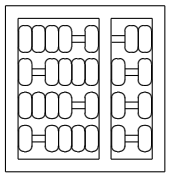

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Difusão Síncrona Totalmente Ordenada de Mensagens em Sistemas Assíncronos Temporizados

Daniel Cason *Luiz E. Buzato*

Technical Report - IC-11-25 - Relatório Técnico

December - 2011 - Dezembro

Difusão Síncrona Totalmente Ordenada de Mensagens em Sistemas Assíncronos Temporizados

Daniel Cason* Luiz Eduardo Buzato*

Resumo

O mecanismo de difusão totalmente ordenada (DTO) de mensagens em sistemas distribuídos assíncronos é fundamental para a construção de aplicações distribuídas tolerantes a falhas. Essencialmente, o mecanismo garante que mensagens enviadas para um conjunto de processos são entregues por todos os processos na mesma ordem total. O problema de difusão totalmente ordenada pode ser reduzido ao problema de consenso distribuído, um outro problema fundamental em algoritmos distribuídos. Mecanismos que implementam soluções para ambos os problemas apresentam diferentes desempenhos e níveis de tolerância a falhas, em função do modelo de computação considerado em seu desenvolvimento. Neste projeto apresentamos um protocolo de difusão síncrona totalmente ordenada de mensagens (DSTO), desenvolvido modularmente e com ênfase em desempenho. O protocolo é direcionado ao ambiente computacional de aglomerados dedicados ao processamento distribuído, cujo comportamento temporal é descrito pelo modelo assíncrono temporizado de computação. A partir de tais premissas, desenvolvemos uma camada de comunicação, que permite que se organize a execução distribuída como uma sequência de etapas síncronas de computação. O protocolo opera sobre esta camada de comunicação e tem seu progresso associado ao comportamento síncrono do sistema, mas garante que a propriedade de ordenação das mensagens seja mantida, mesmo quando processos e canais de comunicação operam assincronamente.

1 Introdução

Um sistema distribuído é um conjunto de processos autônomos, que detêm uma quantidade finita de informação e são interconectados via um conjunto de canais de comunicação. A transferência de informação entre diversos processos se dá exclusivamente através de trocas de mensagens, sendo que esta comunicação pode ser feita de dois modos distintos. Na comunicação ponto-a-ponto (*unicast*), um processo envia uma mensagem para um outro processo específico. Já na comunicação por difusão (*multicast* ou *broadcast*), um processo envia uma mensagem para um subconjunto ou para todos os processos do sistema.

Este trabalho trata do problema de difusão de mensagens com ordenação total em sistemas distribuídos. Nele considera-se um conjunto de processos que podem difundir mensagens na rede e deseja-se que a entrega destas mensagens por parte de cada um dos processos

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro parcial da FAPESP, processo 2010/14555-6

obedeça duas propriedades. A primeira é uma propriedade de progresso e determina que se um processo correto difunde uma mensagem, esta mensagem será entregue em algum momento a todos os demais processos corretos. A segunda é uma propriedade de segurança e define que a ordem da entrega das mensagens em diferentes processos deve ser a mesma. Além disto, considera-se que o protocolo é consistente, ou seja, que os processos só entregam uma mensagem se ela tiver sido enviada por algum processo.

Uma forma simples de resolver este problema é determinar um processo para ser o sequenciador das mensagens, responsável por ordená-las e repassá-las aos demais. Outra abordagem, com uma semântica mais distribuída, consiste em construir um protocolo que permita a cada processo determinar, a partir das mensagens recebidas e do estado dos demais processos, quando é possível entregar uma mensagem recebida para o seu destinatário sem violar a propriedade de ordenação total. Ou seja, determinar se os processos devem ou não aguardar a chegada de uma mensagem m' antes de entregar a mensagem m recebida. Dado que a comunicação seja confiável (mensagens enviadas chegam a todos os destinatários) e falhas não ocorram, os protocolos elaborados consistem na definição de regras para determinar se uma mensagem m recebida é *estável*, ou seja, pode ser entregue ao destinatário (aplicação) sem violar a ordem total.

O problema torna-se mais complicado quando se considera que processos ou canais de comunicação podem falhar, uma vez que o processo sequenciador pode deixar de ser responsivo ou a estabilidade de uma mensagem pode depender do recebimento de outra mensagem, proveniente de um processo falho, que pode não chegar nunca. Neste contexto, o problema de ordenação total de mensagens difundidas se reduz ao problema de obter um *consenso* entre os processos de que mensagens entregar e em que ordem. Por conta desta redução ao consenso, verifica-se que este problema, aparentemente simples, não pode ser resolvido em um sistema assíncrono, mesmo que somente um dos processos participantes possa falhar [10]. Em contraste, ao se considerar o modelo síncrono de computação, dado que o consenso pode ser resolvido [20, 11], a solução deste problema se torna trivial. Uma vez que se tem relógios sincronizados e as ações do sistema tem um tempo máximo para serem realizadas, é possível determinar, a partir de aspectos temporais, a estabilidade de uma mensagem; e a presença de relógios permite que se distingua os processos falhos dos corretos. A questão é que os sistemas reais não especializados não se comportam segundo as suposições do modelo síncrono de computação, pois a complexidade dos seus componentes e a ocorrência de eventos arbitrários fazem com que prazos sejam violados [1, 5]. Assim, algoritmos elaborados com base no estrito cumprimento de prazos não podem ser considerados confiáveis para aplicações reais, que exigem que certas propriedades de segurança sejam sempre garantidas, independentemente do comportamento do sistema subjacente [1].

Por conta disto, surge o nosso interesse pelo modelo assíncrono temporizado de computação [5], que busca se aproveitar das vantagens das duas modelagens extremas quanto à sincronia apresentada pelo sistema. Nele os processos e canais de comunicação são considerados assíncronos, ou seja, não respeitam prazos na realização de ações distribuídas. Porém, ao mesmo tempo, considera-se que os serviços são temporizados, ou seja, que o sistema será, em certas condições de execução, capaz de cumprir etapas de processamento dentro de prazos pré-estabelecidos. Por conta disto, algoritmos elaborados para este modelo não podem, tal como ocorre no modelo síncrono de computação, ter suas propriedades de segurança

diretamente associadas a fatores temporais. Mas, ao contrário do modelo assíncrono de computação, podem empregar tais fatores para construir predicados de progresso.

O restante do trabalho está organizado da seguinte maneira. A próxima seção contém uma especificação do modelo síncrono de computação utilizado neste trabalho e os aspectos práticos na sua implementação em sistemas reais. Na seção 3 apresenta-se uma protocolo de comunicação que implementa, sobre um sistema com comportamento assíncrono temporizado, as abstrações de computação síncrona apresentadas na seção 2. A partir dos serviços fornecidos pelo procotocolo síncrono de comunicação, a seção 4 apresenta um novo protocolo para a difusão de mensagens com ordenação total. A avaliação experimental do seu desempenho e da eficiência obtida pela camada síncrona é realizada na seção 5, que também discute o comportamento temporal observado no ambiente experimental. E, finalmente, a seção 6 discute os resultados obtidos e, a partir deles, as possibilidades de trabalhos futuros.

2 O Modelo Síncrono

O modelo síncrono de computação supõe que o sistema que hospeda um algoritmo distribuído é capaz de cumprir requisitos temporais, ou seja, que tanto os processos que participam da computação distribuída, como os canais de comunicação que os interligam, são capazes de realizar as ações a eles solicitadas dentro de prazos pré-estabelecidos.

Para cumprir tal requisito é necessário que: (i) os processos sejam capazes de receber e processar requisições e eventos dentro de um intervalo de tempo limitado e que (ii) mensagens enviadas cheguem aos seus destinatários com um atraso limitado. Em particular, processos que violem (i) e canais que violem (ii) são considerados falhos, de forma que a primeira suposição do modelo síncrono de computação é [15]:

> Sejam dois processos p e q corretos, interligados por um canal de comunicação também correto. Seja t o instante que o processo p trata um evento (local ou uma recepção de mensagem de algum processo), que o leva ao envio de uma mensagem a q. Então conhece-se uma constante δ tal que q recebe a mensagem enviada por p no máximo no instante $t + \delta$.

A constante δ é um parâmetro do sistema distribuído, um valor previamente conhecido por todos os processos, que é empregado pelos algoritmos distribuídos executados por ele, inclusive como garantia de sua corretude. Assim, no modelo síncrono os processos tem acesso a uma fonte de tempo, que os permite mensurar intervalos entre eventos e agendar ações para instantes específicos do futuro. E como a suposição acima define os prazos em função de instantes t de tempo e da constante δ, a segunda suposição do modelo é [14, 15]:

> Sejam dois processos p e q corretos, e t^p e t^q os instantes de ocorrência de um mesmo evento segundo a fonte de tempo empregada por cada um deles, então conhece-se uma constante ϵ tal que $|t^p - t^q| < \epsilon$.

