Processamento Justo de Transações em Bancos de Dados Tolerantes a Faltas Bizantinas

Aldelir Fernando Luiz[1,2], Lau Cheuk Lung[2,3], Miguel Correia[4]

[1]Câmpus Avançado de Blumenau - Instituto Federal Catarinense - Brasil

[2]Departamento de Automação e Sistemas - Universidade Federal de Santa Catarina - Brasil

[3]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina - Brasil

[4]Instituto Superior Técnico - Universidade Técnica de Lisboa / INESC-ID - Portugal

aldelir@das.ufsc.br, lau.lung@inf.ufsc.br, miguel.p.correia@ist.utl.pt

Resumo. *Nos últimos tempos, tem havido um crescente interesse na investigação e proposição de mecanismos para tolerar faltas Bizantinas em transações. As soluções propostas recentemente são baseadas no método de processamento e concorrência otimista, porém, tal método permite que parcela de transações sejam suscetíveis ao fenômeno conhecido como inanição (ou starvation). Este artigo apresenta uma solução para este problema, em que é proposto um protocolo para o processamento justo de transações, por meio de uma abordagem adaptativa e tolerante a faltas Bizantinas. O protocolo apresentado é robusto e garante a terminação de uma transação em até $f+1$ tentativas, no melhor caso.*

Abstract. *Nowadays, there is an increasing interest on mechanisms for Byzantine fault tolerance in transaction processing. Several recently proposed mechanisms follow an optimistic approach. However, optimistic transaction processing is inherently susceptible to a phenomenon known as starvation. This paper presents a solution to the starvation problem: a fair and starvation-free protocol for an environment subject to Byzantine faults. Our protocol is adaptive and robust, guaranteeing the termination of a transaction in $f+1$ attempts in favorable conditions.*

1. Introdução

A replicação de dados tem sido amplamente empregada no contexto de sistemas computacionais, em especial em sistemas de computação distribuída, como ferramenta para permitir o gerenciamento confiável dos dados processados nas computações. Em geral, os dados são manipulados por meio de abstrações conhecidas na literatura como transações [Bernstein et al. 1987]. Uma transação consiste em uma unidade lógica de trabalho que assegura a execução atômica de uma sequência de operações de manipulações de dados em um sistema de computação, a qual garante que todas estas operações sejam refletidas corretamente no sistema, ou nenhuma o será. Neste sentido, a replicação de dados representa ainda, um desafio em se tratando do gerenciamento de transações em ambientes sujeitos a faltas de alguma natureza. Não obstante, a evolução de tecnologias e facilidades para o processamento de transações incorre no surgimento de novos modelos e classes de aplicações, que por sua vez trás novos desafios à confiabilidade.

Neste sentido, trabalhos recentes têm proposto soluções para o gerenciamento de transações sobre dados replicados, mais especificamente em sistemas

de bancos de dados (BD) [Gashi et al. 2007, Vandiver et al. 2007, Luiz et al. 2011, Garcia et al. 2011, Pedone et al. 2011]. Dentre os trabalhos encontrados na literatura, [Luiz et al. 2011, Garcia et al. 2011, Pedone et al. 2011] usam a abordagem otimista para o processamento de transações [Kung and Robinson 1981], enquanto [Gashi et al. 2007, Vandiver et al. 2007] são protocolos pessimistas. Em protocolos otimistas se pressupõe que durante o processamento das transações são raras as interferências e/ou conflitos, então a serialização é verificada apenas na confirmação da transação. Por outro lado, em protocolos pessimistas se admite que ocorrerão interferências e/ou conflitos entre transações, e por isso eles adotam mecanismos de bloqueios para garantir a serialização enquanto a transação está ativa. Um aspecto de grande importância em se tratando de transações concorrentes é a suscetibilidade destas a um fenômeno conhecido por **inanição** (do inglês *starvation*), em que parcela de transações tenta ser processada diversas vezes, mas nunca consegue ser finalizada. Em ambientes sujeitos a ações arbitrárias por parte dos processos, o problema torna-se ainda mais agravado, pois a ocorrência de faltas pode impedir que determinadas transações atinjam seu término, de modo que elas são reiniciadas infinitas vezes, sem obter êxito no processamento de suas operações.

Nenhum dos trabalhos propostos no contexto do gerenciamento de transações em ambientes sujeitos a faltas Bizantinas (BFT - *Byzantine Fault-Tolerance*) [Gashi et al. 2007, Vandiver et al. 2007, Luiz et al. 2011, Garcia et al. 2011, Pedone et al. 2011] tratam ou propõem soluções para o problema da inanição. Desta forma, em situações onde há a violação da consistência por parte das transações, não é realizado nenhum tratamento para garantir o término destas, e elas tem de ser canceladas não espontaneamente. Este artigo apresenta uma solução para o problema de inanição em transações sujeitas a faltas Bizantinas, e até onde sabemos é o primeiro trabalho a considerar uma solução para este cenário, e, portanto, representa uma contribuição na direção de protocolos de replicação BFT para transações. A característica inovadora introduzida por este trabalho é a possibilidade de se garantir a terminação e o progresso justo de uma transação, a despeito de faltas Bizantinas (i.e. benignas ou maliciosas) da parte dos clientes e das réplicas.

Nossa proposta visa prover um ambiente confiável para a execução de transações de longa e de curta duração (i.e. OLTP e OLAP [Hasse and Weikum 1997]). No entanto, a combinação destes dois tipos de transações em um mesmo ambiente facilmente causa inanição, onde em geral, transações de curta duração cancelam transações concorrentes de longa duração [Weikum and Vossen 2002]. Com isso, ao invés de utilizar um único mecanismo para o processamento de transações (i.e. pessimista ou otimista), propõe-se o uso de um mecanismo adaptativo [Tai and Meyer 1996] conjuntamente ao uso de um contador monotônico (i.e. um *ticket*). Para tanto, a estratégia proposta integra três elementos de controle de concorrência já conhecidos na literatura: controle de concorrência baseado em *ticket* [Georgakopoulos et al. 1994]; controle de concorrência pessimista [Bernstein et al. 1987] e; o controle de concorrência otimista [Kung and Robinson 1981]. Este mecanismo adaptativo visa permitir que as transações canceladas não espontaneamente em detrimento de outras, sejam novamente submetidas por meio de um algoritmo de processamento e terminação justa, a fim de garantir que as estas alcancem seu término em um número finito de tentativas de execução, evitando assim sua inanição.

