Um Serviço de Detecção de Falhas com QoS Auto-Ajustável para Múltiplas Aplicações Simultâneas

Rogério C. Turchetti[1,2], Elias P. Duarte Jr.[1], Luciana Arantes[3], Pierre Sens[3]

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

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

[3]Laboratoire d'Informatique de Paris 6 / Université Pierre et Marie Curie, Paris, France

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

{luciana.arantes, pierre.sens}@lip6.fr

Abstract. *Failure detectors are used to monitor the state of the processes of a distributed application. In order to provide information of whether a monitored process is correct or has failed, a failure detector makes assumptions about time delays in the system. In this work we propose a failure detector service called QoS-CFDS (Quality of Service-Configurable Failure Detection Service). QoS-CFDS service is self-tuning, i.e., the service adapts its monitoring parameters according the environment variations and applications needs. We propose two strategies to adjust the detector in accordance with the requirements provided by multiple concurrent applications. The service was implemented using SNMP and experimental results are presented showing the performance of the detector, in particular the advantagens of using the self-tuning strategies to address the requirements for concurrent applications.*

Resumo. *Detectores de falhas monitoram os estados de processos de uma aplicação distribuída, efetuando hipóteses temporais sobre atrasos no sistema e disponibilizando informações sobre os estados destes processos. Neste trabalho é proposto o serviço de detecção de falhas denominado QoS-CFDS (Quality of Service-Configurable Failure Detection Service). O serviço QoS-CFDS é auto-ajustável, isto é, adapta seus parâmetros de monitoramento conforme as variações percebidas no ambiente e de acordo com as necessidades de cada aplicação. São propostas estratégias para ajustar a QoS do detector de acordo com os requisitos fornecidos por múltiplas aplicações simultâneas. Um protótipo do serviço proposto foi implementado com o protocolo SNMP e resultados experimentais são descritos para o desempenho do detector, incluindo os benefícios dos ajustes realizados para atender requisitos das aplicações simultaneamente.*

1. Introdução

Detectores de falhas são blocos fundamentais para auxiliar no desenvolvimento de aplicações distribuídas tolerantes a falhas. Um detector provê informações a respeito dos estados dos processos de um sistema. Os detectores de falhas foram propostos por Chandra e Toueg [Chandra and Toueg 1996] e sua abstração possibilita encapsular as premissas

temporais dos sistemas assíncronos, e, dependendo das propriedades que oferecem, permitem resolver problemas que não poderiam ser resolvidos de forma determinística em tais ambientes [Fischer et al. 1985]. Um detector de falhas é não confiável pois pode cometer erros, suspeitando erroneamente de um processo que não está falho. Porém, ao se dar conta que se enganou, corrige o seu erro parando de suspeitar do respectivo processo. Do ponto de vista funcional, os detectores de falhas monitoram os processos efetuando hipóteses temporais sobre atrasos no sistema. Em geral, o monitoramento realizado pelos detectores de falhas é baseado em troca de mensagens. O processo monitorado envia periodicamente mensagens de *heartbeat*. Caso o detector de falhas não receba um *heartbeat* dentro de um limite de tempo especificado, o respectivo processo monitorado passa a ser considerado suspeito de ter falhado.

Ao longo dos anos, foram propostas aplicações de detectores de falhas em diferentes contextos: sistemas de larga escala [Hayashibara et al. 2002], redes sem fio [Greve et al. 2011], computação em nuvem [Xiong et al. 2012], redes definidas por software [Turchetti and Duarte Jr. 2015]). Diversos aprimoramentos do serviço de detecção também foram propostos, como a melhoria de desempenho [Fetzer et al. 2001], interfaces mais adequadas para as aplicações [Felber et al. 1999] e parâmetros que podem ser adaptados de acordo com mudanças no ambiente [Bertier et al. 2003, Dixit and Casimiro 2010, Xiong et al. 2012, Xiaohui and Yan 2014, Tomsic et al. 2015]. Neste sentido, uma característica importante é a capacidade do detector de prever o instante de tempo em que a próxima mensagem de um monitoramento será recebida. Em geral, as abordagens são baseadas em estimativas que visam corrigir o tempo de espera (*timeout*) de acordo com o comportamento do ambiente de execução.

Um desafio prático dos detectores de falhas é oferecer um serviço que possa ser também adaptado para monitorar processos de acordo com as necessidades de cada aplicação distribuída. Em particular, é importante considerar que existem aplicações que possuem restrições temporais distintas umas das outras [Chen et al. 2000]. As restrições temporais dizem respeito ao tempo necessário para a detecção de uma falha, ou ao tempo para a correção de uma falsa suspeita ou ainda, ao intervalo de tempo entre duas falsas suspeitas. Por exemplo, definir um tempo curto para a detecção de falhas pode ser uma boa estratégia para certas aplicações, mesmo que este serviço esteja sujeito a detectar falhas de forma incorreta. Por outro lado, para outras aplicações que acomodam maiores atrasos, ter um serviço que comete menor número de erros pode ser melhor, mesmo que o tempo para a detecção de falha tenha um período mais longo. Portanto, é relevante oferecer um serviço capaz de se adaptar às necessidades de cada aplicação; há poucas soluções que abordam este problema [de Sá and de Araújo Macêdo 2010].

Chen et al. [Chen et al. 2000] propuseram um grupo de métricas que permitem avaliar a qualidade de serviço (QoS - *Quality of Service - QoS*) oferecida pelos algoritmos de detecção de falhas. Bertier et al. [Bertier et al. 2003] utilizam o algoritmo proposto por Chen para fornecer um serviço para detecção de falhas que pode ser compartilhado por diversas aplicações com base em suas restrições temporais. O presente trabalho tem por objetivo propor um serviço para detecção de falhas com QoS "auto-ajustável". Isto é, o serviço adapta seus parâmetros de monitoramento conforme as variações percebidas no ambiente (como atrasos de comunicação e sobrecarga de CPU) e de acordo com as necessidades de cada aplicação. O serviço proposto é denominado de QoS-CFDS (*QoS-*

Configurable Failure Detection Service) e pode ser compartilhado por múltiplas aplicações simultaneamente, fornecendo as garantias de qualidade de serviço requisitadas.

