Projeto e Implementação de um Serviço de Detecção de Falhas Perfeito*

Ely W. de Oliveira, Andrey E. M. Brito, Francisco V. Brasileiro

Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Laboratório de Sistemas Distribuídos
Av. Aprígio Veloso, 882, Campina Grande, Paraíba, 58.109-970, Brazil
Tel: (+55) 83 310 1433 Fax: (+55) 83 310 1365

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

Abstract. *An unreliable failure detector is an important abstraction to support the implementation of fault-tolerant protocols on distributed asynchronous systems. Several classes of failure detectors, with varying semantics, have been proposed. The class of perfect failure detectors provides the strongest semantics. In this paper we present the design and implementation of a failure detection service with perfect semantics. The service is implemented at the operating system level by adding extra system calls to a standard Linux kernel. We have also implemented both C and Java APIs to provide applications with access to the service. Special low-cost wireless communication devices have been built to support the service. These devices are connected to each machine running the service in a local area network. They create an ad-hoc network that is used solely to convey messages of the failure detection service.*

Resumo. *Um detector de falhas não-confiável é uma importante abstração para viabilizar a implementação de protocolos tolerantes a falhas em sistemas distribuídos assíncronos. Algumas classes de detectores de falhas, com semânticas variadas, foram propostas. A classe dos detectores de falhas perfeitos apresenta a semântica mais forte. Neste trabalho nós apresentamos o projeto e a implementação de um serviço de detecção de falhas com semântica perfeita. O serviço é implementado no nível do sistema operacional através da adição de chamadas de sistema extras ao núcleo padrão do Linux. Nós também implementamos APIs em C e Java para disponibilizar às aplicações acesso ao serviço. Dispositivos sem fio de baixo custo foram construídos para viabilizar o serviço. Estes dispositivos são conectados em cada máquina de uma rede local que execute o serviço. Eles criam uma rede ad-hoc que é utilizada exclusivamente para o tráfego de mensagens do serviço de detecção de falhas.*

1 Introdução

A maioria das infraestruturas disponíveis para implantação de aplicações distribuídas são caracterizadas pela ausência de limites superiores nos atrasos de transmissão de mensa-

*Este trabalho foi financiado com recursos do CT-PETRO, ANP, e CNPq/Brazil (processo 300.646/96).

gens e de escalonamento, *i.e.* elas são sistemas assíncronos. O conhecido resultado apresentado pelo trabalho de Fischer *et al.* [11] prova que é impossível atingir consenso [6] (requisito básico para diversos mecanismos para tolerância a faltas) em um sistema distribuído assíncrono sujeito a faltas. Este resultado pode ser resumido da seguinte forma: devido às incertezas no envio e entrega das mensagens, é impossível distinguir entre um processo que falhou e um que está muito lento.

A partir deste resultado, foram propostos vários modelos que adicionam algum sincronismo ao modelo assíncrono puro, permitindo soluções para o problema do consenso. O modelo assíncrono equipado com um detector de falhas não-confiável é um desses modelos. Ele assume a existência de um "oráculo", o detector de falhas, que 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 precisa o suficiente para permitir soluções determinísticas para o problema do consenso [6].

A semântica de um detector de falhas é caracterizada por duas propriedades: *abrangência* e *exatidão*. A primeira propriedade define quão abrangente é a detecção, enquanto a segunda restringe os erros que o detector pode cometer. Quanto mais forte é a semântica do detector de falhas, menos restritivas são as considerações necessárias para garantir a corretude de protocolos de mais alto nível que o utilizam [6]. Por outro lado, mais restritivas são as considerações feitas sobre o sistema em que executam, para que suas propriedades possam ser atendidas. Por causa disso, a maioria dos trabalhos práticos sobre detectores de falhas presentes na literatura são implementações de detectores de falhas da classe $\diamond S$ ([10, 12], por exemplo), a classe mais fraca de detectores de falhas que permitem uma solução determinística para o problema do consenso [7].

Um detector de falhas com semântica perfeita é o mais forte detector de falhas e é definido pelas seguintes propriedades [6]: *i)* abrangência forte: após um tempo, todo processo que falha por parada é permanentemente suspeitado por todos os processos corretos; e *ii)* exatidão forte: um processo correto nunca é suspeitado por processo algum. Detectores de falhas perfeitos não podem ser implementados em sistemas assíncronos[1]. Entretanto, a adoção de algumas restrições no sistema, como por exemplo, a instalação de um segundo canal de comunicação dedicado à troca de mensagens do serviço de detecção, ou alterações no sistema operacional, pode criar um mínimo de sincronismo no sistema, necessário para que as propriedades da semântica perfeita de detectores de falhas possam ser garantidas.

Outro problema na implementação de detectores de falha é que eles são especificados em termos de um comportamento "eventual" (após um tempo finito o detector apresentará um certo comportamento). Essa especificação é adequada para sistemas assíncronos puros, entretanto, aplicações frequentemente 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 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 [9].

[1]Se pudessem, o consenso poderia ser resolvido no ambiente assíncrono, o que foi provado ser impossível por Fischer *et al.* [11].

Neste trabalho, nós apresentamos o projeto e a implementação do Delphus, um serviço de detecção de falhas baseado em detectores de falhas da classe P (perfeito). O Delphus oferece garantias de qualidade de serviço e se destina a redes locais com pouca dispersão geográfica. O serviço é implementado no nível do sistema operacional e é construído com o suporte de uma rede de comunicação adicional que é utilizada exclusivamente para o tráfego de mensagens do serviço. Na medida que a utilização desta rede adicional é cuidadosamente controlada e atenção especial é dada ao escalonamento das *threads* do sistema operacional que implementam o núcleo do serviço de detecção de falhas em cada máquina, os atrasos máximos na comunicação podem ser definidos com probabilidade muito alta, criando um ambiente síncrono de execução sobre o qual o serviço de detecção de falhas é implementado. A rede síncrona é implementada por um dispositivo de comunicação sem fio de baixo custo especialmente desenvolvido.