2. Trabalhos Relacionados

Poucos são os trabalhos que investigam transações e faltas Bizantinas. O primeiro [Molina et al. 1986] apenas investigou o uso de acordo bizantino e replicação de máquina de estados no contexto de bancos de dados (BD), onde propôs a execução das operações em BDs de maneira sequencial, sem concorrência entre transações. Em [Gashi et al. 2007] foi apresentado o uso de diversidade como mecanismo para tolerar faltas em transações em BDs, onde os autores demonstraram que uma série de *bugs* podem induzir os BDs a manifestar faltas Bizantinas. Uma solução apresentada para o problema foi uma arquitetura de *middleware* que isolou as faltas observadas, por meio da submissão das operações das transações sequencialmente às réplicas, e da verificação da consistência dos resultados através de um mecanismo de voto majoritário; o que limitou em muito a concorrência entre as transações naquele ambiente.

Em [Vandiver et al. 2007] os autores apresentaram o HRDB, em que um protocolo de escalonamento por barreira de confirmação (do inglês - *commit barrier scheduling*) é a base para prover um controle de concorrência robusto, e permite assegurar o comportamento correto sistema e a consistência forte dos dados, ao mesmo tempo que obtém elevado grau de concorrência, a despeito de faltas bizantinas. Uma fraqueza observada neste trabalho é a dependência de um coordenador centralizado - uma entidade confiável do sistema, cujo papel é restringir a ordem em que as operações serão enviadas às réplicas, a fim de evitar conflito entre as transações e preservar a concorrência. Recentemente foi proposto o Byzantium [Garcia et al. 2011], um protocolo que provê um suporte à execução de transações em BDs sujeitos a faltas Bizantinas, que relaxa a consistência a fim de obter um maior grau de concorrência no processamento das transações. Em suma, o protocolo de acordo Bizantino é o PBFT [Castro and Liskov 1999], e o suporte de transações adota como semântica de consistência o *snapshot isolation* [Gray et al. 1996]. No entanto, o *snapshot isolation* não provê uma semântica equivalente à execução sequencial de transações, tal como ocorre na serialização [Bernstein et al. 1987]. Além disso, estudos evidenciaram que esta semântica é suscetível a anomalias, que afetam em potencial a integridade dos dados manipulados nas transações [Berenson et al. 1995]. Isso implica que o sistema pode vir a sofrer um colapso em decorrência da corrupção de dados e violação de integridade, causadas por uma entidade faltosa (benigna ou maliciosa).

Por fim, duas contribuições que visam a busca de soluções para as lacunas encontradas nos trabalhos mencionados até então, são os protocolos BFT-*Deferred Update* [Pedone et al. 2011] (ou BFT-DU) e o apresentado em [Luiz et al. 2011]. Ambos os trabalhos apresentam soluções baseadas no processamento otimista, em que os protocolos não dependem de uma entidade centralizada, tal como ocorre no HRDB, e ambos adotam como critério de consistência a serialização. A principal diferença entre eles, é que [Pedone et al. 2011] parte do pressuposto de que apenas as réplicas são suscetíveis a faltas Bizantinas. Com isso, o protocolo realiza as operações de leituras apenas sobre a réplica primária, e as operações de escritas são efetuadas primeiramente em um *buffer* local do cliente, sendo propagadas às réplicas apenas no pedido de confirmação da transação, que parte do cliente. Já o protocolo de [Luiz et al. 2011], embora apresente latência superior no processamento de uma transação, é o único que assume réplicas e clientes Bizantinos e é capaz de manter as propriedades do sistema em condições de faltas nestas entidades. Não obstante, embora os trabalhos aqui mencionados permitam gerenciar transações em ambientes Bizantinos, todos os protocolos que fazem parte destes podem sofrer por inanição.

3. Contextualização do Problema

Existem dois tipos de aplicações que fazem uso de transações em bases de dados: OLAP (*Online Analytical Processing*) e OLTP (*Online Transactional Processing*) [Hasse and Weikum 1997]. De um lado, estão as aplicações OLAP, que são orientadas à síntese, análise e consolidação de dados, onde efetuam o processamento analítico e on-line dos dados, por meio de transações que realizam diversas operações de leitura, de longa duração (p. ex. *business intelligence*). De outro lado, as aplicações OLTP são orientadas à manipulação de dados operacionais (p. ex. transações bancárias), em que as transações são de curta duração, e em geral contêm operações de leitura e de escrita.

A conciliação das transações destes dois tipos de aplicações é uma tarefa difícil, principalmente porque a contenção de recursos da parte de uma ou outra pode implicar no cancelamento daquela que não detém a posse dos mesmos. Neste caso, privilegiar uma ou outra pode induzir o sistema a situações de inanição daquelas transações não privilegiadas. Este cenário é ainda mais agravado se considerado o comportamento arbitrário oriundo de faltas Bizantinas por parte das entidades do ambiente, o que é factível em nosso modelo/sistema. É digno de nota, que a inanição pode ocorrer em qualquer situação em que for verificada a existência de conflitos entre os itens de dados das transações em execução, independente do tipo de aplicação em uso (i.e. OLTP ou OLAP).

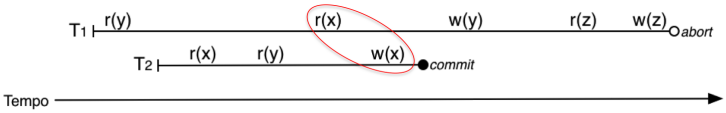


Figura 1. Cenário com transações de longa e de curta duração em conflito.

A Figura 1 ilustra um conflito entre uma transação de longa duração T_1 e uma transação de curta duração T_2. No caso, T_2 é iniciada após e concluída antes de T_1. Como T_2 escreve em um item de dados que foi lido por T_1, fica comprometida a confirmação de T_1, já que T_2 é bem sucedida e confirmada por não violar a serialização. T_1 por sua vez, se confirmada, violará o critério de serialização por ter realizado uma leitura-suja de x. Este cenário pode se repetir inúmeras vezes até que T_1 consiga, de fato ser terminada, e confirmada ou cancelada ao seu término. Uma solução trivial para o problema poderia ocorrer pela concessão de prioridades para as transações canceladas involuntariamente pelo protocolo, e também pelo uso dos esquemas clássicos para o tratamento de inanição em bancos de dados, tal como o *wound-wait* e *wait-die* [Weikum and Vossen 2002]. Porém, como estamos a lidar com faltas Bizantinas, o uso de prioridades seria algo um tanto delicado, já que uma transação privilegiada de um cliente faltoso poderia ficar indefinidamente em atividade, sem manifestar interesse na confirmação ou cancelamento, a fim de impedir o progresso de outras transações honestas que fazem referência aos mesmos itens de dados daquela privilegiada. Não obstante, se observa que os esquemas clássicos foram propostos sem hipóteses acerca da ocorrência de falhas [Weikum and Vossen 2002].

