Uma Nova Arquitetura para a Implementação de um Serviço de Detecção de Falhas na Internet

Rogério C. Turchetti[1,2], **Elias P. Duarte Jr.**[2]

[1]CTISM - Universidade Federal de Santa Maria (UFSM)
Avenida Roraima, 1000 – 97105-900 – Cidade Universitária
– Santa Maria – RS – Brasil

[2]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-980 – Curitiba – PR – Brasil

turchetti@redes.ufsm.br, elias@inf.ufpr.br

Abstract. *This work describes a novel SNMP-based architecture for deploying a failure detection service in the Internet. The architecture is based on the* $fdMIB$ *through which the state of processes and hosts is monitored using heartbeat messages. The* $fdMIB$ *encompasses all the taskes required for monitoring a given LAN, without requiring any additional components. Monitors at different LANs communicate across the Internet using Web Services. A prototype was implemented and evaluated considering the quality of service in terms of the failure detection time, communication cost and CPU usage.*

Resumo. *Este trabalho propõe uma arquitetura para monitoramento de falhas em sistemas distribuídos. A arquitetura é composta por uma MIB (*$fdMIB$*) através da qual são monitorados os estados de hosts e processos utilizando mensagens heartbeat. As atividades de monitoramento na rede local são realizadas inteiramente no âmbito da* $fdMIB$*, ou seja, independe da existência de componentes extras presentes no ambiente. A execução do monitoramento em redes locais distintas via Internet, ocorre com o auxílio de Web Services que assumem o papel de Gateway SNMP. Um protótipo foi implementado e avaliado considerando a qualidade do serviço como o tempo de detecção de falhas, custo de comunicação e consumo de CPU.*

1. Introdução

Frequentemente a coordenação de eventos em um sistema distribuído exige alguma forma de acordo, mas a realização de tal tarefa não é possível em sistemas assíncronos sujeitos a falhas. Isso se deve à impossibilidade em determinar quando um processo está falho ou simplesmente lento [Fischer et al. 1985]. Chandra e Toueg [Chandra and Toueg 1996] propuseram os detectores de falhas como uma abstração que, quando atendem certas propriedades, possibilitam que o consenso possa ser atingido, mesmo em sistemas assíncronos sujeitos a falhas *crash*.

A partir da proposta original iniciou-se uma pesquisa no aperfeiçoamento dos serviços oferecidos pelos detectores de falhas. Sendo assim, acredita-se que baseado nesta literatura, o conceito de detectores de falhas já está suficientemente amadurecido, permitindo propor uma arquitetura para o serviço de monitoramento dos estados funcionais de *hosts* e processos de sistemas reais como, por exemplo, a própria Internet.

Este trabalho objetiva aplicar os conceitos de detecção de falhas em uma arquitetura prática, através da implementação de um serviço completamente independente da aplicação que será executada. O trabalho é uma extensão do serviço proposto em [Moraes and Duarte Jr. 2011]. A principal diferença da nova arquitetura para a do trabalho citado, é que elimina-se a necessidade de incluir uma *thread* de envio de mensagens de vida (*heartbeats*) para que um processo de aplicação possa ser monitorado. Em outras palavras, a nova arquitetura livra o programador de aplicação desta tarefa. Outra proposta é diferenciar o tratamento entre *hosts* e seus processos. A razão deste tratamento diferenciado é a redução do uso do canal de comunicação. Além disso, as funções de monitoramento para detecção de falhas na arquitetura proposta não dependem da existência de componentes extras no ambiente.

O objetivo não é oferecer um serviço para determinada aplicação, e sim, uma arquitetura genérica que possa ser utilizada para monitorar qualquer aplicação interessada. Esta arquitetura define uma MIB (*Management Information Base*) que pode ser consultada via aplicação SNMP (*Simple Network Management Protocol*). A MIB é denominada de $fdMIB$ (*failure detector MIB*) e disponibiliza informações de monitoramento dos estados de *hosts* e processos. Destaca-se que para utilizar a $fdMIB$ basta o dispositivo oferecer acesso via agentes SNMP. Portanto, a arquitetura pode ser implementada nos diversos tipos de dispositivos, por exemplo, equipamentos de redes de computadores.

Em [Wiesmann et al. 2006] foi proposto um *framework* denominado de SNMP-FD, o qual disponibiliza um serviço para detecção de falhas através de uma MIB. Tanto a aplicação que deseja monitorar processos no ambiente como as aplicações que serão monitoradas necessitam ser registradas em um *daemon*. Este *daemon* é responsável por trocar informações de monitoramento somente na rede local. Na arquitetura da $fdMIB$ há a possibilidade de realizar monitoramento através de múltiplos sistemas autônomos, utilizando um *gateway* SNMP implementado com *Web Services*. As mensagens vindas pela Internet são traduzidas pelo *gateway* em chamadas na rede local, via agentes SNMP, possibilitando buscar informações de monitoramento no *host* monitor. Além disso, o monitoramento na rede local é realizado pela própria $fdMIB$, sem a necessidade de *daemons* adicionais.

Um protótipo da arquitetura proposta foi implementado e resultados experimentais mostram as funcionalidades do serviço de detecção de falhas. A flexibilidade adquirida pela arquitetura, utilizando um *timeout* adaptativo, é perceptível com a variação de atrasos na comunicação. É apresentada também a facilidade de gerenciar *hosts* em diferentes sistemas autônomos através de *gateway* SNMP via Web Services. Portanto, é avaliado o custo de comunicação adicional imposto pela tradução de protocolos.

O trabalho está organizado da seguinte forma. A seção 2 apresenta os detectores de falhas. Para contextualizar o ambiente e a metodologia para a detecção de falhas, a seção 3 detalha o modelo do sistema. A seção 4 apresenta a estrutura geral da $fdMIB$ e sua arquitetura. Os experimentos e as conclusões do trabalho são apresentados na seção 5 e 6, respectivamente.

2. Detectores de Falhas

Em sistemas distribuídos tolerantes a falhas, são frequentes as situações em que os processos precisam entrar em um acordo referente a uma decisão a ser tomada. Como exemplo,

pode-se citar transações atômicas como *commit* em banco de dados, *broadcast* atômico, *membership* de grupo, eleição de líder, entre outras [Charron-Bost et al. 2010]. A realização do acordo distribuído não é uma solução trivial, podendo-se afirmar que é um dos principais problemas em sistemas tolerantes a falhas [Borran et al. 2012].

