


Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS

Sérgio Gorender

Related papers

[Download a PDF Pack](#) of the best related papers 



[Implementação e Análise de Desempenho de um Mecanismo Adaptativo para Tolerância a F...](#)
Sérgio Gorender

[VS Gráfica VS Gráfica Conceitos e Projeto 5ª Edição SISTEMAS DISTRIBUÍDOS Conceitos e Projeto 5ª ...](#)
Anderson Ribeiro

[DisCusS: desenvolvendo um Serviço de Consenso generico, simples e modular](#)
Lasaro Camargos

Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS

Sérgio Gorender^{1 2}, Raimundo José de Araújo Macêdo³

LaSiD - Laboratório de Sistemas Distribuídos, UFBA
Prédio do CPD, Av. Adhemar de Barros, S/N, 40170-110, Salvador/Ba.
{gorender, macedo}@ufba.br

Abstract

Fault Tolerance is a fundamental requirement for the correct functioning of the new safe-critical Internet applications where service interruption may result in great losses (such as e-commerce or environmental monitoring, etc.). Nevertheless, none of the existing fault tolerant distributed system models consider the new architectures intended for providing services with QoS (Quality-of-Service), such as IntServ and DiffServ proposed by the IETF [1,2]. This paper presents a novel approach to deal with fault tolerance in such environments. Our model is adaptable to the dynamic QoS conditions, allowing the exploitation of the best of the two extreme possible distributed system scenarios (synchronous and asynchronous). Furthermore, our model is particularly powerful in the sense that it allows for processes with distinct QoS views to continue their computations and cooperating in a safe, fault tolerant manner. To realize that, we introduce the concept of timely and not timely failure detectors and present a consensus protocol which works correctly even if different processes have distinct views of the local failure detector quality. The consensus protocol is optimum. That is, if all failure detectors are timely (synchronous system), it tolerates $f = n-1$ crash faults, and for the worse case scenario (asynchronous system with $\diamond S$ failure detectors [3]), it tolerates $n/2 - 1$ faults, where n is the number of processes.

Resumo

Tolerância a falhas é fundamental para o bom funcionamento das novas aplicações de segurança crítica da Internet, onde a interrupção do serviço pode resultar em prejuízos ou perdas importantes (*e-commerce*, monitoramento ambiental, etc.). No entanto, nenhum dos modelos de sistemas distribuídos tolerantes a falhas existentes consideram as novas arquiteturas para prover serviços com qualidade (QoS), como as IntServ e DiffServ propostas pela IETF [1,2]. Este artigo apresenta uma proposta inovadora para o tratamento de tolerância a falhas em tais ambientes. Nosso modelo é adaptável às condições dinâmicas que ocorram na QoS, permitindo a exploração do melhor dos dois cenários extremos nos modelos de sistemas distribuídos (síncrono e assíncrono). Mais ainda, nosso modelo é particularmente poderoso no sentido de permitir que processos com diferentes visões da QoS possam continuar cooperando e executando suas ações de forma segura (i.e., tolerantes a falha). Para isso, introduzimos o conceito de detectores de falhas isócronos e não isócronos e apresentamos um protocolo de consenso que é capaz de funcionar corretamente mesmo quando diferentes processos têm acesso a detectores com qualidades distintas. O protocolo de consenso apresentado é ótimo. Ou seja, se todos os detectores são isócronos (sistema síncrono), o algoritmo tolera $f = n-1$ falhas do tipo *crash* e no pior caso (sistema assíncrono com detectores $\diamond S$ [3]) tolera $n/2 - 1$ falhas, onde n é o número de processos.

Palavras-chave: sistemas distribuídos, tolerância a falhas, consenso, QoS, detectores de falhas.

¹Professor da Faculdade Ruy Barbosa e do DCC/UFBA

² Aluno de doutorado do Centro de Informática da UFPE.

³ Este autor agradece o apoio parcial do projeto ARGO, cooperação CNPQ/INRIA, Fase II.

1. Introdução

Um sistema distribuído é um conjunto de N processos $\Pi = \{p_1, p_2, \dots, p_n\}$ localizados em um ou mais sítios (dispositivos computacionais), conectados entre si através de canais de comunicação e executando de forma concorrente. A criação e utilização de sistemas distribuídos tem crescido proporcionalmente à evolução das redes de computadores e ao crescimento de seu uso. A demanda por mais e melhores serviços de rede é também crescente, assim como a busca por custos menores. Empresas que iniciaram o uso de aplicações distribuídas em suas redes privadas, pretendem passar a usar as redes públicas, especialmente a Internet, como ambiente de comunicação para estas aplicações. Estas empresas esperam obter da Internet um ambiente de comunicação seguro, confiável, e a um custo mais acessível.

Para cumprir com estas demandas, novas tecnologias e arquiteturas de rede têm sido desenvolvidas, testadas e padronizadas. Com a nova geração da Internet, obtemos serviços com qualidade no que diz respeito a aspectos como velocidade e confiabilidade. As novas aplicações distribuídas, em desenvolvimento para esta nova geração da Internet, necessitam seguir um modelo de sistema distribuído que leve em consideração a QoS (Quality-of-Service) obtida. Este modelo deve viabilizar características como recursos compartilhados, alta disponibilidade, requisitos fim-a-fim, concorrência, adaptação (automaticamente acompanhar o estado do ambiente e se adaptar a modificações), entre outras [4].

Dentre as aplicações viabilizadas pela nova geração da Internet estão aquelas tidas como de segurança crítica, onde a interrupção da aplicação devido a falhas pode resultar em prejuízos ou perdas importantes (*e-commerce*, monitoramento ambiental, etc.). Portanto, tolerância a falhas passa a ser um requisito de fundamental importância nesses ambientes, pois um sistema tolerante a falhas continua funcionando mesmo na presença de falhas de componentes [5].

O problema do consenso distribuído tem sido estudado por vários pesquisadores como forma de avaliar os modelos de sistemas distribuídos quanto a tolerância a falhas [6, 7, 3]. Isso se deve ao fato do consenso ser um problema básico que pode ser usado para resolver outros problemas de concordância (decisão de transações, *atomic commitment*, ordenação de mensagens, *membership*, entre outros) [3, 8, 9, 10]. Colocado de uma forma simples, o problema de consenso consiste em um grupo de processos concordar com um determinado valor como resultado de uma computação, uma vez que cada um possa ter proposto um valor diferente [11].

Os modelos existentes de sistemas distribuídos variam dependendo de restrições temporais, vinculadas a determinados aspectos do ambiente (exemplo, tempo de transferência de mensagens ou computações de ações locais). Quando os limites temporais existem e são conhecidos, permitem que haja uma estimativa com relação ao tempo que levará para se executar protocolos do sistema e funções da própria aplicação distribuída. Esses sistemas são denominados de síncronos [6, 12, 13, 14, 15]. Os sistemas síncronos são particularmente eficientes para tolerância a falhas pois permitem a detecção de falhas simplesmente a partir do uso de *timeouts*. Por outro lado, nos sistemas ditos assíncronos (ou *time free*) não há qualquer tipo de restrições temporais. Para esses modelos foi provado ser impossível obter o consenso na presença de falhas [7, 16]. Isso se deve à dificuldade inerente desse tipo de sistema em distinguir entre um processo falho ou meramente lento.

