Consenso com Recuperação no Modelo *Partitioned Synchronous*

Sérgio Gorender e Raimundo Macêdo

[1]Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação
Universidade Federal da Bahia
Campus de Ondina - Salvador - BA - Brasil

{macedo,gorender}@ufba.br

Abstract. *The partioned synchronous distributed system model has been introduced to take advantage of synchronous partitions of hybrid distributed systems, as such synchronous partitions are implementable in many real scenarios. In this paper we present for the first time a consensus algorithm for proceses that can recover, and its formal proofs, devoted to the partioned synchronous model. The main advantage of the proposed algorithm is that it can tolerate up to n-k process failures in a system with n processes and k synchronous partitions - not all processes need to belong to synchronous partitions. In particular, such robustness is valid even if the majority of processes does not belong to synchronous partitions, which is an advantage in terms of robustness when compared with algorithms for conventional distributed system models.*

Resumo. *O modelo síncrono particionado (partioned synchronous foi introduzido para tirar proveito de partições síncronas em sistemas distribuídos híbridos, uma vez que estas são implementáveis em muitos cenários reais. No presente artigo apresentamos pela primeira vez um algoritmo para consenso com recuperação de processos, e respectivas provas formais, adequado ao modelo síncrono particionado. O algoritmo proposto tem como principal vantagem a capacidade de tolerar n-k defeitos de processos, onde k é o número de partições síncronas e n o tonal de processos no sistema - sendo que podem existir processos que não fazem parte de partições síncronas. Em particular, a robustez do protocolo se aplica mesmo se a maioria dos processos não estiver em partições síncronas, o que representa uma vantagem em termos de robustez quando comparado com soluções para modelos convencionais.*

1. Introdução

Os sistemas distribuídos são compostos por um conjunto de processos residentes em diversos computadores de uma rede de comunicação, onde os processos se comunicam por troca de mensagens. Uma das principais vantagens dos sistemas distribuídos é a possibilidade de se implementar aplicações tolerantes a falhas, por exemplo, através da replicação de processos, garantindo a continuidade do serviço sendo executado mesmo que ocorram defeitos em um determinado número de processos e canais de comunicação. A capacidade de resolver problemas de tolerância a falhas em sistemas distribuídos está intimamente ligada à existência de um modelo de sistema adequado onde se possa demonstrar a possibilidade de solução de

tais problemas. Portanto, há algumas décadas, pesquisadores vêm propondo uma variedade de modelos de resolução de problemas, onde os modelos assíncronos (ou livres de tempo) e síncronos (baseados no tempo) têm dominado o centro das atenções, por serem considerados modelos extremos em termos de resolução de problemas de tolerância a falhas. Por exemplo, o problema de difusão confiável - na presença de canais confiáveis e falhas silenciosas de processos - é solúvel em ambos os modelos [Lynch 1996]. Contudo, o problema de consenso distribuído é solúvel no modelo síncrono, mas não no modelo assíncrono [Fisher et al. 1985].

A impossibilidade relativa aos sistemas assíncronos levou à pesquisa de modelos alternativos, onde o consenso distribuído pode também ser garantido. Um desses modelos mais utilizados é o parcialmente síncrono, que assume que o "comportamento síncrono" se estabelece durante períodos de tempo suficientemente longos para a execução do consenso [Dwork et al. 1988] (tal propriedade é chamada de *Global Stabilization Time*). Num outro modelo, chamado de detectores de defeitos não confiáveis, a propriedade *Global Stabilization Time* é necessária para garantir que os protocolos de consenso baseados no detector de defeitos $\diamond\mathcal{S}$ funcionem de forma adequada [Chandra and Toueg 1996]. Já o modelo assíncrono temporizado depende de períodos de estabilidade síncrona suficientemente longos para prover serviços [Cristian and Fetzer 1999].

Todos esses modelos de sistema acima são caracterizados por configurações homogêneas e estáticas no que toca os aspectos temporais. Ou seja, uma vez definidas as características temporais dos processos e canais de comunicação, essas não se modificam durante a vida do sistema (estática) e todos os processos e canais de comunicação são definidos com as mesmas características temporais (homogêneo). Uma das exceções no aspecto homogêneo é o sistema TCB [Veríssimo and Casimiro 2002], onde um o sistema assíncrono é equipado com componentes síncronas que formam uma *spanning tree* de comunicação síncrona, ou *wormholes*. No entanto, os modelos baseados em *wormholes* são estáticos em relação às mudanças de qualidade de serviços das componentes síncronas.