Em um sistema distribuído onde há necessidade de se ter uma decisão em conjunto entre os processos participantes do sistema, um mecanismo fundamental para garantir a decisão unânime dos participantes é a inclusão de algoritmos de consenso, onde todos os processos devem entrar em comum acordo sobre um determinado valor [Borran et al. 2012]. Em síntese, o algoritmo de consenso deve oferecer um serviço que garanta a mesma decisão entre todos os participantes. Em sistemas síncronos ou isentos de falhas, as propriedades do algoritmo podem ser garantidas e o consenso pode ser resolvido trivialmente.

Entretanto, o principal impasse para resolvê-lo ocorre em ambientes assíncronos sujeitos a falhas, onde não há algoritmos determinísticos que possam solucionar o consenso [Fischer et al. 1985]. Uma alternativa é utilizar um modelo parcialmente síncrono formado por um assíncrono incrementado com detectores de falhas, proposto por [Chandra and Toueg 1996]. Para o protocolo de consenso o detector de falhas é um oráculo que permite encapsular o indeterminismo, garantindo as propriedades do algoritmo.

O monitoramento de processos em sistemas distribuídos utilizado pelos detectores de falhas é realizado através da troca de mensagens, respeitando um limite de tempo para tomar qualquer decisão. Dois algoritmos tradicionais denominados de *Pull* e *Push* [Felber et al. 1999] descrevem exatamente como a tarefa de monitoramento é realizada.

Em síntese, no algoritmo *Pull* as mensagens de controle seguem no sentido oposto ao fluxo de controle. Os processos monitorados periodicamente são questionados pelo detector de falhas com uma mensagem de requisição de vida (`AreYouAlive?`). Se um processo monitorado responder às requisições feitas pelo detector, dentro de um determinado tempo *(timeout)*, significa que ele está operacional. No algoritmo de detecção *Push* as mensagens de controle (`IAmAlive`) geradas pelos detectores seguem o mesmo sentido do fluxo das informações. Os processos monitorados enviam periodicamente mensagens de vida (*heartbeat*), indicando que eles ainda estão operacionais.

Decidir qual algoritmo utilizar requer saber os aspectos positivos e negativos de cada opção. Considerando o detector *Pull* os processos monitorados não necessitam ser ativos (somente respondem aos estímulos, nenhuma configuração prévia é necessária), o que proporciona ao detector maior flexibilidade no monitoramento dos processos. Por outro lado, seu estilo de monitoramento requer maior número de mensagens. O modelo *Push* necessita de parâmetros configurados tanto no monitor quanto no processo a ser monitorado, o que dificulta sua configuração. Entretanto, é mais eficiente em termos de número de mensagens, pois em um sistema composto por n processos, em um ciclo de detecção, deverão ser geradas $n(n-1)$ mensagens, enquanto que o detector *Pull* irá gerar $2n(n-1)$ mensagens em cada ciclo.

A $fd{-}MIB$ proposta neste trabalho segue o modelo *Push* para monitoramento de processos. Para resolver o problema gerado pelo detector *Push*, onde os parâmetros (em particular o intervalo de *timeout*) devem ser ajustados no monitor e no processo monito-

rado, o esquema de detecção utiliza um *timeout* (Δ_{to}) adaptativo, evitando a necessidade de configurar este parâmetro para cada processo . Em outras palavras, somente há a necessidade de configurar, nos processos monitorados, a periodicidade de envio de mensagens (Δ_i).

Além destes modelos tradicionais apresentados, outros detectores estão presentes na literatura, tal como o modelo Gossip [Renesse et al. 1998], o modelo Dual [Felber et al. 1999] e o próprio detector *heartbeat* proposto por [Aguilera et al. 1997], o qual possui características diferenciadas do modelo *Push* pois não utiliza *timeouts* somente incrementa um contator a cada mensagem de vida.

3. Modelo do Sistema

Neste trabalho, considera-se um sistema distribuído S composto por um conjunto finito de n *hosts* (H), e até m processos (p), onde $S = \{H_1, H_2, ..., H_n, \} = \{\{p_{11}, p_{12}, ..., p_{1n}, \}, \{p_{21}, p_{22}, ..., p_{2n}, \}, ..., \{p_{n1}, p_{n2}, ..., p_{nm}, \}\}$. Portanto, o sistema é um conjunto de conjuntos - os *hosts*, cada um dos quais representados pelos processos que executa. O processo p_{ij} é o j-ésimo processo em execução no *host* H_i. A comunicação é realizada por troca de mensagens através de um canal confiável, isto é, o canal não cria, não duplica, não altera e nem perde mensagens de controle.

Cada H_i pode ter seu estado assinalado como suspeito pelo detector, se um *heartbeat* for recebido depois de um instante t, onde t é o *timeout*. Neste instante todos os p_{im} processos contidos em H_i são considerados suspeitos. H_i é considerado correto se a mensagem de *heartbeat* chegar em t', onde $t' < t$. O monitoramento de um processo p_{im} ocorre localmente pelo *host* hospedeiro H_i. Neste caso, a seguinte situação pode ocorrer: um processo suspeito p_{im} pode estar em um *host* H_i sem-falha. Tal situação ocorre quando p_{im} é questionado localmente por H_i e não obtém resposta. A mensagem de *heartbeat* enviada por H_i chega antes de t unidades de tempo informando que p_{im} não está mais operacional.

Assume-se o modelo assíncrono incrementado com detectores de falhas não confiáveis. Cada rede local tem acesso a somente um detector de falhas, o qual nunca falha. O compartilhamento dos estados dos *hosts* e processos pode ocorrer através de redes independentes pela Internet, mas nenhuma hipótese é feita referente à consistência entre as saídas produzidas pelos detectores. A interação entre os detectores em redes locais distintas é prevista no modelo. Os processos falham por colapso (*crash*), ou seja, deixam de executar suas tarefas prematura e completamente.

4. A $fdMIB$

Nesta seção a $fdMIB$ proposta para implementar o detector de falhas é descrita. A seção 4.1 detalha a estrutura geral da MIB. A seção 4.2 descreve como a $fdMIB$ é usada para o monitoramento e detecção de falhas. A seção 4.3 mostra a arquitetura do serviço de detecção de falhas baseado na $fdMIB$, detalhando em particular seu uso em redes locais e na Internet.

4.1. Estrutura da $fdMIB$

A $fdMIB$ proposta neste trabalho está dividida em três seções, onde cada parte corresponde às funcionalidades disponíveis para cada dispositivo, os quais desempenham

papel específico no ambiente, sendo eles: **Seção Monitor**, **Seção Monitorado** e **Seção Notificar**. Vale ressaltar que as funcionalidades estão disponíveis dentro da mesma MIB $fdMIB$, entretanto, cada dispositivo irá fornecer e obter informações na porção da MIB de seu interesse. A estratégia utilizada para a implementação é executar todas as funções para gerenciamento e detecção de falhas através de um serviço disponível via agente SNMP. Isto, evita a necessidade da utilização de *daemons* extras como previsto em outras soluções [Wiesmann et al. 2006] [Moraes and Duarte Jr. 2011].

