Protocolo Tolerante a Faltas Bizantinas para Bases de Dados Transacionais

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

[1]Departamento de Automação e Sistemas - PGEAS - UFSC

[2]Departamento de Informática e Estatística - PPGCC - UFSC

[3]LaSIGE - Faculdade de Ciências da Universidade de Lisboa

aldelir@das.ufsc.br, lau.lung@inf.ufsc.br, mpc@di.fc.ul.pt

Resumo. *A confirmação de transações é um problema bastante discutido na literatura, tanto em termos teóricos quanto práticos. Este artigo apresenta uma solução modular para o problema da confirmação de transações em ambientes sujeitos a faltas bizantinas, onde emprega-se a abstração de difusão com ordem total para concretização de um protocolo robusto. O protocolo apresentado é factível de implementação e, a despeito dos trabalhos correlatos, assegura a consistência forte e não requer controle centralizado.*

Abstract. *Transactions commit is a problem much discussed both in theoretical and systems research. This paper presents a modular approach to solve this problem on environments that are subject to Byzantine faults. Our protocol is safe and is built on top of total order broadcast abstraction. On the contrary of previous solutions in the literature, our solution assures strong consistency and does not need centralized control.*

1. Introdução

O conceito de transação foi introduzido no contexto de sistemas de bases de dados, como mecanismo para assegurar a execução consistente de um conjunto de operações, a despeito de concorrência e conflitos sobre dados compartilhados [Lampson 1981]. Não obstante, a noção de transações tem sido aplicada em diferentes cenários de aplicação (ex. telecomunicações, controle automático, comércio eletrônico), e especialmente em sistemas distribuídos, como forma de preservar a confiabilidade e consistência dos dados manipulados pelo sistema. Além disso, alguns autores têm estimulado o uso de abstrações comumente usadas na construção de sistemas distribuídos (p.ex. difusão atômica) como suporte a soluções para replicação de sistemas de transações [Schiper and Raynal 1996, Pedone et al. 2003], para incrementar a disponibilidade e a integridade dos dados a despeito de faltas de parada (*crash*).

Muitos são os desafios encontrados na proposição de protocolos de replicação para o modelo de transações. Dentre eles, os principais são: (*i*) assegurar a consistência das réplicas, de modo a prover um alto nível de concorrência; (*ii*) a estratégia a ser empregada para garantir a execução das transações na mesma ordem/sequência pelas réplicas, ou alguma ordem equivalente; e principalmente (*iii*) o protocolo para confirmação de transações, pois, uma vez que a transação atua sobre dados residentes em diversos sítios, o protocolo deve assegurar não somente a atomicidade local, mas também a global. A confirmação de transações é um problema bastante discutido na literatura, onde ele é referenciado como confirmação atômica de transações (do inglês *atomic commitment*). O problema da confirmação atômica pode ser visto como pertencente à classe de problemas de acordo em

sistemas distribuídos [Babaoğlu and Toueg 1993, Raynal and Singhal 2001], pois ele captura a essência do consenso devido à necessidade de um acordo entre os processos, o que o caracteriza com uma redução ao consenso. Esta é uma das razões pela qual o consenso é considerado um serviço fundamental, e pode ser usado como base para a construção de novos protocolos para sistemas confiáveis.

Na literatura, é encontrado um vasto conjunto de soluções para replicação de dados em sistemas transacionais, que são restritas apenas a faltas de parada [Gray et al. 1996, Agrawal et al. 1997, Pedone et al. 2003]. Diante disso, é verificada a carência de propostas que toleram outros modelos de falhas, no contexto de transações. Esta carência é mencionada em um trabalho recente, onde os autores realizam uma completa caracterização de falhas em sistemas de transações [Gashi et al. 2007], e evidenciam que estes são potencialmente suscetíveis a erros e *bugs*, que manifestam faltas de natureza arbitrária. De outro lado, as soluções usuais para replicação tolerante a faltas bizantinas (*Byzantine Fault-Tolerance*) [Castro and Liskov 1999, Kotla et al. 2007] não são totalmente adequadas para o modelo de transações, em razão dos requisitos estipulados para o modelo de replicação de máquina de estados (ex. acordo e ordem), onde todas as requisições submetidas ao sistema devem ser previamente ordenadas antes de serem executadas, e a execução das requisições tem de ocorrer de forma sequencial (uma após a outra). Adaptar este requisito para o modelo de transações não é uma tarefa trivial, pois uma característica comum do modelo é as instruções não serem conhecidas *a priori*, no início da transação (elas são dinâmicas).

Por outro lado, a definição do problema de confirmação atômica desconsidera os aspectos atinentes à execução e ao processamento da transação, mas apenas se concentra em como obter a confirmação a despeito da ocorrência de faltas [Babaoğlu and Toueg 1993, Raynal and Singhal 2001]. De fato, o problema considera apenas os aspectos relacionados ao acordo/decisão entre os processos, porém, para que a decisão seja favorável à confirmação de uma transação pelo protocolo de confirmação atômica, é imperioso que a execução das operações desta transação tenha ocorrido em sua plenitude. Esta questão tem relação direta ao problema de acordo definido para a confirmação atômica, mas é desprezada pelos trabalhos especificados para o problema [Babaoğlu and Toueg 1993, Raynal and Singhal 2001]. Neste sentido, verifica-se que é necessário investigar por soluções que permitam preservar não só a consistência, que é o principal requisito de transações, mas também a integridade de dados, a despeito de faltas arbitrárias.

Assim, este artigo apresenta uma solução para suportar faltas bizantinas em sistemas de processamento de transações. Para tanto, o trabalho se baseia na abordagem de replicação de máquina de estados para bases de dados (RMEBD) [Pedone et al. 2003], onde o protocolo proposto é resultado da aplicação de uma abordagem modular, que explora o uso de difusão com ordem total [Défago et al. 2004] em conjunto com mecanismos e ferramentas criptográficas para a consolidação de uma solução factível que visa a consistência, integridade e segurança das transações submetidas ao sistema. A principal contribuição do artigo é um protocolo para execução e confirmação de transações tolerante a faltas bizantinas, que assegura a consistência ideal e forte (ex. serialização) e não requer controle centralizado. Até onde sabemos, existem apenas dois trabalhos anteriores que suportam transações no modelo de faltas bizantinas. No entanto, o primeiro deles usa um controlador centralizado que não pode falhar de forma bizantina [Vandiver et al. 2007] e o segundo provê a semântica de consistência *snapshot isolation*, que não assegura uma completa serialização das transações (um requisito fundamental para preservar a integridade) [Garcia et al. 2011]. Desta forma, o trabalho deste artigo é um avanço significativo em relação a esses dois trabalhos. O restante do

artigo está organizado da seguinte forma. A seção 2. descreve os trabalhos correlatos com a proposta em questão. Em seguida, na seção 3. apresentamos os detalhes e conceitos que amparam a solução proposta. A seção 4. avalia o protocolo. Por fim, a seção 5. descreve as considerações em relação à solução apresentada.

