Programando um Subsistema Síncrono para Suporte a Mecanismos Eficientes de Tolerância a Falhas

Andrey E. M. Brito , Francisco V. Brasileiro

Coordenação de Pós-graduação em Informática
Universidade Federal de Campina Grande
Laboratório de Sistemas Distribuídos
Av. Aprigio Veloso, 882, Bodocongó, CEP 58.109-970
Campina Grande, Paraíba, Brasil

{andrey,fubica}@dsc.ufcg.edu.br

Abstract. *In this work we propose the design and implementation of a wormhole - a synchronous subsystem - to be appended to an asynchronous system to allow the solution of fault-tolerant distributed problems that otherwise would have no deterministic solution in a pure asynchronous system. The wormhole architecture encompasses basic services such as clock synchronization and node level failure detection, as well as a programming interface that allows the deployment of specialized synchronous services. One of these services is presented to illustrate the use of the wormhole by an application.*

Resumo. *Nesse trabalho nós apresentamos o projeto e a implementação de um subsistema síncrono (wormhole) a ser incorporado em um sistema assíncrono, a fim de viabilizar a solução de problemas distribuídos tolerantes a falhas, que de outra maneira não teriam solução determinística em um sistema puramente assíncrono. A arquitetura do wormhole incorpora serviços básicos, como sincronização de relógios e detecção de falhas de máquinas do sistema, como também oferece uma interface de programação que permite a instalação de serviços síncronos especializados. Um desses serviços é apresentado para ilustrar a utilização do wormhole por uma aplicação.*

1 Introdução

A maioria das infraestruturas disponíveis para implantação de aplicações distribuídas são caracterizadas pela ausência de limites superiores conhecidos nos atrasos de transmissão de mensagens e de escalonamento de processos, *i.e.* elas são sistemas assíncronos. O resultado apresentado por Fischer *et al.* [Fischer et al., 1985] prova que é impossível atingir consenso [Chandra and Toueg, 1996] (requisito básico para diversos mecanismos para tolerância a falhas) em um sistema distribuído assíncrono sujeito a faltas. Este resultado pode ser resumido da seguinte forma: devido às incertezas no envio e entrega das mensagens, é impossível distinguir entre um processo que falhou e um que está muito lento.

Por outro lado, os sistemas síncronos garantem um limite máximo de tempo nas transmissões de mensagens e de escalonamento de processos, permitindo de forma mais simples soluções para o problema do consenso. A maior parte dos sistemas

práticos não são assíncronos puros, tampouco são síncronos. Eles possuem algum grau de sincronismo. Desta forma, foram propostos vários modelos de sistemas que adicionam algum sincronismo ao modelo assíncrono puro de forma a melhor representar os sistemas práticos. Alguns destes modelos permitem soluções deterministas para o problema do consenso [Cristian and Fetzer, 1999, Verissimo and Almeida, 1995, Chandra and Toueg, 1996].

Entre estes modelos, o modelo assíncrono com detectores de falhas de Chandra e Toueg [Chandra and Toueg, 1996] tem recebido bastante atenção. Isto se deve ao fato de nenhuma consideração sobre os tempos de comunicação e de escalonamento ser feita na construção das aplicações. O sistema é modelado como puramente assíncrono e o sincronismo necessário para resolver problemas como o consenso é abstraido pelo detector de falhas que fornece uma interface e comportamento bem definidos. Desta forma, uma aplicação desenvolvida para este modelo ganha portabilidade, pois independentemente de que componentes do sistema possuem sincronismo, a aplicação não será afetada, apenas a implementação do detector de falhas mudará.

A semântica do serviço de detecção é caracterizada através da definição de duas propriedades básicas: i) abrangência, que determina o mínimo de processos cujas falhas deverão ser assinaladas através de uma suspeição; e ii) exatidão, que limita as falsas suspeições sobre processos que não falharam. Graduando os níveis de abrangência e exatidão, Chandra e Toueg criaram oito classes de detectores [Chandra and Toueg, 1996].

Resolver o problema do consenso utilizando um detector de falhas requer um nível de sincronismo que pode ser descrito da seguinte forma: (*abrangência forte*) após um tempo, todo processo que falha é permanentemente suspeitado por todos os processos corretos; (*exatidão forte consequente*) após um tempo, ao menos um processo correto jamais será erroneamente suspeitado. Essas propriedades definem um detector de falhas conhecido como $\diamond S$ [Chandra and Toueg, 1996] e permitem ao detector suspeitar de processos que não falharam, contanto que em algum momento, deixem de suspeitar erroneamente de algum processo que permaneceu correto.