Para garantir parâmetros de QoS definidos pelas aplicações, o QoS-CFDS implementa uma estratégia que permite que aplicações especifiquem seus requisitos, por exemplo, uma aplicação pode requisitar um limite de tempo para a detecção de uma falha e o QoS-CFDS ajusta os parâmetros de monitoramento do detector de falhas com base nestas requisições de entrada. Em outras palavras, com base nos parâmetros fornecidos e no comportamento observado da rede, o QoS-CFDS é configurado para tentar satisfazer as solicitações das aplicações. Um algoritmo que recebe os valores de QoS fornecidos por uma aplicação e ajusta os parâmetros de monitoramento do detector, é proposto.

Uma contribuição deste trabalho é justamente propor, implementar e avaliar duas estratégias que permitem compartilhar o serviço de detecção de falhas com QoS entre diversas aplicações simultâneas. Em outras palavras, as estratégias ajustam a periodicidade de envio (representado por η) das mensagens de *heatbeat* que são carregadas via protocolo SNMP, para tentar cumprir com todas as requisições de QoS. A estratégia denominada de η_{max} busca encontrar um valor amplo o suficiente para acomodar todos os requisitos especificados pelas múltiplas aplicações, ao passo que, a estratégia denominada de η_{GCD} busca encontrar um máximo divisor comum dos requisitos de entrada para ser atribuído a η. Os resultados dos cômputos das estratégias definem um valor para η que pode ser aplicado em diferentes propósitos (e.g. considerando o tempo de detecção) e para diferentes ambientes de execução (e.g. considerando os atrasos na comunicação).

Um protótipo do serviço proposto foi implementado utilizando o protocolo SNMP (*Simple Network Management Protocol*) [Harrington et al. 2002]. Uma MIB (*Management Information Base*) denominada $fdMIB$ (*failure detection*) que permite a integração do serviço com as aplicações distribuídas e os processos a serem monitorados, é proposta. A $fdMIB$ armazena informações importantes sobre todos os componentes da arquitetura, como também executa a própria atividade de monitoramento dos processos. Resultados experimentais obtidos mostram o desempenho do QoS-CFDS, em especial os benefícios de oferecer o serviço para múltiplas aplicações simultâneas.

O restante do trabalho está organizado da seguinte forma. A Seção 2 traz uma breve introdução aos detectores de falhas e apresenta o modelo de sistema considerado neste trabalho. Na Seção 3 é apresentado o processo para a configuração de *timeout* adaptativo, como também estratégias para configurar o detector de falhas de acordo com as necessidades de múltiplas aplicações simultâneas. Na Seção 4 é descrita a arquitetura, implementação e resultados experimentais do QoS-CFDS. Por fim, a conclusão segue na Seção 5.

2. Detectores de Falhas e Modelo de Sistema

Os detectores de falhas foram propostos por Chandra e Toueg [Chandra and Toueg 1996] como uma abstração que possibilita encapsular o indeterminismo temporal dos sistemas assíncronos. O indeterminismo é a raiz da impossibilidade FLP [Fischer et al. 1985] e suscita que, devido à falta de conhecimento temporal dos sistemas assíncronos, no caso da falha de algum processo, não há algoritmos determinísticos que possam garantir a resolução do consenso. Por trás ao problema está a impossibilidade em determinar quando um processo está falho ou muito lento, devido a fatores como, por exemplo, aumento no

tráfego da rede ou sobrecargas de CPU.

Sendo assim, um detector de falhas permite encapsular o indeterminismo por indicar os estados de cada processo participante do sistema. Os estados indicam se um processo está ou não suspeito, de acordo com o monitoramento que ocorre por troca de mensagens e respeitando um limite de tempo (*timeout*). Um detector de falhas pode ser acessado por um processo p_i para obter informações referentes aos estados de outros processos. A informação retornada por um detector de falhas pode ser incorreta, isto é, um processo não falho pode ser considerado suspeito e vice-versa. Além disso, um detector de falhas pode fornecer informações inconsistentes divergindo de outros detectores.

Chandra e Toueg classificam os detectores de falhas através de duas propriedades: (1) completude (*completeness*) e (2) precisão (*accuracy*). Informalmente, a completude caracteriza a capacidade do detector de falhas suspeitar de todos os processos falhos (*crashed*), enquanto que a precisão caracteriza a capacidade do detector de falhas não suspeitar de processos não falhos (corretos). Estas propriedades permitem investigar os problemas que podem ser resolvidos com o uso de detectores, ou quais propriedades devem ser garantidas por um detector para resolver um determinado problema.

Modelo de Sistema

Neste trabalho considera-se um sistema distribuído assíncrono composto por um conjunto Π de n processos/hosts, incrementado com detector de falhas [Chandra and Toueg 1996]. Os processos falham por parada total (*crash*), ou seja, deixam de executar suas tarefas prematuramente. O monitoramento dos processos, em cada rede local, ocorre por um detector de falhas não confiável, o qual nunca falha. O monitoramento dos processos é realizado através de mensagens de *heartbeat* enviadas periodicamente pelos processos monitorados ao detector. O monitoramento resulta nos seguintes estados: um processo é considerado *suspeito* quando uma mensagem de *heartbeat* não é recebida dentro do intervalo de tempo esperado; *não suspeito* ou *correto* ocorre quando a mensagem de *heatbeat* é recebida pelo detector de falhas dentro do intervalo de tempo estabelecido. A comunicação é realizada por troca de mensagens através de um canal confiável, isto é, o canal não cria, não duplica, não perde e não altera mensagens de controle.