Uma abordagem semelhante é adotada pelo TCB (*Timely Computing Base*), um *framework* apresentado por Veríssimo e Casimiro [13], cujo objetivo é fornecer suporte a aplicações que necessitam executar tarefas com requisitos temporais. Dentre seus serviços estão a execução de tarefas em instantes exatos de tempo, ou respeitando prazos, e a detecção de falhas nas execuções destas tarefas. Este serviço de detecção possui semântica perfeita e considera falhas temporais, que englobam as falhas por parada abordadas por Chandra e Toueg [6]. Para a implementação do TCB, um módulo síncrono deve ser construído em cada máquina do sistema. Em outro trabalho [5], Casimiro *et al.* mostram como este módulo pode ser construído utilizando o Real-Time Linux e uma rede síncrona privativa para comunicação.

A maior diferença entre o Delphus e o TCB é que o Delphus se proprõe exclusivamente a fornecer um serviço de detecção de falhas com semântica perfeita, ao invés de serviços de suporte para computação temporal. Ele também se destina a redes locais que utilizam versões genéricas do Linux. Além do mais, o TCB exige que as aplicações a serem monitoradas sejam escritas especialmente para sua utilização. Isso aumenta o grau de acoplamento das aplicações com o serviço, reduzindo sua portabilidade. O Delphus se propõe a ser um componente de fácil adoção e utilização pela maioria dos sistemas distribuídos, sem exigir a implantação de um sistema operacional de tempo real, nem maiores adaptações das aplicações clientes.

O restante deste trabalho está estruturado da seguinte forma. A Seção 2 apresenta as principais idéias por trás do projeto do serviço de detecção de falha. Os protocolos necessários à implementação e a interface de programação fornecida às aplicações são apresentados na Seção 3. Nesta seção são discutidos tanto os aspectos de *software* quanto de *hardware*. Aspectos de performance são também discutidos nessa seção. Finalmente, conclusões e trabalhos futuros são apresentados na Seção 4.

2 Projeto do Serviço

2.1 Modelo do Sistema

O sistema considerado neste trabalho é formado por um ambiente assíncrono, equipado com um subsistema síncrono. O ambinete assíncrono é o ambiente onde o sistema operacional e as aplicações executam e trocam mensagens. O subsistema síncrono é um canal

de comunicação confiável, usado exclusivamente para troca de mensagens do serviço de detecção de falhas, e um temporizador cão-de-guarda.

Uma série de cuidados foram adotados ao construir este canal de comunicação, para garantir que as mensagens não sejam perdidas ou corrompidas, e ainda, que atinjam seus destinatários em um limite de tempo definido. As transmissões entre os dispositivos de comunicação são feitas repetidas vezes e em várias frequências de rádio. Além do mais, o canal de comunicação tem seu uso controlado, evitando assim sobrecarga, o que poderia atrasar transmissões. Um código de detecção de erros é utilizado durante a recepção das mensagens para verificar a integridade das cópias recebidas e descartar as que tiverem sido corrompidas.

O tipo de falhas considerado por este trabalho é a por parada. Consideramos que um processo pára de executar quando seu identificador não faz mais parte da tabela de processos do sistema operacional ou quando seu estado passa a ser *zumbi* [3].

2.2 Descrição do Serviço

Antes de apresentar a interface de programação disponibilizada pelo Serviço de detecção de falhas, nós apresentamos os conceitos principais que suportam o projeto do mesmo. O serviço de detecção de falhas é utilizado por dois tipos de entidades: *monitoráveis* e *notificáveis* [10]. Tanto os processos como as máquinas podem ser entidades monitoráveis, *i.e.* eles podem ser submetidos à monitoração, e neste caso, suas falhas devem ser detectadas pelo serviço. Por outro lado, entidades notificáveis são processos interessados em ser notificados da ocorrência de falhas das entidades monitoráveis. Além disso, qualquer processo pode explicitamente consultar o serviço de detecção de falhas sobre a ocorrência de falhas de quaisquer entidades monitoráveis.

Outro conceito importante é o de domínio de detecção. O serviço utiliza uma rede privada e síncrona para troca de mensagens de monitoração entre os módulos que, em conjunto, implementam o serviço. Esta rede define um domínio de detecção[2]. Todas as entidades associadas a um domínio de detecção em particular são unicamente identificadas por um identificador global.

O serviço de detecção de falhas é implementado por módulos de detecção de falhas independentes executando em cada máquina participante de um domínio de detecção em particular. Cada módulo implementa um detector de falhas do estilo *push* [10] que periodicamente envia mensagens de *heartbeat* de suas entidades monitoráveis locais. Essas mensagens são enviadas pela rede síncrona. A utilização desta rede síncrona obedece um protocolo TDMA (*Time Division Multiple Access*). No TDMA a utilização do canal é dividida em fatias de tempo (*slots*). Cada módulo tem sua própria fatia de tempo (de comprimento fixo) para transmitir suas mensagens de *heartbeat*. Sempre que um *heartbeat* de uma entidade monitorável está ausente, todo módulo correto detecta a falha desta entidade. Então, cada módulo notifica todas as entidades notificáveis locais que solicitaram uma notificação para a falha daquela entidade monitorável cuja falha foi detectada. Cada módulo tem ao menos uma entidade monitorável; esta entidade é a máquina na qual o módulo executa. Desta forma, em cada período, todo módulo deve utilizar sua fatia de tempo no canal para transmitir ao menos o *heartbeat* da sua máquina.

[2]No momento nós consideramos apenas um domínio de detecção, mas esperamos estender a atual arquitetura de domínio único para uma arquitetura multi-domínio.

Os módulos independentes precisam estar sincronizados entre si para implementar o protocolo TDMA. Para isso, eles utilizam um protocolo baseado em um líder. Sempre que houver ao menos um módulo executando o serviço, haverá exatamente um módulo como líder. Além de enviar os *heartbeats* de suas entidades monitoráveis locais, o líder é responsável também pelo envio de uma mensagem de sincronização no início de cada período TDMA. Esta mensagem de sincronização é utilizada pelos outros módulos para a ressincronização.

