Dealing with Interactive Transactions in a Byzantine Fault Tolerant STM

Tulio Alberton Ribeiro*, Lau Cheuk Lung*, Hylson Vescovi Netto*†
*Department of Informatics and Statistics,
Federal University of Santa Catarina - UFSC, Florianópolis, Brazil
†Federal Institute of Santa Catarina - IFC, Blumenau, Brazil
tulio.ribeiro@posgrad.ufsc.br, lau.lung@ufsc.br, hylson.vescovi@blumenau.ifc.edu.br

***Abstract*—Recently, researchers have shown an increased interest in concurrency control using distributed Software Transactional Memory (STM). However, there has been little discussion about certain types of fault tolerance, such as Byzantine Fault Tolerance (BFT), for kind of systems. The focus of this paper is on tolerating byzantine faults on optimistic processing of interactive and declared transactions using STM. The result is an algorithm named Mesobi. The processing of a transaction runs with an optimistic approach, benefiting from the high probability of messages being delivered in order when using Reliable Multicast on a local network (LAN). The protocol performs better when messages are delivered ordered. In case of a malicious replica or out-of-order messages, the Byzantine protocol is initiated.**

***Keywords*-Software Transactional Memory, Byzantine Fault Tolerance, Optimistic Commit**

I. Introdução

Os sistemas de Memória Compartilhada Distribuída (DSM - Distributed Shared Memory) abstraem os mecanismos de comunicação remota do desenvolvedor da aplicação [1]. As DSM podem ser descritas como um espaço de endereçamento virtual que é compartilhado por um conjunto de processadores [2]. Nesses sistemas, os processadores acessam cada endereço de memória como se fosse local. As DSM por sua vez, permitem o compartilhamento coerente de dados através de acesso uniforme às escritas e leituras (sincronização baseada em travas - *locks*). Como uma alternativa para a sincronização baseada em travas, surgiram as Memórias Transacionais em Software (STM - Software Transactional Memory). As STM permitem acesso uniforme às escritas e leituras através de transações. As transações em STM são semelhantes as transações em bancos de dados, porém, sem a propriedade durabilidade.

Pesquisadores têm mostrado um aumento de interesse no controle de concorrência utilizando STM, como pode ser visto no trabalhos D^2STM [3], *RAM-DUR* [4], *SPECULA* [5], *OSARE* [6], *SCert* [7], *AGGRO* [8], *STR* [9], Granola [10] e *Zhang* [11]. Ao utilizar STM, os programadores não precisam lidar com mecanismos explícitos de controle de concorrência (como monitores, *locks* ou semáforos). Ao invés disso, apenas precisam delinear quais objetos necessitam ser tratados como concorrentes, através do uso de transações. Com isso, é possível focar mais na lógica da aplicação do que nos mecanismos explícitos de controle de concorrência, que é feito pela STM.

A maioria dos trabalhos citados acima, tratam apenas tolerância a faltas de parada (*crash*). Entre eles, *AGGRO*, *SPECULA*, *OSARE* e *STR* utilizam a alta probabilidade das mensagens serem entregues em ordem pela rede [12], [13], mas necessitam que a ordem final de entrega de mensagens seja definida para concluir sua execução. Por sua vez, somente o trabalho de [11] menciona tolerância a faltas Bizantinas no contexto de STM. O modelo proposto por Zhang não utiliza a alta probabilidade das mensagens serem entregues em ordem pela rede, e aborta transações somente leitura.

Este trabalho apresenta o Mesobi, uma arquitetura de STM tolerante a faltas Bizantinas que executa, em simultâneo, tanto transações interativas (ou online) quanto transações pré-declaradas utilizando um mecanismo otimista no processamento das mesmas. O protocolo proposto no Mesobi é baseado na execução paralela de transações não conflitantes, semelhante ao proposto por [14]. O presente trabalho é uma evolução do OB-STM [15] que lida somente com transações pré-declaradas.

Transações interativas são um recurso útil para aplicações que seguem uma lógica dependente de resultados de operações anteriores. Em ambientes onde é necessária a confidencialidade das operações a lógica do negócio deve estar junto ao cliente. Existem algumas abordagens que permitem a execução confidencial de algumas operações, como média, desvio padrão e regressão; porém operações mais complexas ainda não podem ser feitas na réplica [16]. Lidar com transações interativas é um problema difícil de se resolver quando se considera a possibilidade de faltas bizantinas no sistema. Processos bizantinos podem de forma maliciosa induzir ao erro processos corretos, através de comportamento arbitrário. O comportamento arbitrário ou malicioso, pode causar a perda de consistência ao enviar respostas diferentes a processos corretos. Protocolos que não consideram tolerância a faltas maliciosas podem ficar estagnados na presença de processos maliciosos; por exemplo, não enviando ou atrasando respostas para outras réplicas que estejam esperando pelas mesmas. Torna-se então importante um protocolo que permita a interação segura entre clientes e réplicas, garantindo que as transações serão corretamente executadas a despeito de faltas arbitrárias.

O Mesobi se beneficia da alta probabilidade das mensagens serem entregues em ordem pela rede usando IP-Multicast [13], [12]; somente em caso de mensagens fora de ordem ou réplicas maliciosas é que o protocolo

978-1-4799-5612-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SBRC.2014.20

Bizantino inicia, não necessitando que a ordem final seja definida para a execução das transações no caso otimista. O Mesobi foi construído utilizando a JVSTM e preserva suas propriedades *weak atomicity* e *opacity*. A propriedade *weak atomicity* garante atomicidade entre transações, e a propriedade *opacity* garante que todas as transações observam um estado consistente do sistema. A JVSTM é uma biblioteca que suporta controle de concorrência com múltiplas versões *(MVCC - MultiVersioning Concurrency Control)*, foi criada por [17] e oferece um excelente desempenho em operações somente leitura.

O artigo está organizado como segue: Na seção II estão os trabalhos relacionados. A seção III descreve o modelo e definições do sistema. Na seção IV é detalhado o protocolo e seu funcionamento. Na seção V encontram-se os algoritmos utilizados. Na seção VI estão as avaliações e resultados encontrados. Por fim, as conclusões na seção VII.

II. Trabalhos relacionados

Nesta seção, sintetiza-se brevemente os trabalhos relevantes para a construção do Mesobi. A tabela I resume as principais características dos trabalhos relevantes em STM. As abordagens D^2STM e $Zhang$ utilizam *Atomic Broadcast* (AB) desde o início de sua execução, não levando em conta uma possível ordenação das mensagens pela rede. Inicialmente a abordagem do Mesobi é otimista, ou seja, não há a necessidade de um mecanismo de entrega de mensagens atômico para execução de transações, quando operando de forma otimista.