2. Trabalhos Relacionados

Na literatura encontramos alguns trabalhos com relação direta à proposta apresentada neste artigo [Agrawal et al. 1997, Pedone et al. 2003, Vandiver et al. 2007, Garcia et al. 2011, Molina et al. 1986]. O principal objetivo destes trabalhos é, sobretudo, assegurar critério de consistência requerido pelo modelo de replicação de transações, conhecido na literatura como 1-SR (ou *one-copy serializability*) [Bernstein et al. 1987]. O trabalho de [Agrawal et al. 1997] foi o pioneiro na exploração de primitivas de difusão, como suporte para a replicação em sistemas de transações no modelo de faltas de parada. O objetivo do trabalho era empregar primitivas de difusão com ordem total (*atomic broadcast*) [Défago et al. 2004] para simplificar o gerenciamento de replicação, prover maior garantia de consistência, e reduzir a probabilidade de *deadlocks*. O trabalho de [Pedone et al. 2003] também especificou um modelo para replicação de bases de dados transacionais, através do uso de primitivas de difusão com ordem total. Neste trabalho, a primitiva de difusão com ordem total foi usada em substituição ao protocolo de confirmação atômica, além de prover o suporte à sincronização e manutenção da consistência das réplicas. Em [Pedone et al. 2003] a terminação de transações está condicionada a execução de um teste de certificação distribuído, de modo que o critério de consistência 1-SR é verificado durante a confirmação da transação. Esta certificação estabelece que as transações, se confirmadas, cumpram os critérios de serialização.

Os trabalhos mencionados até então são restritos apenas ao modelo de faltas de parada. Assim, a primeira tentativa de tratar faltas de natureza arbitrária em sistemas de transações foi proposta por [Molina et al. 1986]. O trabalho, de caráter seminal, discorreu apenas sobre a integridade e consistência das respostas, descartando aspectos atinentes ao controle de concorrência de transações, além de não ter apresentado indícios quanto à implementação da proposta. Um estudo mais recente [Gashi et al. 2007] discorreu sobre a caracterização de faltas bizantinas em sistemas de transações. O trabalho realizou a verificação de uma série de *bugs* que manifestaram faltas arbitrárias. Para isolar as faltas, foi proposta uma solução de *middleware*, para submeter as transações às réplicas e determinar a consistência dos resultados por meio do voto majoritário. Infelizmente, este trabalho não permite a execução concorrente de transações, sendo, portanto, não factível em ambientes reais. Diante disso, dois trabalhos recentes trataram os aspectos de concorrência de transações e consistência de resultados em ambientes sujeitos à faltas bizantinas, são eles HRDB [Vandiver et al. 2007] e Byzantium [Garcia et al. 2011]. Os dois trabalhos são baseados na replicação de bases de dados, porém, ambos apresentam fraquezas em pontos diferentes. O HRDB define um protocolo de escalonamento que assegura a consistência forte e gerencia a execução concorrente de transações. A fraqueza do HRDB está na sua dependência por um coordenador centralizado (e confiável), que é o responsável por determinar e resolver as situações de conflito entre as transações, de modo a evitar a ocorrência de *deadlocks*. Apesar de a solução ser interessante, na prática é difícil justificar o pressuposto de uma entidade confiável em um ambiente sujeito a faltas bizantinas. Por outro lado, o Byzantium define um protocolo de suporte a execução de transações em ambientes sujeitos a faltas bizantinas, que relaxa a consistência dos dados, visando obter maior grau de concorrência na execução de transações. Contudo, a semântica de consistência adotada no Byzantium (o *snapshot isolation*) não é equivalente à execução sequencial de transações, tal como ocorre no modelo de serialização. Além disso, há evidências

de que essa semântica é suscetível a anomalias, que afetam potencialmente a integridade dos dados [Berenson et al. 1995]. Desta forma, em um ambiente sujeito a faltas bizantinas isso se torna algo ainda pior, pois um adversário que toma o conhecimento desta fraqueza pode introduzir uma série de conflitos no sistema, de modo a causar a corrupção do mesmo.

3. Um Protocolo para Transações Tolerante a Faltas Bizantinas

Conforme mencionado, é verificada na literatura uma carência de soluções com foco à execução e processamento de transações, a despeito de faltas bizantinas. Neste ensejo, o propósito deste trabalho é a proposição de um protocolo de suporte ao processamento de transações tolerante a faltas bizantinas, baseado em replicação. Este trabalho pode ser visto como a concretização do modelo RMEBD [Pedone et al. 2003] sobre o modelo de faltas bizantinas [Lamport et al. 1982], já que definição original do modelo considera apenas faltas de parada. A abordagem RMEBD tem se mostrado uma alternativa factível para a implementação de sistemas replicados no modelo de transações, por algumas razões: (*i*) ela prevê a execução das transações de forma otimista pelas réplicas; (*ii*) apesar de replicada, não há a necessidade de concessão de bloqueios distribuídos para os dados manipulados; (*iii*) permite obter um nível de consistência forte, de maneira distribuída; e, sobretudo, (*iv*) usa um protocolo de difusão atômica em substituição ao clássico protocolo de confirmação atômica, geralmente empregado em sistemas de transações replicados. Além disso, o desempenho verificado para a abordagem RMEDB, no modelo de faltas de para é bastante satisfatório [Pedone et al. 2003].