Para lidar com aspectos dinâmicos e híbridos de modelos de sistemas distribuídos, atendendo às demandas dos novos ambientes com qualidades de serviço variadas, modelos híbridos e dinâmicos foram introduzidos em [Gorender and Macêdo 2002, Macêdo et al. 2005, Macêdo 2007, Gorender et al. 2007, Macêdo and Gorender 2009]. Em [Gorender and Macêdo 2002], foi apresentado um algoritmo de consenso que requer uma *spanning tree* síncrona no sistema distribuído, onde processos são síncronos e canais de comunicação podem ser síncronos ou assíncronos. Nos trabalhos [Macêdo et al. 2005, Gorender et al. 2007], o requisito de *spanning tree* síncrona foi removido e apresentado soluções para o consenso uniforme em ambientes dinâmicos. Em [Macêdo 2007], o modelo foi generalizado para que processos e canais de comunicação pudessem variar entre síncrono e assíncrono e foi apresentado um algoritmo de comunicação em grupo capaz de ligar com ambientes híbridos e dinâmicos em [Macêdo 2007, Macêdo and Freitas 2009], e finalmente, em [Macêdo and Gorender 2008, Macêdo and Gorender 2009] foi introduzido o modelo híbrido e dinâmico *partitioned synchronous* que requer menos garantias temporais do que o modelo síncrono e onde foi provado ser possível a implementação de detectores perfeitos (mecanismo fundamental para a solução de consenso). Vale salientar que a implementação de detectores perfeitos no modelo *partitioned synchronous* não requer a existência de um *wormhole* síncrono [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona [Gorender and Macêdo 2002], onde seria possível implementar ações síncronas globais em

todos os processos, como sincronização interna de relógios. No sistema *partitioned synchronous*, proposto, componentes do ambiente distribuído necessitam ser síncronos, mas os mesmos não precisam estar conectados entre si via canais síncronos, o que torna impossível a execução de ações síncronas distribuídas em todos os processos do sistema. E mesmo que parte dos processos não esteja em qualquer das componentes síncronas, pode-se ainda assim tirar proveito das partições síncronas existentes para melhorar a robustez das aplicações de tolerância a falhas.

Neste artigo, exploramos o modelo *partitioned synchronous* para propor uma solução robusta para o consenso distribuído. Para isso apresentamos e provamos a correção de um algoritmo para consenso com recuperação de defeitos de processos. Nosso algoritmo se baseia no algoritmo Paxos projetado por Leslie Lamport [Lamport 1998] aplicado ao modelo *partitioned synchronous*. Para sua terminação, o algoritmo proposto não depende da propriedade *Global Stabilization Time* característica dos sistemas parcialmente síncronos; portanto, não dependendo de estabilidade do ambiente em questão, desde que o modelo subjacente seja *partitioned synchronous*. O algoritmo proposto tem como principal vantagem a capacidade de tolerar n-k defeitos de processos, onde k é o número de partições síncronas e n o total de processos no sistema - sendo que podem existir processos que não fazem parte de partições síncronas. Em particular, a robustez do protocolo se aplica mesmo se a maioria dos processos não estiver em partições síncronas, o que representa uma vantagem em termos de robustez quando comparado com soluções para modelos convencionais.

O estudo de soluções de tolerância a falhas para o modelo *partitioned synchronous* tem interesse prático uma vez que muitas configurações reais incluem componentes síncronas, como, por exemplo, processos em *clusters* locais que se comunicam com processos clientes através de redes de longa distância (WAN).

O restante deste artigo está estruturado da seguinte forma. Na seção 2 discutimos trabalhos correlatos. Na seção 3 fazemos uma breve apresentação do modelo *partitioned synchronos* introduzido em [Macêdo and Gorender 2009]. Na seção 4 é apresentado o algoritmo de consenso com recuperação e as respectivas provas de correção. Finalmente, na seção 5 apresentamos nossas conclusões.

2. Trabalhos correlatos

Na seção anterior, fizemos um breve relato dos vários modelos de sistemas distribuídos. Para estes modelos, têm sido propostos algoritmos de consenso considerando diferentes modelos de falha.

Em [Aguilera et al. 1998] são apresentados dois algoritmos de consenso com recuperação de defeitos, utilizando detectores de defeitos, sendo que um dos algoritmos utiliza armazenamento estável, e o outro não. Estes algoritmos utilizam detectores de defeitos que, além de suspeitar do defeito de processos, constroem uma estimativa do número de vezes que cada processo falhou, classificando os processos como maus (*bad*) - processos instáveis, que falham e se recuperam com freqüência, ou que falharam permantemente, ou bons (*good*) - processos corretos, que nunca falharam, ou que após terem se recuperado de falha permanecem estáveis. Tanto processos quanto canais de comunicação são assumidos como sendo assíncronos.

Freiling, Lambertz e Cederbaum apresentam em [Freiling et al. 2008] algoritmos de

consenso para o modelo assíncrono, desenvolvidos a partir de algoritmos existentes para o modelo de falhas *crash-stop*.

O algoritmo Paxos, apresentado por Lamport em [Lamport 1998, Lamport 2001], executa sobre um sistema assíncrono dotado de um mecanismo para eleição de líder que apresente a propriedade mínima de que em algum momento de sua execução irá indicar como líder um processo que não irá falhar, o que irá garantir a terminação do protocolo. O Paxos tolera a recuperação de defeitos, desde que uma maioria dos processos esteja correta, para garantir tanto a terminação quanto o acordo uniforme.

Diferente destes algoritmos propostos para o consenso, assumimos um modelo híbrido de sistema distribuídos, no qual existem componentes síncronos e assíncronos.