3. QoS-CFDS: Adaptação do Timeout e Configuração dos Parâmetros de QoS

Indiscutivelmente, uma das funções mais importantes executadas pelos detectores de falhas é o cálculo do intervalo de *timeout*. Uma abordagem comumente utilizada é a adaptativa [Bertier et al. 2003, Dixit and Casimiro 2010, Xiong et al. 2012, Xiaohui and Yan 2014]. O serviço para detecção de falhas proposto neste trabalho calcula o intervalo de *timeout* com base na estimativa do tempo de chegada da próxima mensagem de *heartbeat*, somado a uma margem de segurança adaptativa. Considere dois processos: p_i e p_j, onde p_j monitora p_i. A cada intervalo de tempo η (η representa a periodicidade de envio de mensagens), p_i envia uma mensagem de *heartbeat* para o monitor p_j. Sejam m_1, m_2,...,m_k as mensagens de *heartbeat*, m_k é a mais recente, recebidas por p_j. Sejam A_1, A_2,...,A_k os instantes de tempo nos quais as mensagens de *heartbeats* foram recebidas por p_j de acordo com o seu relógio local. EA pode ser estimado da seguinte forma:

$$EA_{k+1} = \frac{1}{k}\left(\sum_{i=1}^{k} A_{(i)} - \eta_i\right) + (k+1)\eta \quad (1)$$

EA é o instante de tempo estimado para a chegada da próxima mensagem de *heartbeat*. Sendo assim, o valor para o instante de tempo em que o próximo *timeout* (τ) irá expirar, é calculado da seguinte forma:

$$\tau_{(k+1)} = EA_{(k+1)} + \alpha_{(k+1)} \quad (2)$$

Para tornar a detecção de falhas mais precisa a margem de segurança α é calculada de acordo com o algoritmo de Jacobson [Jacobson 1988], que trabalha com base nos instantes de tempo em que as mensagens de *heartbeat* (hb) são recebidas, isto é, $diff_{(k)}$ apresentado na expressão 3, representa a diferença entre a '*k-ésima*' e a '*k-ésima*-1' mensagens:

$$diff_{(k)} = hb_k - hb_{k-1} \quad (3)$$

Na expressão (4), $delay_{(k+1)}$ é o valor do atraso que é calculado em termos da diferença $diff_{(k)}$, e γ representa o peso das novas amostras, Jacobson sugere o valor de $\gamma = 0.1$:

$$delay_{(k+1)} = (1-\gamma) \cdot delay_{(k)} + \gamma \cdot diff_{(k)} \quad (4)$$

$var_{(k+1)}$ representa a variação da diferença:

$$var_{(k+1)} = (1-\gamma) \cdot var_{(k)} + \gamma \cdot (|diff_{(k)} - var_{(k)}|) \quad (5)$$

Por fim, a margem de segurança $\alpha_{(k+1)}$ é obtida através da expressão 6, onde β e ϕ são constantes que tipicamente recebem os seguintes valores [Bertier et al. 2003]: $\beta = 1$, $\phi = 4$ e $\gamma = 0.1$.

$$\alpha_{(k+1)} = \beta \cdot delay_{(k+1)} + \phi \cdot var_{(k+1)} \quad (6)$$

Com base nos cálculos apresentados nesta seção, o intervalo de *timeout* τ é ajustado a cada nova mensagem de *heartbeat* recebida pelo detector de falhas.

Configurando o Detector de Falhas com Base nos Parâmetros de QoS

Um desafio dos detectores de falhas é oferecer um serviço que possa ser adaptado de acordo com as necessidades de cada aplicação. Nesta seção é proposto um algoritmo que permite configurar o detector de falhas de acordo com as necessidades de QoS de cada aplicação. Inicialmente, as funcionalidades do algoritmo são apresentadas considerando requisições de uma única aplicação. Subsequentemente, o algoritmo é estendido para funcionar com duas estratégias (denominadas η_{max} e η_{GCD}) que trabalham de acordo com as requisições de QoS para múltiplas aplicações. Desta forma, mesmo se múltiplas aplicações usarem o detector de falhas para monitorar o mesmo processo, a estratégia deve suportar diferentes requisitos de QoS, e um único intervalo de *heartbeat* deve ser calculado, satisfazendo todas as requisições.

Uma aplicação entra com seus requisitos de QoS através dos seguintes parâmetros, que representam as métricas primárias propostas por Chen et al. [Chen et al. 2000]:

- T_D^U: limite máximo para o tempo de detecção;
- T_M^U: limite máximo para a duração de um erro;
- T_{MR}^L: limite mínimo para a ocorrência entre dois erros consecutivos.

A partir destes parâmetros, o detector de falhas encontra um valor apropriado para η, como descrito a seguir inicialmente para uma aplicação e depois para múltiplas aplicações.

Ativando QoS: *Uma Única Aplicação*

Para que o serviço de detecção de falhas possa ativar a QoS, a aplicação requisitante deve entrar com os parâmetros descritos acima. Além disso, três outros parâmetros são necessários e providos pelo detector de falhas: probabilidade de perda de mensagens (p_L), variância do atraso ($V(D)$) e estimativa do tempo da próxima mensagem de *heartbeat* (EA). A probabilidade de perda de mensagens é encontrada calculando $p_L = L/k$, onde k é o número total de mensagens que foram enviadas e L é o número de mensagens perdidas. $V(D)$ é calculado observando a variância nos tempos das mensagens recebidas. Uma aplicação cliente envia os dados de entrada que são processados através de dois passos apresentados no Algoritmo 1. O Algoritmo 1, adaptado de [Chen et al. 2000], tem por objetivo encontrar um valor apropriado para η de forma a atender as necessidades de cada aplicação.