O primeiro e principal objetivo do protocolo é preservar as propriedades básicas de transações, propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade), de modo que o sistema se mantenha correto e consistente, desde que até $f \leq \lfloor \frac{n-1}{3} \rfloor$ réplicas se desviem arbitrariamente de suas especificações. O segundo objetivo é prover escalabilidade através da aplicação da abordagem otimista/especulativa [Kung and Robinson 1981, Lampson 2006] na execução das operações das transações. Entretanto, é lícito ressaltar que o grande problema do gerenciamento de transações advém do fato da concorrência e consistência rumarem em sentidos opostos, isto é, quanto mais elevado é o nível de consistência mais penalizado é o grau de concorrência (e vice-versa) [Berenson et al. 1995]. Diante disso, o modelo de propagação adotado em nosso protocolo é o mesmo usado na abordagem RMEDB, o de "adiamento de atualizações" (*deferred update*) [Bernstein et al. 1987]. Este modelo preconiza que a interação do cliente durante o curso de uma transação ocorra apenas com uma réplica líder, e a propagação das operações juntamente ao *Read-Set* (RS) e *Write-Set* (WS)[1] para as demais réplicas ocorra somente na confirmação da transação.

3.1. Modelo de Sistema

O modelo de sistema adotado consiste em um ambiente de base de dados replicado, onde os processos estão divididos em dois conjuntos $C = \{c_1, c_2, ..., c_n\}$ e $R = \{r_1, r_2, ..., r_n\}$ que correspondem aos clientes e réplicas, respectivamente. O conjunto C contém um número arbitrário (não infinito) de clientes, e o conjunto R, por sua vez tem cardinalidade $|R| \geq 3f+1$. Deste modo, o modelo admite que um número ilimitado de clientes e até $f \leq \lfloor \frac{n-1}{3} \rfloor$ réplicas estão sujeitos a faltas bizantinas. Os processos faltosos podem desviar arbitrariamente de suas especificações, podendo parar, omitir o envio, a recepção ou a entrega das mensagens, enviar respostas incorretas às operações dos clientes, bem como atuar em conluio com outros processos visando a corrupção do sistema. Todavia, assumimos a independência de falhas por

[1]RS e WS são os conjuntos de itens de dados afetados pelas operações de leitura e escrita da transação, respectivamente.

meio do uso de diversidade [Obelheiro et al. 2005]. Assim, a ocorrência de uma falta em uma determinada réplica é independente da ocorrência daquela falta nas demais.

As transações não têm assegurada sua terminação em ambientes assíncronos [Bernstein et al. 1987], logo, o modelo de interação adotado é o de sincronismo terminal (*eventually synchronous system*) [Dwork et al. 1988]. Este modelo de interação é bastante realista, pois assume que o sistema se porta de forma assíncrona em grande parte do tempo, mas que durante períodos de estabilidade o tempo de transmissão de mensagens e das computações é limitado, porém desconhecido.

A despeito da diversidade, as réplicas são deterministas, pois todas suportam o mesmo subconjunto de operações de manipulação de dados (ex. SQL ANSI). Logo, o resultado da execução de uma operação é o mesmo em todas as réplicas. Com isso, assumimos que os clientes submetem apenas operações pertencentes a esse subconjunto. Em nosso modelo, um cliente submete apenas uma transação por vez, e no contexto de uma transação uma operação é enviada apenas se não há operações pendentes de execução. Todas as comunicações entre os processos ocorrem por meio de canais ponto-a-ponto confiáveis e autenticados, e com ordenação FIFO. Por fim, convém salientar que o protocolo de difusão com ordem total usado no modelo é baseado no algoritmo PAXOS Bizantino [Zielinski 2004].

3.2. Princípio de Funcionamento do Protocolo

Em nosso protocolo, cada transação consiste em um bloco de operações de manipulação de dados (leitura e/ou escrita), que é iniciado pelo cliente através da instrução *BEGIN* e finalizado pela instrução *COMMIT* ou *ABORT*. A especificação do protocolo considera as noções de correção (*safety*) e vivacidade (*liveness*), com observância às seguintes propriedades:

- **Consistência (*safety*):** o protocolo provê um sistema consistente e equivalente-centralizado, e satisfaz o critério de consistência 1-SR (*one-copy serializability*).
- **Terminação (*liveness*):** em uma confirmação onde há transações em situação de conflito e/ou pendentes de execução, a transação em confirmação sempre termina.

Os requisitos básicos da abordagem de Replicação de Máquina de Estados (RME) [Schneider 1990], que estipulam a ordenação de todas as operações ante a sua execução, bem como a execução sequencial das operações é restritivo para o modelo de transações. Estes requisitos em conjunto ao uso de bloqueios, que são usados como mecanismo de controle de concorrência no modelo de transações, impede que a RME seja factível neste modelo, pois na hipótese da execução de operações conflitantes, uma operação que é bloqueada pelo mecanismo pode causar o bloqueio de todo o sistema. Diante disso, e considerando que no modelo de transações uma transação toma efeito no sistema somente após sua confirmação, a técnica de propagação de transações mais adequada para este trabalho é aquela baseada no modelo *deferred update* [Bernstein et al. 1987]. Esta técnica visa à execução otimista/especulativa das operações da transação em apenas uma única réplica (a líder), de tal forma que ordenação total seja necessária apenas para o processo de confirmação da transação, para que esta ocorra em todas as réplicas na mesma ordem. Além disso, nesta técnica a propagação das operações executadas de forma especulativa ocorre apenas no momento da confirmação da transação.

O uso de difusão com ordem total permite o envio de mensagens ao um conjunto de processos, com a garantia de que todos os processos corretos irão acordar sobre as mensagens a serem entregues, bem como a ordem na qual elas serão entregues [Défago et al. 2004]. Formalmente, a difusão com ordem total assegura que: (*i*) se algum processo entrega a mensagem

m, então todos os processos corretos entregam **m** (atomicidade); (*ii*) todos os processos corretos entregam as mensagens na mesma ordem (ordem); (*iii*) os processos corretos entregam a mensagem **m**, apenas se **m** foi previamente difundida (integridade); (*iv*) todos os processos corretos terminar por entregar **m** (terminação). É importante notar que as propriedades da difusão com ordem total são definidas em termos de entrega ao protocolo superior e não recepção de mensagens na máquina.

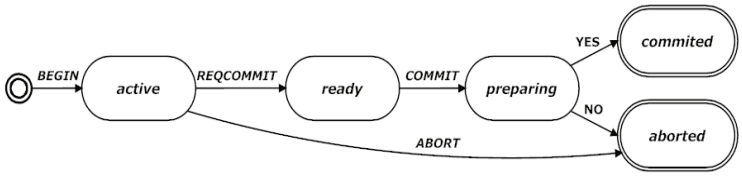


Figura 1. Estados da transação.