A constante ϵ também é um parâmetro do sistema e define a precisão entre os relógios locais dos processos. Assim, na primeira suposição, se t foi mensurado no relógio local de p

então q deverá receber a mensagem no máximo no instante $t+\delta+\epsilon$ de seu relógio local [15], o que engloba a possibilidade do relógio de q estar ϵ adiantado em relação ao de p.

De forma geral e com trocas de mensagens por difusão (um para todos), a sincronização entre os relógios locais permite que se defina: (i) um instante t global, no qual os processos iniciam uma etapa de processamento e de eventual envio de mensagens para os demais processos; e (ii) uma duração Δ, também global e grande o suficiente para que todos os processos sejam capazes de receber as mensagens enviadas (em função das constantes δ e ϵ conhecidas). Como resultado, o intervalo $[t, t+\Delta]$ constitui uma etapa global de processamento distribuído, ao final da qual todo processo irá receber uma mensagem proveniente de cada processo não-falho. Ao iterar esta estratégia – definindo-se $t_1 = t_0 + \Delta$ e, de modo geral, $t_{i+1} = t_i + \Delta$ – nota-se que é possível organizar a execução do sistema como uma sequência de *rounds síncronos* de computação distribuída.

Um round síncrono é o período de tempo em que ocorre uma etapa de processamento e de comunicação dentro da execução distribuída. Os rounds são sequenciais e identificados por inteiros subsequentes. Cada processo envia no máximo uma mensagem por round síncrono e as mensagens enviadas possuem o identificador do round a que pertecem [20, 15].

Do ponto de vista de um processo p qualquer, o i-ésimo round síncrono é o intervalo $[t_i, t_i + \Delta]$ do seu relógio local no qual ele recebe as mensagens do round i. Ao final deste prazo, p irá processar as mensagens, os eventos e as requisições recebidas durante o round i e gerar a mensagem a ser enviada no início do round $i+1$. O envio desta mensagem consiste na primeira fase do $i+1$-ésimo round síncrono e se dará, de forma geral, no instante $t_i + \Delta'$ do relógio de p, com $\Delta' \geq \Delta$. O atraso de processamento $\Delta' - \Delta$ em muitos casos pode ser considerado negligível ou ser considerado como a terceira e última fase do round i. Ou então tal atraso pode ser especificado e configurar um "período de silêncio" na execução, uma vez que, durante ele, o processo não envia nem recebe mensagens.

A estratégia de rounds síncronos consiste, então, em organizar a computação distribuída em uma sequência de rounds subsequentes, tal que uma mensagem do round i enviada por um processo correto p seja recebida por qualquer processo q, conectado a p por um canal de comunicação correto, no decorrer do mesmo round i [20]. Esta estratégia propicia uma plataforma computacional muito poderosa na elaboração de sistemas distribuídos, uma vez que torna possível a todos processos corretos terem, ao final de cada round (ou seja, periodicamente), uma visão global, consistente e sincronizada do sistema como um todo.

2.1 Aspectos Práticos da Implementação de Sistemas Síncronos

Para que seja possível implementar a estratégia de rounds síncronos em um sistema distribuído qualquer, é necessário que ele cumpra uma série de requisitos temporais, que podem ser sintetizados pelas seguintes propriedades: (i) os processos devem ser síncronos; (ii) os canais de comunicação devem ser síncronos e confiáveis; e (iii) deve existir um procedimento capaz de ϵ-sincronizar os relógios dos processos.

Pelos processos serem síncronos entende-se que eles são capazes de realizar tarefas com atrasos de processamento similares, ou seja, que as velocidades relativas dos diferentes processos pertencentes ao sistema são limitadas superiormente. Esta propriedade engloba também a capacidade dos processos em agendar ações futuras, de forma que que elas sejam

efetivamente realizadas no instante de tempo agendado.

Por um lado, na prática, como sistemas reais são formados por várias camadas de software e hardware e por várias tarefas que compartilham recursos, o cumprimento *preciso* de prazos só é obtido quando se emprega sistemas de tempo real, que contam com software e hardware especializado a fim de garantir, através de restrições e balanceamento da carga do sistema, propriedades temporais estritas no escalonamento de tarefas e no acesso aos recursos. Por outro lado, a observação empírica de sistemas não especializados, mostra que os processos são capazes de cumprir prazos na maior parte do tempo [5], desde que estejam dedicados ao processamento distribuído e operem sob carga controlada. De forma que é possível computar prazos que serão cumpridos pelos processos do sistema com alta probabilidade, tão alta quanto for necessário para garantir seu funcionamento síncrono.

Por canais de comunicação síncronos entende-se que a latência ponto a ponto no envio de mensagens é conhecida e que suas variações são limitadas. Esta latência é formada pelo acúmulo de atrasos ocorridos nas camadas de rede dos processos comunicantes, no hardware de rede e pelos *switches* que conectam as máquinas. Todos estes dispositivos e camadas operam no modelo *store-forward* – armazenam as mensagens em *buffers* antes de passá-las adiante – e tanto a largura de banda, como os *buffers* são finitos. Como consequência, quando a carga imposta ao sistema se aproxima da sua capacidade de transmissão, observa-se um aumento nos valores médios e na variância da latência de envio de mensagens e, principalmente em casos extremos, o aumento das perdas de mensagens [5] – descartadas pelas camadas de rede intermediárias, por falta de espaço em seus *buffers*.

É importante ressaltar que as perdas de mensagens são eventos fatais para sistemas síncronos, pois a perda de uma mensagem qualquer é indistinguível, do ponto de vista dos demais processos, da falha do seu emissor – que não teria conseguido cumprir com os requisitos temporais do sistema e seria, desta forma, falho. Assim, o modelo síncrono de comunicação supõe a existência de um serviço confiável de entrega de mensagens, com mecanismos capazes de detectar perdas (mensagens de confirmação, temporizadores etc.) e de remediá-las, através do reenvio das mensagens pendentes. Como consequência, o tempo máximo de transmissão deve considerar, além da latência máxima da rede, o custo extra destes mecanismos e a possibilidade que uma mensagem tenha que ser retransmitida.

Assim, da mesma forma que os processos, o tempo máximo para a execução de uma tarefa pelos canais deve levar em consideração uma série de atrasos, que são particularmente sensíveis à carga imposta ao sistema. Mas, desde que a carga seja controlada, os canais se comportam de forma temporizada – latências estáveis e poucas perdas –, o que torna possível estabelecer prazos que serão cumpridos com alta probabilidade pelo sistema.

2.2 Sincronização de Relógios

O último requisito do modelo síncrono é a existência de fontes de tempo sincronizadas, acessíveis por parte dos processos. Estes as empregam, por exemplo, para concordar nos instantes de início e fim dos rounds síncronos. A priopriedade que deve ser garantida é que o desvio nas leituras de tempo efetuadas por diferentes processos deve ser limitado superiormente por uma constante ϵ previamente conhecida.

Dentre as fontes de tempo que podem ser acessadas pelos processos, a mais simples

encontra-se nos próprios processadores das máquinas, que mantém uma contagem de tempo baseada na sua frequência de funcionamento. O acesso a este contador se dá por uma chamada bastante simples, que aplica a este valor um deslocamento ajustável, para que ele represente um instante do tempo físico, atuando como uma espécie de "relógio de parede". A sua precisão é da ordem de milissegundos, mas a sua *taxa de progressão* depende de vários fatores (entre eles carga e temperatura) e é bastante variável. Como resultado, mesmo que estes relógios sejam perfeitamente sincronizados em algum instante inicial, as suas diferentes razões de incremento em relação ao tempo físico, farão com que eles divirjam progressivamente um dos outros, aferindo valores diferentes para o mesmo instante de tempo.

Este fenômeno não é exclusivo dos relógios dos processadores: por menor que seja, há sempre um desvio na taxa de progressão de diferentes relógios. Em termos de sincronização, emprega-se uma constante ρ para determinar a razão máxima entre as taxas de progressão de diferentes relógios. Esta suposição comporta a propriedade de que um intervalo Δ' medido no relógio local de qualquer processo terá uma duração Δ no tempo físico (ou segundo uma fonte de tempo de referência), limitada por $(1-\rho)\Delta \leq \Delta' \leq (1+\rho)\Delta$ [14, 4].

Uma consequência direta desta relação é que, dado que dois relógios estão, no início do processamento, perfeitamente sincronizados, por menor que seja o desvio ρ entre eles, sempre haverá um intervalo Δ grande o suficiente tal que $\rho\Delta > \epsilon$. E assim, a partir do final deste intervalo Δ, a ϵ-sincronização dos relógios não será mais válida, o que faz com que um algoritmo de sincronização dos relógios seja sempre necessário.