Na **seção Monitor** denominada na $fdMIB$ de `monitorNodeGroup`, há informações dos processos, tais como: endereço IP, identificação do processo través do número da porta e do *pid* (*process identifier*). Outras informações incluem o estado do processo, o objeto que recebe as mensagens de *heartbeat*, entre outros dados estatísticos. Os detalhes de funcionamento são descritos na seção 4.2.

Dentro da **seção Monitorado** são armazenadas informações dos estados de cada processo monitorado. Na $fdMIB$ esta seção é denominada de `monitoredNodeGroup` e contém informações como: periodicidade das mensagens de *heartbeat*, IP do processo monitor para encaminhamento das mensagens, o OID (*Object IDentifier*) correspondente na tabela do monitor na seção `monitorNodeGroup` (maiores detalhes na seção 4.3) e informações dos processos/serviços registrados para monitoramento.

Finalmente na **seção Notificar**, as aplicações que desejam receber informações sobre os estados dos processos, se registram na $fdMIB$. O registro é realizado no *host* monitor, identificando o seu endereço IP. Quando ocorre o registro de alguma aplicação na seção Notificar, a notificação ocorre por um monitoramento passivo, isto é, as aplicações são notificadas sobre mudanças de estado, deixando-as livres para realizar consultas periódicas. Opcionalmente, a aplicação pode consultar, quando desejar, os estados dos processos monitorados, fazendo uso do módulo SNMP. Para este último procedimento, não é necessário o registro na seção Notificar, uma vez que a notificação ocorre por monitoramento ativo.

4.2. Detecção de Falhas com a $fdMIB$

A $fdMIB$ trabalha com as mensagens de controle para detecção de falhas seguindo o mesmo sentido do fluxo de informações, ou seja, o *host* monitorado envia mensagens de *heartbeat* para o *host* monitor. Estas informações são computadas na $fdMIB$ tendo em vista a identificação do processo monitorado.

A figura 1 apresenta a troca de informações entre um processo monitor *q* e um processo monitorado *p* utilizando a $fdMIB$. A cada mensagem recebida por *q*, ele reinicia o *timeout* correspondente ao processo emissor. Dessa forma, existem certas restrições a serem efetuadas na definição dos parâmetros Δ_i (periodicidade de envio de mensagens) e Δ_{to} (*timeout*). Considerando a figura 1, pode-se observar que Δ_{to} deve ser maior que Δ_i [Sergent et al. 2001], caso contrário as mensagens *heartbeat* não chegarão ao seu destino em tempo hábil.

A limitação entre os parâmetros Δ_{to} e Δ_i é facilmente perceptível. Considerando aspectos de implementação, se tais parâmetros forem fixos é provável que a qualidade de serviço do detector de falhas sofra com influências de atrasos de processamento ou sobrecarga nos canais de comunicação. No contexto da Internet, é impraticável fixar qualquer

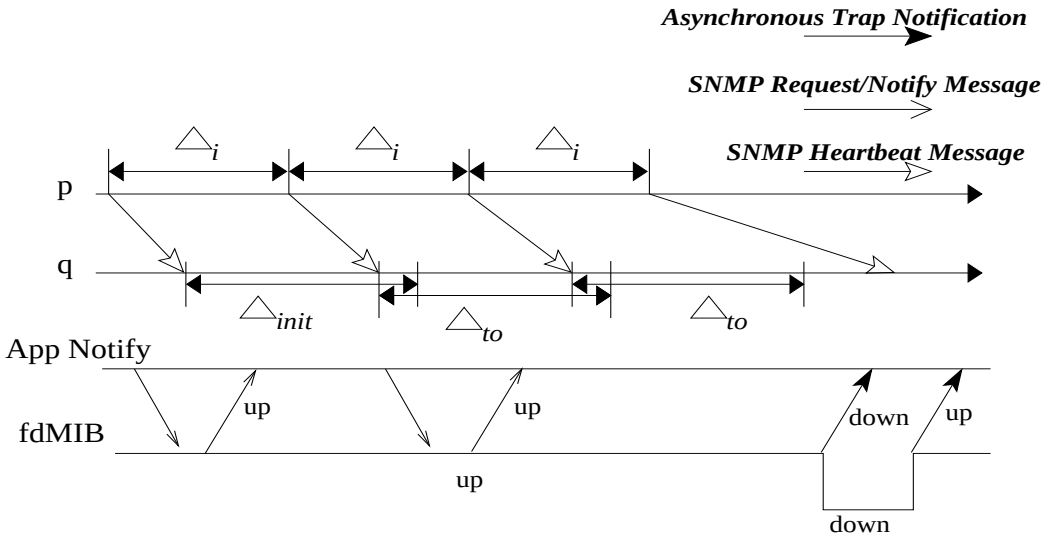


Figura 1. Exemplo de detecção e notificação de falhas utilizando a $fdMIB$.

limite de tempo, pois neste cenário as variações são mais constantes. Para minimizar este problema em detecção de falhas, diversos trabalhos foram propostos baseadas em estimativas que visam corrigir o valor do *timeout* de acordo com o comportamento do ambiente ou a necessidade da aplicação [Bertier et al. 2003] [Nunes and Jansch-Pôrto 2004] [dos Santos Sá and de Araújo Macêdo 2005].

Para o trabalho em questão, a $fdMIB$ realiza a mesma função para o cálculo do *timeout* que o protocolo TCP proposto em [Jacobson 1988]. O cálculo adaptativo, no caso de mensagens de *heartbeat*, deve levar em consideração não somente a variação do Δ_{to}, mas também a variação no tempo de chegada das mensagens, ou seja, o parâmetro Δ_i. Por esta razão, justifica-se o uso da expressão apresentada em (1) e descrita a seguir, pois considera ambos parâmetros para adaptar o limite de tempo.

$$\begin{aligned} Diferenca &= Tempo\_hb_i - Tempo\_hb_{i-1} \\ Media_i &= \alpha * Media_{i-1} + (1-\alpha) * Diferenca \\ Desvio_i &= \alpha * Desvio_{i-1} + (1-\alpha) * |Media - Diferenca| \\ \Delta_{to} &= Media_i + \beta * Desvio_i \end{aligned} \tag{1}$$

De acordo com a expressão apresentada, é possível observar que o cálculo do Δ_{to} está relacionado à periodicidade das mensagens *heartbeat* enviadas ($Diferenca$), bem como as variações de tempo durante o tráfego das mensagens entre o *host* monitor e os processos monitorados ($Media$ e $Desvio$). Com este cálculo tenta-se respeitar os limites entre os parâmetros de periodicidade e de *timeout*. Os valores utilizados para as constantes α e β, são próximos dos propostos por Jacobson, sendo 0.9 para α e 4 β. O cálculo é refeito para cada mensagem recebida considerando a fonte emissora.