Pesquisas têm sido desenvolvidas no intuito de se conseguir soluções satisfatórias para tolerância a falhas, mesmo no ambiente assíncrono. Uma motivação para estas pesquisas é o fato de o serviço de melhor-esforço, provido como serviço padrão da Internet, ser caracterizado como um ambiente assíncrono. Uma das formas encontradas foi tentar inserir algum tipo de sincronismo nos sistemas assíncronos, gerando os modelos parcialmente

síncronos. Este sincronismo é, em geral, inserido na forma de estimativas realizadas com relação a algumas ou todas as restrições temporais que possam existir. Estas estimativas não são, entretanto, sempre confiáveis, dependendo de uma certa estabilidade do sistema para serem concretizadas. Em [17, 18] é feita uma análise sobre a possibilidade e eficiência de soluções para o problema de consenso em diferentes modelos parcialmente síncronos. O modelo Assíncrono Temporizado foi proposto em [19]. Esse modelo parcialmente síncrono supõe a existência de um tempo a partir do qual o sistema passa a ter um comportamento próximo ao síncrono, utilizando estimativas de tempo para identificar este momento. Em [20, 21] é proposto a *Base de Computação Temporizada*, que executa em um ambiente síncrono, podendo detectar falhas da aplicação assíncrona e executar algumas ações temporizadas desta. Em [3] Chandra & Toueg propuseram o uso dos detectores de falhas não confiáveis, nos quais estimativas com relação ao tempo de transferência de mensagens são inseridas nestes detectores.

Os modelos de sistemas distribuídos existentes baseiam-se ou nas características síncronas das redes locais (com protocolos de controle específicos) ou em ambientes de melhor esforço como os da Internet. Nenhum dos modelos propostos consideram as novas arquiteturas para prover serviços com qualidade (QoS) como as IntServ e DiffServ propostas pela IETF, orientadas para a Internet [1, 2]. Estas arquiteturas permitem que os sistemas de comunicação possam fornecer serviços com diferentes níveis de qualidade, considerando aspectos tais como os limites fixados para a transferência de dados e diferentes prioridades com relação à possibilidade de se perder pacotes, etc.

O desafio a ser enfrentado é o de criar um ambiente de execução que leve em consideração os diferentes níveis de serviço providos pelas arquiteturas de QoS e seja satisfatório às necessidades das novas aplicações que estão sendo desenvolvidas.

Este artigo apresenta uma proposta inovadora para o tratamento de tolerância a falhas em sistemas distribuídos cujos componentes (ou processos) comunicam-se através dos protocolos com QoS como o IntServ e o DiffServ. Nosso modelo é adaptável às condições dinâmicas que ocorram na QoS provida pelo ambiente de execução, permitindo a exploração plena do melhor dos dois cenários possíveis (síncrono e assíncrono). Mais ainda, nosso modelo é particularmente poderoso no sentido de garantir tolerância a falhas mesmo quando há mudanças na QoS do sistema, permitindo que processos com diferentes visões da QoS possam continuar cooperando e executando suas ações de forma segura (i.e., tolerantes a falha). Para isso, introduzimos o conceito de detectores de falhas isócronos e não isócronos e descrevemos um algoritmo para determinação da qualidade de serviço (QoS) do detector. Baseados no modelo e detector propostos, apresentamos um protocolo de consenso baseado em *quorum* que é capaz de funcionar corretamente mesmo quando diferentes processos têm acesso a detectores com qualidade distintas. O protocolo de consenso apresentado é ótimo no número de falhas toleradas. Ou seja, se todos os detectores distribuídos são isócronos (sistema síncrono), o algoritmo tolera $f = n-1$ falhas do tipo *crash* e no pior caso (ambiente assíncrono com detectores do tipo \hat{S}) o sistema tolera $n/2 - 1$ falhas, onde n é o número de processos envolvidos no consenso.

O restante deste artigo estrutura-se da seguinte forma. Após examinar trabalhos correlatos na seção 2 e discutir as novas arquiteturas de QoS para a Internet na seção 3, apresentamos na seção 4 nossa visão de um modelo de sistema distribuído para QoS. Na seção 5 introduzimos o conceito de detectores de falhas isócronos e não isócronos e descrevemos um algoritmo para determinação da qualidade de serviço (QoS) do detector. Na seção 6 apresentamos o algoritmo de consenso e provamos sua correção. Finalmente, apresentamos as nossas

conclusões e possíveis desdobramentos de nosso trabalho na seção 7.

2. Trabalhos correlatos

Diversas pesquisas têm sido desenvolvidas com objetivo de se obter soluções eficientes para tolerância a falhas nos ambientes de execução existentes. Embora estas soluções sejam ótimas em um ambiente síncrono [13, 6], soluções para o ambiente assíncrono têm sido procuradas a partir da constatação da impossibilidade de consenso em sistemas assíncronos sujeitos a falhas de processos [7]. Algumas destas pesquisas apresentam modelos de sistemas distribuídos propostos com o objetivo de inserir algum nível de sincronismo no ambiente assíncrono que permitam solucionar o problema básico de consenso a partir do qual outros problemas são resolvidos. De uma maneira geral, este sincronismo é caracterizado pela definição de limites temporais estimados em relação a alguns aspectos do sistema.

Em [18] são estudadas soluções considerando a existência de sincronismo nos seguintes aspectos dos sistemas: a transferência de mensagens, o tipo de primitiva de comunicação utilizada, o mecanismo de transmissão e a ordenação de mensagens. São verificadas quais destas restrições são necessárias para que o consenso possa ser obtido, e com que nível de tolerância a falhas.

Dwork et al [17] analisaram a tolerância a falhas de protocolos de consenso considerando diferentes tipos de falhas e alternando a existência de sincronismo na execução de processos e no sistema de comunicação. É apresentada a idéia da existência de um Tempo de Estabilização Global (GST – *Global Stabilization Time*) em todas as execuções, a partir do qual o sistema de comunicação passa a funcionar com um limite definido para a transferência de mensagens.

Chandra & Toueg propuseram, em [3], o uso de detetores de falhas não confiáveis. Estes detetores foram organizados em 8 classes diferentes, baseados na combinação das propriedades *completeness* e *accuracy*. *Completeness* diz respeito à capacidade do detetor detectar falhas, podendo ser *strong*, quando todas as falhas são detectadas por todos os processos corretos, e *weak*, na qual todos os processos que falham serão suspeitos permanentemente por algum processo correto. A propriedade *accuracy* diz respeito à possibilidade de o detetor suspeitar erroneamente de alguma falha. Pode ser *strong*, quando nenhum processo será suspeito erroneamente, *weak*, na qual ao menos um processo não será suspeito erroneamente por nenhum outro, *eventually strong*, existindo um momento na execução, a partir do qual a propriedade passa a ser *strong*, e *eventually weak*, no qual a partir de um certo momento, a propriedade passa a ser *weak*. Entre as classe definidas se destacam a P (*Perfect*), com as propriedades *strong completeness* e *strong accuracy*, S (*strong*), com as propriedades *strong completeness* e *weak accuracy*, e \diamond S (*eventually strong*), com as propriedades *strong completeness* e *eventually weak accuracy*. Chandra & Toueg provaram ser o detetor de falhas \diamond S o mais fraco no qual se pode obter o consenso. Vários autores propuseram soluções para problemas de consenso baseados nesses detectores [22, 9, 23, 24].