A utilização de um protocolo baseado em líder introduz dois problemas a serem resolvidos: i) como definir quem é o primeiro líder, na iniciação do serviço; e ii) como lidar com falhas do líder.

O principal problema enfrentado pelo serviço durante a fase de iniciação é descobrir se uma máquina é a primeira a se inscrever em um domínio de detecção ou não. Se esta máquina for a primeira, ela deve assumir a liderança do domínio e tornar-se responsável pela coordenação dos períodos TDMA. Caso contrário, ela deve encontrar o líder e requisitar sua inscrição no domínio.

O primeiro passo para um módulo que está iniciando é escutar o canal síncrono por um intervalo de tempo mínimo predefinido, o qual é longo o suficiente para receber uma mensagem de sincronização de qualquer líder já eleito. Se uma mensagem de sincronização for recebida, a máquina é capaz de identificar o líder do domínio. Caso contrário, a máquina tenta se impor como líder enviando uma mensagem de proposta de liderança no canal síncrono. Então, a máquina aguarda um período de tempo longo o suficiente para receber outras mensagens de proposta concorrentes de outras máquinas. Se apenas a sua proposta de liderança for ouvida, dentro do período de espera, ela se considera a líder do domínio. Se mais de uma proposta de liderança for recebida, a máquina espera por um período de tempo aleatório e repete todos os passos de iniciação.

Tolerar falhas do líder é uma tarefa mais simples. Falhas do líder impactam o protocolo apenas quando há ao menos um outro módulo correto em operação no mesmo domínio de detecção. Logo, a ausência de uma mensagem de sincronização pode ser utilizada para indicar a falha do líder. Como será explicado em breve, após o estabelecimento do líder no procedimento de iniciação, todo novo módulo precisa explicitamente requisitar ao líder sua inscrição no domínio. O líder responde tal requerimento anunciando todo novo membro, um em cada período TDMA. Este anúncio utiliza uma fatia de comprimento fixo no período TDMA, e é recebida por todo membro que já está inscrito (além do membro que se inscreve). Desta forma, este procedimento estabelece uma ordenação para os módulos que já se inscreveram no serviço. Esta ordenação é utilizada para definir o novo líder no momento que uma falha do líder é detectada (esta ordenação também é utilizada para definir em qual fatia de tempo do período TDMA o módulo deve transmitir seus *heartbeats*). O módulo escolhido para ser o novo líder imediatamente se torna ciente sobre seu novo papel. Na próxima fatia de tempo de sincronização, ele irá começar a enviar mensagens de líder (sincronização e anúncio).

Como mencionado anteriormente, após a mensagem de sincronização, todos os membros do domínio devem escutar a outra mensagem enviada pelo líder no canal síncrono: a mensagem de anúncio. Cada mensagem de anúncio contém informação sobre uma única inscrição ou remoção. O anúncio de um novo monitorável contém toda

informação relevante para definir a identificação da entidade e a frequência com que os *heartbeats* daquela entidade precisam ser transmitidos. Por outro lado, o anúncio de uma remoção contém apenas a identificação da entidade a ser removida. Depois de recebido o anúncio, todos os membros atualizam suas listas de monitoráveis, assegurando que todos têm o mesmo conjunto de monitoráveis e a mesma informação de configuração para eles.

Antes de realizar os anúncios, o líder precisa receber requerimentos de inscrição ou remoção dos módulos. A rede assíncrona é utilizada para permitir toda a comunicação necessária antes da realização do anúncio. Um requerimento para inscrever uma máquina na lista de monitoráveis é realizada sempre que o serviço é iniciado na máquina, enquanto que um requerimento de remoção para uma máquina é feito sempre que o serviço é encerrado. Depois que o serviço é iniciado em uma máquina em particular, processos naquela máquina podem solicitar inscrições ou remoções. Estes requerimentos são endereçados ao módulo local que os redireciona para o líder. Antes de redirecionar uma requisição de inscrição de um processo local, cada módulo executa um protocolo de admissão que verifica se a qualidade de serviço requisitada (expressa pelo limite superior da latência de detecção requerida) pode ser garantida. Se não, a chamada de sistema de inscrição retorna um erro para o processo solicitante. Caso contrário, o módulo continua o procedimento de inscrição.

Quando um líder recebe um requerimento de inscrição através da rede assíncrona, ele verifica se a identificação requisitada já esta em uso. Se estiver, uma mensagem de recusa é enviada para o requisitante e o procedimento de inscrição é encerrado. Caso contrário, a ação tomada irá depender do tipo de entidade que está requisitando a inscrição. Se a entidade monitorável for um processo, o líder o escalona para ser anunciado na próxima mensagem de anúncio disponível. Se a entidade é uma maquina, o líder envia ao módulo requisitante uma lista com todas as entidades monitoráveis correntes e o número do último período em que a lista foi atualizada. No recebimento desta mensagem, o módulo atualiza seus dados locais, envia uma mensagem de confirmação para o líder, e então espera pelo seu anúncio. A partir do momento que um módulo envia sua primeira requisição (relativa a sua máquina), ele começa a escutar mensagens de anúncio. Isto garante que sua informação local, construída após o recebimento da lista de entidades monitoráveis do líder, será consistente com a lista de todos os outros módulos. Apenas após receber a mensagem de confirmação, o líder escalona o anúncio da entidade máquina. Em todos os casos, o procedimento de inscrição/remoção é concluído apenas quando o anúncio correspondente é recebido através do canal síncrono, ou uma mensagem de recusa é recebida através do canal assíncrono.

Como um canal assíncrono é utilizado para enviar as mensagens do procedimento de inscrição/remoção, é possível que algumas delas sejam perdidas ou atrasadas. Dessa forma, tanto o módulo de origem da requisição, quanto o líder, possuem temporizadores para detectar quando devem retransmitir uma mensagem. Eles devem também estar preparados para descartar duplicatas. Além disso, no caso de particionamento da rede assíncrona, um número máximo de retransmissões pode ser alcançado, e o procedimento de inscrição/remoção irá terminar e retornar um erro para o processo requisitante.

