e sua retirada do conjunto $live$.

O algoritmo de detecção, apresentado na Figura 2, monitora os processos do sistema. Este procedimento é composto de 5 tarefas.

A primeira tarefa do algoritmo de detecção ($task1$) efetua o envio de mensagens $are - you - alive(p_i)$ para todos os processos do sistema (linha 3). Estas mensagens são enviadas periodicamente, a cada intervalo de monitoramento, indicado pela variável $MonitoringInterval$. Ao enviar cada mensagem *are-you-alive*(p_i), a tarefa calcula um *timeout* para o recebimento da mensagem *I-am-alive*(p_j) em resposta, para cada processo p_j do sistema. Os valores de *timeout* são armazenados no *array* $timeout_i$, indexado pela identificação do processo monitorado p_j. Estes tempos são calculados somando o tempo atual, obtido do relógio local, com o RTT calculado para o canal $c_{i/j}$, obtido através da função $Delay(p_i, p_j)$, do QoS Provider, com um valor de erro ($timeout_i[p_j] = CT_i() + Delay(p_i, p_j) + \alpha$ na linha 2).

A tarefa seguinte deste algoritmo ($task2$) efetua o monitoramento dos processos do sistema. Sempre que o $timeout$ para um processo não notificado (processo que não pertença ao conjunto $down_i$) seja alcançado (tempo obtido do relógio local for superior ao do *timeout*) sem o recebimento de uma mensagem $I-am-alive(p_j)$ do processo monitorado, este processo será analisado. Se o processo monitorado pertencer ao conjunto $live_i$ e o canal de comunicação utilizado para a monitoração for isócrono (linhas 5 e 6), o processo monitorado (p_j) tem sua falha notificada (linhas 7 a 9). Para verificar se o canal $c_{i/j}$ é de fato isócrono é utilizado o grafo DS e a função $QoS(p_i, p_j)$ do QoS Provider. A Função $QoS(p_i, p_j)$ é utilizada para se ter certeza de que um canal de comunicação identificado no grafo DS como isócrono permanece isócrono, uma vez que os canais podem ter a sua *QoS* alterada durante a execução. Se o canal de comunicação utilizado para a monitoração não for isócrono, nada é realizado (linha 10). Se o processo monitorado pertencer ao conjunto $uncertain_i$ (linha 12), terá sua identificação inserida no conjunto $suspected_i$ (linha 13).

```
     Task T1: every monitoringInterval do
(1)          for_each p_j, p_j ≠ p_i do
(2)                   timeout_i[p_j] ← CT_i() + Delay(p_i, p_j) + α;
(3)                   send "are-you-alive?" message to p_j
(4)          end_do

     Task T2: when ∃p_j : (p_j ∉ down_i) ∧ (CT_i() > timeout_i[p_j])) do
(5)          if ((p_j ∈ live_i)
(6)            then if ((DS_i[p_i, p_j] = timely) ∧ (QoS(p_i, p_j) = timely))
(7)                   then down_i ← down_i ∪ {p_j};
(8)                        live_i ← live_i − {p_j};
(9)                        send notification (p_i, p_j) to every p_x such that p_x ≠ p_i ∧ p_x ≠ p_j
(10)                  else do nothing (wait for a remote notification)
(11)                 end_if
(12)           else if ((p_j ∈ uncertain_i) ∧ (p_j ≠ suspected_i))
(13)                  then suspected_i ← suspected_i ∪ {p_j} end_if
(14)         end_if

     Task T3: when "I-am-alive" is received from p_j do
(15)         timeout_i[p_j] ← ∞; /* cancels timeout */
(16)         if (p_j ∈ suspected_i) then suspected_i ← suspected_i − {p_j} end_if
     Task T4: when notification(p_x, p_j) is received do
(17)         if p_j ∉ down_i then down_i ← down_i ∪ {p_j};
(18)                              live_i ← live_i − {p_j}
(19)         end_if

     Task T5: when "are-you-alive?" is received from p_j do
(20)         send "I-am-alive" to p_j
```

Figura 2. Algoritmo do módulo do detector de defeitos de (p_i)

A terceira tarefa (*task 3*) é executada sempre que uma mensagem $I - am - alive(p_j)$ é recebida pelo detector de defeitos. Se o processo que enviou a mensagem for um processo suspeito ($p_j \in suspected_i$), ele será retirado do conjunto $suspected_i$ (linha 16). Esta situação caracteriza uma suspeita errônea de uma falha.