O funcionamento do protocolo se dá por meio do mapeamento das transações para uma sequência finita de estados, conforme a figura 1. As transições de estados são disparadas de acordo com as mensagens entregues, e eventos computados. O caso normal para uma transação é o seguinte. A transação é iniciada quando o cliente executa a operação *BEGIN*, onde a transação parte de um estado inicial para o estado *active*. Enquanto no estado *active*, apenas a réplica líder recebe e executa as operações que compõe a unidade lógica da transação. Após a execução de todas as operações da transação, o cliente solicita a confirmação da transação, que é efetuada através do envio da mensagem *REQCOMMIT* por meio de difusão com ordem total. Ao entregar a mensagem *REQCOMMIT* todas as réplicas passam a ter conhecimento das operações executadas na transação, e assim alteram o estado da mesma para *ready*. Neste momento, a réplica líder também propaga as operações executadas na transação ao grupo de réplicas, para confirmar o pedido enviado pelo cliente. A propagação é realizada no envio da mensagem de confirmação *COMMIT* via difusão com ordem total, onde a entrega desta mensagem às réplicas implica na transição para o estado *preparing*. Quando as réplicas vão para o estado *preparing* é iniciado o processo de confirmação, onde a transação passa por um teste de certificação para determinar se ela cumpre os critérios de serialização (ex. consistência serializável) e, em sendo, a transação vai para o estado *commited* e torna-se persistente na base de dados. Do contrário, a transação vai para o estado *aborted* e não toma efeito na base de dados. Além disso, em situações onde a transação está em execução (no estado *active*) e o cliente solicita o cancelamento por motivos particulares, a transação passa diretamente para o estado *aborted*. É importante notar que as operações *BEGIN*, *REQCOMMIT*, *COMMIT* e *ABORT* serão executadas por todas as réplicas, na mesma sequencia/ordem equivalente, uma vez que as mensagens para estas operações são difundidas no sistema através do protocolo de difusão com ordem total.

3.3. Protocolo de Início, Execução e Pré-Confirmação

Como assumimos que todas as mensagens são assinadas e autenticadas (vide seção 3.1.), logo, as réplicas entregam e recebem apenas as mensagens oriundas de clientes autenticados. Os acesso aos objetos contidos na base de dados são regulados por um mecanismo de controle de acesso discricionário, por exemplo, baseado em listas de controle de acesso [Amoroso 1994]. Conforme estipulado no modelo de sistema, os canais são confiáveis e com ordem FIFO, desta forma, as instruções de uma transação são executadas pela réplica líder exatamente na ordem em que foram submetidas pelo cliente. Para efeito de simplificação, o protocolo executado pelas réplicas é descrito em duas partes, nas figuras 2 e 3. A formalização da parte do protocolo que trata do início (linhas 22-27), da execução otimista das operações (linhas 28-41) e do pedido de confirmação (linhas 12-21) para as transações é apresentada na figura 2. A interação da parte do cliente foi omitida por restrições de espaço.

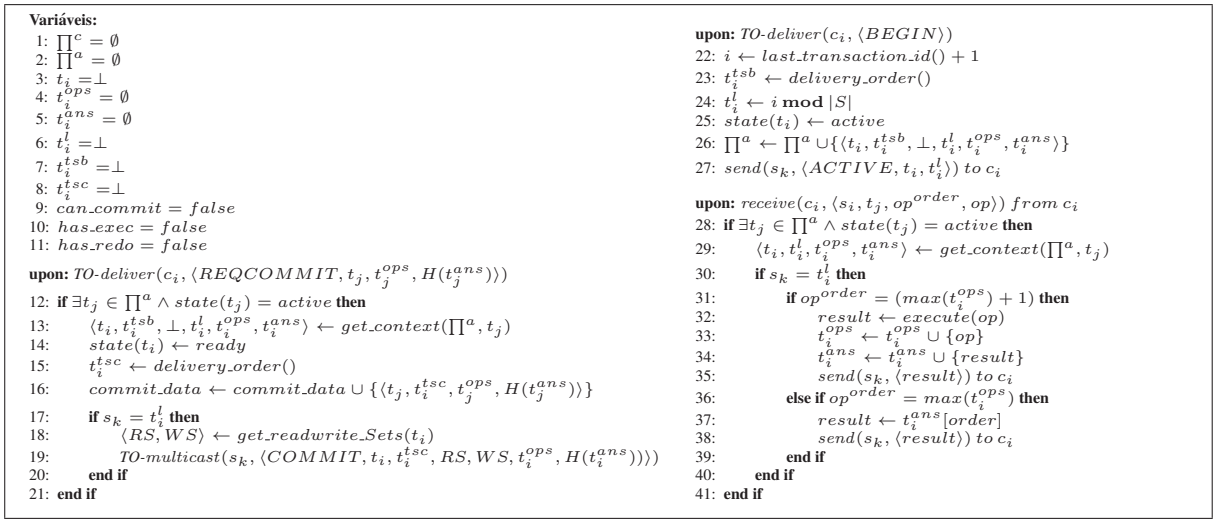


Figura 2. Protocolo de início, execução e pré-confirmação de transações.

Para facilitar a compreensão do protocolo da figura 2, consideremos um cliente c_i que inicia a transação t_j, tendo como líder a réplica s_k. A transação é iniciada quando as réplicas entregam a mensagem *BEGIN*, oriunda do cliente c_i (linhas 22-27). Na entrega, primeiramente atribui-se como identificador da transação (linha 22), o valor do id da última transação iniciada acrescido de uma unidade (a função *last_transaction_id()* retorna o id da última transação iniciada). Em seguida a réplica atribui o número da ordem de entrega da mensagem como *timestamp* de início da transação (a função *delivery_order()* retorna o número de sequência da mensagem entregue). O passo da linha 24 determina a réplica líder da transação, sob a qual ficará a responsabilidade de executar as operações da transação. A partir daí as réplicas definem o estado da transação como *active*, incluem as variáveis inicializadas nos passos anteriores em um conjunto de informações das transações ativas ($\prod^a$), e enviam a confirmação ao cliente (linhas 25, 26 e 27, respectivamente). O cliente aceita a transação, se constatar o recebimento de pelo menos $f + 1$ confirmações de diferentes réplicas, a partir da qual tomará o conhecimento do líder da transação (identificador t_i^l).

