Consenso Distribuído Eficiente no Modelo Síncrono Particionado

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

{gorender,macedo}@ufba.br

Abstract. *The Partitioned 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 a consensus algorithm for such a system under the crash-recovery faulty model. The proposed algorithm tolerates up to $n - k$ process crashes, where k is the number of synchronous partitions and n the total number of processes, and terminates in up to $s-k+1$ communication steps, s being the number of processes in synchronous partitions. The consensus protocol utilizes a failure detector and a leader election mechanism built for the Partitioned Synchronous model. The main advantage of the proposed solution, when compared with consensus for partially synchronous models, is the existence of an upper-bound for termination and its better resilience. Besides formal proofs for the consensus, an implementation is briefly presented.*

Resumo. *O modelo síncrono particionado foi introduzido para tirar proveito de partições síncronas em sistemas distribuídos híbridos, uma vez que estes são implementáveis em muitos cenários reais. No presente artigo apresentamos um algoritmo de consenso distribuído para falhas do tipo crash-recovery, executando neste modelo. O algoritmo proposto tolera $n - k$ falhas de processos, onde k é o número de partições síncronas e n o total de processos no sistema, e termina em até $s - k + 1$ rodadas de comunicação, para s processos em partições síncronas. O consenso utiliza detectores de defeitos e um mecanismo para eleição de líderes desenvolvidos para o modelo citado. A vantagem desse algoritmo, quando comparado com soluções para modelos síncronos parciais, é a existência de um limite superior no número de rodadas e sua maior resiliência. Além das provas formais para o consenso, uma implementação é brevemente apresentada.*

1. Introdução

O problema de consenso entre processos distribuídos tem recebido especial atenção da comunidade científica por tratar-se de um mecanismo fundamental para a solução de diversos problemas de tolerância a falhas em sistemas distribuídos. Por exemplo, os problemas de replicação ativa [Schneider 1990] e decisão em transações distribuídas atômicas [Guerraoui 2002] têm em seus cernes o problema de consenso. No primeiro problema, réplicas precisam concordar numa seqüência de requisições para manter consistência de estado; no segundo, agentes transacionais devem decidir unanimemente pelo resultado final da transação: *abort* ou *commit*. Essencialmente, o problema de consenso consiste numa decisão unânime e irrevogável a partir de um conjunto de valores propostos, um por processo participante.

A capacidade de resolver problemas de consenso em sistemas distribuídos está intimamente ligada à existência de um modelo de sistema onde se possa demonstrar a possibilidade da solução. Uma vez provada a computabilidade do problema, é comum se buscar pela solução mais eficiente em determinado modelo. Portanto, há algumas décadas, pesquisadores vêm propondo uma variedade de modelos, onde os modelos assíncronos (ou livres de tempo) e síncronos (baseados no tempo) são considerados modelos extremos em termos de resolução de consenso: 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 fosse solúvel. 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 *GST - Global Stabilization Time*). Num outro modelo, chamado de detectores de defeitos não confiáveis, a propriedade *GST* é necessária para garantir que os protocolos de consenso baseados no detector de defeitos da classe $\diamond\mathcal{S}$ 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]. No algoritmo de consenso PAXOS a terminação depende de condições de estabilidade que concorram para a escolha de um único processo líder [Lamport 2001]. Nesses modelos o consenso é solúvel, mas não existe um limite superior (*upper-bound*) no tempo de execução em termos de número de rodadas de comunicação. Sendo *GST* o tempo em que o sistema estabiliza, se pode apenas provar um limite superior $\beta + GST$, para β conhecido e valores arbitrários de *GST*.

De outro lado, no modelo síncrono com falhas de processos do tipo *crash*, é sabido que f + 1 rodadas são necessárias e suficientes para terminar o consenso, para o máximo de f processos faltosos [Fischer et al. 1981].

Todos esses modelos de sistema acima são caracterizados por configurações homogêneas. Ou seja, 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 o sistema assíncrono é equipado com componentes síncronas que formam uma *spanning tree* de comunicação síncrona. Nesse modelo o consenso é também solúvel, pois a componente síncrona garante as propriedades de um sistema síncrono.