Para o esquema de monitoramento de vários processos hospedados no mesmo *host*, uma estratégia é proposta objetivando a flexibilidade e redução no número de mensagens de vida. Considere, por exemplo, um *host* que hospeda 3 processos a serem monitorados, se cada processo tiver implementando uma *thread* para encaminhamento das

mensagens de *heartbeat*, a cada Δ_i intervalo de tempo, haverão 3 mensagens de monitoramento encaminhadas ao mesmo *host* monitor, pelos respectivos processos. Além disso, se for assumido que estas frequências são iguais ou muito próximas, é completamente desnecessário o encaminhamento de múltiplas mensagens. A ideia então é, entre todas as frequências definidas pelas aplicações monitoradas, utilizar o menor valor. A escolha da menor frequência justifica-se pelo fato de que, teoricamente, é a aplicação que necessita de um menor tempo para detecção de falhas.

É importante ressaltar que além da redução de mensagens de monitoramento, a estratégia apresentada livra a aplicação da implementação de uma *thread* para envio de mensagens a cada Δ_i. Neste caso, o monitoramento para as aplicações hospedadas em um *host* é realizado localmente, sendo o procedimento da seguinte forma: instantes antes da expiração do intervalo Δ_i, a $fdMIB$ monitora localmente os estados dos processos. Após esta verificação local, em Δ_i é encaminhada uma única mensagem indicando os estados de todos os processos para o monitor.

Notificação de Processos Suspeitos

A figura 1 apresenta os dois formatos possíveis para que uma aplicação fique sabendo os estados dos processos e *hosts*. Na primeira possibilidade consistem em fazer com que a própria aplicação consulta os estados de todos os processos através de uma requisição `snmptable`. O retorno efetuado pela $fdMIB$ é uma tabela indicando informações sobre cada processo e seu *host* hospedeiro. A segunda possibilidade é implementada na $fdMIB$ o tipo `NOTIFICATION-TYPE` que encaminha uma mensagem `Trap` informando, para a aplicação cadastrada na $fdMIB$, que um evento de troca de estado ocorreu. A troca de estado ocorre quando um *host* ou processo passa de correto para suspeito ou vice-versa.

Existem dois formatos possíveis para que uma aplicação fique sabendo os estados dos processos e *hosts*. Na primeira possibilidade consiste em fazer com que a própria aplicação consulta os estados de todos os processos através de uma requisição `snmptable`. O retorno efetuado pela $fdMIB$ é uma tabela indicando informações sobre cada processo e seu *host* hospedeiro. A segunda possibilidade é implementada na $fdMIB$ o tipo `NOTIFICATION-TYPE`, que encaminha uma mensagem `Trap` informando que um evento de troca de estado ocorreu.

4.3. Arquitetura do Serviço de Detecção de Falhas

A figura 2 descreve a arquitetura do serviço de detecção em uma rede local. A $fdMIB$ é utilizada pelo *Monitor Host* e *Monitored Host*, sendo exatamente a mesma MIB mas com uso de suas funcionalidades de maneiras distintas. O *Monitor* informa ao *Monitored Host* os processos a serem monitorados e seu endereço para receber as mensagens de *heartbeat*. O *Monitored Host* busca no *Monitor* o OID correspondente ao seu endereço. O OID é a referência do *Monitored Host* dentro da tabela. Tal referência é utilizada pelo *Monitor* para atualizar seu estado a cada mensagem de *heartbeat*.

Na figura 2 o sistema operacional (OS) no *Monitored Host* disponibiliza informações sobre a execução das aplicações (App_1,..., App_n) a serem monitoradas localmente pela $fdMIB$. A *Application* é a parte interessada sobre os estados dos processos e *hosts*,

ou seja, é a aplicação que por algum motivo necessita de informações sobre a detecção de falhas dos processos/hosts.

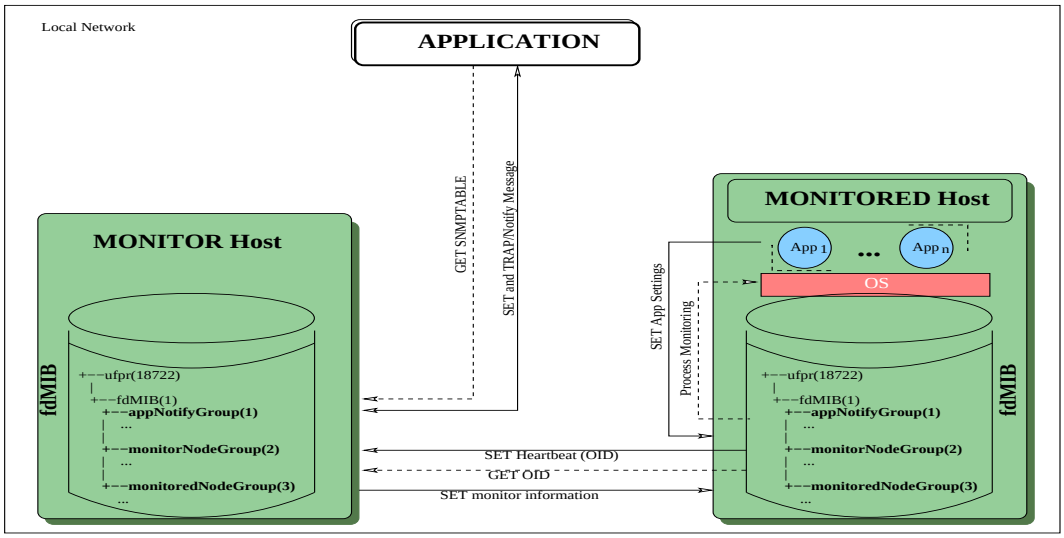


Figura 2. A arquitetura da $fdMIB$ em uma rede local.

