**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE**

Tulio Alberton Ribeiro

# MESOBI: MEMÓRIA TRANSACIONAL EM SOFTWARE TOLERANTE A FALTAS BIZANTINAS

Florianópolis(SC)

2015

Tulio Alberton Ribeiro

# MESOBI: MEMÓRIA TRANSACIONAL EM SOFTWARE TOLERANTE A FALTAS BIZANTINAS

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Prof. Lau Cheuk Lung, Dr.

Florianópolis(SC)

2015

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://portalbu.ufsc.br/ficha

Tulio Alberton Ribeiro

**MESOBI: MEMÓRIA TRANSACIONAL EM SOFTWARE TOLERANTE A FALTAS BIZANTINAS**

Esta Dissertação de Mestrado foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis(SC), 3 de março 2015.

---

Prof. Ronaldo dos Santos Mello, Dr.
Universidade Federal de Santa Catarina
Coordenador do Programa

**Banca Examinadora:**

---

Prof. Lau Cheuk Lung, Dr.
Orientador
Universidade Federal de Santa Catarina

---

Prof. Lásaro Jonas Camargos, Dr.
Universidade Federal de Uberlândia

---

Prof. Antônio Augusto Medeiros Fröhlich, Dr.
Universidade Federal de Santa Catarina

---

Prof. Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina

Aos meus familiares.
Telmo, pai. Cátea, mãe. Eduardo,
irmão. Telmo, irmão.

## AGRADECIMENTOS

Gostaria imensamente de poder citar todos, mas se começar por uns e acabar não terminando em outros, chato esses ficarão. Sendo assim, agradeço inefavelmente àqueles que passaram por minha vida em mais essa jornada, o mestrado. Com certeza se não houvessem pessoas tão altruístas no LaPeSD - Laboratório de Pesquisa em Sistemas Distribuídos, essa dissertação não seria o que é. Não poderei deixar de agradecer ao meu grande amigo e co-autor Hylson Vescovi Netto, meu Orientador Lau Cheuk Lung e o CNPq.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

## SUMÁRIO

# 1 INTRODUÇÃO

Nas últimas décadas, muito do ganho em desempenho pelos processadores deve-se ao aumento da frequência do processador. Contudo, o aumento da frequência do processador reduziu-se e estagnou devido a limitações físicas e o foco voltou-se para uso de múltiplos processadores com múltiplos núcleos. A utilização de processadores com múltiplos núcleos permite aumento de desempenho em aplicações que exploram paralelismo de dados através do uso de múltiplas *threads*. Tais aplicações necessitam de controle de concorrência no acesso a dados compartilhados. O controle de concorrência tradicional é alcançado com a combinação de *locks* e condições, tais como, monitores.

Os sistemas de Memória Compartilhada Distribuída (DSM - *Distributed Shared Memory*) surgiram como uma abstração para facilitar o acesso a dados concorrentes, sem a necessidade de uso explícito[1] dos mecanismos clássicos de controle de concorrência como: *locks*, semáforos e algoritmos de exclusão mútua (PROTIC; TOMASEVIC; MILUTINOVIC, 1996). Entretanto, a abordagem utilizada nas DSM para controle de concorrência sofre com alguns inconvenientes, detalhados a seguir.

Primeiro, o programador precisa decidir entre *lock* refinado, onde o *lock* é associado a um dado específico ou *lock* grosseiro, onde o *lock* é associado a um conjunto de dados. *Locks* grosseiros são mais fáceis de implementar mas permitem pouca ou nenhuma concorrência. Dessa forma, o potencial dos proces-

[1] Internamente as DSM podem utilizar *locks* como controle de concorrência mas não ficam a cargo do desenvolvedor tratá-los e sim do programador da DSM.

sadores com múltiplos núcleos não é totalmente explorado. Já com o uso de *locks* refinados, a implementação é mais complexa, pois necessita que as *threads* adquiram todos os *locks* necessários.

Segundo, questões básicas como o mapeamento de dados e *locks*, ou seja, qual *lock* protege qual dado e a ordem em que os *locks* são adquiridos e liberados são importantes. Violações no acesso aos dados são difíceis de detectar e corrigir. Por essas razões é difícil desenvolver, corrigir, detectar e manter aplicações concorrentes.

Como alternativa para a sincronização baseada em *locks*, surgiram as Memórias Transacionais em Software (STM - *Software Transactional Memory*). As STM permitem acesso uniforme às escritas e leituras através de transações. Transações em STM são semelhantes a transações em bancos de dados, porém, sem a propriedade durabilidade. A abordagem utilizada nas transações permite que programadores definam seções de código como atômicas. Uma seção ou bloco de código definido como atômico significa que: ou todo bloco atômico é executado ou nada dentro do bloco atômico é executado. Em caso de insucesso na execução de um bloco atômico, as alterações parciais feitas pela execução não devem ser propagadas, ou seja, é como se não houvesse executado o código atômico dentro do bloco.