Diversos algoritmos foram propostos nas últimas décadas para realizar a sincronização de relógios em sistemas distribuídos, voltados para diversos modelos de sincronia e de falhas, empregando abordagens determinísticas ou probabilísticas. A sua análise está fora dos objetivos deste trabalho, mas de modo geral, os processos realizam a cada τ unidades de tempo uma rodada do protocolo de sincronização [13]. O *período* τ é determinado de forma que a ϵ-sincronização seja válida durante ele, não obstante os desvios existentes entre os relógios locais dos processos. Já o valor da constante ϵ obtida por tais procedimentos é limitado inferiormente, num sistema com n processos, por $\xi(1-1/n)$, onde a constante ξ determina a diferença entre as maiores e as menores latências de comunicação no sistema [19].

Em termos práticos, mesmo em sistemas de tempo real, normalmente não se emprega somente os relógios dos processadores, visto que o ρ a eles associado é alto e de difícil determinação. Uma opção são os relógios implementados pelo hardware dos processos – baseados nas vibrações de cristais de quartzo e independentes da carga do sistema – com ρ na ordem de 10^{-5} a 10^{-7} [12] ou fontes externas de tempo, também com ρ reduzido. A desvantagem destes relógios são seus custos de consulta, que não são desprezáveis. Assim, normalmente se emprega relógios de hardware para periodicamente calibrar os relógios dos processadores e também durante os procedimentos de sincronização global dos relógios.

Um último aspecto a ser abordado é a *granularidade* na sincronização dos relógios. As fontes de tempo empregadas pelos processos representam a passagem do tempo físico através da geração periódica de sinais, interpretados como os "tiques" de um relógio digital. A precisão e a granularidade obtidas por tais relógios estão associadas à taxa de discretização empregada por cada um deles, ou seja, à frequência da fonte geradora dos sinais. Da mesma forma, a ϵ-sincronização dos relógios locais permite a obtenção de uma *base de tempo global aproximada* [12, 13], que fornece aos processos a abstração da existência de um relógio

global. O i-ésimo tique deste relógio é gerado de forma independente por todos processos no instante t_i de seus relógios locais, de forma que o tique i corresponde a um intervalo do tempo físico que contém o instante t_i e tem duração máxima (ou seja, precisão) de ϵ unidades de tempo. Já a granularidade deste relógio global deve ser computada de forma a garantir que os intervalos do tempo físico nos quais dois tiques consecutivos podem ocorrer sejam disjuntos. Ou seja, se dois processos corretos p e q geram os ticks i e $i+1$, respectivamente, nos instantes t_i^p e t_{i+1}^q do tempo físico, então $t_{i+1}^q - t_i^p > \epsilon$.

Assim, de forma geral, a sincronização dos processos pode ser obtida em três níveis, com complexidade de implementação e exigências de sincronia distintas. O primeiro nível de sincronização requer que os relógios locais de cada processo possuam um desvio nas suas taxas de progressão limitado por uma constante ρ. Através desta suposição é possível aos processos medir períodos Δ de tempo com erros limitados, possibilitando o uso de temporizadores para aferir prazos na realização de uma tarefa. O segundo nível de sincronização depende da existência de um protocolo para sincronização dos relógios locais, periodicamente executado pelos processos a fim de que eles, a menos de um erro limitado por uma constante ϵ, sejam capazes de concordar no instante de ocorrência de um evento. Este nível requer que também os canais de comunicação tenham um comportamento estável, visto que ϵ está limitado inferiormente pela variância ξ nas latências de rede. Na literatura, este nível é referido como sincronização *interna* de relógios [13, 4]. O terceiro nível de sincronização – referido na literatura como como sincronização *externa* de relógios [13, 4] – emprega as propriedades obtidas pelo segundo nível para implementar a abstração de um relógio global aproximado, com propriedades de precisão e granularidade análogas às de um relógio local. E isto permite aos processos o máximo de sincronia: ordenar a ocorrência dos eventos e sincronizar a realização de ações com base em uma fonte de tempo compartilhada.

3 Camada Síncrona de Transporte

A seção 2 mostrou como no modelo síncrono de computação é possível organizar a execução distribuída em uma sequência de etapas de processamento e comunicação, chamadas de *rounds síncronos*. Nesta seção apresenta-se uma camada síncrona de transporte de mensagens, isto é, uma camada que implementa o modelo de rounds síncronos sobre um sistema distribuído que adere a um modelo de computação menos estrito de sincronia: o modelo assíncrono temporizado.

A escolha deste modelo se deve à sua capacidade de representar de forma mais fidedigna o comportamento de sistemas distribuídos *off-the-shelf* reais. Estes, como descrito na seção 2, apresentam comportamento temporal síncrono durante a maior parte do tempo. Porém, em certas circunstâncias—normalmente aleatórias e imprevisíveis—seus componentes passam a se comportar de maneira assíncrona: mensagens são perdidas ou tem atrasos muito altos e processos demoram muito mais que o esperado para executar seus passos de processamento. Ou seja, a menos que se estipulem prazos máximos extremamente elevados—o que tornaria diminuta a probabilidade de sua violação por parte do sistema, mas que, em contrapartida, também reduziria seu desempenho—os sistemas reais não são síncronos. E foi com base nestas observações e no estudo do comportamento temporal dos sistemas reais, que Cristian

e Fetzer [5] propuseram o modelo assíncrono temporizado.

3.1 O Modelo Assíncrono Temporizado

Neste modelo, assim como no modelo síncrono, os serviços são temporizados e estipula-se prazos para que processos e canais de comunicação realizem ações distribuídas. Porém, ao contrário do modelo síncrono, tais prazos não são requisitos para a operação do sistema e não devem necessariamente ser cumpridos durante toda execução distribuída. Assim, enquanto no modelo síncrono a violação de um prazo configura uma falha e o número de falhas toleradas é limitado, no modelo assíncrono temporizado as violações de prazos configuram *falhas de desempenho* e não há limites para a frequência com que elas ocorrem [5].

Esta caracterização do sistema, que considera a inexistência de limitantes temporais no comportamento de processos e canais, é uma suposição do modelo assíncrono de computação [10]. Porém, ao contrário deste (chamado pelos autores de modelo *time-free* [5]), o modelo assíncrono temporizado supõe que os processos tem acesso a relógios. Mas, uma vez que os processos são assíncronos, os desvios entre os relógios de seus processadores tendem a ser arbitrários e ilimitados [3]. De forma que se supõe que os processos tem acesso a relógios de hardware, com desvios na suas taxas de progressão limitados por uma constante ρ, o que configura o primeiro nível de sincronização (vide seção 2) e permite aos processos aferir intervalos de tempo e, assim, empregar temporizadores para aferir prazos.

Uma particularidade deste modelo computacional é relacionada ao seu modelo de comunicação: ao contrário do modelo síncrono e da maioria dos modelos assíncronos, considera-se que os canais não são confiáveis e perdem mensagens [5]. Assim como no caso de se ter um atraso de transmissão acima do máximo estipulado, a perda de uma mensagem constitui também uma falha de desempenho – e, a priori, estas duas situações são indistinguíveis.

O modelo assíncrono temporizado prevê a ocorrência tanto de falhas de desempenho, como de falhas por parada [5]. A diferença entre as duas é que falhas de desempenho são reversíveis, ou seja, o processo falho volta a participar da computação e a enviar mensagens após algum tempo, o que não ocorre no caso de falhas por parada, que são definitivas. A priori, os processos não são capazes de distinguir estas duas classes de falhas, uma vez que a consequência imediata de sua ocorrência será a mesma: *a não finalização de uma ação dentro do prazo previsto.* De forma que o mecanismo para tolerá-las será o mesmo e consiste no uso de temporizadores para aferir o cumprimento dos prazos estipulados.

3.2 O Protocolo Síncrono

A camada síncrona de transporte de mensagens implementa um protocolo síncrono que tem sua execução organizada como uma sequência de rounds síncronos, identificados unicamente por valores i, comparáveis entre si e totalmente ordenados. Cada processo da camada envia uma única mensagem por *round* e as mensagens carregam o identificador do round e do processo que a enviou. A comunicação é realizada através de UDP/IP (que fornece um serviço de transporte não confiável baseado em datagramas) e as mensagens são enviadas via *multicast* para todos os demais processos do sistema.

Os processos são reativos, isto é, tem execução baseada no tratamento de eventos, que

podem ser de dois tipos: (i) tiques i que levam ao início do i-ésimo round síncrono e (ii) mensagens de um round j provenientes de algum processo p. A cada instante do tempo, o estado de cada processo é determinado pelo round síncrono i do qual ele participa. Durante o round i o processo só aceitará, dentre as mensagens recebidas, aquelas que fazem parte do mesmo round i. Mensagens de rounds $j < i$ (atrasadas) e de rounds $j > i$ (adiantadas) serão descartadas. As mensagens aceitas são repassadas ao cliente para pré-processamento e considera-se que tal procedimento tenha custo desprezável, ou seja, não seja bloqueante e tenha duração da mesma ordem que o custo de recebimento e filtragem de mensagens.