4. Visão Geral da Proposta

Conforme já citado, nossa proposta parte da integração dos elementos de controle de concorrência baseado em *ticket* [Georgakopoulos et al. 1994], pessimista [Bernstein et al. 1987] e otimista [Kung and Robinson 1981]. Contudo, diferente do propósito para o qual o método do *ticket* foi inicialmente definido

[Georgakopoulos et al. 1994], empregamos o uso do contador monotônico (i.e. um sequenciador) baseado em *ticket* para assegurar a ordem na qual uma transação poderá ter êxito em sua execução plena, independente se ao término ela seja confirmada ou cancelada. Em outros termos, o *ticket* é usado para determinar a ordem na qual as transações que são canceladas não espontaneamente devido à conflitos, faltas, etc., possam ser de fato, processadas e concluídas com êxito. A intenção em se evitar a inanição, é, principalmente, reduzir o desperdício de computações e recursos para impedir uma degradação de desempenho do sistema, em decorrência de reprocessamentos sucessivos de transações.

É importante destacar, que apenas o uso do *ticket* não permite resolver completamente o problema da inanição em transações. E para garantir o progresso e terminação justa das transações e evitar a inanição, a solução apresentada adota uma estratégia de escalonamento adaptativa e dinâmica de transações que, quando apropriado, alterna o método de controle de concorrência da transação de otimista para pessimista. Além disso, propomos a adaptabilidade não apenas para o controle de concorrência empregado, mas também para o tipo de escrita de transações, isto é, pré-declaradas e dinâmicas (vide Seção 4.3.).

4.1. Modelo de Sistema

O modelo de sistema adotado é composto por dois tipos de processos: as réplicas (ou servidores) $R = \{r_1, r_2, ..., r_n\}$ implementam o ambiente de BDs replicado, e os clientes $C = \{c_1, c_2, ..., c_n\}$ que fazem uso da base de dados. Assim, assumimos que C contém um número arbitrário (não infinito) de processos, e que R tem cardinalidade $|R| \geq 3f + 1$. Neste caso, um número ilimitado de clientes e até $f \leq \lfloor \frac{n-1}{3} \rfloor$ réplicas podem se desviar de forma arbitrária de suas especificações, com possibilidade de parar; omitir o envio e/ou a recepção ou ainda a entrega das mensagens; enviar respostas incorretas às operações dos clientes, bem como atuar em conluio com outros processos visando à corrupção do sistema. De outro modo, admitimos a independência de falhas por meio do uso de diversidade [Obelheiro et al. 2005], de tal forma que a ocorrência de uma falta em uma determinada réplica não necessariamente causa a mesma falta nas demais.

A saber que o modelo de interação assíncrono não permite assegurar a terminação de transações [Bernstein et al. 1987], adotamos o modelo de sincronismo terminal (*eventually synchronous system*) [Dwork et al. 1988]. A escolha por este modelo se justifica pela sua característica realista, onde o sistema se porta de forma assíncrona em grande parte do tempo, mas durante os períodos de estabilidade o tempo de transmissão de mensagens e das computações é limitado, porém, desconhecido. O critério de consistência adotado é o *one-copy serializability* (1-SR) [Bernstein et al. 1987], em que a execução concorrente de transações em uma base de dados "replicada" é equivalente à execução sequencial das mesmas transações em uma base de dados "não replicada". A despeito da diversidade, as réplicas são deterministas (i.e. todas as réplicas suportam o mesmo subconjunto de operações de manipulação de dados). Logo, o resultado de uma determinada operação é o mesmo para todas as réplicas. Por fim, nosso modelo de transações admite que um cliente submeta apenas uma transação por vez, e no contexto de uma transação, uma operação é enviada apenas se não há operações pendentes de execução para aquela transação.

Todas as comunicações entre os processos ocorrem por meio de canais ponto-a-ponto confiáveis e autenticados, e com ordenação FIFO; e o protocolo subjacente para difusão com ordem total adotado é o PAXOS Bizantino [Zielinski 2004], onde assumimos o uso de uma função de resumo criptográfico resistente à colisão para assegurar a integridade,

bem como códigos de autenticação de mensagens (p. ex. vetores de MACs) para assegurar a autenticidade das comunicações. Este mecanismo também é empregado para prover o controle de acesso dos clientes à base de dados, de modo que apenas mensagens oriundas de clientes devidamente autenticados são recebidas e entregues pelas réplicas. Por fim, o acesso aos objetos da base de dados é regulado por um mecanismo de controle de acesso discricionário, assim como ocorre em listas de controle de acesso (ACL).

4.2. Preliminares

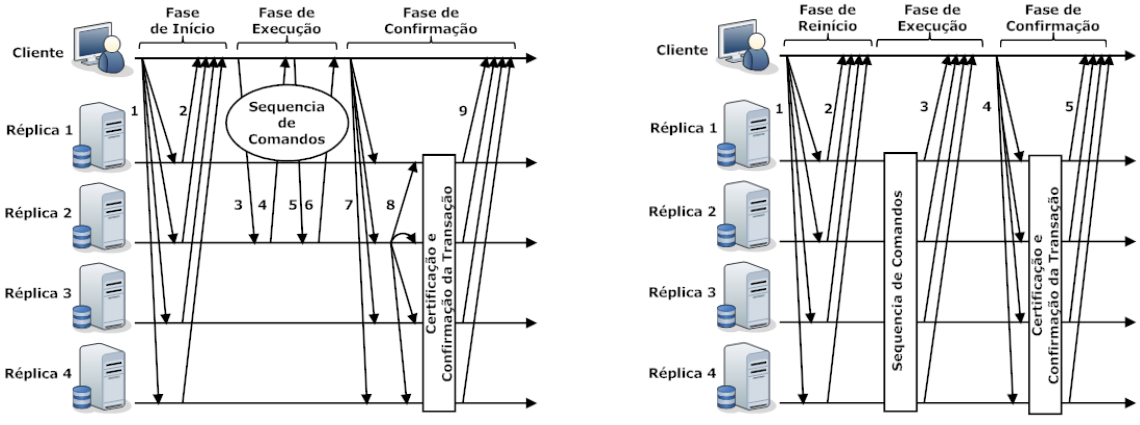
Conforme preconiza o modelo de transações [Bernstein et al. 1987], uma transação consiste em uma sequencia finita de operações de leitura e escrita, iniciada por uma instrução ***begin*** e finalizada por uma instrução ***commit*** ou ***abort***. As operações de uma transação devem satisfazer as propriedades de atomicidade, isolamento e durabilidade [Bernstein et al. 1987]. Para tanto, nossos algoritmos devem atender as condições de correção (*safety*) e vivacidade (*liveness*), e satisfazer as seguintes propriedades:

- **Consistência (*safety*):** o protocolo assegura o critério de consistência 1-SR (*one-copy serializability*) e o mantém para cada transação confirmada.
- **Terminação (*liveness*):** em circunstâncias normais, onde há uma transação em confirmação e transações concorrentes estão em situação de conflito, a transação em estado de confirmação sempre termina.
- **Terminação Justa (*fairness*):** uma transação que sofre um cancelamento não espontâneo é executada em completude, em no mínimo $f + 1$ e no máximo $2f + 1$ tentativas de execução.