Nos trabalhos $AGGRO$, $SPECULA$, $OSARE$ e STR existe execução otimista das transações, mas diferente do Mesobi, é necessário que a ordem final de entrega das mensagens seja definida para que as transações possam ser confirmadas. Mais especificamente, o modelo utilizado em STR é baseado na completa exploração de todas ordens distintas plausíveis recebidas pelo *Optimistic Atomic Broadcast* (OAB). O Mesobi utiliza uma abordagem semelhante à utilizada por STR, $AGGRO$, $SPECULA$ e $SCert$ onde transações são executadas localmente logo que são recebidas, sem necessidade de troca de mensagens inter-réplica. É interessante notar que as abordagens STR, $AGGRO$ e $OSARE$ bloqueiam a *thread* que executou a transação, impossibilitando que código não transacional seja executado. No $SCert$ novas transações podem ser bloqueadas temporariamente antes de serem executadas.

Semelhante ao Mesobi, STR, $AGGRO$, $OSARE$, $RAM-DUR$ e $SCert$ não necessitam do conhecimento *a-priori* dos dados acessados por uma transação para sua execução (transações interativas, ver seção III-A). O Mesobi utiliza para transações interativas mecanismo semelhante ao $RAM-DUR$ para verificação de conflitos entre transações, onde os conjuntos de leitura e escritas são enviados para cada réplica. Diferente de $RAM-DUR$, ao invés de utilizar AB como mecanismo de validação, Mesobi utiliza um modelo de certificação total, melhor detalhado na seção IV-A. Adicionalmente, o Mesobi trata da execução simultânea de transações pré-declaradas e interativas, sendo o cliente quem define por qual utilizar. Nosso trabalho utiliza mecanismo semelhante ao *Granola* para certificação entre transações, onde é necessário que todos repositórios respondam positivamente a uma requisição de *commit*. Nosso modelo também utiliza mecanismo semelhante quando transações interativas necessitam confirmar em paralelo: o desempate é feito priorizando a transação com menor identificador.

	A-B	Byzantine.	Réplica.	A-R	Modelo
D2STM	Sim	Não	>1	Não	Pré-declarado
RAM-DUR	Sim	Não	>1	Não	Interativo
AGGRO	Sim[1]	Não	>1	Não	Interativo
STR	Idem AGGRO	Não	>1	Não	Interativo
SPECULA	Idem AGGRO	Não	>1	Sim	Interativo
SCert	Idem AGGRO	Não	>1	Sim	Interativo
OSARE	Idem AGGRO	Não	>1	Não	Interativo
Granola	Não	Não	$(2f+1)*c_2$	Não	Pré-declarado
Zhang	Sim	Sim	3f+1; 2f+1	Sim	Pré-declarado
Mesobi	**Adaptive**[3]	**Sim**	**3f+1**	**Sim/Não**[4]	Ambos

Tabela I: Trabalhos relacionados e proposta.

Dentre os trabalhos citados, somente Zhang menciona tolerância a faltas Bizantinas no contexto de STM. Sua abordagem utiliza $3f+1$ réplicas para consenso e $2f+1$ réplicas para execução. No trabalho de Zhang, transações somente leitura podem ser abortadas, não se utiliza a alta probabilidade em que as mensagens são entregues pela rede utilizando IP-Multicast e o conceito de transações interativas não é considerado.

III. Modelo e Definições

Esta seção descreve as definições básicas do sistema, premissas consideradas para o protocolo e sua arquitetura.

A. Definições básicas do sistema e premissas

Consideramos um sistema distribuído assíncrono clássico, consistindo de um conjunto arbitrário de clientes (não infinito) não Bizantinos C = $\{c_1, c_2, ..., c_n\}$ e um finito conjunto de réplicas R = $\{r_1, r_2, ..., r_n\}$ [18]. Assumimos em nosso modelo um nível parcial de sincronia, onde o sistema se comporta de forma assíncrona em grande parte do tempo mas existem períodos de estabilidade [19]. As réplicas se comunicam via passagem de mensagem e podem falhar de acordo com o modelo de faltas Bizantinas [20] em até f réplicas. A cardinalidade do conjunto de réplicas consiste de $\mid R \mid \geq 3f+1$.

Uma réplica é dita faltosa ou Bizantina quando se desvia de suas especificações, uma réplica faltosa pode parar de

[1]Precisa esperar pela ordem final de entrega de mensagens para terminar.

[2]Existem 2f+1 réplicas dentro de cada repositório, onde *c* é o número de repositórios.

[3]*Reliable Multicast* quando otimista e *Total Order Byzantine Multicast* quando não otimista.

[4]Aborta read-only somente para transações interativas.

enviar mensagens, enviar mensagens fora de ordem, omitir o envio ou recebimento de mensagens, atrasar mensagens e corromper mensagens. Todas as mensagens são assinadas e trafegam em um canal confiável.

O protocolo utiliza transações pré-declaradas e interativas. Em transações pré-declaradas, o cliente necessita enviar todo código transacional para as réplicas [5]. Nas transações interativas, o cliente interage com as réplicas, enviando operação por operação. As transações interativas têm prioridade sobre as transações pré-declaradas, sendo que em caso de confirmação simultânea as transações interativas são confirmadas.

B. Arquitetura

A figura 1 provê uma visão em alto nível da arquitetura composta em cada réplica Mesobi. O componente Mesobi recebe as solicitações do cliente e faz a comunicação inter-réplica. Todas as solicitações ao serem recebidas pelas réplicas são analisadas através do analisador de transações paralelas (*PTA*); este módulo é divido em duas partes, uma específica para transações interativas denominado *Gerente TI* e outra específica para transações pré-declaradas denominado *Gerente TD*. Após serem analisadas, as transações são encaminhadas para os executores de transações caso não possuam conflitos (JVSTM), mais detalhes sobre conflitos podem ser vistos na seção IV-B. Os executores JVSTM são compostos por *threads* e cada transação é vinculada a uma *thread*. Após o término da transação, o certificador otimista faz a verificação de possibilidade de confirmar a transação de maneira otimista, caso isso seja possível, a transação é confirmada. Caso contrário, o protocolo Bizantino precisa ser executado.