Em [19] Cristian & Fetzer propuseram o sistema Assíncrono Temporizado. Neste sistema são definidos relógios locais, vistos pelos processos, que são mantidos com desvios limitados, através de algoritmos de sincronização. Além disso, é considerada a existência de um tempo t , a partir do qual o sistema alcança alguma estabilidade⁴, suficiente para que os seus algoritmos sejam executados. O Modelo Assíncrono Temporizado também propõe o conceito de *fail*

⁴ Semelhante ao tempo definido nas propriedades *eventually strong accuracy* e *eventually weak accuracy* dos detetores de falhas não confiáveis de Chandra & Toueg [3] e também ao GST (*Global stability time*) proposto por Dwork et al em [17].

awareness, no qual o sistema sinaliza os relógios cujo desvio estão acima do limite estabelecido, os processos cujo tempo de execução está acima do definido, e as mensagens cujo tempo de transferência foram superiores ao tempo previsto.

Casimiro e Veríssimo propuseram em [20, 21] a Base de Computação Temporizada (TCB - *Timely Computing Base*). A TCB é composta de módulos executando em cada sítio do sistema distribuído, se comunicando através de canais específicos de comunicação com restrições temporais estabelecidas para transferir mensagens. A TCB executa em um ambiente síncrono, garantindo propriedades de limites máximos na execução de ações, sincronização de relógios e entrega de mensagens. Na TCB executa-se protocolos próprios ao ambiente síncrono. A TCB fornece às aplicações os serviços de execução de ações temporizadas (*timely*), medições de durações e detecção de falhas temporais, entre outros. Possui uma interface com o sistema de aplicação, funcionando como uma interface síncrono/assíncrono. As ações não temporizadas da aplicação são executadas no ambiente assíncrono tradicional. A TCB utiliza conceitos de *fail-awareness* em sua implementação.

Os modelos de sistemas distribuídos existentes baseiam-se ou nas características síncronas das redes locais (com protocolos de controle específicos) ou em ambientes de melhor esforço como os da Internet. Nenhum dos modelos propostos consideram as novas arquiteturas propostas pela IETF, orientadas para a Internet [1, 2], descritas na próxima seção.

3. QoS

Uma melhor qualidade nos serviços de comunicação diz respeito a aspectos como o atraso na transferência de pacotes, o *jitter* (variação do atraso), o número de pacotes perdidos e a disponibilidade dos serviços. Os provedores disponibilizam recursos tais como taxas de transferência de pacotes entre cada dois roteadores em uma rota e *buffers* nestes roteadores intermediários para prover comunicação com QoS. QoS implica então em se negociar rotas com características específicas, e em se gerenciar estas rotas, com relação aos fluxos sendo transferidos, e aos recursos disponíveis na rede.

Neste artigo consideramos as arquiteturas para prover qualidade aos serviços de comunicação propostas pela IETF (*Internet Engineering Task Force*) [1, 2]. Esta Força Tarefa padronizou duas arquiteturas básicas para prover QoS: os Serviços Integrados e os Serviços Diferenciados. Apesar de existirem diversas outras arquiteturas para QoS, elas são em geral orientadas a ambientes específicos (proprietários), enquanto as arquiteturas citadas foram desenvolvidas para a Internet. Estas arquiteturas são baseadas no modelo de referência TCP/IP, rodando no nível de rede, e provendo no nível de transporte uma comunicação fim-a-fim com um determinado nível de serviço. O nível do serviço provido por uma rota depende da negociação efetuada entre a aplicação distribuída e o sistema de comunicação, podendo variar do serviço de melhor-esforço, padrão na Internet, a um serviço isócrono, o qual determina limites para o tempo de transferência das mensagens, o *jitter* e taxa de perdas.

A arquitetura de Serviços Integrados é baseada no conceito de se reservar recursos das redes para fluxos de dados específicos, gerados entre dois processos. A reserva efetuada somada às características do fluxo definem o comportamento do tráfego sendo transferido. Para facilitar a negociação das reservas a serem efetuadas, são definidas algumas classes de serviço. A classe melhor-esforço representa o comportamento padrão da Internet atual, o Serviço Garantido provê uma reserva de recursos que garanta um serviço isócrono e o Serviço de Carga Controlada efetua reservas que propiciem um tráfego confiável, porém sem limites temporais definidos. A arquitetura de Serviços Diferenciados utiliza a idéia de prioridades aplicadas a grupos (agregados) de fluxos de tráfego. Os fluxos são agrupados em classes de serviço, de acordo com suas características básicas, e com o acordo feito com o provedor do

serviço de comunicação. Cada classe de serviço receberá uma prioridade e um tratamento diferenciado. Podem existir diversas classes de serviço, sendo que a IETF padronizou as classes de melhor-esforço, que mantém o serviço padrão da Internet atual, Serviço Expresso como a classe de maior prioridade, provendo um serviço isócrono e Serviço Assegurado que provê um serviço não isócrono com diferentes níveis de confiança

Apesar de utilizarem conceitos diferentes, as duas arquiteturas fornecem uma classe definida para serviços isócronos (Serviço Expresso e Serviço Garantido), uma classe para serviços confiáveis (serviço Assegurado e Serviço de Carga Controlada) e uma para serviço de melhor-esforço.

A QoS de um canal é provida sob contrato negociado entre o cliente e o sistema de comunicação. Esta negociação depende da solicitação do cliente e da disponibilidade de recursos do provedor de comunicação. Este contrato pode também estabelecer situações nas quais o canal venha a perder a QoS previamente negociada. Não é possível, entretanto, obter um serviço com melhor qualidade sem negociar e estabelecer um novo canal. As arquiteturas para prover qualidade aos serviços de comunicação tratam os fluxos de tráfego estabelecidos entre dois processos, estabelecendo a fonte geradora do tráfego e o destino deste. Para estabelecer um canal de comunicação bidirecional entre dois processos é necessário que se negocie a qualidade do serviço a ser provido para os fluxos de tráfego em ambas as direções. Assumimos os canais de comunicação estabelecidos no sistema proposto como permitindo a comunicação em ambas as direções, com uma determinada qualidade de serviço. Na seção a seguir propomos um modelo de sistema distribuído para QoS.

4. Modelo do Sistema

O sistema distribuído é composto por um conjunto $\Pi = \{p_1, p_2, \dots, p_n\}$, de N processos, interligados pelo conjunto $C = \{c_{12}, c_{13}, \dots, c_{1n}, c_{21}, \dots, c_{nn-2}, c_{nn-1}\}$ de canais de comunicação. O tamanho do conjunto C é $((N * (N-1)) / 2)$. O sistema é então representado por um grafo $SD(\Pi, C)$, no qual Π é o conjunto de nós (processos) e C é o conjunto de arestas (canais de comunicação) (Figura 1). SD é um grafo completo. Consideramos que este sistema não pode ser particionado, mantendo o grafo sempre completo.

No nosso sistema consideramos que tanto o sistema de comunicação quanto o sistema operacional que gerencia a execução dos processos possuem diferentes níveis de serviço. O limite Δ , quando conhecido, determina o tempo máximo para a transferência de mensagens através de um canal de comunicação. O limite Φ , se determinado, restringe o tempo que leva para o sistema operacional executar uma ação de um processo. Apesar de os processos possuírem acesso a um relógio local, estes relógios não são sincronizados entre si.

Assumimos a existência de um tempo real, representado pela sequência T de números reais. Um determinado momento no tempo é representado por $t \in T$.