O Algoritmo 1 é composto por dois passos. No passo 1 as aplicações fornecem seus parâmetros de QoS e os valores de p_L e $V(D)$ são obtidos na linha 3 e, a seguir, o menor valor de η_{max} é obtido (linha 7 ou linha 9), dependendo de qual dos valores é menor: $min(\gamma * T_M^U, T_D^U)$. Vale ressaltar que existem duas condições em que a QoS não pode ser ativada, sendo elas: (i) se no passo 1 o valor de η_{max} for igual a zero ou (ii) se no passo 2 o intervalo de *heartbeat* η não for obtido. Entretanto, se no passo 1 $\eta_{max} \geq 0$, então a linha 10 do algoritmo é executada invocando o passo 2.

O objetivo do passo 2 é encontrar um valor para η que cumpra com os requisitos da aplicação. Como η deve ser maior do que zero e menor do que η_{max}, propomos inicializar η igual a η_{max} (linha 16). A cada novo valor definido para η, a seguinte condição é testada $f_\eta \geq T_{MR}^L$ (linha 17). Enquanto esta condição não for verdadeira, f_η é recalculado e um novo teste é executado.

Algorithm 1: Algoritmo para configuração de QoS.

```
1  Step1 (T_D^U, T_M^U, T_MR^L)                          ▷ no Step1 calcula-se o valor o valor para η_max
2
3    γ ← (1 − p_L)(T_D^U)^2/(V(D) + (T_D^U)^2);
4
5    if ((γ > 0) and (T_D^U > 0) and (T_M^U > 0)) then
6        if ((γ * T_M^U) > T_D^U) then
7            η_max ← T_D^U;
8        else
9            η_max ← γ * T_M^U;
10       run Step2(T_D^U, T_MR^L, η_max)
11   else
12       return ("QoS não pode ser ativada");

13 Step2 (T_D^U, T_MR^L, η_max)                          ▷ no Step2 calcula-se o valor para η
14
15   f_n ← 1;
16   η ← η_max;                                          ▷ η_max é utilizado como valor para inicializar η
17   while (fη < T_MR^L) do
18       for (j ← 1; j ≤ T_D^U/η; j++) do
19
20           f_n ← f_n * (V(D)+(T_D^U − jη)^2)/(V(D)+(p_L(T_D^U − jη)^2);
21
22       f_n ← f_n* η;
23       η ← η − (η * 0.01)                              ▷ para o valor de fη crescer reduzimos o valor de η;
24   return (η);
```

Para cada valor de η, f_η é calculado em sucessivas iterações com j=1 até $j > T_D^U/n$ (linhas 18 e 20). Note que, quando η diminui, f_η aumenta, o oposto também é verdadeiro. Por esta razão, sempre reduzimos o valor de η (linha 23, reduz em 1% o valor de η) até que $f_\eta \geq T_{MR}^L$. Neste ponto o valor de η é finalmente encontrado. Vale destacar que este é o maior valor possível que permite cumprir com os parâmetros de QoS fornecidos pela aplicação, assim pode ser usado um η menor, mas que gera mais mensagens de monitoramento.

Ativando QoS: *Múltiplas Aplicações*

Para encontrar um valor para η que satisfaça aos requisitos de múltiplas aplicações, são propostas duas diferentes estratégias, η_{max} e η_{GCD}, descritas a seguir. A proposta η_{max} busca um valor do intervalo de monitoramento grande o suficiente para acomodar todos os requisitos. A proposta η_{GCD} parte do princípio que, se um processo envia uma mensagem de *heartbeat* a cada x unidades de tempo, ele também pode enviar um *heartbeat* a cada y unidades de tempo, se x divide y.

Estratégia η_{max}: ao invés de atribuir $\eta_{max} = min(\gamma T_M^U, T_D^U)$ para uma única aplicação, para n aplicações $\eta_{max} = min(\gamma T_{M1}^U, T_{D1}^U, \gamma T_{M2}^U, T_{D2}^U, ..., \gamma T_{Mn}^U, T_{Dn}^U)$, onde $i = 1...n$, T_{Mi}^U e T_{Di}^U correspondem aos requisitos da aplicação i (App_i). Em outras palavras, $\eta_{max} = min(\eta_1, \eta_2, ..., \eta_n)$, Com esta única modificação, é possível usar o Algoritmo 1 para calcular o valor apropriado de η.

Estratégia η_{GCD}: esta estratégia calcula o máximo divisor comum (*Greatest Common Divisor - GCD*) entre todos os η_i, onde $i = 1...n$, e n é o número aplicações.

Inicialmente, para cada η_i um novo η'_i é encontrado da seguinte forma: $\eta'_i = 2^n$ onde $2^n < \eta_i$ e $\eta'_i \in \mathbb{Z}^+$. Desta forma, $\eta_{GCD} = GCD(\eta'_i, ..., \eta'_n)$. Se $\eta_{GCD} > 0$, o intervalo de monitoramento fica definido; caso contrário esta estratégia não pode ser aplicada. Observe que o valor de η_{GCD} é sempre menor do que o valor da estratégia η_{max}. Neste sentido, esta estratégia resulta em um número maior de mensagens na rede. Por outro lado, há benefícios como a redução do tempo de detecção de falhas (T_D).

Vale ressaltar que, utilizando múltiplas aplicações, a proposta apresentada se adapta mesmo em condições em que ocorre violação de QoS causadas por atrasos na comunicação. Por exemplo, se o QoS-CFDS detectar que não é possível sustentar os valores mínimos de QoS requeridos, ele reajusta o valor de η aumentando para um outro valor pré-estabelecido, caso haja. Por exemplo, considere duas aplicações utilizando a estratégia η_{max}, o valor final é $\eta_{max} = min(\eta_1, \eta_2)$, se η_1 foi escolhido por ser o menor e caso este valor não cumpra com os requisitos de QoS, o próximo valor utilizado será η_2. Nesse caso o serviço garante, mesmo que de forma parcial, a QoS requerida para determinadas aplicações.