Os sistemas síncronos são mais robustos que os modelos parcialmente síncronos. Nos modelos síncronos, o consenso suporta a falha de crash de até n - 1 processos, enquanto os modelos parcialmente síncronos suportam apenas a minoria de falhas de processos e nenhum limite superior no número de rodadas é possível estabelecer (como vimos acima). No entanto, apesar das vantagens do modelo síncrono, esses não são facilmente implementáveis (às vezes impossível) em ambientes de larga escala como Internet. Foram essas motivações que nos levaram a propor o modelo *síncrono particionado* (*partitioned synchronous*) [Macêdo and Gorender 2008, Macêdo and Gorender 2009]. O modelo *síncrono particionado* requer menos garantias temporais que o modelo síncrono e provamos ser possível a implementação de detectores perfeitos de defeitos em tal modelo. Vale salientar que tal modelo exclui a existência de um *wormhole* síncrono [Veríssimo and Casimiro 2002] ou *spanning tree* síncrona [Gorender and Macêdo 2002]: no modelo *síncrono particionado*

existem componentes síncronas, mas essas são interligadas por canais assíncronos. Existe um interesse prático crescente para o modelo síncrono particionado, uma vez que muitas configurações reais incluem componentes síncronas onde processos em *clusters* locais se comunicam com processos remotos através de redes de longa distância (WAN). Por exemplo, pode-se considerar um sistema de nuvem federada, onde nós da nuvem são interligados via Internet. Em cada cluster da nuvem, se pode assumir um comportamento síncrono, mas a nuvem federada não é um sistema síncrono.

Neste artigo, exploramos o modelo *síncrono particionado* para propor uma solução eficiente para o problema de consenso. Para isso apresentamos e provamos a correção de um algoritmo para consenso tolerante a falhas do tipo crash com recuperação (modelo crash-recovery). Nosso algoritmo foi inspirado no algoritmo PAXOS projetado por Leslie Lamport [Lamport 1998] e adaptado ao modelo *síncrono particionado*. Para sua terminação, o algoritmo proposto não depende da propriedade *GST* característica dos modelos parcialmente síncronos; portanto, não dependendo de estabilidade do ambiente em questão. O algoritmo proposto tem a capacidade de tolerar $n-k$ falhas 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 configurações típicas o valor de n tende a ser bem maior que k. Ou seja, $n-k$ tende a ser maior que $n/2$, o que representa uma vantagem, em termos de resiliência, de nosso protocolo com relação às soluções para sistemas parcialmente síncronos que toleram no máximo a falha da minoria dos processos. Melhor ainda, mostramos que em nossa solução existe um limite superior no número de rodadas para a terminação do consenso. Assumindo que o sistema tem s processos em partições síncronas, o número de rodadas máximo (*upper-bound*) para terminação do consenso é $s-k+1$, o que representa uma vantagem quando comparado com sistemas parcialmente síncronos onde um limite máximo não pode ser definido. Se considerarmos todos os processos em partições síncronas ($n = s$) verificamos que o *upper-bound* de nosso algoritmo se aproxima do *upper-bound* para sistemas sincronos. Por exemplo, para $k = 2$ e um modelo síncrono não trivial ($n > 1$ e $f = n - 2$), os dois limites superiores no número de rodadas são idênticos: $n - 1$.

O consenso proposto neste artigo utiliza detectores de defeitos e um mecanismo para eleição de líderes, desenvolvidos para o modelo citado, os quais são também descritos com suas propriedades. Além das provas formais para o algoritmo de consenso, descrevemos brevemente uma implementação dos algoritmos. O presente artigo é uma versão ampliada e aperfeiçoada de [Gorender and Macêdo 2010]. 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 synchronous* introduzido em [Macêdo and Gorender 2009]. Na seção 6 é apresentado o algoritmo de consenso com recuperação e as respectivas provas de correção. Finalmente, na seção 8 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 [Lynch 1996]. Os trabalhos descritos na seção anterior assumem que processos falham por parada definitiva silenciosa (crash).

Posteriormente, foram introduzidos algoritmos de consenso onde processos falhos podem se recuperar, o que é mais condizente com sistemas reais. 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 permanentemente, 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.