O ambiente de comunicação é formado por canais de comunicação confiáveis⁵, sendo garantida a entrega de mensagens em qualquer classe de serviço do sistema de comunicação. Cada canal é estabelecido a partir de uma rota entre os dois processos que se comunicam, sobre uma rede de computadores. Estes canais podem ser providos com diferentes níveis de serviço, de acordo com as arquiteturas para QoS desenvolvidas e padronizadas pelo IETF para a Internet. Podem ser utilizados os Serviços Integrados e os Serviços Diferenciados.

⁵ Canais confiáveis podem ser implementados usando-se números sequenciais nos identificadores das mensagens para ordenação e retransmissão (ex.: TCP/IP).

O limite de tempo Δ na entrega de cada mensagem durante a comunicação, depende da classe de serviço definida para cada canal de comunicação específico. Se Δ existir, ele será conhecido e informado pelo sistema de comunicação aos processos. Os canais de comunicação com o limite Δ conhecido são isócronos. Os canais de comunicação são identificados pelos processos nas suas extremidades. A função $QoS(px, py)$ definida por $\Pi^2 \rightarrow \{I, Ni\}$, informa a QoS de um canal de comunicação, dada a sua identificação. I representa um canal isócrono e Ni um canal não isócrono.

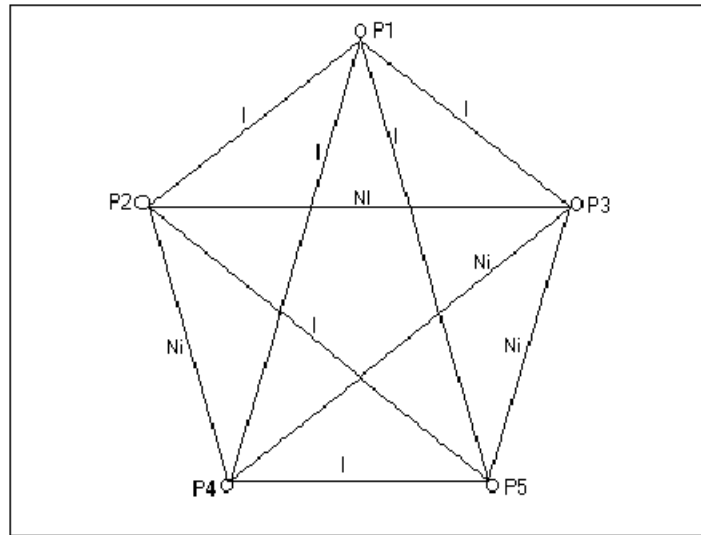


Figura 1: Grafo SD

Assumimos que o sistema distribuído não pode ser particionado. Esta possibilidade será levada em consideração em trabalhos futuros.

Restrições no tempo para executar ações dos processos podem ser obtidas com o uso de um Sistema Operacional de Tempo Real. Com este sistema se estabelecem prioridades na execução de determinadas ações. Estas ações passam a ser executadas com um limite Φ determinado e conhecido. Tanto ações da aplicação como dos protocolos do sistema podem ser executadas com um Φ conhecido. Estas ações possuem um comportamento isócrono.

Assumiremos que os processos falham de forma silenciosa (*fail-silent*), interrompendo a sua execução. Estes processos não irão gerar informações incorretas como resultado de seu processamento. Estas falhas são percebidas através dos canais de comunicação, a partir da demora na entrega das mensagens. Se o canal utilizado para a detecção for isócrono a falha será do tipo *fail-stop*. No caso de o canal de comunicação for não isócrono a falha será do tipo *crash*. Neste primeiro estudo não consideramos falhas bizantinas, nas quais processos com falha continuam a enviar mensagens, prejudicando o funcionamento do sistema como um todo.

Definimos um padrão de falhas como uma função F , que aplicada a um determinado momento do tempo real, $t \in T$, denota o conjunto de processos que falharam até aquele momento ($F(t)$). Como os processos não se recuperam $F(t) \subseteq F(t+1)$.

5. Um detetor de falhas com QoS

O detetor de falhas tem o objetivo de monitorar o sistema, informando aos processos as falhas que possam ter ocorrido. O detetor funciona distribuído em módulos associados a cada processo. Estes módulos trocam informações entre si, através dos canais de comunicação

estabelecidos entre os processos.

Os módulos do detetor de falhas testam os processos, através dos canais de comunicação, com mensagens do tipo “I am alive” ou “*heartbeat*”. O tempo esperado para as respostas dos processos, em cada canal de comunicação, é estimado a partir de informações fornecidas pelo provedor do serviço de comunicação. Este tempo pode ser um limite máximo exato para a transferência, se o nível de serviço provido ao canal for isócrono. O tempo estimado é utilizado na identificação ou suspeita de falhas dos processos. Os módulos do detetor de falhas divulgam suas informações entre si, o que irá permitir que as detecções de falhas sejam difundidas para o restante do sistema.

Os módulos do detetor de falhas também obtêm do sistema de comunicação a identificação de que canais de comunicação são isócronos através da função $QoS(px,py)$. As informações obtidas permitem a utilização dos canais de comunicação isócronos para efetuar detecções de falhas de forma precisa. Com esta diferenciação é possível estabelecer uma qualidade de serviço na detecção de falhas, com suspeitas de falhas, obtidas através de canais de comunicação não isócronos, e detecções confiáveis de falhas, obtidas através dos canais isócronos.

Considerando as detecções diretas realizadas por cada módulo do detetor de falhas, podem ser obtidas as seguintes informações sobre cada processo:

Correct - se receber uma mensagem do processo sem atraso com relação ao limite máximo estabelecido para o canal de comunicação;

Suspect - se não receber mensagem do processo e o canal de comunicação não for isócrono; e

Fail - se não receber mensagem do processo e o canal de comunicação for isócrono.

Cada módulo do detetor de falhas informa ao processo ao qual está ligado o conjunto *Suspect* de processos suspeitos, o conjunto *Correct* de processos corretos e o conjunto *Fail* com os processos cujas falhas foram detectadas. Quanto mais próxima estiver a visão dos conjuntos gerados pelo módulo do detetor de falhas, com relação ao padrão de falhas F , maior será a qualidade do detetor. Como o conjunto *Fail* só contém processos que de fato falharam $Fail \subseteq F(t)$. Considerando então um processo p_i , o conjunto $Suspect_i$, informado pelo módulo do detetor de falhas ligado a este processo, representa os processos que são suspeitos de falha pelo processo p_i , enquanto o conjunto $Fail_i$ representa os processos que p_i tem certeza que falharam.

Além das detecções obtidas de forma direta, os módulos do detetor de falhas podem identificar falhas, mesmo em processos com os quais não possuem canais de comunicação isócronos, a partir das informações enviadas por outros módulos. Desta forma o conjunto *Fail* tende a crescer sem depender da QoS dos canais de comunicação que o processo possui.