No entanto, existem problemas que são mais complexos que o problema do consenso e não toleram suspeitas incorretas (por exemplo, a eleição [Sabel and Marzullo, 1995]). Além disso, melhor desempenho pode ser alcançado se os protocolos não precisarem considerar suspeitas incorretas. Para estas situações, entre as classes de detectores de falhas propostos em [Chandra and Toueg, 1996], a classe P (Perfeito) é a mais forte delas e satisfaz as seguintes propriedades: (*abrangência forte*) após um tempo, todo processo que falha é permanentemente suspeitados por todos os processos corretos; (*exatidão forte*) nenhum processo correto é suspeitado antes que falhe.

Implementar um detector de falhas perfeito requer um sistema síncrono [Larrea et al., 2001]. Para contornar as limitações de um sistema síncrono, algumas abordagens foram propostas [Veríssimo and Casimiro, 2002, Fetzer, 2003, Oliveira et al., 2003]. Essencialmente elas assumem um sistema onde ao menos uma pequena parte do sistema irá funcionar de forma síncrona, independente de quão assíncrono seja o resto do sistema. A implementação do detector de falhas perfeito na parte síncrona torna-se então um problema muito simples.

Duas grandes dificuldades existem na implementação de um subsistema síncrono

a ser acoplado a um subsistema assíncrono: a dependência de máquinas comuns para a realização de tarefas síncronas e a interface com a parte assíncrona do sistema. Alguns trabalhos propõem mecanismos para contornar estas dificuldades [Casimiro et al., 2000, Fetzer, 2003, Oliveira et al., 2003]. Esses trabalhos lidam com a imprevisibilidade dos sistemas assíncronos através de pedaços do mesmo que são mais confiáveis (do ponto de vista de sincronismo) e que monitoram o resto do sistema. Entretanto, esses trabalhos fornecem serviços simples e previamente implantados que não precisam lidar com alguns aspectos da interface entre os ambientes síncrono e assíncrono.

Os sistemas baseados em sistemas assíncronos (ou parcialmente síncronos) que poderiam, apenas quando necessário, tirar proveito de um recurso escarço, um subsistema síncrono, são denominados sistemas híbridos. O subsistema síncrono, por sua vez, é chamado de *wormhole* [Veríssimo, 2003] e é simples o suficiente para que sua implementação confiável seja possível.

Finalmente, uma vez implementado o subsistema síncrono, resta a seguinte dúvida: *se sistema síncronos são mais poderosos que detectores de falhas perfeitos na solução de problemas, será que subsistemas síncronos não poderiam ser utilizados para criar abstrações mais fortes que os detectores de falhas perfeitos?* De fato, existem problemas que não podem ser resolvidos em sistemas assíncronos mesmo quando equipados com detectores de falhas perfeitos e requerem um sistema síncrono [Charron-Bost et al., 2000]. É um problema em aberto determinar se existem abstrações que possibilitam a um sistema assíncrono resolver problemas que exigem sistemas síncronos (ou ao menos problemas que exigem mais que um detector de falhas perfeito). Então, equipar sistemas assíncronos com componentes que implementem essas novas abstrações pode habilitá-los a resolver mais problemas que os solúveis utilizando detectores de falhas perfeitos. Além disso, outros problemas já solúveis podem ter seu desempenho melhorado, seja no tempo de terminação ou no número de falhas suportadas.

Nesse trabalho nós apresentamos uma arquitetura para a implementação de um *wormhole* a ser acoplado a um sistema assíncrono. Esta arquitetura fornece além dos serviços básicos de sincronização de relógios e detecção de falhas de nós, providas por outras soluções relatadas na literatura, uma interface de programação que permite a instalação de serviços síncronos especializados. A implementação da abstração de um *compilador de estados globais* usando o *wormhole* é apresentada como uma forma de exemplificar esta interface de programação. O restante deste artigo está estruturado da seguinte forma. A Seção 2 apresenta as principais idéias do projeto de um sistema híbrido. A Seção 3 detalha algumas considerações feitas. Os aspectos de *hardware* e software no projeto são discutidos na Seção 4. A forma de uso do *wormhole* para a implementação de serviços e protocolos é apresentada na Seção 5. Finalmente, conclusões e trabalhos futuros são discutido na Seção 6.