3. Modelo Spa *(*Partitioned Synchronous)

Um sistema é composto por conjunto $\Pi = \{p_1, p_2, ..., p_n\}$ de processos que estão distribuídos em sítios possivelmente distintos de uma rede de computadores e por um conjunto $\chi = \{c_1, c_2, ..., c_m\}$ de canais de comunicação. Sítios computacionais formam topologias arbitrárias e processos se comunicam por meio de protocolos de transporte fim-a-fim. A comunicação fim-a-fim define canais que podem incluir várias conexões físicas no nível da rede. Portanto, um canal de comunicação c_i conectando processos p_i e p_j define uma relação do tipo "é possível se comunicar" entre p_i e p_j, ao invés de uma conexão ao nível da rede entre as máquinas que hospedam p_i e p_j. Assumimos que o sistema definido por processos e canais de comunicação forma o grafo simples e completo $DS(\Pi, \chi)$ com $(n \times (n-1))/2$ arestas. Particionamento de rede não é considerado em nosso modelo.

Um processo tem acesso a um relógio local com taxa de desvio limitado por ρ. Processos e canais de comunicação podem ser *timely* ou *untimely*. Timely/untimely é equivalente a *synchronous/asynchronous* como apresentado em [Dwork et al. 1988]. Contudo, os modelos parcialmente síncronos considerados em [Dwork et al. 1988] não consideram configurações híbridas onde alguns processos/canais são síncronos e outros assíncronos. Um processo p_i é dito *timely* se existe um limite superior (*upper-bound)* ϕ para a execução de um passo de computação por p_i. De forma análoga, um canal c_i é *timely* se existe um limite superior δ para o atrazo de transmissão de uma mensagem em c_i, e c_i conecta dois processos *timely*. Caso essas condições não se verifiquem, processos e canais são ditos *untimely* e os respectivos limites superiores são finitos, porém arbitrários.

Um canal $c_i = (p_i, p_j)$ implementa transmissão de mensagem em ambas as direções, de p_i para p_j e de p_j para p_i. δ and ϕ são parâmetros do sistema computacional subjacente, fornecidos por mecanismos adequados de sistemas operacionais e redes de tempo real. Também assumimos que os processos em Π sabem a QoS de todos os processos e canais antes da execução da aplicação.

É assumido a existência de um oráculo de *timeliness* definido pela função *QoS* que mapeia processos e canais para valores T ou U (timely ou untimely). Portanto, tal oráculo informa os processos sobre a QoS atual em termos de *timeliness* de processos e canais de comunicação. O oráculo é assumido ser preciso: a execução de *QoS(x)* no instante t retorna T/U se e somente se a QoS do elemento x (processo ou canal) no instante t for timely/untimely. Uma vez que assumimos que a QoS dos processos e canais é estática e conhecida, uma implementação trivial para o oráculo pode ser realizada durante uma fase de inicialização do

sistema: por exemplo, mantendo dois arrays binários, um para processos e outro para canais, cujo valor "1" or "0" representa timely e untimely, respectivamente.

Canais de comunicação são assumidos como confiáveis: não perdem ou alteram mensagens. Processos falham por parada silenciosa, mas podem recuperar-se (crash/recovery). Como em [Aguilera et al. 1998], assumimos que processos podem falhar e se recuperar seguidamente, apresentando um comportamento instavel. Estes processos podem manter este comportamento instável durante o tempo todo, ou a partir de algum momento no tempo se tornar permanentemente em execução, ou em crash. Um processo que não falha durante um intervalo de tempo de interesse, ou que após um tempo de instabilidade não mais falha, é considerado correto.

Sub-grafos Síncronos e Assíncronos

Dado $\Pi' \subseteq \Pi$, $\Pi' \neq \emptyset$ e $\chi' \subseteq \chi$, um sub-grafo de comunicação conectado $C(\Pi', \chi') \subseteq DS(\Pi, \chi)$ é síncrono se $\forall p_i \in \Pi'$ and $\forall c_j \in \chi'$, p_i e c_j são *timely*. Se essas condições não se verificam, $C(\Pi', \chi')$ é dito não síncrono. Utilizamos a notação Cs para denotar um sub-grafo síncrono e Ca para um sub-grafo não síncrono.

Partições Síncronas

Dado $Cs(\Pi', \chi')$, definimos partição síncrona como o maior sub-grafo $Ps(\Pi'', \chi'')$, tal que $Cs \subseteq Ps$. Em outras palavras, DS não contém $Cs'(\Pi''', \chi''') \supset Cs$ com $|\Pi'''| > |\Pi''|$.

Assumimos que existe pelo menos um processo correto em cada partição síncrona [1].

No sistema distribuido Spa, a propriedade de *strong partitioned synchrony* é necessaria para implementar detecção perfeita de defeitos como demonstrado em [Macêdo and Gorender 2009]

strong partitioned synchrony: $(\forall p_i \in \Pi)(\exists Ps \subset DS)(p_i \in Ps)$.

Observamos ainda que o fato de $Ps \subset DS$ exclui dessa especificação sistemas tipicamente síncronos com uma única partição com todos os processos do sistema.