Uma vez iniciada a transação t_j, o cliente pode executar um sequência de operações de leitura e de escrita, o que é realizado nas réplicas por meio das linhas 28-41. Neste ponto, a interação do cliente ocorre apenas com a réplica líder (s_k), que ao receber uma operação verifica se t_j é válida e se o estado é *active*, pois é o único estado em que a transação pode receber operações. Uma transação é considerada válida se ela foi previamente iniciada. Se estas verificações não forem bem sucedidas, a operação recebida é descartada. Do contrário, a réplica recupera as variáveis que contém os dados de t_j a partir do conjunto de transações ativas ($\prod^a$ - a função *get_context()* recupera as variáveis do conjunto de transações ativas, com base no id da transação) e verifica se ela é de fato, a líder de t_j (linha 30), pois um cliente malicioso pode enviar operações espúrias às replicas incorretas. Adiante, a réplica s_k verifica se a operação recebida segue o critério estabelecido pelo protocolo, isto é, se o id da operação é uma unidade maior que o da última operação executada, e, em sendo a réplica s_k executa a operação, guarda o(s) comando(s) executado(s) e o(s) resultado(s) nos *buffers* de retenção (t_i^{ops} e t_i^{ans}) para uso futuro, e então envia a resposta ao cliente (linhas 31-35). Caso o id da operação seja igual ao da última operação executada, a réplica trata como um pedido de retransmissão para o resultado da última operação, e então reenvia a resposta ao cliente (linhas 36-39).

O protocolo entre as linhas 12-21 trata da entrega do pedido de confirmação para a

transação t_j, o qual é enviado pelo cliente através da mensagem *REQCOMMIT*, via difusão com ordem total. Ao entregar a mensagem *REQCOMMIT*, as réplicas verificam se t_j é válida e se encontra no estado *active*, pois são as únicas condições lícitas para a execução deste procedimento. Em seguida, as réplicas recuperam as variáveis de t_j a partir do conjunto de transações ativas e a transação evolui para o estado *ready* (linhas 13 e 14). A partir deste ponto, t_j não está mais apta a receber e executar operações, mas apenas a aguardar pela confirmação. No próximo passo, as réplicas marcam o *timestamp* (a ordem de entrega da mensagem) do momento do pedido de confirmação, para que seja possível determinar quais transações precedem t_j. Adiante, as réplicas guardam os conteúdos das variáveis de controle da transação no conjunto $commit\_data$, para futuramente confrontar com as informações enviadas pelo líder, para a confirmação da transação. Os passos das linhas 17-20 são executados apenas pela réplica s_k, onde ela obtém os itens de dados afetados pelas operações de leitura e de escrita (RS e WS) da transação t_j, e anexa-os as informações recebidas na mensagem *COMMIT* para difusão às demais réplicas. Este é o momento onde ocorre a propagação da transação às demais réplicas do grupo. A parte que trata da confirmação será discutida na próxima seção.

3.4. Protocolo de Confirmação de Transações

A formalização do protocolo de confirmação e terminação de transações é descrita na figura 3, e, portanto, é a parte mais importante do protocolo. Esta parte do protocolo é executada por todas as réplicas, no momento em que elas entregam a mensagem de confirmação, juntamente com as operações da transação. Conforme mencionado na seção 3.2., tanto o pedido de confirmação como a própria confirmação ocorrem de forma sequencial, e na mesma ordem em cada uma das réplicas. Em nosso protocolo a conclusão de uma transação ocorre quando o cliente envia o pedido de confirmação, que ao ser entregue pelas réplicas é difundido ao grupo, junto à mensagem de confirmação (*COMMIT*) enviada pela réplica líder. O propósito desta parte do protocolo é assegurar a terminação da transação, em consonância ao critério de correção 1-SR. A mensagem que contém a confirmação de uma transação traz consigo o identificador da transação para a qual está sendo solicitada a confirmação (t_j), as operações executadas (t_j^{ops}), o resumo criptográfico dos resultados obtidos do líder para cada uma das operações ($H(t_j^{ans})$), os itens de dados afetados pela transação (RS e WS) e o *timestamp* marcado no momento da entrega do pedido de confirmação (t_j^{tsc}).

Ao entregar a mensagem *COMMIT* para a transação t_j, as réplicas primeiramente verificam se esta é válida e se a mesma encontra-se no estado *ready*, pois, conforme descrito na seção 3.2., é o único estado lícito a receber/entregar a confirmação de uma transação. A verificação da linha 2 é realizada no intuito de confrontar se as informações que estão sendo entregues são consonantes àquelas entregues no pedido de confirmação (mensagem *REQCOMMIT*). Essa verificação permite identificar pedidos de confirmação para transações espúrias criados por réplicas faltosas, ou ainda, pedidos de confirmação que foram alterados por um líder faltoso. A partir daí a transação t_j evolui para o estado *preparing*, onde ela passa a adquirir prioridade na concessão de bloqueios sobre as demais transações em execução. O indicador da linha 5 será usado e explicado adiante. Os passos das linhas 6-13 são executados apenas pela réplica líder da transação. Ali é checado se o indicador de re-execução foi ligado (linha 7) devido a t_j, quando no estado *ready* ter sido desfeita por uma transação que a precedeu, no intuito de liberar os bloqueios que a impediam de fazer progresso (linha 23). Neste caso, a réplica líder desfaz todo o efeito das operações executas (linha 8) e liga o indicador de execução (linha 9), para processar novamente as operações da transação antes de efetuar a confirmação.