Note que a diferença entre as propriedades de "terminação" e "terminação justa" reside no fato de que, na primeira, uma transação em execução pode sofrer cancelamentos não espontâneos, sempre que verificado conflito com uma transação em confirmação. Já na segunda, uma transação em execução tem seu término assegurado a despeito de conflitos, e do êxito ou não de sua execução, em um número finito de tentativas de execução.

4.3. Dinâmica de Funcionamento do Protocolo

O protocolo proposto adota uma estratégia adaptativa para o processamento de transações, a fim de obter a capacidade de assegurar a terminação justa das transações submetidas ao sistema. Neste sentido, o protocolo faz uso dos métodos pessimista e otimista de processamento de transações, de acordo com as condições verificadas no ambiente.



(a) Execução normal (otimista/dinâmica). (b) Execução justa (pessimista/pré-declarada).

Figura 2. Funcionamento básico do protocolo.

Primeiramente, a transação é executada de maneira tentativa (caso normal da Figura 2(a), onde o método otimista com escrita dinâmica (operações enviadas/processadas uma a uma, conforme a necessidade de interação dos clientes com as réplicas) é usado. Se porventura a transação vier a sofrer alguma falha ou conflito durante a execução otimista, em que a situação ocasione no seu cancelamento não espontâneo (o que pode se repetir indefinidas vezes), ela é novamente enviada para processamento por meio do protocolo de execução justa (Figura 2(b)). A execução justa é acionada para evitar que a transação fique indefinidamente em tentativa de processamento, e venha a sofrer por inanição. Neste caso, a transação é novamente submetida, mas por meio do método pessimista e com escrita pré-declarada, onde todas as operações são enviadas de uma única vez, no reinício da transação.

O propósito da execução justa é garantir a terminação de uma transação, de modo a evitar que sucessivas tentativas de processá-la causem a inanição da mesma, seja pelos conflitos verificados durante o processamento da transação ou pelo efeito transiente de faltas Bizantinas. No caso, o uso de escrita pré-declarada é proposital para evitar que um cliente com comportamento Bizantino possa manter uma transação em atividade sem nunca confirmá-la (o que é possível com escrita dinâmica), visando impedir o progresso das demais transações em execução normal ou justa. O uso de escrita pré-declarada é a única garantia de que a transação terá início e fim, já que todas as operações são enviadas de uma única vez, no momento do reinício da transação.

4.3.1. Gerenciamento e Execução de Transações

Nesta seção apresentamos a formalização dos algoritmos que compõem o protocolo proposto. Conforme especificado na Seção 4.1., a comunicação é confiável e autenticada, bem como o acesso aos objetos da base de dados é regulado por um mecanismo de controle de acesso discricionário. Todavia, para simplificar os códigos e facilitar o entendimento dos algoritmos por parte do leitor, os detalhes das operações de geração, verificação e controles criptográficos foram omitidas dos mesmos. Conforme descrito na Seção 4., nosso protocolo consiste na primeira e única proposta que resolve o problema da inanição em transações, mediante a faltas Bizantinas. Como os protocolos apresentados naquela seção são aptos a processar transações em bases de dados sujeitas a faltas Bizantinas, utilizamos como algoritmo para o processamento de transações em condições normais (Figura 2(a)) aquele apresentado no trabalho de Luiz et al. [Luiz et al. 2011]. Com isso, devido a restrições de espaço, nos limitados apenas em fazer um breve comentário a respeito de seu funcionamento, de modo que é facultada ao leitor a verificação destes na íntegra em [Luiz et al. 2011].

Em suma, o protocolo opera da seguinte maneira: quando um cliente tem a intenção de iniciar uma transação na base de dados, ele submete-a ao protocolo, que inicia a transação por meio do sub-protocolo de **execução normal** (Figura 2(a)). Ao iniciar a transação as réplicas definem dentre elas um líder, o qual irá realizar a execução da transação (passos 1 e 2 - Figura 2(a)). A partir dai, a interação do cliente com a base de dados ocorre de maneira otimista e dinâmica, isto é, a comunicação ocorre apenas com a réplica líder e as operações são enviadas de acordo com a necessidade (e vontade) do cliente (passos 3 a 6 - Figura 2(a)). Quando não há mais operações para uma transação, o cliente solicita a confirmação, por meio da difusão com ordem total de um pedido de confirmação à todas as réplicas (passo 7 - Figura 2(a)). Ao entregar esta mensagem, as réplicas não-líder passam a ter o conhecimento das operações que compõem a transação, que vão apensadas à mensagem que acabara de ser entregue. Neste ponto, a réplica líder

propaga às demais réplicas as operações por ela executadas para aquela transação, como forma de notificar que a transação enviada pelo cliente está em consonância com aquela executada de maneira otimista por ela (passo 8 - Figura 2(a)). A entrega desta última mensagem pelas réplicas, implica no inicio do processo de confirmação, em que as operações da transação são executas pelas réplicas não-líder, ou também pela líder, caso seja verificado que o cliente pediu a confirmação para uma transação diferente daquela executada por ela. Após estas verificações, a transação é submetida a um teste de certificação para determinar se ela cumpre os requisitos necessários à sua confirmação (i.e. serialização) e, ao passar pela certificação, a transação é aceita e se torna persistente na base de dados. Do contrário, se por alguma razão a transação não puder ser confirmada, ela é cancelada pelas réplicas, e é marcada localmente como "cancelada não espontaneamente" para indicar que é requerido o reprocessamento da mesma. No caso de um cancelamento unilateral por parte da réplica líder, esta envia uma mensagem de notificação de cancelamento às demais réplicas, por meio de difusão confiável, para que todas as réplicas tenham ciência de que a transação foi cancelada. Ao término do processo, as réplicas notificam o cliente quanto à situação da transação (confirmada ou cancelada), no passo 9 da Figura 2(a).