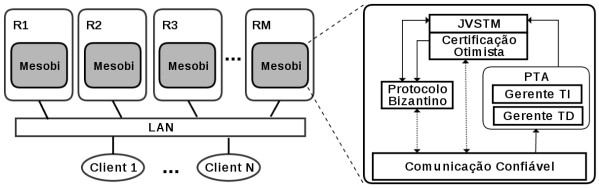

Figura 1: Arquitetura de uma réplica Mesobi.

IV. Protocolo Mesobi

Para melhor compreensão, será explicado o fluxo de operação (figura 2) junto com os algoritmos (seção V). Inicialmente o cliente envia uma solicitação às réplicas (etapa TX.Begin, figura 2). A solicitação do cliente pode ser de dois tipos, pré-declarada (T_i) ou interativa (TX_i). Cada cliente pode apenas enviar uma solicitação por vez, independente se pré-declarada ou interativa. As transações pré-declaradas e interativas são definidas em forma de tuplas, e necessariamente precisam conter: $T = \{T_i.B, T_i.Op_1, ..., T_i.Op_n, T_i.C\}$ para as pré-declaradas e $TX = \{TX_i.B, TX_i.Op_1, ..., TX_i.Op_n, TX_i.C\}$ para as interativas, onde $B \equiv$ Begin, $Op \equiv$ Operação e $C \equiv$ Commit.

A execução otimista e não otimista das transações pré-declaradas, na ausência de transações interativas, segue fluxo idêntico ao protocolo OB-STM [15]. O Mesobi é uma evolução do protocolo OB-STM onde transações interativas e pré-declaradas concorrem pela execução, sendo que as transações interativas tem prioridade sobre as pré-declaradas. O Mesobi utiliza o conceito *WiP - WorkInProgress*, semelhante ao utilizado por [8], diferindo no quesito propagação de atualização de objetos. Uma transação está em andamento (WiP) se a mesma já foi iniciada e ainda não foi concluída. A área sombreada na figura 2b representa o PBFT [21], utilizado na fase de resolução bizantina (mensagens fora de ordem ou réplicas maliciosas).

No caso de transações interativas, o cliente necessita efetuar alguns passos a mais. Inicialmente o cliente solicita às réplicas a intenção de iniciar uma transação (etapa TX.Begin figura 2, linha 1 algoritmo 2). Cada réplica retorna ao cliente o seu *timestamp*. O *timestamp* (ts) utilizado segue o modelo proposto por [22], para evitar o crescimento arbitrário do espaço de endereçamento por réplicas maliciosas. O cliente espera por $2f + 1$ respostas das réplicas contendo o ts em questão. Com o ts de $2f+1$ réplicas, é possível definir o valor que será considerado pelo gerenciador de contenção caso exista concorrência entre transações interativas. A figura 2 detalha a interação cliente-réplicas e réplica-réplica. Após a definição do ts pelo cliente (etapa TX.Begin figura 2), a primeira operação ($TX_i.Op$) pode ser enviada às réplicas (etapa TX.Op figura 2, linha 6 algoritmo 2). A operação somente é executada se não houver conflitos com outras transações em execução ou esperando para ser executada (linha 12 do algoritmo 2). A cada operação executada pelas réplicas um novo ts é retornado ao cliente para determinar a ordem das operações; essa ordem de operação é utilizada pelo gerenciador de contenção como critério de desempate.

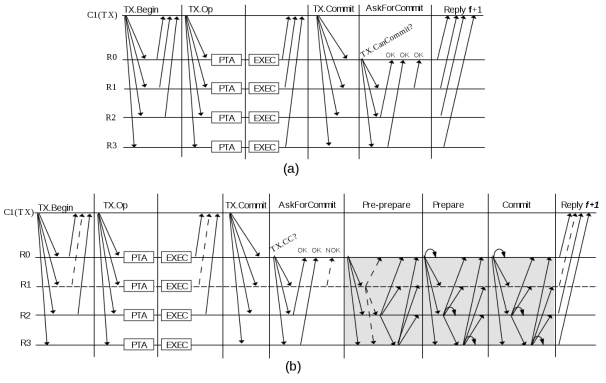

Figura 2: Transações interativas: fluxo de operação otimista (a) e não otimista (b).

Após o cliente terminar o envio de operações, o cliente precisa enviar o comando de término da transação ($TX_i.C$) (etapa TX.Commit figura 2). Ao receber do

[5]Na prática *templates* das transações são criados nas réplicas e apenas os parâmetros são enviados.

cliente a solicitação de confirmação (linha 20 algoritmo 2) cada réplica envia para o conjunto de réplicas no sistema ($\prod^R$) uma mensagem de permissão de confirmação da transação em questão (etapa AskForCommit, figura 2). Cada réplica necessita receber a confirmação de todas outras réplicas (linha 39 algoritmo 3) para confirmar a transação. Caso receba uma não-permissão (mensagem NOK!, etapa AskForCommit, figura 2b) devido a uma réplica maliciosa ou transações fora de ordem, o protocolo Bizantino precisa ser iniciado (etapa Pre-prepare, figura 2b, linha 42 algoritmo 3). O protocolo Bizantino, para evitar a sobrecarga imposta pelo consenso, quando iniciado, define a ordem das transações pré-declaradas e interativas existentes nos *buffers* $\prod^T$ e $\prod^{TX}$. Os critérios definidos para confirmar ou abortar uma transação serão explicados a seguir.

A. Histórico, premissas e snapshot

Nessa seção será detalhado o tratamento das transações quando executadas no caso normal (sem falhas e entrega ordenada das mensagens) e no caso com falhas (mensagens fora de ordem ou réplicas maliciosas). Para ambas abordagens, as definições abaixo se mantêm.

O princípio da teoria de seriabilidade [23] garante que a execução do histórico $\mathcal{H}$ de transações em todos os processos replicados é equivalente à execução de $\mathcal{H}$ em um ambiente não replicado. O histórico de execução $\mathcal{H}$ (também conhecido como *schedule*) é definido sobre um conjunto de transações $T = \{T_1, T_2, ..., T_n\}$ e especifica um conjunto de operações (podendo ser operações entrelaçadas) dessas transações. Cada transação é composta por uma *tupla*: $T = \{B, Op_1, ..., Op_n, C\}$, e nas transações interativas as operações podem ser enviadas uma-a-uma. Formalmente, um histórico completo $\mathcal{H}^c_T$ definido sobre um conjunto de transações $T = \{T_1, T_2, ..., T_n\}$ é um histórico de ordem parcial $\mathcal{H}^p_T = \{\Sigma_T, \prec_\mathcal{H}\}$ onde:

(1) $\Sigma_T = \cup^n_{i=1}\Sigma_i$. (2) $\prec_\mathcal{H} \supseteq \cup^n_{i=1} \prec T_i$. (3) Para quaisquer duas operações conflitantes $Op_i, Op_j \in \Sigma_T$, ou $Op_i \prec_\mathcal{H} Op_j$, ou $Op_j \prec_\mathcal{H} Op_i$.