O estado do protocolo é modificado pelo recebimento de tiques por seus processos. Ao receber um tique i num instante t do tempo local, um processo inicia o i-ésimo round sícrono do protocolo, formado por três fases: (i) processamento do round i; (ii) envio da mensagem do round i; e (iii) recepção e pré-processamento das mensagens do round i. Na primeira fase, o cliente será sinalizado do início do round i e iniciará sua etapa de processamento, que leva em consideração as mensagens recebidas no round anterior e pode constar de operações bloqueantes. Num instante $t_s > t$ o cliente finaliza seu processamento e retorna a mensagem a ser enviada no round i. O custo de processamento $t_s - t$ não é considerado desprezável e configura o *timestamp* (carimbo de tempo) da mensagem enviada. A partir do instante t_s um processo inicia a última fase do round e aguarda o recebimento das mensagens do round i enviadas pelos demais processos. Esta fase se encerra como a chegada do próximo tique.

O protocolo síncrono descrito acima implementa a camada síncrona de transporte de mensagens. O envio das mensagens se dá como consequência do recebimento de um evento de sincronização (um tique) e somente uma mensagem é enviada por round, o que se reflete num controle da carga imposta aos canais de comunicação. Cada processo irá receber e processar somente as mensagens recebidas dos processos que estejam sincronizados com ele, ou seja, estejam executando o mesmo round síncrono e que não sofram falhas de desempenho no decorrer do round.

3.3 Sincronização dos Processos

Um mecanismo de sincronização de processos, implementado pelo protocolo síncrono, é responsável por gerar os tiques que delimitam os rounds síncronos, de forma que são duas as propriedades desejáveis na sua implementação: (i) o mesmo tique i deve ser recebido por diferentes processos num intervalo do tempo físico com duração ϵ mínimo e (ii) o intervalo entre o recebimento de dois tiques consecutivos pelo mesmo processo deve ser regular e da ordem de uma constante Δ pré-estabelecida.

Estas propriedades de precisão e periodicidade caracterizam um problema de computação distribuída, que consiste num relaxamento do problema de sincronização externa de relógios e é chamado de *sincronização de pulsos* [7, 6]. Nele considera-se um sistema com n processos que se comunicam por trocas de mensagens e são capazes de gerar pulsos de sincronização. O objetivo do protocolo é fazer com que os processos corretos, com o passar do tempo, passem a gerar pulsos sincronizadamente e com periodicidade que se aproxime do tamanho do ciclo (round) desejado. Na presença de até $f < n/3$ *falhas abitrárias (bizantinas)*, os autores obtém precisão de δ quando elas são definitivas [7] e da ordem de 3δ quando elas são transientes [6], onde δ é a duração máxima estipulada para uma etapa de

processamento e envio de uma mensagem.

O modelo de falhas considerado pela camada síncrona não inclui a possibilidade de comportamento arbitrário e não limita a ocorrência de falhas transientes de desempenho. Além disto, a duração dos ciclos (rounds) obtidos por estes trabalhos é muito superior à duração esperada para os round síncronos. Assim, apesar do problema a ser resolvido ser o mesmo, o mecanismo implementado pela camada síncrona aqui apresentada difere consideravelmente dos mecanimos definidos nos trabalhos acima citados, por ser otimizada para obter as referidas propriedades de sincronização durante os períodos de "bom comportamento" temporal do sistema.

O protocolo de sincronização implementado pela camada síncrona é composto por um lado cliente e um lado servidor. O lado servidor, quando ativo, é responsável pela geração periódica de sinais de sincronização para todos os seus clientes. Dado um instante t_0 de ativação e um ciclo Δ, o servidor irá agendar, através do relógio do processador, a execução de ações nos instantes $t_1, t_2, t_3, \ldots$ gerados pela fórmula de recorrência $t_i = t_{i-1} + \Delta$. Ao ser acordado no instante t_i, o processo irá enviar aos demais preocessos a i-ésima mensagem de sincronização. As mensagens contém os seguintes campos: a *lease* do servidor, o número de sequência i, a duração Δ do ciclo e, como *timestamp*, a diferença entre o instante em que ocorreu o envio e o instante t_i no qual ele foi agendado. Tais mensagens são chamadas de *tiques remotos* e são enviadas empregando uma camada de transporte UDP/IP não confiável e um endereço *IP multicast* distinto do empregado pelo protocolo síncrono para transporte das mensagens da aplicação.

O *lease* de um tique identifica o servidor de sincronização que o gerou, sendo escolhida a partir de uma sequência crescente de valores, única para cada servidor e incrementada a cada (re)ativação. O par formado pelo *lease* e pelo número de sequência constitui o identificador único do tique. É através da análise deste identificador que o lado cliente do protocolo de sincronização avalia se deve aceitar um tique remoto: se ele for maior que o identificador do round síncrono atual, o tique será aceito e repassado à camada síncrona, que irá iniciar um novo round síncrono; caso contrário, ele será descartado. Além disto, se o servidor de sincronização de um processo estiver ativo e o cliente associado receber um tique com *lease* superior ao empregada pelo servidor, ele será desativado.

Assim, após um período inicial de instabilidade, a ordenação total dos leases empregados irá garantir que apenas um único processo da camada síncrona de transporte terá o seu servidor de sincronização ativo e será este processo que irá ditar o ritmo de criação de novos rounds síncronos.

Quando isto for válido, a precisão da sincronização será definida pela *variabilidade* nas latências de entrega de mensagens pela rede e será independente do número n de processos participantes. Em particular, em períodos de comportamento síncrono do sistema, é possível limitar a variabilidade das latências de rede a uma constante ξ. De forma que, em tais circunstâncias, o resultado obtido pelo protocolo é equivalente ao *menor erro* que é possível se obter na sincronização de relógios no modelo síncrono de computação – dado por $\xi(1 - 1/n)$ [19], conforme apresentado na seção 2 – ao se considerar um número infinito de processos.

3.4 Tolerância a Falhas

Se por um lado, o fato a geração de tiques ser gerenciada por um único processo propicia uma boa precisão ao serviço de sincronização, por outro, a criação de novos rounds síncronos depende diretamente do seu funcionamento correto. Assim, por ser um ponto único de falhas, o servidor de sincronização será monitorado por todos os seus clientes, que usarão temporizadores para aferir o cumprimento dos prazos no envio dos tiques remotos.

Para tal, eles irão empregar o instante de recebimento de uma mensagem de sincronização e as informações carregadas por ela, para estipular o instante em que o próximo tique deve ser recebida do mesmo servidor. Se o i-ésimo tique remoto tiver sido recebido num instante t do seu relógio local, o cliente espera receber o i+1-ésimo tique do mesmo servidor aproximadamente no instante $t + \Delta$. Para transformar esta esperança em um prazo, deve-se considerar tanto a variabilidade na latência de transmissão dos tiques, como a diferença nas taxas de progressão de diferentes relógios (*clock skew)*, valores que serão condensados pelos parâmetro γ[1] da camada síncrona. Assim, o prazo para a chegada de um novo tique remoto válido será de $\Delta + \gamma$ instantes de tempo após o recebimento do último tique e a sua violação constitui uma falha de desempenho do servidor de sincronização.

Considera-se, a priori, que tais falhas sejam transientes, ou seja, que o servidor volte a enviar tiques remotos. Ou então, que o não recebimento da mensagem de sincronização tenha se dado pela sua perda. Assim, o lado cliente do protocolo irá tolerá-las através da geração de *tiques locais* que substituem os tiques perdidos. Estes, tais como os remotos levam ao início de um novo round síncrono e carregam o mesmo lease, o mesmo período e números de sequência subsequentes ao do último tique recebido.

O problema desta abordagem é que a precisão dos tiques locais é menor que as dos tiques remotos, uma vez que o impacto da diferença de velocidade dos relógios locais será maior. Assim, após um certo período sem receber nenhum tique remoto, os clientes de sincronização irão ativar os seus servidores associados, o que levará à eleição de um novo processo que irá substituir o servidor de sincronização anterior, que falhou.

4 Difusão Síncrona Totalmente Ordenada de Mensagens

O problema da difusão totalmente ordenada de mensagens (*total order broadcast*) considera um conjunto de processos que enviam mensagens para os demais processos empregando um protocolo de comunicação comum. Informalmente, este protocolo deve garantir é que todas as mensagens enviadas serão entregues por todos os processos e que a entrega de diferentes mensagens se dará numa mesma ordem em todos processos. Mais formalmente, a versão *uniforme* deste problema pode ser definida pelas seguintes propriedades [8]:

Validade : se um processo correto envia uma mensagem m então ele irá entregar m.

Acordo : se um processo entrega m então todo processo correto irá entregar m.

[1] O parâmetro γ é definido com base nos limites da variabilidade nas latências de rede, ξ, e nas taxas de progressão dos relógios ρ, de forma que $\gamma \geq \Delta(1 + \rho) + 2\xi$.

Integridade : uma mensagem m será entregue por qualquer processo somente uma vez e se, e somente se, m tiver sido enviada por algum processo.