No início da execução do sistema, ao estabelecer o detetor de falhas, seus módulos trocam entre si informações a respeito dos canais de comunicação que cada um possui e sua qualidade de serviço. De posse destas informações cada módulo constrói uma representação do grafo SD. Associado a cada aresta é adicionado um campo Qualidade (Q). O campo Qualidade terá o valor I se o canal for isócrono, e Ni caso o canal não seja isócrono. A cada canal de comunicação do próprio processo é adicionado o limite máximo no tempo para transferência, que pode ser um valor estimado ou exato, dependendo da QoS do canal. Com o grafo construído, o módulo executa o algoritmo Detecção de Cobertura⁶ Isócrona (Figura 2). A existência de uma cobertura isócrona determina a QoS do detetor de falhas e por

⁶ Detectamos a cobertura através de uma árvore geradora do grafo. Uma árvore geradora de um grafo G é um sub-grafo de G na forma de árvore que contenha todos os seus vértices.

consequência a classe à qual este detetor pertence. A cobertura isócrona representa a existência de rotas isócronas entre cada dois processos do sistema⁷ e portanto entre cada dois módulos do detetor de falhas. Através da cobertura isócrona pode-se garantir que qualquer falha será detectada por todos os módulos do sistema, não havendo portanto suspeitas incorretas. O tempo para que uma falha seja detectada por cada módulo é também conhecido e limitado. Chamaremos o detetor de falhas com este comportamento de Isócrono. O detetor Isócrono possui, portanto, as características de um detetor de falhas da classe P (Perfeito) de acordo com Chandra & Toueg [3], possuindo as propriedades *strong completeness* (todas as falhas serão detectadas por todos os processos corretos, como *Fail*) e *strong accuracy* (como todas as falhas são detectadas como *fail* por todos os processos corretos, as suspeitas são ignoradas pelos módulos do detetor, não sendo informadas aos processos). Caso não seja possível estabelecer uma cobertura isócrona sobre o grafo das conexões do sistema chamaremos o detetor de Não Isócrono. Assumimos que este detetor de falhas possui as características do detetor de falhas $\diamond S$, com as propriedades *strong completeness* (todas as falhas de processos serão detectadas pelos demais processos corretos, como *Fail* ou como *Suspect*) e *eventually weak accuracy* (a partir de um determinado tempo t existirá um processo que não será suspeito erroneamente por nenhum outro). A classe do detetor de falhas é informada pelo módulo ao processo através do campo Classe (C), que pode assumir os valores I para Isócrono e NI para Não Isócrono.

```

1 Procedure Detecção de Cobertura Isócrona (no-corrente, lista-processos, lista-canais, i)
2
3 begin
4     i = i + 1
5     lista-processos[i] = no-corrente
6     while (i < n  $\wedge$   $\exists$  canal, (Qualidade = I  $\wedge$  no-destino  $\notin$  lista-processos)) do
7         select canal t.q. Qualidade = I  $\wedge$  no-destino  $\notin$  lista-processos
8         lista-canais[i] = canal
9         Detecção de Cobertura isócrona(canal.no-destino, lista-processos, lista-canais, i)
10 end

```

Figura 2: Algoritmo para detectar uma cobertura isócrona

O algoritmo para se identificar a existência de uma cobertura isócrona (figura 2) percorre o grafo SD, através dos seus canais de comunicação isócronos, procurando uma cobertura isócrona na forma de uma árvore. Nesta árvore, o processo que estiver executando o algoritmo será a raiz e iniciará a execução como *no-corrente*. A variável i é iniciada com valor 0. Esta árvore, se existir, é percorrida recursivamente, sendo que cada processo (nó) só é percorrido uma vez. Cada novo processo percorrido é inserido na *lista-processos*, enquanto o canal de comunicação utilizado é inserido na *lista-canais*. Quando a *lista-processos* possuir N componentes o procedimento não fará mais chamadas recursivas, encerrando as chamadas pendentes, apresentando na *lista-canais* as conexões que fazem parte da cobertura isócrona. O comando *while* da linha 7 permite que todos os canais de comunicação de um processo sejam verificados. O *select* da linha 8 seleciona um canal ainda não percorrido, cujo processo destino não tenha sido adicionado à *lista-processos*. Se não houverem mais canais isócronos a serem percorridos e a *lista-processos* não possuir N processos, não foi encontrada uma cobertura isócrona.

A cobertura isócrona pode ser danificada por uma falha de um processo ou pela perda de QoS de um canal de comunicação que faça parte da cobertura (o canal com a QoS alterada deve ser pesquisado em *lista-canais*). Em ambos os casos o módulo do detetor de falhas irá procurar

⁷ Este caminho não será, necessariamente, o caminho mais curto.

por outra cobertura isócrona que possa existir, executando novamente o algoritmo da Figura 2 toda vez que for identificado mudanças no grafo. Se o algoritmo não mais achar uma cobertura no grafo, haverá então uma modificação no comportamento do detetor de falhas, que passará de Isócrono a Não Isócrono. Esta mudança é repassada ao processo através do campo Classe (C). A prova de correção do algoritmo foi omitida por limitações de espaço. Ela pode ser encontrada em [25].

6. Consenso

No problema do consenso deve-se garantir que um grupo de processos decida por um mesmo valor v como resultado de uma determinada computação, dado que cada processo p_i possa ter proposto um valor diferente. Este problema é definido pelas seguintes propriedades:

P1: Validade – se um processo decidir por um valor v , v deve ter sido proposto por algum processo;

P2: Terminação – todos os processos corretos devem decidir por algum valor;

P3: Concordância Uniforme – dois processos não devem decidir de forma diferente.

O algoritmo proposto é baseado nos conceitos de *quorum* e coordenadores circulantes amplamente usados em algoritmos de consenso [3, 26], sendo utilizado com o Detetor de Falhas com QoS quer seja Isócrono ou Não Isócrono. Este algoritmo, apresentado na Figura 3, é dividido em duas tarefas (*tasks*), sendo que a *task 1* realiza rodadas assíncronas com um coordenador rotativo até que o processo decida (*decide = true*). As rodadas são identificadas pela variável *round* e cada processo coordenador pela variável *coord*. As rodadas são executadas da seguinte forma: o processo coordenador envia o seu valor proposto *vpc* para todos os processos executando o *send* da linha 21. Para o coordenador a sua estimativa, *vpc*, é igual ao seu valor proposto, *vp*. Inicialmente o valor proposto de um processo, *vp*, é igual ao seu valor inicial *vi*. Os demais processos esperam pela estimativa do coordenador ou pela informação do detetor de suspeita ou falha do processo coordenador (*wait* da linha 16). Cada processo atualiza a sua estimativa do coordenador *vpc* para o valor recebido v (linha 18) ou para um valor nulo se a mensagem do coordenador não foi recebida (linha 20) Os processos enviam o valor *vpc* para todos os demais (linha 21). Os processos esperam pelas mensagens enviadas por todos os processos corretos (linha 23) se o detetor de falhas for Isócrono ou pelas mensagens enviadas por uma maioria de processos (linha 26), representada pelo valor X , se o detetor de falhas for Não Isócrono. Se todas mensagens recebidas por um processo possuírem o valor proposto pelo coordenador o processo decide por este valor (linhas 27 a 30). O *quorum* representa o conjunto de processos dos quais se espera uma mensagem, determinando o número X . Enquanto o consenso com o detetor Isócrono tolera $N-1$ falhas de processos, o consenso só é obtido com o detetor Não Isócrono se ao menos uma maioria dos processos $((N + 1) / 2)$ não falharem. A *task 2* é executada quando o processo recebe uma mensagem de decisão e interrompe a execução do algoritmo com o processo decidindo pelo mesmo valor.

O exemplo na figura 4 considera o sistema com 5 processos apresentado na figura 1. Apesar de nem todos os canais de comunicação serem isócronos, a existência de uma cobertura isócrona garante ao detetor de falhas um comportamento Isócrono.

