Universidade Federal de Campina Grande

Centro de Ciências e Tecnologia

Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Uma Arquitetura Híbrida para o Suporte de Protocolos Distribuídos Tolerantes a Falhas

Andrey Elísio Monteiro Brito

Campina Grande, Paraíba, Brasil

Fevereiro - 2004

Uma Arquitetura Híbrida para o Suporte de Protocolos Distribuídos Tolerantes a Falhas

Andrey Elísio Monteiro Brito

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Vilar Brasileiro

(Orientador)

Campina Grande, Paraíba, Brasil

Resumo

A maioria das infra-estruturas 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. Esses sistemas são tidos como o melhor ambiente para o desenvolvimento de aplicações devido à portabilidade e escalabilidade decorrente da falta de considerações temporais fortes sobre o sistema. Entretanto, alguns dos mais fundamentais problemas dos sistemas distribuídos não podem ser resolvidos nesses sistemas quando estes estão sujeitos a falhas. Os sistemas síncronos, por outro lado, permitem soluções triviais para os problemas básicos dos sistemas distribuídos tolerantes a falhas. Isso acontece porque eles garantem um atraso máximo conhecido no escalonamento de processos e nos atrasos na transmissão de mensagens. Entretanto, a maior parte dos sistemas práticos não são síncronos, e isso motivou a criação de modelos intermediários entre esse dois tipos, chamados de modelos parcialmente síncronos. Um dos mais populares modelos parcialmente síncronos é o modelo assíncrono equipado com detectores de falhas não confiáveis. Neste modelo cada processo possui acesso a um módulo que pode fornecer informações sobre quais processos falharam.

Entre os detectores de falhas propostos, o detector de falhas com semântica perfeita é o mais forte e o único que não comete erros (por exemplo, suspeitando de processos que não falharam). Infelizmente, detectores de falhas perfeitos só podem ser implementados em sistemas síncronos. Como algumas aplicações exigem detectores de falhas perfeito, estes detectores têm recebido uma crescente atenção na literatura. Para implementá-los, subsistemas síncronos de capacidade bastante limitada têm sido desenvolvidos sob o sistema parcialmente síncrono onde as aplicações executam. Este trabalho detalha a implementação de um subsistema síncrono em um ambiente assíncrono formado por uma rede de PCs conectados por uma rede local e executando um sistema operacional de prateleira.

Finalmente, embora a implementação de detectores de falhas perfeitos, por si só justifique a implementação de um subsistema síncrono, nós acreditamos que abstrações mais fortes que os detectores de falhas perfeitos podem ser desenvolvidas. Apresentamos uma destas abstrações, o Compilador de Estados Globais, que possibilita soluções mais rápidas para o problema do consenso, onde vários processos tentam entrar em acordo sobre um valor proposto.

Abstract

Most of the infra-structures for deploying distributed applications are characterized by the absence of known upper bounds in the communication and process scheduling delays, *i.e.* they are asynchronous systems. Asynchronous systems are considered to be the best environment to develop applications for its portability and scalability, consequence of the lack of strong timing assumptions about the system. Unfortunately, most of the basic problems of distributed systems cannot be solved in such a system when they are subjected to failures. On the other hand, the synchronous systems allow trivial solutions to the basic problems of distributed systems. This is consequence of the existence of known upper bounds in communication and scheduling delays. However, most of the practical systems are not synchronous, and this fact motivates the conception of intermediate models between these two systems, named partially synchronous models. One of the most popular partially synchronous model is the asynchronous model equipped with an unreliable failure detector. This model consists in an asynchronous model, in which each process has access to a module that gives information about which processes have failed.

Among the proposed failure detectors, the perfect failure detector is the strongest and is the only one which does not make mistakes (for example, by suspecting processes that did not fail). Unfortunately, perfect failure detectors can only be implemented in synchronous systems. However, as some applications require perfect failure detectors, the interest in them has recently raised. A recent approach taken by the designers of such systems is to implement synchronous subsystems with very limited capacity underneath the partially synchronous system where the applications execute. This work details the implementation of a synchronous subsystem in an environment composed of standards PCs connected through a local area network and running an off-the-shelf operating system.

Finally, although the implementation of perfect failure detectors by itself justify the development of such synchronous subsystem, we believe that stronger abstractions than the perfect failure detectors can be developed. We introduce one of these abstractions, the Global State Digest Provider, which allows faster solutions to the consensus problem, where processes try to agree in a proposed value.

Agradecimentos

A minha família pelo suporte em tempo integral e por me ajudar a contornar os obstáculos menos triviais sem me desviar do caminho.

A Esther pela paciência e apoio durante todo o mestrado, pela companhia durante o (segundo ela, pouco) tempo livre e especialmente, por me ajudar a ser mais organizado.

A Fubica, meu orientador, pelas constantes boas idéias e atenção no que foi preciso.

A Lívia e Ely, com quem trabalhei várias vezes e sempre foram pessoas atenciosas e simpáticas. A Raquel que, por ter sua máquina vizinha à minha, foi vítima de todas as minhas dúvidas sobre qualquer assunto.

Aos meus amigos Helon, Rodrigo, Márcio, Luciano, Paulo e Wilson, pelas inúmeras horas em atividades “menos acadêmicas”.

A Cadu e Buriti, que nesses últimos meses me ajudaram na “materialização” de algumas idéias do projeto.

E finalmente, a todo pessoal do LSD, pelo ambiente estimulante e descontraído, o que tornou o trabalho ainda mais agradável.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A maioria das infraestruturas disponíveis para implantação de aplicações distribuídas são caracterizadas pela ausência de limites superiores nos atrasos de transmissão de mensagens e de escalonamento de processos, *i.e.* elas são sistemas assíncronos. O conhecido resultado apresentado por Fischer *et al.* [FLP85] prova que é impossível atingir consenso [CT96] de forma determinística em um sistema distribuído assíncrono sujeito a faltas. O problema do consenso consiste na capacidade de vários processos distribuídos entrarem em acordo sobre um valor proposto e é um requisito básico para diversos mecanismos para tolerância a falhas. Este resultado pode ser resumido da seguinte forma: devido às incertezas temporais no envio, entrega e processamento das mensagens, é impossível distinguir entre um processo que falhou e um que está muito lento.

Por outro lado, os sistemas síncronos são baseados na existência de limites de tempo nas transmissões de mensagens e de escalonamento de processos, permitindo de forma trivial soluções para o problema do consenso. Os sistemas práticos não são assíncronos puros, tampouco são síncronos.

A maior parte dos sistemas práticos possuem algum grau de sincronismo que pode estar localizado em alguns de seus componentes, como por exemplo: um relógio interno com desvio de velocidade limitado, uma rede com qualidade de serviço que garante um tempo máximo de comunicação ou uma rede que não perde mensagens nem inverte sua ordem. 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.

O modelo assíncrono temporizado [CF99] assume que o sistema irá, de tempos em tempos, se comportar de forma síncrona e que durante estes períodos síncronos os protocolos em execução poderão progredir em sua execução (*liveness*). O modelo quase-síncrono [VA95] assume limites para o tempo de comunicação e de escalonamento de processos e atribui a esses limites uma probabilidade de não serem cumpridos. Nos momentos que os limites são ultrapassados, o que é detectado através de um componente síncrono no sistema (por exemplo, um relógio local), algum mecanismos de segurança[1] deve lidar com essa falha de desempenho. O modelo assíncrono equipado com um detector de falhas não-confiável [CT96], por sua vez, assume a existência de um "oráculo", o detector de falhas. Este oráculo possui conhecimento sobre quais processos podem ter falhado. Embora este oráculo possa cometer erros (por exemplo, suspeitando de processos corretos), a informação fornecida por ele é suficiente para permitir soluções determinísticas para os problemas básicos dos sistemas distribuídos.

Entre estes modelos, o modelo assíncrono com detectores de falhas 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 (do inglês, *completeness*), que determina o mínimo de processos cujas falhas deverão ser assinaladas através de uma suspeição; e ii) exatidão (do inglês, *accuracy*), 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 [CT96].

Resolver o problema do consenso utilizando um detector de falhas requer um nível de

[1] Neste trabalho, a palavra *segurança* é usada com o sentido da palavra *safety* do inglês e corresponde às propriedades do sistema que não podem ser descumpridas.

sincronismo que pode ser descrito da seguinte forma: (1) após um tempo, todo processo que falha é permanentemente suspeitado por todos os processos corretos; (2) após um tempo, ao menos um processo correto jamais será erroneamente suspeitado. Essas propriedades definem um detector de falhas conhecido como $\Diamond S$ [CT96] 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 [SM95]). 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, a classe P (Perfeito) é a mais forte delas e satisfaz as seguintes propriedades: (1) após um tempo, todo processo que falha é permanentemente suspeitado por todos os processos corretos; (2) nenhum processo correto é suspeitado antes que falhe.

Implementar um detector de falhas perfeito requer um sistema síncrono [LFA01]. Para contornar esse requisito, algumas abordagens foram propostas [VC02; Fet03; OBB03]. Essencialmente elas assumem um sistema híbrido 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 trivial.

Duas grandes dificuldades existem na implementação de um subsistema síncrono: 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 [CMV00; Fet03; OBB03]. 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 previamente implantados e que não precisam lidar com alguns aspectos da interface entre os ambientes síncrono e assíncrono, como por exemplo, a diferença entre a taxa que o sistema assíncrono consome e taxa que o subsistema síncrono gera informações.

Uma vez que implementar detectores de falhas perfeitos requer um subsistema síncrono e que sistemas síncronos são mais poderosos que detectores de falhas perfeitos na solução de problemas, talvez sistemas síncronos pudessem ser utilizados para criar abstrações mais

fortes que os detectores de falhas perfeitos [Ver03]. De fato, existem problemas que não podem ser resolvidos em sistemas assíncronos mesmo quando equipados com detectores de falhas perfeitos, por exemplo, problemas que requerem um sistema síncrono [CBGS00]. Determinar se existem abstrações que possibilitem a um sistema assíncrono resolver problemas que exigem sistemas síncronos (ou ao menos problemas que exigem mais que um detector de falhas perfeito) é um problema em aberto. Então, equipar sistemas assíncronos com componentes que implementem essas novas abstrações poderiam habilitá-los a resolver mais problemas que os solúveis utilizando detectores de falhas perfeitos. Além disso, outros problemas já solúveis poderiam ter melhorado seu desempenho, seja no tempo de terminação ou no número de falhas suportadas.

1.2 Objetivo

O objetivo deste trabalho é a construção de um suporte síncrono para ambientes assíncronos. Esse suporte será utilizado na construção de sistemas híbridos, compostos de uma parte assíncrona e uma parte síncrona. Na parte assíncrona as aplicações e atividades normais de um sistema convencional serão executadas. Na parte síncrona, de capacidade limitada, serão executados protocolos que forneçam informações que permitam que a parte assíncrona amplie seu domínio de problemas solúveis ou melhore seu desempenho.

Os objetivos específicos deste trabalho são:

- Desenvolver um canal de comunicação síncrono para ser utilizado pelos nós de uma rede local de prateleira;
- Desenvolver um ambiente de execução síncrono, de capacidade limitada;
- Desenvolver um mecanismo de interface entre a parte síncrona e a assíncrona que permita que a parte síncrona possa atender as requisições da parte assíncrona, que chegam de forma arbitrária, sem comprometer seus prazos ou interferir em outros nós;
- Desenvolver exemplos que utilizem o suporte desenvolvido e mostrem como um sistema assíncrono pode tirar proveito do fato de que uma pequena parte do sistema tem um comportamento síncrono.

1.3 Organização do trabalho

Este trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os principais conceitos e trabalhos relacionados. Neste capítulo é discutida a teoria dos detectores de falhas não confiáveis e é feita uma comparação entre os trabalhos existentes, discutindo os resultados obtidos pelos mesmos.

O Capítulo 3 detalha os principais aspectos envolvidos no projeto de um sistema híbrido, ou seja, as decisões de projeto sobre as tecnologias disponíveis. Primeiramente, os requisitos do trabalho são detalhados e, a partir deles, as alternativas de projeto para implementação da comunicação e do processamento síncronos são discutidas. Uma vez tomadas as decisões de projeto, a implementação propriamente dita é discutida no Capítulo 4, onde é detalhado o funcionamento do sistema.

Uma vez implementado o *Wormhole*, o Capítulo 5 mostra como novas abstrações podem ser desenvolvidas e apresenta um exemplo, o Compilador de Estados Globais, que tira proveito da existência de um canal síncrono para agilizar a solução do problema de consenso em um sistema híbrido.

Por fim, o Capítulo 6 resume os principais resultados obtidos e discute os trabalhos futuros.

Capítulo 2

Trabalhos relacionados

2.1 Modelos de sistemas computacionais

Os sistemas computacionais distribuídos podem ser classificados em sistemas síncronos e assíncronos. Os sistemas síncronos são aqueles sistemas onde existe um limite de tempo na entrega de mensagens e na execução de tarefas. Na prática, sistemas síncronos são construídos a partir de componentes especiais: máquinas especiais executando um sistema operacional de tempo-real, equipadas com uma rede com garantia de qualidade de serviço e principalmente, o sistema está sujeito a uma carga limitada.

Os sistemas assíncronos são aqueles onde não existe sequer a noção de tempo, nenhuma consideração é feita sobre a velocidade relativa ou absoluta dos processos ou sobre o tempo de entrega de mensagens, que podem ser atrasadas indefinidamente. Na prática qualquer sistema pode ser considerado um sistema assíncrono, por isso, desenvolver aplicações para sistemas assíncronos é provê-las com o máximo de portabilidade e de escalabilidade.

Entretanto, se por um lado os sistemas assíncronos são o melhor ambiente para o desenvolvimento de aplicações, por outro lado, até mesmo alguns dos problemas mais básicos dos sistemas distribuídos não podem ser resolvidos nesse ambiente quando o mesmo está sujeito a falhas. Por exemplo, o consenso[1] não pode ser resolvido em sistemas assíncronos sujeitos a falhas devido à impossibilidade de se distinguir um processo que falhou de um que está muito lento [FLP85]. Esta impossibilidade impulsionou pesquisas com o

[1]No problema do consenso, cada processo p_i propõe um valor v_i e todo processo correto no sistema deve decidir por um mesmo valor v entre os valores propostos mesmo que até f processos falhem, $f < n$.

objetivo de identificar o menor grau de sincronismo que um sistema sujeito a falhas deve prover, de modo a permitir soluções determinísticas para o consenso e outros problemas fundamentais dos sistemas distribuídos. Como resultado destas pesquisas, novos modelos de sistemas foram propostos, com graus de sincronismo intermediários entre o assíncrono puro e o síncrono puro. Esses sistemas são denominados de parcialmente síncronos [DDS87; DLS88].

Dentre os modelos de sistemas parcialmente síncronos destacam-se o modelo assíncrono equipado com detectores de falhas não confiáveis e o quasi-síncrono, que foram a base deste trabalho. O modelo assíncrono equipado com detectores de falhas não confiáveis [CT96] não faz considerações sobre a velocidade dos processos e tempos de comunicação, da mesma forma que no sistema assíncrono puro, mas cada processo está equipado com um módulo detector de falhas. Este módulo é um "oráculo" que conhece quais processos falharam. Os detectores de falhas são não confiáveis pois podem cometer erros, suspeitando de um processo que não falhou ou não suspeitando de um que falhou. Entretanto, a informação sobre as falhas é suficiente para permitir a solução de certos problemas fundamentais dos sistemas distribuídos.

Os detectores de falhas são especificados em termos de propriedades abstratas ao invés de especificações de implementação. Isso permite que as aplicações distribuídas sejam desenvolvidas e testadas a partir das definições das propriedades ao invés de características das implementações, provendo portabilidade.

As propriedades que definem um detector de falhas são a abrangência (*completeness*) e a exatidão (*accuracy*). A abrangência define quantos processos terão suas falhas detectadas pelos módulos detectores de falhas. Existem dois níveis de abrangência: (1) a abrangência forte, que requer que a falha de um processo seja detectada por todos os módulos detectores de falhas; (2) a abrangência fraca, que requer que a falha de um processo seja detectada por pelo menos um módulo detector de falhas.

A exatidão, por sua vez, restringe as falsas suspeições que um módulo pode fazer. Existem quatro níveis de exatidão: (1) a exatidão forte, que requer que os processos não sejam suspeitados antes que falhem; (2) a exatidão forte após um tempo, que requer que, após um tempo finito, a exatidão forte seja atendida; (3) a exatidão fraca, que requer que pelo menos um processo não seja suspeitado antes que falhe; (4) a exatidão fraca após um tempo, que

requer que, após um tempo finito, a exatidão fraca será atendida.

A partir das combinações dos níveis de abrangência e de exatidão Chandra e Toueg propuseram oito classes de detectores de falhas [CT96], listadas na Tabela 2.1.

	Exatidão			
Abrangência	**Forte**	**Fraca**	**Forte após um tempo**	**Fraca após um tempo**
Forte	P	S	$\Diamond P$	$\Diamond S$
Fraca	Q	W	$\Diamond Q$	$\Diamond W$

Tabela 2.1: Classes de detectores de Falhas

Duas dessas classes definidas são especialmente interessantes, a classe dos fortes após um tempo ($\Diamond S$) e a classe dos perfeitos (P). A classe $\Diamond S$ destaca-se por possuir a semântica mais fraca que permite solução determinística para o problema do consenso. A classe P, a mais forte definida, é a única capaz de resolver certos problemas de sistemas distribuídos como por exemplo a eleição de um líder [SM95][2]. Além disso, em sistemas assíncronos sujeitos apenas à falhas por parada, os detectores da classe P permitem soluções para o consenso sem limitar o número máximo de falhas.

Os detectores de falhas perfeitos receberam menos atenção por necessitarem de um sistema síncrono para serem implementados [LFA01]. Entretanto, esses detectores são capazes de reduzir as considerações necessárias nos protocolos de alto nível, e também possibilitar o desenvolvimento de protocolos mais eficientes, que não precisam lidar com as suspeitas incorretas dos detectores mais fracos.

Um problema na implementação de detectores de falha é que eles são especificados em termos de um comportamento não perpétuo, ou seja, só após um tempo finito não conhecido é que o detector apresentará um certo comportamento. Essa especificação é adequada para sistemas assíncronos puros, entretanto, aplicações freqüentemente possuem restrições de tempo, ainda que essas restrições não sejam rígidas. Neste caso, tal comportamento não é útil. Um detector de falhas que suspeita de um processo falho após horas, certamente não

[2]O detector de falhas perfeito é o detector mais fraco que resolve o problema da eleição se considerado apenas os detectores definidos por Chandra e Toueg [CT96]. Fetzer e Cristian [FC96] definiram em um novo conjunto de detectores de falhas, os detectores cientes (do inglês *aware*) de falhas, entre eles $\Diamond S^{WF}$, uma classe de detectores mais fracos que P e ainda capazes de resolver o problema da eleição.

será útil em um sistema que realiza diversos consensos por segundo ou em um sistema interativo. Portanto, aplicações podem necessitar de um detector de falhas que dê garantias de qualidade de serviço [CTA00]. Desta forma, a maioria dos trabalhos que implementam um detector de falhas perfeito oferece alguma garantia no tempo máximo de detecção de uma falha [CMV00; Fet03; OBB03].

Enquanto o modelo assíncrono com detectores de falha é baseado no modelo assíncrono puro, o modelo quasi-síncrono [VA95] é baseado no modelo síncrono. O modelo quasi-síncrono foi proposto para atender o crescente número de sistemas distribuídos de tempo-real que possuem uma rigidez não-crítica em relação aos prazos das tarefas. Esses sistemas podem perder alguns prazos desde que os prazos mais importantes não sejam perdidos.

Um sistema síncrono é especificado através dos limites das propriedades de tempo para a entrega de mensagens, para o atraso no escalonamento e limites no desvio máximo dos relógios locais. De forma semelhante, um sistema quasi-síncrono é especificado através de pares formados pelos mesmos limites que caracterizam um sistema síncrono associados às respectivas probabilidades que estes limites não sejam cumpridos. Esta abordagem é útil para sistemas onde os limites no pior caso não são mensuráveis ou são muito além do caso comum e considerá-los resultaria em uma má utilização dos recursos existentes. Como a probabilidade dos prazos não serem cumpridos passa então a não ser desprezível, o sistema deve então possuir planos de contingência para lidar com as possíveis falhas de desempenho.

2.2 Sistemas híbridos

Sistemas híbridos são baseados em sistemas assíncronos (ou parcialmente síncronos) que poderiam, apenas quando necessário, tirar proveito de um recurso escarço, um subsistema síncrono. Esse subsistema síncrono é chamado de *Wormhole* [Ver03] e é simples o suficiente para que sua implementação confiável seja possível. Um *Wormhole* permite que os sistemas deixem de lado as incertezas do ambiente assíncrono durante certos momentos de sua operação onde determinismo é necessário.

Implementar um sistema híbrido, no entanto, requer um sistema síncrono que seja capaz de interagir com um sistema assíncrono. Como na prática sincronismo só pode ser obtido através de acesso controlado, o subsistema síncrono, de capacidade limitada, deve ser capaz

de lidar com as requisições do sistema assíncrono chegando de forma esporádica e até mesmo concorrentes e ainda assim realizar suas tarefas de forma síncrona. O problema oposto também acontece. É possível que o subsistema síncrono produza resultados mais rapidamente do que o sistema assíncrono pode consumir. Em um ambiente de memória limitada, esta situação levará a um esgotamento da memória e uma subseqüente perda de informações.

Além disso, a implementação de um subsistema síncrono depende de certas considerações. Deve haver portanto, um mecanismo que impeça que as propriedades de segurança do sistema sejam quebradas quando essas considerações não se mantiverem. Dependendo do sistema utilizado, o estado seguro pode ser uma parada ou a reiniciação da máquina.