A primeira condição declara que o histórico global de execução das operações é a união da execução das operações individuais. A segunda, define a relação de ordenação do histórico como um super-conjunto das ordenações de operações individuais (a ordem das operações dentro de cada transação é mantida). A terceira, define a ordem em que as operações conflitantes são executadas no histórico.

A ordem serial imposta pelo Mesobi é garantida através das seguintes premissas: (i) toda transação pré-declarada que não apresentar conflito com transações interativas e pré-declaradas (em execução ou na fila de execução) pode ser executada de imediato (transações sem conflito a ordem de execução não é importante); (ii) transações pré-declaradas que tenham conflitos com operações de transações interativas não podem ser executadas de imediato; (iii) operações de transações interativas são executadas somente se não houver conflitos com transações pré-declaradas em execução ou que estejam marcadas como *WiP*. As verificações das premissas (i) e (ii) encontram-se no algoritmo 1, na função PTA (linha 1). A premissa (iii) é verificada no algoritmo 2, linha 12. O Mesobi utiliza *weak snapshot isolation*, que permite transações somente leitura sejam feitas em um *snapshot* que reflete um estado correto (*committed*) dos dados. O modelo *snapshot isolation* que o Mesobi utiliza possui algumas variações. O *snapshot* é capturado no momento em que ocorre a primeira operação ($TX_i.Op$) emitida pelo cliente nas transações interativas; o momento considerado consiste no momento sem conflitos, onde a operação poderá ser executada. Operações pré-declaradas somente leitura não são abortadas, pois retornam os dados do último *snapshot* confirmado. Essas situações são exemplificadas na figura 3.

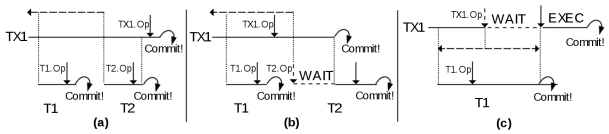

Figura 3: Transações interativas e pré-declaradas: (a) sem conflito. (b) com conflito ($TX_1.Op \prec T_2.B$). (c) com conflito ($T_1.Op \prec TX_1.Op$), TX_1 precisa esperar T_1 confirmar. O símbolo $\downarrow$ deve ser interpretado como a operação $z = z + 1$;

B. Caso otimista

A figura 3 detalha as relações de conflitos entre transações interativas e declaradas. No detalhe (a), não existem conflitos entre as transações, pois não existem operações entrelaçadas (visão local). Formalmente, $\mathcal{H}^c_T = \{T_1.Op, T_1.C, T_2.Op, T_2.C, TX_1.Op, TX_1.C\}$[6]. É importante lembrar que o *snapshot* somente é capturado quando a operação ($TX_1.Op$) for recebida pela réplica; nesse caso, como ($TX_1.Op$) não possui conflitos, o *snapshot* é capturado. No detalhe (b), existe conflito entre as operações ($TX_1.Op$) e ($T_2.Op$). Nesse caso, à transação T_2 não será permitido executar de imediato, pois viola a premissa (ii). Somente após o término (*commit*|*abort*) da transação TX_1 é que T_2 será executada. No detalhe (c), existe conflito entre as operações ($T_1.Op$) e ($TX_1.Op$). A operação ($TX_1.Op$) não poderá ser executada de imediato, pois viola a premissa (iii). Somente após o término (*commit*|*abort*) da transação T_1 é que ($TX_1.Op$) será executada. Ressalta-se que no primeiro momento em que a operação ($TX_1.Op$) foi recebida pela réplica havia conflito (seta pontilhada), e com isso o *snapshot* não pode ser capturado. Somente após o (*commit*|*abort*) da transação T_1 é que o *snapshot* realmente será capturado. É importante ressaltar que essas propriedades se mantêm no Mesobi, mesmo quando a ordem em que as transações são recebidas pelas réplicas forem diferentes. Isso se dá devido ao protocolo Bizantino, que em caso de mensagens fora de ordem, ordena as mesmas. O caso não otimista será melhor detalhado na seção IV-C.

[6]As operações de *Begin* das transações interativas foram omitidas pois não influenciam na análise de conflito.

C. Caso não otimista

Conforme a figura 4, suponha que a ordem de chegada das operações nas réplicas seja: $R_{0,1} = \{T_1.B, T_2.B, TX_1.Op\}$ e $R_{2,3} = \{T_2.B, T_1.B, TX_1.Op\}$[7]. A transação T_2 nas réplicas $R_{0,1}$ e a transação T_1 nas réplicas $R_{2,3}$ não poderão ser executadas de imediato pois não atendem a premissa (i). Os históricos de operações das réplicas $R_{0,1}$ são diferentes das réplicas $R_{2,3}$, o que viola a seriabilidade. Para fins de simplicidade, definiremos que a réplica R_0 terminou a execução de T_1 primeiro. Após o término, uma mensagem chamada *AskForCommit* (AFC) é enviada para cada réplica solicitando permissão para confirmar T_1. Cada réplica ao receber a mensagem AFC, verifica se a transação pode ser confirmada. Devido às ordens de execução distintas entre as réplicas $R_{0,1}$ e $R_{2,3}$, onde $\mathcal{H}^p_{R_{0,1}} = \{T_1.Op\}$ e $\mathcal{H}^p_{R_{2,3}} = \{T_2.Op\}$, uma mensagem de retorno contendo uma não-permissão de confirmação será gerada e devido a mensagem de não confirmação, o protocolo Bizantino será iniciado. O protocolo Bizantino definirá a ordem de execução ou confirmação das transações através de trocas de mensagens utilizando *total order deliver* (*TO-Deliver* - PBFT). Após a definição da ordem pelo protocolo Bizantino, as réplicas corretas seguirão a definição imposta.