4. Implementação e Resultados Experimentais

O serviço QoS-CFDS foi implementado usando o protocolo de gerência de redes da Internet, Simple Network Management Protocol (SNMP) [Harrington et al. 2002]. A Figura 1 apresenta a arquitetura do QoS-CFDS. As aplicações (App) podem se registrar no *Host Monitor* para receber informações sobre os estados dos processos que executam em um *Host Monitorado*. O *Host Monitor* configura um *timeout* adaptativo (τ), como descrito na seção anterior, em cada *Host Monitorado*, que encaminha *heartbeats* ao *Host Monitor* periodicamente com intervalo η. São utilizadas mensagens SNMP para transportar as mensagens de *heartbeat*.

Conforme pode ser visto na Figura 2, para uma aplicação encaminhar seus valores de QoS, algumas informações são repassadas por parâmetro, como exemplo: endereço IP do *Host Monitor*, o identificador do objeto onde os dados são armazenados (*Object Identifier* - OID), os requisitos de QoS, entre outras informações. Além das informações de QoS, o componente *Estimator* é responsável por realizar cálculos estatísticos (p_L, $V(D)$ e EA) e o componente *Configurator* executa o Algoritmo 1.

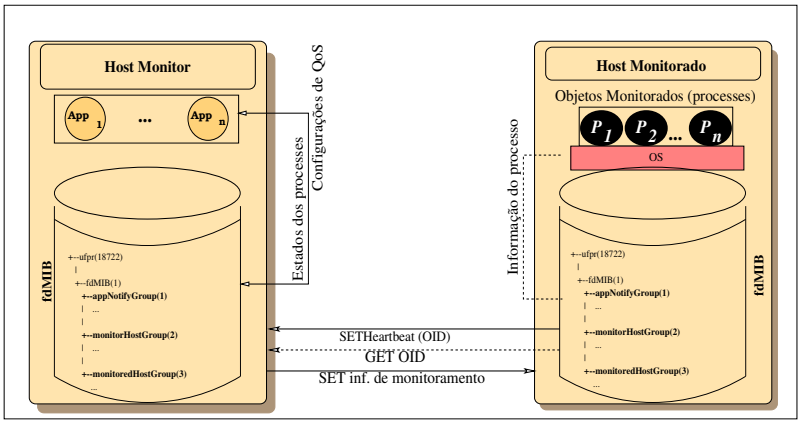


Figura 1. Arquitetura do QoS-CFDS.

Uma MIB ($fdMIB$) implementada no *Host Monitor* executa o monitoramento

dos estados dos processos e armazena informações a respeito de todos os componentes existentes na arquitetura. A $fdMIB$ consiste de três grupos, `appNotifyGroup`, `monitorHostGroup` e `monitoredHostGroup`, descritos a seguir.

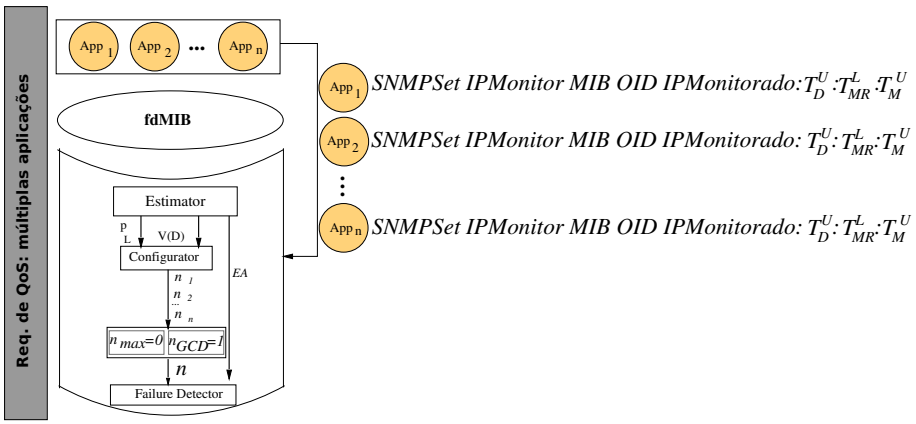


Figura 2. A aplicação configura os parâmetros de QoS na $fdMIB$.

O grupo `appNotifyGroup` é responsável por manter informações sobre os parâmetros de QoS fornecidos pelas aplicações. Cada aplicação que desejar receber informações sobre os estados dos processos deve informar, além do seu endereço de IP e porta, os seguintes parâmetros: T_D^U, T_M^U e T_{MR}^L. Se existirem múltiplas aplicações, é possível definir qual estratégia utilizar η_{max} η_{GCD}. Se o objeto não é setado, η_{max} é utilizada por default. O `appNotifyGroup` também possui um objeto denominado de `receiveHB`, usado para controlar as mensagens de *heartbeat* recebidas dos processos monitorados. A cada mensagem de *heartbeat* recebida, a $fdMIB$ atualiza o estado do correspondente processo. A mensagem é identificada pelo seu OID (*Object IDentifier*), atualizando o valor do objeto que pertence ao `monitorHostGroup`. Por fim, para aplicações registradas na $fdMIB$, é possível enviar notificações de mudanças de estados dos processos via SNMP *traps* (`notifyTrap`).

O grupo `monitorHostGroup` mantém informações sobre os processos monitorados, incluindo: endereço IP, porta e estado do processo (0=não suspeito, 1=suspeito). Outras informações incluem: número de falsas suspeitas, tempo de detecção da falha, intervalo de *heartbeat*, cálculos estatísticos conforme apresentado na Figura 2, entre outras. No `monitorHostGroup` são gerenciadas todas as atividades de monitoramento, em particular o gerenciamento e controle das *threads* responsáveis por implementarem o *timeout* para cada processo monitorado.