No trabalho apresentado por Fetzer [Fet03], a atividade síncrona, detecção de falhas com semântica perfeita, é realizada utilizando um mecanismo especial de troca de mensagens. Esse mecanismo consegue detectar mensagens que "demoraram demais" para serem entregues, além disso, um cão-de-guarda[3] garante que nós suspeitados realmente falharam. Essa abordagem exige prazos (*deadlines*) mais conservadores, resultando em um tempo máximo de detecção de falhas muito alto (na ordem de alguns segundos no sistema reportado em [Fet03]). Por oferecer um serviço bastante limitado, este trabalho não possui o problema de interface com a parte assíncrona. Isto ocorre porque o serviço disponibilizado, a detecção de falhas, monitora a ocorrência de falhas em um número fixo de processos (3, no protocolo proposto em [Fet03]) e a informação da falha é um valor apenas de leitura que pode ser compartilhado entre todas as aplicações que consultam o serviço. Além disso, do ponto de vista do consumo de informações geradas pela parte síncrona, uma saída do detector de falhas pode ser perdida pois o estado é monotônico, uma suspeita aprendida passa a estar contida em todas as saídas seguintes. Como o detector de falhas monitora um conjunto pré-definido de nós, não há fluxo de informação da parte assíncrona para a parte síncrona.

O trabalho de Oliveira [OBB03; Oli03] é também orientado apenas à detecção de falhas. O Delphus consiste em um detector de falhas com semântica perfeita capaz de monitorar tanto os nós de uma rede local como os processos que executam nestes nós. O Delphus utiliza um cão-de-guarda e uma interface de comunicação exclusiva para troca de mensa-

[3]Cães-de-guarda, do inglês *Watchdogs*, são temporizadores que precisam ser reiniciados periodicamente, caso contrário é assumido que algum evento inesperado aconteceu e ações adequadas devem ser tomadas (por exemplo forçar a máquina a parar.

gens do serviço de detecção. Esta abordagem apresenta a vantagem de possibilitar melhor desempenho que o trabalho anterior, além de ser mais escalável (o protocolo sugerido por Fetzer atende apenas a 3 nós). As informações geradas pela parte síncrona são informações do detector de falhas e por serem monotônicas, podem ser perdidas. Por outro lado, existe um fluxo de informação da parte assíncrona para a parte síncrona. A parte assíncrona pode solicitar a inscrição de novos nós ou processos para serem monitorados pelo serviço.

Na implementação do Delphus proposta por Oliveira [Oli03] essas requisições da parte assíncrona para a parte síncrona são todas direcionadas para um nó líder através da rede assíncrona e quando a inscrição é aceita, a confirmação, junto com as informações da nova entidade monitorada, são transmitidas a todos os nós através da rede síncrona privativa. Nesta implementação, a interface de comunicação para as mensagens do serviço é uma placa de rede padrão adicionada à máquina. O processamento das tarefas do serviço de detecção de falhas, é escalonado a partir de estouros de alarmes no próprio sistema operacional Linux. Essa abordagem impõe que valores muito conservadores sejam escolhidos para os prazos do serviço, uma vez que existem muitas incertezas a serem consideradas, especialmente no escalonamento das tarefas. O desempenho do Delphus pode ser medido pela periodicidade com que cada nó envia seu conjunto de *heartbeats*. Nesta implementação o período varia de $0,784s$ em uma máquina Pentium 1.7 GHz com o tempo entre as interrupções do relógio alterado no núcleo do Linux de $10ms$ para $1ms$ até $1,477s$ em uma máquina Pentium MMX 240 MHz com o núcleo padrão (não alterado) do Linux[Oli03].

O TCB [VC02] possui os serviços de detecção de falhas de desempenho (mais abrangente que o modelo de falhas por parada), medição de intervalo de tempo entre dois eventos distribuídos e execução de tarefas com instantes de início e/ou fim predeterminados. Apesar do TCB, em sua definição, não impor tecnologia para sua implementação, sua única implementação documentada foi realizada utilizando um sistema operacional de tempo-real (RT/Linux), canais de comunicação privativos e *drivers* especiais para as interfaces de comunicação [CMV00]. A interface com a parte assíncrona do sistema é realizada através de um módulo intermediário que aceita as requisições assíncronas e interage de forma determinista com a parte síncrona do sistema. Uma limitação desta interface é que a execução de tarefas em instantes de inicio e/ou fim predeterminados só é possível se o tempo de execução da tarefa a ser executada for previamente e corretamente fornecido (um tempo fornecido

menor que o tempo real necessário para a execução de uma tarefa gera uma falha de desempenho). Além disso, as tarefas devem ser curtas o suficiente de modo que sua execução mais a execução das tarefas de verificação de prazos sejam completamente executadas entre duas interrupções de relógio do RT/Linux. A freqüência das interrupções do relógio no RT/Linux é programada sobre demanda. Esse valor afeta diretamente a variação no escalonamento, quanto menor o valor do intervalo, menor a incerteza do escalonamento e menor o tamanho possível das tarefas. O trabalho de Casimiro *et al.* [CMV00] não faz considerações de desempenho.

Enquanto os trabalhos apresentados por Fetzer e por Oliveira dispõem apenas de um serviço de detecção de falhas, o TCB propõem um conjunto maior de serviços. Entretanto, em nenhum dos três trabalhos um protocolo novo poderia ser implementado para utilizar as garantias do subsistema síncrono de forma diferente da previamente disponibilizada.

Como mencionado anteriormente, existem problemas que não podem ser solucionados em sistemas assíncronos mesmo quando estes são acrescidos de detectores de falhas perfeitos. Charron-bost *et al.* [CBGS00] provaram que um sistema síncrono é mais forte que um sistema assíncrono equipado com um detector de falhas perfeito (P) mesmo em problemas onde o tempo não está envolvido. A prova consiste em mostrar que existem problemas que não envolvem diretamente tempo e que podem ser resolvidos em sistemas síncronos mas não podem ser solucionados em sistemas assíncronos equipados com o detector de falhas.

Informalmente, o problema é o seguinte: (1) considere dois processos *p* e *q*, cada um possui um valor inicial; (2) após um tempo, *q* deve decidir um valor; (3) se *p* era inicialmente correto, o único valor de decisão possível para *q* é o valor inicial de *p*, caso contrário, *q* decide pelo seu próprio valor inicial.

Em um sistema síncrono, o problema pode ser resolvido de forma simples: a partir do instante inicial, *q* aguarda durante D unidades de tempo (onde D é o valor do atraso máximo de comunicação); depois disso, se *q* recebeu a mensagem de *p*, ele decide por ela, caso contrário, ele decide pelo seu próprio valor.

No sistema assíncrono equipado com P não há solução para este problema. Essa impossibilidade existe mesmo que P seja implementado em um subsistema síncrono (de fato, implementar P exige um sistema síncrono [LFA01]) e que, portanto, exista um limite de tempo máximo para que uma falha de *p* seja suspeitada por *q*. Isso acontece pois devido à

incerteza sobre se *p* falhou antes ou depois de enviar a mensagem, não há como *q* decidir se deve ou não esperar essa mensagem (que pode ser atrasada indefinidamente, pois o sistema é assíncrono).

Os resultados apresentados acima defendem a importância de um sistema híbrido pela sua necessidade na implementação de detectores de falhas perfeitos. Entretanto, também pode ser percebido que há um espaço de problemas não resolvidos entre a abstração dos detectores de falhas perfeitos e os sistemas totalmente síncronos.

Capítulo 3

Projeto de um sistema distribuído híbrido

O ambiente de suporte ao desenvolvimento de sistema híbridos proposto por este trabalho visa atender às seguintes necessidades:

1. Pouca intrusão no sistema assíncrono de destino.
2. Garantia que o sistema não se comporta de forma incorreta (ou seja, que o sistema não contraria sua especificação), o que poderia violar as propriedades de segurança dos protocolos implementados sobre ele.
3. Baixo custo de desenvolvimento e de produção.
4. O sistema deve ser escalável, ou seja, deve permitir fácil expansão.
5. Um ambiente síncrono capaz de oferecer suporte à implementação de novas abstrações e protocolos deve ser fornecido.

A lista a seguir analisa como as implementações do TCB, proposta por Casimiro *el al.* [CMV00], do Delphus, proposta por Oliveira [Oli03], e do esquema de licenças e cães-de-guarda, proposta por Fetzer [Fet03], satisfazem os requisitos enumerados acima.

1. O TCB exige a instalação de um sistema operacional de tempo-real, o RT/Linux, e uma rede privativa de comunicação. O Delphus necessita apenas da rede privativa de comunicação. O trabalho de Fetzer utiliza apenas um cão-de-guarda. Dentre eles, a instalação do sistema operacional de tempo-real pode ser considerada como o requisito mais forte, mais intrusivo.

2. Todos eles prevêem um mecanismo de segurança contra falhas de temporização e contra a não contaminação dos outros nós. O trabalho de Fetzer e o Delphus utilizam um cão-de-guarda, o TCB usa uma camada de *software* executando no sistema operacional de tempo-real para verificar a pontualidade das tarefas.

3. Todos eles apresentam um baixo custo de desenvolvimento e produção (utilizando componentes *off-the-shelf*).

4. Os recursos utilizados pelo TCB para implementação dos serviços, a saber, uma rede privativa de comunicação e capacidade de processamento (para execução dos módulos de *software* no sistema operacional de tempo-real), são escaláveis pela atualização dos processadores e das interfaces de rede (*Fast-Ethernet* por *Gigabit Ethernet*, *10-Gigabit Ethernet*, etc.). O Delphus utiliza a rede privativa de comunicação, escalável da mesma forma que o TCB, mas depende do tempo de resposta do escalonador de processos, diretamente ligado ao tempo entre interrupções do relógio de uma máquina Linux. Este tempo não tem diminuído ao longo dos anos[1]. O trabalho de Fetzer não detalha como é feita a implementação do *software*. Caso seja em um sistema operacional de tempo-real, passa a ser mais intrusivo no sistema de destino. Caso seja em um sistema operacional comum, passa a sofrer as mesmas limitações de escalabilidade do Delphus.

5. O TCB disponibiliza um serviço de execução de tarefas com início e/ou fim programados para instantes exatos de tempo. Entretanto, estas tarefas não dispõem de mecanismos para comunicação síncrona, o que impede a implementação, utilizando esse serviço, de qualquer protocolo distribuído que necessite de todos os recursos do subsistema síncrono. O Delphus e o esquema de licenças e cães-de-guarda de Fetzer, implementam apenas o serviço de detecção de falhas e não oferecem nenhum suporte ao desenvolvimento de outras abstrações ou protocolos.

Com base nos requisitos levantados e nas vantagens e limitações de cada uma das três abordagens discutidas, os requisitos podem ser reescritos de forma mais específica.

[1] O tempo entre as interrupções do relógio do Linux é um compromisso entre rapidez de resposta do sistema e sobrecarga do processador com o tratamentos destas interrupções. O valor atual de $10ms$ vem sendo o padrão há vários anos, mesmo com a evolução dos processadores.

Em primeiro lugar, o desenvolvimento de novas tarefas exige a existência de um ambiente onde tarefas possam ser executadas com menos incertezas no escalonamento. Este requisito exige um mecanismo mais preciso de medição de tempo e que este force o escalonamento das tarefas síncronas nos momentos adequados. No nosso caso, as tarefas serão executadas de maneira periódica e a eficiência do *Wormhole* pode ser medida pela duração destes períodos.

Em segundo lugar, as tarefas que implementam as abstrações também necessitam de garantias de limite na comunicação no sistema distribuído. Para isso, é necessária uma rede dedicada para o tráfego das mensagens do serviço ou que a rede de dados possua garantias de qualidade de serviço (o que não é o caso da maioria das redes locais utilizadas nos sistemas reais), de modo que a comunicação síncrona não sofra interferência do tráfego de dados entre as aplicações assíncronas.

Ainda, deve haver uma proteção contra falhas de desempenho no sistema síncrono, que se baseia em considerações que, dependendo das tecnologias em que foi implementado, podem ser bastante fortes. Finalmente, o sistema deve objetivar minimizar as restrições no ambiente onde será instalado.

Para escolher entre as alternativas possíveis para a implementação do *Wormhole*, foram analisadas as alternativas para implementação das duas principais características: a garantia de limite nos atrasos de escalonamento e a garantia de limite de tempo na troca de mensagens. Para isso, os requisitos para cada um deles foram enumerados e confrontados com as tecnologias disponíveis e suas características relevantes. Por fim, as tecnologias mais adequada foram escolhidas através de uma análise comparativa.

3.1 Alternativas de processamento

Assumindo que o sistema assíncrono é composto por um conjunto de computadores executando um sistema operacional Linux e conectados por uma rede local, existem três alternativas principais para fornecer suporte às necessidades de escalonamento dos componentes de *software* do *Wormhole*, são elas:

Kernel Threads em um sistema operacional Linux padrão: implementar os módulos de *software* como *threads* que irão executar dentro do núcleo e com a maior prioridade possível [OBB03; BOB03; Oli03];

Módulos em um sistema operacional RT-Linux: os módulos de *software* serão implementados como módulos que executarão quando escalonados pelo sistema operacional [CMV00];

Dispositivo de hardware dedicado: módulos de *software* serão implementados em um processador dedicado e sua execução não dependerá das atividades do processador de sua máquina associada [Bri03], e os módulos implementados na máquina serão escalonados segundo indicação do dispositivo.

3.1.1 Kernel threads em um sistema operacional Linux padrão

O núcleo (*kernel*) é o sistema operacional de fato, ele controla os recursos da máquina e escalona os processos de usuário para execução. O núcleo do Linux é um programa monolítico e não-preemptivo. Todo o código do núcleo permanece na memória RAM e não sofre *swap* da memória. Além disso, a implementação do padrão POSIX.1b-1993 [POS96; POS03] no Linux define algum suporte para tarefas de tempo-real no Linux padrão. Este suporte permite que tarefas de tempo-real não estritas (*soft real-time*) usufruam de políticas de escalonamento diferentes das utilizadas paras as tarefas comuns, recebendo prioridade sobre estas [BC03]. Implementar os componentes de *software* do *Wormhole* como *threads* de tempo-real permite que estas tenham prioridade sobre todos os processos e também que elas não estejam sujeitas aos atrasos imprevisíveis causados pelo seu *swap* na memória.

Uma desvantagem desta abordagem é que, sendo o núcleo do Linux não-preemptivo, as tarefas de baixa prioridade que estão executando em modo núcleo (por exemplo, quando estão executando uma chamada de sistema) não irão ceder o processador até que retornem ao modo usuário. Desta forma, embora uma *Kernel Thread* tenha prioridade sobre todos os outros processos, mesmo quando ela se torna pronta para executar, não existe garantia de quando exatamente ela será escalonada. Além disso, durante a execução de trechos críticos de código, o Linux desabilita interrupções e novamente, o escalonamento das tarefas de tempo-real e até mesmo o atendimento a outras interrupções serão atrasados por um período indeterminado. Finalmente, quando a tarefa de tempo-real está sendo executada, interrupções podem acontecer e suspender a execução da mesma, atrasando sua execução. Se habilitadas, as interrupções têm prioridade sobre todos os processos ou *threads*, até mesmo

as do núcleo. Elas são causadas por ações externas ou como conseqüência de ações passadas dos processos de baixa prioridade (por exemplo, uma solicitação de leitura de uma unidade de disco bloqueia o processo de baixa prioridade e quando o resultado da leitura está disponível, uma interrupção é gerada), portanto, a sua ocorrência está relacionada com a carga do sistema.

O *Wormhole* requer comunicação, desta forma ele tem que acessar a sua rede através de uma interface de rede, que é um componente de *hardware*. O acesso a um componente de *hardware* pela *thread* irá introduzir outros atrasos, e como o acesso será realizado através de funções do sistema operacional que não são de tempo-real, os atrasos são imprevisíveis.

Enfim, para lidar com as incertezas e garantir um comportamento síncrono, é necessário estimar os cenário para o pior caso. Quanto maior for o número de incertezas, maior será a dificuldade de estimar corretamente este limite superior. Conseqüentemente, limites conservadores precisam ser escolhidos, impactando substancialmente o desempenho. Ainda assim, os limites conservadores podem ser violados em tempo de execução, o que requer algum tipo de plano de contingência para lidar com as falhas de desempenho geradas pelos atrasos. Uma abordagem é utilizar temporizadores de cão-de-guarda em *hardware* ou *software* que observam quando prazos foram perdidos e preservam as propriedades de segurança do sistema. Ambos os tipos de cães-de-guarda são suportados pelo Linux.

Vantagens.

- Não exige requisitos especiais do Sistema Operacional.
- Pouca intrusão na máquina do usuário.
- Não introduz custos adicionais.

Desvantagens.

- A falta de limites no atrasos das funções de processamento e comunicação impõem valores conservadores para os prazos das tarefas do *Wormhole*, o que impacta diretamente o desempenho [Oli03].

- Uma pequena melhora de desempenho poderia ser obtida alterando a freqüência de interrupção do relógio da máquina, mas isso requer uma recompilação do núcleo do sistema operacional.

Dependências. Esta abordagem assume que o sistema, mesmo executando em um sistema operacional comum, irá se comportar de forma síncrona. Como se trata de uma consideração bastante forte, deve haver um plano de contingência, por exemplo um cão-de-guarda, para quando esta consideração não for mantida (*i.e.* quando uma falha de desempenho acontecer). Este cão-de-guarda, seja implementado em *hardware* ou *software*, é um componente extra a ser desenvolvido.

Maior risco. Selecionar esta alternativa pode resultar em tempos de resposta demasiadamente altos e com em uma redução na disponibilidade do sistema que está conectado ao *Wormhole* (devido ao tratamento das falhas de desempenho).

3.1.2 Módulos em um sistema operacional RT-Linux

Um dos maiores problemas na utilização de um sistema operacional Linux padrão na implementação dos módulos de *software* do *Wormhole* é a incerteza no escalonamento. Este problema pode ser resolvido utilizando um sistema operacional de tempo-real. Existem duas abordagens na utilização de um sistema operacional Linux de tempo-real. Uma delas é utilizar uma variante de tempo-real do sistema operacional padrão, o que implica em modificações no Linux padrão de modo a disponibilizar às aplicações certas funcionalidades de tempo-real. A outra abordagem é utilizar o RT/Linux ou RTAI, que são sistemas operacionais simples de tempo-real que controlam a máquina e que executam Linux como uma de suas tarefas [BY97; FSM04; DIA04].

Variantes de tempo-real do Linux permitem que aplicações sejam desenvolvidas como aplicações Linux normais, tirando proveito de um ambiente de programação familiar. Sistemas de arquivos e comunicação em rede podem ser implementados de forma a oferecer certas garantias de tempo. Entretanto, esta abordagem não é capaz de manter-se atualizada com a velocidade com que o Linux padrão muda. Cada versão do Linux tem que ser modificada para implementar as funcionalidades de tempo-real e, além de exigir muita reescrita

de código, não é resistente a mudanças profundas do núcleo. Além disso, o Linux é muito complexo para ser um sistema de tempo-real como um todo.

RT/Linux (Real-Time Linux) [BY97; FSM04] e RTAI (*Real-Time Application Interface*) [DIA04] são sistemas operacionais simples de tempo-real que possuem uma API para executar aplicações de tempo-real e executam o sistema operacional Linux padrão como uma tarefa de tempo-real com baixa prioridade. Esses sistemas intermedeiam o Linux e o *hardware* e são capazes de tomar o controle da CPU do Linux, mesmo quando este está executando em modo núcleo, sempre que as tarefas de tempo-real precisam ser escalonadas. A tarefa que executa o Linux desabilitar as interrupções também não impede o núcleo de tempo-real de ter suas interrupções tratadas ou que as tarefas de tempo-real sejam escalonadas.

A utilização de um RT-Linux possui algumas vantagens como o determinismo no processamento, garantia de tratamento das interrupções e pouca intrusão no Linux padrão. Entretanto, as tarefas de tempo-real, não são aplicações Linux normais e também não existe, previamente implementado, sistemas de E/S de tempo-real (como sistemas de arquivos ou comunicação em rede). Desta forma, *drivers* de dispositivos para permitir a comunicação síncrona devem ser desenvolvidos. Existem ainda alguns casos que até mesmo as tarefas de tempo-real podem ser interrompidas, por exemplo: dispositivos podem, de forma autônoma, iniciar uma transferência por DMA (*Direct Memory Access*) dos seus *buffers* para a memória principal. Estas transferências têm prioridade sobre o processador no acesso ao barramento da máquina. Além disso, dispositivos de *hardware* podem gerar interrupções que serão tratadas pelo sistema de tempo-real, atrasando a execução das atividades de tempo-real.

Uma grande desvantagem prática dessa alternativa é a instalação do sistema operacional de tempo-real no sistema que irá utilizar o Wormhole.

Vantagens.

- Menos fatores influem na incerteza de escalonamento do Linux padrão.
- Não introduz custos adicionais.

Desvantagens.

- Tarefas de tempo-real não são aplicações Linux normais.

- Necessidade de instalação de um RT-Linux.
- Utilização da camada de tempo-real do sistema operacional introduz carga no sistema.
- Tarefas de tempo-real ainda podem ser afetadas por interrupções e transferências por DMA.
- Comunicação síncrona requer que os *drivers* sejam implementados.

Dependências. RT-Linux não implementa *drivers* de rede para comunicação em tempo-real, necessitando o esforço adicional de implementação. Esta abordagem assume que os módulos de tempo-real executando em uma máquina tipicamente assíncrona irão se comportar de forma síncrona. Como se trata de uma consideração forte, deve haver um plano de contingência, por exemplo um cão-de-guarda, para quando esta consideração não for mantida (*i.e.* quando uma falha de desempenho acontecer). Este cão-de-guarda, seja implementado em *hardware* ou *software*, é um componente extra a ser desenvolvido.