Há duas formas distintas para a $fdMIB$ alterar o estado de um *host* ou processo. Considerando o monitoramento de um *host*, a cada mensagem de *heartbeat* recebida, o cálculo do parâmetro Δ_{to} é adaptado às condições de atraso momentâneas. Concomitantemente, uma *thread* é lançada para execução e posteriormente colocada para dormir, por um período igual ao Δ_{to}, de acordo com a expressão (1). Caso o Monitor receba a próxima mensagem de *heartbeat* antes de terminar o Δ_{to} (período em que a *thread* ainda está dormindo), a *thread* vigente é cancelada e o estado do *Monitored Host* é atualizado para 0 (correto). Se nenhuma mensagem for recebida dentro do período Δ_{to}, a *thread* ao acordar, irá alterar o estado do respectivo processo, no objeto correspondente, dentro da $fdMIB$, para 1 (suspeito).

Já para o monitoramento de um processo, assume-se que o *Monitored Host* esteja operacional e que, por algum motivo, o processo de aplicação executando no *host* monitorado parou de responder. A $fdMIB$ localizada no *Monitored Host* percebe o ocorrido consultando localmente o estado do processo no sistema operacional, instantes antes de encaminhar a mensagem de *heartbeat*. Este processo é denominado na figura 2 de *Process Monitoring* e ocorre instantes antes a cada Δ_i. A mensagem *heartbeat* será encaminhada ao *Monitor* indicando que um dos processos parou de responder. O processo *Monitor* gerencia as entradas para cada *host* e processo da seguinte maneira: cada *host* possui seus processos a serem monitorados com suas respectivas portas e *pid* local. O conjunto de porta, endereço IP e *pid*, possibilita, ao *Monitor*, saber qual processo deixou de responder. Supondo que no exemplo da figura 2 a aplicação App_1, que está sendo executada no *Monitored Host* e escutando na porta 80, tenha deixado de responder (somente o processo, não o *host* hospedeiro), o *Monitored Host* ao encaminhar a mensagem de *heartbeat* irá indicar que a App_1 não está mais respondendo. Neste caso, o estado do *host* será 0 (correto) ao passo que a App_1 terá o estado alterado para 1 (suspeito).

Executando o Serviço Sobre Múltiplos Sistemas Autônomos

A utilização de agentes SNMP possibilita facilmente estender as funcionalidades para diversos dispositivos e ambientes. Neste trabalho estas facilidades são utilizadas em conjunto com *Web Services* (WS), os quais possibilitam estender a arquitetura de uma rede local para múltiplos sistemas autônomos. Os WS oferecem um conjunto de funcionalidades através de uma interface padrão, garantindo a interoperabilidade entre arquiteturas diferentes [Dialani et al. 2002]. Um fator importante que motiva o uso dos serviços oferecidos pelos WS na arquitetura proposta, é a possibilidade de utilização de portas que, em geral, são abertas nos sistemas autônomos, o que facilita a comunicação através de *firewalls*

Para a implementação do WS apresentado na arquitetura estendida da figura 3, foram utilizadas as seguintes tecnologias: SOAP (*Simple Object Access Protocol*), WSDL (*Web Services Description Language*) e a liguagem PHP para implementar as chamadas via SNMP com a $fdMIB$. O protocolo SOAP define o formato das mensagens resultando em documentos XML via HTTP. Para definir a interface dos *Web Services* a especificação WSDL é utilizada, permitindo fornecer detalhes do serviço a ser disponibilizado para acesso remoto.

Por exemplo, para a aplicação requisitante acessar os *Web Services*, com a finalidade de buscar informações na $fdMIB$, há uma definição do tipo de mensagem, bem como os parâmetros necessários para contato. O acesso da aplicação requisitante para executar um `get`, por exemplo, deverá informar o OID desejado. Essa chamada será traduzida em uma requisição SNMP cujo retorno resulta nas informações que também estarão definidas na interface WSDL. Em outras palavras, o processo de tradução das mensagens em chamadas SNMP torna o WS um *gateway* de protocolos.

A utilização de WS como *gateways* para gerenciamento via SNMP, não é uma novidade. Em [de Lima et al. 2006] foi realizado um estudo que compara a performace das chamadas SNMP utilizando estratégias que implementam traduções via WS.

O WS pode ter como requisitante uma aplicação cliente, como ilustrado na figura 3, ou outro WS que irá trabalhar como um 'cliente'. As mensagens de `set` permitem que um processo ou *host* possa ser monitorado via WS. Entretanto, cada rede que possui um *Monitor Host* gerencia a sua lista de processos monitorados, não havendo uma consistência entre as diversas listas.

Em outras palavras, a visão do detector de falhas é da rede local, a aplicação requisitante da lista poderá estar em outra rede, como mostrado na figura 3. Além disso, para que as informações de monitoramento possam ser exportadas para outras redes, deverá haver um WS trabalhando como *gateway* SNMP. Por exemplo, na arquitetura estendida somente a rede B possui um *gateway* SNMP a rede A não tem a habilidade de exportar seus dados para outras redes.

5. Avaliação

No âmbito de detectores de falhas, Chen, Toueg e Aguilera [Chen et al. 2000] propuseram um grupo de métricas que permitem avaliar a qualidade de serviço dos algoritmos de detecção. As principais métricas são referentes à **velocidade** com que os serviços con-

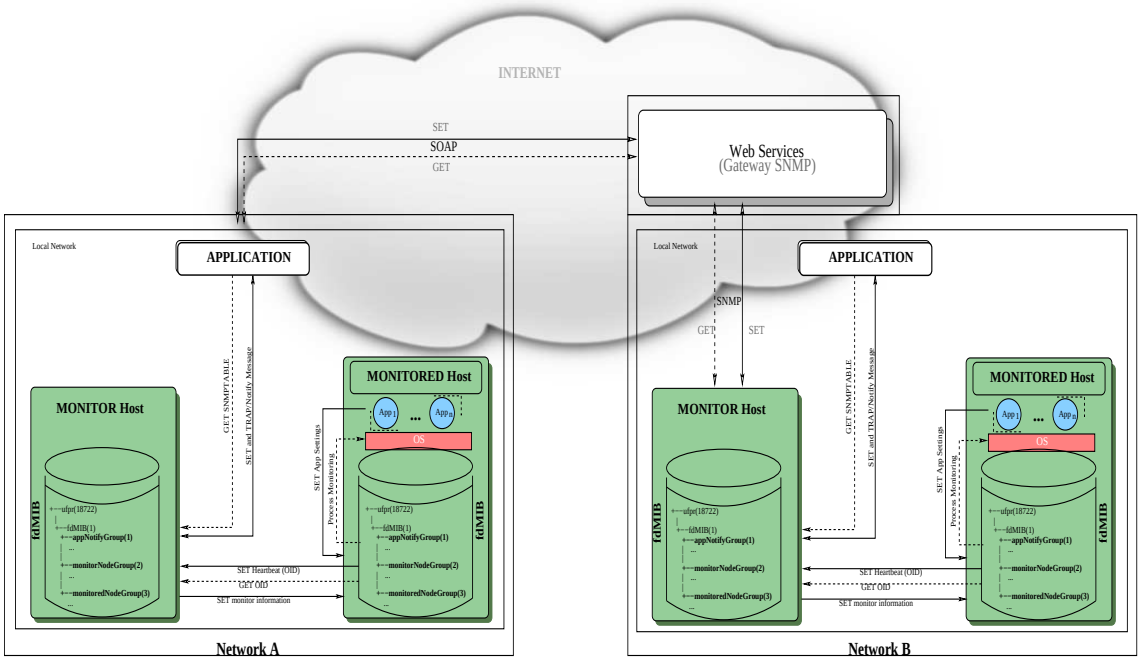


Figura 3. Arquitetura estendida da $fdMIB$ utilizando *Web Services*.