Em [Hurfin et al. 1998] é apresentado um protocolo para o modelo *crash-recovery* onde cada processo é equipado com memória estável e um detector de defeitos. O protocolo tolera perda e duplicação de mensagens.

Em [Oliveira et al. 1997] é apresentado um algoritmo de consenso para o modelo *crash-recovery* que tolera omissões de mensagens (*fair-lossy channels*) e que termina em apenas duas rodadas em execuções onde não há falhas de processos ou suspeitas de falhas.

Esses trabalhos acima citados consideram definições alternativas para o modelo de falhas, tais como processos *green/red* em [Oliveira et al. 1997] ou *bad/good* em [Aguilera et al. 1998]. A razão disso é que, nessas soluções, uma maioria de processos tem que estar operacional para garantir terminação. Como o modelo convencional de parada silenciosa (*crash*) não admite recuperação, essas novas definições foram utilizadas para processos que falham e se recuperam.

Em nosso algoritmo, assumimos a definição tradicional de falha silenciosa, pois o protocolo não depende de processos recuperados para formar maioria que garanta terminação. Essa característica torna nosso protocolo mais simples e eficiente no sentido que dispensa o controle de processos que falham e se recuperam.

O algoritmo PAXOS, apresentado por Lamport em [Lamport 1998, Lamport 2001], executa sobre um sistema assíncrono dotado de um mecanismo para eleição de líder. Para garantir a terminação do consenso, o algoritmo para eleição de líder deve garantir que em algum momento de sua execução seja indicado como líder um único processo correto. O PAXOS tolera a recuperação de processos, desde que uma maioria dos processos esteja correta, para garantir tanto a terminação quanto o acordo uniforme.

Devido ao nosso modelo de sistema, nosso algoritmo não depende das restrições dos modelos síncronos parciais: maioria de processos corretos (ou *good*), tampouco de condições de estabilidade para a escolha de um único líder permanente. Nesse sentido, construímos um mecanismo de eleição de líder adequado ao nosso modelo. No que se segue, faremos uma breve apresentação do modelo *síncrono particionado*, para em seguida apresentar o protocolo de consenso. Uma descrição completa do modelo *síncrono particionado* pode ser encontrada em [Macêdo and Gorender 2009].

3. Modelo Spa *(*Síncrono Particionado)

Um sistema é composto por um 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. 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.

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 atraso 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 a transmissão de mensagens 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 e que a QoS não muda durante a execução.

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*). 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"ou "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*). Processos ditos corretos nunca falham.

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 distribuído Spa, a propriedade de *strong partitioned synchrony* é necessária para implementar detecção perfeita de defeitos como demonstrado em [Macêdo and Gorender 2009]

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

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

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. 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. Adaptando os Detectores de Defeitos para uso com Recuperação

O detector de defeitos apresentado em [Macêdo and Gorender 2009] monitora processos membros de partições síncronas e implementa um detector P (perfeito) caso a propriedade *strong partitioned synchrony* seja satisfeita ou um detector xP, caso a propriedade *weak partitioned synchrony* seja satisfeita. O detector xP satisfaz a propriedade *Strong Accuracy* (nenhum processo correto será detectado erroneamente), e a nova propriedade *Partially Strong Completeness* (todo processo pertencente a uma partição síncrona que falha será detectado por todo processo correto em um tempo finito). O detector de defeitos xP produz detecções confiáveis, não monitorando processos que não façam parte de alguma partição síncrona.

Na Figura 1 apresentamos uma versão modificada do algoritmo de detecção de defeitos, para considerar a recuperação de processos. Assumimos que os processos armazenam o seu estado em meio estável. O mecanismo de recuperação de defeitos utiliza este armazenamento estável para recuperar o estado do processo. O tempo para a recuperação de um processo a partir do seu armazenamento estável é assumido como sendo muito superior ao tempo de uma rodada de comunicação. Esta suposição é necessária para a satisfação da propriedade *completeness* do detector de defeitos, sendo conseqüentemente, necessária à prova da propriedade terminação do consenso (Teorema 3). As provas das propriedades do detector de defeitos são similares às apresentadas em [Macêdo and Gorender 2009], não sendo apresentadas neste artigo.