2.3 Interface de Programação de Aplicação (API)

De modo a permitir a utilização do serviço por aplicações distribuídas, nós definimos uma API (*Aplication Programming Interface*), dividida em três grupos de funções: administração do serviço, configuração de monitoramento e notificação de falha. Esta API já foi implementada em C e em Java, mas aqui a apresentamos utilizando a notação em C, que é mais concisa e simples de ser explicada. Nós apresentamos as principais funções, mostrando suas assinaturas, explicando seu comportamento, e descrevendo seus argumentos e valores de retorno. Quando omitido, o retorno da função é simplesmente um código de erro indicando se ela executou com sucesso ou não.

2.3.1 Administração do Serviço

O serviço precisa prover um modo de ser gerenciado por usuários especiais como administradores de sistema. Desta forma, um conjunto de funções foi definido para permitir que os módulos distribuídos do serviço possam ser iniciados e interrompidos, de acordo com as necessidades do sistema.

int start_monitoring(char *id) Esta função é utilizada para inscrever uma máquina em um domínio de detecção. Ela recebe o identificador proposto para a máquina (*id*) no domínio de detecção.

int stop_monitoring() Remove a máquina, onde o processo invocador está executando, do domínio de detecção.

2.3.2 Notificação de Falha

A forma mais simples de se estar ciente das falhas é consultar o estado das entidades monitoráveis. A função a seguir pode ser utilizada para recuperar esta informação.

int is_correct(char *id, struct timeval *time) Esta função informa se um monitorável está correto ou não. Ela recebe como argumento o identificador no serviço da entidade monitorável cujo estado está sendo consultado. Retorna o valor 1 se o monitorável estiver correto, e 0 caso contrário. O argumento *time* é utilizado para armazenar o tempo local no qual a informação foi coletada.

Existe também uma forma mais sofisticada de ser informado sobre a ocorrência de falhas. É possível configurar o serviço para notificar um processo sobre a ocorrência de falhas em entidades monitoráveis. A notificação é realizada através de um sinal do sistema operacional enviado ao processo notificável. Esta funcionalidade é disponibilizada pelas seguintes funções.

int add_notifiable(char *n_id, int pid, char *m_id, unsigned long int signal) Esta função configura um processo para ser notificado da ocorrência de falha em uma entidade monitorável específica. Seus argumentos são: *n_id*, a identificação no serviço, da entidade notificável; *pid*, o identificador do processo da entidade notificável (se omitido, o *pid* do processo que fez a invocação será utilizado); *m_id*, o identificador da entidade monitorável sendo monitorada; e *signal*, o sinal do sistema operacional que será enviado para a entidade notificável no caso da detecção de falha na entidade monitorável especificada.

int remove_notifiable(char *id) Remove a entidade notificável da lista de notificáveis de uma entidade monitorável especifica. Ela recebe como parâmetro a identificação, no serviço, da entidade notificável.

2.3.3 Configuração de Monitoração

Para configurar os processos que precisam ser monitorados pode-se usar as funções descritas abaixo.

int add_monitorable(char *id, int pid, unsigned long int detection_time) Esta função é utilizada para adicionar um processo novo para ser monitorado pelo serviço. Ela possui os seguintes argumentos: *id*, o identificador no serviço, para o processo; *pid*, o identificador do processo da entidade monitorável (se omitido, o *pid* do processo invocador será utilizado); e *detection_time*, o intervalo máximo de tempo, com precisão de milisegundos, dentro do qual uma falha no processo deve ser detectada e conhecida por todos os módulos do serviço.

O último parâmetro da função anterior define a qualidade do serviço requisitada. É importante notar que a latência máxima de detecção é garantida apenas com relação ao tempo que o módulo de detecção de falha local detecta a falha da entidade monitorável e envia os sinais apropriados para a entidade notificável correspondente. Nenhuma garantia é dada para o tempo que a entidade notificável vai efetivamente tratar o sinal. Por outro lado, como será explicado mais tarde, qualquer processo pode consultar seu módulo de detecção de falhas local para saber se uma entidade monitorável está correta. Quando o serviço retorna a informação que uma entidade está correta no tempo T, é garantido que a entidade estava correta no tempo $T - detection\_time$.

int remove_monitorable(char *id) Remove um processo da lista corrente de monitoráveis. Recebe como argumento, apenas a identificação no serviço do processo a ser removido.

2.4 Exemplo da utilização da API do Serviço

Dois programas foram implementados como um exemplo simples da utilização da API do serviço: *Monitoravel* e *Monitor*. *Monitoravel* chama a função *add_monitorable* para configurar a si mesmo como uma entidade monitorável. Os argumentos utilizados são: ”Mon1”, como seu identificador no serviço, e $1000ms$ como o tempo máximo para detecção de suas falhas. A Figura 1 apresenta a porção relevante do código fonte do programa *Monitoravel*.

Monitor utiliza a função *add_notifiable* para configurar a si mesmo como uma entidade notificável. Ela será notificada na falha de *Monitoravel*. O sinal escolhido para ser enviado pelo núcleo para *Monitor* como aviso da notificação foi SIGALRM. O manipulador de sinais do programa é chamado quando o programa recebe um sinal SIGALRM. Ele imprime na tela uma mensagem indicando que foi notificado do término do programa *Monitoravel*.A Figura 2 apresenta apresenta a porção relevante do código fonte do programa *Monitor*.

```
int main(int argc, char *argv[]) {
        char c;
        // Inicia o serviço. Identifica máquina como Maq1
        start_monitoring("Maq1");

        // Configura o processo atual para ser monitorado
        add_monitorable("Mon1", 1000);
        printf("Monitorável está executando. ");
        printf("Tecle ENTER para sair.");
        scanf("%c", &c);
}
```

Figura 1: Monitoravel.c

```
void handler(int sig) {
        printf("Monitoravel foi encerrado!");
        exit(0);
}

int main(int argc, char *argv[]) {
        struct sigaction sa;
        sa.sa_handler = handler;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sigaction(SIGALRM, &sa, NULL);

        // Inicia o serviço. Identifica máquina como Maq2
        start_monitoring("Maq2");

        // Configura o processo atual como notificável
        add_notifiable("Not1", "Mon1", SIGALRM);

        // Aguarda notificação
        printf("Monitor está aguardando a notificação.");
        while(1);
}
```

Figura 2: Monitor.c

3 Implementação do Serviço

O serviço de detecção de falha associado a um domínio de detecção é implementado parte no núcleo do sistema operacional executando em cada máquina, e parte nos dispositivos de comunicação que implementam a rede síncrona. Estas duas partes são descritas nas próximas duas subseções.