O grupo `monitoredHostGroup` é responsável por enviar periodicamente as mensagens de *heartbeat* ao *Host Monitor* sendo, portanto, executado pelo *Host Monitorado*. Com a finalidade de se comunicar com o *Host Monitor*, as seguintes informações são mantidas pelos objetos: endereço IP do monitor, OID do próprio *Host Monitorado* (o OID é usado para identificar o *Host Monitorado* na MIB) e o intervalo de *heartbeat*. Vale lembrar que η é calculado no *Host Monitor* de acordo com as requisições de QoS, para então ser enviado ao *Host Monitorado*, de forma a configurá-lo para encaminhar mensagens na periodicidade indicada.

4.1. Avaliação Experimental do QoS-CFDS

Nesta seção são apresentados experimentos executados com o objetivo de avaliar o serviço para detecção de falhas proposto. Os experimentos foram executados em uma rede local. Com a finalidade de avaliar o QoS-CFDS trabalhando com diferentes requisições de QoS, três aplicações com seus respectivos parâmetros de QoS são consideradas e apresentadas na Tabela 1. Por exemplo, considerando a App_1 os seguintes valores de QoS são requisitados: uma falha deve ser detectada no período máximo de 8 segundos (T_D^U=8000ms), o detector de falhas corrige um erro no limite máximo de 1 minuto (T_M^U=60000ms) e o detector de falhas comete no máximo 1 erro por mês (T_{MR}^L=2592000000ms).

Tabela 1. Parâmetros de QoS de cada aplicação.

Aplicações	T_D^U **(ms)**	T_M^U **(ms)**	T_{MR}^L **(ms)**
App_1	8000	60000	2592000000
App_2	14000	120000	2592000000
App_3	16000	240000	2592000000

Com base no Algoritmo 1, nos dados de entrada apresentados na Tabela 1 e considerando as estratégias para o cômputo do η, tem-se os seguintes resultados: η_{max}= $min(1.954467, 3.901890, 4.694764) \rightarrow \eta$=1.95 e η_{GCD}= $GCD(1, 2, 4) \rightarrow \eta$=1.

Nos experimentos p_L=0.01 e $V(D)$=0.02. Além disso, como apresentado na Expressão 1, o cálculo de EA utiliza um histórico das últimas mensagens de *heartbeat* recebidas, o tamanho da janela é indicado nos experimentos.

Resultados Experimentais

Para os experimentos foram utilizadas as seguintes máquinas físicas conectadas em rede Ethernet 100Mbps: processador Intel Core i5 CPU 2.50GHz com 4 núcleos e sistema operacional Ubuntu 12.04.4 executando a $fdMIB$ como monitor, o *host* monitorado 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. A MIB foi projetada com o *toolkit* Net-SNMP[1], versão 5.4.4.

Nos gráficos das Figuras 3, 4(a) e 4(b) são ilustrados os envios de 200 mensagens de *heartebeat* que ocorrem entre o monitor e o *host* monitorado. Durante as primeiras 50 mensagens, o intervalo de envio de mensagens de *heartbeat* é configurado igual a 1000ms. Este valor justifica-se por ser um valor que contempla tanto a estratégia η_{max}=1.95s, quanto a η_{GCD}=1s. Após o envio de 50 mensagens, o intervalo de *heartbeat* é alterado para 5000ms. Novamente, este valor justifica-se por não contemplar nenhuma das estratégias, cujos máximos são: $\eta_{max} \leq$ 4.69s e $\eta_{GCD} \leq$ 4s.

Além disso, essa mudança no valor de η é para verificar o comportamento do serviço de detecção de falhas através da simulação de sobrecarga nos processos ou canais de comunicação (por exemplo, quando o intervalo de *heartbeat* passa de 1000ms para 5000ms, esse atraso poderia ser causado por alguma sobrecarga), como também o comportamento do QoS-CFDS considerando os parâmetros de QoS.

[1]http://www.net-snmp.org

Para calcular o EA, o tamanho da janela deslizante é definido para 5 mensagens. Pode-se observar na Figura 3 que há um tempo inicial para a estabilização do *timeout*. Este mesmo cenário pode ser observado quando ocorre a mudança no intervalo de emissão de η=1000ms para η=5000ms. Além disso, devido a esta mudança brusca no valor de η, em que o *timeout* atinge o maior valor, é possível observar na Figura 4(a) que ocorrem 6 falsas detecções. Estas podem ser observadas através do parâmetro de T_M. A Figura 4(a), também apresenta as 3 aplicações com seus respectivos parâmetros de QoS (vide Tabela 1), onde cada aplicação especifica seu próprio T_D^U. Com base nestes parâmetros, é possível observar, na Figura 4(a), que o tempo T_D calculado nos experimentos é sempre menor do que T_D^U especificado pela App_1, T_D^U da App_2 e T_D^U especificado pela App_3. Por esta razão, as falsas suspeitas detectadas e mostradas na Figura 4(a), não são notificadas para as aplicações, mesmo quando η atinge o valor de 5.