Neste exemplo as linhas horizontais representam a execução de cada processo em tempo sequencial. A linha vertical representa o início da 1ª rodada. As setas mostram as mensagens enviadas (e recebidas quando tocam a linha horizontal). O símbolo X sobre a linha horizontal significa a falha do processo. Inicialmente o coordenador da rodada, processo p_1 envia para os demais processos seu valor proposto, v_1 . O processo p_4 falha antes de receber esta mensagem e o processo p_3 logo depois de receber este valor. Os processos p_2 e p_5 , após

receberem a mensagem do processo p1 com o valor v1, enviam mensagem com o mesmo valor. Estas mensagens só não são recebidas pelos processos p4 e p3 que falharam. Como ainda existe uma cobertura isócrona o detetor de falhas continua Isócrono, e cada módulo deste pertencente aos processos corretos irá detectar as falhas como *Fail*. Após receberem as mensagens transmitidas e obterem o conjunto *Fail* do detetor de falhas com todas as falhas ocorridas, os processos p1, p2 e p5 decidem pelo valor v1.

O algoritmo de consenso proposto possui as seguintes características:

- possibilidade de o algoritmo executar tanto com o detetor de falhas Isócrono, quanto com o detetor de falhas Não Isócrono;
- o algoritmo continua a sua execução quando o detetor de falhas perde sua QoS passando de Isócrono para Não Isócrono; e

```

1 Procedure Consenso(vi)
2   Var
3     vp = vi
4     round = 0
5     pid
6     coord
7     decide = false
8   begin
9     task1
10      while not decide do {rodadas coordenadas com quorum}
11        coord = (round mod n) + 1
12        round = round + 1
13        if pid = coord then
14          vpc = vp
15        else
16          wait until received (coord, v)  $\vee$  coord  $\in$  suspecti  $\vee$  coord  $\in$  faili
17          if (coord, v) received then
18            vpc = v
19          else
20            vpc =  $\perp$ 
21          send (round, pid, vpc) to all
22          if Classe = I then
23            wait until received (round, pid, v)  $\vee$  pid  $\in$  faili  $\vee$  CI = N
24          if Classe = NI then
25            X =  $\lfloor n + 1 \rfloor / 2$ 
26            wait until received (round, pid, v) for X process
27          if (vpc  $\neq \perp$ )  $\wedge$  ( $\forall$  received pid, v = vpc) then
28            vd = vpc
29            decide = true
30            send (pid, decide, vd) to all
31          else if ( $\exists$  received pid, v  $\neq \perp$   $\vee$  vpc  $\neq \perp$ )  $\wedge$  ( $\exists$  received pid, v =  $\perp$   $\vee$  vpc =  $\perp$ ) then
32            if vpc  $\neq \perp$  then
33              vp = vpc {Trava o valor estimado do coordendor}
34            else
35              vp = v, v  $\neq \perp$ 
36      task2
37      when receive (pid, decide, v)
38        if not decide then
39          vd = v
40          decide = true
41   end

```

Figura 3: Algoritmo de Consenso

- o algoritmo executa e obtém o consenso mesmo que os módulos do detector de falhas vinculados aos processos possuam, em um mesmo momento, estados diferentes, sendo uns Isócronos, e outros Não Isócronos.

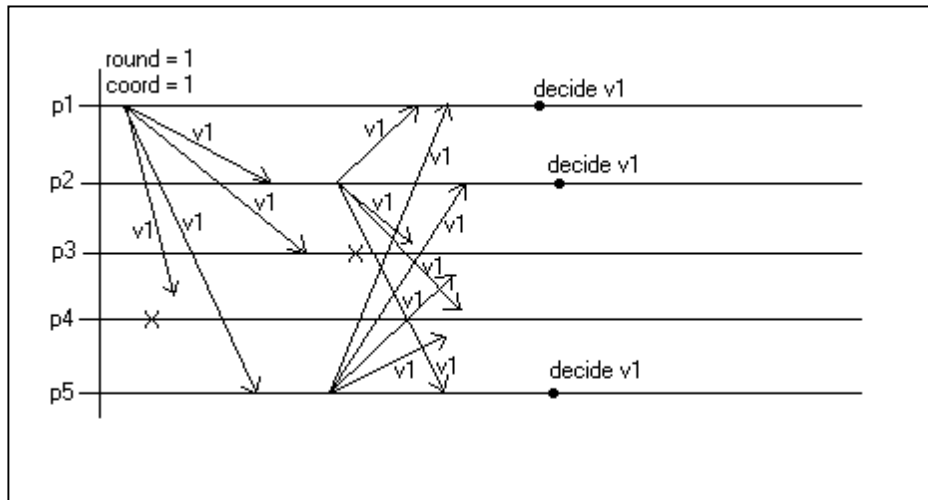


Figura 4: Exemplo de Consenso

Com o detector de falhas funcionando de maneira Isócrona o consenso é obtido no melhor caso com 1 rodada, quando o coordenador desta rodada não falhar. No pior caso, quando cada processo falhar no momento em que for o coordenador de uma rodada, o consenso é obtido com a execução de n rodadas, após a falha consecutiva de $n-1$ processos. Em ambos os casos obtemos uma situação ótima, tolerando $n-1$ falhas [14]. No caso do detector de falhas Não Isócrono não é possível se ter um limite no número de rodadas para obter o consenso. Entretanto, a partir do momento em que o sistema se estabilizar, os processos chegam a uma decisão quando o processo que não é suspeito erroneamente por nenhum outro for o coordenador (isso se verifica pela propriedades do $\Diamond S$). Isto acontecerá, no pior caso, em n rodadas a partir do momento t , da estabilização do sistema. Neste caso se exige que uma maioria de processos estejam corretos, o consenso será concretizado, sendo tolerada falhas em $n/2-1$ destes processos (conforme linhas 25-26). Em cada rodada o número de mensagens trocadas é $O(n^2)$.

Nosso algoritmo é semelhante ao proposto por Mostefaoui e Raynal em [26] que também utiliza o conceito de *quorum*, com soluções tanto para detectores S quanto para $\Diamond S$. Contudo, na solução deles, esses detectores não podem conviver no mesmo consenso. Nossa proposta, ao contrário, permite a co-existência no mesmo consenso de processos com detectores de classes distintas (no caso, P e $\Diamond S$). Além disso, nosso algoritmo se adapta dinamicamente a alterações das classes dos detectores.

6.1. Provas de Correção

6.1.1. Validade

Lemma 1 A cada rodada, o valor proposto por um coordenador será sempre o valor inicial de algum processo.

Prova A prova deste Lemma é feita por indução no número de rodadas.

Caso básico - Na primeira rodada do algoritmo $round = 1$ e o processo coordenador será p_1 . O valor v_p deste processo será o seu próprio valor inicial, v_i , de acordo com a linha 3 do algoritmo, valendo esta definição para todos os demais processos. O valor proposto pelo

coordenador, v_{pc} , será também igual a este valor inicial, de acordo com a linha 14. A princípio cada processo terá o seu valor v_p , igual ao seu próprio valor inicial (linha 3 do algoritmo). Ao final da rodada o processo coordenador manterá o seu próprio valor proposto, v_p , caso não decida, de acordo com a linha 33 do algoritmo. Os demais processos poderão ou não alterar o seu valor proposto, v_p , caso executem a linha 33 do algoritmo. Se executarem esta linha, o valor v_p do processo, será alterado para o valor proposto pelo processo coordenador, v_{pc} . Este último valor é o valor inicial do processo coordenador. Ao final desta rodada cada processo mantém o seu próprio valor inicial como valor proposto, ou assumem o valor inicial do processo p_1 , coordenador da rodada. O lemma vale para esta rodada.