A tarefa 4 (*task* 4) é executada quando uma mensagem $notification(p_x, p_j)$ for recebida pelo detector de defeitos, indicando a falha de um processo (p_x notifica a falha de p_j). Ao receber esta mensagem, o módulo do detector de defeitos insere a identificação do processo notificado (p_j) no conjunto $down_i$, caso ainda não seja um elemento deste conjunto (linha 17). A identificação do processo notificado é retirada do conjunto $live_i$ (linha 18).

A última tarefa (*task* 5) é executada quando uma mensagem *are-you-alive*(x, p_j) enviada pelo processo p_j é recebida pelo detector de defeitos. É enviada em resposta uma mensagem *I-am-alive*(x, p_i) para o processo p_j.

3.3. Propriedades e provas para o detector perfeito

Apresentamos a seguir as provas das propriedades *strong completeness* e *strong accuracy* do detector de defeitos. Estas provas levam em consideração que o detector executa em Spa, e que a propriedade *Sincronia Particionada Forte* é satisfeita. Também assumimos a existência do QoS Provider e do Detector de Estados dos Processos, e que este Detector de Estados satisfaz

as propriedades *strong completeness* e *strong accuracy*, para detector de estado.

Teorema 3.1 *O detector de defeitos apresenta a Propriedade Strong Completeness - Todas as falhas de processos serão detectadas permanentemente por todos os processos corretos em um tempo finito.*

Prova

Esta prova é construída por contradição, supondo existir um processo, p_x, que falha no tempo t, e cuja falha não será detectada por algum processo correto. Conseqüentemente, supõem-se a existência de um processo correto, p_y, que não irá detectar a falha do processo p_x. É assumido que $t' > t$ é o momento em que o processo p_y já recebeu a última mensagem $I - am - alive(p_x)$ enviada por p_x (após falhar de forma silenciosa p_x interrompe o envio de mensagens), e no qual o valor do relógio local de p_y se torna superior ao *timeout* calculado para a recepção da próxima mensagem $I - am - alive(p_x)$, ou seja, o momento em que p_y detecta a ausência de uma mensagem $I - am - alive(p_x)$.

No tempo t', p_y inicia a execução da tarefa 2 do algoritmo de detecção, uma vez que a condição $CT_y() > timeout_y[p_x]$ é satisfeita, e a partir deste tempo p_y não receberá mensagens $I - am - alive(p_x)$ do processo p_x. De acordo com a propriedade *Sincronia Particionada Forte*, temos que todos os processos do sistema, incluindo p_x, pertencem a alguma Componente Síncrona, e possuem canais de comunicação isócronos, tendo a sua identificação previamente inserida no conjunto $live_y$. Portanto, o if da linha 5 é satisfeito. Existem duas possibilidades de execução dependendo de o canal $c_{x/y}$ ser ou não isócrono.

1. $c_{x/y}$ é isócrono no tempo t'.
   - Ao executar o algoritmo de detecção no tempo t', o comando if da linha 6 é satisfeito, e a identificação de p_x é transferida do conjunto $live_y$ para o conjunto $down_y$ (linhas 7 e 8). Uma mensagem $notification(p_y, p_x)$ é enviada para todos os processos. Conseqüentemente, a partir do tempo t', p_y detecta a falha de p_x permanentemente.
2. $c_{x/y}$ não é isócrono no momento t'.
   - Como p_x possui ao menos um canal isócrono e o canal $c_{x/y}$ não é isócrono, existe um canal de comunicação $c_{x/z}$ que é isócrono, ligando p_x ao processo p_z. O processo p_z detecta a falha do processo p_x ao executar a tarefa 2 do algoritmo de detecção, quando a condição $CT_z() > timeout_z[p_x]$ é satisfeita (como descrito no item anterior desta prova). p_z transfere a identificação de p_x do conjunto $live_z$ para o conjunto $down_z$, e envia mensagens $notification(p_z, p_x)$ para todos os processos do sistema. Como os canais de comunicação são confiáveis, p_y recebe a mensagem $notification(p_z, p_x)$, informando a falha de p_x.