Variáveis:

1: $TICKET = \perp$ {Ticket a ser atribuído às transações (global)}
2: $ticket = 0$ {Ticket da última transação atendida (local)}
3: $last\_commited\_ticket = 0$ {Última transação confirmada pelo ticket}
4: $transaction\_data = \emptyset$ {Dados de controle das transações}
5: $commit\_data = \emptyset$ {Dados temporários para confirmação}
6: $\prod^{rst} = \emptyset$ {Conjunto que contém as transações reenviadas}
7: $\prod^{stv} = \emptyset$ {Conjunto que contém as transações suscetíveis à inanição}
8: $\prod^{act} = \emptyset$ {Conjunto que contém as transações atuamente em atividade}

upon: $TO\text{-}deliver(c_i, \langle RESTART, t_i, t_i^{tsb}, t_i^{ops}\rangle))$
9: **if** $\exists c_i \in \mathcal{C}$ **then**
10: **if** $((\exists t_i \in \prod^{stv}) \wedge (state(t_i) = aborted)) \wedge (check\_data(t_i, t_i^{tsb}) = true)$ **then**
11: $[\prod^{stv} \leftarrow \prod^{stv} \setminus \{t_i\}\,;\, \prod^{rst} \leftarrow \prod^{rst} \cup \{t_i\}\,]$
12: $TICKET \leftarrow TICKET + 1$ {Incremento do ticket global}
13: $ticket(t_i) \leftarrow TICKET$ {Ticket atribuído à transação t_i}
14: $state(t_i) \leftarrow active$
15: $transaction\_data \leftarrow transaction\_data \cup \{\langle ticket(t_i), t_j, t_i^{tsb}, t_i^{ops}\rangle\}$
16: $send(s_i, \langle ACTIVE, t_i, ticket, (ticket(t_i) - ticket)\rangle)\ to\ c_i$
17: **end if**
18: **else**
19: $discards\ the\ transaction$
20: **end if**

Figura 3. Algoritmo da fase de reinício de transações da Figura 2(b).

Por outro lado, quando a transação não puder ser concluída em sua execução normal, seja pelo cancelamento em detrimento de outras ou pela verificação de situação de falha, o protocolo executa os passos ilustrados na Figura 2(b). Neste caso, a transação passa a ser executada de maneira pessimista, e é aí que ocorre a adaptação do protocolo à condição do ambiente. A execução justa de transações, conforme apresentada na Figura 2(b), é realizada pelo sub-protocolo por meio de três algoritmos, sendo um para cada fase. Os algoritmos para as fases são apresentados nas **Figuras** 3, 4 e 5, respectivamente. A primeira fase do protocolo, que é o reinício da transação, é formalizada no algoritmo da **Figura** 3. Neste, é possível observar a existência de um *TICKET* global, que é o principal mecanismo usado para assegurar o atendimento em plenitude das transações. Para tanto, primeiramente o *TICKET* é inicializado por todas as réplicas do sistema como nulo (linha 1), e é incrementado a cada mensagem *RESTART* entregue pelo protocolo, o que indica o reenvio de uma transação. Também é usado um *ticket* local para o controle interno de atendimento de cada réplica (linha 2), pois durante o processamento da transação, não há qualquer tipo de sincronização entre as réplicas.

Assim, quando uma transação é novamente submetida após uma tentativa de execução mal sucedida, primeiramente as réplicas verificam se a transação está sendo soli-

citada por um cliente já conhecido no sistema (linha 9). Esta verificação é necessária para impedir que clientes Bizantinos que entram no sistema, tentem iniciar uma transação com privilégios/prioridade em relação às honestas. Também é realizada uma verificação se a transação já foi anteriormente executada e sofreu um cancelamento, bem como se os dados de controle da transação recebida são consonantes com os daquela cancelada. Se qualquer uma das verificações falhar, as réplicas simplesmente descartam o pedido de reinício para a transação. Por outro lado, se a transação é aceita, ele é incluída em um conjunto de transações em atividade prioritária (linha 11), e o protocolo atribui a ela um valor de *ticket*, que indica a ordem na qual a transação terá a prioridade para sua execução em completude (linhas 12 e 13). Por fim, as réplicas inserem os dados de controle da transação em uma estrutura de dados do tipo "tabela de dispersão" (*hashtable*), em que a chave de cada entrada é o próprio *ticket*. A finalização da fase se dá quando as réplicas enviam ao cliente a notificação de que a transação foi iniciada, além de uma previsibilidade para sua execução, isto é, quantas transações estão à sua frente aguardando a execução (linha 16).

task TRANSACTION_PROCESS:
1: **while** $true$ **do**
2: **wait until** $|transaction\_data| > 0$
3: $ticket \leftarrow ticket + 1$
4: $\langle ticket(t_i), t_i, t_i^{tsb}, t_i^{ops} \rangle \leftarrow get\_by\_ticket(transaction\_data, ticket)$
5: $request\_priority\_locks(\langle t_i^{ops}, ticket(t_i) \rangle))$
6: $\langle RS, WS \rangle \leftarrow get\_readwrite\_Sets(t_i)$
7: **for each** $t_k \in \prod^{act} : \{ticket(t_k) = \perp \wedge (RS(t_k) \cap WS \neq \emptyset \vee WS(t_k) \cap WS \neq \emptyset \vee WS(t_k) \cap RS \neq \emptyset)\}$ **do**
8: **if** $(state(t_k) = active \vee state(t_k) = preparing)$ **then**
9: $abort\_transaction(t_k)$
10: **else if** $state(t_k) = ready$ **then**
11: $undo\_transaction(t_k)$
12: $has\_redo(t_k) \leftarrow true$
13: **end if**
14: **end for**
15: **wait until** $get\_priority\_locks(ticket(t_i)) =$ **true**
16: **for each** $op_i \in t_i^{ops}$ **do**
17: $result \leftarrow execute(op_i)$
18: $t_i^{res} \leftarrow t_i^{res} \cup \{result\}$
19: **end for**
20: **if** $\exists$ **exception** $\in t_i^{res}$ **then**
21: $commit\_data \leftarrow commit\_data \cup \{\langle t_i, ticket, H(t_i^{ops}), H(t_i^{res}), RS, WS, \text{“}not - OK\text{“} \rangle\}$
22: **else**
23: $commit\_data \leftarrow commit\_data \cup \{\langle t_i, ticket, H(t_i^{ops}), H(t_i^{res}), RS, WS, \text{“}OK\text{“} \rangle\}$
24: **end if**
25: $state(t_i) \leftarrow preparing$
26: $\nexists$ **exception** $\in t_i^{res}$? $status \leftarrow \text{“}OK\text{“} : status \leftarrow \text{“}not - OK\text{“}$
27: $transaction\_data \leftarrow transaction\_data \setminus \{\langle ticket(t_i), t_j, t_i^{tsb}, t_i^{ops} \rangle\}$
28: $send(s_i, \langle END, t_i, t_i^{ops}, t_i^{res}, status \rangle)\ to\ p_i$
29: **end while**

Figura 4. Algoritmo da fase de execução de transações da Figura 2(b).