Devido as facilidades inerentes às STM, pesquisadores têm mostrado um aumento de interesse no controle de concorrência utilizando STM em um ambiente distribuído, como pode ser visto nos trabalhos $D^2STM$ (COUCEIRO; ROMANO, 2009), *RAM-DUR* (SCIASCIA; PEDONE, 2012), *SPECULA* (PELUSO et al., 2012), *OSARE* (PALMIERI; QUAGLIA; ROMANO, 2011), *SCert* (CARVALHO; ROMANO; RODRIGUES, 2011), *AGGRO* (PALMIERI; QUAGLIA; ROMANO, 2010), *STR* (ROMANO et al., 2010), Granola (COWLING;

LISKOV, 2012), *Cluster − STM* (BOCCHINO; ADVE; CHAMBERLAIN, 2008) e *Zhang* (ZHANG; ZHAO, 2012). 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 (dados) 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, trata 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 (PEDONE; SCHIPER, 1998; KEMME et al., 2003), mas necessitam que a ordem final de entrega de mensagens seja definida para concluir sua execução. Por sua vez, somente o trabalho de (ZHANG; ZHAO, 2012) explora 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 transações somente leitura podem ser abortadas devido a leituras inconsistentes.

Esta dissertação apresenta uma arquitetura de STM tolerante a faltas Bizantinas denominada Mesobi[2] que executa, em simultâneo, tanto transações interativas (ou *online*) quanto transações pré-declaradas. O Mesobi utiliza mecanismo otimista no processamento de transações. Também foi desenvolvido anteriormente ao Mesobi o OB-STM (seção 4.3), para tratamento apenas de transações pré-declaradas. Um mecanismo semelhante a execução paralela de operações não conflitantes proposta por (KOTLA; DAH-

[2]Mesobi - Memória Transacional em Software Tolerante a Faltas Bizantinas.

LIN, 2004) é utilizado em ambas abordagens, a saber, OB-STM e Mesobi.

## 1.1 PROBLEMA

O acesso concorrente a dados distribuídos através do uso explícito de *locks* é complexo e propenso a erros (HARRIS et al., 2007). Modelos de bases de dados têm possibilitado um avanço significativo no controle de concorrência, mas ainda carecem em desempenho. Para contornar tal carência, surgiram os modelos: Memória Transacional inicialmente em *hardware* e depois em *software*. Bases de dados e memória transacional em *software* utilizam o conceito de transações para resolver conflitos inerentes aos acessos a dados compartilhados diretamente em memória. Embora avanços na resolução de conflitos a dados compartilhados através de transações tenham se consolidado, modelos tolerantes a faltas arbitrárias ou bizantinas são necessários.

Estudos recentes (SCHROEDER; PINHEIRO; WEBER, 2009; SCHROEDER; GIBSON, 2007; COSTA et al., 2013) feitos nos centros de dados do *Google* revelam que é mais comum do que se previa a ocorrência de erros em *hardware* que passam despercebido pela aplicação. O corrompimento de *bits* em memórias *Dual Inline Memory Module* (DIMM) variam em torno de 8% ao ano segundo estudos do *Google*, mesmo utilizando ECC (*Error Correcting Codes*) os erros não foram detectados. Outro estudo realizado pela *Microsoft* com um milhão de consumidores, mostrou que faltas na unidade central de processamento (CPU) e no núcleo do *chipset* são frequentes (NIGHTINGALE; DOUCEUR; ORGOVAN, 2009). *Amazon S3*, por exemplo, teve seus serviços parados por oito horas devido ao corrompimento de um único

bit em algumas mensagens internas com informações de estado (CORREIA et al., 2012). Esses problemas podem ir muito além do relatado nesses casos, uma vez que empresas frequentemente não divulgam dados de incidentes nos quais há corrompimento de dados confidenciais. Esses problemas poderiam ter sido evitados utilizando tolerância a faltas Bizantinas.

A abordagem adotada na literatura para tolerar faltas bizantinas é a Replicação Máquina de Estados (RME) Tolerante a Faltas Bizantinas (BFT). Esta abordagem tem sido alvo de intensas pesquisas há mais de trinta anos (LAMPORT; SHOSTAK; PEASE, 1982a; SCHNEIDER, 1990). No entanto, somente no fim da década de 90 é que surgiram propostas que apresentaram viabilidade prática para tal abordagem (CASTRO; LISKOV, 2002; CORREIA; NEVES; VERISSIMO, 2004; KOTLA et al., 2007; STUMM et al., 2010; LUIZ; LUNG; RECH, 2014; CORREIA; VERONESE; LUNG, 2010). Protocolos BFT, em geral, são compostos pela replicação de um serviço através de um grupo de máquinas que se comunicam de modo a oferecer um serviço replicado que atenda aos requisitos de confiabilidade, integridade e disponibilidade, que são fundamentais para que se obtenha Confiança no Funcionamento (*Dependability* (AVIZIENIS; KELLY, 1984)). Nesta dissertação algumas características dos algoritmos BFT encontrados na literatura são adaptados para lidar com transações em DSM tolerantes a faltas.

A execução concorrente de transações interativas e pré-declaradas levando em conta a capacidade de tolerar faltas Bizantinas, juntamente com a necessidade de orquestrar o correto funcionamento das transações interativas e pré-declaradas, não é uma tarefa trivial. Quando busca-se tolerância a faltas maliciosas, o protocolo necessário para que o sistema funcione conforme

suas especificações torna-se mais complexo. Tal abordagem somente foi encontrada no trabalho de Zhang (ZHANG; ZHAO, 2012) onde o mesmo tolera faltas Bizantinas no contexto de Memória Transacional em Software. Sua estrutura se diferencia em alguns aspectos à proposta nessa dissertação. Os aspectos mais significativos são: aborto de transações somente leitura e a necessidade de muitos nós físicos para realizar acordo e execução, além de o mesmo não lidar com transações interativas.