Os passos entre as linhas 16 e 36 tratam da execução das operações da transação pelas

```
upon: TO-deliver(s_k, ⟨COMMIT, t_j, t_j^tsc, RS, WS, t_j^ops, H(t_j^ans)⟩)
 1: if ∃t_j ∈ ∏^a ∧ state(t_j) = ready then
 2:     if ⟨t_j, t_j^tsc, t_j^ops, H(t_j^ans)⟩ ∈ commit_data then
 3:         ⟨t_i, i, t_i^r, t_i^l, t_i^ops, t_i^ans⟩ ← get_context(∏^a, t_j)
 4:         state(t_i) ← preparing
 5:         can_certify ← true
 6:         if t_i^l = s_k then
 7:             if has_redo = true then
 8:                 undo_transaction(t_i)
 9:                 has_exec ← true
10:             else
11:                 has_exec ← false
12:             end if
13:         else
14:             has_exec ← true
15:         end if
16:         if has_exec = true then
17:             t_i^ops ←⊥
18:             request_locks(⟨RS, WS, t_j^ops⟩, delivery_order())
19:             for all t_k ∈ ∏^a : {RS(t_k) ∩ WS ≠ ∅ ∨ WS(t_k) ∩ WS ≠ ∅} do
20:                 if state(t_k) = active then
21:                     abort_transaction(t_k)
22:                 else if state(t_k) = ready then
23:                     [ undo_transaction(t_k) ; has_redo(t_k) ← true ]
24:                 end if
25:                 ∏^a ← ∏^a \{t_k}
26:             end for
27:             if acquire_locks() = true then
28:                 for all op_j ∈ t_j^ops do
29:                     result ← execute(op_j)
30:                     [ t_i^ops ← t_i^ops ∪ {op_j} ; t_i^ans ← t_i^ans ∪ {result} ]
31:                     t_j^ops ← t_j^ops \ {op_j}
32:                 end for
33:             else
34:                 can_certify ← false
35:             end if
36:         end if
37:         if (¬(∃T_j ∈ ∏^c : {T_j ⇝ t_i ∧ (WS(T_j) ∩ RS ≠ ∅)}) ∧ (can_certify = true) then
38:             can_commit ← true
39:         else
40:             can_commit ← false
41:         end if
42:         if can_commit = true ∧ matches(⟨H(t_i^ops), H(t_i^ans)⟩, ⟨H(t_j^ops), H(t_j^ans)⟩) then
43:             [ outcome ← COMMITED ; T_i ← t_i ; ∏^c ← ∏^c ∪{T_i} ; ∏^a ← ∏^a \{t_i} ]
44:             [ state(t_i) ← commited ; commit_transaction(T_k) ]
45:         else
46:             ∏^a ← ∏^a \{t_i}
47:             [ outcome ← ABORTED ; abort_transaction(t_i) ]
48:         end if
49:         send(s_k, ⟨outcome, t_i⟩) to c_i
50:     else
51:         ∏^a ← ∏^a \{t_i}
52:         [ state(t_j) ← aborted ; abort_transaction(t_j) ]
53:         send(s_k, ⟨ABORTED, t_i⟩) to c_i
54:     end if
55: end if
```

Figura 3. Protocolo de confirmação de transações.

réplicas, exceto a líder que já as executou de forma especulativa, conforme ilustrado nas linhas 28-41 do protocolo da figura 2. A réplica líder estará apta a executar as operações apenas se for verificada a condição da linha 7. No passo seguinte a réplica inicializa a lista de operações para execução, e solicita na linha 18, a concessão dos bloqueios para os itens de dados afetados pela transação (RS e WS) e entregues junto ao pedido de confirmação, bem como para as operações recebidas (t_j^{ops}). A concessão dos bloqueios é propositalmente efetuada sobre as operações e itens de dados, para verificar a consistência das operações com os respectivos itens, pois como as operações de sistemas de transações são, em geral, baseadas em predicados, é possível que uma entidade faltosa envie operações que afetam mais itens de dados do que aqueles contidos em RS e WS. Desta forma, a concessão dos bloqueios terá êxito apenas se as operações e os RS e WS forem consonantes. Nos passos seguintes (linhas 19-26), a réplica verifica se existe alguma transação em execução local, que detém a concessão dos bloqueios para algum dos itens de dados afetados pela transação em confirmação, e está impedindo a concessão dos bloqueios devido ao conflito (condição da linha 19). Caso esta condição seja verificada, a transação em execução (não a que está em confirmação) é: (*i*) cancelada, se o estado desta é *active*; (*ii*) desfeita e sinalizada para re-execução, se o estado desta é *ready* (neste ponto não é possível cancelar a transação, pois o cliente já solicitou a confirmação). Em seguida, se a transação puder obter os bloqueios ela inicia a execução das operações recebidas

na confirmação de t_j (linhas 27-32). De outro modo, se a concessão dos bloqueios não puder ocorrer pelas razões já expostas o indicador de obrigatoriedade de certificação será desligado (linha 34), e a transação será cancelada.

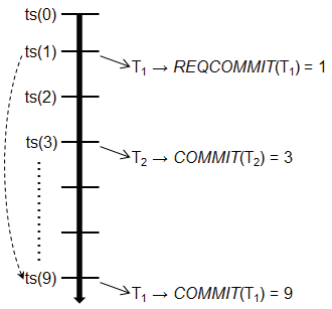


Figura 4. Relação de precedência entre transações.

Uma vez que os bloqueios foram concedidos e as operações executadas, as réplicas devem certificar a transação para que ela possa ser confirmada. Esta certificação determinará se a transação cumpre os critérios definidos para a serialização (condição da linha 37), que tem como base a verificação da existência e conflitos dos itens afetados pela transação em questão, em relação às transações já confirmadas, mas que não a precedem. Uma transação t_i não precede t_j se ela foi confirmada após a entrega do pedido de confirmação e antes da entrega da confirmação de t_j. Para facilitar a compreensão do teste de certificação, a figura 4 ilustra a situação. No exemplo da figura, o pedido de confirmação para a transação T_1 foi entregue no *timestamp* 1 ($ts(1)$). Porém, antes da entrega da confirmação de T_1 (no *timestamp* 9), a transação T_2 foi confirmada (em $ts(3)$). Desta forma, de acordo com as definições do modelo de execução otimista [Kung and Robinson 1981], T_1 é potencialmente conflitante com T_2 e deve ser certificada em relação a ela. Contudo, como as transações são entregues e confirmadas pelas réplicas corretas na mesma ordem, o único conflito que deve ser verificado na certificação é o de leitura, e T_1 deve ser abortada caso tenha lido alguma informação que foi modificada e confirmada por T_2 (caracterização de "leitura-suja", um fenômeno que fere a serialização). Por outro lado, os conflitos de escrita são resolvidos diretamente pela ordem de entrega e da confirmação das transações, assim, as escritas são consonantes com o critério de serialização [Bernstein et al. 1987]. Por fim, se a transação passa no teste de certificação, resta verificar se os resultados obtidos para a transação são equivalentes aos resultados enviados pelo líder da transação, e que já foram enviados de forma especulativa ao cliente. Esta comparação é efetuada através da função *matches* (linha 42), que recebe como argumentos os resumos criptográficos das operações e resultados processados para a transação, e os resumos dos mesmos itens recebidos da réplica líder no pedido de confirmação. Se for verificada a equivalência, a transação é confirmada (linhas 43-44), e, do contrário é cancelada (46-47).