Após o reinício da transação e da obtenção de seu *ticket* para execução, a próxima etapa do protocolo consiste na entrada na fase de execução da transação, por parte das réplicas. A tarefa que realiza a execução justa das transações, baseado no *ticket* a ela atribuído, é formalizado no algoritmo da **Figura** 4. O algoritmo da Figura 4 é bastante simples, onde inicialmente a tarefa entra em um laço infinito, em razão da necessidade desta permanecer em execução enquanto a réplica estiver em atividade. É digno de nota, que o valor inicial do *ticket* local é 0 (linha 2 do algoritmo da Figura 3). A cada interação do laço, o algoritmo primeiramente verifica se há alguma transação pronta para ser executada com base na tabela de dispersão (linha 2), e caso houver, o valor do *ticket* para controle de atendimento local é incrementado em uma unidade, e então a tarefa recupera da tabela de dispersão os dados da transação, cuja chave de inserção é igual ao valor atribuído ao *ticket* (linhas 3 e 4).

O passo seguinte consiste na requisição dos bloqueios sobre os itens de dados a que a transação faz referência (linha 5), e na sequência são obtidos os conjuntos de leituras e de escritas da transação (RS e WS - linha 6), estes necessários para verificar possíveis con-

flitos que possam estar impedindo a aquisição dos bloqueios. A concessão dos bloqueios para as transações em fase de execução justa ocorre de forma prioritária, e com base no *ticket* obtido para a transação. Neste caso, se durante a aquisição dos bloqueios houver alguma transação em **execução normal** (i.e. sem um *ticket*), e que esteja impedindo a concessão dos bloqueios, esta transação é cancelada de forma compulsória (condição indicada na linha 7) para liberar os bloqueios e então permitir a concessão para a transação indicada pelo *ticket*. Não obstante, as transações canceladas não espontaneamente recebem um tratamento adequado, de acordo com o estado em que elas se encontram (linhas 8 a 13). Ao obter a concessão dos bloqueios, a transação inicia a execução das operações que compõem sua unidade atômica e, cada resultado obtido é armazenado em um *buffer* de retenção de resultados para uso posterior (laço entre as linhas 16 a 19).

É importante salientar, que o mecanismo de bloqueio adotado é aquele que opera em duas fases de maneira conservadora (*strict two-phase locking* ou 2PL), de modo que, após a aquisição dos bloqueios estes são mantidos pela transação até o seu término, a fim de evitar a preempção po interferências alheias. Embora este mecanismo seja essencialmente pessimista, por meio de seu uso é possível assegurar o término da transação, além de evitar a ocorrência de *deadlocks* envolvendo-a [Weikum and Vossen 2002]. Ao término da execução das operações da transação, é verificado se dentre as operações executadas, alguma delas produziu uma exceção ou um erro. Se a condição não for verificada, a transação é marcada como "*OK*" (linha 23), e do contrário é marcada como "*not - OK*" (linha 21). Em ambos os casos, a transação e seus dados de controle são inseridos em um *buffer* de dados temporários de confirmação, para uso posterior durante a fase de confirmação. Note que uma decisão pelo cancelamento da transação, poderia ser tomada com base na(s) exceção(ões) verificada(s). Todavia, como uma exceção também pode caracterizar uma falta Bizantina (p. ex. um *bug*) isolada em uma réplica, por esta razão, a decisão é deixada para o cliente na fase de confirmação. A fase de execução se encerra com o envio do resultado da transação, ao cliente responsável pela mesma (linha 28).

Por fim, o algoritmo para a última fase do protocolo que é a confirmação, é formalizado na **Figura** 5. Neste caso, diferente do que ocorre na execução normal, onde a confirmação, de fato, é enviada pela réplica líder da transação, aqui ela é enviada diretamente pelo cliente - em ambos os casos enviado por meio de difusão com ordem total. Com isso, quando as réplicas entregam uma mensagem de confirmação (i.e. *COMMIT*), o primeiro passo do algoritmo é a verificação acerca da identificação do cliente, pois um cliente faltoso recém-chegado ao sistema pode vir a tentar confirmar uma transação espúria, o que por conseguinte deve ser rejeitado. Se o cliente já é conhecido no sistema, indica que pelo menos uma transação dele já foi submetida para processamento, neste caso, resta verificar se os dados enviados na mensagem de confirmação são consistentes, o que é efetuado na linha 2. Na condição da linha 2 são verificadas duas questões, primeiro se a transação se encontra em atividade e se, além disso, a transação está no estado de preparação - único estado possível para se confirmar uma transação, conforme o autômato que define as transições de estado do protocolo [Luiz et al. 2011]. Se as condições forem satisfeitas, as réplicas verificam se os dados entregues junto à mensagem estão em consonância com os dados contidos no conjunto de informações temporárias de confirmação, inseridos na fase de execução (linha 3). E em sendo, o procedimento segue adiante, ou do contrário, a transação é descartada.

No procedimento de confirmação, o protocolo recupera os dados de controle da

transação, bem como os dados temporários de confirmação (linhas 4 e 5). Na sequência, é verificado se o *ticket* daquela transação é igual ao valor *ticket* da última transação atendida, acrescido em uma unidade (i.e. o próximo a ser atendido). Se o *ticket* da transação for o próximo da sequência, é verificado se naquela réplica a transação teve sua execução sem a ocorrência de exceções (linha 7), pois como já foi dito, uma exceção pode ser oriunda de uma réplica faltosa. Do contrário, se a transação não for a próxima ela é simplesmente descartada (linha 23). Caso não tenha havido nenhuma exceção, a transação é marcada para confirmação, ou do contrário, para cancelamento (linhas 8 e 10, respectivamente). Neste sentido, se a transação puder ser confirmada, a réplica então verifica se os dados enviados pelo cliente às réplicas no pedido de confirmação, são exatamente iguais àqueles que foram executados para a transação, na fase anterior. E se assim o for, a transação é confirmada e os dados tornam-se permanentes na base de dados (linhas 12 a 14). Por outro lado, caso a verificação não se confirme, a transação é cancelada e o efeito (ou estado) desta é descartado da base de dados (linhas 15 a 17). Por fim, as réplicas enviam ao cliente o resultado final da transação, liberam os bloqueios mantidos sobre os itens de dados e incrementam o *ticket* de controle de transações confirmadas em uma unidade (linhas 19 a 21). O cliente por sua vez, aceita o resultado da transação se recebe $f + 1$ respostas iguais de diferentes réplicas, o que indica que pelo menos uma réplica correta concluiu com êxito a transação.