Maior risco. A necessidade da instalação de um sistema operacional de tempo-real nos sistemas que utilizarão o *Wormhole* pode ser um requisito forte demais.

3.1.3 Dispositivo de *hardware* dedicado em um sistema Linux padrão

Um dispositivo de *hardware* poderia ser capaz de executar algumas das tarefas do *Wormhole* e obrigar (através de interrupções) a execução dos módulos de *software* do *Wormhole* na máquina. O dispositivo de *hardware* consiste em um processador, uma interface com sua máquina associada e uma interface de comunicação. A utilização de um dispositivo dedicado possibilita que a execução das tarefas de tempo-real associadas a alguns módulos de *software* do *Wormhole* sejam completamente isoladas, sem sofrer qualquer interferência das tarefas que executam na máquina associada. Outras tarefas, por exemplo, serviços e abstrações implementados como aplicações na máquina teriam seu escalonamento programado através de interrupções.

Para executar serviços para a parte assíncrona, o sistema síncrono precisa ter uma interface com a mesma. O problema de fazer uma interface entre um sistema síncrono de

capacidade limitada com requisições assíncronas chegando de forma esporádica é um problema encontrado em qualquer uma das abordagens especificadas nas seções anteriores. Esta interface pode ser implementada como um intermediário localizado na parte assíncrona que recebe requisições das tarefas que estão executando na parte assíncrona. O dispositivo síncrono se comunica com o intermediário através de interrupções de *hardware*, o dispositivo dita quando ele pode receber um conjunto de requisições, independentemente do que ocorre na parte assíncrona. Além disso, o dispositivo pode ser conectado diretamente a uma interface de rede tornando possível comunicação em tempo-real, sem qualquer interferência do sistema assíncrono.

Utilizar um processador dedicado alivia ainda mais o *Wormhole* das dificuldades de escalonamento existentes na parte assíncrona do sistema. Interrupções de alta prioridade e transferências por DMA ainda podem atrasar as ações da parte síncrona. Entretanto, a interferência esperada é bastante menor que os atrasos imprevisíveis de escalonamento do Linux comum e o dispositivo percebe quando foi atrasado. O dispositivo por sua vez, sendo síncrono por construção, pode observar quando esses tempos excedem as expectativas e assegurar as propriedades de segurança do sistema através de ações de contingência (por exemplo, forçando um falha silenciosa). A comunicação síncrona irá ocorrer através de funções do próprio dispositivo que introduzirão atrasos limitados e conhecidos.

Vantagens.

- Não impõe requisitos ao sistema operacional do usuário.
- Facilita o desenvolvimento de um *Wormhole* para sistemas operacionais diferentes de Linux.
- Um cão-de-guarda para preservar as propriedades de segurança do sistema quando as considerações não se mantiverem (por exemplo, quando uma falha de desempenho ocorrer) pode ser facilmente implementado.
- Suporte fácil à comunicação em tempo-real.

Desvantagens.

- Requer a utilização de um dispositivo de *hardware* personalizado.

- Requer uma porta de comunicação livre (por exemplo, uma porta serial ou USB, dependendo da implementação).

Dependências. Esta alternativa requer a implementação de um dispositivo de *hardware* personalizado. Esta abordagem assume que a comunicação entre a parte síncrona e a parte assíncrona irá ser possível periodicamente. Para garantir a segurança, deve haver um plano de contingência, por exemplo um cão-de-guarda, para quando esta consideração não for mantida (*i.e.* quando uma falha de desempenho acontecer). Entretanto, um cão-de-guarda pode ser facilmente implementado em um processador dedicado.

Maior risco. A implementação do projeto do dispositivo de *hardware* exige processos de montagem não triviais.

3.1.4 Análise comparativa entre as alternativas de processamento

De acordo com os requisitos dos módulos de *software* do *Wormhole*, embora a segurança possa ser garantida pelo cão-de-guarda, o sistema baseado em Linux padrão não daria um desempenho vantajoso. Isso ocorre porque existem incertezas demais a serem consideradas [Oli03]. RT/Linux resolve a maioria dos problemas do Linux padrão, mas é uma camada separada que executará sob o Linux padrão, adicionando uma carga considerável à máquina original. RT/Linux também necessita da implementação de drivers de comunicação na medida que sua comunicação é manipulada pelo Linux padrão. Por fim, a utilização de um dispositivo de *hardware* dedicado resolve a maioria dos problemas mas introduz a desvantagem da utilização de um dispositivo de *hardware* personalizado. Entretanto, este dispositivo é bastante simples (consistindo apenas de um processador com duas interfaces de comunicação, uma para a máquina e outra para os outros dispositivos) e pode ser construído para ser conectado através de uma porta serial ou USB, permitindo a sua conexão a máquinas já em funcionamento. O *driver* necessário para utilização do dispositivo de *hardware* é bastante simples (mais simples que um *driver* de mouse) e pode ser carregado como um módulo do Linux padrão. Além disso, o custo adicional do dispositivo é desprezível.

Com base nos dados resumidos na Tabela 3.1 e na discussão acima, a escolha para a tecnologia de processamento do *Wormhole* ficou entre a utilização de um RT-Linux e um

dispositivo de *hardware* dedicado conectado a cada máquina. Por fim, o dispositivo dedicado foi escolhido pela menor interferência na máquina (comparado a instalação do RT-Linux) e a melhor portabilidade para outros sistemas.

	Linux padrão	RT Linux	Dispositivo dedicado
Sensibilidade à carga assíncrona	Atrasos no escalonamento, suspensão da execução, interface	Suspensão da execução, interface	Interface dispositivo/PC
Comunicação síncrona	Não oferece garantias	Apenas se os drivers de comunicação forem implementados	Diretamente pelo dispositivo
Segurança	Através de cão-de-guarda em *hardware* ou *software*	Através de cão-de-guarda em *hardware* ou *software*	Através de cão-de-guarda em *hardware* ou *software*, que podem ser facilmente implementados no processador dedicado
Interferência no sistema original	Nenhuma	Instalação do sistema operacional de tempo-real	Nenhuma
Requisitos extras	Cão-de-guarda em *hardware* opcional	Cão-de-guarda em *hardware* opcional	O dispositivo de *hardware*
Custo	0	0	Em torno de US$ 15

Tabela 3.1: Resumo das alternativas de processamento

3.2 Alternativas de rede

Uma vez escolhida a alternativa de processamento, é necessário escolher a tecnologia para implementar a rede de comunicação. Existe um grande número de tecnologias de rede dis-

poníveis, algumas são orientadas a redes de informações (como negócios e redes de computadores pessoais) e outras são orientadas a redes de controle (como redes industriais de sensores, atuadores, CLPs). As redes orientadas a controle são normalmente chamadas de *fieldbus*. De forma a facilitar a escolha da tecnologia mais apropriada, os principais requisitos do *Wormhole* foram enumerados:

- **Velocidade:** Os nós que compõem o *Wormhole* precisam trocar pequenos volumes de informação como *heartbeats* e mensagens de controle. A comunicação é periódica e baseada em difusões, em cada período cada nó envia uma mensagem de algumas dezenas de bytes de comprimento. Uma rede contém poucas dezenas de nós e o período deve estar na ordem de dezenas de milissegundos.

- **Sincronismo:** O atraso de entrega de uma mensagem deve ser limitado.

- **Multi-mestre:** Determinismo na comunicação é facilmente obtido utilizando-se de um nó para controlar a comunicação dos outros nós. O *Wormhole* necessita de uma arquitetura que considere a possibilidade de falha do nó mestre, permitindo a continuação do serviço sem que os compromissos sejam afetados.

- **Alcance:** Os nós estão conectados através de uma rede local, o que significa um alcance de aproximadamente 100 metros. A rede privativa do *Wormhole* deve ter o mesmo comprimento.

- **Confiabilidade:** A perda de mensagens representa um perigo grave para o correto funcionamento do sistema.

Das tecnologias disponíveis, cujas aplicações tinham semelhanças com o *Wormhole* e amplamente aceitas e aplicadas, foram analisadas: ControlNet, Interbus, Profibus, Ethernet and CAN[2].

[2]As primeiras soluções avaliadas foram as de rede sem fio dada a vantagem prática de não requerer o recabeamento na adição de uma nova rede. Entretanto, elas foram descartadas pois a sua utilização em controle e aplicações de alta confiabilidade ainda está nos primeiros estágios de desenvolvimento. Bluetooth e padrões baseados no 802.11 não satisfazem nossas necessidades pois eles não escalam ou não possibilitam a comunicação síncrona. Bluetooth [Bha01; Blu04a; Blu04b], por exemplo, possibilita apenas 8 nós em uma rede (chamada piconet). Em outros padrões baseados em 802.11, mesmo protocolos baseados em *polling* têm

3.2.1 CAN

CAN (*Control Area Network*) foi desenvolvida pela Bosch em 1986 para utilização em dispositivos automotivos inteligentes [Rob04]. Tem sido utilizada como uma interface entre vários componentes digitais em um automóvel, tais como a injeção eletrônica e a caixa de marchas eletrônica: tarefas críticas com relação ao tempo que requerem um canal de comunicação de tempo-real. Suas características básicas são:

- **Velocidade:** 1 Mbps (CAN 2.0).
- **Cabeamento:** 2 fios.
- **Número máximo de nós:** 120.
- **Tamanho máximo da rede:** 30 metros (1 Mbps) a 1000 metros (62.5 Kbps).
- **Tamanho de mensagem:** 0 a 8 bytes.
- **Aplicações:** Comunicação com sensores e atuadores; embora desenvolvida para uso automotivo passou a ser amplamente utilizada em aplicações de automação.
- **Organização de suporte:** CAN in Automation - www.can-cia.de.
- **Camada física:** CSMA/CD+AMP (*Carrier Sense Multiple Access with Collision Detection with Arbitration on Message Priority*).
- **Confiabilidade:** Mensagens prioritárias sobrevivem a colisões.
- **Abertura do padrão:** Grande número de fornecedores.
- **Desenvolvedor:** Bosch GmbH.

desempenho limitado [Wil03]. Em geral, tecnologias baseadas em rádio sofrem da alta incerteza dos canais de rádio, mesmo em ambientes controlados [GW02]. Foram realizados experimentos utilizados módulos de rádio comerciais [Tex04b; Rad03] na construção de um sistema proprietário utilizando faixas ISM (*Industrial Scientific Medical*) para avaliar o impacto das altas taxas de erro associadas à comunicação por rádio no nosso ambiente. A conclusão foi que não seria possível obter a confiabilidade necessária ao mesmo tempo que um desempenho satisfatório.

- **Implementação:** Através de módulos prontos e ASICs (*Application Specific Integrated Circuits*).
- **Custo:** Em torno de US$ 2,50 (Bosch).

CAN não é uma rede de acesso controlado, mas usa CSMA/CD acrescido de um mecanismo de arbitragem por prioridade de mensagens. Este mecanismo garante que em caso de colisão, a mensagem com maior prioridade, o que é sinalizada pelo identificador mais baixo, ganha o barramento. As mensagens que perdem o barramento na colisão são automaticamente retransmitidas.

Existem diversas tecnologias derivadas da rede CAN, como DeviceNet, SDS e CANopen. Essas tecnologias utilizam das funcionalidades primitivas da rede CAN para oferecer serviços de mais alto nível (por exemplo, fragmentação automática de pacotes grandes).

Vantagens.

- Multi-mestre.
- Difusões.
- Amplamente difundida e suportada.
- Colisões não-destrutivas (para a mensagem prioritária).

Desvantagens.

- Velocidade cai com a distância.
- Mensagens muito curtas.

Observações. A tecnologia CAN tem sido largamente utilizada em sistemas embarcados e tem robustez para ser aplicada em comunicação de tempo-real. Existem ASICs de baixo custo que implementam a tecnologia e até mesmo processadores (microcontroladores) com suporte incluso. Sua utilização na implementação do *Wormhole* impõe a fragmentação de mensagens, além disso para uma distância de 100 metros sua velocidade máxima cai para cerca de 500 Kbps.

3.2.2 ControlNet

ControlNet é uma rede *fieldbus* de alto nível e alta velocidade desenvolvida pela Allen Bradley em 1995 [Con04]. Esta tecnologia foi desenvolvida para comunicação de dados de alto nível entre módulos de *hardware* complexos. É conhecida pela sua redundância e determinismo enquanto ainda permite que todos os nós iniciem um acesso ao canal. Suas características básicas são:

- **Velocidade:** 5 Mbps.
- **Cabeamento:** RG6/U (Conector BNC gêmeo redundante, cabo coaxial).
- **Número máximo de nós:** 99.
- **Tamanho máximo da rede:** 250 a 5000 metros (com repetidores).
- **Tamanho de mensagem:** 0 a 512 bytes.
- **Aplicações:** Tarefas críticas, rede de PC de abrangência de planta, redes de CLPs (Controladores Lógicos Programáveis) e subredes, redes de controle de processos industriais, e situações requerindo alta velocidade de transporte tanto para dados críticos de entrada e saída, como mensagem de dados.
- **Organização de suporte:** ControlNet Internacional - www.controlnet.org.
- **Camada física:** Acesso múltiplo concorrente por domínio de tempo (CTDMA - *Concurrent Time Domain Multiple Access*).
- **Confiabilidade:** Duplo caminho (cabeamento) para redundância pré-construída.
- **Abertura do padrão:** Poucos fornecedores.
- **Desenvolvedor:** Allen Bradley.
- **Implementação:** Através de módulos prontos e ASICs.
- **Custo:** Implementação do protocolo (circuito integrado) em torno de US$ 30 + Transceptor em torno de US$ 10.

O acesso à rede é controlado por um algoritmo chamado Acesso Múltiplo Concorrente por Domínio de Tempo (CTDMA), que controla a oportunidade dos nós de transmitir em cada intervalo de rede. A configuração de quão freqüente ocorre a repetição do intervalo de rede é ajustada a partir do parâmetro Intervalo de Atualização de Rede (NUI - *Network Update Interval*). O NUI mais rápido possível é de 2 ms. Cada nó tem um MAC ID, um identificador único. Todos os outros parâmetros de configuração são enviados periodicamente no canal.

Cada NUI (período) tem três partes, a primeira parte é a parte de tempo-real, chamada de parte escalonada, a segunda parte é a parte de *melhor-esforço*, chamada parte não escalonada, e por último, a parte de configuração de rede, chamada de *bandguard*. Na parte de tempo-real cada nó pré-alocou a largura de banda desejada e por isso possui uma fatia de tempo específica para suas transmissões de dados. Na parte de *melhor-esforço*, um nó pode ter várias oportunidades (fatias) de transmissão ou nenhuma, de acordo com a carga de rede. Na parte *bandguard*, o nó moderador, um nó mestre, re-sincroniza todos os nós e transmite os parâmetros de configuração da rede.

Não há necessidade para um mecanismo de manipulação de colisões em ControlNet, já que se trata de uma rede sincronizada por tempo onde cada nó sabe o instante de tempo exato onde pode transmitir. Quando um nó moderador falha, o nó com o próximo identificador mais alto assume a responsabilidade de moderador, ocorrendo portanto, uma eleição automática de líder. O novo líder passa a transmitir as informações de sincronização e os parâmetros de configuração.

Vantagens.

- Alta velocidade.
- Determinismo.
- Redundância natural.
- Uso eficiente da largura de banda da rede.
- Difusões.

Desvantagens.

- Poucos fornecedores e aplicações.
- Alto custo (do *hardware* e de desenvolvimento).

Observações. A ControlNet Internacional vende kits de desenvolvimento que contêm o ASIC que implementa o protocolo e os *drivers* para acessar o meio físico. Existe pouca literatura sobre aplicações práticas dessa tecnologia em redes confiáveis de informação, e também de sua implementação em dispositivos inteligentes simples. A ControlNet Internacional foi contactada a respeito de informações adicionais sobre o desenvolvimento de projetos usando essa tecnologia, mas nenhuma resposta foi recebida.

A tecnologia ControlNet possui todas as características técnicas para implementação de uma rede síncrona, especialmente o determinismo e a largura de banda. Entretanto, é uma tecnologia proprietária, com poucos fornecedores e existe pouca indicação que seja apropriada para dispositivos simples.

3.2.3 Ethernet

Ethernet foi concebida e implementada nos laboratórios de pesquisa da Xerox em 1973 [MB76] e é a tecnologia mais utilizada em redes de informação. Suas características básicas são:

- **Velocidade:** 10 ou 100 Mbps (utilizados em sistemas embarcados, mas padrões de até 10 Gbps já foram desenvolvidos).
- **Cabeamento:** Coaxial (10Base2 e 10Base5) ou par trançado de 4 fios (10BaseT e 100BaseTX).
- **Número máximo de nós:** depende do cabeamento, 30 (10Base2), 100 (10Base5), 1024 (10BaseT, 100BaseTX); sem limites com a utilização de repetidores.
- **Tamanho máximo da rede:** 100 metros do ponto central (hub) para 10BaseT e 100BaseTX, 200 metros para 10Base2 e 500 metros para 10Base5; sem limites com a utilização de repetidores.

- **Tamanho de mensagem:** 46 a 1500 bytes.
- **Aplicações:** Redes de informações; recentemente, cada vez mais redes de controle estão usando Ethernet acrescida ou não de protocolos especiais de alto nível.
- **Organização de suporte:** Industrial Ethernet Organization - www.industrialethernet.com, IAONA - www.iaona-eu.com.
- **Camada física:** *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD).
- **Confiabilidade:** Apenas um mecanismo de verificação de erros (CRC32)
- **Abertura do padrão:** Grande número de fornecedores.
- **Desenvolvedor:** Xerox.
- **Implementação:** Através de módulos prontos e ASICs.
- **Custo:** Cerca de US$ 6 (Realtek, 10 Mbps).

O acesso ao meio físico utilizado pela rede Ethernet é o CSMA/CD, que consiste em ouvir o barramento antes de transmitir, e caso dois nós escutem um barramento ao mesmo tempo e decidam transmitir ao mesmo tempo uma colisão acontecerá. Quando a colisão acontece, o dispositivo Ethernet aguarda um tempo e retransmite a mensagem. Caso uma nova colisão ocorra ele espera um novo período de tempo e retransmite, o tempo de espera cresce exponencialmente. Este esquema faz com que a rede Ethernet seja não determinista no baixo nível, onde uma mensagem pode ser atrasada por um longo tempo (se comparado com o tempo de transmissão médio).

Novos padrões foram propostos para solucionar o problema do não determinismo [Le 87; IEE]. Os padrões IEEE 802.3p e 802.3q incluem prioridades nas mensagens e têm sido amplamente implementados nos comutadores de rede utilizados em redes Ethernet 10BaseT e 100BaseTX, os mais utilizados atualmente. A utilização de comutadores, em conjunto com o tráfego em duplo sentido (*full duplex*), eliminam as colisões criando barramentos diferentes para cada dispositivo Ethernet conectado a uma de suas portas.

Vantagens.

- Multi-mestre.
- Difusões.
- Crescente tendência em utilizar Ethernet para dispositivos inteligentes e redes de controle.

Desvantagens.

- É não determinista no baixo nível, e portanto, requer a utilização de protocolos de mais alto nível para intermediar a comunicação (implementando determinismo e confiabilidade).
- Tamanho mínimo de mensagem impõe grande *overhead* para mensagens pequenas.

Observações. Sendo a mais popular tecnologia de rede, existe uma larga literatura sobre a implementação de redes Ethernet em sistemas embarcados. Além disso, existem kits de desenvolvimento, módulos e ASICs de diversos fornecedores (Realtek, Cirrus Logic, National Semiconductors, ASIX, etc.), vários deles com interface de desenvolvimento totalmente compatíveis (NE2000-compatível).

Além da vantagem de múltiplos fornecedores, Ethernet beneficia-se da produção em larga escala, reduzindo o custo. Sua velocidade é maior que a maioria das redes de controle. Determinismo e confiabilidade podem ser obtidos através de protocolos simples de alto nível, como por exemplo um mecanismo de controle de acesso por ficha (implícita ou explícita) [OBB03; BOB03; Oli03].

Por fim, existe uma tendência de utilizar Ethernet em ambientes industriais e redes de controle de modo a obter o benefício de velocidade, custo e homogeneidade de redes entre os ambientes de controle e informação[vN02].

3.2.4 Interbus

Interbus é caracterizada como uma rede *fieldbus* de "alta velocidade e máximo diagnóstico" e foi desenvolvida pela Phoenix Contact em 1984 [Int04]. As aplicações típicas do Interbus in-

cluem redes de sensores e atuadores e controle de sistemas de produção. Suas características básicas são:

- **Velocidade:** 500 Kbps.
- **Cabeamento:** 4 fios (RS-485).
- **Número máximo de nós:** 256.
- **Tamanho máximo da rede:** 12,8 Km (máximo de 400 metros entre os nós para cabeamento de cobre).
- **Tamanho de mensagem:** 512 bytes de recebimento e 512 bytes de envio por ciclo.
- **Aplicações:** Cabeamento único para múltiplos sensores, válvulas, leitores de código de barra, atuadores e interfaces de operação.
- **Organização de suporte:** The Interbus Club - www.interbusclub.com.
- **Camada física:** Cada nó recebe e repassa os dados, não competindo pelo canal.
- **Confiabilidade:** Apenas um mecanismo de verificação de erros (CRC16).
- **Abertura do padrão:** Vários fornecedores.
- **Desenvolvedor:** Phoenix Contact.
- **Implementação:** Módulos conversores de protocolo e ASICs.
- **Custo:** US$ 350 (módulo conversor de protocolo).