Em *Spa*, mesmo que *strong partitioned synchrony* não possa ser satisfeita, é possível tirar proveito das partições síncronas existentes para implementar detectores parcialmente perfeitos, desde que alguma partição síncrona exista [Macêdo and Gorender 2009]. Definimos essa propriedade a seguir.

weak partitioned synchrony: o conjunto não vazio de processos que pertencem a partições síncronas é um sub-conjunto próprio de Π. Mais precisamente, assumindo que existe pelo menos uma partição síncrona Ps_x: $(\exists p_i \in \Pi)(\forall Ps_x \subset DS)(p_i \notin Ps_x)$.

No que se segue exploramos as propriedades de *strong partitioned synchrony* e *weak partitioned synchrony* sobre Spa para implementar um algoritmo de consenso onde processos podem falhar e se recuperar.

4. Consenso com Recuperação no Modelo *Spa*

Como já visto anteriormente, assumimos o modelo de falhas *crash-recovery*, no qual os processos podem falhar por colapso, e se recuperar, voltando a executar protocolos distribuídos

[1] Observe que essa hipótese é bastante plausível se consideramos *clusters* (partições síncronas) com tamanhos razoáveis - digamos, com mais de três unidades por *cluster*

em andamento (por exemplo, o consenso). Quando um processo se recupera de um defeito, retorna ao estado anterior à ocorrência do defeito, mantendo seus canais de comunicação e seu estado de sincronismo. Desta forma, o processo volta a ser membro de uma partição síncrona, se era membro desta partição antes do defeito, ou retorna como processo assíncrono. O mecanismo de recuperação de defeitos deve utilizar armazenamento estável para recuperar o estado do processo. O armazenamento é utilizado tanto para garantir a recuperação do estado geral do processo, inclusive a determinação de quais protocolos estão sendo executados, como para armazenar informação necessária à participação do processo nestes protocolos, como será visto na seção 4.3. A recuperação de um processo, efetuada por este mecanismo de recuperação, não é instantânea, e como utiliza armazenamento estavel, leva um tempo consideravel se comparada com a execução dos protocolos de detecção de defeitos e do consenso.

4.1. Adaptando os Detectores de Defeitos para uso com Recuperação

O protocolo para detecção de defeitos apresentado em [Macêdo and Gorender 2009] monitora processos membros de partições síncronas que não estejam apresentando falha, ou seja, não tenham sua identificação inserida no conjunto $faulty_i$ para cada processo $p_i \in \Pi$. Este algoritmo implementa um detector P no caso de a propriedade *strong partitioned synchrony* ser satisfeita, e um detector de defeitos xP, caso a propriedade *weak partitioned synchrony* seja satisfeita. Nesta última situação, existem processos que não são membros de partições síncronas e estes processos não são monitorados. O detector xP satisfaz as propriedades *Partially Strong Completeness* (todo processo pertencente a uma partição síncrona que falha será detectado por todo processo correto em um tempo finito) e *Partially Strong Accuracy* (se um processo pertencente a uma partição síncrona for detectado, este processo falhou). Para garantir a propriedade *Partially Strong Completeness*, o intervalo de monitoração é configurado para um valor sempre inferior ao tempo mínimo de recuperação dos processos.

Na figura 1 apresentamos uma versão modificada do algoritmo de detecção de defeitos, para considerar a recuperação de processos. Esta modificação foi inserida na tarefa $T3$ do algoritmo.

Na versão original deste algoritmo, mensagens *are-you-alive* continuam a ser enviadas a todos os processos, periodicamente, provocando a resposta destes através de mensagens *i-am-alive*. A tarefa $T3$ é executada quando uma mensagem *i-am-alive* é recebida, com o objetivo de cancelar o *timeout* definido para a recepção desta mensagem (linha 10). Nesta nova versão, quando a tarefa $T3$ é executada, além do cancelamento do *timeout*, caso a identificação do processo emissor da mensagem faça parte do conjunto $faulty_i$, assume-se que este processo falhou e apresentou uma recuperação desta falha. A identificação deste processo é retirada do conjunto $faulty_i$ (linha 11). Ao ter sua identificação retirada do conjunto $faulty_i$, o processo volta a ser monitorado. Processos que não são componentes de partições síncronas não são monitorados pelo detector xP, e podem falhar e se recuperar, voltando ao seu estado anterior, sem interferir com o detector. A identificação destes processos não é inserida no conjunto $faulty_i$ para os processos p_i.