2 Arquitetura do Sistema

O sistema híbrido é formado por uma parte assíncrona e uma parte síncrona e o seu objetivo é possibilitar que um número limitado de serviços simples (por exemplo, um detector de falhas) possa ser implementado na porção síncrona enquanto as aplicações executam na porção assíncrona do sistema. Para possibilitar a implementação de serviços síncronos,

a porção síncrona fornece duas funcionalidades: um serviço de comunicação com garantias de limite máximo de tempo para a entrega de mensagens e um serviço que força o escalonamento dos serviços síncronos nos momentos em que isto é necessário.

A porção assíncrona do sistema é então formada por um conjunto de máquinas interligadas por uma rede local de acesso livre. A porção síncrona, por sua vez, corresponde a um conjunto de dispositivos conectados a essas máquinas e uma rede local privativa que interconecta os dispositivos. Os dispositivos, máquinas associadas e suas redes estão ilustrados na Figura 1, onde um nó é constituído de um dispositivo e uma máquina associada.

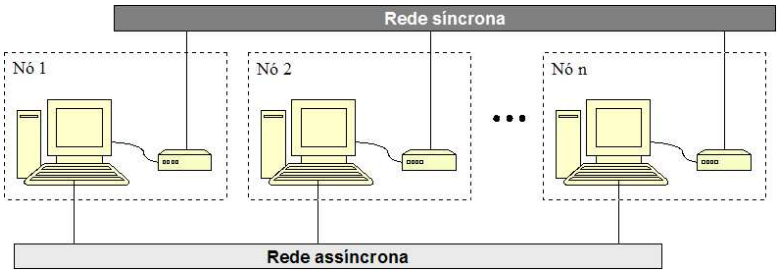

Figura 1: Arquitetura básica

Para que o subsistema síncrono seja utilizado pelo sistema assíncrono *drivers* e serviços são implementados nas máquinas, conforme ilustrado na Figura 2. O dispositivo é acessado exclusivamente pelo seu *driver*. Este controla o acesso, pelos serviços, às funcionalidades do subsistema síncrono. Por fim, as aplicações obtém seus benefícios através de interfaces específicas para cada serviço. Esta organização tem a função primordial de possibilitar a interface entre os subsistema síncrono e o sistema assíncrono na medida que: (1) o subsistema síncrono não pode atender às requisições esporádicas, e talvez concorrentes, das aplicações assíncronas; (2) as aplicações assíncronas podem não conseguir consumir as informações geradas pelo subsistema síncrono na mesma velocidade que são produzidas.

Desta forma, esta organização isola o subsistema síncrono das possíveis interferências das aplicações assíncronas, que poderiam fazer requisições em taxas imprevisíveis e acima da capacidade do sistema síncrono. Por outro lado, os serviços a serem oferecidos às aplicações que executam no sistema assíncrono devem ser construídos de tal forma a tolerar possíveis perdas da informação gerada pelo sistema síncrono (quando a aplicação não consegue executar rápido o suficiente para consumir a informação sendo produzida).

O funcionamento do sistema está ilustrado na Figura 3 e compreende os seguintes passos: (1) um protocolo híbrido se inscreve junto ao *wormhole* e solicita sua execução, caso existam recursos suficientes livres (como largura de banda da rede síncrona), a solicitação é aceita; (2) periodicamente, o dispositivo sinaliza que o protocolo deve ser escalonado; (3) o *driver* força o escalonamento da tarefa que implementa o protocolo; (4) uma vez em execução, o protocolo tem acesso ao *buffer* de envio e recepção da rede síncrona e pode enviar mensagens de comprimento limitado e receber mensagens endereçadas a ele (as mensagens circulam pela rede síncrona e têm um limite máximo para o tempo de entrega); (5) periodicamente e em momentos determinados pelo dispo-

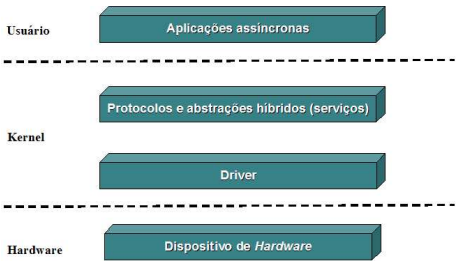

Figura 2: Divisão em camadas do sistema

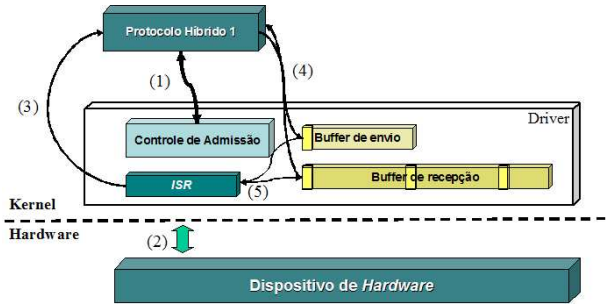

Figura 3: Funcionamento do sistema

sitivo, o *buffer* de envio é repassado ao dispositivo e o *buffer* de recebimento é lido do dispositivo pelo *driver*.