Interbus é uma rede em anel onde cada nó possui dois conectores, um de entrada e um de saída. Um quadro é recebido na entrada e é então repassado para o nó seguinte. A comunicação é realizada utilizando-se um único quadro, onde cada nó preenche na sua célula, o espaço destinado a suas mensagens que serão transmitidas. No recebimento, cada nó verifica sua célula de recebimento, onde estarão quaisquer mensagens destinadas a ele. O quadro é criado e destruído apenas por um nó mestre previamente definido.

Se por um lado o método de comunicação em anel aumenta o determinismo, a existência de apenas um mestre que possui um comportamento e implementação diferente dos escravos

cria um ponto único de falhas. Se o mestre falha, nenhuma comunicação pode acontecer entre os escravos. Além disso, a topologia em anel faz com que uma conexão falha desabilite toda a rede.

Vantagens.

- Auto-endereçamento.
- Amplamente utilizada.
- Pouco *overhead*.
- Rede pode fornecer também energia aos dispositivos.

Desvantagens.

- Uma conexão falha desabilita toda a rede.
- Apenas um nó mestre.

Observações A implementação do Interbus em um dispositivo simples não é uma tarefa trivial. Existem módulos de conversão de protocolo, interfaces de rede para PCs e ASICs que implementam o protocolo. Entretanto muito pouca informação sobre a utilização destes ASICs está disponível. A Phoenix International foi contactada para obtenção de maiores informações sobre a utilização e cotação dos ASICs, mas nenhuma resposta foi recebida.

3.2.5 Profibus

Profibus é a rede de controle mais instalada no mundo [Pro04]. Foi desenvolvida na Alemanha em 1989 por um consórcio de várias empresas. É composta por três camadas: Profibus PA, o mais baixo nível, orientada para automação de processos, permitindo que sensores e atuadores sejam conectados no mesmo barramento, mesmo em áreas hostis; Profibus DP, para comunicação entre sistemas de controle de automação e Entrada/Saída distribuída para dispositivos inteligentes; e Profibus FMS, para comunicação entre plantas, com topologias complexas e alta velocidade.

- **Velocidade:** de 9,6 Kbps a 12 Mbps (dependendo do padrão).
- **Cabeamento:** 2 fios (RS-485).
- **Número máximo de nós:** 127 com repetidores ou 32 sem (limitação da RS-485).
- **Tamanho máximo da rede:** 12 Km para cabeamento de cobre (distância máxima entre nós de 400 metros) e 24 Km para fibra ótica.
- **Tamanho de mensagem:** até 244 bytes.
- **Aplicações:** Existem três padrões, que permitem a utilização do Profibus desde redes pequenas de sensores até redes complexas de informação e controle.
- **Organização de suporte:** Profibus International - www.profibus.org.
- **Camada física:** Controle de acesso ao meio por ficha rotatória.
- **Confiabilidade:** Apenas um mecanismo de verificação de erros (CRC).
- **Abertura do padrão:** Vários equipamentos compatíveis, mas poucos fornecedores para o desenvolvedor.
- **Desenvolvedor:** Profibus Consortium.
- **Implementação:** Poucos ASICs e módulos complexos.
- **Custo:** US$ 30 (Siemens).

Profibus é uma rede com um arquitetura multi-mestre. Qualquer nó mestre pode iniciar uma comunicação e o controle de acesso ao canal é realizado através de um esquema de fichas. Nós escravos podem existir, mas não podem iniciar uma comunicação. Existem dois tipos de nós mestres: os mestres da classe 1, que podem apenas iniciar comunicação para seus nós escravos; e os mestres da classe 2, que podem iniciar comunicação entre seus escravos e entre outros mestres.

Vantagens.

- Comunicação em tempo-real para os padrões Profibus PA e Profibus DP.
- Difusão.

Desvantagens.

- Alto custo do ASIC e do desenvolvimento (devido ao pouco suporte).
- *Overhead* para mensagens pequenas.
- Uma falha pode causar a perda da ficha rotativa.

Observações Profibus é uma tecnologia cara e complexa que dispõe de pouca documentação sobre a utilização dos ASICs no desenvolvimento. O padrão de mais alto nível, o Profibus FMS, capaz de uma velocidade de até 12 Mbps, não possui comunicação de tempo-real.

3.2.6 Análise comparativa entre as alternativas de comunicação

Considerando os requisitos do *Wormhole* enumerados anteriormente e as características da tecnologias resumidas na Tabela 3.2, a decisão foi feita entre CAN e Ethernet.

Ambas as tecnologias são largamente difundidas, beneficiam da economia de escala e são amplamente empregadas em sistemas embarcados. Embora CAN tenha um grau maior de determinismo, por possuir um esquema de colisões não destrutivas, seria necessário um protocolo de mais alto nível que implemente a fragmentação de mensagens e o determinismo necessário (um esquema TDMA, por exemplo, que também seria necessário para a tecnologia Ethernet). Por fim, Ethernet foi escolhida pelas seguintes razões:

- Alta velocidade e possível porte para outros padrões Ethernet (Fast-Ethernet, Gigabit Ethernet, 10-Gigabit Ethernet).
- O tamanho das mensagens permite uma posterior adição de serviços ao *Wormhole* (sem a necessidade de fragmentar mensagens).
- Comutadores de rede e fluxo bidirecional (*full-duplex*) podem aliviar o problema das colisões, e como a carga da rede será bem conhecida e limitada a perda de mensagens também pode ser contornada.
- Conectar sistemas embarcados a Ethernet é uma tendência para as aplicações industriais e de controle [vN02].

- Fácil desenvolvimento, depuração e integração com PCs equipados com as placas Ethernet-padrão.

Alternativa	Vantagens	Desvantagens	Custo do ASIC	Avaliação
CAN	Amplamente utilizado, implementação simples, baixo custo, colisões não destrutivas	Mensagens pequenas, apesar de possuir prioridade de mensagens ainda é não determinista (sem controle de acesso)	US$ 2,50	Mensagens curtas impõem fragmentação, ainda requer um protocolo de alto nível para implementar determinismo
ControlNet	Alto determinismo, alta velocidade	Complexidade de implementação, alto custo	US$ 40	Caro, aceitação limitada, complexidade
Ethernet	**Amplamente Utilizado, implementação simples, alta velocidade**	**Determinismo precisa ser implementado em protocolo de camada mais alta**	**US$ 6**	**Facilmente escalável, tendência para aplicação em redes de controle, economia de escala, homogeneidade**
Interbus	Alto determinismo	Uma conexão falha desabilita toda a rede	US$ 350 (módulo completo de conversão)	Alta vulnerabilidade
Profibus	Determinismo, velocidade	Complexidade de implementação, alto custo, risco de perda da ficha	US$ 30	Complexidade de implementação, padrão mais rápido não é determinista

Tabela 3.2: Resumo das alternativas de rede

3.3 Considerações

De acordo com as decisões tomadas com relação às tecnologias a serem empregadas, o sistema passa a ser formado por um conjunto de máquinas interligadas por uma rede local de acesso livre, e um conjunto de dispositivos conectados a essas máquinas e interligados por um rede local privativa. O conjunto de máquinas e sua rede local compreendem o sistema distribuído assíncrono, onde o sistema operacional e as aplicações do usuário executam e trocam mensagens. O conjunto de dispositivos, suas redes privativas e os seus *drivers* (em execução nas máquinas associadas) consistem no subsistema síncrono, *i.e.* o *Wormhole*, acessível apenas a determinados serviços especiais.

Os dispositivos, máquinas associadas e suas redes estão ilustrados na Figura 3.1, onde um nó é constituído de um dispositivo e uma máquina associada.

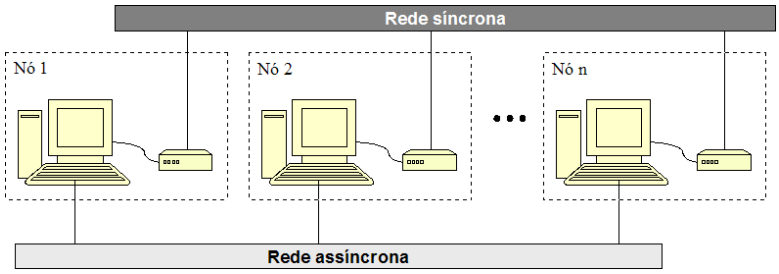

Figura 3.1: Arquitetura básica

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 todos os nós transmitem. 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. Na prática, essas características podem ser obtidas através de alguns mecanismos simples como códigos de detecção de erros, retransmissão de mensagens e associação de identificadores únicos para essas mensagens.Assumindo-se que

na rede síncrona não há falhas devido aos conectores, ao cabeamento ou ao concentrador (no caso de uma rede padrão 10BaseT), não ocorrem particionamentos na rede. Para uma rede Ethernet de acesso controlado, esta não é uma consideração forte.

Quanto ao processamento, o dispositivo é síncrono por construção. Além disso, ele é capaz de perceber quando os *drivers* executando no PC sofreram 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 falhar por parada.

3.4 Interface do *hardware*

O dispositivo é construído de forma que a parte de *software* do *Wormhole* tenha alguns recursos para a implementação de serviços que executem de maneira síncrona, esses recursos são:

- **Interrupção programada**: Os serviços a serem executados pelo *Wormhole* devem ser escalonados em um período constante. Como a percepção do tempo pelo PC é deficiente devido às várias fontes de eventos e incertezas, o dispositivo será o responsável por sinalizar os momentos que os serviços precisam ser escalonados.

- **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. Este recurso é a base do *Wormhole* e é necessário para a implementação de quaisquer serviços síncronos. O envio de mensagens é realizado enviando-se um pacote de comprimento fixo pela porta serial.

- **Cão-de-guarda**: Um cão-de-guarda é configurado no dispositivo para que, quando a porção assíncrona do sistema sofre uma falha de desempenho ou de parada, ela seja colocada em um estado seguro e não comprometa as propriedade de segurança do *Wormhole*.

3.5 Interface do *software*

A porção de *software* do *Wormhole* consiste em módulos carregáveis do sistema operacional Linux [RC01; BC03]. 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 ou arquivos especiais. Estes serviços são os seguintes:

- Relógio Global;
- Serviço de detecção de falhas 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 a lista atual dos identificadores dos nós que fazem parte 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 de comunicação. Este valor corresponde ao intervalo de tempo entre duas execuções de um serviço, ao tempo que uma mensagem enviada pela rede síncrona pode demorar a ser processada no nó destino e ao tempo de detecção máximo do serviço de detecção de falhas.

Os serviços básicos são detalhados a seguir.

Relógio global sincronizado. As mensagens periódicas no *Wormhole* são utilizadas na construção de um relógio global. Este relógio pode ser acessado a partir da função exportada para o núcleo:

- $current\_time \Leftarrow get\_global\_time()$: retorna o valor do relógio global.

O serviço de Detecção de Falhas. O serviço de detecção de falhas identifica quais 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 uma camada intermediária 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 o relógio global têm como resultado um valor monotônico e somente de leitura, que pode ser compartilhado por todos as aplicações interessadas. Entretanto, existirão serviços e protocolos carregados dinamicamente. Estes serviços utilizam a largura de banda do *Wormhole* para comunicar-se com suas outras instâncias em outros nós da rede síncrona. Desta forma, a quantidade de serviços dinâmicos carregados deve ser limitada. Esta limitação depende da largura de banda da rede e do intervalo de tempo disponível para execução dos serviços (duração do período de comunicação). A iniciação destes serviços dinâmicos é gerida pelo controle de admissão. Como a largura de 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:

- $return \Leftarrow wh\_subscribe\_service(name, periodicity, bandwidth, handler)$: cadastra um serviço na lista de serviços localmente disponíveis em um nó do *Wormhole*. 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*.

- $handler \Leftarrow wh\_request\_service(name, periodicity, service\_parameters, participants, quorum)$: faz uma difusão na rede síncrona, alocando uma porção do canal síncrono em todos os nós para um serviço e carregando esse serviço. Retorna um apontador para a função do serviço (os parâmetros necessários devem ser conhecidos). O parâmetro $quorum$ correponde ao número mínimo de participantes para que o serviço seja iniciado. Para alguns serviços, a aplicação que solicita a iniciação, pode modificar sua periodicidade padrão (parâmetro $periodicity$).

- $wh\_remove\_service(name)$: encerra a execução de um serviço.

Capítulo 4

Implementação de um sistema distribuído híbrido

4.1 Seleção dos componentes de hardware

Uma vez escolhida a utilização de um dispositivo de *hardware* dedicado para implementar os módulos de *software* do *Wormhole* e escolhida a Ethernet como a tecnologia de rede, podemos escolher o processador e o controlador Ethernet a ser usado.

4.1.1 O processador dedicado

O processador dedicado será responsável pelas seguintes tarefas: controlar o acesso à rede (ao controlador Ethernet), o que exige um determinado número de entradas e saídas; implementar o cão-de-guarda para ações de contingência quando as considerações feitas pelo *Wormhole* não se mantiverem; formatar mensagens enviadas pelo seu *driver*, executando em sua máquina associada, para serem transmitidas na Ethernet privada; enviar informações do sistema síncrono, como por exemplo, as mensagens recebidas pelo controlador Ethernet, para o seu *driver*.

Para estes propósitos não existe necessidade para um processador complexo, que exija a implementação de um sistema digital completo composto de RAM, memória não volátil, controladores de interface (por exemplo, serial RS-232, paralela, USB), temporizadores, etc. Uma alternativa seria a utilização de microcontroladores, que integram um processador de

baixo desempenho (normalmente de 4 a 20 MHz), temporizadores, pequenas quantidades de memória volátil e não volátil, além de alguns periféricos diversos (por exemplo, portas seriais, cães-de-guarda). Uma outra alternativa seria a utilização de FPGAs (*Field Programmable Gate Arrays*). FPGAs possuem um conjunto de módulos e portas lógicas não conectadas. Estes componentes podem ser interligados a partir de um *software* de configuração, fazendo com que eles implementem a funcionalidade lógica desejada.

Comparando a utilização de FPGAs e microcontroladores, os FPGAs são capazes de um maior desempenho que os microcontroladores, entretanto os microcontroladores apresentam uma série de vantagens que os tornam mais adequados à sua utilização neste trabalho: melhor custo/benefício para o baixo poder de processamento necessário; menor quantidade de componentes do sistema digital; facilidade de projeto e de mudança de projeto (e requisitos); rapidez de implementação; ferramentas de desenvolvimento de baixo custo.

Uma vez desenvolvido o sistema e comprovadas suas vantagens, um sistema de melhor desempenho poderia ser implementado com FPGAs. Então, em troca de maiores recursos de desenvolvimento (ferramentas de desenvolvimento, mão-de-obra), o FPGA substituiria tanto o microcontrolador quanto o controlador Ethernet.

Entre um grande número de micro-controladores disponíveis, escolhemos o MSP430 da Texas Intruments [Tex04a], mais especificamente, o modelo MSP430F149 pelas seguintes razões:

- CPU de 16 bits, contra as CPUs de 4 e 8 bits, o que facilita a utilização de números de 8, 16 e 32 bits.
- 48 entradas/saídas, que são mais que suficiente para conectar o controlador Ethernet e quaisquer periféricos que venham a ser utilizados.
- Ferramentas de desenvolvimento (compilador C e kit de desenvolvimento e programação) de qualidade e de custo acessível.
- Qualidade do suporte técnico e da documentação disponibilizada.

4.1.2 O controlador Ethernet

Uma das vantagens da tecnologia Ethernet, como discutido nas seções anteriores, é a disponibilidade de vários ASICs que implementam um controlador Ethernet. Alguns dos controladores Ethernet mais utilizados em sistemas embarcados estão listados na Tabela 4.1.

	RTL8019As	CS8900A	AX88796
Fabricante	Realtek	Cirrus Logic	ASIX
Preço	US$ 35	US$ 50	US$ 55
NE2000-compatível	Sim	Não	Sim
Velocidade	10 Mbps	10 Mbps	10/100 Mbps
8 bits/16 bits	Sim/Sim	Sem interrupções/Sim	Sim/Sim
DMA	Sim	Sim	Sim
Buffer	8K*16 bits	4,5K*8 bits	8 KB*16 bits
Complexidade SW	Simples (NE2000)	Complexa	Simples (NE2000)
3V/5V	Não/Sim	Sim/Sim	Sim/Não

Tabela 4.1: Principais módulos com controladores Ethernet disponíveis

O preço dos componentes na Tabela 4.1 corresponde a um pequeno módulo com o controlador, alguns componentes externos obrigatórios, o transformador/conector RJ-45 (10BaseT e 100BaseTx) e um conjunto de pinos para sua soldagem em uma placa de circuito impresso[1]. É preferível utilizar os que trabalham com uma velocidade de 10 Mbps por três razões: (1) um controlador de 100 Mbps exige um processador mais poderoso e complexo; (2) o tamanho das mensagens do serviço é pequena (tempo de resposta é mais importante que largura de banda e 10 Mbps já supera em muito as outras tecnologias avaliadas); (3) a comunicação com o PC será realizada através da porta serial ou USB do PC, o que limita o desempenho do sistema.

[1] A utilização de módulos é indicada por dois motivos principais: (1) o controlador Ethernet é um circuito integrado compacto e com pinos muito próximos que são montados com soldagem em superfície; (2) o controlador exige alguns componentes passivos (resistores, capacitores) de precisão que são difíceis de se comprar em pequenas quantidades.

Estes controladores são os mais utilizados em sistemas embarcados por possuírem algumas características especiais: altamente integrados, exigindo apenas alguns poucos componentes externos; buffer de memória interno, permitindo que processadores menos poderosos possam lidar com a alta velocidade do barramento Ethernet; funcionamento em modo de 8 bits, que simplifica a montagem do *hardware* e desenvolvimento dos *drivers*. Finalmente, foi escolhido o RTL8019AS pelo seu custo, funcionamento de 8-bits, quantidade de memória no buffer e por ser NE-2000[2] compatível, usufruindo de larga documentação e de outros controladores compatíveis (incluindo controladores de 100 Mbps).

4.2 Implementação do hardware

Os principais componentes do *hardware* estão ilustrados na Figura 4.1 (um diagrama detalhado do circuito elétrico pode ser encontrado na Figura 4.2). O módulo Maxim MAX232 (1) converte as tensões de trabalho do microcontrolador (2) para as tensões de trabalho da porta serial do PC (padrão RS232). O microcontrolador utilizado foi o MSP430F449 (2) da Texas Instruments [Tex04a] (o modelo MSP430F1491, mais simples, poderia ter sido utilizado, mas não estava disponível). Por fim, o controlador Ethernet (3) está conectado ao microcontrolador e ao barramento Ethernet.

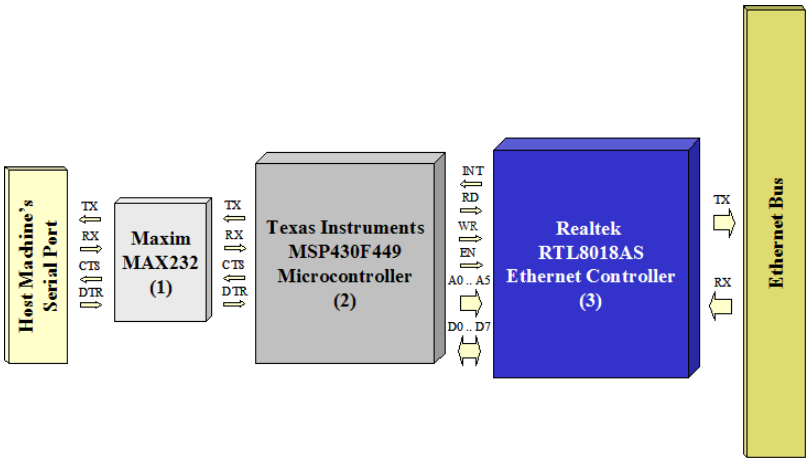

Figura 4.1: Diagrama dos módulos do dispositivo

A conexão do MAX232 ao microcontrolador é composta por quatro sinais: o sinal de

[2]Controlador Ethernet padrão derivado do chip 8390 da National Semiconductors.

dados do PC para o dispositivo (RX), o sinal de dados do dispositivo para o PC (TX), o sinal de controle de fluxo na comunicação do dispositivo para o PC (DTR) e o sinal de controle de fluxo na comunicação do PC para o dispositivo (CTS). O sinal de controle de fluxo na comunicação do dispositivo para o PC é controlado pelo *hardware* da porta serial do PC que reconhece quando os caracteres foram lidos pelo *driver*. O sinal de controle de fluxo na comunicação do PC para o dispositivo é controlado pelo *software* embarcado do dispositivo. Esse controle de fluxo é necessário para que nenhum dos dois lados perca caracteres devido a uma lentidão do *driver* executando no PC ou pela rapidez de envio da máquina para o dispositivo.

O controle de fluxo pode atrasar a comunicação, criando uma dependência entre a velocidade de execução do *driver* na máquina e o tempo de comunicação. Como o tempo de comunicação precisa ser limitado, uma demora excessiva na comunicação, ou seja, uma falha de desempenho, causaria uma pertubação no comportamento síncrono do dispositivo. Desta forma, uma falha de desempenho na comunicação serial é transformada em uma falha por parada do PC e do dispositivo. Essa falha é percebida através de um alarme programado no dispositivo.

A comunicação com o controlador Ethernet é um pouco mais complexa. O controlador funciona como uma memória, na qual os pacotes a serem enviados são copiados, e do qual os pacotes recebidos são copiados. Conforme ilustrado na Figura 4.3, o controlador possui 8 pinos de comunicação de dados (no modo 8-bits), 5 pinos de endereçamento, 3 pinos para seleção da direção do barramento de dados e habilitação do controlador e 1 pino de interrupção. Internamente, o RTL8019AS possui um conjunto de registradores e uma área de memória volátil. O RTL8019AS é controlado a partir desses registradores que são alcançados selecionando o endereço correspondente no barramento de endereços nos pinos $A0$ a $A4$. Então, o valor do registrador pode ser lido ou escrito, de acordo com o valor dos pinos de direção (*READ*, *WRITE*, *ENABLE*), no barramento de dados (pinos $D0$ a $D7$).