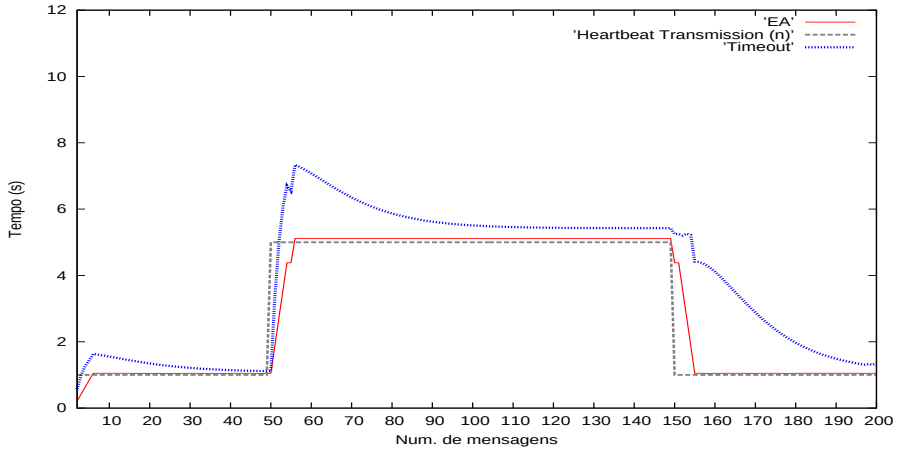
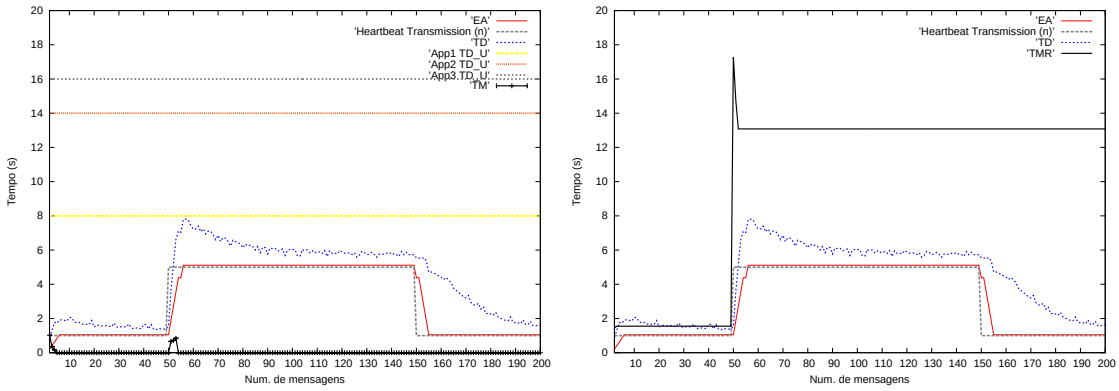


Figura 3. *Timeout* **adaptativo para diferentes intervalos de envio de** *heartbeats*.

Uma situação similar ocorre na Figura 4(b), onde a média do T_{MR} é de 13086ms, considerando as falsas detecções, de acordo com as requisições de QoS, T_{MR} deveria ser maior ou igual ao valor especificado pelas aplicações, isto é, T_{MR}^L=2592000000ms. Entretanto, como foi observado na Figura 4(a), nenhuma das 6 falsas suspeitas foram reportadas para as aplicações ($T_D \leq \min({T_D}^U{}_1, {T_D}^U{}_2, {T_D}^U{}_3)$), por esta razão, o valor do T_{MR} é 0 e não existem parâmetros de QoS que foram violados.

Na Figura 5 é avaliada a utilização de CPU e memória do contexto do *Host Monitor*. Para sobrecarregar o uso destes recursos pelo serviço de monitoramento do detector de falhas, os experimentos foram executados com diferente número de objetos na MIB, sendo que cada objeto corresponde a um processo monitorado, e para cada processo monitorado são escalonadas, periodicamente, *threads* responsáveis pelo controle do *timeout*. Um único *host* físico é responsável por enviar mensagens de *heartbeat*, identificando cada um dos processos monitorados inclusos na MIB. Os experimentos envolvem a inclusão de cada um dos objetos na $fdMIB$ e o recebimento das mensagens de *heartbeat*, o intervalo de envio de *heartbeat* é de 1ms. Os experimentos são executados no intervalo de uma hora, variando o número de objetos para monitoramento. O uso da memória ilustrado na Figura 5(a), cresce quase linearmente até 400 objetos, havendo uma estabilização na utilização de memória, não chegando a atingir 0.2%.

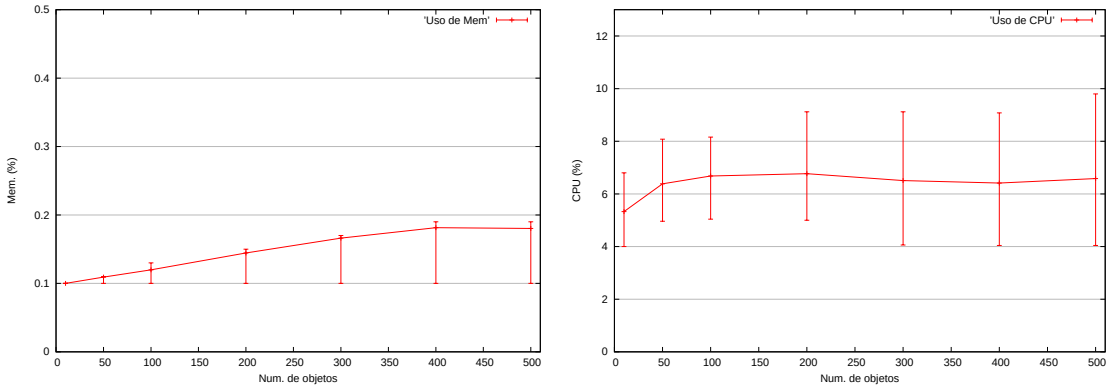
Quanto à utilização de CPU, pode-se observar na Figura 5(b) que cresce quando o



(a) T_D (Tempo de detecção) e T_M (Tempo de duração de um erro).

(b) T_{MR} (Tempo para recorrência ao erro).

Figura 4. Os parâmetros de QoS não são violados.



(a) Utilização de memória.

(b) Utilização de CPU.

Figura 5. Desempenho da $fdMIB$ no aumento de objetos de monitoramento.

número de objetos aumenta até 100, permanecendo quase constante para os experimentos em que o número de objetos cresce de 100 até 500, evidenciando que o serviço proposto é escalável. Por outro lado, pode-se observar que os valores máximos para a utilização de CPU apresentam um crescimento quase linear, estes picos são resultantes do registro inicial dos objetos na $fdMIB$. Entretanto, vale ressaltar que o procedimento para o registro de um processo na $fdMIB$, é necessário somente na primeira vez em que for monitorado.