A recuperação de um processo é detectada pela execução da tarefa $T3$, ao receber uma mensagem *I-am-alive* de um processo em $faulty_i$. Neste caso, a identificação deste processo é retirada do conjunto $faulty_i$ (linha 12). Por não serem monitorados, processos não pertencentes a partições síncronas podem falhar e se recuperar, voltando ao seu estado anterior, sem interferir com o detector. Quando o processo líder falhar e for detectado, o protocolo de eleição de líder apresentado na próxima seção é chamado para executar (linha 8).

5. Mecanismo para eleição de líder

O protocolo para eleição de líder apresentado na Figura 2 é baseado no detector de defeitos utilizado (Figura 1). Este mecanismo indica como líder, o processo de menor identificação que seja membro de alguma partição síncrona e cuja identificação seja maior do que a identificação do líder anterior - e que não tenha a sua identificação inserida no conjunto $faulty_i$. Na primeira execução, com $Leader = 0$ (linha 1), o líder será o processo de menor identificação que satisfaça as restrições da linha 5.

Function Failure Detector
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) **if** $p_j = Leader$ then LeaderElection().Task1
(9) **else** do nothing (wait for a remote notification)
(10) **end_if**
Task $T3$: **when** "I-am-alive" is received from p_j **do**
(11) $timeout_i[p_j] \leftarrow \infty$; /* cancels timeout */
(12) **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**
(13) **if** $p_j \notin faulty_i$ **then**
(14) $faulty_i \leftarrow faulty_i \cup \{p_j\}$;
(15) **end_if**
Task $T5$: **when** "are-you-alive?" is received from p_j **do**
(16) *send* "I-am-alive" *to* p_j

Figura 1. Algoritmo do detector de defeitos com recuperação para um processo p_i

Function Leader Election
Task $T0$:
(1) $Leader \leftarrow 0$
(2) Task 1
(3) **end**
Task $T1$:
(4) Select mínimo j, such that
(5) $p_j \in synchronous\ partition \wedge p_j \notin faulty \wedge p_j > Leader$
(6) $Leader \leftarrow p_j$
(7) **end**

Figura 2. Algoritmo do mecanismo para eleição de líder

Durante a execução do sistema novos processos líderes podem ser indicados pelo mecanismo, à medida 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. Como o novo líder eleito deve ter a identificação maior do que a do líder anterior, os líderes serão sempre escolhidos em ordem crescente de suas identificações. Este dispositivo evita que processos instáveis que falham e se recuperam com freqüência, voltem a liderar o grupo, gerando inconsistência na execução de protocolos.

O protocolo proposto satisfaz as propriedades (*safety*) e (*liveness*), como descrito nos teoremas a seguir.

Teorema 1. *Safety - ao final da execução da eleição, apenas um processo se elegeu como líder, tendo a sua própria identificação atribuída à variável* $Leader$.

Teorema 2. *Liveness - todos os processos que executam o protocolo para eleição de líder terminam a execução deste protocolo, atribuindo a identificação de um processo à variável Leader.*

Devido à falta de espaço e simplicidade dos Teoremas, deixamos as respectivas provas para o leitor.

6. 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 o modelo *Spa*, 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 como descrito na Seção 4, 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 5. Este protocolo de consenso executa sem modificações com um detector de defeitos das classes P ou xP.

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 (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 ao iniciar o consenso, e fica ativa até o fim do consenso. Para cada processo p_i, quando este processo for indicado como líder do grupo pelo mecanismo de eleição de líder, a tarefa $T1$ passa a ser executada pelo agente *Proposer* deste processo (linhas 2 e 3). 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, na recepção de mensagens ACK-ACCEPT-REQUEST e DECISION-MESSAGE.