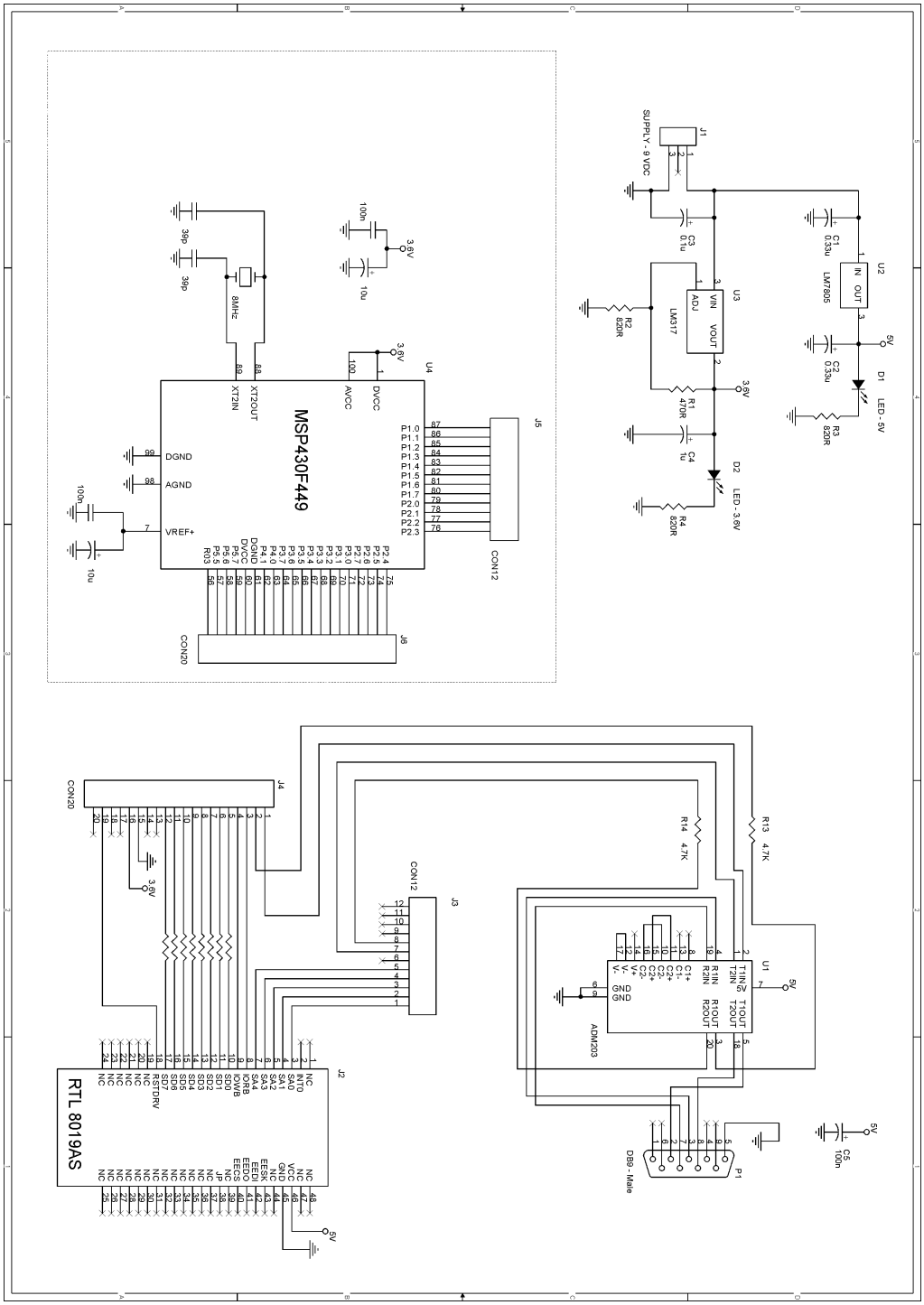

Figura 4.2: Circuito elétrico do dispositivo

O protótipo do dispositivo, chamado de *Wormhole Interface Board* (WIB), é composto de uma placa de circuito impresso personalizada e um kit de desenvolvimento MSP430P440 da Texas Instruments [Tex04a]. A placa personalizada contém o controlador Ethernet, o conver-

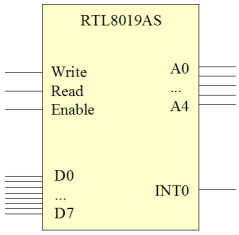

Figura 4.3: Controlador Ethernet RTL8019AS - pinos utilizados

sor MAX232 e alguns componentes discretos. O MSP430P440 contém o microcontrolador e um conector para programação do microcontrollador. Foram montados três protótipos, um deles mostrado na Figura 4.4.

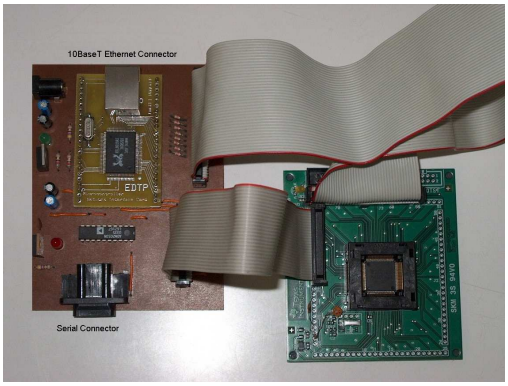

Figura 4.4: Protótipo do dispositivo

4.3 Implementação do *software* embarcado

O *software* para o microcontrolador foi desenvolvido utilizando dois compiladores C, o CROSSWORKS [Row04] e o QUADRAVOX AQ430 [QA04]. Ambos se mostraram bastante eficientes na compilação e geração do código *Assembler* além de oferecerem depuração em circuito (*in-circuit*). Estes mecanismos de depuração permitem a execução e depuração do código desenvolvido já nas placas, ao mesmo tempo que os registradores e memória podem ser inspecionados. As ferramentas são quase que totalmente compatíveis. Mais de 99% do código desenvolvido em uma executava na outra e vice-versa. Foi gerado também uma documentação estilo JAVADOC utilizando a ferramenta Doxygen [van04].

De forma geral, o *software* embarcado possui dois comportamentos distintos, o comportamento de líder e o de escravo. Durante a iniciação, se não há um líder na rede síncrona o nó entra no modo líder, caso contrário, o nó se inscreve na rede e, sendo o processo de inscrição bem sucedido, entra no modo escravo. Uma vez iniciado, o comportamento do nó permanece o mesmo até que ele saia do sistema ou detecte que deve se tornar líder (por exemplo, porque o líder corrente falhou). Estes comportamentos são ilustrados na Figura 4.5.

Um nó escravo realiza os seguintes passos: (A.1) espera a mensagem de sincronização do líder; (A.2) recebe as mensagens de dados dos nós que o antecedem na ordem de transmissão; (A.3) transmite sua própria mensagem; (A.4) recebe as mensagens de dados dos nós que o sucedem na ordem de transmissão; (A.5) comunica-se com o PC para enviar-lhe as mensagens recebidas na rede síncrona e recuperar a próxima mensagem de dados a ser transmitida.

Por outro lado, o líder realiza os seguintes passos: (B.1) transmite a mensagens de sincronização; (B.2) transmite a mensagens de dados; (B.3) espera uma mensagem de inscrição (ou iniciação de serviço); (B.4) recebe as mensagens de dados dos outros nós; (B.5) comunica-se com o PC para enviar-lhe as mensagens recebidas na rede síncrona e recuperar a próxima mensagem de dados a ser transmitida. Em ambos os tipos de nó, se uma falha de desempenho ocorrer, o dispositivo força uma falha por parada em todo o nó.

O *software* tem dois componentes principais: o *driver* Ethernet e um protocolo TDMA que transforma a rede Ethernet em uma rede síncrona. Para utilizar o controlador Ethernet, foi desenvolvido um *driver* simplificado no microcontolador com as seguintes funções: iniciar o controlador, transmitir e receber mensagens. A implementação da rede síncrona será detalhada em seguida.

4.3.1 Implementação do *driver* de rede embarcado

A iniciação do controlador é realizada escrevendo valores de configuração em seus registradores (endereçados pelos barramentos de endereço e escritos pelo barramento de dados). Basicamente, a iniciação consiste em ajustar o controlador para operar no modo de 8 bits, desabilitar interrupções, ativar os buffers de transmissão e recepção, desabilitar as retransmissões em caso de colisão e executar operações padrão de configuração inicial e auto-teste.

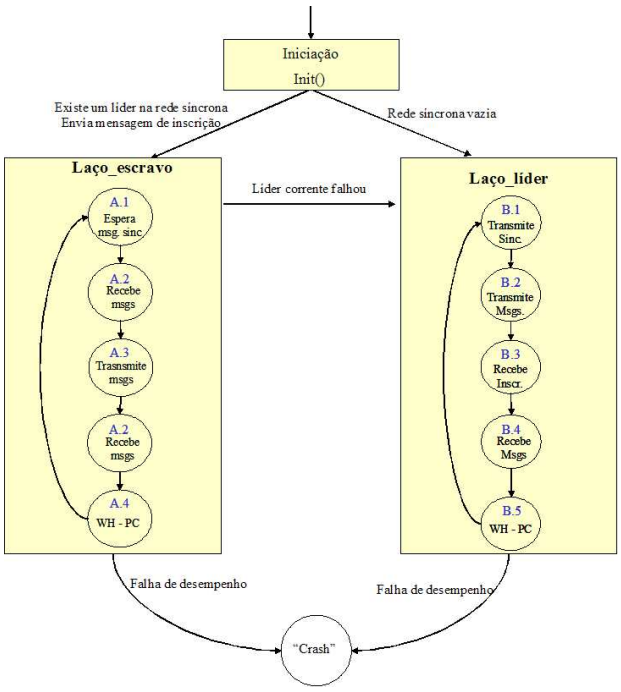

Figura 4.5: Comportamento dos dispositivos

Um envio de mensagem, depois de habilitado o controlador, acontece da seguinte forma: (1) prepara-se, através dos registradores, uma transferência de um determinado número de bytes do microcontrolador para o controlador; (2) o primeiro byte é colocado no barramento de dados; (3) um pulso é dado no pino de escrita; (4) o controlador automaticamente incrementa o endereço de memória e passa a aguardar o segundo byte, esse procedimento continua até que toda a mensagem seja transferida; (5) o comando de envio é dado a partir dos registradores.

Durante o processo descrito acima, uma interrupção pode ser lançada quando a transferência da mensagem do microcontrolador para o controlador é terminada, quando a transmissão na rede é concluída ou quando algum erro acontece. A ocorrência ou não de cada uma dessas interrupções pode ser configurada. Como o funcionamento do dispositivo é baseado na passagem de tempo e não na ocorrência de eventos, as interrupções geradas pelo controlador não geram interrupções no microcontrolador. Neste elas são verificadas através de consultas ao pino correspondente no microcontrolador (*polling*).

No recebimento de uma mensagem acontece o processo inverso. Quando a mensagem é recebida na rede, ela é colocada na memória do controlador e uma interrupção é lançada. O

microcontrolador pode então realizar a transferência da mensagem da memória do controlador para a sua. Novamente, como o sistema é digirido pela passagem de tempo e não pelos eventos externos, o controlador fica desativado e é ativado apenas durante os intervalos de tempo em que uma mensagem é esperada. Os momentos exatos em que isso acontece serão detalhados a seguir.

Maiores detalhes sobre o funcionamento do controlador Ethernet podem ser obtidos no seu manual de referência [Rea03].

4.3.2 Rede de comunicação síncrona

Utilizar a tecnologia Ethernet traz os benefícios de uma tecnologia padrão, como a economia de escala e a vasta documentação disponível, além das altas velocidade e a facilidade de implementação através de módulos prontos. Embora o mecanismo de acesso ao meio das redes Ethernet seja essencialmente não determinista na presença de colisões, neste trabalho colisões são evitadas através de acesso controlado ao meio de transmissão. Este acesso controlado é determinado por um protocolo TDMA (*Time Division Multiple Access*) implementado nos microcontroladores. Isso permite que mesmo que a rede seja estruturada em forma de barramento, como as redes Ethernet 10Base-2 e 10Base-5, colisões não devem acontecer. Nas redes em forma de estrela (10BaseT), com um concentrador (*hub*) como ponto central, formas adicionais de evitar colisões são a conexão de cada nó a uma porta de um concentrador comutado (*switched hub*) e a comunicação em duplo sentido (*full-duplex*).

Divisão do TDMA e tipos de mensagens

A rede síncrona tem um líder; este líder determina quando cada nó pode acessar o meio de comunicação. O protocolo TDMA é baseado em períodos de comprimento fixo que são divididos em fatias. A organização do período TDMA pode ser vista na Figura 4.6 e é composta por quatro tipos de fatias de tempo: fatia de sincronização, fatia de inscrição, fatia de transmissão e fatia de comunicação.

A *fatia de sincronização* é atribuída ao líder e é nela onde é enviada a *mensagem de sincronização* que sinaliza o início de um período. A mensagem de sincronização contém todos os parâmetros de configuração da rede, informações sobre o líder, as atribuições das

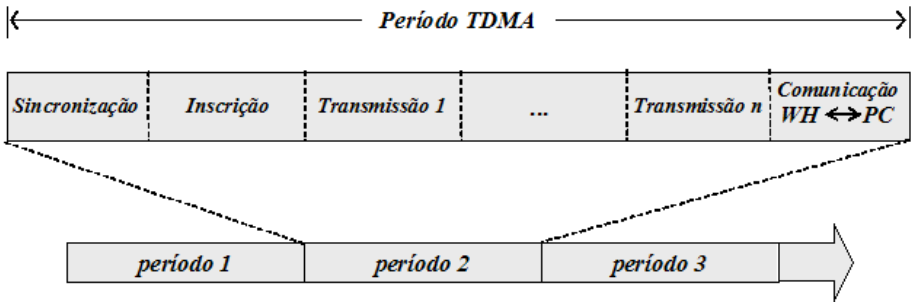

Figura 4.6: Organização do período TDMA

fatias de tempo do período TDMA e finalmente, uma mensagem de dados (enviada pelo *driver* executando no PC associado ao líder).

Os parâmetros de configuração são repetidos periodicamente para que um nó que deseje inscrever-se na rede possa configurar-se corretamente. As informações sobre o líder são necessárias para execução dos serviços que dependem da rede assíncrona. Na atribuição das fatias estão listados os nós que terão acesso a uma fatia de transmissão no período TDMA. Esta seqüência é repetida periodicamente para permitir alocação/remoção de fatias na medida que nós são inscritos/removidos. Outro campo presente na mensagem de sincronização é a mensagem enviada pelo *driver*, ou seja, a mensagem do serviço de comunicação com garantia de tempo da máquina associada ao dispositivo líder. Este campo é enviado, mesmo que a máquina não tenha nenhuma informação a ser transmitida. Neste caso a mensagem é preenchida com zeros. Por fim, há o campo de erro de inscrição; este campo sinaliza um erro na tentativa de algum nó se inscrever na rede síncrona. Um erro pode ocorrer em duas situações: quando a rede está completa (o número de fatias no período TDMA é limitado), que é sinalizado através do código de erro NETWORK_FULL_ERROR, ou quando um serviço está sendo iniciado e as inscrições estão suspensas, que é sinalizado através do código de erro SUBSCRIPTION_SUSPENDED. A Tabela 4.2 detalha os campos presentes na mensagem de sincronização.

Após a fatia de sincronização vem a *fatia de inscrição*. A fatia de inscrição pode conter uma *mensagem de inscrição*, enviada por um nó que ainda não possui uma fatia de transmissão alocada no período TDMA. O formato da mensagem de inscrição é detalhado na Tabela 4.3.

Ainda na fatia de inscrição, podem ser enviadas três outros tipos de mensagens: *mensa-*

Campo	Tamanho	Função
Tipo	1 byte	Tipo da mensagem.
Id	1 byte	Identificador único na rede para o nó.
IP	4 bytes	Endereço IP da máquina associada ao nó líder.
Relógio	4 bytes	Sincronização do valor do relógio global.
Duração do período	1 byte	(T_p) Parâmetro de configuração da rede. Duração de um período TDMA, em centenas de microssegundos.
Duração da fatia de comunicação	1 byte	(T_{PC}) Duração da fatia de comunicação com o PC, em centenas de microssegundos.
Número máximo de nós	1 byte	Número máximo de nós.
Mensagem	60 bytes	Mensagem do serviço de comunicação com garantia de tempo a ser enviada pela rede síncrona.
Fatias TDMA	31 bytes	Lista de identificadores dos nós com acesso às fatias de comunicação do período TDMA.
Erro inscrição	2 bytes	Sinalizar um erro durante uma inscrição, o primeiro byte corresponde ao identificador do nó que tentou inscrever-se e o segundo ao código do erro.

Tabela 4.2: Formato da mensagem de sincronização

gens de iniciação de serviço, *mensagens de confirmação de serviço* e *mensagens de negação de serviço*. Essas mensagens têm o objetivo de carregar os serviços disponibilizados pelo *Wormhole* nos vários nós pertencentes à rede. Essas mensagens possuem o formato detalhado na Tabela 4.4.

A mensagem de iniciação do serviço envia o código do serviço desejado e a lista de participantes. Um nó incluso na lista da mensagem de iniciação, pode enviar uma mensagem de confirmação, que simplesmente repete a mensagem de iniciação de serviço alterando o seu tipo para o tipo de confirmação e substituindo o identificador do solicitante pelo seu próprio. Um nó pode também rejeitar a participação em um serviço, repetindo a mensagem de iniciação de serviço alterando o seu tipo para o tipo de rejeição e substituindo o identificador do

Campo	Tamanho	Função
Tipo	1 byte	Tipo da mensagem.
Id	1 byte	Identificador único na rede para o nó.
IP	4 bytes	Endereço IP da máquina associada ao nó.

Tabela 4.3: Formato da mensagem de inscrição/remoção

Campo	Tamanho	Função
Tipo	1 byte	Tipo da mensagem.
Id	1 byte	Identificador do iniciador.
Serviço	10 bytes	Código de identificação do serviço.
Participantes	32 bytes	Mapa de bits indicando os participantes envolvidos
Quorum	1 byte	Número mínimo de participantes para que o serviço seja carregado.
Posição	1 byte	Utilizado na confirmação de participação. Sinaliza em que posição da mensagem síncrona estarão as mensagens daquele serviço.

Tabela 4.4: Formato da mensagem de iniciação/confirmação/negação de serviço

solicitante pelo seu próprio.

O terceiro tipo de fatia é a *fatia de transmissão*. O número de fatias de transmissão é o parâmetro que determina a duração máxima de um período, e conseqüentemente o tempo de resposta do sistema síncrono. A posse das fatias de transmissão é determinada pela ordem na lista dos nós com acesso às fatias de comunicação transmitida na mensagem de sincronização. Uma fatia de transmissão pode conter três tipos de mensagens: uma mensagem de transmissão, uma mensagem de remoção e uma mensagem especial de líder.

Uma *mensagem de transmissão* e uma *mensagem especial de líder* possuem o mesmo formato, detalhado na Tabela 4.5.

Uma mensagem de transmissão é uma mensagem enviada por um nó, contendo a mensagem do serviço de comunicação com garantias de tempo. Uma mensagem especial de líder é enviada quando acontece uma falha do líder, e sinaliza que o nó que enviou esta mensagem

Campo	Tamanho	Função
Tipo	1 byte	Tipo da mensagem.
Id	1 byte	Identificador único na rede para o nó.
Mensagem	60 bytes	Mensagem do serviço de comunicação com garantia de tempo a ser enviada pela rede síncrona.

Tabela 4.5: Formato da mensagem de transmissão e mensagem especial de líder

assumirá o papel do líder no próximo período. O mecanismo de tolerância a falhas do líder será detalhado mais adiante.

Uma mensagem de remoção é enviada por um nó que deseja abdicar de sua fatia de transmissão no período TDMA. Seu formato é o mesmo da mensagem de inscrição detalhada na Tabela 4.3.

O quarto tipo de fatia é a *fatia de comunicação*. A fatia de comunicação é uma fatia vaga e corresponde ao tempo reservado para que cada dispositivo envie para a máquina a qual está conectado as mensagens recebidas de outros nós e também, que recupere todas as mensagens que serão enviadas na rede síncrona no próximo período (incluindo alguma mensagem de iniciação, remoção ou confirmação de serviço, etc.).

Por fim, existe um último tipo de mensagem que não pertence a nenhuma fatia do período TDMA. Ela é enviada enquanto o período TDMA ainda não foi estabelecido e não existe ainda um líder eleito. Esta mensagem é a *mensagem de proposta de liderança*, ela é enviada quando um nó é ligado e não existe nenhum outro nó, e portanto nenhum líder, na rede. O formato da mensagem de proposta de liderança é detalhado na Tabela 4.6 e sua utilização será discutida nas seções seguintes.

Campo	Tamanho	Função
Tipo	1 byte	Tipo da mensagem.
Id	1 byte	Identificador único na rede para o nó.
IP	4 bytes	Endereço IP da máquina associada.

Tabela 4.6: Formato da mensagem de proposta de liderança

4.3.3 Implementação do TDMA

Iniciação

Quando um nó é ligado e seu dispositivo iniciado, o dispositivo escuta o canal síncrono por um tempo igual à duração máxima de um período TDMA para verificar se algum nó já está presente na rede. No caso afirmativo ele tenta inscrever-se na rede. Caso nenhum nó exista ainda na rede síncrona, este primeiro nó envia uma *mensagem de proposta de liderança*, propondo que ele se torne o líder da rede síncrona.