3 Considerações

Como a rede local não garante limites de tempo nas comunicações, ela será chamada de rede assíncrona. O dispositivo de *hardware*, referenciado como "o dispositivo", tem acesso a uma rede privativa. Esta rede garante limites nos tempos de comunicação, desta forma será chamada de rede síncrona. As mensagens serão chamadas de mensagens síncronas ou mensagens assíncronas de acordo com a rede na qual trafegam.

Na rede síncrona, toda comunicação é realizada através de difusões e de forma periódica. Em cada período cada um dos os nós pode transmitir uma mensagem. Assume-se que estas difusões são atômicas, *i.e.* ou o enviador foi bem sucedido em difundir a mensagem e portanto todos os nós corretos irão recebê-la em um intervalo de tempo limitado, ou o enviador falhou durante (ou antes) da difusão e nenhum nó vai receber a mensagem.

Quanto ao processamento, o dispositivo é síncrono por construção. Além disso, ele é capaz de perceber quando a parte síncrona (*driver* e serviços síncronos) executando no PC sofre uma falha de desempenho. Estas falhas de desempenho podem ser conseqüência de uma falha de desempenho de algum serviço do *wormhole* ou do próprio sistema operacional. Em ambos os casos o mecanismo aciona um mecanismo de segurança, que força todo o nó a entrar em um estado seguro (por exemplo, falhar por parada ou reiniciar). Desta forma, falhas em um nó não contaminam os outros nós através da rede síncrona.

4 Implementação

4.1 Interface de *harwdware*

O dispositivo é construído de forma que a parte de *software* do *wormhole* tenha alguns recursos para garantir a comunicação com garantias de tempo e o escalonamento das tarefas. Esses recursos são acessados através da comunicação com uma interface de *hardware*[1] e são os seguintes:

[1]Nossas implementações atuais utilizam a porta paralela ou a porta serial do PC.

- **Interrupção programada**: O dispositivo interrompe periodicamente a máquina, a cada interrupção seu *driver* é executado e algumas pequenas tarefas podem ser executadas de forma síncrona.
- **Comunicação com garantia de limite de tempo**: O dispositivo aceita mensagens para serem enviadas em blocos, periodicamente, com garantias de limite de tempo até sua entrega aos nós de destino.
- **Cão-de-guarda**: Um cão-de-guarda é configurado no dispositivo para que quando alguma tarefa que deveria ter sido executada pelo *driver* ou pelos serviços (em conseqüência das interrupções periódicas do dispositivo) sofra uma falha de desempenho ou de parada, o nó seja colocado em um estado seguro e não comprometa as propriedade de segurança do sistema.

Para garantir o sincronismo e a periodicidade na rede síncrona o dispositivo implementa o protocolo TDMA (*Time Division Multiple Access*) proposto por Brito [Brito, 2004]. Este protocol controla o acesso ao canal de comunicação dividindo-o em períodos. Em cada período, todos os nós que fazem parte da rede síncrona possuem uma fatia de tempo. Para implementar o protocolo TDMA, um líder é utilizado para sincronizar todos os nós e para alocar fatias de tempo aos nós que desejam fazer parte da rede. A falha deste líder é tolerada através de um mecanismo que elege automaticamente um dos dispositivos corretos para assumir as responsabilidades do líder.

4.2 Interface do software

A porção de *software* do *wormhole* consiste em módulos carregáveis do sistema operacional Linux [Bovet and Cesati, 2003]. Um módulo é um pedaço de código compilado que pode ser acoplado dinamicamente ao sistema operacional em execução. Esses módulos utilizam as funcionalidades disponibilizadas pelo dispositivo e implementam três serviços básicos. Estes serviços básicos são acessados a partir de funções que eles exportam para o núcleo do sistema operacional. Desta forma, outros módulos do sistema operacional podem acessá-las diretamente e, da mesma forma, podem disponibilizá-las para os processos de usuário a partir de chamadas de sistema, arquivos especiais ou soquetes de comunicação

local. Estes serviços são os seguintes:

- Relógio Global;
- Serviço de detecção de falhas a nível de nós; e
- Controle de admissão.