Como mostrado em todas as situações possíveis, o processo p_y detecta a falha do processo p_x permanentemente, em um tempo finito de sua execução, o que contradiz a suposição inicial da prova, provando o teorema. ∎

Teorema 3.2 *O detector de defeitos apresenta a propriedade Strong Accuracy - Nenhum processo será suspeito erroneamente por nenhum outro processo.*

Prova

Esta prova é desenvolvida por contradição, assumindo a existência de um processo correto p_x, e de um processo correto p_y, que irá continuamente suspeitar de p_x.

De acordo com a propriedade *Sincronia Particionada Forte*, todos os processos, inclusive p_x, fazem parte de alguma Componente Síncrona, tendo portanto a sua identificação inserida, previamente, nos conjuntos $live$ de todos os processos, inclusive $live_y$. Como, segundo a Propriedade *Strong Accuracy* do detector de estados dos processos, um processo não terá o seu estado erroneamente identificado, temos que a identificação de p_x permanece no conjunto $live_y$.

Temos as seguintes possibilidades de detecção:

1. Suspeita (linha 13 do algoritmo de detecção) - Como a condição $p_x \in uncertain_y$ não é satisfeita, a identificação do processo p_x não poderá ser inserida no conjunto $suspected_y$ (não satisfaz a condição da linha 12 do algoritmo de detecção), e o processo p_x não poderá ser erroneamente suspeito.
2. Notificação através do canal $c_{x/y}$ (linhas 6 a 9 do algoritmo de detecção) - Se $c_{x/y}$ for isócrono, todas as mensagens transferidas através deste canal serão recebidas dentro dos limites de tempo determinados. Neste caso a condição da linha 6 não será satisfeita, e p_x não poderá ser erroneamente detectado.
3. Notificação através da recepção da mensagem $notification(p_z, p_x)$ (linhas 17 a 19 do algoritmo de detecção) - Para que p_y receba uma mensagem *notification*(p_z, p_x), esta mensagem deve ter sido enviada por um processo p_z. Para enviar esta mensagem, p_z teria de executar as linhas 6 a 9 do algoritmo de detecção, e o canal $c_{z/x}$ tem de ser isócrono. Neste caso, como as mensagens transferidas através do canal $c_{z/x}$ são entregues dentro dos *timeouts* determinados (canais de comunicação isócronos fornecem comunicação com limites de tempo conhecidos e garantidos), e como p_x não falhou e continua a enviar mensagens, obtemos para p_z a situação do caso anterior. Neste caso, a condição da tarefa 2 do algoritmo de detecção, executado por p_z, não é satisfeita, e p_z não envia mensagens $notification(p_z, p_x)$ para nenhum processo. Como p_y não recebe nenhuma mensagem $notification(p_z, p_x)$, não irá executar as linhas 17 a 19 do algoritmo de detecção. p_x não será erroneamente detectado.

p_x não é detectado por p_y em nenhuma situação possível, o que contradiz a suposição inicial da prova. ∎

3.4. Detector Parcialmente Perfeito

Definimos um detector de defeitos como sendo da classe xP quando satisfizer a propriedade *strong completeness* e a nova propriedade *partially strong accuracy*. Esta propriedade determina que: nenhum processo que pertença a uma Componente Síncrona será suspeito erroneamente por nenhum outro processo. Este detector de defeitos é implementado pelo mesmo algoritmo descrito na seção anterior, ao executar sobre Spa, sendo satisfeita a propriedade *Sincronia Particionada Fraca*.

Sincronia Particionada Fraca: O conjunto dos processos que pertencem a componentes síncronos é menor que Π. Mais formalmente, seja $\Sigma = Cs_1 \cup Cs_2 \cup ... \cup Cs_k$, $k \geq 1$, o conjunto união de todos os possíveis componentes síncronos em DS. Então, $\Pi - \Sigma \neq \emptyset$. Ou seja, $\exists p_i \in \Pi$, tal que $p_i \notin \Sigma$ e $2 \leq |\Sigma| < |\Pi|$.