3.1 Descrição do Software

3.1.1 Threads do Núcleo

Um módulo local do serviço é implementado por três *threads* no núcleo do sistema operacional. A *thread do_period* é responsável pela implementação das principais funcionalidades do serviço. Além disso, ela envia as mensagens na rede síncrona. A *thread sync_rec* é responsável por receber mensagens pela rede síncrona. Ela permanece a maior parte do tempo bloqueada, esperando pela chegada de mensagens no canal síncrono. Depois de

receber uma mensagem, ela identifica o tipo da mensagem, e invoca o manipulador específico para processá-la. Nós definimos quatro tipos de mensagens síncronas[3], são elas: *proposta de liderança*, *sincronização*, *anúncio* e *heartbeat*.

Estas duas *threads* implementam o núcleo do serviço de detecção de falhas, e são configuradas para terem a maior prioridade de tempo real disponível. *Threads* com esta prioridade têm privilégios na política de escalonamento do Linux, e são escalonados tão logo se tornem prontas para executar, *i.e.* seu atraso de escalonamento é limitado pela resolução das interrupções do relógio ($1ms$ no nosso sistema).

A terceira *thread*, *async_rec*, é a versão assíncrona do *sync_rec*. Ela recebe mensagens através da rede assíncrona. Existem cinco tipos de mensagens que podem ser recebidas, elas são: *requisição de inscrição*, *requisição de remoção*, *recusa de inscrição*, *atualização* e *confirmação*. A única tarefa desta *thread* é implementar os procedimentos de inscrição e remoção, gerenciando as mensagens de anúncio que devem ser realizadas na fatia de tempo destinada aos anúncios pelo líder. Na medida que ela não está associada a prazos, esta *thread* não tem uma prioridade de tempo real.

3.1.2 Iniciação e Execução do Serviço

A iniciação do serviço é realizada através da chamada à função *start_monitoring*. Ela repassa sua invocação para a chamada de sistema *sys_start_monitoring*, cujo primeiro passo é iniciar as *threads sync_rec* e *async_rec* no núcleo. Então, ela força o processo que fez a invocação a dormir por $n - 1$ períodos TDMA, onde n é o número máximo de máquinas que podem fazer parte de um domínio de detecção[4]. Quando ela acorda, ela verifica se o líder do domínio ainda é desconhecido. Isto só acontecerá se nenhuma mensagem de sincronização for recebida pela *thread sync_rec* durante o tempo em que o processo estava dormindo. Neste caso, é iniciado o protocolo para estabelecer o primeiro líder no domínio. Isto é implementado enviando uma mensagem de proposta de liderança, dormindo por um tempo e, logo após, verificando quantas mensagens de proposta de liderança foram recebidas, como explicado na Seção 2.

Após a descoberta do líder, se o líder não é a máquina que está iniciando, um pedido de inscrição no domínio deve ser enviado para o líder. O procedimento de inscrição utiliza a rede assíncrona como explicado na Seção 2. Se o processo de inscrição for bem sucedido[5], a *thread do_period* é criada e um código de sucesso é retornado para o processo que iniciou a requisição.

A *thread do_period* executa um laço que continuamente envia mensagens síncronas nas suas fatias de tempo do protocolo TDMA. As mensagens enviadas dependem do módulo ser líder do domínio ou não. Desta forma, o primeiro passo a ser tomado pela *thread do_period* é testar se o módulo local é o líder corrente. Neste caso,

[3]Nós iremos usar os termos mensagem síncrona/assíncrona denotando mensagens que são enviadas através da rede síncrona/assíncrona.

[4]A espera por $n - 1$ períodos TDMA está associada ao pior caso, onde já existem $n - 1$ máquinas no domínio, e todas elas falharam enquanto uma nova máquina estava iniciando.

[5]O procedimento de inscrição pode falhar se um particionamento acontecer na rede assíncrona, ou se o líder corrente falhar. No primeiro caso, um erro é retornado ao processo que fez a solicitação, enquanto no último, um novo procedimento de inscrição é iniciado assim que um novo líder for eleito.

primeiro uma mensagem de sincronização é enviada. Esta mensagem contém o número do período corrente e o endereço IP do líder. Em seguida, a *thread do_period* executando no líder envia uma mensagem de anúncio, se necessária. Em seguida, a *thread do_period* de qualquer membro (líder ou não) dorme até que sua fatia de *heartbeat* comece, e então envia sua mensagem de *heartbeat*. Finalmente, ela dorme até o início do próximo período TDMA.

A *thread do_period* é também responsável pela detecção de falhas de líderes. Isto acontece com a ajuda da *thread sync_rec*. Sempre que uma mensagem de sincronização é recebida, o manipulador *receive_synchronization* é executado. Esta função atualiza as variáveis *current_period* e *period_start_time*. A variável *current_period* é incrementada cada vez que uma mensagem de sincronização é recebida. Uma falha do líder é detectada pela *thread do_period* quando ela está pronta para enviar sua mensagem de *heartbeats* e descobre que a variável *current_period* não mudou desde a última vez que uma mensagem de *heartbeats* foi enviada. Isto implica que entre as duas mensagens consecutivas, nenhuma mensagem de sincronização foi recebida, e portanto o líder falhou. Neste momento, um novo líder é eleito, utilizando a ordem com que as máquinas se inscreveram no domínio de detecção. A variável *period_start_time* é utilizada para ressincronizar relógios. Temporizadores são calculados utilizando o tempo da última mensagem de sincronização recebida. Este tempo é utilizado para calcular o início do período TDMA.