Além destes, um conjunto de funções de propósito geral foi definido para permitir a recuperação de informações de configuração do *wormhole*. As funções são as seguintes:

- $id\_list \Leftarrow get\_ids()$: retorna uma lista dos identificadores dos nós que fazem parte atualmente da rede síncrona.
- $max\_nodes \Leftarrow get\_max\_nodes()$: retorna o número máximo de nós que podem fazer parte da rede síncrona.
- $max\_delay \Leftarrow get\_max\_delay()$: retorna o tamanho máximo do período TDMA, calculado a partir da quantidade máxima de nós da rede síncrona. O tamanho máximo do período corresponde ao atraso máximo que uma mensagem enviada pela rede síncrona pode sofrer.

Os serviços básicos são detalhados a seguir.

Relógio global sincronizado. As mensagens periódicas no *wormhole* são utilizadas na construção de uma referência de tempo global. Esta referência de tempo pode ser acessado a partir da função exportada para o núcleo:

- $current\_time \Leftarrow get\_global\_time()$: retorna o valor da referência de tempo global.

Serviço de Detecção de Falhas. O serviço de detecção de falhas identifica que nós participantes da rede síncrona estão corretos. Este serviço pode ser acessado a partir das seguintes funções:

- $ip\_list \Leftarrow get\_corrects()$: retorna a lista de IPs dos nós corretos que fazem parte atualmente da rede síncrona.
- $correct \Leftarrow is\_correct(ip)$: verifica se o nó cujo IP é ip faz parte atualmente da rede síncrona.

O controle de admissão. Na prática, sincronismo só pode ser obtido através de acesso controlado. Além disso, para que o sistema síncrono de capacidade limitada possa atender as requisições do sistema assíncrono chegando de forma esporádica, deve haver um intermediário no sistema assíncrono que colete as requisições assíncronas e se comunique de forma controlada com a parte síncrona.

Os serviços básicos não necessitam de um controle de acesso pois o detector de falhas e a referência de tempo global têm como resultado um valor monotônico e somente de leitura que pode ser compartilhado por todas as aplicações interessadas. Entretanto, existirão serviços e protocolos carregados dinamicamente, estes serviços utilizam a banda do *wormhole* para comunicar-se com outras instâncias suas em outros nós da rede síncrona. Desta forma, a quantidade de serviços dinâmicos carregados deve ser limitada. A iniciação destes serviços dinâmicos é gerida pelo controle de admissão. Como a banda é compartilhada entre os serviços dinâmicos e eles precisam ser carregados simultaneamente em todos os nós que participam do serviço, os nós devem entrar em acordo em todas as requisições de iniciação de um serviço.

O serviço de controle de admissão especifica o serviço a ser carregado, uma lista dos participantes e o número mínimo de participantes que precisam confirmar a participação para que o serviço seja carregado. O serviço de controle de admissão é acessado a partir das seguintes funções:

- $return \Leftarrow wh\_subscribe\_service(name, bandwidth, handler)$: cadastra um serviço na lista de serviços localmente disponíveis em um nó do *wormhole*. O parâmetro $name$ corresponde ao nome do serviço, o parâmetro $bandwidth$ corresponde ao número de bytes que o serviço utilizará para o envio de mensagens em cada período do *wormhole* e o parâmetro $handler$ é um apontador para a função responsável pelo serviço (detalhada nas próximas seções). Retorna um número negativo em caso de falha na inscrição.
- $wh\_unsubscribe\_service(name)$: remove um serviço da lista de serviços localmente disponíveis em um nó do *wormhole*.

- $handler \Leftarrow wh\_request\_service(name, parameters, participants, quorum)$: faz uma difusão na rede síncrona, solicitando uma alocação de uma porção do canal síncrono do serviço $name$ nos nós especificados pelo parâmetro $participants$ e solicitando também o carregamento desse serviço. Os parâmetros necessários ao serviços devem ser conhecidos e são encapsulados na cadeia de bytes $parameters$. O parâmetro $quorum$ corresponde ao número mínimo de participantes para que o serviço seja iniciado.
- $wh\_remove\_service(name)$: encerra a execução de um serviço.

5 Usando o sistema implementado

5.1 O Compilador de Estados Globais

O Compilador de Estados Globais (CEG) é um exemplo de implementação de um serviço que utiliza o *wormhole* implementado. O CEG produz uma representação resumida, limitada e consistente dos estados locais de todos os processos que executam um protocolo, na forma de uma seqüência ordenada de "resumos de estados globais" (REGs).