Essa dissertação aborda o problema de tolerar faltas Bizantinas em ambientes de memória transacional em software. Utiliza processamento otimista de transações através da alta probabilidade de ordenação espontânea em redes locais (LAN). O protocolo Bizantino somente é executado quando mensagens são entregues fora de ordem ou quando há réplica(s) maliciosa(s) no sistema.

## 1.2 JUSTIFICATIVA

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. Nem todos os tipos de operações podem ser executadas de forma confidencial na réplica. Apenas algumas operações podem ser executadas de forma confidencial, são elas: média, desvio padrão e regressão; operações mais complexas ainda não podem ser feitas na réplica (NAEHRIG; LAUTER; VAIKUNTANATHAN, 2011). Transações interativas podem contornar a incapacidade de executar de forma confidencial operações na réplica; o processamento de determinadas operações é feito no cliente, sendo assim, as réplicas não têm

conhecimento do que está sendo executado. 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. Essa dissertação não trata da execução confidencial de operações mas cria um protocolo onde as mesmas poderiam ser executadas interativamente em ambientes hostis.

## 1.3 OBJETIVOS

- Objetivo Geral
  - Modelar uma arquitetura tolerante a faltas Bizantinas em Memória Transacional em Software.
- Objetivos Específicos
  - Definir um protocolo que seja capaz de lidar com transações interativas e pré-declaradas em conjunto, definindo prioridades sobre estas.
  - Verificar as diferenças entre os modelos de transações pré-declarado e interativo.

– Analisar a taxa de ordenação espontânea pela rede.

## 1.4 METODOLOGIA

A sequência abaixo explana a metodologia de trabalho que será utilizada na pesquisa e elaboração do projeto aqui apresentado.

- Elaboração do projeto lógico (desenho do modelo e passos de comunicação).
- Pesquisa e levantamento das ferramentas que serão utilizadas no desenvolvimento do projeto, preferindo ferramentas livres mas sem por em risco a qualidade do projeto.
- Implementação do modelo em uma arquitetura específica.
- Realização de testes visando desempenho e consistência dos dados.
- Realização de testes comparativos.
- Publicação de artigos em congressos e periódicos.
- Elaboração do relatório técnico para o Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ).
- Escrita da dissertação.

## 1.5 ORGANIZAÇÃO DO TEXTO

A dissertação está organizada como segue; no Capítulo 2 é exposta uma breve fundamentação teórica para melhor compreensão dessa dissertação. No Capítulo 3 são discutidos os trabalhos relacionados. No Capítulo 4 é detalhado o funcionamento

do protocolo, arquitetura do protocolo, definições básicas do sistema, premissas e os algoritmos OB-STM e Mesobi. No Capítulo 5 são retratadas as avaliações e algumas otimizações que podem ser feitas. Por fim, as conclusões e contribuições são apresentadas no Capítulo 6.

# 2 FUNDAMENTAÇÃO

Para melhor compreensão dessa dissertação, nesse capítulo serão expostos: conceitos básicos em sistemas distribuídos, suas características, modelos de faltas, métodos de comunicação, coordenação, acordo, memória compartilhada distribuída, memória transacional e memória transacional em software.

## 2.1 CARACTERIZAÇÃO DE SISTEMAS DISTRIBUÍDOS

Formalmente, um sistema distribuído pode ser definido como: componentes de *hardware* e *software* localizados em redes de computadores que se comunicam e coordenam suas ações via passagem de mensagem (COULOURIS et al., 2012). Informalmente, sistemas distribuídos são constituídos de diferentes máquinas e infra-estruturas. A Internet por exemplo, pode ser vista como um enorme sistema distribuído, onde as máquinas diferem em processador, velocidade de processamento, fabricante, como também sua infra-estrutura de comunicação se diferencia em latência e vazão, entre outras características.

Processos podem ser definidos como unidades abstratas capazes de executar tarefas em sistemas distribuídos (GUERRAOUI; RODRIGUES, 2006). É considerado que o sistema é composto por $n$ processos identificados unicamente, denotados por $p_1, p_2, ..., p_n$ e o conjunto de processos compostos pelo sistema é expresso por $\prod^P$ É assumido que todos processos no sistema rodam o mesmo algoritmo local, e a soma dos processos constitui o algoritmo distribuído. A comunicação é feita por troca de mensagens utilizando *links* de comunicação e cada mensagem tem um identifica-

dor único no sistema.

Existem muitas maneiras em que um sistema ou aplicação pode parar de funcionar, o tipo de parada é definido no modelo de faltas. Quando no estado correto de funcionamento, uma aplicação ou sistema deve operar dentro de suas especificações e domínios, conhecidos como tempo ($TS$) e valor ($VS$). As respostas às solicitações feitas pela aplicação devem estar dentro desses dois domínios, ou seja, $(vs_i \in VS_i) \wedge (ts_i \in TS_i)$. Quando a resposta à solicitação não se encaixa no domínio do tempo, o modelo é classificado como *Fail-stop Faults* e quando as respostas se encaixam em ambos domínios, tempo e valor, mas não é correta $(vs_i \notin VS_i) \wedge (vs_i \in VS) \wedge (ts_i \in TS_i)$, o modelo é classificado como *Byzantine-Faults* (LAMPORT; SHOSTAK; PEASE, 1982b). Segundo Avizienis (AVIZIENIS et al., 2004), um serviço faltoso, frequentemente abreviado como falta, é um evento que ocorre quando a resposta do serviço se desvia de sua correta execução (definida em suas especificações). Um serviço falta se ele não mais condiz com sua especificação funcional ou suas especificações não mais se adequam as funções descritas no sistema.