Se a transação é confirmada ela é incluída no conjunto de transações confirmadas ($\prod^c$), retirada do conjunto de transações ativas ($\prod^a$) e tornada persistente na base de dados (linha 44 - função *commit_transaction*). Se a transação é cancelada, ela é retirada do conjunto de transações ativas ($\prod^a$) e as operações desfeitas na base de dados (linha 47 - função *abort_transaction*). A última tarefa do protocolo é enviar ao cliente o resultado final da transação (linha 49). O cliente por sua vez, aceita o resultado se receber pelo menos $f + 1$ mensagens iguais, advindas de diferentes réplicas.

As provas de que o protocolo satisfaz as propriedades definidas na seção 3.2., bem como as propriedades básicas do modelo de transações, propriedades ACID, foram omitidas devido a restrições de espaço. Entretanto, estas provas estão disponíveis na versão estendida

deste artigo [Luiz et al. 2010].

4. Avaliação do Protocolo

Esta seção descreve os resultados verificados para a solução proposta neste trabalho. Para facilitar a comparação, nossa solução é referenciada nas avaliações pelo nome INTACT, cujo significado é ***IN**trusion **T**olerance **A**pplication on **C**onsistent **T**ransactions*. E, visto que a avaliação de sistemas de computação distribuída pode ser realizada de forma analítica e/ou empírica, para demonstrar a viabilidade de nosso protocolo realizamos estas duas avaliações.

4.1. Avaliação Analítica

A primeira avaliação foi realizada a fim de verificar a eficiência do protocolo proposto, em termos das complexidades e das propriedades observadas nos protocolos avaliados. Nesta avaliação consideramos apenas os protocolos do INTACT, do HRDB [Vandiver et al. 2007] e do Byzantium [Garcia et al. 2011] devido a eles compartilharem os mesmos princípios de funcionamento. Assim, a tabela 1 enumera alguns dos resultados observados em cada um dos protocolos mencionados. É lícito ressaltar que a tabela considera apenas o protocolo de confirmação dos trabalhos avaliados, pois é a parte que despende o maior custo/complexidade no processamento da transação. O primeiro item da avaliação nos mostra que, em termos de número de réplicas necessárias, o HRDB é a melhor solução já que o mesmo mantém um coordenador centralizado, e por esta razão não executa um algoritmo de acordo bizantino para ordenar as transações, como os demais.

Por outro lado, se analisarmos as métricas usuais da avaliação de algoritmos distribuídos, isto é, em termos de passos de comunicação e de complexidade de mensagens, no primeiro caso temos uma equivalência para o HRDB e Byzantium. O fato de o INTACT fazer uso de duas difusões com ordem total para a confirmação, de fato, incorre em acréscimo no número de passos de comunicação[2]. Todavia, como assumimos que o protocolo não é executado em um ambiente de larga escala, o custo associado a transmissão e consequentemente os passos de comunicação são moderados (vide seção 4.2.). No caso do HRDB um número menor de passos já era esperado, pois, como ele mantém o controle centralizado, o coordenador é quem decide a ordem de execução das operações e transações, e a confirmação considera apenas as respostas mais frequentes para as operações da transação, não sendo necessário qualquer tipo de acordo entre as réplicas. Para o Byzantium, este resultado advém do fato dele invocar apenas uma vez o protocolo do PBFT [Castro and Liskov 1999], pois como o modelo de consistência assegurado é o *snapshot isolation* [Berenson et al. 1995], não é necessário verificar os conflitos de serialização, mas apenas os conflitos de escritas concorrentes observados nas transações sobre os *snapshots* locais. Porém, é lícito salientar que o modelo *snapshot isolation* não garante a consistência forte dos dados e neste modelo as transações observam apenas o instante da base de dados correspondente ao da última transação confirmada no sistema desde o início da transação em questão. Além disso, o modelo é suscetível a anomalias [Berenson et al. 1995] que podem violar a consistência de dados, mesmo que a certificação seja válida e a confirmação aceita. Assim, até mesmo réplicas corretas estão sujeitas a este problema, o que pode afetar a integridade dos dados. Em se tratando da complexidade de mensagens, tanto o Byzantium como o INTACT são equivalentes, e superiores ao HRDB. Novamente isso ocorre porque o HRDB não realiza nenhum acordo, devido ao uso do coordenador centralizado. O Byzantium e o INTACT possuem a mesma complexidade em razão de ambos utilizaram protocolos de difusão com ordem total equivalentes.

[2]Considerando o número de passos do protocolo, mais os passos usados nas difusões com ordem total.

Tabela 1. Comparação dos protocolos de confirmação de transações.

Característica	HRDB	Byzantium	INTACT
Número de Réplicas	$2f + 2$	$3f + 1$	$3f + 1$
Passos de Comunicação	4	4	7
Complexidade de Mensagens	$\mathrm{O}(n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$
Consistência Forte	S	N	S
Controle Centralizado	S	N	N

Todavia, apesar da necessidade de um número maior de passos comunicação e de complexidade superior ao HRDB, o INTACT é o único protocolo que assegura a consistência forte dos dados em ambientes sujeitos a faltas bizantinas, e, sobretudo, por meio de um protocolo totalmente distribuído. Em nosso ponto de vista, isto pode ser considerado um grande avanço em relação aos demais protocolos avaliados.