No caso do consenso, o problema escolhido para exemplificar a utilização do CEG, ele fornecerá informações que possibilitam que o protocolo se adapte às variações nos níveis de contenção vividos pelo sistema durante uma execução do protocolo - uma característica que não está presente em nenhum outro protocolo de consenso baseado em detectores de falhas perfeito. Dessa forma, o consenso terminará tão rápido quanto o nó mais rápido consiga difundir sua mensagem de proposta entre os outros nós.

Para resolver o problema do consenso entre um conjunto de n processos $\Pi = \{p_1, p_2, ..., p_n\}$, o CEG provê REGs com a seguinte estrutura:

- *detection_vector*: um vetor de bits com n bits, onde o elemento i representa o estado operacional do processo p_i (inicialmente 0 e ajustado para 1 se p_i falha);
- *reception_matrix*: uma matriz de bits de dimensões $n \times n$ onde o bit $[i, j]$ indica se p_i recebeu uma mensagem do protocolo de consenso vinda de p_j (inicialmente 0 e ajustado para 1 quando uma mensagem é recebida); e
- *consensual_identity*: um campo com $\lceil log_2 n \rceil$ bits[2] que contém a identidade do processo que propôs a mensagem consensual do protocolo (inicialmente $\perp$).

5.1.1 Implementando o CEG

Na implementação do *wormhole* e do CEG detalhada no trabalho de Brito [Brito, 2004], os serviços foram mapeados em arquivos especiais no sistema de arquivos. Para isso foi adicionado um parâmetro $file\_operations$ à função $wh\_subscribe\_service()$. Este parâmetros contém um apontador para as funções que manipulam os acessos a um arquivo especial. Desta forma, as aplicações podem acessar os REGs acessando um arquivo especial diretamente conectado ao CEG. Para acessar o CEG, uma aplicação acessa um arquivo especial de número maior 252 e número menor 31 (criado com $mknod\ wh\_adm\_control\ c\ 252\ 31$)[3]. Esse arquivo está diretamente conectado ao serviço

[2] $\lceil log_2 n \rceil$ é a quantidade de bits necessária para representar um numero x, onde $0 < x < n$.

[3] O número maior e o número menor são dois parâmetros que o Linux usa para identificar os *drivers* associados a um arquivo especial [Bovet and Cesati, 2003]

de controle de admissão. Uma vez aberto o arquivo, a aplicação escreve um pacote contendo o nome do serviço (10 bytes), os parâmetros do serviço (10 bytes), a lista de participantes (10 bytes, limitada atualmente a 10 participantes) e o quórum (1 byte) – bytes não utilizados devem ser preenchidos com o caracter $'\backslash 0'$. Quando o processo da aplicação escreve nesse arquivo especial, ele fica bloqueado enquanto o serviço é iniciado. Depois de iniciado, o processo é desbloqueado e as funções que manipulam acessos a esse arquivo passam a ser as funções do próprio CEG. O CEG tem três funções associadas ao arquivo especial: leitura, escrita e o fechamento do arquivo (já que a abertura do arquivo foi tratada pelo próprio *wormhole*).

A leitura do arquivo especial é feita em blocos que representam REGs. O tamanho de cada REG é proporcional a quantidade de participantes. De forma semelhante, na escrita, cada bloco representa o estado local do processo que participa do protocolo. O estado local consiste no conjunto dos identificadores dos remetentes cujas mensagens do protocolo já foram recebidas. Para encerrar o serviço é necessário que um valor especial seja escrito no arquivo especial antes que ele seja fechado. Caso o arquivo seja fechado sem a escrita deste valor, o serviço continuará em execução até que algum nó encerre o serviço ou que todos os nós falhem. Esse procedimento é necessário pois uma falha no processo pode causar o fechamento acidental do arquivo, o que poderia encerrar o serviço em todos os nós que o executam.

Por baixo do arquivo especial, existe um módulo que implementa o serviço em si. Este módulo é composto de seis funções básicas: a função $init\_module()$, a função $cleanup\_module()$, a função de processamento do serviço, além de outras três funções necessárias para manipular os acessos de leitura, escrita e fechamento do arquivo especial.