seguem suspeitar de processos falhos e à **exatidão** destas suspeitas. Note que a **velocidade** corresponde aos processos que deixam de operar enquanto que a **exatidão** refere-se aos processos corretos.

Dentre as métricas propostas em [Chen et al. 2000] para avaliar o detector de falhas proposto neste trabalho, são realizadas medidas para o tempo de detecção de uma falha (*detection time* - T_D) variando o parâmetro Δ_i. A eficiência da adaptação do Δ_{to} em diferentes condições de carga, também é apresentada. Por fim, para avaliar o impacto da solução proposta, é medida a utilização da CPU tando no *Monitor* quanto no *Monitored Host* com diferentes cargas de execução.

É importante comentar que, para realizar o cálculo do tempo de detecção (T_D) de forma precisa, há a necessidade de um sincronismo de relógios, de forma a saber o exato instante em que o processo monitorado falhou até o tempo em que o *timeout*, no detector de falhas, atinja seu limite. Realizar este cálculo precisamente não é uma tarefa simples. Considerando um algoritmo que trabalha com mensagens de *heartbeat*, o pior caso, segundo [Chen et al. 2000] para o tempo de detecção, é aquele em que o processo monitorado falha assim que envia a mensagem de vida. Aliado a este cenário, a última mensagem de vida recebida deverá ter levado o máximo atraso possível. Este cenário é apresentado na figura 4, onde o cálculo é representado por td'.

Configurações do Ambiente

Os experimentos foram realizados em duas máquinas físicas distintas com as seguintes configurações: processador Intel Core i5 CPU 2.50GHz com 4 núcleos e sistema operacional Ubuntu 12.04.4 executando a $fdMIB$ como monitor, o *host* moni-

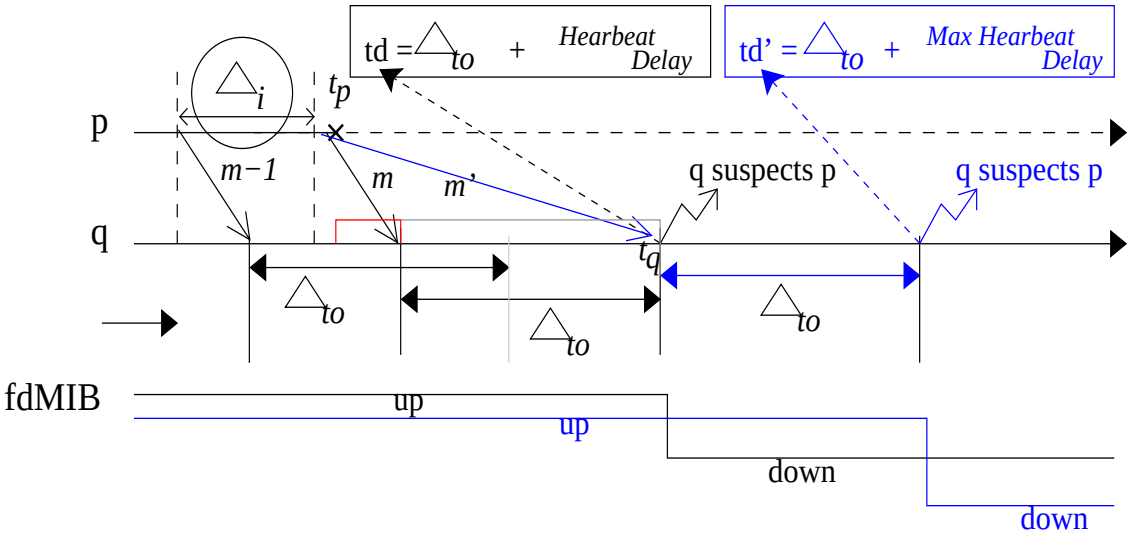


Figura 4. Cálculo para o tempo de detecção.

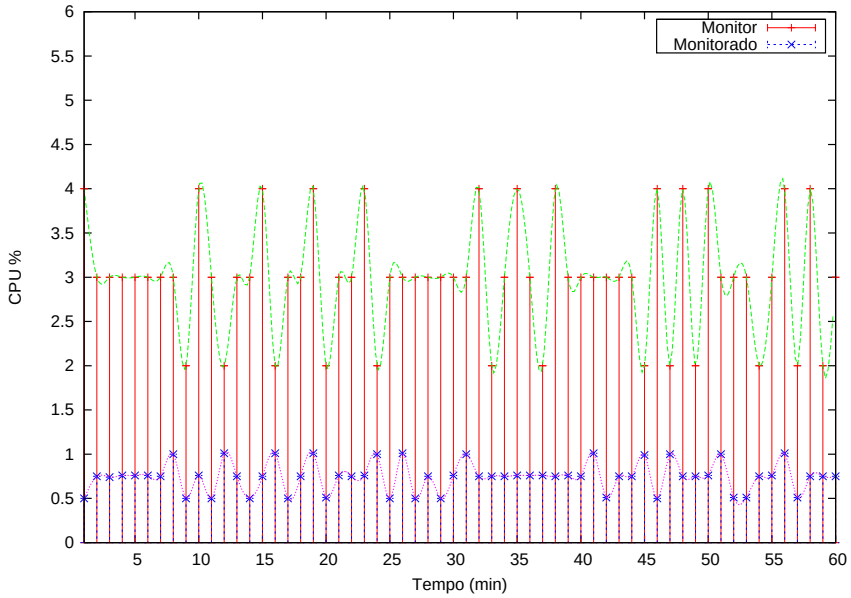
torado possui um processador Intel Core i5 CPU 3.20GHz com 4 núcleos executando o sistema operacional Ubuntu 13.10, com kernel 3.2.0-58 ambas as máquinas. O canal de comunicação na rede local é Ethernet 100Mbit/s. A MIB foi projetada com o auxílio do pacote Net-Snmp versão 5.4.4 [Net-SNMP 2014], para a implementação do WS foi utilizado o servidor Apache versão 2.2.22.