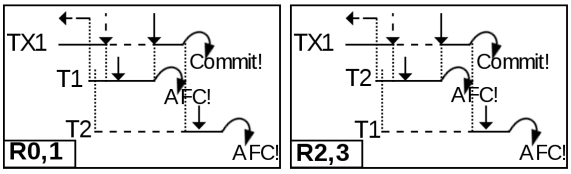

Figura 4: Transações pré-declaradas e interativas. Com conflito de dados.

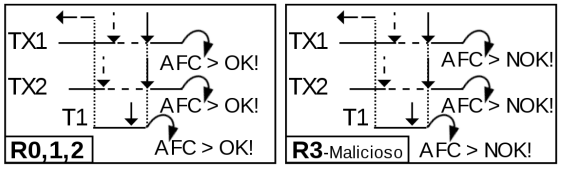

Figura 5: Transações pré-declaradas e interativas. Com réplica maliciosa.

A figura 5 retrata o caso com réplica maliciosa, a ordem de recebimento das transações é a mesma em todas as réplicas: $R_{0,1,2,3} = \{T_1.B, TX_2.Op, TX_1.Op\}$. Nesse exemplo transações interativas e declaradas são conflitantes; conforme as premissas (i) e (ii), a transação $T_1.B$ não possui conflitos com as operações ($TX_1.Op$) e ($TX_2.Op$) nem com outras transações declaradas, isso permite com que T_1 seja executada. Porém, enquanto T_1 está executando, as operações das transações TX_1 e TX_2 chegam. Com isso as operações da transação TX_1 e TX_2 devem esperar, conforme premissa (iii). Após o término da execução da transação T_1, caso não existisse réplica maliciosa (R3), o protocolo deveria terminar, mas ao solicitar permissão para confirmar a transação T_1, as réplicas solicitantes recebem uma mensagem (NOK!); com isso o protocolo Bizantino deve ser iniciado. O protocolo Bizantino define a ordem e as réplicas corretas seguem sua imposição.

As relações de conflitos entre transações interativas e pré-declaradas são detectadas em função das seguintes condições: para as operações interativas, o ponto importante são as operações e para as pré-declaradas o início (*Begin*). Então, conflitos acontecem se: $T_i.B \prec TX_i.Op$ ou $TX_i.op \prec T_i.B$; nesses casos as premissas (i, ii e iii) devem ser seguidas.

V. Algoritmos

Essa seção contém o algoritmo que é executado em cada réplica; o algoritmo do cliente foi omitido por restrição de espaço. Os algoritmos foram divididos para melhor enquadramento no texto e melhor compreensão. A tabela II detalha as variáveis utilizadas. No algoritmo 1 é feita análise de conflitos entre transações interativas e pré-declaradas e execução das transações pré-declaradas. O algoritmo 2 trata a execução das transações interativas. A verificação de possibilidade de confirmação das transações entre réplicas e confirmação das mesmas caso possível, é detalhada no algoritmo 3. Por fim, o algoritmo 4 define a ordem de confirmação das transações caso mensagens estejam fora de ordem ou existam réplicas maliciosas; além disso re-executa transações, aborta transações e define qual será a próxima a ser executada. Os algoritmos são citados na seção IV onde são explicados em paralelo com a figura 3.

Tabela II: Variáveis.

1: $\prod^R$	▷ Replica Set
2: $\prod^{Reply}$	▷ Reply Set
3: $\prod^T$	▷ Pre-Declared Transactions Set
4: $\prod^{WS}$	▷ Pre-Declared Transactions Write Set
5: $\prod^{RS}$	▷ Pre-Declared Transactions Read Set
6: BCO	▷ Pre-Declared Buffer Commit Order
7: $\prod^{TXBegin}$	▷ Interactive Transactions Begin Set
8: $\prod^{TXOp}$	▷ Interactive Transactions Operation Set
9: $\prod^{TXCommit}$	▷ Interactive Transactions Commit Set
10: $\prod^{XWS}$	▷ Interactive Transactions Write Set
11: $\prod^{XRS}$	▷ Interactive Transactions Read Set
12: f	▷ Number of tolerated faults
13: timeout	▷ Max time to wait for response of replicas

A. Otimização

Existe uma otimização que pode ser feita para melhorar o desempenho e diminuir o número de mensagens trafegadas pela rede. A fase de certificação de uma transação (linha 21 do algoritmo 1) pode ser melhorada através da seguinte verificação: (i) cada réplica ao receber uma mensagem *AskForCommit* armazena o identificador da réplica solicitante; (ii) antes de enviar uma mensagem *AskForCommit*, a réplica verifica se já recebeu uma mensagem *AskForCommit* sobre a mesma solicitação feita por outra réplica. Caso essas premissas sejam verdadeiras, menos mensagens serão trocadas. Outro ponto que pode

[7]Para as transações pré-declaradas o *Begin* é importante na análise de conflito, pois os conjuntos de leitura e escrita já são conhecidos.

Algoritmo 1 Transações pré-declaradas.

1: **function** PTA(T_i, C_i) ▷ Make conflict analysis among all transactions
2: **if** ($size(\prod^{T}) > 1 \vee size(\prod^{TXBegin}) > 0$) **then**
3: **for all** $Tj \in \prod^{T}$ minus T_i **do**
4: **if** (¬ ((WS(T_i) ∩ RS(T_j) = ∅) ∧ (WS(T_j) ∩ RS(T_i) = ∅) ∧ (WS(T_i) ∩ WS(T_j) = ∅))) **then**
5: exit ATP
6: **for all** $TX_i.Op \in \prod^{TXOp}$ **do**
7: **if** (¬ ((WS(T_i) ∩ XRS(TX_i) = ∅) ∧ (WS(TX_i) ∩ RS(T_i) = ∅) ∧ (WS(T_i) ∩ XWS(TX_i) = ∅))) **then**
8: exit ATP
9: Call **EXEC-T**(T_i, C_i)
10: **upon:** receive($\langle REQUEST, T_i, C_i \rangle \sigma$) from client
11: $\prod^{T} \leftarrow \prod^{T} \cup (T_i, C_i)$
12: $\prod^{WS} \leftarrow \prod^{WS} \cup$ (getWriteSet(T_i), C_i)
13: $\prod^{RS} \leftarrow \prod^{RS} \cup$ (getReadSet(T_i), C_i)
14: **Call PTA**(T_i)
15:
16: **function** Exec-T(T_i, C_i)
17: **set** WS(T_i) and RS(T_i) as WorkInProgress
18: Executes T_i atomically
19: **set** T_i as Commiting
20: ReliableMulticast($\langle$ASKFORCOMMIT$\rangle$,
21: $T_i, \prod^{R}, C_i, WS, RS$)

Algoritmo 2 Transações interativas.