3.1.3 Calculando o Tamanho do Período TDMA

Crucial para o funcionamento correto do serviço é garantir que, a qualquer momento, apenas uma máquina está transmitindo informação através da rede síncrona. Logo, manter os relógios das máquinas sincronizados, assim como precisamente definir o tamanho de cada fatia de tempo no período TDMA, são aspectos fundamentais na implementação do serviço.

Como discutido previamente, os relógios de todas as máquinas em um domínio de detecção são mantidos sincronizados por mensagens de sincronização enviadas pelo líder. Sempre que uma mensagem de sincronização é recebida, quaisquer desvios que tenham sido acumulados desde a última recepção de uma mensagem de sincronização são compensados. A diferença máxima entre os tempos que quaisquer duas máquinas recebem uma mensagem de sincronização é o erro de sincronização (T_{sinc}). Para nossa implementação, o erro de sincronização é dado pela soma de três fatores: (1) a diferença máxima de recebimento das mensagens no canal síncrono, dada pelo produto da velocidade da luz pela distância máxima entre quaisquer duas máquinas, que é da ordem de $10^{-5}ms$, e portanto desprezível; (2) o tempo de acionamento da rotina de tratamento de interrupção depois de recebida a mensagem de sincronização - este tempo representa quanto o tratamento de uma interrupção pode ser adiado e, experimentalmente, valores de $1ms$ são bastante realistas[6]; e (3) o tempo de processamento da sincronização, que corresponde a uma rotina rápida de tratamento de interrupções - diferenças na velocidade de execução desta rotina são desprezíveis.

Por outro lado, o desvio máximo acumulado é dado por $T_{drift} = (n-1) \times (n +$

[6]O temporizador cão-de-guarda é utilizado para garantir que a síncronização não é realizada depois deste tempo limite.

$2) \times T_{slot} \times (2\rho)$, onde T_{slot} é a duração da fatia de tempo TDMA e $\rho = 10^{-4}$ é a taxa máxima de desvio do relógio de qualquer máquina[7].

Para calcular o tamanho da fatia de tempo TDMA, diversas variáveis precisam ser consideradas. Algumas delas são fontes de incerteza que podem invalidar o cálculo se não forem propriamente consideradas.

A primeira fonte de incerteza é o tempo máximo para o escalonamento de uma *thread* depois que ela se torna pronta para executar. No pior caso, uma *thread* atinge o estado de pronta para execução logo após execução do escalonador. Então esta *thread* será escolhida apenas para ser executada da próxima vez que o escalonador for chamado, que pode ser até $1ms$ mais tarde, no nosso sistema. Outra fonte de incerteza é a manipulação de interrupções. Interrupções podem suspender a execução de uma *thread*, e forçar o processador a alternar para a rotina manipuladora. Como é impossível predizer quantas interrupções acontecerão durante o tempo em que a *thread do_period* estiver pronta para transmitir uma mensagem, assumimos como pior caso a situação onde a manipulação de interrupções consome 50% dos ciclos da CPU, em qualquer período de tempo observado. Nós consideramos esse atraso como sendo T_{inter}, que será igual ao tempo processamento útil realizado pela máquina. Nossas medições em sistemas extremamente carregados indicam que esta é uma consideração realista.

Finalmente, o tempo real para transmitir a informação necessária em uma fatia de tempo também deve ser medida. Em nossa implementação todas as fatias de transmissão têm o mesmo tamanho (10 bytes), portanto, é suficiente medir o tempo máximo necessário para enviar 10 bytes através da rede síncrona. Este tempo é dado pela fórmula $T_{com} = T_{transf} + T_{transm}$, onde T_{transf} é o tempo gasto para que os bytes da mensagem sejam transferidos para o buffer do dispositivo e T_{transm} é o tempo gasto para que o dispositivo envie a mensagem pela rede. Em nossa implementação, $T_{transf} = 10ms$ e $T_{transm} = 25ms$, logo, $T_{com} = 35ms$.

Em resumo, o tamanho de uma fatia de tempo do protocolo TDMA é $T_{slot} = T_{sinc} + T_{sched} + T_{inter} + T_{com} + T_{drift}$, onde $T_{sched} = 1ms$, e $T_{inter} = T_{transf}$[8]. O período TDMA é dado por $(n + 2) \times T_{slot}$, que para um sistema de 5 máquinas é igual a $504ms$. (falta atualizar cálculos)

3.1.4 Formatando Heartbeats

Como cada máquina utiliza uma única fatia TDMA para enviar as informações de *heartbeats* de todos os seus monitoráveis em cada período TDMA, e cada *heartbeat* requer um tempo para ser devidamente remetido e recebido, o número máximo de monitoráveis atendidos por uma única máquina é limitado. Desta forma, consideramos que uma fatia de tempo TDMA para *heartbeats* é dividida em um número fixo de sub-fatias. De modo a incrementar a disponibilidade de sub-fatias de tempo livres em cada período TDMA, o serviço escalona monitoráveis para enviarem seus *heartbeats* na menor frequência

[7] O termo $n + 2$ refere-se ao número de fatias de tempo no período TDMA (n fatias de tempo para *heartbeats*, mais duas para sincronização e anúncio), enquanto o termo $n - 1$ refere-se ao pior caso, quando $n - 1$ máquinas falham, e a máquina restante foi a última a unir-se ao domínio de detecção.