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, através do envio via *broadcast* da mensagem PREPARE-REQUEST e da espera de mensagens ACK-PREPARE-REQUEST de um quorum de processos, formado por todos os processos membros de partição síncrona que não estejam com falha logo antes do *broadcast*, menos os processos que falham enquanto o *wait* é executado. Na segunda fase um valor é proposto para o consenso, através do envio, também via *broadcast* da mensagem ACCEPT-REQUEST, com um valor escolhido entre os recebidos nas mensagens ACK-PREPARE-REQUEST, relativo à rodada mais recentemente executada. Junto com o valor proposto a mensagem ACCEPT-REQUEST envia também o Quorum de processos a ser considerados para a decisão nesta rodada, formado por processos membros de partição síncrona que não estejam com falha logo antes do *broadcast*.

Function Consensus (v_i)

Task $T0$:
(1) $r_i \leftarrow 0$; $lr_i \leftarrow 0$; $Quorum = \oslash$
(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) $Quorum \leftarrow$ every correct process that is a member of a synchronous partition and does not belong to $faulty$
(6) *broadcast* PREPARE-REQUEST(r_i);
(7) **wait until** $\big($(ACK-PREPARE-REQUEST(r_i, r_j, v_j) has been received from $Quorum$ such that
$Quorum \leftarrow Quorum - (Quorum \cap Faulty)$
——————————Fase 2 da rodada r: accept request——————————
(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) $Quorum \leftarrow$ every correct process that is a member of a synchronous partition and does not belong to $faulty$
(11) *broadcast* ACCEPT-REQUEST$(r_i, v_i, Quorum)$;
Task $T2$ - *Acceptor*:
(12) **upon** the reception of PREPARE-REQUEST(r_c) from p_c do
(13) **if** $r_c > max(r_i, lr_i)$ **then**
(14) *send* ACK-PREPARE-REQUEST(r_c, r_i, v_i) to p_c;
(15) $lr_i \leftarrow r_c$;
Task $T3$ - *Acceptor*:
(16) **upon** the reception of ACCEPT-REQUEST$(r_c, v_c, Quorum_c)$ from p_c do
(17) **if** $r_c \geq max(lr_i, r_i)$ **then**
(18) $lr_i \leftarrow r_c$; $v_i \leftarrow v_c$; $r_i \leftarrow r_c$; $Quorum \leftarrow Quorum_c$
(19) *broadcast* ACK-ACCEPT-REQUEST(r_i, v_i);
Task $T4$ - *Learner*:
(20) **wait until** ACK-ACCEPT-REQUEST(r_j, v_j) have been received from $Quorum$ such that
$Quorum \leftarrow Quorum - (Quorum \cap Faulty)$, for the same round r_j or
a DECISION-MESSAGE (v) is received
(21) *broadcast* DECISION-MESSAGE(v);
(22) *return* (v_j)

Figura 3. Algoritmo de consenso para um processo $p_i \in \Pi$

A tarefa $T2$ é executada pelos agentes Acceptors de todos os processos ao receber uma mensagem PREPARE-REQUEST de um *Proposer*. Se a rodada informada nesta mensagem for superior às rodadas nas quais o processo tenha participado, é encaminhada ao coordenador da rodada uma mensagem ACK-PREPARE-REQUEST com o seu valor atual e a última rodada na qual participou.

A tarefa $T3$ é executada ao receber uma mensagem ACCEPT-REQUEST. Se a rodada referenciada na mensagem for 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* da mensagem ACK-ACCEPT-REQUEST, com o número de rodada atual, e o valor proposto. Na mensagem ACCEPT-REQUEST é também recebido o quorum de processos a ser inicialmente considerado pelo consenso (*Learner*) para a recepção das mensagens ACK-ACCEPT-REQUEST.

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 forem recebidas mensagens para uma mesma rodada dos processos indicados no Quorum informado pelo *Proposer*, menos os processos que falharem durante o *wait* da linha 20, o consenso é terminado, sendo indicado o valor adotado nesta rodada como o valor acordado.

Em suma, todos os n processos de Π participam do consenso, mas apenas os de partição síncrona podem ser escolhidos como líder (tarefa *T1-Proposer*). Observamos ainda que cada partição síncrona possui ao menos um processo correto, o que garante a interseção dos quóruns, não sendo necessário esperar por mensagens de processos membros de partições não síncronas.