A função $init\_module()$ é responsável pelo procedimento de iniciação do módulo e é executada pelo próprio Linux. A função $init\_module()$ de um serviço do *wormhole* realiza as seguintes tarefas: (1) inscreve-se no *wormhole* através da função $wh\_subscribe\_service()$, informando seu nome, a largura de banda necessária (em bytes por período) e o apontador para a função que realiza o processamento de suas mensagens; (2) realiza os procedimentos necessários para interagir com o ambiente assíncrono (por exemplo, alocando números maiores ou especificando a estrutura responsável pelas operações no arquivo especial) e quaisquer recursos que o módulo utilize durante a realização de seus serviços.

A função $cleanup\_module()$ é responsável pela procedimento de finalização do módulo antes de sua remoção da memória e do núcleo. O *wormhole* requer que esta função remova o cadastro do serviço junto ao mesmo. Além disso, esta função deve liberar quaisquer recursos utilizados durante sua execução.

A função de processamento do serviço é uma função que será executada como um *tasklet* pelo *driver* do *wormhole* sempre que um período acaba. Esta função tem acesso aos *buffers* de entrada e de saída do *wormhole*, de forma que ela tem acesso às mensagens recebidas no período anterior e pode escrever mensagens para serem enviadas no período seguinte. No caso do CEG, a função de processamento constrói os REGs a partir do *buffer* de mensagens recebidas (dos outros nós que participam do serviço) e escreve estados locais no *buffer* de mensagens a serem enviadas. Desta forma, em cada máquina, a instância do CEG envia através do *wormhole* apenas seu estado local e recebe através

do *wormhole* os estados locais de todas as máquinas que executam o CEG. De posse de todos os estados locais, cada instância do CEG constrói a matriz $reception\_matrix$ e a partir dela, o identificador $consensual\_identity$. O vetor $detection\_vector$ pode ser construindo simplesmente avaliando se a instância ceg_i executando na máquina i enviou seu estado local durante o último período.

As funções de manipulação dos acessos aos arquivos especiais, são encapsuladas numa estrutura do tipo $file\_operations$ e então atribuídas ao respectivo campo na estrutura $file$ do arquivo aberto. Um acesso $READ$ retorna o REG mais recente, de comprimento igual a soma das estruturas $detection\_vector$, $reception\_matrix$ e $consensual\_identity$ ($n+n^2+\lceil log_2 n \rceil$ bits). Um acesso $WRITE$ deve informar o estado local, ou seja, um vetor de n bits onde o i-ésimo bit indica se o processo em questão recebeu ou não uma mensagem de proposta do consenso do processo p_i. Um acesso $CLOSE$ ativa a função que libera o arquivo e pode encerrar a execução do serviço, desde o encerramento do arquivo tenha sido liberado por um acesso $WRITE$ com o valor especial de encerramento.

5.1.2 Resolvendo consenso com o CEG

O problema do consenso uniforme consiste em cada processo p_i propor um valor v_i e todos os processos que decidem devem decidir por um dos valores propostos. Formalmente, o consenso pode ser definido pelas seguintes propriedades [Charron-Bost et al., 2000]:

- **terminação**, após um tempo finito todo processo correto decide algum valor;
- **integridade uniforme**, todo processo decide apenas uma vez;
- **validade uniforme**, se um processo decide por um valor v, então v foi proposto por algum processo; e,
- **acordo uniforme**, todos os processos que decidem, corretos ou não, decidem o mesmo valor.

O CEG suporta uma família de protocolos de consenso que se diferenciam por dois parâmetros. O primeiro, denominado *quórum*, define o número de processos que são necessários para "eleger" o processo que propôs o valor consensual. Este parâmetro afeta apenas a parte síncrona do protocolo. O valor do quórum é tal que $f + 1 \leq quorum \leq n$. O segundo parâmetro, denominado *proponentes*, define o número de processos que irão propor um valor durante a execução do protocolo. Este, afeta apenas a parte assíncrona do protocolo e o seu valor é tal que $f + 1 \leq proponentes \leq n$. Desta forma, cada membro da família de protocolos de consenso é descrita como *Consenso-CEG(Q,P)* onde Q e P são os parâmetros quórum e proponentes, respectivamente.

A parte assíncrona do protocolo (a aplicação) é estruturada na forma de três tarefas concorrentes. Na primeira tarefa, a tarefa de *proposição*, P processos enviam mensagens para os outros processos contendo suas propostas. A segunda tarefa, a tarefa de *recebimento*, é responsável por receber e armazenar as mensagens de propostas enviadas por outros processos. Ela também notifica o CEG que determinada mensagem foi recebida. A tarefa final, a tarefa de *decisão*, é responsável por detectar que uma decisão pode ser tomada e que a execução do protocolo está terminada.