1: **upon:** receive($\langle REQUEST - B, TX_i.Begin, C_i \rangle \sigma$) from client
2: $\prod^{TXBegin} \leftarrow \prod^{TXBegin} \cup (TX_i.Begin, C_i)$
3: **set** TX_i as running
4: SendReliable($\langle$RSEQ-NUMBER$\rangle$, C_i, R_i, RTS) ▷ Return Replica TS (RTS) to client
5:
6: **upon:** receive($\langle REQUEST - OP, TX_i.Op, TX_i.CTS, C_i \rangle \sigma$) from client ▷ CTS - Client Timestamp
7: $\prod^{TXOp} \leftarrow \prod^{TXOp} \cup (TX_i.Op, C_i)$
8: $\prod^{XWS} \leftarrow \prod^{XWS} \cup$ (getWriteSet($TX_i.Op$), $TX_i.CTS$, C_i)
9: $\prod^{XRS} \leftarrow \prod^{XRS} \cup$ (getReadSet($TX_i.Op$), $TX_i.CTS$, C_i)
10: initExec ← true
11: **for all** $T_j \in \prod^{T} \wedge$ WiP(T_j) **then**
12: **if** (¬ (((WS(T_j) ∩ (XRS($TX_i.Op$))=∅) ∧ (RS(T_j) ∩ (XWS($TX_i.Op$))=∅) ∧ (WS(T_j) ∩ (XWS($TX_i.Op$))=∅)))) **then**
13: initExec ← false
14: stop **for all**
15: **if** (initExec)
16: EXEC-TX ($TX_i.Op$)
17: **else**
18: Set $TX_i.Op$ as waiting until be informed of a commit.
19:
20: **upon:** receive($\langle$ REQUEST-C, $TX_i.Commit$, C_i $\rangle \sigma$) from client
21: ReliableMulticast($\langle ASKFORCOMMIT - TX, TX_i.Commit, \prod^{R}, XWS, XRS \rangle \sigma$)
22:
23: **function** EXEC-TX($TX_i.Op$)
24: $Result \leftarrow$ Execute ($TX_i.Op$) ▷ Execute an operation in an interactive transaction
25: SendReliable($\langle RSEQ - NUMBER, C_i, R_i, RTS, Result \rangle \sigma$)

Algoritmo 3 Certificação e confirmação.

1: **upon:** receive($\langle ASKFORCOMMIT, T_i, R_i, C_i \rangle \sigma$) from replica
2: **if** (T_i can commit) **then** ▷ verification of conflicts, resembling lines 4 and 7 algorithm 1
3: SendReliable($\langle R{-}ASKFORCOMMIT, T_i, R_i, C_i, "OK!" \rangle \sigma$)
4: **else**
5: SendReliable($\langle R - ASKFORCOMMIT, T_i, R_i, C_i, "NOK!" \rangle \sigma$)
6:
7: **upon:** receive($\langle ASKFORCOMMIT - TX, TX_i, R_i, C_i \rangle \sigma$) from replica
8: **if** (TX_i can commit) **then** ▷ verification of conflicts , resembling lines 4 and 7 algorithm 1
9: SendReliable($\langle R - ASKFORCOMMIT - TX, TX_i, R_i, C_i, "OK!" \rangle \sigma$)
10: **else**
11: SendReliable($\langle R - ASKFORCOMMIT - TX, TX_i, R_i, C_i, "NOK!" \rangle \sigma$)
12: **function** COMMIT-T(T_i)
13: Commit transaction T_i!
14: $\prod^{T} \leftarrow \prod^{T} \setminus T_i$
15: $\prod^{WS} \leftarrow \prod^{WS} \setminus$ (getWriteSet(T_i), C_i)
16: $\prod^{RS} \leftarrow \prod^{RS} \setminus$ (getReadSet(T_i), C_i)
17: Informs waiting TX.Ops a Transaction was committed
18: SendReliable($\langle$REPLY$\rangle$, Ri, $Reply$)
19: Call **NEXT-T-EXEC**()
20:
21: **function** COMMIT-TX(TX_i)
22: Commit transaction TX_i!
23: $\prod^{TXBegin} \leftarrow \prod^{TXBegin} \setminus TX_i.Begin$
24: $\prod^{XWS} \leftarrow \prod^{XWS} \setminus$ (getWriteSet(TX_i), C_i)
25: $\prod^{XRS} \leftarrow \prod^{XRS} \setminus$ (getReadSet(TX_i), C_i)
26: $\prod^{TXCommit} \leftarrow \prod^{TXCommit} \cup TX_i$
27: SendReliable($\langle$REPLY$\rangle$, Ri, $Reply$)
28: Call **NEXT-T-EXEC**()
29:
30: **upon:** receive($\langle R{-}ASKFORCOMMIT, T_i, R_i, C_i, Reply \rangle \sigma$) from replica
31: acceptCommit ← WaitForAcceptance(timeout)
32: **if** (acceptCommit = 3f) **then**
33: Call **COMMIT-T** (T_i)
34: **else**
35: TO-Deliver($\langle PRE - PREPARE, R_i, Order, OrderTX, v, n, d \rangle \sigma$)
36:
37: **upon:** receive($\langle R - ASKFORCOMMIT - TX, TX_i, R_i, C_i, Reply \rangle \sigma$) from replica
38: acceptCommit ← WaitForAcceptance(timeout)
39: **if** (acceptCommit = 3f) **then**
40: Call **COMMIT-TX** (TX_i)
41: **else**
42: TO-Deliver($\langle PRE - PREPARE, R_i, Order, OrderTX, v, n, d \rangle \sigma$)

ser verificado, é a possibilidade de na fase certificação total (AskForCommit), a certificação ser feita utilizando um conjunto de transações (*batch*) ao invés de uma a uma.

VI. Avaliação

As avaliações foram feitas utilizando cinco computadores conectados por uma rede Local (LAN) 10/100. Dentre estes, quatro foram utilizados para processamento nas réplicas, com sistema operacional Linux Ubuntu Server 3.8.0-33-generic x86_84 Intel(R) Core I7 com oito núcleos 1.6GHz, com 12GB de memória e capacidade de 2 *threads* por núcleo. Um computador para simular os clientes, rodando sobre o sistema operacional Slackware Linux 2.6.37.6-smp 2 SMP i686 Intel(R) Core(TM)2 com quatro núcleos 3.00GHz GenuineIntel GNU/Linux, com 3GB de memória e capacidade de uma *thread* por núcleo. A representação do cliente no sistema se fez através de *threads*.