Existem, portanto, processos que pertencem a Componentes Síncronas, mas nem todo processo em Π pertence a alguma Componente Síncrona. Conseqüentemente, temos que: $\forall p_i \in \Pi, live_i \neq \oslash \wedge live_i \neq \Pi$, ou seja, existem processos identificados no estado $live$, e existem processos identificados no estado $uncertain$.

A propriedade *strong completeness* é provada de forma similar ao apresentado na seção anterior. A seguir apresentamos a prova formal da propriedade *partially strong accuracy*, a partir de Spa e da propriedade Sincronia Particionada Fraca, além das informações fornecidas pelo QoS Provider, e das propriedades *strong completeness* e *strong accuracy* do Detector de Estados dos Processos.

Teorema 3.3 *O detector de defeitos apresenta a propriedade partially Strong Accuracy - Nenhum processo que pertença a uma Componente Síncrona será suspeito erroneamente por nenhum outro processo.*

Prova

Esta prova é desenvolvida por contradição, assumindo a existência de um processo correto p_x, que pertence a uma Componente Síncrona, e de um processo correto p_y, que irá continuamente suspeitar de p_x.

Como o processo p_x pertence a uma Componente Síncrona, ele possui ao menos um canal de comunicação isócrono, interligando p_x a algum outro processo. Segundo a propriedade *Strong Completeness* do detector de estado, todo processo que possui ao menos um canal de comunicação isócrono é identificado no estado $live$, e terá sua identificação inserida no conjunto $live_i$ para todo processo p_i de Π, portanto, $p_x \in live_y$. Como, segundo a Propriedade *Strong Accuracy* deste mesmo detector, um processo não terá o seu estado erroneamente identificado, temos que a identificação de p_x permanece no conjunto $live_y$.

Temos as seguintes possibilidades de detecção:

1. Suspeita (linha 13 do algoritmo de detecção) - Como a condição $p_x \in uncertain_y$ não é satisfeita, a identificação do processo p_x não poderá ser inserida no conjunto $suspected_y$ (não satisfaz a condição da linha 12 do algoritmo de detecção), e o processo p_x não poderá ser erroneamente suspeito.
2. Notificação através do canal $c_{x/y}$ (linhas 7 a 9 do algoritmo de detecção) - Se o canal $c_{x/y}$ for isócrono, todas as mensagens transferidas através deste canal serão recebidas dentro dos limites de tempo determinados. Neste caso a condição da tarefa 2 não será satisfeita, e p_x não poderá ser erroneamente detectado.
3. Notificação através da recepção da mensagem $notification(p_z, p_x)$ (linhas 17 a 19 do algoritmo de detecção) - Para que p_y receba uma mensagem *notification*(p_z, p_x), esta mensagem deve ter sido enviada por um processo p_z. Para enviar esta mensagem, p_z teria de executar as linhas 6 a 9 do algoritmo de detecção, e o canal $c_{z/x}$ tem de ser isócrono. Neste caso, como as mensagens transferidas através do canal $c_{z/x}$ são entregues dentro dos *timeouts* determinados (definição dos canais isócronos), e como p_x não falhou e continua a enviar mensagens, obtemos para p_z a situação do caso anterior. Neste caso, a condição da tarefa 2 do algoritmo de detecção, executado por p_z, não é satisfeita, e p_z não envia mensagens $notification(p_z, p_x)$ para nenhum processo. Como p_y não recebe nenhuma mensagem $notification(p_z, p_x)$, não irá executar as linhas 17 a 19 do algoritmo de detecção. p_x não será erroneamente detectado.

p_x não é detectado por p_y em nenhuma situação possível, o que contradiz a suposição inicial da prova. ∎