A identificação de que processos são membros de partição síncrona e de quais não são é feita por um oráculo, como definido em [Macêdo and Gorender 2008, Macêdo and Gorender 2009]. O estado destes processos é definido no início da execução do sistema, permanecendo estável durante sua execução. Processos e canais de comunicação não

Task $T1$: **every** *monitoringInterval* **do**
(1) **for_each** $p_j,\ p_j \neq p_i$ **do**
(2) $timeout_i[p_j] \leftarrow CT_i()$ + $2\delta + \alpha$;
(3) *send* "are-you-alive?" message to p_j
(4) **end**
Task $T2$: **when** $\exists p_j : (p_j \notin faulty_i) \wedge (CT_i() > timeout_i[p_j]))$ **do**
(5) **if** $(QoS(c_{i/j}) = T)$ **then**
(6) $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(7) *send* notification (p_i, p_j) *to every* p_x *such that* $p_x \neq p_i \wedge p_x \neq p_j$
(8) **else** do nothing (wait for a remote notification)
(9) **end_if**
Task $T3$: **when** "I-am-alive" is received from p_j **do**
(10) $timeout_i[p_j] \leftarrow \infty$; /* cancels timeout */
(11) **if** $(p_j \in faulty_i)$ **then**
$faulty_i \leftarrow faulty_i - p_j$;
Task $T4$: **when** notification(p_x, p_j) is received **do**
(12) **if** $p_j \notin faulty_i$ **then**
(13) $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(14) **end_if**
Task $T5$: **when** "are-you-alive?" is received from p_j **do**
(15) *send* "I-am-alive" *to* p_j

Figure 1. Algoritmo do detector de defeitos xP para o processo p_i com recuperação.

alteram seu estado entre síncrono/assíncrono.

Como o número de partições síncronas existentes é estável, digamos k partições, e por definição [Macêdo and Gorender 2008, Macêdo and Gorender 2009], toda partição síncrona possui ao menos um processo que não falha, permanecendo correto, temos no mínimo k processos que não falham durante a sua execução.

4.2. Mecanismo para eleição de líder

Assumimos a existência de um mecanismo para eleição de líder para o modelo *Spa*, baseado no detector de defeitos utilizado (classe P ou xP). Este mecanismo sempre indica como líder um processo pertencente a alguma partição síncrona (caso a propriedade *strong partitioned synchrony* seja válida será qualquer processo). Inicialmente, o mecanismo indica como líder do grupo, para cada processo $p_i \in \Pi$, o processo de menor identificação, que seja membro de alguma partição síncrona, e que não tenha a sua identificação inserida no conjunto $faulty_i$.

Durante a execução do sistema novos processos líderes podem ser indicados pelo mecanismo, à medida em que os líderes atuais falhem. Neste caso, quando o módulo do detector de defeitos de cada processo $p_i \in \Pi$ detectar a falha do líder atual, sendo a identificação deste inserida no conjunto $faulty_i$, um novo processo líder será escolhido para substituir o que falhou. Este novo líder será um processo membro de partição síncrona que não apresente falha, e cuja identificação seja a de menor número, desde que maior do que a identificação do líder a ser substituído. Desta forma, os líderes serão sempre escolhidos em ordem crescente de suas identificações. Este dispositivo evita que processos instáveis possam falhar e se recuperar com freqüência, voltando a liderar o grupo, e gerando instabilidade na execução de protocolos.

No pior caso, como existem por definição ao menos k processos corretos, um para

cada partição síncrona, quando o de menor identificação entre estes for escolhido como líder, será obtida uma estabilidade no sistema, havendo então apenas um líder indicado por todos os processos corretos do sistema.

4.3. O Algoritmo de Consenso no Modelo Spa com Recuperação de Defeitos

Nesta seção apresentamos um algoritmo de consenso que executa sobre o modelo *Spa*, com recuperação. O consenso assume que o modelo *Spa* com uma das propriedades *Weak Partitioned Synchrony* ou *Strong Partitioned Synchrony* como válidas, e a existência de um oráculo que indica quais processos pertencem a partições síncronas e quais não pertencem (utilizado caso existam processo não síncronos). O algoritmo também assume a existência de um detector de defeitos (classe P ou xP, dependendo da propriedades que é satisfeita pelo sistema), que indica para cada processo $p_i \in \Pi$, através do conjunto $faulty_i$, que processos membros de partições síncronas falharam e permanecem com falha. Este algoritmo também utiliza um mecanismo para eleição de líder baseado no detector de defeitos utilizado, o qual é descrito na subseção 4.2. Este protocolo de consenso executa sem modificações com um detector de defeitos de qualquer uma das classes definidas.

A estrutura básica deste algoritmo é baseada no algoritmo Paxos, apresentado por Lamport em [Lamport 1998, Lamport 2001]. O consenso é realizado em 2 fases, identificadas como PREPARE-REQUEST (preparação da rodada) e ACCEPT-REQUEST (rodada para proposição e aceitação de valor).

O algoritmo é dividido em cinco tarefas, e cada processo executa dividido em três agentes: Proposer, Acceptor e Learner. A tarefa $T0$ é executada sempre que um novo processo é indicado como líder do grupo, pelo mecanismo de eleição de líder, e neste caso a tarefa $T1$ passa a ser executada pelo agente Proposer deste processo. As tarefas $T2$ e $T3$ são executadas pelos agentes Acceptor de todos os processos, e são iniciadas pela recepção das mensagens PREPARE-REQUEST e ACCEPT-REQUEST, respectivamente. A tarefa $T4$ é executada pelos agentes Learner dos processos, o tempo todo.