Ordem total : seja dois processos p e q que entregam duas mensagens m e m', então p entrega m antes de m' se, e somente se, q entrega m antes de m'.

Esta seção descreve a implementação de um protocolo, denominado *difusão síncrona totalmente ordenada de mensagens*, que resolve este problema através da abstração de rounds síncronos de comunicação (descrita na seção 2) implementada pela camada síncrona transporte descrita pela seção anterior. O protocolo é destinado ao modelo assíncrono temporizado de computação, de forma que sua corretude (propriedades de integridade e ordem total) é garantida mesmo na ocorrência de um número ilimitado de falhas de desempenho, mas seu progresso (propriedades de validade e acordo) depende do comportamento síncrono do sistema, daí o seu nome, e, em particular, da ausência de falhas definitivas de processos e de canais de comunicação.

4.1 Definições Gerais

Considera-se que o sistema é formado por n processos e que cada processo possui um identificador único, anexado a todas mensagens por ele enviadas. Os processos podem podem incorrer num número ilimitado de falhas de desempenho e sua comunicação se dá exclusivamente através da camada síncrona de transporte. Uma *configuração* do sistema é uma permutação dos identificadores únicos dos n processos participantes. A posição do identificador de cada processo nesta permutação configura o seu *slot*.

A cada processo estão associados um ou mais clientes, que requisitam o envio de mensagens. Tais mensagens são mantidas pelos processos em uma fila local e serão chamadas de mensagens da aplicação. Sempre que se afirma que um processo enviou uma *nova* mensagem, considera-se que ele removeu um conjunto de mensagens da aplicação presentes em sua fila local e gerou uma *proposta*. Se tal fila não tiver nenhuma mensagem, será gerada uma proposta *nula*. Caso contrário, à proposta serão adicionadas as k primeiras mensagens da fila, tal que k é limitado pelo tamanho máximo, em bytes, das propostas, que é um parâmetro da execução. As propostas geradas contém um identificador único (o que permite distingui-las) e um *número de sequência* associado. Todas as mensagens enviadas pelos processos carregam sempre uma proposta e o seu número de sequência i. Seja m uma mensagem qualquer, usa-se a notação $seq(m)$ para se referir ao seu número de sequência.

A associação de uma proposta a um número de sequência é definitiva e é definida no seu envio. Assim, se uma proposta p é enviada com o número de sequência i, qualquer reenvio com o número de sequência i deverá carregar a mesma proposta p. Como todos os processos iniciam o protocolo com um mesmo número de sequência inicial i_0, é possível definir o i-ésimo *bloco* de propostas como sendo o conjunto das propostas associadas a i em cada processo do sistema. A ordenação deste conjunto é dada pela configuração vigente: a j-ésima proposta é aquele gerada pelo processo que ocupa o *slot* j da configuração.

Da mesma forma que os clientes alimentam o protocolo com mensagens, o protocolo retorna ao cliente as mensagens requisitadas, segundo a ordem total definida. Assim, *entregar* um bloco de propostas consiste em retornar para aplicação o conteúdo de todas as

propostas do bloco, segundo sua ordem interna e segundo a ordem das mensagens dentro de cada proposta, desconsiderando-se as propostas nulas, pois estas não contém mensagens.

4.2 Visão Geral do Protocolo

O protocolo opera através dos rounds gerados gerados pela camada síncrona de tranporte. Ao final de cada round síncrono r o processo acumulou um conjunto, possivelmente vazio, de mensagens recebidas durante aquele round. Denota-se por M_r este conjunto e considera-se que ele contenha no máximo uma mensagem de cada processo e que todas elas empreguem o *slot* de envio determinado pela configuração vigente[2]. A análise do conjunto M_r – em particular sua cardinalidade e os números de sequência empregados pelos demais processos – determina qual é o novo estado de processo do protocolo de difusão e qual ação será realizada no próximo round.

O caso mais simples – e, espera-se, o mais frequente – ocorre quando o conjunto M_r recebido pelo processo é formado por n propostas, uma de cada processo, todas elas com o mesmo número de sequência. Quando isto ocorre, diz-se que o round r teve *sucesso*:

> Seja r um round síncrono e p um processo do protocolo de difusão, se ao final do round r o processo p obteve um conjunto M_r com $|M_r| = n$ e $\forall m \in M_r : seq(m) = i$, então p teve sucesso no round r e recebeu o i-ésimo bloco de propostas.

Se um processo teve sucesso em um round r então: (i) os demais processos estavam sincronizados com ele, ou seja, executavam simultaneamente o mesmo round síncrono; e (ii) os demais processos estavam tentando aprovar o mesmo bloco de propostas que ele, ou seja, estavam na mesma etapa de processamento do protocolo. Nestas condições, e *somente nestas*, a difusão totalmente ordenada progrede e o processo irá enviar uma mensagem com o número de sequência seguinte. De forma geral, é possível definir que:

> Um processo p envia uma mensagem m com $seq(m) = i + 1$ num round r se, e somente se, p tiver recebido o i-ésimo bloco de propostas em um round $r' < r$.

Como colorário imediato das duas propriedades destacadas, se um processo tem sucesso em um round r e recebe o bloco de propostas $i + 1$ então *todos os processos* já receberam, em rounds anteriores, o bloco i. Com isto, a propriedade de integridade é válida para as propostas do i-ésimo bloco (e, assim, para suas mensagens) e o bloco pode ser entregue:

> Se um processo tem sucesso num round r e recebe o bloco de propostas $i+1$ então i-ésimo bloco de propostas se torna estável e será entregue para a aplicação, caso $i \geq i_0$ e caso o i-ésimo bloco já não tenha sido entregue.

[2] A camada de transporte assícrona, neste caso o UDP, empregada pela camada síncrona de transporte pode perder ou até duplicar mensagens. O tratamento destes eventos pode ser feito tanto pela camada síncrona de transporte, como pelo protocolo de difusão totalmente ordenada de mensagens. Já a questão dos *slots* visa garantir que processos em configurações distintas não contaminem a ordenação estabelecida.

As propriedades até aqui apresentadas tratam sempre de rounds com sucesso, em que o processo envia a i-ésima proposta e recebe as i-ésimas propostas dos demais processos. Como o sistema precisa tolerar falhas de desempenho, deve-se considerar o caso em que este predicado não é válido, ou seja, rounds em que um processo p não obteve sucesso na difusão da mensagem.

4.3 Tolerância a Falhas de Desempenho

O primeiro caso que faz um round não ter sucesso ocorre quando um processo q do sistema incorre em uma falha de desempenho e não consegue enviar uma mensagem durante um round r. Neste caso, a cardinalidade o conjunto M_r recebido por todo processo será inferior a n, mas todas as propostas recebidas são relativas a um mesmo número de sequência i. Assim, nenhum processo terá sucesso no round r, nenhum processo irá entregar o bloco $i-1$ e todo processo irá reenviar a sua i-ésima proposta no round subsequente.

O segundo caso a ser considerado ocorre quando dois processos p e q tem uma visão distinta do sucesso de um round: para p o round r teve sucesso e ele recebeu o i-ésimo bloco de propostas, mas o mesmo round r não teve sucesso no processo q. A consequência é que no round subsequente p irá enviar sua proposta $i+1$, mas receberá ao menos uma propostas do i-ésimo bloco, reenviada por q. Da mesma forma, q estará *recuperando* o i-ésimo bloco, mas receberá pelo menos uma mensagem do bloco $i+1$, enviada por p. Assim, mesmo que p e q recebam as n mensagens do round $r' > r$, nenhum deles terá sucesso no round r'.

Como esta situação impediria o progresso do protocolo, um processo p que esteja decidindo o i-ésimo bloco durante um round r, mas receba uma mensagem $m \in M_r$ de q com $seq(m) = i - 1$ irá realizar um *roll back* para o bloco de propostas $i - 1$. Isto significa que no round $r' > r$ subsequente, p irá reenviar a sua i-1-ésima proposta e repetirá este procedimento até que tiver sucesso em receber (novamente) as n propostas do bloco $i - 1$, ou seja, até conseguir recuperar o processo q atrasado na sequência de blocos.

Estas duas situações podem ocorrer e se combinar nos vários processos do sistema e em vários rounds síncronos, o que torna complexa a sua análise. Assim, ao invés disto, apresenta-se a seguinte propriedade:

> Sejam dois processos p e q participantes do sistema, um round síncrono r qualquer e $current_p$ e $current_q$ os blocos de propostas que p e q estão tentando aprovar no round r, então sempre será válido $|current_p - current_q| \leq 1$.

Para provar a validade desta propriedade define-se uma nova variável $last_p$ que define o número de sequência da maior proposta enviada pelo processo p, ou seja, o maior bloco de propostas que ele tentou aprovar. Pela propriedade de progresso acima apresentada, p envia sua i+1-ésima proposta se, e somente se, tiver recebido a i-ésima proposta de todos processos, ou seja, $last_p = i + 1 \Rightarrow \forall q : last_q \geq i$. Assim, aplicando simetricamente a propriedade para qualquer outro processo q, obtém-se a relação: $|last_p - last_q| \leq 1$.