A tarefa de decisão é também bastante simples, ela permanece em um laço consultando o CEG. Quando um resumo de estado global é entregue com um campo

$consensual\_identity$ preenchido com o identificador x de algum processo, a tarefa de decisão verifica se a mensagem de p_x já foi recebida. Caso não tenha sido, a tarefa espera até que ela seja recebida. Em ambos os casos, depois de recebida a mensagem de p_x, a tarefa decide pelo valor contido na mensagem e termina a execução enviando a mensagem de p_x para todos os processos corretos que ainda não a receberam. O Algoritmo 1 é o pseudo-código das tarefas que implementam a parte assíncrona do protocolo (a prova formal que este algoritmo resolve o problema do consenso uniforme pode ser encontrada em [Brito, 2004]).

Algoritmo 1 Pseudo-código do protocolo Consenso-CEG(Q, P) executado pelo processo p_i

```
% variáveis compartilhadas
bagOfMessages_i = ∅
decided_i = falso

% Tarefa de proposição
quando execute propose(v_i)
  se i ≤ p então envie m_i(v_i) para todos os processos fim se
fim
||
% Tarefa de recebimento
enquanto não decided_i faça
  quando recebe m_j(v_j) de p_j
    se m_j(v_j) não pertence à bagOfMessages_i então
      adicione m_j(v_j) à bagOfMessages_i
      notifique ceg_i do recebimento de uma mensagem de proposta vinda de p_j
    fim se
  fim
fim enquanto
||
% Tarefa de decisão
enquanto não decided_i faça
  reg = read(CEG)
  x = reg.consensusal_identity
  se x ≠⊥ então
    espera até m_x(v_x) em bagOfMessages_i
    m_x(v_x) = getConsensualMessage(x, bagOfMessages_i) % recupera mensagem de p_x
    envie m_x(v_x) para todo p_k tal que reg.detection_vector[k] = 0 ∧ reg.reception_matrix[k, x] = 0
    decided_i = true
    return(v_x) % decide pelo valor proposto por p_x
  fim se
fim enquanto
```

6 Conclusão

Neste trabalho nós apresentamos como um *wormhole* pode ser construído com o objetivo de possibilitar que serviços simples possam ser implementados na porção síncrona enquanto as aplicações executam na porção assíncrona do sistema.

Como exemplo de um serviço, detalhamos como o *wormhole* foi utilizado para implementar uma nova abstração, o Compilador de Estados Globais (CEG). O CEG é uma abstração mais forte que os detectores de falhas perfeitos e no entanto, sua implementação requer as mesmas considerações que a implementação de um detector de falhas perfeito. Nós discutimos a utilização do CEG para solução do problema do consenso uniforme que atinge o consenso em uma única rodada de comunicação.

Agradecimentos

Os autores agradecem a Walfredo Cirne pelas discussões ao longo do desenvolvimento desse trabalho e aos revisores anônimos, pelos comentários pertinentes. Este trabalho foi parcialmente financiado pelo CNPq (processo 300646/1996-8) e pelo PRH-25/ANP.

Referências

Bovet, D. and Cesati, M. (2003). *Understanding the Linux Kernel*. O'Reilly, 3 edition.

Brito, A. E. M. (2004). Uma arquitetura híbrida para o suporte de protocolos distribuídos tolerantes a falhas. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande.

Casimiro, A., Martins, P., and Veríssimo, P. (2000). How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–1343, Porto, Portugal. IEEE Industrial Electronics Society.

Chandra, T. 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 IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, pages 523–532, New York, USA. IEEE Computer Society.

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

Fetzer, C. (2003). Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112.

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

Larrea, M., Fernández, A., and Arévalo, S. (2001). On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Brief Announcements 15th Int'l Symp. Distributed Computing (DISC 2001)*.

Oliveira, E. W., Brito, A. E. M., and Brasileiro, F. V. (2003). Projeto e implementação de um serviço de detecção de falhas perfeito. In *Simpósio Brasileiro de Redes de Computadores*, pages 697–712, Natal/RN, Brasil.

Sabel, L. S. and Marzullo, K. (1995). Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University.

Verissimo, P. and Almeida, C. (1995). Quasi-synchronism: a step away from the tradicional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39.

Veríssimo, P. (2003). Uncertainty and predictability: Can they be reconciled? *Future Directions in Distributed Computing, Springer Verlag LNCS 2584*, pages 108–113.

Veríssimo, P. and Casimiro, A. (2002). The Timely Computing Base model and architecture. *Transactions on Computers*, 51(8):916–930.