4. Conclusões

A incapacidade em detectar defeitos de forma precisa está no cerne da impossibilidade de resolver problemas de consenso em sistemas assíncronos. Portanto, acreditava-se que um sistema assíncrono equipado com detector de faltas perfeito (P) se equivaleria a um sistema síncrono (onde se pode facilmente implementar detectores perfeitos). No entanto, existia uma crença estabelecida de que detectores perfeitos de defeitos somente seriam implementáveis em sistemas síncronos. No presente artigo desmontamos essa crença, mostrando como um detector perfeito de defeitos pode ser implementado num sistema mais fraco que o sistema distribuído síncrono. Para isso, introduzimos o sistema síncrono particionado (Spa) que é estritamente mais fraco que o sistema síncrono, haja vista que em um sistema Spa processos podem estar conectados por canais sem garantias temporais - tornando impossível a implementação de ações síncronas globais como sincronização interna de relógios. Através da propriedade que definimos como *Sincronia Particionada Forte*, mostramos como implementar P em Spa. Melhor ainda, mostramos que mesmo que *Sincronia Particionada Forte* não possa ser garantida, podemos ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas, através de um detector parcialmente perfeito, denominado por nós de xP. Um algoritmo de consenso adaptativo que tira proveito desse tipo de detector foi apresentado em [Gorender et al. 2007]. Sistemas Spa se adéquam em especial a configurações de aglomerados (*clusters*) interligados por redes de longa distância (como a Internet). Nessas circunstâncias, processos de um *cluster* formam um componente síncrono e a redistribuição de carga, por exemplo, devido a falha de processos, pode ser feita de forma mais eficiente uma vez que processos são detectados falhos de forma não ambígua. As propriedades e algoritmos necessários para implementar P e xP foram introduzidos no artigo, assim como as provas de correção relacionadas.

Referências

Aurrecoechea, C., Campbell, A. T., and Hauw, L. (1998). A survey of qos architectures. *ACM Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Charron-Bost, B., Guerraoui, R., and Schiper, A. (2000). Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the International Conference on Dependable System and Networks*, pages 523–532.

Chen, W., Toueg, S., and Aguilera, M. K. (2000). On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, pages 561–580.

Cristian, F. and Fetzer, C. (1999). The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

Falai, L. and Bondavalli, A. (2005). Experimental evaluation of the qos of failure detectors on wide area network. In *Proceedings of the International Conference on Dependable Systems and Networks, DSN*, pages 624–633.

Fetzer, C. and Cristian, F. (1996). Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*, pages 314–321.

Fisher, M. J., Lynch, N., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Gorender, S. (2005). Um modelo híbrido e adaptativo para sistemas distribuídos tolerantes a falhas. Tese de Doutorado aprovada pelo CIN/UFPE.

Gorender, S., Macêdo, R., and Raynal, M. (2007]). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31.

Gorender, S. and Macêdo, R. J. d. A. (2002). Um modelo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do Simpósio Brasileiro de Redes de Computadores, SBRC 2002*, pages 277–292.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lima, F. R. L. and Macêdo, R. J. d. A. (2005). Adapting failure detectors to communication network load fluctuations using snmp and artificial neural nets. In *Second Latin-American Symposium on Dependable Computing (LADC2005), Lecture Notes in Computer Science*, pages 191 – 205.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc.

Macêdo, R. (2000). Failure detection in asynchronous distributed systems. In *Proc. of II Workshop on Tests and Fault-Tolerance (II WFT 2000)*, pages 76–81.

Macêdo, R., Gorender, S., and Cunha, P. (2005). The implementation of a qos-based adaptive distributed system model for fault tolerance. In *Anais do Simposio Brasileiro de Redes de Computadores (SBRC 2005)*, pages 827–840.

Mullender, S. (1993). *Distributed Systems*. Addison-Wesley Pub Co.

Nunes, R. C. and Jansch-Pôrto, I. (2004). Qos of time-out based self-tuned failure detector: the effects of its communication delay predictor and its safety margin. In *IEEE Int. Conf. on Dependable System and Network, track on Performance and Dependability Symposium (DSN-PDS)*, pages 753–761.

Schiper, A. (1997). Early consensus in an asynchronous systems with a weak failure detector. *Distributed Computing*, 10(3):149–157.

Veríssimo, P. and Casimiro, A. (2002). The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930.