Experimentos

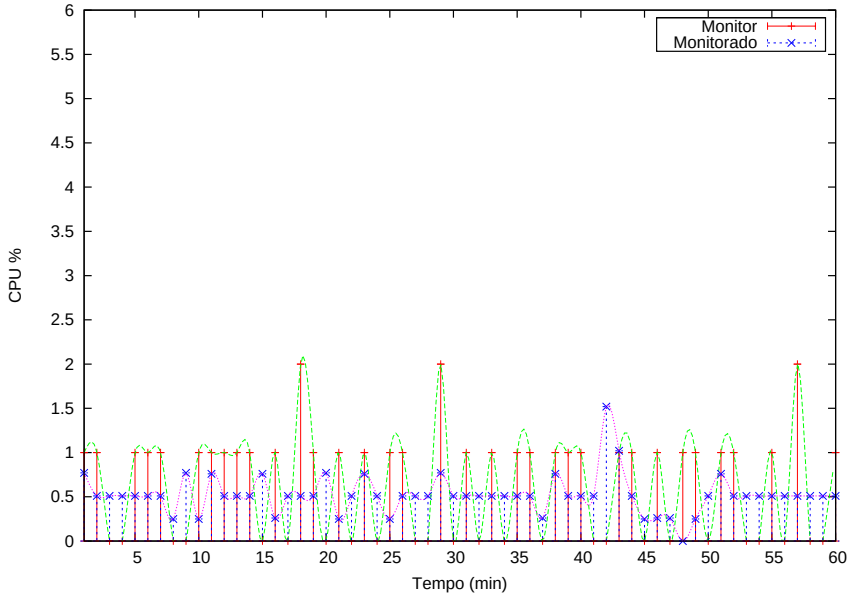
Para medir o uso da CPU pelo processo monitor e processo monitorado, o parâmetro Δ_i é configurado para 1 milisegundo (ms) e, posteriormente, para 10 ms. Estas periodicidades correspondem à maior e menor carga considerada neste trabalho, respectivamente. O gráfico da figura 5(a) apresenta a carga de utilização da CPU durante 1 hora de execução com Δ_i = 1 ms. Neste período, a utilização média de CPU pela $fdMIB$ no processo monitor é de exatos 3%, ao passo que para o processo monitorado a média observada foi de 0,74%.

A utilização de CPU, considerando o processo monitor, é 4 vezes superior em relação ao monitorado. A principal razão se deve ao fato de que há a criação e execução de uma *thread*, a cada mensagem recebida pelo monitor. Conforme o tempo de envio de mensagem é aumentada, a utilização da CPU é reduzida. O gráfico da figura 5(b) apresenta a carga de utilização da CPU durante 1 hora de execução com Δ_i = 10 ms. A redução é mais significativa no processo monitor. A média de utilização da CPU no processo monitor foi de 0,63%, ou seja, há uma redução de 79% em relação à maior carga. Para o processo monitorado a média apresentada foi de 0,52%, uma redução não muito significativa.

Para calcular o tempo de detecção de uma falha, é realizada a simulação de uma falha, onde o processo monitorado deixa de enviar mensagens de *heartbeat*. Na prática, se os *clocks* forem sincronizados, pode-se utilizar a estratégia de enviar a mensagem de *heartbeat* juntamente com a hora local no processo monitorado. Na figura 4, supondo o pior caso (td'), juntamente com m' é enviado o tempo (t_p) em que a mensagem sai do



(a) Uso da CPU, envio mensagens 1 ms.



(b) Uso da CPU, envio mensagens 10 ms.

Figura 5. Uso da CPU com o processo monitor e monitorado executando a $fdMIB$.

host p, chegando ao seu destino no tempo t_q. No nosso cenário, o seguinte cálculo foi realizado:

$$T_D = (Media_i + \beta * Desvio_i) + \overline{(rtt/2)} \quad (2)$$

Desta forma, neste experimento estamos considerando que a média do rtt dividido por 2 é um valor razoável para estimar o momento da falha.

O gráfico da figura 6(a) apresenta o tempo médio de detecção de falhas variando o Δ_i. Foi simulada, para cada Δ_i, 10 falhas no processo monitorado. A diferença entre o *timeout* para o tempo de detecção é justamente o atraso no canal, ou seja, o cálculo do $\overline{rtt/2}$ na expressão (2), que apresentou aproximadamente 0,200 (ms), praticamente sem variações na comunicação, como mostra a variável $Desvio$ a qual sofre influência com mudanças no tempo de comunicação. Para apresentar a adaptação do *timeout* foi realizado o experimento apresentado na figura 6(b) onde o processo emissor envia 100 mensagens, em que as 50 primeiras mensagens utilizam um Δ_i = 1 ms, uma variação brusca foi forçada passando Δ_i para 10 ms, nas próximas 50 mensagens. No instante em que a variação de Δ_i ocorre, 3 falsas suspeitas são notificadas até a $fdMIB$ ajustar o *timeout* para a nova periodicidade de envio.

Para verificar o tempo de comunicação adicionado pelo *gateway* SNMP do WS, dois estilos de `get` foram realizados acessando um único objeto na $fdMIB$. A primeira comunicação foi um `snmpget` utilizando diretamente o protocolo SNMP via porta 161. A segunda comunicação foi por um cliente PHP via SOAP, que acessa o WS via porta 80, a comunicação é redirecionada pelo servidor realizando um `get` por SNMP na $fdMIB$. Este último estilo de comunicação também pode ser realizado entre dois WS.

Tabela 1. Comparação entre WS e comunicação direta SNMP.