[8] O tempo T_{transf} é o único que pode sofrer influência dos atrasos em decorrência do tratamento de interrupções.

possível. Esta frequência deve ser alta o suficiente para ainda garantir que as suas falhas serão detectadas com o limite de latência desejado. Assim, nem todos os monitoráveis têm os seus *heartbeats* transmitidos todo período. Eles são escalonados para serem enviados apenas quando estritamente necessários para prover a latência de detecção necessária. Por exemplo, se um processo é adicionado para ser monitorado com um tempo máximo de detecção de falha de $1000ms$ e cada período TDMA dura $500ms$, este processo pode ter seus *heartbeats* enviados em períodos alternados e ainda assim garantir que sua falha será detectada em no máximo $1000ms$.

Antes de requisitar a inscrição de um novo processo monitorável, o módulo local precisa decidir que frequência mínima de transmissão deve ser usada. Ele inicia tentando utilizar a frequência mais baixa possível, obtida pela divisão do tempo total de um período TDMA pelo tempo de detecção requisitado. Caso não seja possível encontrar uma seqüência de sub-fatias livres para o novo processo monitorável com a frequência calculada, ele incrementa o valor da freqüência e analisa novamente[9]. Se a frequência atingir o valor máximo (1 *heartbeat* a cada período TDMA) sem uma alternativa válida, o processo de admissão termina e um código de erro é retornado para o processo que fez a invocação. Observe que a duração do período TDMA define um limite mínimo para a latência de detecção (*i.e.* um limite máximo para a frequência de transmissão) que pode ser requerida por uma entidade monitorável.

3.2 Descrição do Hardware

3.2.1 Componentes

O dispositivo de comunicação utiliza uma das faixas de rádio frequência UHF não licensiadas. Estas faixas são livres para aplicações industriais, médicas e científicas. Os principais componentes deste dispositivo são: i) um transceptor de rádio UHF que permite a comunicação com outros dispositivos de comunicação dentro de um domínio de detecção local; ii) microprocessador, memória e temporizadores (embarcados em um único microcontrolador de baixo custo), que implementam um temporizador de cão-de-guarda e um protocolo de comunicação básico; e iii) um dispositivo de interface com a máquina associada. O diagrama básico do circuito pode ser visto na Figura 3.

Em nosso protótipo, nós utilizamos como transceptor de rádiofrequência um módulo Radiometrix BiM2-433-160 [2], destinado à faixa UHF de 433.92 MHz. Este dispositivo permite um alcance de 50 metros (interno), o que define o diâmetro máximo para um domínio de detecção local. Sua taxa máxima de transferência alcança os 160 Kbps e utiliza um esquema de modulação FSK (*Frequency Shifting Keying*). Por ser *half-duplex*, o transceptor precisa ser chaveado entre as funções de recebimento e transmissão,

[9]Em cada fatia de *heartbeat* um módulo local envia *heartbeats* correspondentes a um subconjunto das suas entidades monitoráveis locais. O envio de *heartbeats* segue uma seqüência que se repete de tempos em tempos, definindo um novo período que engloba vários periodos TDMA. O escalonamento de uma determinada freqüência de *heartbeats* não será possível quando não existir uma seqüência apropriada de fatias do período TDMA com sub-fatias livres. Embora contra intuitivo, o aumento da freqüência de emissão de *heartbeats* para um determinado monitorável pode viabilizar o escalonamento. Considere por exemplo, a seguinte situação: a seqüência de *heartbeats* de um módulo está se repetindo a cada 6 períodos do TDMA, e dentro de cada uma dessas seqüências, as fatias 1, 3 e 5 não têm sub-fatias livres; neste caso não é possível escalonar uma freqüência de 1 *heartbeat* a cada 3 períodos TDMA, mas é possível escalonar uma freqüência de 1 *heartbeat* a cada 2 períodos TDMA.

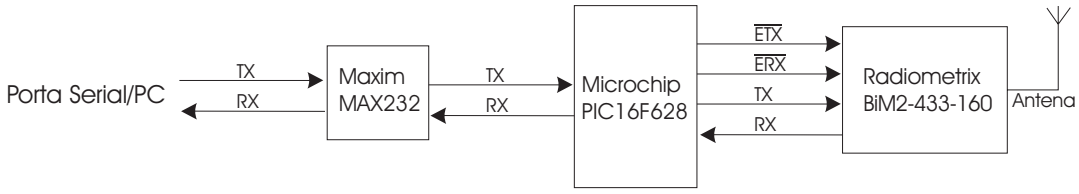

Figura 3: Diagrama do dispositivo de comunicação

isso é realizado através dos sinais ETX e ERX (*i.e. Enable TX* e *Enable RX*, respectivamente), controlados pelo microcontrolador.

O microcontrolador utilizado foi o Microchip PIC16F628 [1], com $2K$ palavras para memória de programa, 228 bytes de RAM, e $200ns$ de tempo mínimo de instrução. Este microcontrolador é equipado com uma USART interna para comunicação serial. Na ausência de um sinal de rádio válido, o transceptor apresenta uma saída de bits aleatórios, é então função do microcontrolador identificar quando esses bits são parte de um pacote sendo transmitido por um outro módulo da rede. Para esta função, cada microcontrolador acrescenta um cabeçalho aos pacotes de 10 bytes que vai transmitir[10]. O microcontrolador também é responsável por controlar o temporizador de cão-de-guarda, que é programado em cada solicitação de envio de mensagem.

A interface serial padrão para conexão com PCs, a RS232C, requer uma lógica negativa, neste caso, o nível lógico alto é representado por tensões entre -3 e -12 volts e o nível lógico baixo por tensões entre +3 e +12 volts. O microcontrolador e o transceptor de rádio trabalham com lógica positiva, com nível lógico alto representado por 5 volts e nível lógico baixo por 0 volts. Desta forma, um conversor é necessário na interface entre a máquina e o módulo. No nosso protótipo foi utilizado o conversor Maxim MAX232.