Em qualquer processo p, inicializa-se $last_p$ e $current_p$ com o mesmo valor i_0. Enquanto não houver um retrocesso (*rollback*) tem-se sempre $last_p = current_p$, uma vez que sempre que uma nova proposta é gerada $last_p$ é incrementado e tal proposta será enviada, de forma

que o bloco em aprovação será $current_p$. Nestes casos, pela relação acima, a propriedade será válida. Seja r o primeiro round em que um retrocesso ocorre durante a execução. Assim, algum conjunto M_r contém mensagens com números de sequência diferentes. Como até r não houve nenhum retrocesso, em qualquer processo p tem-se $last_p = current_p$. Assim M_r pode conter no máximo dois números de sequências: i e $i+1$. Em todos rounds $r' > r$ subsequentes em que houver: (i) algum processo p com $last_p = current_p = i$ e (ii) algum processo q com $last_q = current_q = i+1$ (ou seja, que não fez retrocesso), nenhum processo terá sucesso ao receber o i-ésimo ou o i+1-ésimo bloco de propostas. Como os processos p na situação (i) não podem avançar seu número de sequência, para conseguir progresso é necessário que os processos q na situação (ii) façam retrocesso. Isto determina, que mesmo quando um retrocess ocorre no sistema, há no máximo dois números de sequência i e $i+1$ em cada conjunto M_r de mensagens recebidas por qualquer processo no round r.

De modo geral, é possível definir que qualquer processo p não faz retrocesso mais de uma vez antes de recuperar o bloco anterior ao tratado no momento, perdido por algum processo, ou seja, $current_p \in \{last_p - 1, last_p\}$. Além disto, durante a recuperação de um processo q, com $last_q = last_p - 1$, o processo p não gera novas propostas. Assim, mesmo durante períodos de recuperação decorrente de retrocessos, $current_p - current_q \leq 1$.

4.4 O Algoritmo

A Figura 1 apresenta o algoritmo executado por todos processos p do protocolo de difusão totalmente ordenada de mensagens. Ele emprega dois parâmetros: o número de processos n e o número de sequência do primeiro bloco a ser aprovado i_0. As variáveis *current* e *next* indicam, respectivamente, qual o bloco que deve ser aprovado naquele round e o próximo bloco a ser entregue.

```
Inicializacao:
    current := i_0 /* Bloco de propostas atual */
    next := i_0    /* Proximo bloco a ser entregue */

Round r com mensagens M_r:
    seq := min_i{i = seq(m) : m ∈ M_r}
    if (|M_r| == n and seq == current) then /* Round com sucesso */
        if (next == current - 1) then
            deliver(next)
            next := current
        current := current + 1
    else if (seq < current) then /* Roll back para seq == current - 1 */
        current := seq
    proposal = load(current)
    send (proposal, current)
```

Algoritmo 1: Difusão com ordenação total de mensagens sobre a camada síncrona.

A função *load(i)* retorna a i-ésima proposta de p, tal que se $i > last_p$ uma nova proposta é gerada, a partir da fila de mensagens da aplicação pendentes para envio e, caso contrário, a i-ésima proposta já foi gerada e se encontra na memória principal. A proposta obtida, junto com o seu número de sequência, formam a mensagem que é entregue para a camada sícrona para ser enviada no início do próximo round $r' > r$.

A função *deliver(i)* é responsável por entregar para a aplicação as mensagens contidas no i-ésimo bloco de propostas. Como já descrito, as mensagens são extraídas das propostas na ordem estabelecida pela *configuração* em curso. A obtenção de uma configuração comum é um procedimento que faz parte da inicialização do protocolo e uma regra simples para estabelecê-la é empregar a permutação ordenada dos identificadores únicos dos processos.

O protocolo resultante implementa a entrega confiável e totalmente ordenada das mensagens geradas pelos processos participantes, através do uso da estratégia de rounds síncronos implementada pela camada síncrona. Cada processo envia uma mensagem do protocolo por round, que carrega um conjunto de mensagens da aplicação ou a informação de que o processo não tem nenhuma mensagem para ser difundida naquele instante. Esta mesma mensagem também implicitamente confirma o recebimento do último bloco de propostas enviado, o que garante o acordo entre os processos que implementam a difusão sobre o conjunto de mensagens anteriormente difundidas e a ordem em que elas devem ser entregues.

Assim, o protocolo de difusão é distribuído e homogêneo, uma vez que não há um processo distinto que age como sequenciador e todos os processos executam o mesmo código. Além disto, todos processos podem enviar suas propostas a cada round síncrono e não há um único processo emissor por vez ou uma divisão temporal do privilégio de difundir as próprias mensagens. Porém, ao mesmo tempo, uma nova proposta não será enviada enquanto a anterior (e todo o bloco do qual ela faz parte) não for aprovada. De forma que, apesar de sua abordagem mista, considera-se que o protocolo desenvolvido emprega um acordo dentre os destinatários das mensagens para definir a sua ordenação total [8].

Para esta classe de algoritmos de acordo ou consenso, o limitante inferior do custo para resolução do problema com tolerância a falhas de processos é conhecido, tanto para o modelo síncrono de computação [9], como para o assíncrono [18]: na presença de t falhas de processos, o *consenso uniforme* pode ser resolvido em no mínimo $t+2$ passos de comunicação. Dado este limitante, o protocolo apresentado é otimo, uma vez que na ausência de falhas de desempenho um processo p emprega dois rounds síncronos para entregar uma mensagem requisitada: r_i para envio da proposta e r_{i+1} para ter a confirmação do recebimento da proposta enviada e do sucesso de r_i por todos os demais processos.

Finalmente, apesar do protocolo apresentado não progredir na presença de falhas definitivas dos processos participantes, ele continua garantindo as propriedades de *safety* (segurança) mesmo na sua ocorrência, ou seja, ele é *uniforme*. Considerando o tipo de falha por parada, o protocolo apresenta a propriedade que se um processo p entregou o i-ésimo bloco de propostas, então todo processo correto do sistema também entregou o i-ésimo bloco ou o tem em memória principal. Assim, desde que o estado de um processo p ($current_p$, $next_p$ e as propostas associadas a estes números de sequência) seja mantido em memória persistente, p pode se recuperar e reingressar no protocolo.

5 Avaliação

Esta seção descreve os experimentos realizados para analisar o desempenho do protocolo de difusão síncrona totalmente ordenada descrito na seção 4. O ambiente computacional considerado é de aglomerados (*clusters*) dedicados ao processamento distribuído, formados

por máquinas idênticas e interconectadas por uma rede local de alta velocidade, nos quais as suposições de comportamento temporal correspondem às premissas do modelo assíncrono temporizado de computação.

O protocolo de difusão é executado sobre uma implementação da camada síncrona (descrita na seção 3), que fornece a abstração de rounds síncronos de comunicação e processamento, e emprega mecanismos para a sincronização dos processos. A qualidade de tal sincronização está diretamente relacionada ao comportamento temporal do sistema, em particular à variabilidade das latências de comunicação multiponto observadas. Já o desempenho do protocolo é limitado por dois fatores: (i) a quantidade de rounds executados por unidade de tempo, uma vez que que uma única mensagem pode ser enviada por round; e (ii) pela fração dos rounds síncronos executados que obtém sucesso, ou seja, em que todos processos enviam e recebem as mensagens dentro do período estipulado. Estas duas condições são contraditórias— quanto maior a duração dos rounds, maior é a sua probabilidade de sucesso, mas menor é a quantidade de rounds executados por unidade de tempo—de forma que o objetivo desta análise é obtenção de uma duração ótima para os rounds, que maximize o desempenho.

5.1 Ambiente experimental

O ambiente experimental empregado é um cluster de máquinas idênticas, equipadas com dois processadores Intel-Xeon Quadricore de 2.40GHz e 12GB de RAM, conectadas por uma rede Ethernet Gigabit dedicada ao processamento distribuído e em topologia estrela. O sistema operacional instalado é o GNU/Debian Linux 6.0 (Squeeze), os experimentos são executados sobre uma máquina virtual Sun Java SE Runtime Environment 1.6 e a comunicação entre os processos se dá pelo protocolo UDP/IP de transporte não confiável.

Como referência para o comportamento temporal do sistema, a Tabela 1 apresenta o resultado de seis séries de medições do tempo total de resposta (*Round Trip Time* ou RTT) para mensagens enviadas em *multicast* no ambiente experimental. As medições são realizadas por um processo servidor, que envia uma mensagem aos n processos clientes e aguarda suas respostas, registrando o intervalo total entre o envio e a recepção da n-ésima resposta, segundo seu relógio local. Este intervalo inclui as latências de envio de duas mensagens idênticas de 100 bytes pela rede, as latências de processamento das diversas camadas de comunicação e o custo de processamento nos clientes, que inclui o custo de serialização, desserialização e enfileiramento de mensagens pela camada de aplicação.