Protocolos	Tempo médio em ms
SOAP/HTTP/SNMP	20.12
SNMP	11.7

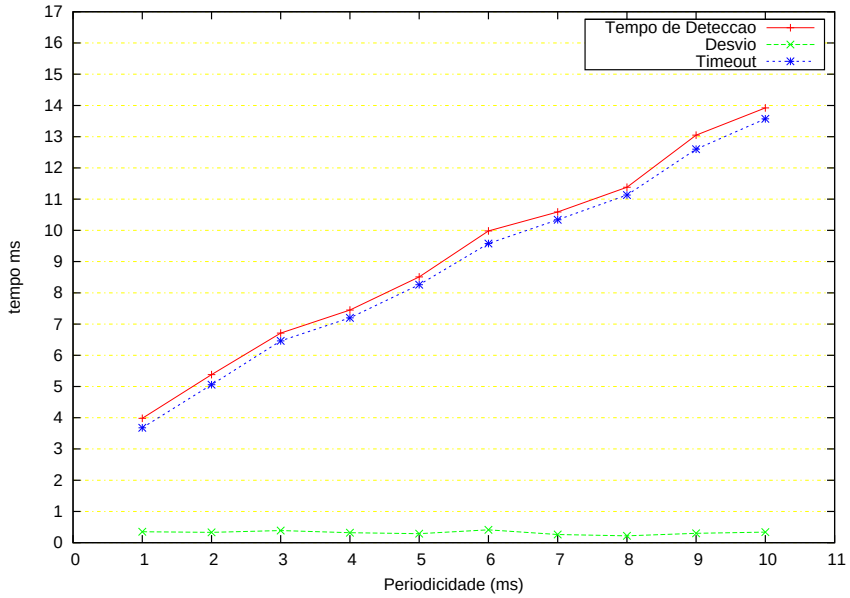
A tabela 1 apresenta o resultado médio de 50 consultas em um único objeto da $fdMIB$, a comunicação foi realizada entre dois *hosts* que estavam conectados a roteadores com IP válido. Entretanto, as duas redes locais estavam localizadas na Universidade Federal do Paraná. Como esperado, o WS adiciona um atraso na comunicação de aproximadamente 100%, mesmo assim, o valor resultante pode ser considerado razoável.

6. Conclusão

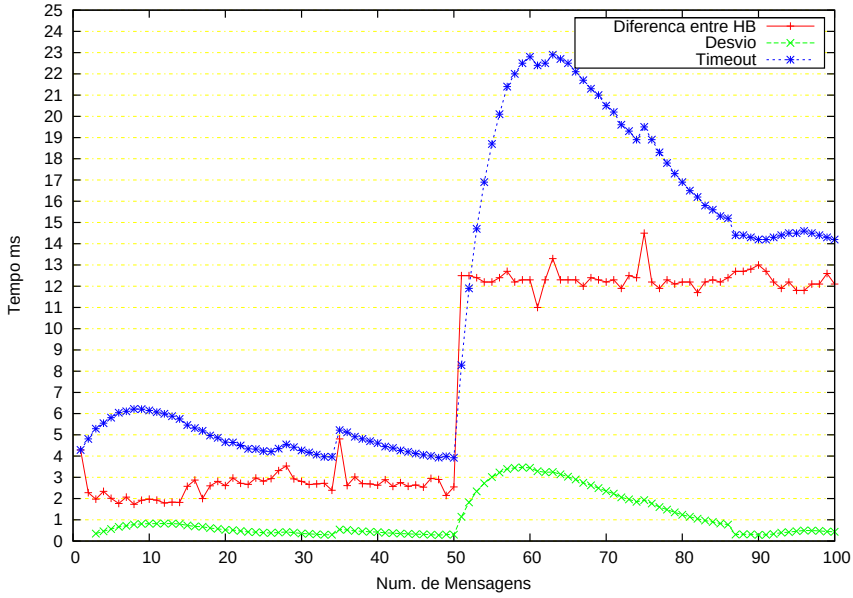
Neste trabalho uma nova arquitetura para um serviço de detecção de falhas em processos e *hosts* foi apresentada. Suas funcionalidades são oferecidas através de uma MIB que disponibiliza todas as tarefas para gerenciamento e detecção de falhas. Os serviços podem ser utilizados em redes locais ou em múltiplos sistemas autônomos, fazendo uso de um *gateway* SNMP realizando as traduções dos protocolos via WS.

Para demonstrar as funcionalidades da arquitetura, experimentos na rede local e entre redes locais foram realizados. A utilização de recursos como a CPU em diferentes periodicidades de monitoramento e a adaptação do *timeout*, demonstraram a eficiência do serviço. A comunicação via WS torna o serviço atraente para gerenciamento em diferentes sistemas autônomos.

Como trabalhos futuros, a arquitetura deverá ser experimentada em um ambiente de larga escala como o PlanetLab. Outras métricas podem ser avaliadas visando o uso da



(a) Tempo de detecção de falhas variando o Δ_i



(b) Δ_{to} Adaptativo

Figura 6. Serviço de detecção de falhas pela $fdMIB$.

$fdMIB$ para o cumprimento de propriedades em algoritmos distribuídos, como exemplo o protocolo de consenso.

Referências

Aguilera, M. K., 0007, W. C., and Toueg, S. (1997). Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *WDAG*.

Bertier, M., Marin, O., and Sens, P. (2003). Performance analysis of a hierarchical failure detector. In *DSN*.

Borran, F., Hutle, M., Santos, N., and Schiper, A. (2012). Quantitative analysis of consensus algorithms. *IEEE Trans. Dependable Sec. Comput.*, 9(2).

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2).

Charron-Bost, B., Pedone, F., and Schiper, A. (2010). *Replication: Theory and Practice*. Springer.

Chen, W., Toueg, S., and Aguilera, M. K. (2000). On the quality of service of failure detectors. In *DSN Proceedings of the 2000 International Conference on Dependable Systems and Networks*. IEEE Computer Society.

de Lima, W. Q., Alves, R. S., Vianna, R. L., Almeida, M. J. B., Tarouco, L. M. R., and Granville, L. Z. (2006). Evaluating the performance of snmp and web services notifications. In *NOMS*.

Dialani, V., Miles, S., Moreau, L., Roure, D. D., and Luck, M. (2002). Transparent fault tolerance for web services based architectures. In *Euro-Par*. Springer.

dos Santos Sá, A. and de Araújo Macêdo, R. J. (2005). An adaptive failure detection approach for real-time distributed control systems over shared ethernet. In *COBEM2005*.

Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *DOA*.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2).

Jacobson, V. (1988). Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM 88.

Moraes, D. M. and Duarte Jr., E. P. (2011). A failure detection service for internet-based multi-as distributed systems. In *ICPADS*. IEEE.

Net-SNMP (2014). *Net-Snmp: http://www.net-snmp.org/*. Acessado em 18/02/2014.

Nunes, R. C. and Jansch-Pôrto, I. (2004). Qos of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin. In *DSN*.

Renesse, R. V., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. Technical report, Cornell University, Ithaca, NY, USA.

Sergent, N., Défago, X., and Schiper, A. (2001). Impact of a failure detection mechanism on the performance of consensus. In *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, Seoul, Korea.

Wiesmann, M., Urbán, P., and Défago, X. (2006). An snmp based failure detection service. In *SRDS*. IEEE Computer Society.