Passo de indução – Vamos assumir que o lemma é válido para uma rodada $r-1$, na qual, ao final, todos os valores v_p dos processos corretos representam o valor inicial de algum processo. O coordenador da rodada r é o processo pc . Sendo assim, o valor proposto pelo coordenador desta rodada, v_{pc} , igual ao valor v_p deste processo, será o valor inicial de algum processo do sistema, de acordo com a suposição do passo de indução do lemma. Ao final desta rodada, não havendo nenhuma decisão, os processos manterão os seus valores da rodada anterior, todos valores iniciais, de acordo com a suposição deste passo do lemma, ou atualizarão o seu valor v_p , para o valor proposto pelo coordenador, v_{pc} , no caso de serem executados os comandos do `if` da linha 31. Este último valor é também um valor inicial, de acordo com a mesma suposição. Desta forma o lemma é provado.

Teorema 1 Se um processo decide por um determinado valor, este valor foi inicialmente proposto por algum processo.

Prova Se um processo decide, a partir do `if` da linha 27 do algoritmo, ele decide pelo valor v_{pc} , recebido do processo coordenador. Este valor é o valor inicial de algum processo, de acordo com o lemma 1. Qualquer decisão efetuada pela execução da *task 2* (linha 36), será, por consequência, por um valor inicial.

6.1.2. Terminação

Precisamos provar que em nenhuma situação o algoritmo fica parado, esperando por alguma mensagem que não irá chegar, e que executando o algoritmo todos os processos corretos venham a decidir. A possibilidade de um processo não decidir está em o processo ficar bloqueado na execução do comando *wait*. Este comando é executado nas linhas 16, 23 e 26 do algoritmo.

Lemma 2 Nenhum processo fica bloqueado eternamente em nenhum trecho do algoritmo.

Prova A prova de que nenhum processo correto fica bloqueado para sempre é feita por contradição. Vamos supor que até a rodada anterior a r nenhum processo decidiu, e que na rodada r um determinado processo, p_i ficou bloqueado. Vamos examinar as possibilidades de bloqueio por linha:

- Na linha 16 um processo só poderá bloquear se ele não for o coordenador da rodada. O processo coordenador não executa esta linha, mas executa, sem possibilidade de bloqueio, o comando *send* da linha 21, a não ser que tenha falhado. Se o *send* foi executado a mensagem será recebida pelo processo p_i , que não irá bloquear. Se o *send* não foi executado, o processo coordenador falhou, e pela propriedade de *strong completeness* do detetor de falhas, a falha do processo coordenador será detectada como *fail* ou *suspect*, dependendo do detetor ser ou não Isócrono, e da QoS do canal de comunicação.
- As linhas 23 e 26 são executadas a depender da classe do detetor de falhas. Se o detetor de falhas for Isócrono, será executada a linha 23. Com este detetor todas as falhas de

processos serão em algum momento detectadas como *fail*, o que é garantido pela propriedade *strong completeness*, e pela característica do detetor Isócrono de só detectar falhas com confiança. No momento em que o conjunto de processos que enviaram mensagens unido ao conjunto *fail* for igual ou maior do que o conjunto de processos, Π , o comando *wait* da linha 23 é encerrado. Caso o detetor perca sua qualidade de serviço, passando a ser Não Isócrono, ele também irá informar ao processo esta alteração, também encerrando o comando *wait*. Desta forma o processo p_i não pode ser bloqueado na linha 23. A linha 26 é executada se o detetor de falhas for Não Isócrono. Neste caso o processo p_i espera por uma maioria de mensagens, que são enviadas por uma maioria de processos corretos, de acordo com a suposição estabelecida para os detetores de falhas da classe $\Diamond S$, à qual pertence o detetor Não Isócrono.

Portanto, nenhum processo fica bloqueado eternamente executando o algoritmo de consenso, contradizendo a suposição efetuada, do bloqueio eterno do processo p_i .

Teorema 2 Todo processo correto irá decidir por algum valor, em algum momento.

Prova Se algum processo tiver decidido, ele enviará a todos os demais a sua decisão por *RMulticast*. Desta forma todos os demais processos corretos receberão a mensagem e também decidirão pelo mesmo valor. Vamos analisar a situação na qual nenhum processo tenha decidido. De acordo com o lemma 2 nenhum processo fica bloqueado pelo algoritmo. A prova segue por contradição, considerando os dois tipos de detetor de falhas. Se o detetor for Isócrono, nenhum processo correto será suspeito erroneamente por nenhum outro, de acordo com a propriedade *strong accuracy*. Na primeira rodada na qual o processo coordenador não falhar, a mensagem deste processo, com o seu valor proposto, será recebida por todos os demais, que divulgarão este mesmo valor. Todos os processos corretos decidirão pelo valor proposto. Se o detetor for Não Isócrono, de acordo com a propriedade *eventually strong accuracy*, haverá um tempo t , a partir do qual existirá um processo, que chamaremos p_i , que não será suspeito erroneamente por nenhum outro. A partir deste momento, na primeira rodada na qual o processo p_i for o coordenador, sua mensagem com o valor proposto chegará a todos os demais processos corretos. Como nenhum valor diferente será divulgado por estes processos, o consenso será obtido para o valor proposto, por todos os processos corretos.

6.1.3. Concordância Uniforme

Teorema 3 Dois processos não decidem de forma diferente.

Prova Assumimos a rodada r como a primeira rodada na qual algum processo decide. Chamaremos este processo de p_i . O processo p_i decide pelo valor v_i . Assumimos também que um processo p_j decide por um valor v_j , na rodada r' , sendo $r' \geq r$.

Se $r' = r$ então os dois processos p_i e p_j decidiram em uma mesma rodada r . A decisão destes processos ocorre a partir do *if* da linha 27 do algoritmo. Os processos receberam um número X de mensagens provenientes de um *quorum* de processos. Para p_i decidir todas as mensagens recebidas continham um mesmo valor $v_i = vpc$, sendo $vpc \neq \perp$ (linha 27). Para p_j decidir todas as mensagens por ele recebidas continham um mesmo valor $v_j = vpc$, sendo $vpc \neq \perp$. O número de mensagens recebidas será relativo a todos os processos corretos naquele instante, caso o detetor seja Isócrono, ou a uma maioria dos processos, caso o detetor seja Não Isócrono. Como o único valor que um processo pode enviar aos demais em uma mesma rodada é o valor proposto pelo coordenador (vpc) ou o valor nulo (linhas 17 a 20), temos que todos os valores recebidos por p_i e por p_j são iguais. Portanto $v_i = vpc = v_j$.

Se $r' > r$ então iremos mostrar que os valores propostos vp de todos os processos que

progrediram para a rodada r' , são iguais ao valor v_i . Devemos considerar a situação na qual o módulo do detetor de falhas do processo p_i na rodada r é Isócrono e a situação na qual este mesmo módulo na rodada r é Não Isócrono.