upon: *TO-deliver*$(p_i, \langle COMMIT, t_j, ticket(t_j), \perp, \perp, H(t_j^{ops}), H(t_j^{res})\rangle)$

```
1:  if ∃p_i ∈ C then
2:      if ∃t_j ∈ ∏^act ∧ state(t_j) = preparing then
3:          if ⟨t_j, ticket(t_j), H(t_j^ops), H(t_j^res)⟩ ∈ commit_data then
4:              ⟨t_i, ticket(t_i), t_i^ops, t_i^res⟩ ← get_context(∏^act, t_j)
5:              ⟨status, RS, WS⟩ ← get_commit_data(commit_data, t_i)
6:              if ticket(t_i) = last_commited_ticket + 1 then
7:                  if status = "OK" then
8:                      can_commit ← true
9:                  else
10:                     can_commit ← false
11:                 end if
12:                 if can_commit = true ∧ matches(⟨H(t_i^ops), H(t_i^res)⟩, ⟨H(t_j^ops), H(t_j^res)⟩) then
13:                     [ outcome ← COMMITED ; T_i ← t_i ; ∏^c ← ∏^c ∪{T_i} ; ∏^act ← ∏^act \{t_i} ]
14:                     [ state(t_i) ← commited ; commit_transaction(T_k) ]
15:                 else
16:                     ∏^act ← ∏^act \{t_i}
17:                     [ outcome ← ABORTED ; abort_transaction(t_i) ]
18:                 end if
19:                 send(s_k, ⟨outcome, t_i⟩) to c_i
20:                 release_locks(t_i)
21:                 last_commited_ticket ← last_commited_ticket + 1
22:             else
23:                 discards the transaction until their turn comes
24:             end if
25:         end if
26:     end if
27: end if
```

Figura 5. Algoritmo da fase de confirmação de transações da Figura 2(b).

Note que um cliente faltoso pode vir a executar apenas as duas primeiras fases do protocolo de execução justa (reinício e execução), e nunca enviar o pedido de confirmação às réplicas, no intuito de fazer com que a transação mantenha os bloqueios sobre os itens de dados, e impedir que o sistema tenha progresso. Este problema é resolvido da seguinte maneira, após o envio do resultado do processamento da transação pelas réplicas ao cliente (ao término da fase de execução), é iniciado um temporizador que é dinamicamente ajustado de acordo com as condições verificadas no ambiente (p. ex. tempo de transmissão e recepção). Se este temporizador vier a se esgotar, as réplicas descartam a transação para liberar os bloqueios e assegurar a progressão do sistema. É importante ressaltar esta medida é adotada em decorrência do fato de que, o próprio modelo de transações não assegura a terminação destas em ambientes assíncronos.

5. Avaliação, Implementação e Resultados

Esta seção apresenta uma análise acerca do desempenho do protocolo proposto, de maneiras analítica e experimental. A análise se dá através da comparação dos custos associados ao processamento de uma transação, onde em particular, estamos interessados em observar a eficiência de nossa solução em relação aos trabalhos correlacionados, no que tange ao caso normal do protocolo, já que os demais não garantem a liberdade de inanição. No caso, a avaliação analítica é apresentada na Tabela 1, onde é realizada uma análise dos custos envolvidos no processamento de uma transação em situação *normal*, o que compreende às fases de início, execução e terminação, dos protocolos. O protocolo proposto neste trabalho aparece na Tabela 1 com o nome BFT-SF (de *Byzantine Fault-Tolerant - Starvation Free Transactions*), os demais protocolos são oriundos dos trabalhos relacionados (vide Seção 2.). Para o caso da latência, as equações ali apresentadas se referem ao número de mensagens requerido para todas as fases da transação, em que o termo s indica o número de operações executadas para a transação.

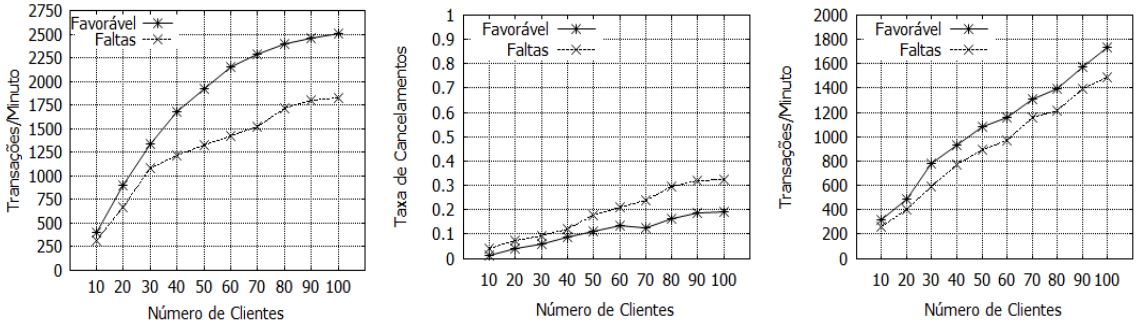
Tabela 1. Propriedades e custos associados ao processamento de uma transação.

Características e Propriedades	***Protocolos Avaliados***			
	BFT-SF	**HRDB**	**Byzantium**	**BFT-DU**
# Replicas	$3f + 1$	$2f + 1 + controlador$	$3f + 1$	$3f + 1$
Latência Normal	$s + 3(TOMCast) + 2$	$s + 4$	$s + 2(TOMCast) + 2$	$s + (TOMCast) + 1$
Consistência	Forte	Forte	Relaxada	Forte
Controle	Distribuído	Centralizado	Distribuído	Distribuído
Faltas	Réplicas e Clientes	Réplicas e Clientes	Réplicas e Clientes	Apenas Réplicas
Livre de inanição	Sim	Não	Não	Não
Latência Justa	$2(TOMCast) + 3$	–	–	–

Como se pode notar, o BFT-SF apresenta latência superior a todos os demais trabalhos. Isto advém da necessidade de se efetuar controles e verificações adicionais em relação aos demais protocolos, já que o BFT-SF é o único que permite o processamento confiável de transações de maneira totalmente distribuída, com o critério de consistência serializável, e sobretudo, com faltas oriundas de réplicas e clientes. Outro aspecto importante decorre do fato de que o BFT-SF é o único livre de inanição, e neste caso, o custo em termos de latência para o protocolo em execuções justas (i.e. latência justa) é inferior ao verificado para as execuções normais. Isto se explica pelo simples fato de que na execução justa, ao invés da transação ser executada de maneira dinâmica, ela é pré-declarada, o que implica na necessidade de apenas uma única mensagem para o envio de todas as operações da transação. Além disso, como na execução justa todas as réplicas executam a transação, não há a necessidade da réplica líder enviar sua afirmação para o pedido de confirmação do cliente, o que reduz em muito o número de mensagens do protocolo.