A tarefa $T1$ é executada pelo agente Proposer do processo líder do grupo e coordenador da rodada. Esta tarefa é dividida em duas fases: na primeira fase uma nova rodada é proposta e na segunda fase um valor é proposto para o consenso. Na $Fase1$ o Proposer propõe a nova rodada realizando um *broadcast* da mensagem PREPARE-REQUEST com o número da nova rodada proposta (linha 5 do algoritmo), e esperando por mensagens ACK-PREPARE-REQUEST de todos os processos membros de partições síncronas que não tenham falhado (linha 6). Como já foi definido anteriormente, assumimos como processo correto aquele que não falha, ou que após ter falhado e se recuperado, se mantenha em execução estável. As mensagens ACK-PREPARE-REQUEST apresentam a última rodada na qual cada processo participou (r_j), e qual foi o valor proposto nesta rodada (v_j). Na $Fase2$ um valor é escolhido entre aqueles informados nas mensagens ACK-PREPARE-REQUEST recebidas, sendo relativo à rodada mais recente na qual os processos tenham participado. É realizado um *broadcast* da mensagem ACCEPT-REQUEST, com a rodada em execução e o valor escolhido.

Function Consensus (v_i)

Task $T0$:
(1) $r_i \leftarrow 0$; $lr_i \leftarrow 0$;
(2) Every time $Leader = p_i$
(3) **Task T1** -Proposer;
Task $T1$ - Proposer:
(4) $r_i \leftarrow max(r_i, lr_i) + n$;
——————————Fase 1 da rodada r: PREPARE-REQUEST——————————
(5) *broadcast* PREPARE-REQUEST(r_i);
(6) **wait until** $\big($(ACK-PREPARE-REQUEST(r_i, r_j, v_j) has been received from every correct process which is a member of a synchronous partition) $\vee\ (lr_i > r_i)\big)$;
——————————Fase 2 da rodada r: accept request——————————
(7) **if** $\big((r_i \geq lr_i) \wedge (\neg\exists$(ACK-PREPARE-REQUEST$(r_i, r_j, v_j)$ received from p_j) such that $r_j > r_i)\big)$**then**
(8) if $\exists$ ACK-PREPARE-REQUEST(r_i, r_j, v_j) received from p_j such that $\big(\forall k$: (ACK-PREPARE-REQUEST(r_i, r_k, v_k) have been received from p_k: $r_j \geq r_k) \wedge ((r_k = r_j) \rightarrow (v_k = v_j))\big)$ then
(9) $v_i \leftarrow v_j$;
(10) *broadcast* ACCEPT-REQUEST(r_i, v_i);
Task $T2$ - Acceptor:
(11) **upon** the reception of PREPARE-REQUEST(r_c) from p_c do
(12) **if** $r_c > max(r_i, lr_i)$ **then**
(13) *send* ACK-PREPARE-REQUEST(r_c, r_i, v_i) to p_c;
(14) $lr_i \leftarrow r_c$;
Task $T3$ - Acceptor:
(15) **upon** the reception of ACCEPT-REQUEST(r_c, v_c) from p_c do
(16) **if** $r_c \geq max(lr_i, r_i)$ **then**
(17) $lr_i \leftarrow r_c$; $v_i \leftarrow v_c$; $r_i \leftarrow r_c$
(18) *broadcast* ACK-ACCEPT-REQUEST(r_i, v_i);
Task $T4$ - Learner:
(19) **wait until** ACK-ACCEPT-REQUEST(r_j, v_j) have been received from every correct process p_k which is a member of a synchronous partition, for the same round r_j or a DECISION-MESSAGE (v) is received
(20) *broadcast* DECISION-MESSAGE(v);
(21) *return* (v_j)

Figure 2. Algoritmo de consenso

A tarefa $T2$ é executada pelos agentes Acceptors de todos os processos. Um processo inicia a execução desta tarefa ao receber uma mensagem PREPARE-REQUEST de um Proposer, sendo que a rodada proposta deve ser mais recente do que qualquer rodada na qual o processo esteja participando ou tenha participado (verificado pelo if da linha 12). Nesta tarefa o processo informa o seu valor atual e a última rodada na qual participou, através da mensagem ACK-PREPARE-REQUEST, enviada ao Proposer.

A tarefa $T3$ é executada pelos agentes Acceptors de todos os processos. Um processo inicia a execução desta tarefa ao receber uma mensagem ACCEPT-REQUEST de um Proposer, sendo que a rodada referenciada na mensagem deve ser mais recente do que qualquer outra rodada na qual o processo esteja participando ou tenha participado. O processo atualiza o seu valor proposto e o seu número de rodada, e realiza um broadcast com mensagem ACK-ACCEPT-REQUEST, com o número de rodada atual, e o valor proposto.

A tarefa $T4$ é executada o tempo todo pelos agentes Learner de todos os processos, recebendo as mensagens ACK-ACCEPT-REQUEST enviadas pelos agentes Acceptor dos processos. Quando um Learner receber mensagens para uma mesma rodada de um Quorum de todos os processos membros de partição síncrona que não estejam apresentando falha, ele termina o consenso e indica o valor adotado nesta rodada como o valor acordado.