- Se o detetor de falhas for Isócrono para o processo p_i na rodada r , este processo decidirá pelo valor v_i , ao receber mensagens de todos os processos corretos nesta rodada com o valor v_{pc} , sendo $v_i = v_{pc}$. Como os processos não se recuperam de falhas, o processo coordenador da rodada r' , que chamaremos de p_c' , também participou da rodada r , enviando uma mensagem com o valor $v_{pc} = v_i$ para o processo p_i . Na rodada r o processo p_c' não decidiu e alterou o seu valor proposto, v_p , para o que recebeu do coordenador da rodada, v_{pc} , enviando-o para os demais processos, sendo $v_p = v_{pc} = v_i$ (linha 33). Na rodada r' , o processo p_c' , coordenador da rodada, propôs o seu valor $v_{pc} = v_p = v_i$. Nesta rodada, ao decidir, o processo p_j receberá mensagens enviadas por um *quorum* de processos, todas com o valor proposto pelo coordenador, v_{pc} . O processo p_j decide por este valor, sendo portanto $v_j = v_{pc} = v_i$. O número de mensagens que p_j deve receber depende do estado do módulo do detetor de falhas deste processo na rodada r' .
- O restante desta prova está disponível em [25]

7. Conclusão

Apresentamos um modelo inédito para tolerância a falhas em sistemas distribuídos baseado na utilização de sistemas de comunicação com QoS. Nosso modelo se adapta aos diferentes níveis de serviço de comunicação providos, obtendo um sistema estável com uma alta tolerância a falhas. O modelo utiliza um Detetor de Falhas com QoS, que provê a detecção de falhas com níveis diferentes de qualidade, dependendo da QoS do ambiente de execução. De acordo com esta QoS o detetor pode ser Isócrono ou Não Isócrono, podendo alterar o seu estado durante a execução.

O algoritmo de consenso proposto, baseado no conceito de *quorum*, é adaptável ao comportamento do detetor de falhas. Este algoritmo executa, obtendo o consenso, com o detetor Isócrono ou Não Isócrono. O consenso será obtido mesmo que ocorram alterações no estado do detetor de falhas durante a execução. Com o detetor Isócrono alcançamos uma situação ótima tolerando $f=N-1$ falhas de processo do tipo *crash* e no pior caso, com o detetor Não Isócrono, são toleradas $n/2 - 1$ falhas.

Nos próximos trabalhos estaremos investigando a solução de outros problemas como *group membership* e eleição de líderes e ampliando o modelo para trabalhar com falhas de particionamento da rede e falhas bizantinas de processos.

8. Bibliografia

- [1] Braden, B., Clark, D. e Shenker S., “Integrated Services in the Internet Architecture: an Overview”, RFC 1633, June, 1994.
- [2] Blake, S. et al, “An Architecture for Differentiated Services”, RFC 2475, Dec, 1998.
- [3] Chandra, T. D. and Toueg, S., “Unreliable Failure Detectors for Reliable Distributed Systems”, Journal of the ACM, Vol. 43, No. 2, March, 1996.
- [4] Schmidt, D. C. et al, “Developing Next Generation Distributed Applications with QoS – Enabled DPE Middleware”, IEEE Communications, pp 112-123, Oct., 2000.
- [5] Jalote, P, “Fault Tolerance in Distributed Systems”, Prentice Hall, 1994.
- [6] Lamport, L., Shostak, R., Pease, M. The Byzantine Generals Problem. ACM Trans Program. Lang. Syst. 4, 3 (July/1982), pp. 382-401.
- [7] Fisher, M. J., Lynch, N. A. and Paterson, M. S., “Impossibility of Distributed Consensus with One Faulty Process”, Journal of the ACM, vol 32, No. 2, April, 1985.

- [8] Hurfin, M., Macêdo, R., Raynal, M., Tronel, F. A General Framework to Solve Agreement Problems. Proc. of the IEEE Int. Symp. on Reliable Distributed Systems, SRDS'99, Lausanne. 1999.
- [9] Badache, N., Hurfin, M., Macêdo, R. Solving The Consensus Problem In A Mobile Environment. Proc. of the IEEE International Performance, Computing, and Communications Conference – IPCCC'99, Phoenix/Scottsdale, USA: IEEE Press, 1999. p.29-35.
- [10] Macêdo, Raimundo and Silva, Flávio, Mobile Groups. Anais do XIX Simpósio Brasileiro de Redes de Computadores, Florianópolis-SC. pp.66-81. May,2001.
- [11] Pease, M., Shostak, R. and Lamport, L, “Reaching Agreement in the Presence of Faults”, Journal of the ACM, Vol. 27, no. 2, April, 1980.
- [12] Babaoglu, Ö and Drummond, Rogério. Streets of Byzantium: Network Architectures for Fast Reliable Broadcast. IEEE Trans. on Software Engineering, 11(6), 1985.
- [13] Cristian, F. Reaching Agreement on Processor-group Membership in Synchronous Distributed Systems. Distributed Comp. 4, 175-187, 1991.
- [14] Lynch, N. A., “Distributed Algorithms”, Morgan Kaufmann Publishers, Inc., 1996.
- [15] Charron-Bost, B., Guerraoui, R. and Schiper, A., "Synchronous System and Perfect Failure Detector: solvability and efficiency issues", Proceedings of the International Conference on Dependable System and Networks, 2000.
- [16] Macêdo, R. J. A. Failure Detection in Asynchronous Distributed Systems.Proceedings of the II Workshop on Tests and Fault-Tolerance. Curitiba-PR, Brazil. pp.76-81. July/200
- [17] Dwork, C., Lynch, N. and Stockmeyer, L., “Consensus in the Presence of Partial Synchrony”, Journal of the ACM, Vol. 35, no. 2, April, 1988.
- [18] Dolev, D., Dwork, C. and Stockmeyer, L., “On the Minimal Synchronism Needed for Distributed Consensus”, Journal of the ACM, Vol. 34, No. 1, January, 1987.
- [19] Cristian, F. & Fetzer C., “The Timed Asynchronous Distributed System Model”, IEEE Transactions on Parallel and Distributed Systems Vol. 10, No. 6, June, 1999.
- [20] Casimiro, A. and Veríssimo, P., “Timing Failure Detection with a Timely Computing Base”, Third Euro. Research Seminar on Advances in Dist. Systems, May, 1999.
- [21] Veríssimo, P., Casimiro, A. e Fetzer, C., “The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness”, Proceedings of the International Conference on Dependable Systems and Networks, June, 2000.
- [22] Schiper, A., Early Consensus in an Asynchronous System with a Weak Failure Detector. Distributed Computing, 10:149-157. 1997.
- [23] Greve, F., Hurfin, M., Macêdo, R., Raynal, M. Consensus Based on Strong Failure Detectors : A Time and Message Efficient Protocol. Lecture Notes in Computer Science, v.1800, p.1258-1267, May/2000.
- [24] Hurfin M., Macêdo R., Mostefaoui A., and Raynal M. A Consensus Protocol based on a Weak Failure Detector and a Sliding Round Window. Proceedings of the 20th IEEE Int. Symposium on Reliable Distributed Systems (SRDS'01). New Orleans, USA, October/2001
- [25] Gorender, S. e Macêdo, R. “Tolerância a Falhas em Redes com QoS”, Relatório Técnico RT001/02, LASID/UFBA, Fevereiro, 2002.
- [26] Mostefaoui, A. and Raynal, M., "Solving Consensus Using Chandra-Toueg's Unreliable Failure detectors: a General Quorum-Based Approach", in Proceedings of the 13th International Symposium on Distributed Computing (DISC'99), september, 1999.