No intuito de analisar BFT-SF experimentalmente, foi implementado um protótipo para o mesmo. Assim, realizamos alguns experimentos em um ambiente composto por 14 máquinas HP 6005 *Pro Microtower* (AMD Phenom II^{TM}X4 3.2 GHz; 4GB RAM; Ethernet Gigabit), sendo todas conectadas em um ambiente de rede local por meio de um *switch* D-Link DGS-3100. Destas máquinas, 4 foram usadas para as réplicas e 10 para os clientes (i.e. com 10 clientes por máquina). O ambiente de software utilizado foi o Ubuntu Server 12.04.1 LTS, com a JVM Sun 1.6.0_29 sendo ativado o compilador Just-In-Time (JIT). Como SGBD, utilizamos o MySQL 5.5.8, sendo que a base de dados foi populada conforme o *benchmark* TPC-C (*http://www.tpc.org/tpcc*), que simula um ambiente de trabalho do tipo fornecedor por atacado. A escolha pelo TPC-C se deu não apenas para medirmos a sobrecarga, mas também para avaliar a escalabilidade do protocolo. Este *benchmark* produz uma carga de elevado nível de concorrência, por meio da mistura de transações

de intensivas atualizações com aquelas de somente-leitura, a fim de simular as atividades encontradas em ambientes transacionais complexos. Cabe ressaltar, que no ambiente do TPC-C predominam as transações de atualização (p. ex. com operações de leitura e escrita), o que compõe 92% da carga de trabalho do ambiente simulado, enquanto apenas 8% compreende a transações somente de leitura.



(a) Vazão (execução normal). (b) Cancelamentos (exec. normal). (c) Vazão (execução justa).

Figura 6. Desempenho verificado para o BFT-SF no *benchmark* TPC-C.

Visto que o BFT-SF é a única solução que previne a inanição, decidimos por não compará-lo com os trabalhos relacionados, já que a partir destes não é possível mostrar os resultados que se pretende avaliar. Assim, os experimentos consideraram apenas o BFT-SF em condições favoráveis (i.e. livres de faltas) e execuções com faltas, em processamento de maneiras normal e justa. A Figura 6(a) apresenta o resultado da vazão do protocolo de execuções normais, com e sem faltas. Note que a ocorrência de faltas incorre na degradação de aproximadamente 25% a 30% no desempenho de execuções normais, sendo que a taxa de cancelamentos não espontâneos é de aproximadamente 20% em condições favoráveis e de 30% em cenários com faltas (conforme a Figura 6(b)). Além disso, como o propósito do protocolo é evitar inanição de transações, avaliamos também a execução justa do protocolo. Esta avaliação foi realizada, a fim de verificar tanto a efetividade como o desempenho do mesmo, em condições favoráveis e com faltas. Estes resultados são reportados na Figura 6(c), onde se observa que o desempenho do protocolo com transações em execução justa é afetado em aproximadamente 30%, para ambos os cenários (com e sem faltas). Note que nesta situação, o protocolo continua a operar de forma normal para as transações concorrentes àquela em execução justa. Assim, se considerado o benefício que a execução justa oferece ao modelo de transações, o desempenho se torna bastante aceitável. É digno de nota que esta degradação já era prevista, justamente porque o processamento das transações em execução justa ocorre de maneira sequencial, a fim de evitar interferências à elas, para impedir o seu cancelamento não espontâneo (i.e. para preservar a execução da transação até sua conclusão). Não obstante, a despeito da execução justa, nesta situação a taxa de cancelamentos se mantém, pois as transações concorrentes em execução normal, quando em conflito, são canceladas para que as em execução justa possam ser concluídas.

6. Conclusão

Neste trabalho apresentamos um novo protocolo para o processamento e terminação justa de transações em ambientes sujeitos a faltas Bizantinas, o primeiro livre de inanição. A despeito do desempenho verificado, acreditamos que os benefícios associados com a possibilidade de evitar as transações de sofrer por inanição justificam os custos, se levado em consideração que as transações em situação normal podem executar infinitas vezes sem êxito algum, incorrendo em um custo mais elevado que o obtido. Também demonstramos,

que apesar do desempenho a eficiência do algoritmo de execução justa é superior a do algoritmo normal de processamento de transações, de todos os trabalhos. Além do mais, como a solução integra mecanismos alternativos para o controle de concorrência de transações, ela é de possível adaptação nos protocolos já existentes (i.e. dos trabalhos correlacionados), sem muita complexidade, bem como da necessidade de despender grande esforço.

Referências

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD'95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA. ACM.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *OSDI '99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Garcia, R., Rodrigues, R., and Preguiça, N. (2011). Efficient middleware for byzantine fault-tolerant database replication. In *Proceedings of the 6th European Conference on Computer Systems - EuroSys'11*. ACM.

Gashi, I., Popov, P. T., and Strigini, L. (2007). Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294.

Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A. P. (1994). Using tickets to enforce the serializability of multidatabase transactions. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):166–180.

Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA. ACM.

Hasse, C. and Weikum, G. (1997). Inter- and intra-transaction parallelism for combined OLTP/OLAP workloads. In Jajodia, S. and Kerschberg, L., editors, *Advanced Transaction Models and Architectures*, pages 279–302. Springer-Verlag.

Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226.

Luiz, A. F., Lung, L. C., and Correia, M. (2011). Protocolo tolerante a faltas bizantinas para bases de dados transacionais. In *Anais do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 559–572. SBC.

Molina, H. G., Pittelli, F., and Davidson, S. (1986). Applications of byzantine agreement in database systems. *ACM Transactions on Database Systems*, 11(1):27–47.

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*.

Pedone, F., Schiper, N., and Armendáriz-Iñigo, J. (2011). Byzantine fault-tolerant deferred update replication. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing - LADC'11*. SBC.

Tai, A. T. and Meyer, J. F. (1996). Performability management in distributed database systems: An adaptive concurrency control protocol. In *Proceedings of the 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '96, pages 212–216, Washington, DC, USA. IEEE Computer Society.

Vandiver, B., Balakrishnan, H., Liskov, B., and Madden, S. (2007). Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP'07: Proceedings of 21st ACM Symposium on Operating Systems Principles*.

Weikum, G. and Vossen, G. (2002). *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Zielinski, P. (2004). Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.