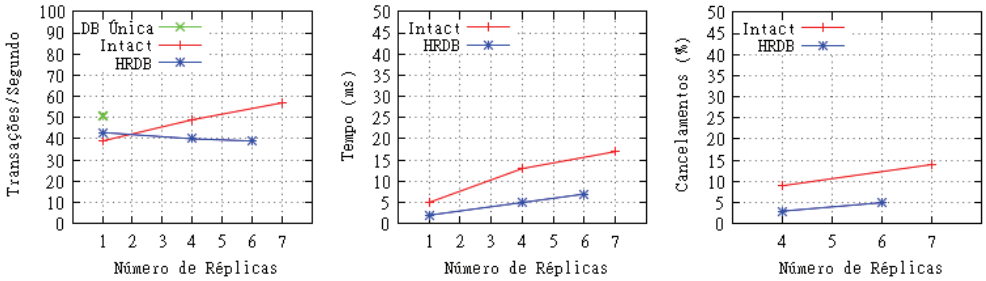
4.2. Avaliação Experimental

A fim de avaliar de forma empírica o protocolo proposto, realizamos uma avaliação experimental preliminar do INTACT, onde os resultados para esta avaliação são reportados na figura 5. Os valores apresentados na figura 5 são oriundos da execução do INTACT e do HRDB, em um ambiente real. As amostras de dados foram obtidas a partir de um protótipo implementado para o INTACT, e de um protótipo existente para o HRDB[3]. O Byzantium não foi considerado nesta avaliação, em razão de não termos tido acesso ao seu código. O ambiente de experimentação foi composto por máquinas Dell Optiplex 755, com a seguinte configuração de *hardware*: 1 microprocessador Intel®Core™2 Duo 2.33GHz, 2GB de memória RAM e uma interface Ethernet Gigabit Intel 82566DM-2. Como sistema operacional adotamos o Linux Ubuntu Server Edition 10.10, e como sistema de gerência de bases de dados usamos o MySQL 5.5.8. Assim, o resultado para a "base de dados única" (da figura 5(a)) é oriundo de conexões efetuadas diretamente ao MySQL. Os dados foram obtidos a partir de 1000 transações, executadas em 1, 4 e 7 réplicas, e emitidas por 25 clientes. Cada transação continha entre 10 e 20 operações, das quais 50% eram de leitura e 50% de escrita.

Os experimentos exploraram a escalabilidade, o tempo necessário para a confirmação de uma transação e a taxa de cancelamentos em razão de conflitos. Conforme se observa na figura 5, nosso protocolo apresenta melhor escalabilidade em relação ao HRDB, quando se aumenta o número de réplicas. A razão para isto é a execução especulativa/otimista realizada pelo INTACT, onde as operações das transações são distribuídas para execução em diferentes réplicas, o que resulta em ganhos de desempenho e escala. Por outro lado, no HRDB, independe do número de réplicas, todas as transações e operações tem de passar pelo coordenador centralizado, o que implica em um gargalo no protocolo.

Embora nossa solução apresente ganhos de escala, a confirmação de transações do INTACT é mais dispendiosa em relação ao HRDB, tal como observado na figura 5(b). Isto já era esperado em virtude do modelo de propagação adotado, pois a propagação de todas as operações da transação ocorre no momento da confirmação. Já no HRDB a propagação é realizada operação a operação, e assim, quando é efetuada a confirmação as réplicas apenas comparam os resultados obtidos para cada uma das operações. Além disso, um aspecto inerente a abordagem especulativa é o cancelamento de parcela das transações, quando se verifica situações de conflito [Lampson 2006]. Contudo, isto é necessário para evitar que as transações violem o critério de consistência (ex. serialização). Naturalmente, o HRDB incorre em menor

[3]Disponível em `http://web.mit.edu/benmv/pmg/www/hrdb.tar.gz`.



(a) Escalabilidade. (b) Tempo de Confirmação. (c) Cancelamentos (%).

Figura 5. Custo dos protocolos de confirmação.

taxa de cancelamentos (figura 5(c)), devido ao protocolo atrasar a execução das transações em que se verifica conflitos, e esta é uma das razões pela qual ele perde no aspecto de escalabilidade. Além disso, é digno de nota que o tempo necessário para a confirmação é proporcional ao tamanho da transação e da quantidade de itens de dados por ela afetados.

5. Considerações Finais

O processamento correto de transações é fundamental e indispensável para orientar o protocolo de confirmação da transação a tomar uma decisão favorável, em face à ocorrência de faltas. Neste trabalho, apresentamos uma nova solução para a confirmação de transações em cenários onde tantos os clientes e um número limitado de réplicas podem exibir o comportamento bizantino. A estratégia adotada para a concepção do protocolo é bastante modular, já que a mesma é baseada na abstração de difusão com ordem total, uma ferramenta largamente usada para a concepção de sistemas distribuídos confiáveis. Assim, a principal contribuição deste trabalho é a concretização de um protocolo robusto, seguro e que tolera a faltas bizantinas tanto por parte das réplicas como dos clientes, durante o processamento e confirmação das transações. Além disso, o protocolo preserva a consistência forte de dados, operando de maneira totalmente distribuída.

Referências

Agrawal, D., Alonso, G., Abbadi, A. E., and Stanoi, I. (1997). Exploiting atomic broadcast in replicated databases (extended abstract). In *Euro-Par'97: 3rd International Euro-Par Conference on Parallel Processing*, pages 496–503, London, UK. Springer-Verlag.

Amoroso, E. G. (1994). *Fundamentals of computer security technology*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Babaoğlu, O. and Toueg, S. (1993). Non-blocking atomic commitment. In *Distributed systems*, pages 147–168, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

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.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

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.

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.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. L. (2007). Zyzzyva: speculative byzantine fault tolerance. In *SOSP'07: Proceedings of 21st ACM Symposium on Operating Systems Principles*, pages 45–58, New York, NY, USA. ACM.

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

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Lampson, B. W. (1981). Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK. Springer-Verlag.

Lampson, B. W. (2006). Lazy and speculative execution in computer systems. In Shvartsman, A. A., editor, *OPODIS'06: Proceedings of the 10th International Conference on Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 1–2. Springer-Verlag.

Luiz, A. F., Lung, L. C., and Correia, M. (2010). Protocolo tolerante a faltas bizantinas para bases de dados transacionais. Relatório técnico, PGEAS/DAS/UFSC. Disponível em `http://www.das.ufsc.br/~aldelir/report2010-01.pdf`.

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., Guerraoui, R., and Schiper, A. (2003). The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98.

Raynal, M. and Singhal, M. (2001). Mastering agreement problems in distributed systems. *IEEE Software*, 18:40–47.

Schiper, A. and Raynal, M. (1996). From group communication to transactions in distributed systems. *Communications of the ACM*, 39:84–87.

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

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*.

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