Os testes foram executados utilizando 100 requisições por cliente, sendo que cada requisição contém apenas uma operação. As operações consistem em incrementar um contador compartilhado, foram criados dois contadores *A* e *B*. Nos gráficos, quando dito operações conflitantes, significa que transações interativas possuem conflitos entre si, transações pré-declaradas (para fins de brevidade, nos gráficos será chamado declaradas) possuem conflitos entre si e que ambas transações possuem conflitos entre si (declaradas e interativas). Quando dito sem conflitos, significa que transações interativas possuem conflitos entre si, transações pré-declaradas possuem conflitos entre si mas não existem conflitos entre os dois tipos de transações (declaradas e interativas).

A figura 6 apresenta a taxa de *commits* por segundo, em um cenário com conflitos (transações interativas e declaradas alteram o mesmo dado). Percebe-se que as

Algoritmo 4 Protocolo Bizantino - PBFT.

1: **upon:** receive($\langle PRE-PREPARE, Ri, Order, OrderTX, v, n, d\rangle\sigma$) from replicas
2: **if** (PRE-PREPARE was accepted) **then**
3: **for all** ($R \in \prod^R$ minus his own replica) **do**
4: TO-Deliver($\langle PREPARE, Ri, Order, OrderTX, v, n, d\rangle\sigma$)
5:
6: **upon:** receive($\langle PREPARE, Ri, Order, OrderTX, v, n, d\rangle\sigma$) from replicas
7: **if** (PREPARE was accepted) **then**
8: **for all** ($R \in \prod^R$ minus his own replica) **do**
9: TO-Deliver($\langle COMMIT - ORDER, Ri, Order, OrderTX, v, n, d\rangle\sigma$)
10:
11: **upon:** receive($\langle COMMIT - ORDER, Ri, Order, OrderTX, v, n, d\rangle\sigma$) from replicas
12: **if** (COMMIT-ORDER was accepted) **then**
13: BCO $\leftarrow$ Order
14: Set all $T \in$ BCO as Work in Progress (WiP)
15: **for all** ($TX_i \in OrderTX$) **do**
16: **if** (TX_i Can Commit) **then**
17: Call **COMMIT-TX**(TX_i)
18: **else**
19: Call **ABORT-TX**(TX_i)
20: Call **NEXT-T-EXEC**()
21: **function** ABORT-TX(TX_i)
22: Abort transaction TX_i!
23: $\prod^{TXBegin} \leftarrow \prod^{TXBegin} \setminus TX_i.Begin$
24: $\prod^{XWS} \leftarrow \prod^{XWS} \setminus$ (getWriteSet(TX_i), C_i)
25: $\prod^{XRS} \leftarrow \prod^{XRS} \setminus$ (getReadSet(TX_i), C_i)
26: SendReliable($\langle$REPLY$\rangle$, Ri, $Reply$)
27: Call **NEXT-T-EXEC**()
28:
29: **function** RE-EXEC-T(T_i)
30: Begins T_i re-execution
31: BCO $\leftarrow$ BCO $\setminus$ T_i
32: **Call Commit-T** (T_i)
33:
34: **function** NEXT-T-EXEC( )
35: **if** (BCO > 0) **then**
36: Call **RE-EXEC-T**(getFirstElement(BCO))
37: **else**
38: Call **ATP**(getFirstElement($\prod^T$))

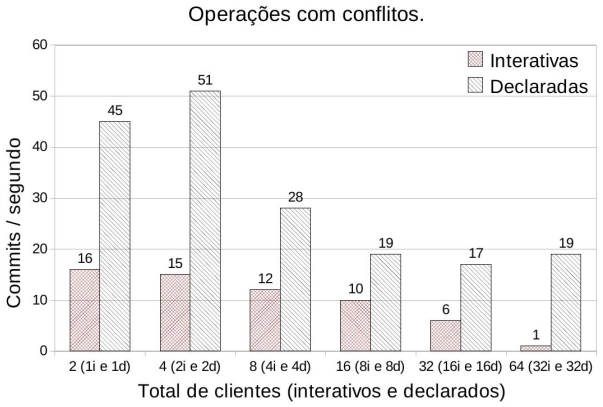

Figura 6: Transações pré-declaradas e interativas, com conflito de dados.

transações declaradas alcançam um número maior de *commits* do que as transações interativas. Isto se deve ao fato de que cada transação declarada consiste de apenas uma mensagem, enquanto cada transação interativa é composta de, no mínimo, três mensagens (*begin, operation* e *commit*). Considerando então a regra de espera[8], podemos dizer que transações interativas esperam mais vezes do que transações declaradas. Exemplificando, quando uma transação interativa está executando, duas ou três declaradas podem ser colocadas em espera, e serão executadas em sequência (pois estão na fila de espera) logo que a interativa terminar. Quando uma declarada está executando, uma interativa será colocada em espera, e será

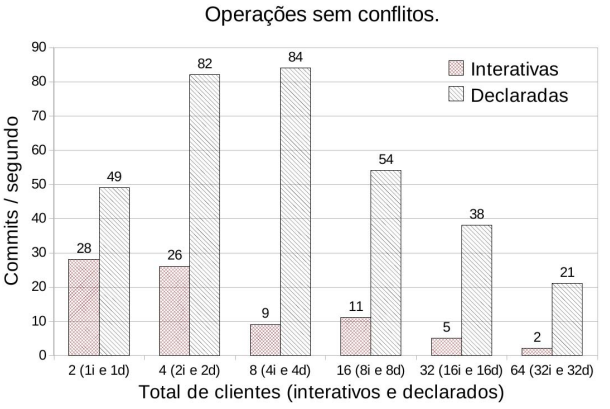

Figura 7: Transações pré-declaradas e interativas, sem conflito de dados.