A Figura 1 representa a hierarquia dos modelos de faltas.

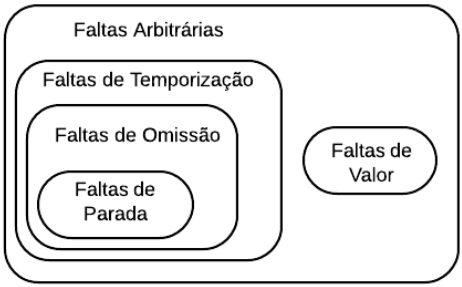


Figura 1: Classificação de Faltas.

Detalhes dos modelos são descritos abaixo:

- Faltas de parada: é o modelo de faltas mais simples, caracterizado pela parada do serviço, quando o processo falta por parada é dito que o processo sofreu *crash*. O sistema pode parar de responder, por exemplo, devido a queda de energia, o processo que executava o serviço pode ter caído, ou pode estar sobrecarregado (acessos simultâneos) ou outros infortúnios. Quando uma falta de parada acontece, o nodo faltoso não mais interage com o resto do sistema. A tolerância a faltas de parada se faz geralmente por replicação. Esse modelo, por exemplo, não tolera erros incorridos em uma execução errada de um serviço, tal como o cálculo errado de uma solicitação (SCHLICHTING; SCHNEIDER, 1983).

- Faltas de omissão: uma falta de omissão ocorre quando um processo deixa de executar alguma ação, tal como enviar, receber ou entregar alguma mensagem que supostamente deveria ser executada pelo algoritmo consoante suas especificações (DEFAGO; SCHIPER; URBAN, 2004; GUERRAOUI; RODRIGUES, 2006). Falta de omissão é geralmente ocasionado por *buffer overflows* ou congestionamento na rede, resultando em perda de mensagens. A perda de mensagens leva à omissão na execução de algumas operações que o algoritmo deveria executar, levando ao desvio das suas especificações (GUERRAOUI; RODRIGUES, 2006).

- Faltas de temporização: uma falta de temporização acontece quando o processo viola alguma restrição temporal imposta pela especificação (somente acontece em sistemas síncronos e parcialmente síncronos) (DEFAGO; SCHIPER; URBAN, 2004).

- Faltas arbitrárias, maliciosas ou Bizantinas: o modelo de faltas Bizantinas é mais complexo. O modelo permite que nodos continuem interagindo com o sistema, apesar de faltoso. As faltas Bizantinas são caracterizadas pelas respostas dentro dos dois domínios, tempo e valor, mas o valor da resposta não é correto, podendo ter sido gerado por erros de *hardware* ou *software*, tanto como de modo intencional (malicioso) (LAMPORT; SHOSTAK; PEASE, 1982b). Faltas arbitrárias são as mais caras e complexas para se tolerar, mas muitas vezes necessária para evitar que sistemas/processos sejam controlados por usuários maliciosos (GUERRAOUI; RODRIGUES, 2006). Uma falta para ser considerada arbitrária, não necessariamente precisa ser maliciosa ou intencional: pode ser causada por erros de implementação, erros no compilador, na linguagem de programação, inversão de *bits* em memória entre outros. Devido às faltas arbitrárias, mensagens incorretas podem ser trocadas, e se não houver tratamento o sistema pode vir a parar ou corromper-se.

### 2.1.1 Comunicação

A abstração de comunicação é utilizada para representar os componentes de rede em um sistema distribuído. Em especial, a confiabilidade entre os canais de comunicação que interligam os nodos, será analisada de modo independente da topologia de rede utilizada nessa dissertação. As mensagens poderão ser enviadas[1] ou entregues[2], diferindo de recebida, onde a mensagem pode ter

[1] Processo envia mensagem para outro processo.

[2] Quando utilizado o termo entregue, significa que: os processos corretos receberam a mensagem, analisaram e só então repassaram a mensagem para aplicação.

sido recebida (pela placa de rede), mas ainda não foi analisada ou repassada para aplicação.

Em ambientes distribuídos é possível que mensagens sejam perdidas quando transmitidas através da rede, isso é conhecido como falha de conexão (*link failure*). Contudo, é razoável assumir que a probabilidade de uma mensagem alcançar seu destino não seja zero, isso porque é muito incomum que todas as mensagens trocadas entre dois processos sistematicamente sejam perdidas, a não ser que haja falha na rede (por exemplo: partição na rede)(GUERRAOUI; RODRIGUES, 2006). Uma maneira para contornar a perda de mensagens é retransmitir a mensagem. As propriedades que definem os modelos de falhas estão caracterizados abaixo:

- *Fair-Loss Links*: a propriedade *Fair-Loss Links* garante que o canal de comunicação não perderá sistematicamente as mensagens enviadas. Mais precisamente, o canal pode ser caracterizado pelas seguintes propriedades (GUERRAOUI; RODRIGUES, 2006):
  - $FL_1$: se uma mensagem $m$ é enviada infinitamente pelo processo $p_i$ para o processo $p_j$, e nem $p_i$ ou $p_j$ sofre *crash*, então $m$ é entregue um infinito número de vezes por $p_j$.
  - $FL_2$: se uma mensagem $m$ é enviada em um número finito de vezes pelo processo $p_i$ ao processo $p_j$, então $m$ não pode ser entregue infinitas vezes pelo processo $p_j$.
  - $FL_3$: se uma mensagem $m$ é entregue pelo processo $p_j$, então $m$ foi previamente enviada para $p_j$ pelo processo $p_i$.