3.2.2 Temporizador de cão-de-guarda

Como discutido anteriormente, há um probabilidade muito baixa de ocorrer uma falha por temporização nos módulos locais (por exemplo, quando a taxa de manipulação de interrupções exceder 50% ou a sincronização for atrasada em mais de $1ms$ depois do recebimento da mensagem de sincronização). Entretanto, este tipo de falha pode ser tratado. Para tal, basta permitir que o dispositivo de comunicação possa parar ou reiniciar a sua máquina associada na ocorrência de uma falha de temporização. Essa abordagem é também útil para tolerar travamentos do sistema operacional. O mecanismo necessário para lidar com essa falhas é o seguinte. Cada máquina periodicamente reinicia um temporizador de cão-de-guarda no dispositivo de comunicação. Isto é implementado pela *thread do_period*, sempre que uma mensagem síncrona precisa ser enviada. Nessas situações, o temporizador de cão-de-guarda é ajustado para disparar tão logo a próxima mensagem deveria ser enviada. Se tal mensagem não for enviada no tempo apropriado, o temporizador de cão-de-guarda dispara e uma interrupção é enviada à máquina. A rotina de manipulação da interrupção correspondente, armazenada em uma área de memória protegida [8], pára a máquina.

[10] Além do cabeçalho é transmitido um byte de *checksum*, e todo o pacote é retransmitido para assegurar confiabilidade ao canal.

3.2.3 Interface com o dispositivo

De forma a integrar as porções de *hardware* e *software* do módulo de detecção, um *driver* de dispositivo é necessário. Este *driver* converte as chamadas às funções genéricas em instruções específicas do dispositivo. A interface desenvolvida segue o modelo *BSD Sockets*, desta forma mensagens podem ser recebidas e enviadas no canal síncrono utilizando-se as operações comuns de soquetes. Isto é realizado criando uma estrutura de operações de soquete cujos campos são comuns a todos os tipos de famílias de soquetes e apontam para as rotinas específicas que implementam as ações. Foram implementadas quatro operações básicas.

static int syncrf_create(struct socket *sock, int protocol) Chamada pela função *sock_create*, esta função aloca o soquete e é utilizada para a iniciação do dispositivo, reservando a porta serial, configurando os parâmetros de comunicação e associando a nossa rotina de tratamento de interrupção com a linha de interrupção da porta serial.

static int syncrf_release(struct socket *sock) Chamada pela função *sock_release*, esta função libera o soquete e a linha de interrupção.

static int syncrf_sendmsg(struct socket *sock, struct msghdr *msg, int len, struct scm_cookie *scm) Chamada pela função *sock_sendmsg*, esta função recebe como parâmetros os mesmos parâmetros de todos os tipos de soquetes, mas considera apenas os dados inclusos na estrutura *msghdr*. Estes dados correspondem ao byte de ajuste do temporizador de cão-de-guarda mais os bytes da mensagem (até 10 bytes).

static int syncrf_recvmsg(struct socket *sock, struct msghdr *msg, int len, int flags, struct scm_cookie *scm) Chamada pela função *sock_recvmsg*, bloqueia o processo que fez a invocação em um fila de espera até que o dispositivo entregue uma mensagem, o que é feito tão logo um pacote seja completamente recebido. Os dados são inseridos na estrutura padrão de mensagens em soquetes, entretanto, todos os campos são inválidos, exceto o dado em si, incluso na estrutura *msghdr*.

Além da interface de soquete, foi implementado o tratador de interrupção que realmente cuida das transferência dos dados e do reconhecimentos dos sinais especiais enviados pelo dispositivos (*i.e.* novo pacote e estouro do temporizador de cão-de-guarda).

4 Conclusões

Em [7] é provado que a classe $\diamond S$ é a mais fraca classe de detectores de falhas que permite uma solução para o problema do consenso. Seguindo este resultado, algumas implementações de detectores de falhas $\diamond S$ foram tratadas na literatura. Nós acreditamos que o resultado apresentado em [7] induziu excessivamente a implementação de detectores de falhas $\diamond S$. Uma vez que estes são os detectores de falhas mais fracos que permitem a solução do consenso, é muito provável que eles sejam os mais baratos para serem implementados, daí o crescente interesse por esse tipo de detector. Entretanto, nossa experiência mostrou que um detector de falhas perfeito pode ser construído por um custo bem razoável (nós gastamos menos que $\$25$ na construção do nosso primeiro protótipo [4]), quando comparado com os detectores $\diamond S$ propostos na literatura.

Acreditamos que a utilização de serviços de detecção de falhas com semânticas

mais fortes pode não apenas reduzir as considerações necessárias para garantir a corretude de protocolos de mais alto nível, mais também resultar em ganhos de performance substanciais para estes, caso os projetistas de tais protocolos possam desenvolver protocolos de alto nível mais eficientes baseados em abstrações mais fortes. No momento nossos esforços de pesquisa têm se concentrado na melhoria do desempenho do serviço de detecção de falhas e no desenvolvimento de protocolos distribuídos de consenso mais eficientes.

Referências

[1] PIC16F628 data sheet. Microchip, 1999. http://www.microchip.com.

[2] 433 MHz high speed FM radio trannsceiver module. Radiometrix, 2002. http://www.radiometrix.co.uk/products/bin2.htm.

[3] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.

[4] F.V. Brasileiro, A.E.M. Brito, H.D.M. Braz, and W.R. Vasconcelos Neto. A failure detector device for local area networks. In *Supplemental Volume of the International Conference on Dependable Systems and Networks*, pages B44–B45, Washington, D.C., USA, Jun 2002.

[5] 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.

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

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, Jul 1996.

[8] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: surviving operating system crashes. *ACM SIGPLAN Notices*, 31(9):74–83, 1996.

[9] 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.

[10] P. Felber, R. Guerraoui, X. Défago, and P. Oser. Failure detector as first class objects. In *International Symposium on Distributed Objects and Applications*, L'Aquila, Italy, Sep 1999.

[11] 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.

[12] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the* 19^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, 2000.

[13] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8), August 2002. A preliminary version of this document appeared as Technical Report DI/FCUL TR 99-2, Department of Computer Science, University of Lisboa, Apr 1999.