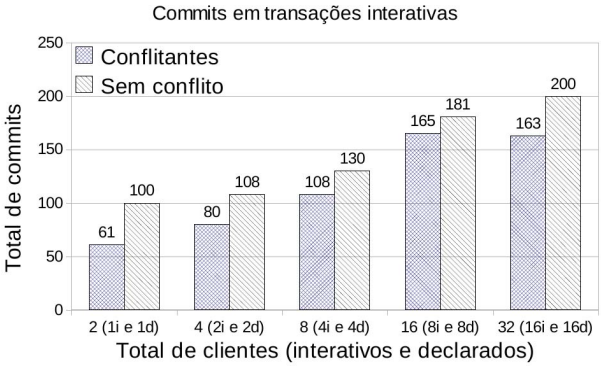

Figura 8: Total de commits em transações interativas.

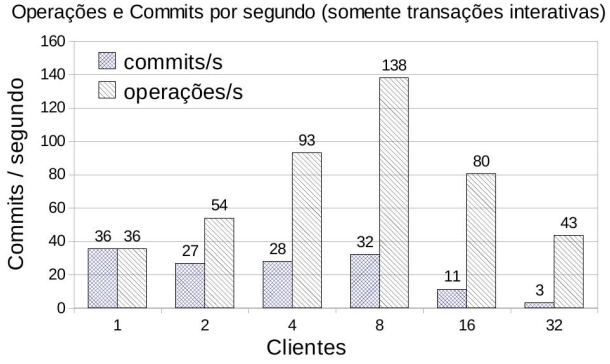

Figura 9: Transações interativas com conflito.

[8]Essa regra é exemplificada na seção IV-A: transações que chegam podem precisar esperar pelo término da execução de outra transação conflitante em execução.

executada quando a declarada terminar. Observamos ainda na figura 6 que em situações de alta disputa (32 clientes interativos), é mínima a taxa de efetivação de operações, sendo este um ponto de saturação.

A figura 7 demonstra novamente um melhor resultado para transações declaradas, em relação às transações interativas, que mantém seu ponto de saturação por volta de 32 clientes. A baixa taxa de confirmações por parte de transações interativas se dá pela quantidade de mensagens trocadas para finalização do protocolo.

Na figura 8 pode-se notar que, considerando apenas o total de *commits* realizados por transações interativas, existe um incremento de solicitações efetivadas, tanto em situações de conflitos quanto em situações sem disputa. A figura 9 apresenta a execução de transações interativas sem concorrência com transações declaradas. Nota-se que a concorrência entre as transações interativas faz com que apenas metade das solicitações sejam efetivadas, para dois clientes, na unidade de tempo de um segundo. No cenário com 32 clientes solicitando transações interativas, apenas três conseguem efetivar suas solicitações, num total de 43 operações realizadas na unidade de tempo considerada.

VII. Conclusão

Existem sistemas nos quais é fundamental a execução de tarefas de forma atômica. Memória transacional em software disponibiliza uma abstração denominada transação para realizar tarefas de acordo com a semântica atômica. Dentre os tipos de transações, consideram-se as interativas e as pré-declaradas. Transações interativas são importantes pois permitem em tempo de execução modificar a lógica da aplicação com base nos resultados obtidos; ao mesmo tempo, transações pré-declaradas são úteis pois permitem certa flexibilidade em relação ao momento em que serão executadas. Neste artigo apresentamos o Mesobi, um protocolo que permite a execução de transações interativas e pré-declaradas no mesmo ambiente. A implementação de um protótipo foi realizada e os resultados demonstram que é viável a execução simultânea de ambos os tipos de transações, sendo que as transações pré-declaradas apresentam uma melhor escalabilidade em relação às interativas.

Acknowledgment

Supported in part by Brazilian National Research Council (CNPq) through process 560258/2010-0 and Coordination of Improvement of Higher Level Personnel (CAPES) through process 400511/2013-4 PVE A039.

References

[1] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 4, no. 2, pp. 63–71, 1996.

[2] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[3] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on*. IEEE, 2009, pp. 307–313.

[4] D. Sciascia and F. Pedone, "RAM-DUR: In-Memory Deferred Update Replication," *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 81–90, Oct. 2012.

[5] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues, "SPECULA: Speculative Replication of Software Transactional Memory," *SRDS*, pp. 91–100, Oct. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6424843

[6] R. Palmieri, F. Quaglia, and P. Romano, "OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems," *SRDS*, no. 257784, pp. 59–64, Oct. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6076762

[7] N. Carvalho, P. Romano, and L. Rodrigues, "Scert: Speculative certification in replicated software transactional memories," in *Proceedings of the 4th Annual International Conference on Systems and Storage*. ACM, 2011, p. 10.

[8] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing," *2010 Ninth IEEE International Symposium on Network Computing and Applications*, pp. 20–27, Jul. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5598236

[9] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "An Optimal Speculative Transactional Replication Protocol," *International Symposium on Parallel and Distributed Processing with Applications*, pp. 449–457, Sep. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5634368

[10] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association, 2012, pp. 21–21.

[11] H. Zhang and W. Zhao, "Concurrent Byzantine Fault Tolerance for Software-Transaction-Memory Based Applications," *International Journal of Future Computer and Communication*, vol. 1, no. 1, 2012.

[12] F. Pedone and A. Schiper, "Optimistic Atomic Broadcast," *Distributed Computing. Springer Berlin Heidelberg*, no. 95, pp. 318–332, 1998.

[13] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *Knowledge and Data Engineering, IEEE Transactions on*, no. 4, 2003.

[14] R. Kotla and M. Dahlin, "High throughput Byzantine fault tolerance," *International Conference on Dependable Systems and Networks, 2004*, pp. 575–584, 2004.

[15] T. A. Ribeiro, L. C. Lung, and H. V. Netto, "OB-STM: An Optimistic Approach for Byzantine Fault Tolerance in Software Transactional Memory," *Simpósio Brasileiro de Engenharia de Sistemas Computacionais - SBESC*, 2013.

[16] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" *Proceedings of the 3rd ACM workshop on Cloud computing security workshop - CCSW '11*, p. 113, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2046660.2046682

[17] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172–185, 2006.

[18] R. Guerraoui and L. Rodrigues, *Reliable Distributed Programming*. Springer, 2006, vol. 138.

[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[20] L. Lamport and M. Fischer, "Byzantine generals and transaction commit protocols," Technical Report 62, SRI International, Tech. Rep., 1982.

[21] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[22] R. A. Bazzi and Y. Ding, "Non-skipping timestamps for byzantine data storage systems," in *Distributed Computing*. Springer, 2004, pp. 405–419.

[23] P. A. Bernstein and N. Goodman, "Serializability theory for replicated databases," *Journal of Computer and System Sciences*, vol. 31, no. 3, pp. 355–374, 1985.