Portanto, se nenhum dos processos remetente e destinatário sofrem *crash*, e se a retransmissão é mantida, certamente a mensagem será entregue.

- *Stubborn Links*: esta abstração de comunicação esconde os mecanismos de retransmissão de mensagens usados pelos processos, quando usam o modelo *Fair-Loss Links* para garantir que suas mensagens chegarão ao destino (GUERRAOUI; RODRIGUES, 2006). As propriedades abaixo devem ser cumpridas:
  - $SL1$: *Stubborn delivery*: supõe-se que $p_i$ é qualquer processo correto que envia uma mensagem $m$ para um processo correto $p_j$. Se $p_i$ não sofre *crash*, então $p_j$ entrega $m$ um infinito número de vezes.
  - $SL2$: *No creation*: se uma mensagem $m$ foi entregue por algum processo $p_j$, então a mensagem $m$ foi previamente enviada a $p_j$ por algum processo $p_i$.
- *Perfect Links*: caracteriza-se por uma melhoria no modelo *stubborn links* onde são adicionados mecanismos para detectar e suprimir mensagens duplicadas (GUERRAOUI; RODRIGUES, 2006). São caracterizados pelas propriedades abaixo:
  - $PL1$: *Reliable delivery*: se um processo $p_i$ envia uma mensagem $m$ a um processo $p_j$, e nem $p_i$ ou $p_j$ sofre *crash*, então $p_j$ receberá $m$.
  - $PL2$: *No duplication*: nenhuma mensagem é entregue por um processo mais de uma vez.
  - $PL3$: se a mensagem $m$ foi entregue por um processo $p_j$, então $m$ foi previamente enviada a $p_j$ por algum processo $p_i$.

### 2.1.2 Especificações sobre *Broadcast*

Nesta seção apresentaremos algumas especificações sobre envio confiável, onde serão consideradas confiáveis as mensagens que satisfizerem as seguintes propriedades: (*i*) todos processos corretos concordam no conjunto de mensagens que eles entregam, (*ii*) todas as mensagens enviadas por processos corretos são entregues, e (*iii*) nenhuma mensagem espúria é entregue.

Embora essas propriedades possam ser suficientes para algumas aplicações, nenhuma ordem na entrega de mensagens é imposta no modelo confiável. Em algumas aplicações a ordem é importante. Quando a ordem necessita ser respeitada é preciso utilizar modelos mais confiáveis como: *FIFO Broadcast* ou *Atomic Broadcast*.

Informalmente, *Reliable Broadcast* requer que todos processos corretos entreguem o mesmo conjunto de mensagens (*agreement*), e que estejam inclusas neste conjunto todas as mensagens enviadas pelos processos corretos (*validity*) sem mensagens espúrias (*integrity*) (HADZILACOS; TOUEG, 1994). Formalmente *Reliable Broadcast* é definido utilizando duas primitivas: *broadcast* e *deliver*, e satisfaz as seguintes propriedades:

- *Validity*: se um processo correto envia uma mensagem $m$, então certamente ela será entregue (*deliver*).

- *Agreement*: se um processo correto entrega a mensagem $m$, então todos processos corretos entregarão $m$.

- *Integrity*: para qualquer mensagem $m$, todo processo correto entrega a mensagem $m$ ao menos uma vez, e somente se $m$ foi previamente enviada pelo remetente.

- *Validity*: juntamente com *Agreement* asseguram que toda mensagem enviada por um processo correto será entregue por todos os processos corretos (HADZILACOS; TOUEG, 1994).

Na ordem FIFO (*FIFO Broadcast*) os processos corretos devem entregar as mensagens na mesma ordem em que foram enviadas. A propriedade FIFO consiste de: se um processo envia uma mensagem $m$ antes da mensagem $m'$, então nenhum processo correto entrega a mensagem $m'$ sem antes ter entregado a mensagem $m$ (HADZILACOS; TOUEG, 1994).

A ordem causal assegura que uma mensagem não é entregue até que todas as mensagens de que possivelmente depende tenham sido entregues. O protocolo que garante ordem causal na entrega das mansagens é conhecido como *Causal Broadcast* e é definido como um *reliable broadcast* que satisfaz:

- *Causal Order*: se o envio causal de uma mensagem $m$ precede o envio de uma mensagem causal $m'$, então nenhum processo correto pode entregar $m'$ antes de ter entregue $m$. A ordem causal é uma generalização da ordem FIFO, sendo que, pode ser definida como a equivalência da ordem FIFO e ordem local (*Local Order*) (HADZILACOS; TOUEG, 1994).

- *Local Order*: se um processo envia uma mensagem $m$ e um processo entrega $m$ antes de enviar $m'$, então nenhum processo correto pode entregar $m'$, salvo se já tenha entregado $m$.