Uma vantagem deste consenso em relação ao consenso para modelos parcialmente síncronos é a existência de um limite superior (*upper bound*) no número de rodadas para obter o consenso. Este limite é igual a $s - k + 1$, sendo s o número de processos em partições síncronas, e k o número de partições síncronas. Temos então que se a propriedade *strong partitioned synchrony* for válida, $s = n$, e o limite será $n - k + 1$. Se tivermos 2 partições síncronas, o limite será $n - 2 + 1 = n - 1$. É interessante notar que se a propriedade *weak partitioned synchrony* for válida, teremos $s < n$, e o limite será menor. De forma geral, quanto menos processos tivermos em partições síncronas, e quanto mais partições síncronas existirem, menor será o *upper bound*. Apresentamos no Teorema 5 a prova formal deste limite superior.

6.1. Provas Formais

A seguir apresentamos as provas formais para as propriedades Terminação e Acordo Uniforme. A prova da propriedade Validade é deixada para o leitor por falta de espaço.

Teorema 3. *Assumimos o modelo Spa equipado com um detector de defeitos P ou xP e com o mecanismo para eleição de líder descrito na seção 5. Todo processo correto decide.*

Prova

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 7 e 20 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 7 ou 20.
  - O comando *wait* da linha 7, espera por mensagens ACK-PREPARE-REQUEST em resposta a mensagens PREPARE-REQUEST de um quorum de processos. Este quorum é formado por processos membros de partição síncrona que não apresentavam falha no momento do envio da mensagem ACK-PREPARE-REQUEST, sendo modificado dinamicamente para retirar processos que venham a falhar durante o wait. Desta forma, processos que apresentem falha e não recebam a mensagem PREPARE-REQUEST, não respondendo com ACK-PREPARE-REQUEST, serão detectados (propriedades *Partially Strong Completeness* ou *Strong Completeness*), sendo retirados do quorum, restando apenas processos que não falharam durante a troca de

mensagens da Fase 1 do consenso. O comando wait da linha 7 não é bloqueado ao esperar por mensagens.

  - O comando *wait* da linha 20, espera por mensagens ACK-ACCEPT-REQUEST em resposta a mensagens ACCEPT-REQUEST enviadas pelo *Proposer*, de um quorum de processos construído pelo *Proposer* antes do envio de sua mensagem. Este quorum é tratado de forma similar ao item anterior, com a retirada de todos os processos que apresentem falha. Como apenas um processo é escolhido líder por vez (propriedade *safety* do algoritmo de eleição de líder), executando o *Proposer*, não existe a possibilidade de sobreposição e interferência dos quóruns. Devido à mesma argumentação do item anterior, este *wait* também não bloqueia, pois o quorum será formado apenas por processos que tenham recebido a mensagem ACCEPT-REQUEST e respondido.

- Possibilidade de o algoritmo executar eternamente sem obter o consenso:
  - Como por suposição existem no mínimo k processos corretos no sistema, 1 para cada uma das k partições síncronas, e como o mecanismo para eleição de líder só escolhe processos membros de partições síncronas como líder, e em ordem crescente, e satisfaz a propriedade *safety*, 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.

$\square_{Theorem\ 3}$

Teorema 4. *Se um processo decide por um valor v, todos os processos que decidem o fazem pelo mesmo valor v.*

Prova 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 20 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 18). 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 7 a 11). 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 $v_x = v_y$. $\square_{Theorem\ 4}$

Teorema 5. *O protocolo de consenso apresentado na Figura 3 apresenta um limite máximo no número de rodadas para se obter o consenso, igual a $s - k + 1$, sendo s o número de processos em partições síncronas e k o número de partições síncronas.*