Os dados foram aferidos por 6 experimentos independentes, empregando máquinas distintas como servidores e como clientes. Em todos os experimentos tem-se $n = 5$ máquinas que respondem às solicitações (os *pings*) do servidor. O valor computado corresponde à diferença, segundo o relógio local do servidor, entre o instante de envio e o instante de recepção da m-ésima mensagem, em microssegundos ($1\mu s = 10^{-6}s$). O intervalo entre o envio de dois *pings* consecutivos é aleatório, mas grande o suficiente para que não haja interferência nos tempos de resposta de cada *ping* individual A análise é realizada sobre um conjunto de $M = 10050$ medições, sendo que os primeiros 50 valores são descartados, de forma a amenizar o efeito dos diferentes instantes de início dos programas nos processos. Os indicadores estatísticos (desvio padrão, média, mediana e percentis) são calculados a partir

	RTT (μs)					
Mínimo	**520.403**	**590.330**	554.243	581.731	561.727	586.292
Máximo	919.617	935.025	909.258	946.783	**4108.802**	**896.976**
Médio	**728.297**	766.932	756.555	**768.052**	757.254	754.172
Desvio padrão	57.738	59.578	60.272	51.002	**121.275**	**50.871**
Mediana	**735.080**	**777.314**	766.349	774.677	763.158	761.715
90%-percentil	**786.651**	**831.992**	827.616	825.699	822.734	814.517
99%-percentil	873.648	**885.093**	876.423	866.904	873.887	**863.077**

Tabela 1: Tempo total de resposta (em μs) a um *ping* por um conjunto de $n = 5$ clientes.

da análise de todas medições e não pressupõem nenhuma distribuição específica.

A análise destes dados confirma a suposição que o comportamento temporal do sistema atende às premissas do modelo assíncrono temporizado de computação distribuída. Os valores mínimos, médios e os percentis observados são estáveis e compatíveis entre si, o que confirma a premissa de um comportamento prevalentemente síncrono. Por outro lado, mesmo para execuções relativamente curtas (cada rodada dura menos de 10 minutos), a latência máxima observada (4108.802μs, no quinto experimento) não é previsível com base nos outros indicadores estatísticos computados, mesmo com uma margem de segurança considerável. Assim, é difícil estipular prazos máximos para realização de ações pelo sistema e, mesmo se computados, tais prazos não são representativos em relação ao comportamento geral observado. Como resultado, ao se associar a duração dos rounds síncronos de comunicação às latências máximas observáveis ou a prazos que serão cumpridos com alta probabilidade pelo sistema, o resultado seria um protocolo ineficiente. Porém, ao associar-se a duração dos rounds síncronos ao comportamento geral do sistema, com alguma margem de segurança, e implementar-se uma solução que tolere a ocorrência de períodos de comportamento assíncrono (*outliers* observados), o protocolo implementado deveria apresentar um bom desempenho.

5.2 Resultados

O principal parâmetro para avaliar o desempenho do protocolo de difusão implementado é o *throughput* máximo obtido, ou seja, a quantidade de dados entregues à aplicação por unidade de tempo. Para computá-lo, emprega-se um cliente que gera mensagens aleatórias de tamanho fixo de 10000 bytes e requisita o seu envio e ordenação ao protocolo. Considera-se que sua capacidade de gerar mensagens é infinita, ou seja, que sempre que o protocolo puder enviar uma nova mensagem da aplicação haverá uma requisição do cliente na fila de mensagens pendentes. Paralelamente, este mesmo cliente recebe as mensagens entregues pelo protocolo e atribui a elas o instante, segundo seu relógio local, em que cada mensagem foi recebida. Assim, seja t_0 o instante de início do processamento, se no instante t o cliente tiver acumulado X bytes em mensagens recebidas, o *throughput* médio será de $X/(t - t_0)$ bytes/s.

Por construção do protocolo, o *throughput* máximo que pode ser obtido depende direta-

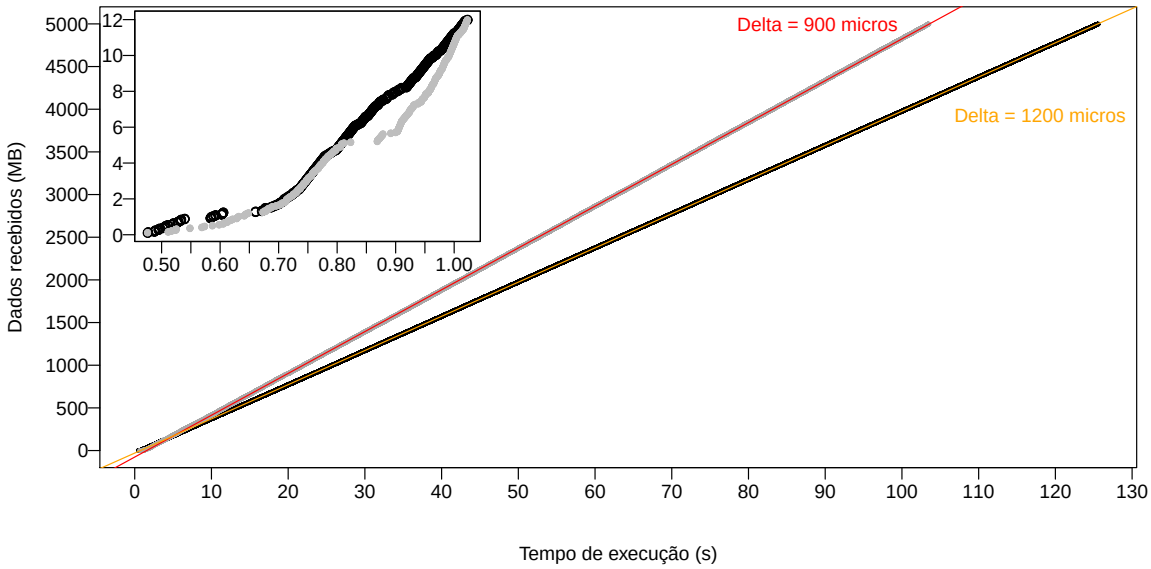

Figura 1: Quantidade de bytes recebidos por duração da execução, para $n = 5$ processos e duas durações de duas durações de rounds síncronos: 900 e 1200μs.

mente da duração dos rounds gerados pela camada síncrona. Num sistema com n processos e com rounds síncronos de Δ segundos de duração, se todo processo envia uma nova mensagem de m bytes a cada round, o valor máximo obtido será de (nm/Δ) bytes/s. Já na execução prática do protocolo, o *throughput* instantâneo está relacionado à frequência na ocorrência de falhas de desempenho, que se reflete na proporção dos rounds executados que tem sucesso. A tendência é que esta frequência seja variável, com períodos de assincronia, em que a maior parte dos rounds não terá sucesso na maior parte dos processos, e períodos "síncronos", nos quais a maioria dos rounds terá sucesso em todos os processos.

A Figura 1 apresenta o comportamento de duas execuções do protocolo com durações distintas para os rounds síncronos: 900μs e 1200μs. Em destaque na parte superior esquerda tem-se o comportamento no primeiro segundo de execução do protocolo de difusão. Nota-se a existência de intervalos praticamente contínuos de entrega e de intervalos descontínuos. Os primeiros refletem uma sequência de rounds síncronos com sucesso, em que se nota progresso na entrega de mensagens com o incremento da quantidade de dados recebidos com o passar do tempo. Já os intervalos de tempo onde há descontinuidades refletem períodos de assincronia, em que os processos não conseguem cumprir os prazos estabelecidos e não se tem progresso na entrega de novos dados aos clientes.