O *Atomic Broadcast* requer que todos processos corretos entreguem todas as mensagens na mesma ordem. Os requisitos de acordo e ordem total do *atomic broadcast* implicam que processos corretos certamente entregarão a mesma sequência de mensagens

(HADZILACOS; TOUEG, 1994). O *Atomic Broadcast* satisfaz o seguinte requisito:

- *Total Order*: se os processos corretos $p$ e $q$ entregam a mensagem $m$ e $m'$, então $p$ entrega $m$ antes de $m'$ se e somente se $q$ entregar $m$ antes de $m'$.

### 2.1.3 Especificações sobre Tempo

Em ambientes distribuídos diversos modelos de sincronia são definidos na literatura. Algumas aplicações necessitam conhecer de antemão os tempos do sistema, sendo esses: tempo de transmissão de mensagens, processamento de mensagens, tempo de resposta, processamento do algoritmo entre outros. Os modelos assíncrono, síncrono e parcial síncrono serão detalhados a seguir.

Um sistema assíncrono consiste em não fazer qualquer suposição de tempo sobre os canais de comunicação e processos (GUERRAOUI; RODRIGUES, 2006). Isto é, os processos e canais de comunicação não têm limites de tempo sobre sua execução e atrasos na comunicação, não existem relógios físicos limitando o tempo gasto por um processo ou na transmissão de mensagens. Mas mesmo sem a existência de relógios físicos, é possível verificar a ordem causal em que as mensagens foram enviadas, recebidas e entregues. O tempo medido em função de ordem causal é conhecido como *logical time* e esta noção de relógio é chamado de *logical clock* (LAMPORT, 1978).

Nos ambientes síncronos os tempos são conhecidos antecipadamente. Propriedades que definem sistemas síncronos:

- *Synchronous computation*: existe um limite superior para

o processamento do algoritmo. O limite de tempo tomado por qualquer processo para executar suas tarefas deve ser menor que o limite pré-estabelecido.

- *Synchronous communication*: existe um limite superior na transmissão de mensagens. O período de tempo entre o envio e o recebimento de uma mensagem deve ser inferior ao limite pré-estabelecido.

- *Synchronous physical clocks*: processos são ajustados para funcionarem com relógios físicos. Existe um limite no qual o relógio local pode se desviar do relógio global (relógio global marca o tempo real).

Sistemas parcialmente síncronos são vistos como um modelo híbrido, onde existem períodos de sincronia e assincronia. De modo geral, sistemas distribuídos aparecem como sistemas parcialmente síncronos. Mais precisamente, para a maioria dos sistemas conhecidos, é relativamente fácil definir os limites de tempo físico que são respeitados a maior parte do tempo (GUERRAOUI; RODRIGUES, 2006). Porém, esses limites de tempos não são respeitados para sempre (períodos de assincronia). Como por exemplo, sobrecarga de rede e processos com escassez de memória (*swap*), podem levar o sistema a se comportar de forma assíncrona.

### 2.1.4 Consenso

O consenso pode ser definido de uma maneira simples como os passos efetuados pelos processos para chegar a um acordo. Todos os processos corretos no sistema devem chegar ao mesmo resultado.

Podemos especificar o consenso através de duas primitivas: *propose* e *decide*. Para se alcançar consenso, cada processo $p_i$ no sistema inicia em um estado não decidido e propõe um único valor $v_i$. Os processos se comunicam um com outro trocando mensagens. Após essa fase, cada processo escolhe por um valor de decisão $d_i$ e entra em uma fase chamada *decided*, na qual o valor não pode mais ser alterado (COULOURIS et al., 2012).

Analisaremos três variantes do consenso: regular, uniforme, e randomizável ou aleatório. O Consenso Regular acontece da seguinte forma: cada processo propõe um valor inicial para acordo através da primitiva *propose*. Os processos trocam suas proposições e no final devem chegar a um acordo. Todos os processos corretos decidem sobre um único valor, através da primitiva (*decide*) (GUERRAOUI; RODRIGUES, 2006). O consenso regular, satisfaz as propriedades:

- $P_1$: *Termination*: todo processo correto decide algum valor.
- $P_2$: *Validity*: se um processo decide-se por $v$, então $v$ foi proposto por algum processo.
- $P_3$: *Integrity*: nenhum processo decide duas vezes.
- $P_4$: *Agreement*: dois processos corretos não decidem diferentemente.

O consenso uniforme é uma variante do consenso regular, o qual difere na propriedade quatro ($P_4$).

- $P_1$ - $P_3$: idem consenso regular.
- $P_4$': *Uniform Agreement*: dois processos não decidem diferentemente.

O consenso randomizável ou aleatório assegura as propriedades *integrity, agreement* e *validity* do consenso uniforme e varia na propriedade *termination*, onde é estipulado com uma probabilidade 1 (um) que cada processo correto decidirá.

- $A_1$: *Termination*: com probabilidade 1 (um), cada processo correto decide algum valor.
- $A_2$: *Validity*: se um processo decide por $v$, então $v$ foi proposto por algum processo.
- $A_3$: *Integrity*: nenhum processo decide duas vezes.
- $A_4$: *Agreement*: dois processos corretos não decidem diferentemente.