Como outro nó pode ter sido ligado no mesmo instante que ele, esse nó pode ter realizado o mesmo procedimento e suas mensagens de proposta de liderança podem ter colidido na rede. Desta forma, logo após a transmissão da mensagem de proposta de liderança é verificado se não ocorreu uma colisão e também se não foi recebida uma proposta de liderança de um outro nó. Caso nenhuma das duas situações tenha acontecido[3] o nó se considera líder e envia uma mensagem de líder marcando o início do período TDMA. Deste momento em diante, o nó passa a enviar mensagens de líder periodicamente, marcando o início de cada período TDMA. Caso uma colisão ou o recebimento de uma outra proposta de liderança tenha acontecido o nó dormirá por um tempo aleatório, e quando acordar, ele reiniciará o procedimento[4].

Uma iniciação bem sucedida termina com a identificação do líder corrente. As máquinas conectadas aos dispositivos não precisam ter conhecimento se o seu dispositivo é o não o líder da rede síncrona, apenas precisam conhecer o IP do líder atual.

Inscrição e remoção

Caso inicialmente exista um ou mais nós ligados e utilizando a rede síncrona, o processo de iniciação irá identificar o líder corrente. O nó que deseja se inscrever na rede aguarda uma mensagem de sincronização, verifica se as inscrições estão habilitadas e logo após, na fatia de inscrição, envia sua mensagem de inscrição.

[3] A verificação se uma mensagem foi recebida logo após a transmissão bem sucedida da mensagem de proposta de liderança se aplica no caso de uma rede 10BaseT com concentradores comutados e comunicação bidirecional onde duas mensagens podem ser transmitidas simultaneamente sem colidirem.

[4] Neste sistema a geração de números aleatórios é baseada no sensor interno de temperatura do microcontrolador.

Como vários nós podem estar tentando se inscrever simultaneamente na rede, colisões podem acontecer. Desta forma, logo após o envio da mensagem de inscrição deve ser verificado se ouve uma colisão. Caso uma colisão tenha acontecido, o nó deverá esperar por um número aleatório de períodos antes de tentar inscrever-se novamente. Caso nenhuma colisão tenha acontecido, o nó passa a escutar o canal síncrono esperando que o líder, durante a sua mensagem de líder, confirme sua inscrição e atribua-lhe uma fatia de transmissão ou sinalize um erro de inscrição.

Quanto à remoção, quando um nó não transmite na sua fatia de transmissão, no próximo período ele é automaticamente removido pelo líder. De outra forma, o nó pode solicitar sua remoção transmitindo uma solicitação de remoção durante sua própria fatia de comunicação.

Falha do líder

Quando uma mensagem de sincronização é recebida pelos nós, um procedimento de resincronização é acionado. Neste procedimento, eles programam um de seus alarmes internos (A_1) para que dispare no início da sua fatia de transmissão, ainda no período corrente, cuja posição é extraída também da mensagem de sincronização. Por segurança, o dispositivo programa um outro alarme (A_2), que estourará na mesma fatia do período seguinte. Se, no início do período seguinte, o dispositivo recebe uma nova mensagem de sincronização, o dispositivo programará novamente os alarmes A_1 e A_2. Desta forma, no funcionamento normal do sistema o alarme A_2 é sempre reprogramado antes que dispare.

Quando um dos alarmes estoura, o dispositivo é acordado indicando o início de sua fatia de transmissão. O nó verifica então qual dos alarmes disparou: caso tenha sido o alarme A_1 o nó continua no seu funcionamento padrão, transmitindo sua mensagem e voltando a dormir; caso tenha sido o alarme A_2, uma mensagem de sincronização não foi recebida durante este período, o que significa que o líder falhou.

O nó que acorda em sua fatia de transmissão e percebe que o líder falhou envia uma mensagem especial de líder. Esta mensagem contém as mesmas informações que uma mensagem de transmissão, mas o seu tipo sinaliza que o líder atual falhou e que já não enviou a mensagem de sincronização do período corrente. No próximo período o nó que enviou a mensagem especial de líder irá se transformar em líder e passará a enviar mensagens de sincronização.

Como o primeiro nó que acorda em um período percebe a falha do líder e envia uma mensagem especial de líder, os outros nós tratam a mensagem especial de líder como uma mensagem de sincronização: eles se sincronizam com o nó que enviou esta mensagem, mantendo seu alarme que indica o início de sua fatia de transmissão neste período; mas atualizando o alarme que indica o início da sua fatia de transmissão no próximo período, para isto, consideram que a ordem das fatias não será alterada. Esse procedimento é importante para prevenir a falta de ressincronização dos nós no caso de faltas sucessivas de mensagens de sincronização. Esta situação ocorre quando após a falha do líder um nó manda uma mensagem especial de líder mas falha antes de enviar sua primeira mensagem de sincronização.

Controle de admissão

O controle de admissão corresponde à iniciação ou remoção de um serviço secundário do *Wormhole*. Para exemplificar o processo, uma solicitação para a iniciação de um serviço começa com uma chamada à função $wh\_request\_service()$, implementada na parte assíncrona do serviço de controle de admissão. Na próxima vez que o *driver* se comunicar com o dispositivo, uma mensagem de iniciação de serviço será entregue para ser enviada no próximo período. Desta forma, no período p_1 uma mensagem de iniciação de serviço será enviada pelo dispositivo durante a fatia de inscrição contendo o identificador do serviço (id), a lista de n nós convidados a participar ($\{n_x, n_y, n_z, ...\}$) e o número mínimo de participantes m. No período seguinte, p_2, a fatia de inscrição é vaga pois o primeiro participante ainda não teve a oportunidade de analisar a proposta de iniciação do serviço (veja detalhes mais adiante). Nos n períodos seguintes, de p_3 a p_{n+2}, cada um dos nós convidados a participar irá confirmar ou negar, também durante a fatia de inscrição, sua participação no serviço a ser carregado.

Desta forma, no fim do período p_1, todos os nós informam ao *driver* o identificador do serviço solicitado e ao fim do período p_2 o *driver* responde se o serviço está disponível e se há banda suficiente na rede síncrona. Esta resposta acontece ao fim do período p_2, pois como ela depende de informações localizadas no *driver* que ainda não estão disponíveis ao fim do período p_1. Portanto, no período p_3, o primeiro nó da lista deve enviar uma confirmação ou uma negação de sua participação do serviço a ser carregado. Um nó pode negar a iniciação do serviço por já possuir toda a banda da rede síncrona alocada a outros serviços ou por

não possuir aquele serviço específico instalado. Se depois de n períodos pelo menos m nós confirmarem a participação, todos percebem que o serviço será iniciado no período seguinte. Uma vez decidido que o serviço será iniciado, isto é sinalizado para o *driver* executando na máquina associada e este inicia o serviço. Então, a partir do próximo período a função correspondente ao serviço será executada, recebendo como entrada um *buffer* de mensagens recebidas e retornando um buffer de mensagens a serem enviadas no próximo período.

Uma situação especial que precisa ser considerada neste processo é que durante a iniciação do serviço, um nó que ainda não faz parte da rede síncrona pode tentar inscrever-se e para isso transmitir na fatia de inscrição. Esta situação é evitada sinalizando que as inscrições estão suspensas durante a fatia de sincronização (ver Seção 4.3.2). Esse procedimento evita também que dois serviços sejam iniciados simultaneamente e interfiram nas mensagens de confirmação um do outro.

Para evitar que atrasos imprevisíveis aconteçam quando serviços devem ser iniciados, eles precisam ser previamente carregados na memória. Para isso o serviço deve ser implementado como um módulo do Linux, que passa a ser parte do núcleo e por isso não sofre *swap* da memória. Desta forma, o módulo que implementa um serviço deve, durante sua iniciação, cadastrá-lo junto ao *Wormhole* (o que pode ser feito através da função $wh\_subscribe\_service()$).

4.4 Driver do *Wormhole*

O *driver* é um módulo do núcleo que, para interagir com o dispositivo, captura as interrupções da porta serial, e para interagir com as aplicações, exporta um conjunto de funções para o núcleo do sistema operacional. Através dessas funções exportadas, o núcleo ou outros módulos podem invocar as funções do *Wormhole*.

Uma vez carregado, o *driver* inicia o dispositivo e aguarda até que ele termine o processo de iniciação e comece a interromper a máquina periodicamente com o conjunto de mensagens que circularam na rede síncrona (no mínimo sua própria mensagem de sincronização). Quando a máquina é interrompida, quatro passos são realizados pelo *driver*.

1. Recebe um conjunto de mensagens do dispositivo, vindas do *Wormhole*.

2. Envia uma mensagem para o dispositivo para ser enviada pelo *Wormhole*. Esta mensagem contém pequenas mensagens a serem enviadas através do serviço de comunicação com limite de tempo. Se a mensagem não está pronta (mesmo que seja vazia) uma falha de desempenho ou de parada aconteceu na máquina e ela será posta em um estado seguro (pro exemplo, forçando a falha por parada).

3. Envia uma requisição de iniciação/remoção de serviço ou de liberação de sua fatia de transmissão (se necessário).

4. Programa o escalonamento das funções que implementam os serviços secundários construídos com o suporte do *Wormhole*.

Quando uma interrupção é gerada pelo dispositivo, as mensagens síncronas são recuperadas e os serviços são agendados para o escalonamento. O escalonamento dos serviços para execução é feito através de *tasklets*. *Tasklets* são funções que serão executadas como sendo a parte lenta de uma rotina de interrupção. Elas podem ser atrasadas pelo acontecimento de interrupções mas serão executadas antes de qualquer processo ou *thread*. Estes serviços tem acesso ao *buffer* de mensagens recebidas, e através de uma lista dos serviços que estão carregados, conseguem recuperar o trecho das mensagens endereçado a eles.

A última *tasklet* a ser executada pertence ao próprio *driver*. Ela sinaliza que todos os serviços terminaram de executar e marca a mensagem a ser enviada no próximo período pela rede síncrona como pronta. A não execução desta *tasklet* antes da próxima interrupção do dispositivo (no final do próximo período) sinaliza que a máquina está muito lenta, ou seja, sofreu uma falha de desempenho, ou que a máquina travou. Quando a próxima interrupção acontecer, a rotina de tratamento da interrupção perceberá que a mensagem não foi marcada como pronta e portanto uma falha ocorreu. Esta situação ocorre se o tempo necessário para executar os serviços for maior que a duração do período TDMA.

Execução dos serviços secundários

Um serviço secundário é basicamente uma função carregada em tempo de execução através de um módulo do núcleo do Linux. Esta função pode utilizar os serviços básicos de envio de mensagens com garantia de tempo, o escalonamento programado pelo *driver*, o serviço de detecção de falhas de nós e o relógio global fornecidos pelo *Wormhole*. Quando um serviço

é instalado (o seu módulo é carregado), ele se cadastra no *Wormhole* através do serviço de controle de admissão, que armazena na tabela de serviços disponíveis, informações como a função que faz o processamento necessário, a periodicidade com que ele executa (em número de períodos) e a largura de banda necessária.

Na implementação atual, quando um nó confirma sua participação em um serviço ele inclui, no campo Posição (ver Tabela 4.4, página 55), em que byte da mensagem síncrona começa a mensagem relativa àquele serviço. Todos os nós conhecem *a priori* o tamanho das mensagens de cada serviço. Assim, cada nó mantém uma lista local com os participantes dos serviços que ele também participa e em que posição das mensagens de cada um, mensagens daqueles serviços são transmitidas. Para enviar a mensagem de algum serviço, ele utiliza a mesma tabela para recuperar em que posição de sua mensagem ela deve ser incluída.

A estrutura de dados que armazena a lista de serviços é detalhada na Figura 4.7. A estrutura é composta do código do serviço (único para cada serviço), uma lista encadeada de participantes, um nome para o serviço (apenas para descrição), a periodicidade, um contador que indica quando o serviço deve ser executado, o tamanho da mensagem do serviço e um apontador para a função de processamento do serviço. Em uma lista de participantes, por sua vez, cada participante possui um identificador (o ID do participante na rede síncrona) e a posição de sua mensagem síncrona que contém a mensagem destinada àquele serviço.

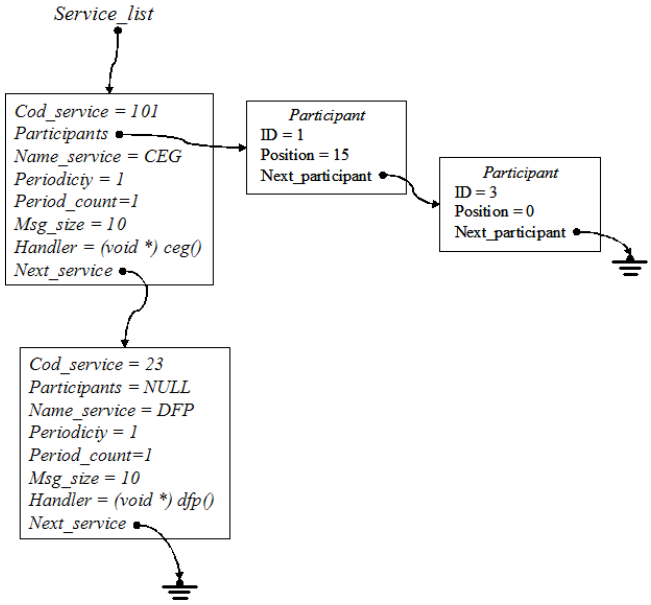

Figura 4.7: Estruturas de dados que armazenam a lista de serviços

Na Figura 4.7 está ilustrada uma lista dos serviços disponíveis em uma máquina. O serviço CEG está carregado, pois possui uma função a ser executada periodicamente e em execução, pois possui dois participantes (de identificadores 1 e 3, um deles sendo o próprio nó). A partir dessa estrutura a função que realiza o processamento do serviço sabe que a partir da posição 15 na mensagem síncrona do nó 3, até a posição 25 (de acordo com o campo Msg_size), está a mensagem que deve ser tratada. O serviço DFP (por exemplo, um detector de falhas de processos) está disponível na máquina (ou seja, seu módulo foi carregado), mas não está em execução .

Interface entre o *Wormhole* e as aplicações

Os serviços básicos do *Wormhole*, bem como os serviços secundários, podem interagir com tarefas que são executadas de forma síncrona (outros serviços) ou com aplicações de usuário. A interface com outros serviços, que são instalados como módulos no próprio núcleo do sistema operacional, acontece através das próprias funções dos serviços (ver Seção 3.5). Entretanto, é preciso alguma forma de interface com o ambiente assíncrono. As formas mais comuns de implementar essa interface são as chamadas de sistema (*system calls*), soquetes locais ou entradas especiais no sistema de arquivos. Chamadas de sistema são menos recomendadas pois exigem a modificação, em tempo de execução, de entradas específicas na tabela de chamadas de sistema do núcleo do sistema operacional. Essa tabela possui uma grande parte de entradas predefinidas, mas as últimas – que seriam modificadas, podem estar associadas a funções diferentes de acordo com a versão do núcleo. A utilização de soquetes consiste em criar conexões locais para comunicação entre processos (tanto orientadas a conexão como a pacote), embora esse mecanismo também seja confiável e eficiente, ele não é o mais comum nem o mais simples. A forma mais utilizada para interface entre *drivers* e aplicações de usuário é o mecanismo de arquivos especiais no sistema de arquivos, geralmente localizados no diretório $/dev$. Cada acesso a estes arquivos ($OPEN$, $CLOSE$, $READ$, $WRITE$, etc.) é então convertido em uma chamada a uma função específica do *driver*.

A ligação entre um arquivo especial e um *driver* é realizada unicamente através do número maior [RC01; BC03], um dos atributos do arquivo. Nem o nome do arquivo em si, nem sua localização, interferem nesta ligação. Cada tipo de dispositivo tem um número maior associado de forma permanente, por exemplo, o *driver* da impressora paralela (*lp*)

está associado ao número maior 6 e o *driver* das unidades de disco flexível (*fd*) está associado ao número maior 2. Números maiores nas faixas de 60 a 63, 120 a 127 e 240 a 254, são reservados para uso local e experimental, desta forma, nenhum dispositivo real utilizará um desses valores. Em nossa implementação o arquivo especial do *Wormhole* tem o número maior igual a 252.

Em adição ao número maior, um arquivo especial tem um outro número associado, chamado de número menor. Esse número menor não é utilizado pelo sistema operacional e serve para que o *driver* possa distinguir entre mais de um dispositivo associado ao mesmo *driver*. Um exemplo disso é o *driver* que trata das unidades de discos flexíveis (identificado pelo número maior 2): o arquivo $/dev/fd0$, associado à primeira unidade de disco flexível, possui numero maior 2 e número menor 0; o arquivo $/dev/fd1$, associado à segunda unidade de disco flexível, possui numero maior 2 e número menor 1.

Um arquivo especial no sistema de arquivos pode ser classificado em dois tipos: orientado a caracter e orientado a bloco. Arquivos especiais orientados a caracter são aqueles que normalmente são acessados como uma seqüência de bytes, como em um arquivo comum. Exemplos de arquivos especiais orientados a caracter são o console (/dev/console) e as portas seriais (/dev/ttyS0 e similares). Arquivos especiais orientados a bloco representam dispositivos de bloco na forma de um disco ou um sistema de arquivos. O dispositivo de bloco é normalmente acessado em blocos de tamanho fixo. Exemplos de dispositivos de blocos são os discos rígidos (/dev/hda, /dev/sda) e discos flexíveis (/dev/fd0). Como o serviços são acessados em forma de blocos pequenos e de tamanho variável, o tipo de arquivo mais adequado é o arquivo a caracter. Um arquivo orientado a caracter pode ser criado com o comando $mknod\ nome\_arquivo\_especial\ c\ 252\ 1$, onde c corresponde ao tipo do arquivo, 252 ao número maior e 1 ao número menor.

O *Wormhole* utiliza números menores para distinguir entre os diversos serviços. Desta forma, cada serviço pode estar associado a um arquivo diferente, mas controlado pelo mesmo *driver*. Isso é possível, pois cada arquivo aberto possui uma estrutura do tipo $file$ associado [RC01; BC03] (essa estrutura é definida no arquivo $include/fs.h$ do código-fonte do Linux) e ela guarda um apontador para uma lista de funções responsáveis pelos diversos tipos de acesso ao arquivo em questão (por exemplo, $OPEN$, $CLOSE$, $READ$, $WRITE$). Quando um arquivo é aberto, o *Wormhole* identifica o número menor do arquivo

aberto e associa a estrutura às respectivas funções. Para isso, um argumento denominado $file\_operations$ contendo um apontador para a estrutura de operações de arquivo foi acrescentado à função $wh\_subscribe\_service$. A atribuição das funcionalidades do *Wormhole* aos números menores é feita da seguinte forma (para maiores detalhes sobre as funções ver Seção 3.5):

- **Relógio global sincronizado**
  - $wh\_get\_global\_time()$: associado ao número menor 11, somente-leitura; um acesso de leitura de 4 bytes retorna o valor do relógio global.
- **O serviço de Detecção de Falhas**
  - $wh\_get\_corrects()$: associado ao número menor 21, somente-leitura; um acesso de n bytes retorna os n primeiros identificadores (um para cada byte) de nós que permanecem corretos; caso o número de identificadores seja menor que n o fim da lista é delimitado com o valor 0.
  - $wh\_is\_correct(ip)$: associado ao número menor 22, leitura e escrita; um acesso de escrita de 4 bytes informa o endereço IP e uma acesso de leitura de 5 bytes, retorna o endereço IP mais um byte que possui o valor 1 se o nó com o IP fornecido permanece correto e o valor 0 caso contrário. Para evitar que dois processos acessando de forma concorrente interfiram um no outro, quando um acesso de escrita de um IP é feito, esse IP é armazenado no campo $private\_data$ da estrutura $file$ associada ao arquivo aberto por aquele processo [RC01]. Desta forma, quando o arquivo é lido, o resultado da leitura é calculado de acordo com as informações fornecidas pelo mesmo processo.
- **O controle de admissão.**
  - $wh\_subscribe\_service(name, periodicity, bandwidth, handler, file\_ops)$: não é acessível de fora do núcleo do sistema operacional.
  - $wh\_unsubscribe\_service(name)$: não é acessível de fora do núcleo do sistema operacional.

  - $wh\_request\_service(name, execution\_period, service\_parameters)$: associado ao número menor 31, leitura e escrita; um acesso de escrita de 21 bytes inicia um serviço (10 bytes para o nome, 1 byte para a periodicidade e 10 para os parâmetros, bytes não utilizados devem ser preenchidos com o caracter $'\backslash 0'$); o acesso bloqueia até que o serviço seja carregado ou um erro de escrita seja sinalizado (em caso de falha na iniciação); depois de iniciado o serviço o arquivo já aberto é associado ao serviço solicitado (*i.e.* as funções que manipulam as operações no arquivo são substituídas pelas funções do serviço específico).
  - $wh\_remove\_service(name)$: executada no fechamento do descritor de arquivo aberto para a iniciação do serviço.

- **Outras funções**

  - $id\_list \Leftarrow get\_ids()$: associado ao número menor 21 (da mesma forma que a função $wh\_get\_corrects()$), somente-leitura; um acesso de n bytes retorna os n primeiros identificadores (um para cada byte) de nós que permanecem corretos; caso o número de identificadores seja menor que n o fim da lista é delimitado com o identificador 0.
  - $get\_max\_nodes()$: associada ao número menor 101, somente-leitura; um acesso de 1 byte retorna o número máximo de nós que podem fazer parte da rede síncrona.
  - $get\_max\_delay()$: associada ao número menor 102, somente-leitura; um acesso de 2 bytes retorna a duração máxima de período TDMA, em dezenas de microssegundos.