Em uma visão macroscópica da execução, mostrada pelo gráfico maior, com cerca de dois minutos de duração ($\approx 127s$ para $\Delta = 1200\mu s$) os intervalos de assincronia não são mais claramente discerníveis e nota-se que há um progresso quase contínuo na entrega de mensagens. Isto permite que os vários pontos de medição (quantidade de dados recebidos por tempo) possam ser representados aproximadamente por uma reta, cujo coeficiente angular determina o que chamaremos de *throughput efetivo* do protocolo, ou seja, uma média

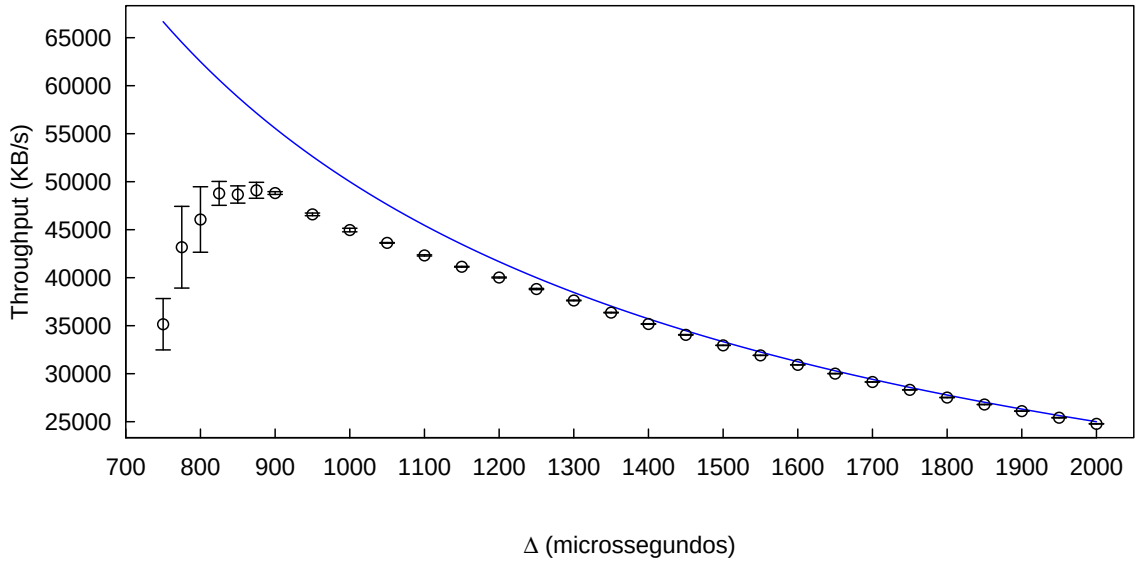

Figura 2: Taxa de entrega de dados em função da duração dos rounds síncronos, para $n = 5$ processos. A curva em azul representa a taxa teórica máxima em função de Δ.

de sua taxa de entrega instantânea.

Com base nestes conceitos, a Figura 2 apresenta a distribuição do comportamento do protocolo em função do tamanho Δ dos rounds síncronos empregados. Cada ponto no gráfico apresenta o *throughput* efetivo para um valor de Δ, computado através da reta de melhor ajuste, conforme visualizado na Figura 1. Em cada experimento, cada um dos $n = 5$ processos envia mensagens de $m = 10000$ bytes cada e a execução é longa o bastante para que o coeficiente angular das retas computadas se estabilize. Cada círculo no gráfico corresponde ao valor médio dos coeficiente angulares medidos pelos n processos em cinco execuções independentes. Dentro de uma mesma execução, o desvio de tais valores é reduzido, mas o desvio padrão entre os resultados de diferentes execuções é representado pela barra que corta cada ponto. A curva em azul representa o *throughput* máximo possível para cada valor de Δ, segundo a função $f(\Delta) = (n * m)/(\Delta/1000)$, que retorna resultados em bytes/milissegundo, equivalente a KB/segundo.

Δ	$2000\mu s$	$1200\mu s$	$1000\mu s$	$900\mu s$	$875\mu s$	$750\mu s$
Throughput Máximo	25000.00	41666.67	50000.00	55555.57	57142.86	66666.67
Throughput Médio	24776.76	40019.55	44964.40	48816.71	49104.02	35152.03
Desvio Padrão	20.46	49.55	187.65	143.78	831.31	2674.80
Eficiência média	99.11%	96.05%	89.93%	87.87%	85.93%	52.73%

Tabela 2: Valores médios para a taxa de entrega de dados (em KB/s) pelo protocolo em função da duração dos rounds síncronos, num sistema com $n = 5$ processos.

O comportamento do gráfico da Figura 2 é detalhado pela Tabela 2 e apresenta um padrão, que é verificável para outros valores de n. Como previsto teoricamente, quanto maior é a duração dos rounds síncronos, menor é a probabilidade dos processos não respeitarem os prazos, tem-se uma menor incidência de falhas de desempenho e uma menor a proporção de rounds que não obtém sucesso global. Consequentemente, tem-se uma maior *eficiência* na sincronização: o valor do *throughput* obtido é estável e sua variação, mesmo em execuções independentes, é reduzida. No outro extremo, tem-se prazos muito curtos para a realização de ações, o que faz com que qualquer pequeno atraso por parte de qualquer um dos processos leve ao insucesso de um round, o que diminui o desempenho da solução. Por exemplo, para Δ de $750\mu s$ tem-se quase 50% de rounds sem sucesso global, o que é aceitável dado que este valor é da ordem das medianas dos RTTs medidos para o sistema (vide Tabela 1).

Como meio termo para tais casos extremos, tem-se, para valores intermediários da duração dos rounds síncronos, dois comportamentos distintos. A partir de $2000\mu s$, ao se reduzir a duração do round, observa-se uma diminuição contínua na eficiência do protocolo, que corresponde a uma maior proporção de rounds síncronos sem sucesso. Porém, para valores de Δ até em torno de $900\mu s$, esta redução na eficiência é acompanhada do aumento do *throughput* efetivo obtido, visto que seus valores médios crescem e sua variância continua sendo reduzida. Ao reduzir ainda mais a duração dos rounds síncronos, o segundo tipo de comportamento é observado: o *throughput* aferido deixa de ser crescente e apresenta variâncias consideráveis. Ou seja, o sistema se torna *saturado*, os processos e canais de comunicação perdem sincronia e o desempenho efetivo do protocolo cai.

6 Conclusão

Este trabalho apresentou um novo protocolo para a difusão síncrona de mensagens com ordenação total, com ênfase na obtenção de alto desempenho em ambientes de clusters genéricos dedicados ao processamento distribuído. O protocolo foi elaborado a partir da premissa de que, mesmo que tais sistemas sejam assíncronos, o seu comportamento temporal, em execuções com carga controlada, será *predominantemente* síncrono. Ou seja, que é possível determinar prazos para a realização de ações distribuídas que serão cumpridos com alta probabilidade, mas também desrespeitados de forma arbitrária pelo sistema.

Tendo em vista esta premissa, foi elaborada uma camada de software que fornece abstrações do modelo síncrono de computação – sincronização dos processos e rounds síncronos – cuja validade está associada aos períodos de comportamento síncrono do sistema. Sobre esta camada foi implementado um protocolo de difusão totalmente ordenada de mensagens, que tem seu progresso associado à qualidade da sincronização obtida, mas cuja corretude é mantida mesmo em períodos em que o sistema se comporta assincronamente. Assim, a solução descrita é otimizada para períodos de sincronia e ausência de falhas, mas não depende destas circunstâncias para garantir suas propriedades de segurança.

Os experimentos realizados confirmam a validade da nossa premissa inicial sobre o comportamento temporal do sistema e a sua modelagem através do modelo assíncrono temporizado de computação. O *throughput* medido para o protocolo, em um cluster com cinco

máquinas, foi da ordem de 45 a 50 MB/s. A título inicial de comparação, a outra implementação que temos para este mesmo problema, o Treplica [2], supõe um modelo assíncrono de computação, falhas com recuperação e é baseada em Paxos [16] e Fast Paxos [17]. Apesar das diferenças no modelo de falhas e no modelo temporal empregados, o *throughput* obtido pelo protocolo aqui descrito é cerca de 20 vezes vezes superior aquele obtido pelo Treplica.

Assim, dada esta grande diferença de desempenho, duas abordagens são possíveis para trabalhos futuros na investigação do papel da sincronia nos protocolos de difusão de mensagens com ordenação total. A primeira consiste em fortalecer a tolerância a falhas deste protocolo e fazê-lo suportar, por exemplo, falhas definitivas de processos, com ou sem um sucessivo retorno ao processamento (o que traria, em particular, o desafio de implementar a persistência do estado do protocolo, conforme feito no Treplica). Uma segunda abordagem possível consiste em empregar este protocolo como uma espécie de camada de transporte síncrona, confiável e com ordenação total de mensagens, que é um módulo (componente) de um outro protocolo que implemente as propriedades de persistência de estado e tolerância a falhas com recuperação.

Referências

[1] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems*. USENIX Association, 2009.

[2] Luiz Eduardo Buzato, Gustavo Maciel Dias Vieira, and Willy Zwaenepoel. Dynamic content web applications: Crash, failover, and recovery analysis. In *DSN 2009: 39th International Conference on Dependable Systems and Networks*, pages 229–238, Estoril, Lisbon, Portugal, June 2009.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989. 10.1007/BF01784024.

[5] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, 1999.

[6] Ariel Daliot and Danny Dolev. Self-stabilizing byzantine pulse synchronization. Technical Report cs.DC/0608092. TR-2005-84, The Hebrew University, Aug 2006.

[7] Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 32–48. Springer Berlin / Heidelberg, 2003.

[8] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[9] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47 – 52, 2003.

[12] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460 –467, jun 1992.

[13] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, C-36(8):933 –940, aug. 1987.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[15] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[16] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[17] Leslie Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, October 2006.

[18] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104—125, June 2006.

[19] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[20] M. Raynal. Consensus in synchronous systems: a concise guided tour. Technical report, Institut de Reserche en Systèmes Alèatoires, Universitè de Rennes 1, July 2002.