## 2.2 MEMÓRIA COMPARTILHADA DISTRIBUÍDA

As Memórias Compartilhadas Distribuídas (DSMs) permitem que dados remotos sejam acessados como se fossem locais. Isso é feito através do *middleware DSM* que captura as chamadas de acesso aos dados e converte em acessos remotos. A granularidade dos dados pode ser diferenciada. O clássico trabalho de (LI; HUDAK, 1989) descreve uma abordagem utilizando granularidade de página, ou seja, um conjunto de blocos de memória de tamanho fixo é alocados para cada página. Todos os blocos nesse sistema compõem uma única página virtual dividida entre os processadores que fazem parte do grupo. Os tratamentos aos acessos paralelos à dados compartilhados são tratados com o uso explícito dos algoritmos de exclusão mútua[3]. Cada acesso é feito como se fosse local, todo acesso a dados locais que não

[3] *locks*, Barreiras, Semáforos, ...

existirem retorna uma exceção. Toda exceção resulta na busca e transferência da página faltante para onde surgiu a exceção. A questão chave nas DSMs está em como manter e localizar os dados em diferentes réplicas. Como descrito no trabalho de (LI; HUDAK, 1989), os dois principais algoritmos são: baseados em *broadcast* e diretório.

Na abordagem baseada em *broadcast*, quando um dado remoto é acessado, uma mensagem é enviada para todos participantes solicitando cópia dos dados. Nessa abordagem nenhuma informação precisa ser mantida a respeito dos dados compartilhados, pois as mesmas são recuperadas a cada *broadcast*.

Na abordagem baseada em diretório, informação sobre os dados compartilhados são mantidas em um diretório global distribuído. Em cada participante existe informação suficiente para recuperar o dado não encontrado localmente. Essa abordagem é dividida em três organizações, são elas:

- Gerenciador Central: todas páginas possuem o mesmo gerenciador, sendo que o mesmo mantém o diretório global. Cada acesso a páginas faltantes é feito através do gerenciador. Com o crescimento do número de réplicas no sistema, o gerenciador pode se tornar um gargalo. Para contornar tal inconveniência, o gerenciador pode ser distribuído entre os participantes.

- Gerenciador Distribuído Fixo: no gerenciador fixo, as informações referente às páginas são armazenadas de forma distribuída entre os participantes. Essa distribuição é feita de forma fixa e nem sempre ideal. Os dados são particionados entre os participantes, sendo que cada um mantém informações sobre um conjunto de páginas.

- Gerenciador Distribuído Dinâmico: no modelo dinâmico cada página ao ser solicitada, passa a ser gerenciada pelo solicitante, cada participante passa a ser o gerenciador central dos dados solicitados.

Para contornar a dificuldade em tratar acessos concorrentes a dados com uso explícito de *locks*, surgiram as Memórias Transacionais.

## 2.3 MEMÓRIA TRANSACIONAL

Os modelos tradicionais de controle de concorrência ao acesso a dados utilizam mecanismos de exclusão mútua para lidar com acesso concorrente (*locks*, semáforos). Os modelos que utilizam *locks* não são adequados em ambientes distribuídos por degradarem o desempenho da aplicação[4] e também por causarem problemas conhecidos na área como: *deadlock*[5], *livelock*[6] e *inversão de prioridade*[7].

Novas alternativas surgiram para mitigar essa deficiência no controle de concorrência e sincronia de aplicações distribuídas.

[4] Quando uma transação invoca um *lock*, outras transações que precisam travar o objeto em questão precisam esperar pela liberação do *lock*; Ocasionando perda de desempenho.

[5] Um *deadlock* ocorre quando dois processos esperam por recursos diferentes em regiões críticas de forma circular. Por exemplo, a transação $T_1$ aguarda pelo recurso $B$ mas mantém *lock* do recurso A e a transação $T_2$ aguarda pelo recurso $A$ mas mantém *lock* do recurso $B$, ocasionando assim uma espera circular.

[6] *Livelocks* são semelhantes a *deadlocks* porém as *threads* não estão bloqueadas. Quando uma *thread* altera seu estado de pronto para que outra *thread* possa ter progresso e essa *thread* altera seu estado para que a anterior tenha progresso, acaba por nenhuma progredir. Seria como se duas pessoas estivessem tentando passar em um corredor, e ambas estão no mesmo lado, cada pessoa muda de lado para que a outra possa passar mas o impasse continua indefinidamente.

[7] Inversão de prioridade acontece quando um processo de maior prioridade é preemptado por um processo de menor prioridade.

Como alternativa, foi concebido o termo Memória Transacional, inicialmente em *hardware* com o trabalho de Herlihy (HERLIHY; MOSS, 1993). O modelo de Herlihy propunha um protocolo de coerência de *cache* em multiprocessadores. Com o avanço nas pesquisas, Shavit e Touitou (SHAVIT; TOUITOU, 1995) propuseram um novo modelo chamado Memória Transacional em Software, que ampliava o modelo antigo em *hardware* para um totalmente em *software*.

### 2.3.1 Transações

O objetivo de uma transação é assegurar que todos os dados gerenciados pela aplicação continuem em um estado consistente mesmo quando acessados por múltiplas transações e na presença de faltas de *crash* (COULOURIS et al., 2012). Uma transação pode ser especificada como um conjunto de operações indivisíveis executadas pela aplicação sobre os dados por ela gerenciados. As transações, por sua vez, são consideradas como execuções sequenciais, onde uma transação não percebe as alterações parciais feitas por outra transação simultânea. Por fim, as transações provêem um mecanismo pelo qual clientes podem especificar sequências de operações que são indivisíveis (atômicas). Com isso, mesmo na presença de outras transações, cada transação é vista pelo cliente como uma operação sequencial.