Uma tentativa de leitura ou escrita com um tamanho diferente dos especificados acima resulta em um erro de escrita do tipo -1 (*Operation not permitted*, a variável $errno$ é ajustada para -1). Os demais erros são sinalizados com o código -5 (*I/O error*, a variável $errno$ é ajustada para o valor -5)

4.5 Desempenho do *Wormhole*

O desempenho do *Wormhole* é calculado a partir da duração do período TDMA. Este, por sua vez, é calculado a partir da duração das fatias de tempo. Uma fatia de tempo é composta, principalmente, do tempo necessário para que uma mensagem seja transferida do microcontrolador para o controlador Ethernet no nó que envia a mensagem, mais o tempo da transferência do controlador para o microcontrolador no nó que recebe a mensagem. Além destes dois componentes principais, a fatia inclui o tempo das tarefas de processamento dentro do microcontrolador e o tempo de transmissão na rede. Como o tempo de transferência entre o microcontrolador e o controlador Ethernet depende apenas do tamanho da mensagem, o tempo gasto no envio é igual ao tempo gasto no recebimento. O tempo na fatia de comunicação corresponde ao tempo gasto em toda a transferência das mensagens do dispositivo para o PC e vice-versa. A Tabela 4.7 detalha esses tempos.

Tipo de fatia	T_{envio}	$T_{recebimento}$	Outras operações	Duração total da fatia
Sincronização	2,93	2,93	2	7,86
Inscrição	1,65	1,65	1,4	4,7
Transmissão	2,05	2,05	0,30	4,4
Comunicação	0	0	71	71

Tabela 4.7: Duração das fatias do TDMA (em milisegundos).

A duração das fatias apresentadas na Tabela 4.7 resulta em um período de $94,79ms$ para uma rede com cinco nós (1 líder e 4 escravos). Entretanto, mais de 60% do período é gasto na transferência das mensagens entre o dispositivo e o PC pela porta serial.

Além da duração de um período, outro parâmetro que precisa ser considerado é o tempo decorrido até o escalonamento dos serviços instalados no núcleo do sistema operacional. Um experimento foi realizado para medir esse tempo: foi inserida uma linha de código que enviava um sinal pela porta paralela no fim da rotina de tratamento de interrupção do *driver*; e um sinal no fim da execução de uma *tasklet* que realizava algumas atividades simples (com a complexidade esperada dos serviços que serão implementados - atualização de tabelas na memória e cálculos simples). Um osciloscópio foi então ligado ao dispositivo (conectado à porta serial do PC) e à porta paralela do PC. O tempo entre o sinal gerado pelo dispositivo

e a resposta na rotina de interrupção, ficou entre 80 e $100\mu s$, estando a grande maioria localizada entre 80 e $90\mu s$. O tempo entre o sinal do dispositivo e o correspondente no fim da *tasklet* foi de 80 a $450\mu s$ (lembrando que uma tasklet pode ser interrompida por interrupções). Neste segundo caso a maioria dos tempos ficou entre 100 e $200\mu s$. Os experimentos foram realizados em uma máquina lenta (Pentium III 866 MHz) e extremamente carregada[5].

Os serviços idealizados para o *Wormhole* não realizarão operações de Entrada/Saída (disco, rede assíncrona, etc.), apenas irão realizar um processamento interno e disponibilizar, através de um buffer intermediário, algum conjunto de informações para as aplicações assíncronas. Desta forma, na medida que as aplicações assíncronas recuperam essa informação fora do tempo de execução da *tasklet*, para períodos TDMA na ordem de milisegundos, o desempenho do *Wormhole* está diretamente e unicamente dependente da duração do período TDMA.

[5] A carga consistia em 2 processos criando e apagando arquivos de 10 Kbytes continuamente, 2 comandos de “ping” na máxima taxa (*flood-ping*), 20 conexões SSH em atividade, entre outros processos.

Capítulo 5

Usando o sistema implementado

5.1 Definindo um serviço

Um serviço é um módulo para o núcleo do Linux e possui no mínimo três funções:

- $int\ init\_module(void)$: executada pelo Linux quando o módulo é carregado. Esta função será responsável pelo cadastramento do serviço junto ao *Wormhole* e pela iniciação do próprio serviço. O valor de retorno deve ser positivo ou o módulo não será carregado.

- $void\ service\_handler(unsigned\ long\ unused)$: executada pelo *driver* do *Wormhole* a cada período, recebendo a mensagem destinada a esse serviço. Esta função é executada na forma de *tasklet* podendo realizar operações de entrada e saída. Ela não é executada de forma concorrente (duas instâncias não executam em paralelo). O parâmetro $unused$ não é utilizado.

- $int\ clean\_module(void)$: executada pelo Linux quando remove um módulo da memória. Ela deve também remover o cadastro do módulo junto ao *Wormhole*.

Além dessas funções o serviço pode possuir funções de interface que disponibilizem funcionalidade do serviço para as aplicações de usuário. Essas outras funções podem implementar chamadas de sistema, conexões de rede ou arquivos especiais para comunicar-se com as aplicações do usuário. A maneira mais comum é a utilização de arquivos especiais [RC01], e um exemplo deste tipo de interface é dado nas seções seguintes.

5.2 O Compilador de Estados Globais

O Compilador de Estados Globais (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. O trabalho de Brasileiro *et al.* [BBCS04] compara o desempenho deste protocolo com outros protocolos conhecidos.

Para resolver o problema do consenso entre um conjunto de n processos $\Pi = \{p_1, ..., p_n\}$, 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 que contém a identidade do processo que propôs a mensagem consensual do protocolo (inicialmente $\perp$). O tamanho deste campo é $\lceil log_2 n \rceil$ bits.

5.2.1 Implementando o CEG

As aplicações podem acessar os REGs a partir de um arquivo especial, uma entrada no sistema de arquivos diretamente conectada ao serviço CEG. Para acessar o CEG, uma aplicação acessa um arquivo especial de número maior 252 e número menor 31 (criado com $mknod\ wh\_adm\_control\ c\ 252\ 31$). Esse arquivo está diretamente conectado ao serviço de controle de admissão. Uma vez aberto o arquivo, a aplicação escreve um pacote de 21 bytes com o nome do serviço (10 bytes), a periodicidade (1 byte) e os parâmetros do serviço (10

bytes). 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 (conforme explicado na Seção 4.4).

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. Na implementação atual, apenas um processo em cada nó pode utilizar o serviço por vez. Esse problema pode ser contornado criando-se vários módulos, cada um com seu arquivo especial associado.

Por trás do arquivo especial, existe um módulo que implementa o serviço em si. Este módulo é composto de seis funções básicas: $init\_module()$, $cleanup\_module()$, 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 a ser carregado pelo núcleo. 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, sua periodicidade de execução, a largura de banda necessária (em bytes por período) e o apontador para a função que realiza o processamento de suas mensagens (veja Figura 4.7, página 62); (2) realiza os procedimentos necessários para interagir com o ambiente assíncrono (por exemplo, alocando números maiores) e quaisquer recursos que o módulo utilize durante a realização de seus serviços.

A função $cleanup\_module()$ é responsável pelo 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 libera quaisquer recursos utilizados durante sua execução (por exemplo, números maiores para utilização de arquivos especiais como interface com o ambiente assíncrono).

A função de processamento do serviço é uma função que será executada como um *tasklet* pelo *driver* do *Wormhole* sempre que um período acaba[1]. 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 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, cuja estrutura é ilustrada na Figura 5.1, 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 $n + n^2 + \lceil log_2 n \rceil$ 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.

5.2.2 Propriedades dos REGs

O CEG pode ser formalmente definido por um conjunto de propriedades que determinam a construção dos REGs:

- **detecção com abrangência forte**: se algum processo p_j falha, então após um tempo finito os REGs fornecidos pelo CEG terão $detection\_vector[j] = 1$;

[1]Nesta implementação todos os serviços executam com periodicidade 1.

```
struct file_operations ceg_fops = {
    read: read_mod,
    write: write_mod,
    release: release_mod
};

ssize_t read_mod(struct file* file, char * buf, size_t count, loff_t *l) {
...
}

ssize_t write_mod(struct file* file, const char * buf, size_t count, loff_t *l) {
...
}

int release_mod(struct inode *inode, struct file * file) {
...
}
```

Figura 5.1: Formato das funções de manipulação dos acesos a um arquivo especial

- **detecção com exatidão forte**: p_j realmente falhou se algum REG fornecido pelo CEG contém $detection\_vector[j] = 1$;
- **recepção com abrangência forte**: se algum processo correto p_i recebeu uma mensagem de um processo p_j, então após um tempo finito, todos os CEGs entregarão um REG que contém $reception\_matrix[i, j] = 1$;
- **recepção com exatidão forte**: p_i realmente recebeu uma mensagem de p_j se algum CEG entrega um REG que contém $reception\_matrix[i, j] = 1$;
- **validade**: se algum REG reg possui $reg.consensual\_identity = x$ e f_{actual} é o número de posições em $reg.detection\_vector$ com valor 1 (*i.e.* o número de processos cujas falhas foram detectadas), então existem ao menos $\zeta - f_{actual}$ ocorrências de i ($1 \leq i \leq n$), onde ζ é um parâmetro de execução do CEG, tal que: $reg.detection\_vector[i] = 0 \wedge reg.reception\_matrix[i, x] = 1$; além disso, se reg tem $reg.consensual\_identity = \perp$, então não existe um processo p_x tal que, existem pelo menos $\zeta - f_{actual}$ ocorrências de i ($1 \leq i \leq n$) que satisfazem: $reg.detection\_vector[i] = 0 \wedge reg.reception\_matrix[i, x] = 1$; e,
- **escrita-única**: se um REG é entregue com $consensual\_identity = x \neq \perp$, então todo REG entregue por qualquer instância do serviço CEG onde $consensual\_identity \neq \perp$

também possui $consensual\_identity = x$.

Cada instância do serviço CEG, ceg_i, mantém duas variáveis de resumo de estado global, denominadas $ready\_for\_delivery_i$ e $local_i$. Inicialmente, todos os bits do $detection\_vector$ e da $reception\_matrix$ são iguais a 0, e seus campos $consensual\_identity$ são iguais a $\perp$. Para acessar o serviço CEG local, e recuperar um resumo do estado global um processo p_i lê do arquivo especial conectado ao serviço. Quando p_i lê deste arquivo, ele recebe como retorno o valor da variável $ready\_for\_delivery_i$. A variável $local_i$ é utilizada para calcular localmente resumos de estados válidos. Sempre que um resumo de estado global válido é formado pela instância ceg_i do serviço CEG, ela copia $local_i$ para $ready\_for\_delivery_i$, disponibilizando esse novo resumo de estado global para entrega.

As propriedades de detecção de abrangência forte e detecção de exatidão forte são garantidas pelo *Wormhole*. O *Wormhole* difunde periodicamente os estados locais de cada máquina, como essa difusão é atômica e o canal é síncrono: após um tempo, todo processo correto receberá ou perceberá a falta de um estado local, o que pode ser interpretado como sinal de que o CEG na outra extremidade está correto ou falho, respectivamente (detecção de abrangência forte); se um estado local de uma máquina p_i não é recebido pela máquina p_j após a duração de um período TDMA (que considera o atraso máximo que uma mensagem síncrona pode sofrer), p_i realmente não enviou seu estado e sua instância do CEG falhou[2].

As propriedades de recepção de abrangência forte e de recepção de exatidão forte são trivialmente satisfeitas. Sempre que um processo p_i recebe uma mensagem do protocolo de p_j, a sua instância do CEG ceg_i é notificada a partir de um acesso de escrita no arquivo especial aberto pelo processo. Então, ceg_i ajusta sua matriz $local_i.reception\_matrix[i, j] = 1$ e a difunde para todas as outras instâncias do CEG. Quando a instância ceg_k recebe tal mensagem, ela ajusta o elemento correspondente em sua matriz $local_k.reception\_matrix$. Isto garante a recepção de exatidão forte, na medida que todos os resumos de estado globais serão copiados para os respectivos $ready\_for\_delivery$. Além disso, como todas as mensagens difundidas por uma instância correta do CEG são sempre recebidas por todas as outras instâncias, a recepção de abrangência forte é também garantida.

[2]O *Wormhole* garante, através de seu cão-de-guarda, que falhas de desempenho serão transformadas falhas por parada.

Como as difusões dos estados locais de cada instância do CEG são atômicas, todos as instâncias receberão os estado na mesma ordem. Desta forma, todas as instâncias têm a mesma visão dos estados locais das outras instâncias, e portanto constroem a mesma matriz $reception\_matrix$, e a mesma visão das falhas, e portanto constroem também o mesmo vetor $detection\_vector$. Logo, na primeira vez que existirem $\zeta - f_{actual}$ (onde ζ é um parâmetro de execução do CEG) ocorrências de i, $1 \leq i \leq n$, que satisfaçam $reg.detection\_vector[i] = 0 \wedge reg.reception\_matrix[i, x] = 1$, o identificador $consensual\_identity$ será ajustado por todas as instâncias, garantindo a propriedade de validade. Uma vez ajustado, o identificador $consensual\_identity$ nunca será modificado, o que satisfaz a propriedade de escrita-única.

5.2.3 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 [CBGS00]:

- **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 $f+1 \leq quorum \leq n$, onde f é o número máximo de falhas que podem ser toleradas. 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(ζ,ψ)* onde ζ e ψ 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*, ψ 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 verificação que o Algoritmo 1 resolve o consenso é detalhada a seguir.

Lema 1 Todo processo correto que executa o protocolo apresentado no Algoritmo 1, decide um valor em um tempo finito (propriedade de *terminação*)

Prova Se um processo correto decide em algum momento ele o faz executando a tarefa de decisão. Nesta tarefa a decisão é realizada apenas se o processo obtém um resumo de estado global reg, tal que $reg.consensual\_identity \neq \perp$. Conseqüentemente, para provar o lema é necessário primeiro mostrar que, após um tempo finito, todo processo correto obtém tal resumo. A partir da propriedade de validade, um resumo de estado global reg com $reg.consensual\_identity \neq \perp$ pode ser entregue apenas se existe um p_x para o qual existem ao menos $\zeta - f_{actual}$ ocorrências de i, $1 \leq i \leq n$, tal que $reg.detection\_vector[i] = 0$ e $reg.reception\_matrix[i, x] = 1$. Como os canais são confiáveis e ao menos ψ ($\psi \geq f + 1$) processos fazem difusão de seus valores para todos os processos, a mensagem de proposição de ao menos 1 processo irá ser recebida por todos os processos que não falharam, *i.e.* $n - f_{actual}$ processos. Sem perda de generalidade, se p_x for tal processo, a propriedade de recepção de abrangência forte garante que, após um tempo finito, todos os re-