Em dois momentos da execução do algoritmo se espera por mensagens de um quorum de processos: - na linha 6 o coordenador da rodada espera por mensagens ACK-PREPARE-REQUEST de todos os processos membros de partições síncronas que não estão com falha; - na linha 19 todos os processos executam um wait, no qual esperam por mensagens ACK-ACCEPT-REQUEST de uma mesma rodada, que tenham sido recebidas também por um quorum de processos formado de forma similar ao da linha 6.

Este algoritmo necessita de armazenamento estável, para garantir que um agente Acceptor não execute a tarefa $T4$ para uma rodada, já tendo executado a tarefa $T3$ para uma rodada de número superior. Este dispositivo garante o compromisso dos processos de não participarem de rodadas de número inferior ao número da última rodada na qual já participaram [Lamport 1998, Lamport 2001].

Provas Formais Nas sub-seções que se seguem apresentamos as provas formais para as propriedades do consenso: Terminação, Acordo Uniforme e Validade.

4.3.1. Terminação

O protocolo de consenso apresentado satisfaz a propriedade terminação, sendo a sua prova apresentada no Teorema 1, a seguir.

Theorem 1. *Assumimos a existência do modelo Spa, que o sistema é equipado com um detector de defeitos P ou xP e o mecanismo para eleição de líder descrito na seção 4.2. Todo processo correto decide.*

Proof

A prova é desenvolvida por contradição, assumindo que o consenso nunca é obtido. Isto seria possível se o consenso ficasse bloqueado em algum dos comandos wait executados, nas linhas 6 e 19 do algoritmo, ou se o algoritmo executasse eternamente sendo que a cada rodada iniciada o mecanismo de eleição de líder detectaria a falha do líder atual, o qual é o coordenador da rodada atual, e passaria a indicar um novo processo líder, que iniciaria uma nova rodada, não permitindo o término da rodada atual e a obtenção do consenso.

- Primeiramente verificamos a possibilidade de o algoritmo bloquear no comando wait da linha 6 ou 19.
  - O algoritmo não bloqueia ao executar o comando wait da linha 6, ao esperar por mensagens de todos os processos corretos que são membros de partições síncronas, uma vez que, todo processo membro de partição síncrona que falha será detectado pelo detector de defeitos (propriedade *Partially Strong Completeness* ou *Strong Completeness*, dependendo da classe do detector).
  - O algoritmo não bloqueia ao executar o comando wait da linha 19 devido à mesma argumentação do ítem anterior, aplicada às mensagens de cada rodada, e

considerando que este wait espera por mensagens de diversas rodadas de forma concorrente, até que ocorra uma rodada com decisão.

- Possibilidade de o algoritmo executar eternamente sem obter o consenso:
  - Como existem ao menos k processos corretos, que não irão falhar em nenhum momento, nas k partições síncronas, e como o mecanismo para eleição de líder só escolhe processos membros de partições síncronas como líder, e em ordem crescente, no momento em que este mecanismo escolher um destes k processos como líder, após o início de uma nova rodada por este processo esta rodada será terminada e o consenso será obtido.

Portanto, como o algoritmo não bloqueia, e não há a possibilidade de as rodadas serem sempre executadas sem encerrarem, o consenso termina a sua execução, o que contradiz a suposição inicial da prova.

$\square_{Theorem\ 1}$

4.3.2. Validade

Todos os processos iniciam com algum valor, e apenas valores indicados pelos processos podem ser utilizados para o acordo.

Theorem 2. *Se um processo decide, o faz pelo valor inicial de algum dos processos participantes do consenso.*

A prova deste teorema se baseia fato de que no início os processos só propõem seu valor inicial, e que a cada rodada, os processos propõem seu valor inicial, ou o valor inicial de algum outro processo, adquirido em uma rodada anterior. Esta prova não será apresentada neste trabalho.

4.3.3. Acordo Uniforme

O consenso apresentado na Figura 2 satifaz a propriedade consenso uniforme. A prova para esta propriedade está descrita no Teorema 3, a seguir.

Theorem 3. *Se um processo decide por um valor v, todos os processos que decidem o fazem pelo mesmo valor v.*

Proof A prova deste teorema é desenvolvida por contradição, assumindo que dois processos, p_x e p_y decidem pelos valores v_x e v_y, respectivamente, e que $v_x \neq v_y$.

Vamos considerar duas possibilidades:

1. Os processos p_x e p_y decidem na mesma rodada:
   - Se os processos p_x e p_y decidem na mesma rodada, o fazem pelo valor recebido nas mensagens ACK-ACCEPT-REQUEST, enviadas por um Quorum de processos. O valor encaminhado nestas mensagens é valor v_c, proposto pelo processo p_c da rodada, sendo, portanto o mesmo em todas as mensagens. Portanto, neste caso p_x e p_y decidem pelo mesmo valor, e $v_x = v_y$.
2. Os processos p_x e p_y decidem em rodadas diferentes:

- Assumimos que o processo p_x decide pelo valor v_x na rodada r_x, e que p_y decide pelo valor v_y, na rodada r_y, posterior. p_x decide ao executar a tarefa $T4$ do algoritmo, e receber mensagens ACK-ACCEPT-REQUEST para esta rodada de todos os processos membros de partição síncrona que não falharam (linha 19 do algoritmo). Chamaremos este grupo de processos de q_x. Todos estes processos assumiram como seu valor proposto (v_i) o valor proposto pelo coordenador da rodada na mensagem ACCEPT-REQUEST, registrando a rodada na qual o valor foi recebido (linha 17). O Processo p_y decide pelo valor v_y em uma rodada r_y posterior, ao receber mensagens ACK-ACCEPT-REQUEST com este valor de todos os processos membros de partição síncrona que não apresentem falha, executando a tarefa $T4$. Estas mensagens foram enviadas pelos processos ao receber a mensagem ACCEPT-REQUEST com este mesmo valor, do coordenador da rodada, processo p_{cy} (tarefa $T3$). O coordenador da rodada, processo p_{cy} escolhe o valor proposto entre os recebidos em mensagens ACK-PREPARE-REQUEST de todos os processos membros de partição síncrona que não estão com falha (linhas 6 a 9). Iremos chamar este grupo de processos de q_{cy}. Estas mensagens são enviadas pelos processos ao executar a tarefa $T2$, em resposta à mensagem PREPARE-REQUEST do coordenador. O coordenador escolhe o valor relativo à rodada mais recente. Como temos que no mínimo k processos são corretos e nunca falham (assumido pelo modelo), estes k processos são membros tanto do conjunto q_x quanto do conjunto q_{cy}, portanto $q_x \cap q_{cy} \neq \oslash$. Temos que, qualquer que seja a rodada seguinte à rodada r_x, estes k processos participarão desta rodada, e o valor a ser assumido pelo coordenador desta nova rodada será o mesmo valor assumido por estes processos na rodada r_x, ou seja, v_x. Portanto, $v_y = v_x$.

Consequentemente, em qualquer possibilidade de execução p_x e p_y decidem pelo mesmo valor, e $v_x = v_y$. $\square_{Theorem\ 3}$

5. Conclusão

Apresentamos neste artigo um novo algoritmo de consenso, desenvolvido para executar sobre o modelo Spa, que tolera a recuperação de defeitos de processos. Este consenso executa utilizando um detector de defeitos, que pode ser da classe P, caso a propriedade *Strong Partitioned Synchrony* seja satisfeita, ou da classe xP, caso a propriedade válida seja a *Weak Partitioned Synchrony*. O algoritmo é baseado no Paxos, apresentado por Lamport em [Lamport 1998].

Este algoritmo não apresenta restrição no número de defeitos e de recuperações de defeitos, apresentadas pelos processos, uma vez que tanto a terminação como o acordo são garantidos pela existência de ao menos k processos corretos, um em cada partição síncrona do sistema. Diferente de outros algoritmos de consenso para modelos parcialmente síncronos, não existe restrição imposta pelo consenso no número de processos corretos para que o consenso seja obtido.

O algoritmo proposto utiliza um mecanismo simples para eleição de líder, o qual garante a terminação do consenso, ao garantir que líderes são escolhidos na ordem crescente de seus identificadores. Este mecanismo é baseado no detector de defeitos sendo utilizado.

References

Aguilera, M. K., Chen, W., and Toueg, S. (1998). Failure detection and consensus in the crash-recovery model. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, pages 231–245.

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

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

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

Fisher, M. J., Lynch, N., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Freiling, F. C., Lambertz, C., and Majster-Cederbaum, M. (2008). Easy consensus algorithms for the crash-recovery model. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 507–508, Berlin, Heidelberg. Springer-Verlag.

Gorender, S. and Macêdo, R. J. A. (2002). Um modelo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do Simpósio Brasileiro de Redes de Computadores, SBRC 2002*, pages 277–292.

Gorender, S., Macêdo, R. J. A., and Raynal, M. (2007). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31.

Lamport, L. (1998). The part time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc.

Macêdo, R. J. A. (2007). An integrated group communication infraestructure for hybrid real-time distributed systems. In *9th Workshop on Real-Time Systems*, pages 81–88.

Macêdo, R. J. A. and Freitas, A. (2009). A generic group communication approach for hybrid distributed systems. (5523(4)):102–115.

Macêdo, R. J. A. and Gorender, S. (2008). Detectores perfeitos em sistemas distribuídos não síncronos. In *IX Workshop de Teste e Tolerância a Falhas (WTF 2008), Rio de Janeiro, Brazil*.

Macêdo, R. J. A. and Gorender, S. (2009). Perfect failure detection in the partitioned synchronous distributed system model. In *Proceedings of the The Fourth International Conference on Availability, Reliability and Security (ARES 2009), IEEE CS Press*. To appear in an extended version in Int. Journal of Critical Computer-Based Systems (IJCCBS).

Macêdo, R. J. A., Gorender, S., and Raynal, M. (2005). A qos-based adaptive model for fault-tolerant distributed computing (with an application to consensus). In *Proceedings of IEEE/IFIP Int. Conference on Computer Systems and Networks (DNS05)*, pages 412–421.

Veríssimo, P. and Casimiro, A. (2002). The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930.