Para comparar as estratégias η_{max} e η_{GCD} propostas neste trabalho, utiliza-se os parâmetros apresentados na Tabela 1, na qual os resultados obtidos para η foram: 1.95s e 1s, respectivamente. A Tabela 2 apresenta um comparativo entre as estratégias η_{max} e η_{GCD}. Para executar este experimento é utilizada a métrica P_A que corresponde à probabilidade de uma resposta exata, ou seja, é a probabilidade com que detectores de falhas geram saídas corretas em instantes de tempo aleatórios. P_A pode ser calculada da seguinte

forma:

$$P_A = 1 - \frac{E(T_M)}{E(T_{MR})} \tag{7}$$

O experimento mostrado na Tabela 2 foi executado durante o período de 1 hora, onde duas falhas foram simuladas através da omissão de duas mensagens de *heartbeat*, forçando o estouro do *timeout*. É possível verificar que η_{max} é melhor em termos de número de mensagens e mais apropriada para aplicações que não necessitam de um pequeno tempo de detecção. Por esta razão, η_{max} é mais apropriada para o monitoramento em redes de longa distância onde, em geral, as aplicações são mais tolerantes a atrasos. Como pode ser visto na Tabela 2, a estratégia η_{GCD} apresenta um menor tempo de detecção e melhor resultado para a métrica P_A. Portanto, sendo mais apropriada para o monitoramento em redes locais onde as aplicações geralmente não toleram períodos longos de atraso.

Tabela 2. Comparativo entre as duas estratégias: η_{max} and η_{GCD}.

	η (s)	T_D (ms)	T_M (ms)	T_{MR} (ms)	P_A	Num. de mensagens HB	Num. de falsas detecções
η_{max}	1.95	2458	1770	61190	0.9711	1754	2
η_{GCD}	1.00	1320	690	60592	0.9998	3551	2

5. Conclusão

Neste trabalho foi proposto o serviço QoS-CFDS para detecção de falhas de processos de sistemas distribuídos. O monitoramento dos estados dos processos é "auto-ajustável", executado com base nos parâmetros de QoS que permitem configurar o detector a partir dos requisitos simultâneos de múltiplas aplicações. Para calcular estes parâmetros de QoS foram propostas duas estratégias (η_{max} e η_{GCD}) que permitem deduzir o valor do intervalo de *heartbeat* (η) com base nas informações de QoS fornecidas para o QoS-CFDS. Neste sentido, observou-se que a estratégia η_{max} é mais apropriada para aplicações que toleram maiores atrasos na comunicação, ao passo que a estratégia η_{GCD} é indicada para casos em que as restrições forem maiores, obtendo maior probabilidade para uma resposta exata (P_A) e menor tempo para a detecção de uma falha. Os resultados experimentais mostram os benefícios dos ajustes realizados pelo QoS-CFDS com base nos valores de QoS, em especial a possibilidade de omitir, para as aplicações, falsas suspeitas cometidas pelo detector de falhas. Resultados para uso de CPU e memória demonstraram o serviço proposto é escalável em termos de número de objetos para monitoramento. Por fim, foi mostrada também a eficiência da estratégia utilizada para a adaptação do *timeout* em diferentes situações de atrasos de comunicação. Trabalhos futuros incluem a implementação de aplicações distribuídas tolerantes as falhas construídas sobre o serviço QoS-CFDS.

Referências

Bertier, M., Marin, O., and Sens, P. (2003). Performance Analysis of a Hierarchical Failure Detector. In *International Conference on Dependable Systems and Networks (DSN)*.

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

Chen, W., Toueg, S., and Aguilera, M. K. (2000). On the Quality of Service of Failure Detectors. In *International Conference on Dependable Systems and Networks (DSN)*.

de Sá, A. S. and de Araújo Macêdo, R. J. (2010). QoS Self-configuring Failure Detectors for Distributed Systems. In *International Conference on Distributed Applications and Interoperable Systems (DAIS)*. Springer-Verlag.

Dixit, M. and Casimiro, A. (2010). Adaptare-FD: A Dependability-Oriented Adaptive Failure Detector. *Symposium on Reliable Distributed Systems*.

Felber, P., Defago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *International Symposium on Distributed Objects and Applications.*

Fetzer, C., Raynal, M., and Tronel, F. (2001). An adaptive failure detection protocol. In *Pacific Rim International Symposium on Dependable Computing*.

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

Greve, F., Sens, P., Arantes, L., and Simon, V. (2011). A Failure Detector for Wireless Networks with Unknown Membership. In *Euro-Par 2011 Parallel Processing*, volume 6853. Springer Berlin Heidelberg.

Harrington, D., Presuhn, R., and Wijnen, B. (2002). An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. Request for Comments 3411 (RFC3411).

Hayashibara, N., Cherif, A., and Katayama, T. (2002). Failure detectors for large-scale distributed systems. In *21st IEEE Symposium on Reliable Distributed Systems*.

Jacobson, V. (1988). Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*.

Tomsic, A., Sens, P., Garcia, J., Arantes, L., and Sopena, J. (2015). 2W-FD: A Failure Detector Algorithm with QoS. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS*.

Turchetti, R. C. and Duarte Jr., E. P. (2015). Implementation of a Failure Detector Based on Network Function Virtualization. In *Workshop on Dependability Issues on SDN and NFV, International Conference on Dependable Systems and Networks (DSN)*.

Xiaohui, W. and Yan, Z. (2014). Adaptive failure detector A-FD. In *7th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*.

Xiong, N., Vasilakos, A., Wu, J., Yang, Y., Rindos, A., Zhou, Y., Song, W.-Z., and Pan, Y. (2012). A Self-tuning Failure Detection Scheme for Cloud Computing Service. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*.