Prova

O consenso é obtido quando um processo correto é escolhido como líder, inicia uma rodada como *Proposer*, a qual é executada até o seu término (propriedade de Terminação do consenso). Por suposição do sistema, existem k processos corretos, 1 para cada partição síncrona. Apenas processos membros de partição síncrona podem ser eleitos líderes e coordenar rodadas, e estes processos são eleitos em ordem crescente monotônica de suas identidades, a partir do processo de menor identidade (algoritmo de eleição de líder). Destes processos (s), apenas $s - k$ podem falhar, pois k destes processos são corretos. Se estes $s - k$ processos tiverem identificação inferior aos k processos corretos, serão escolhidos como líder primeiro. Se cada um destes processos falhar logo após ser escolhido líder, teremos $s - k$ rodadas não terminadas (caracterizando um pior caso de execução do consenso). Como a eleição de líder é feita em ordem crescente monotônica de suas identidades, após a eleição e falha dos $s - k$ processos, um dos k processos corretos será eleito líder, e iniciará uma nova rodada. Como este processo não falha, a rodada terminará e o consenso será obtido (propriedade de Terminação do consenso). Conseqüentemente, para qualquer número de processos em partições síncronas (s), e para qualquer número de partições síncronas (k), no pior caso, o número de rodadas para a obtenção do consenso será $s - k + 1$, caracterizando um limite superior no número de rodadas. $\square_{Theorem\ 5}$

7. Implementação

Os algoritmos foram implementados em *C++*, em um ambiente composto por uma rede local de computadores executando o sistema operacional Linux Debian. Utilizamos o compilador

g++ com a biblioteca **Boost** para *Threads* e sincronização. Foram construídas classes para executar o código básico do processo, assim como classes para executar o detector de defeitos, eleição de líder e o consenso. A classe *canal* representa as informações com relação aos canais de comunicação entre os processos.

Os resultados obtidos em testes preliminares são compatíveis com implementações semelhantes publicadas na literatura, não sendo apresentados aqui apenas por falta de espaço. Os testes iniciais foram feitos em uma rede roteada (utilizando um roteador *Cisco*) com o sistema operacional *Linux/Xenomai* para prover processos síncronos, e uma infraestrutura de comunicação com qualidade de serviço, conforme descrita em ([Gorender et al. 2010]) , para prover canais de comunicação síncronos.

8. Conclusão

Apresentamos neste artigo um novo algoritmo de consenso, desenvolvido para executar sobre o modelo *síncrono particionado*, que tolera o defeito e recuperação de processos (modelo *crash-recovery*). 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 foi inspirado no PAXOS apresentado por Lamport em [Lamport 1998], adaptado para o modelo *síncrono particionado*.

O algoritmo proposto 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 a exigência de uma maioria de processos corretos para que o consenso seja obtido.

O algoritmo proposto utiliza um mecanismo simples para eleição de líder desenvolvido para o modelo *síncrono particionado*, e tolera $n - k$ falhas de processos, onde k é o número de partições síncronas e n o total de processos no sistema. Além disso, o consenso termina em até $s - k + 1$ rodadas de comunicação - para s processos em partições síncronas. A grande vantagem desse algoritmo, quando comparado com soluções para modelos *síncronos parciais*, é a existência de um limite superior no número de rodadas e sua maior resiliência. Apresentamos no artigo provas formais para o consenso. Os algoritmos descritos foram implementados e testados, sendo os dados de desempenho omitidos por falta de espaço. Em trabalho futuro, mostraremos a utilização desse algoritmo num sistema de replicação em nuvens federadas em desenvolvimento no LaSiD/DCC/UFBA.

Referências

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.

Fischer, M. J., Lynch, N. A., and Lynch, N. A. (1981). A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186.

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.

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. and Macêdo, R. J. A. (2010). Consenso com recuperação no modelo partitioned synchronous. In *Anáis do XI Workshop de Testes e Tolerância a Falhas, WTF2010*, pages 3–16.

Gorender, S., Macêdo, R. J. A., and Pacheco Jr., W. L. (2010). Controle de admissão para qos em sistemas distribuídos híbridos, tolerantes a falhas. In *Anáis do XI Workshop de Testes e Tolerância a Falhas, WTF2010*, pages 45–58.

Guerraoui, R. (2002). Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distrib. Comput.*, 15:17–25.

Hurfin, M., Mostéfaoui, A., and Raynal, M. (1998). Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, pages 280–, Washington, DC, USA. IEEE Computer Society.

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. 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).

Oliveira, R., Guerraoui, R., and Schiper, A. (1997). Consensus in the crash-recover model. *Technical Report 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland*.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

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