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

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

% Tarefa de proposição
quando execute propose(v_i)
  se i ≤ ψ 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
```

sumos de estados globais entregues pelo CEG, terão $reception\_matrix[i, x] = 1$, para cada p_i correto. Por outro lado, a propriedade de detecção com abrangência forte garante que, após um tempo finito, todos os resumos do estado global entregues pelo CEG terão $detection\_vector[j] = 1$ para todo p_j que falhou. Portanto, após um tempo finito, todos os resumos do estado global entregues terão $n - f_{actual}$ ocorrências de i ($1 \leq i \leq n$), tal que, $detection\_vector[i] = 0$ e $reception\_matrix[i, x] = 1$. Como $n - f_{actual} \geq \zeta - f_{actual}$, então algum deve ter $decision\_identity \neq \perp$. Como a tarefa de decisão permanece constantemente consultando o CEG, após um tempo finito, um resumo de estado global ceg com $reg.consensual\_identity \neq \perp$ é entregue a todos os processos corretos.

Para completar a prova deve ser mostrado que a mensagem de proposição enviada por um processo cujo identifcador é $reg.consensual\_identity$ realmente foi recebida por todos os processos corretos p_i, e portanto, pode ser recuperada de $bagOfMessages_i$. A propriedade de detecção com exatidão forte garante que para todo processo correto p_i, o CEG não en-

tregará um resumo de estado global com $reg.detection\_vector[i] = 1$, assim, os processos corretos são um subconjunto dos processos que o resumo não indica como falhos. A propriedade de recepção com exatidão forte garante que se $reg.reception\_matrix[i, x] = 1$ então p_i realmente recebeu a mensagem enviada por p_x. Como $\zeta \geq f + 1$ e $f \geq f_{actual}$, então, existe pelo menos 1 processo correto entre os $\zeta - f_{actual}$ processos que recebeu a mensagem de p_x. Esse processo irá completar a difusão de p_x, então mesmo que p_x falhe durante a difusão de sua mensagem, é garantido que todo processo correto irá recebê-la. □

Lema 2 Todo processo que executa o protocolo apresentado no Algoritmo 1 decide no máximo uma vez (propriedade de *integridade uniforme*).

Prova Este é trivialmente satisfeito pelo protocolo apresentado no Algoritmo 1. Como pode ser visto no pseudo-código do protocolo, existe apenas um ponto de decisão para o processo que executa o protocolo e logo após a decisão o protocolo é encerrado. □

Lema 3 Se um processo que executa o protocolo apresentado no Algoritmo 1 decide por um valor v, então v foi proposta por algum processo (propriedade de *validade uniforme*).

Prova No Algoritmo 1, o valor de decisão de um processo p_i é um dos que foi encapsulado em uma mensagem contida em $bagOfMessages_i$, e as únicas mensagens que entram no repositório $bagOfMessages_i$ de um processo p_i são as mensagens de proposição enviadas por processos executando o protocolo. □

Lema 4 Não existem dois processos que executam o protocolo apresentado em Algoritmo 1 que decidem diferente (propriedade de *acordo uniforme*).

Prova O Lema 1 mostra que todo processo correto decide. Seja p_x o processo cuja proposição foi o valor de decisão de um processo p_i. Então, p_i deve ter recebido um resumo de estado global reg tal que $reg.consensual\_identity = x$. Pela propriedade de escrita-única, qualquer processo que entregou um resumo de estado global r' com $r'.consensual\_identity \neq \perp$, tem $r'.consensual\_identity = x$. Desta forma, todo processo que decide deve decidir pelo valor proposto por p_x. □

Teorema 1 O protocolo apresentado no Algoritmo 1 resolve o consenso.

Prova A prova é conseqüência direta dos lemas 1, 2, 3 e 4. □

Capítulo 6

Conclusão

6.1 Considerações finais

Neste trabalho, foi apresentada uma infraestrutura para a implementação de sistemas distribuídos híbridos a partir de novas abstrações. O ambiente desenvolvido permite a comunicação com garantias de entrega e também força o escalonamento previsível de funções instaladas no núcleo do sistema operacional. Este suporte permite implementações simples para os detectores de falhas perfeitos, com limites no atraso de detecção de uma falha, que foi a motivação inicial deste trabalho. Os detectores de falhas teriam um tempo de detecção da mesma ordem do período de comunicação e de escalonamento das tarefas do *Wormhole*. Estes tempos são da ordem de $100ms$ para uma rede de 5 máquinas, podendo ser alcançados tempos de até $10ms$ para uma rede de 10 máquinas (desde que aplicadas certas otimizações detalhadas na próxima seção). Esses tempos são muito bons comparados com os tempos de 700 a $1500ms$ obtidos por Oliveira [Oli03] e de 3 a $6s$ obtidos por Fetzer [Fet03] e compensam a desvantagem de utilizar um dispositivo de *hardware* dedicado.

Entretanto, durante o desenvolvimento deste trabalho, foi percebido que abstrações mais fortes que um detector de falhas poderiam ser criadas. Foi apresentado o Compilador de Estados Globais (CEG), um primeiro exemplo dessas novas abstrações. O CEG fornece informações que possibilitam que o protocolo de consenso 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. No protocolo de consenso apresentado, por exemplo, a adaptabilidade pode ser ob-

tida através de um resumo das mensagens que cada processo recebeu. A implementação do *Wormhole* e o CEG tornam-se então um grande contribuição para a área de sistemas distribuídos. Mais que isso, essa implementação abre espaço para que novos trabalhos identifiquem novas abstrações ainda mais poderosas, confiantes na existência de uma implementação de um *Wormhole*.

Como subproduto do *Wormhole*, embora não menos importante, os dispositivos desenvolvidos são plenamente capazes de formar uma rede síncrona e trocar mensagens, mesmo na ausência de um PC. A utilização de um microcontrolador, equipado com diversas interfaces para o mundo exterior (como barramentos I2C e serial, entradas e saídas digitais, conversores Analógico-Digital), permite a criação de uma rede síncrona de sensores, atuadores e até PCs. Esta rede de monitoramento permitiria a observação, em tempo-real, de estados de um equipamento (temperatura, tensão, corrente, etc.), de estados de uma máquina (utilização do processador, da memória, etc.) e ainda, de estados de processos em execução em outras máquinas (correto ou falho, por exemplo).

Finalmente, pode-se concluir que os objetivos deste trabalho foram atendidos (ver Seção 1.2, página 4):

- Desenvolver um canal de comunicação síncrono para ser utilizado pelos nós de uma rede local de prateleira: foi construído um dispositivo síncrono que utiliza uma rede Ethernet com controle de acesso e que oferece aos nós de uma rede local um serviço de envio de mensagens com um limite máximo de tempo de entrega garantido.

- Desenvolver um ambiente de execução síncrono, de capacidade limitada: o dispositivo desenvolvido interrompe a máquina associada em intervalos períodicos e, no tratamento destas interrupções, força o escalonamento de determinadas tarefas; desta forma, além do mecanismo de comunicação síncrona detalhado acima, obtém-se também um limite de tempo superior para o atraso no escalonamento das tarefas síncronas (os serviços).

- Desenvolver um mecanismo de interface entre a parte síncrona e a assíncrona que permita que a parte síncrona possa atender as requisições da parte assíncrona, que chegam de forma arbitrária, sem comprometer seus prazos ou interferir em outros nós: foi desenvolvido um *driver* de dispositivo que executa na máquina e que agrupa

as requisições assíncronas, repassando-as para o ambiente síncrono em momentos e tamanhos pré-determinados.

- Desenvolver exemplos que utilizem o suporte desenvolvido e mostrem como um sistema assíncrono pode tirar proveito do fato de que uma pequena parte do sistema tem um comportamento síncrono: foi desenvolvido o Compilador de Estados Globais.

6.2 Trabalhos futuros

Este trabalho faz parte de um trabalho maior que ainda está em desenvolvimento. De forma a melhorar a infra-estrutura disponível e estimular o desenvolvimento de protocolos que utilizem o *Wormhole*, algumas expansões e melhoramentos já estão previstos.

Em um primeiro momento, algumas questões de desempenho serão abordadas. A velocidade de comunicação utilizada na porta serial é de 115 Kbps, cerca de 11,5 Kbytes/segundo. Novos padrões para a porta paralela (ECP/EPP) permitem uma velocidade de 500 a 2000 Kbytes/segundo e a porta USB, mesmo nas primeiras versões, possui uma velocidade de 1000 Kbytes/segundo. O microcontrolador utilizado atualmente pode lidar com uma velocidade de comunicação de até 200 Kbytes/segundo, permitindo uma diminuição de quase 20 vezes no tempo de comunicação com o PC. Este fator diminuiria a duração de um período para cerca de $27ms$ para uma rede de 5 máquinas. Esta é uma modificação que já está em andamento.

Outra otimização é habilitar a comunicação com o PC para que esta possa ocorrer em paralelo com o período TDMA, durante os momentos que o microcontrolador está ocioso. Como o tempo de duração de uma fatia TDMA considera o tempo de envio de uma mensagem e o tempo de recepção da mesma mensagem e um nó ou envia ou recebe uma mensagem durante uma fatia, cada nó passa metade de todas as fatias ocioso. Além disso, a utilização de um microcontrolador mais rápido (o atual usa um relógio de 8 MHz), o tempo de comunicação com o controlador Ethernet e o tempo de realização das tarefas internas de processamento cairia proporcionalmente. Como microcontroladores de até 60 MHz podem ser encontrados (além de FPGAs nesta mesma faixa de velocidade), períodos de $10ms$ para redes de 10 máquinas são concebíveis.

No lado do *driver* em execução no PC, algumas otimizações também estão sendo estu-

dadas. Por exemplo, uma forma mais sofisticada para lidar com as falhas de desempenho de um serviço, criando mecanismos de segurança que atuem apenas no serviço falho (e não em todo o nó); um escalonador local que possibilite o compartilhamento de uma parte da banda do *Wormhole* por vários serviços que não necessitam executar em todos os períodos; um mecanismo de segurança que permita que apenas serviços confiáveis tenham acesso ao *Wormhole*; um mecanismo que permita várias instâncias de um serviço executarem paralelamente; e finalmente, desenvolver novas abstrações que utilizem o *Wormhole* e aplicar o CEG a outros problemas fundamentais dos sistemas distribuídos.

Bibliografia

[BBCS04] F. Brasileiro, A. Brito, W. Cirne, and L. Sampaio. Fast adaptable uniform consensus using global state digests. Technical report, Universidade Federal de Campina Grande / HP Brasil, 2004.

[BC03] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 3 edition, 2003.

[Bha01] P. Bhagwat. Bluetooth: Technology for short-range wireless apps. *IEEE Internet Computing*, pages 96–103, May-Jun 2001.

[Blu04a] The official bluetooth membership site, 2004. http://www.bluetooth.org.

[Blu04b] The official bluetooth website, 2004. http://www.bluetooth.com.

[BOB03] A. Brito, E. Oliveira, and F. Brasileiro. Delphus: Uma ferramenta para detecção de falhas em redes locais. In *Salão de Ferramentas do Simpósio Brasileiro de Redes de Computadores*, pages 881–888, Natal/RN, Brasil, May 2003.

[Bri03] A. Brito. Projeto e implementação de um módulo síncrono para a implementação de um serviço de detecção de falhas com semântica perfeita para redes locais. M. Sc. Proposal, COPIN/DSC/UFCG, Março 2003.

[BY97] M. Barabanov and V. Yodaiken. Real-time linux. *Linux Journal*, Feb 1997.

[CBGS00] B. Charron-Bost, R. Guerraoui, and A. Schiper. 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, 2000. IEEE Computer Society.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, Jun 1999.

[CMV00] A. Casimiro, P. Martins, and P. Veríssimo. 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, September 2000. IEEE Industrial Electronics Society.

[Con04] ControlNet International. ControlNet, 2004. http://www.controlnet.org.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of sevice of failure detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)*, pages 191–200, New York, USA, Jun 2000.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan 1987.

[DIA04] DIAPM. RTAI - realtime application interface, 2004. http://www.aero.polimi.it/~rtai/.

[DLS88] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr 1988.

[FC96] C. Fetzer and F. Cristian. Fail-aware failure detectors. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS 1996)*, pages 200–209, Los Alamitos, Ca., USA, 1996. IEEE Computer Society Press.

[Fet03] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb 2003.

[FLP85] M. J. Fischer, N. A. Lynch, and M. D. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, Apr 1985.

[FSM04] FSMLabs. RTLinux, 2004. http://www.fsmlabs.com.

[GW02] J. Gross and A. Willig. Measurements of a wireless link in different rf-isolated environments. In *In Proc. European Wireless 2002 (EW2002)*, Florence, Italy, February 2002.

[IEE] IEEE. ANSI/IEEE standard 802.3p and standard 802.3q. http://www.ieee.org.

[Int04] Phoenix Contact International. Phoenix contact, 2004. http://www.phoenixcontact.com.

[Le 87] G. Le Lann. A deterministic 802.3 protocol for industrial local area networks. Technical report, INRIA, INRIA-BP 105, F-75153 Le Chesnay Cedex, France, 1987.

[LFA01] M. Larrea, A. Fernández, and S. Arévalo. On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Brief Announcements 15th Int'l Symp. Distributed Computing (DISC 2001)*, 2001.

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *ACM Communications*, 7(19):395–404, July 1976.

[OBB03] E. Oliveira, A. Brito, and F. Brasileiro. Projeto e implementação de um serviço de detecção de falhas perfeito. In *Simpósio Brasileiro de Redes de Computadores*, pages 697–712, Natal/RN, Brasil, May 2003.

[Oli03] E. W. Oliveira. Projeto e implementação de um serviço de detecção de falhas com semântica perfeita para redes locais. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, Agosto 2003.

[POS96] 1996 revision of POSIX.1. ISO/IEC 9945-1:1996, 1996.

[POS03] POSIX.1b standard. IEEE Standard 1003.1b-1993, 2003.

[Pro04] Profibus International. Profibus, 2004. http://www.profibus.org.

[QA04] Quadravox Inc. and Archelon Inc. AQ430 C compiler, 2004. http://www.quadravox.com/AQ430.htm.

[Rad03] Radiometrix. 433 MHz high speed FM radio transceiver module, 2003. http://www.radiometrix.co.uk/products/bim2.htm.

[RC01] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2001.

[Rea03] Realtek. RTL8019AS full-duplex ethernet controller, 2003. http://www.realtek.com.tw.

[Rob04] Robert Bosch GmbH. CAN Homepage, 2004. http://www.can.bosch.com.

[Row04] Rowley Associates. Crossworks C compiler, 2004. http://www.rowley.co.uk.

[SM95] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, 17, 1995.

[Tex04a] Texas Instruments. MSP430 microcontrollers, 2004. http://www.msp430.com.

[Tex04b] Texas Instruments. TRF6901 - single chip RF transceiver, 2004. http://www.ti.com.

[VA95] P. Verissimo and C. Almeida. 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, Dec 1995.

[van04] D. van Heesch. Doxygen documentation system, 2004. http://www.doxygen.org.

[VC02] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *Transactions on Computers*, 51(8):916–930, August 2002.

[Ver03] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? *Future Directions in Distributed Computing, Springer Verlag LNCS 2584*, pages 108–113, May 2003.

[vN02] R. van Ness (Hewlett-Packard). The importance of IEEE 1451t. *The Industrial Ethernet Book*, Issue 1, 2002.

[Wil03] A. Willig. Polling-based MAC protocols for improving real-time performance in a wireless profibus. *IEEE Transactions on Industrial Electronics*, 50(4), August 2003.

Apêndice A

Implementando o CEG e o Consensus-CEG(q, p)

O exemplo desenvolvido a seguir detalha o código-fonte para a implementação de um Compilador de Estados Globais simplificado, neste exemplo, o número máximo de processos distribuídos executando o Consenso-CEG é 7, cada processo tem um identificador de 0 a 6 e executa em uma máquina com um dispositivo do *Wormhole* instalado. Por simplicidade, existem 7 nós na rede síncrona e todos executam o protocolo. O protocolo implementado é o Consenso-CEG(n, n), onde todos os processos propõem valores e todos os corretos precisam receber uma dada mensagem antes que a decisão seja tomada.

Uma implementação do CEG no *Wormhole* pode ser dividida em duas partes. Na primeira parte, a função de processamento do serviço, cadastrada no *Wormhole*, enviaria e receberia mensagens na rede síncrona. Essas mensagens têm o comprimento de 1 byte. Os 7 bits menos significativos de uma mensagem correspondem a uma linha da matriz $detection\_matrix$, cada bit corresponde a uma coluna. Desta forma, o bit j, contando a partir do menos significativo, sinaliza se o processo corrente recebeu uma mensagem de proposta do processo p_j. O bit mais significativo da mensagem é o *heartbeat*, onde o valor 1 sinaliza que o processo que utiliza o CEG permanece correto. Um processo pode ser considerado correto enquanto o arquivo especial que funciona como interface entre o CEG e a aplicação permanece aberto[1] ou enquanto o mesmo mantém o respectivo bit atualizado.

[1] Isso decorre do fato de que quando um processo é encerrado os seus arquivos abertos são automaticamente fechados

Quando o serviço é iniciado, o CEG envia a cada período 1 byte com o estado do nó local (que pode ter sido atualizado ou não pela parte assíncrona) e recebe 6 bytes, correspondendo as outras mensagens recebidas na rede síncrona pelo *Wormhole*.

O código na Figura A.1 mostra a primeira parte do serviço. O arquivo-cabeçalho $wh.h$ inclui as definições dos *buffers* de mensagens recebidas (wh_rcvmsg_buffer) e da mensagem a ser enviada ($wh\_sendmsg\_buffer$) na rede síncrona. Neste exemplo simplificado, a mensagem do serviço CEG corresponde sempre ao primeiro byte de uma mensagem, não sendo necessário consultar a lista de serviços presente no driver (ver Figura 4.7, página 62).

Na Figura A.1, as linhas 14 a 18 definem a estrutura ceg_fops com as operações válidas no arquivo especial (fechar, ler e escrever). A função $init\_module()$ cadastra o serviço junto ao *Wormhole* (linha 23) no momento de seu carregamento. O cadastro registra a função $funcao\_servico1()$ como sendo a função de processamento do serviço. Ela é cadastrada de forma a executar com periodicidade 1 (em todos os períodos) e com um tamanho de mensagens 1 (um byte reservado em cada mensagem síncrona). Também é cadastrada a estrutura ceg_fops como a estrutura que encapsula as funções que tratam os acessos ao serviço CEG.

A função $funcao\_servico1()$ é escalonada logo depois do fim de um período na rede síncrona e tem acesso aos *buffers* de mensagens recebidas e da mensagem a ser enviada no próximo período. Quando chamada, a função recupera o primeiro byte de cada uma das sete mensagens síncronas e armazena-os no vetor $estado\_global$. Para cada estado local recuperado de uma mensagem síncrona, ela verifica se o bit de *heartbeat* está ativo (linha 34) e se estiver, realiza uma operação lógica que avalia se algum bit está ativo em todas as mensagens (ou seja, se todos os processos receberam alguma mensagem). Ao fim da função, o estado local é copiado para o *buffer* de transmissão para ser enviado no próximo período. A atualização do estado local é realizada pela aplicação e será detalhada mais adiante.

A função $cleanup\_module()$, chamada quando o módulo é removido da memória e remove o cadastro da função do serviço no *Wormhole*.

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <asm/bitops.h>
#include "wh.h"   // define *wh_rcvmsg_buffer, *wh_sendmsg_buffer
MODULE_LICENSE("GPL");

char * funcao_servico(unsigned long unused);
char estado_local, estado_global[7];
int terminado=0;

struct file_operations ceg_fops = {
    read: read_mod,
    write: write_mod,
    release: release_mod,
};

int init_module(void) {
    int cod;
    printk("Carregando serviço.\n");
    cod = wh_subscribe_service("CEG", 1, 1, &funcao_servico1, &ceg_fops);
    if (cod < 0) // Caso a função não possa ser inscrita encerre.
        return -1;

    return 0;
}

void funcao_servico1(unsigned long unused) {
    int c;

    if (terminado == 0) {
        for (c=0; c<7; c++) {
            // O estado global neste exemplo é o primeiro byte de cada mensagem:
            // wh_rcvmsg_buffer[0], wh_rcvmsg_buffer[WH_MSG_SIZE+0], ...
            estado_global[c] = wh_rcvmsg_buffer[c*WH_MSG_SIZE];
            if (estado_global[c] & 0x80)
                terminado &=  estado_global[c] & 0x7F;
        }
    }
    wh_sendmsg_buffer[0] = estado_local;
}

void cleanup_module(void) {
    printk("Removendo serviço.\n");
    wh_unsubscribe_service("CEG");
}
```

Figura A.1: Código-fonte - Compilador de Estados Globais (CEG)(parte 1/3)

A segunda parte do código-fonte do módulo pode ser vista na Figura A.2. Essa segunda parte do serviço implementa a interface com o usuário, ou seja, as funções que manipulam

os acessos ao arquivo especial criado, consultam o estado global ou atualizam o estado local.

```
1  // Retorna sempre 15 bytes:
2  //      7 (reception_matrix) + 7 (detection_vector) + 1 (consensual_identity)
3  ssize_t read_mod(struct file* file, char * buf, size_t count, loff_t *l) {
4      int c;
5      int detection_vector=0;
6      char saida;
7
8      // Verifica se o buffer fornecido é válido
9      if (count != 9) return -1;
10     if (verify_area(VERIFY_WRITE, buf, count) == -EFAULT) return -EFAULT;
11
12     for (c = 0; c < 7; c++) {   // Escreve a reception_matrix
13         put_user(estado_global[c] & 0x7F, buf);
14         buf++;
15     }
16
17     for (c = 0; c < 7; c++) {   // Escreve o detection_vector
18         if ((estado_global[c] & 0x80) != 1) {
19             saida = (char) 1;
20             put_user(&saida, buf++);
21         } else {
22             saida = (char) 0;
23             put_user(&saida, buf++);
24         }
25     }
26     for (c = 0; c < 7; c++) {   // Escreve o consensual_identity
27         if (terminado & (1 <<c)) {
28             saida = c;
29             put_user(&saida, buf++);
30             return count;
31         }
32     }
33     saida = (char) 255;
34     put_user(&saida, buf++);    // Escreve o consensual_identity vazio
35     return count;
36 }
```

Figura A.2: Código-fonte - Compilador de Estados Globais (CEG) (parte 2/3)

A função $read\_mod$ é chamada sempre que o arquivo especial é lido. Neste caso, o arquivo deve ser lido em blocos de 15 bytes, que contêm a matriz $reception\_matrix$ (7 bytes), o vetor $detection\_vector$ (7 bytes) e o identificador $consensual\_identity$ (1 bytes). Quando é chamada, a função $read\_mod$ transforma a variável $estado\_global$, atualizada sempre que um novo conjunto de mensagens chega do *Wormhole*, nas estruturas definidas pelo CEG: a matriz $reception\_matrix$ é formada pelo 7 bits menos significativos de cada um dos 7 bytes dos estados locais de cada processo (linhas 18 a 21); o vetor $detection\_vector$ é composto de 7 bytes com valor 0 ou 1 de acordo com o bit mais significativo de cada estado local rece-

bido; por fim, o $consensual\_identity$ é calculado a partir do bit menos significativo presente em todos os estados locais, ou seja, o menor identificador cuja mensagem foi recebida por todos os processos.

A última parte do código-fonte do módulo pode ser vista na Figura A.3. A função $write\_mod$ é chamada sempre que um acesso de escrita é feito no arquivo especial. Esta função aceita acessos de 1 e 2 bytes. No acesso de 1 byte, o byte é convertido no estado local da máquina, que será enviado para os outros nós que executam o serviço. Neste exemplo, o *heartbeat* é informado pela própria aplicação através do estado local. No acesso de 2 bytes, é verificado se o código de encerramento do serviço foi fornecido e em caso afirmativo encerra o serviço.

Por fim, a função $close\_mod$ simplesmente imprime uma mensagem quando o arquivo especial é fechado.

```
1  // Aceita 1 bytes = estado local, 2 bytes = código de encerramento
2  ssize_t write_mod(struct file* file, const char * buf, size_t count, loff_t *l) {
3      char entrada[2];
4
5      if (count > 2) return -1;
6      if (verify_area(VERIFY_READ, buf, count) == -EFAULT) return -EFAULT;
7      if (count == 2) {
8          get_user(entrada[0], buf[0]);
9          get_user(entrada[1], buf[1]);
10         if ( (entrada[0] == 0xAA) && (entrada[1] == 0x55)) {
11             wh_remove_service("CEG");
12             return count;
13         }
14         return -1;
15     }
16
17     get_user(&entrada, buf[0]);
18     estado_local=(char) saida;
19     estado_local |= 0x80; // Ativa o heartbeat
20     return count;
21 }
22
23 int release_mod(struct inode *inode, struct file * file) {
24     printk("Fechando arquivo especial!\n");
25     return 0;
26 }
```

Figura A.3: Código-fonte - Compilador de Estados Globais (CEG) (parte 3/3)

A função que faz o processamento periódico do serviço é ativada para execução pela rotina de tratamento de interrupção do *driver* do *Wormhole*. Logo, o serviço precisa ser cons-

truído na forma de módulo do núcleo para que não seja removido da memória (*swapped-out*). O protocolo de consenso em si é implementado na parte assíncrona, como uma aplicação comum de usuário. Por questão de simplicidade, apenas um esboço do algoritmo de consenso é apresentado na Figura A.4. Na linha 12, o arquivo especial $/dev/CEG$ (ou qualquer outro com número maior 252 e número menor 31) é aberto para escrita e leitura da mesma forma que um arquivo comum. Este arquivo é a interface entre a aplicação e o serviço de controle de admissão do *Wormhole*. Na linha 18, o processo escreve o comando que solicita a iniciação do serviço CEG, se bem sucedido, o arquivo aberto será imediatamente conectado ao módulo do CEG, caso contrário um código de erro será retornado. Na linha 24, o processo faz a difusão de sua proposta. Em seguida entra em uma laço que executa repetidas vezes até que um valor do identificador $consensual\_identity$ seja fornecido. Este laço realiza as seguintes tarefas: (1) recebe alguma mensagem de proposta que esteja na fila de entrada da rede assíncrona; (2) calcula o novo estado com base nas mensagens recebidas (ajustando o bit do estado de acordo com o identificador contido nas mensagens); (3) envia seu novo estado para o CEG; (4) lê o novo resumo do estado global, dividindo-o em seus componentes $reception\_matrix$, $detection\_vector$ e $consensual\_identity$. Por fim, depois que um identificar válido for recebido, o algoritmo recupera a respectiva mensagem do buffer de mensagens e decide por ela.

Neste exemplo, as estruturas $detection\_vector$ e $reception\_matrix$ não são utilizadas pois trata-se de um Consensus-CEG(n,n). Desta forma, antes de uma decisão ser possível (e $consensual\_identity$ seja preenchido) todos os processos precisam receber uma dada mensagem de proposta e atualizar o seu estado local no CEG. Em outros protocolos de consenso, poderia ser necessário que o processo enviasse a mensagem de decisão para todos os processos corretos que não tivessem recebido a respectiva mensagem (por exemplo, por causa da falha do enviador).

```
void consenso(void) {
    int terminado=0, ceg, i;
    char proposta_consenso[TAMANHO_PROPOSTA], reception_matix[7], detection_vector[7];
    char consensual_identity, estado_local, buffer[15];
    char mensagens_recebidas[TAMANHO_BUFFER];
    char cmd_inicia_ceg[]={ "CEG", 0, 0, 0, 0, 0, 0, 0, /*nome do serviço*/
                            1, /* periodicidade */
                            0, 0, 0, 0, 0, 0, 0, 0, 0, 0 /*parâmetros do serviço*/};

    proposta_consenso=constroi_proposta();

    ceg=open("/dev/CEG", O_RDWR); /* conecta com o serviço de controle de admissão */
    if (ceg == -1) {
        printf("\n Erro abrindo o arquivo do controle de admissão!");
        return;
    }

    i=write(ceg, cmd_inicia_ceg, 21); /* wh_request_service */
    if (i < 0) {
        printf("\n Erro na iniciação do serviço! ");
        return;
    }

    envia_broadcast(proposta_consenso);
    while (consensual_identity == 255) {
        // Recebe mensagens assíncronas
        // armazenando-as no buffer "mensagens_recebidas"
        recebe_mensagens(mensagens_recebidas);

        // Calcula o estado a partir das mensagens recebidas
        estado_local=recupera_estado(mensagens_recebidas);

        estado_local |= 0x80; /* ativa o "heartbeat" */

        write(ceg, estado_local, 1);
        read(ceg, buffer, 15);

        for (i = 0; i < 7; i++) reception_matrix[i]=buffer[i];
        for (i = 7; i < 14; i++) detection_vector[i-7]=buffer[i];
        consensual_identity=buffer[14];
    }
    decide(consenual_identity, mensagens_recebidas);
}
```

Figura A.4: Esboço do aplicativo de consenso