### 2.3.2 Propriedades

A semântica usada em memória transacional tem suas raízes nos sistemas de banco de dados. Em banco de dados, as propriedades que definem uma transação são: atomicidade, consistência,

isolação e durabilidade, mais conhecidos pelo acrônimo ACID. Como a propriedade durabilidade não existe em memória transacional, tem-se apenas as propriedades ACI.

- Atomicidade: consiste na execução completa da transação ou nada. Caso a transação não possa ser executada por completo, nada deve ser feito e qualquer alteração aos dados efetuada pela transação abortada devem ser desfeitas (isso se estiver utilizando *direct-update*).

- Consistência: garante que durante a execução de uma transação, as restrições impostas pelo sistema não são violadas. Por exemplo: chaves primárias, restrições, gatilhos ou outras propriedades que podem ser configuradas ao banco de dados. Antes do início de cada transação o sistema se encontra em um estado consistente. Por sua vez, serão expostas duas subdivisões referentes à consistência: forte e fraca:

  - *Consistência Forte*: após a atualização completa[8] de um dado, os acessos subsequentes a esse dado sempre retornarão o dado atualizado.
  - *Consistência Fraca*: devido a janela de inconsistência[9] que existe em algumas abordagens, pode acontecer que o valor retornado não seja o mais atual.

- Isolação: a propriedade isolação assegura que transações concorrentes são vistas pelo sistema como transações sequenciais (LARUS, 2006). Sendo que uma transação não interfere nos dados de outra transação em tempo de execução

[8] Dado alterado e confirmado (*committed*).

[9] Período entre a atualização de um dado e o momento em que ele será garantidamente observado com seu valor mais atual, é definido como *inconsistency window* ou janela de inconsistência.

(dependendo do modelo utilizado), somente após a confirmação da transação é que os dados podem ser vistos pela aplicação, ou seja, cada transação não sabe da existência de outra transação. A isolação possui duas subdivisões: fraca e forte:

  - *Isolação Fraca*: as variáveis que compõem o sistema podem ser acessadas de dentro ou de fora de um bloco transacional.
  - *Isolação Forte*: as variáveis dentro de uma transação só podem ser acessadas de dentro de um bloco transacional. Todas as operações fora de bloco atômico são convertidas automaticamente em operações transacionais individuais (LARUS, 2006).

- Durabilidade: após a confirmação de uma transação, suas alterações devem ser permanentes (salvas em memória durável), somente podendo ser alteradas por outra transação.

### 2.3.3 Memória Transacional em Software

Os Sistemas de Memória Transacional em Software (STMs) podem ser vistos como um sistema de acesso a dados concorrente não bloqueante. Em STM os programadores não precisam se preocupar com mecanismos explícitos de controle de concorrência, apenas necessitam delinear qual parte do código deve ser tratado como concorrente. Pode ser definido como uma abstração que pretende facilitar o desenvolvimento de aplicações concorrentes.

Os algoritmos de sincronização não bloqueantes são classificados em três principais categorias. São elas: *wait-freedom, lock-freedom* e *obstruction-freedom* (MARATHE; SCOTT, 2004). Essas

propriedades determinam o progresso em sistemas concorrentes:

- *Obstruction-freedom* (HERLIHY; LUCHANGCO; MOIR, 2003) é a mais fraca garantia de progresso em algoritmos de sincronização não bloqueantes, exclui a ocorrência de *deadlock* mas *livelocks* podem ocorrer. Para um algoritmo ser *obstruction-freedom*, em algum momento da execução do programa, uma única *thread* executará suas operações de forma isolada com um limitado número de passos até completar suas operações.

- *Lock-freedom* se encaixa como intermediário, garantindo a não existência de *livelock* mas ainda pode ocorrer *starvation* (MASSALIN; PU, 1992). Com *lock-freedom* é garantido que ao menos uma *thread* termine suas operações.

- *Wait-freedom* (HERLIHY, 1991) é a mais forte no sentido de garantir progresso em algoritmos de sincronização não bloqueante concorrente. Com *wait-freedom* é garantido que cada *thread* termine suas operações, não levando em conta os tempos de execução de outras *threads*. Essa propriedade exclui a ocorrência de *deadlock* e *inanição*[10] .

Já os algoritmos de sincronização bloqueantes garantem consistência utilizando exclusão mútua ou *lock* (MARATHE; SCOTT, 2004), o que em sistemas distribuídos reduz muito o desempenho e é difícil de implementar conforme a complexidade do sistema aumenta.

Diferenças entre transações em memória e bases de dados: Dados em BDs residem no disco mais do que em memória. O

[10] *Inanição* é um problema encontrado em programação concorrente. Acontece quando um processo é eternamente negado de acessar recursos necessários. Pode ocorrer devido ao escalonamento de processos por prioridades, processos com baixa prioridade podem sofrer de *starvation*

tempo de acesso em disco varia em torno de 5-10ms[11](LARUS, 2006). Esse tempo é suficiente para executar milhões de instruções em memória, dependendo do processador. A memória transacional não é persistente, ou seja, os dados não se mantém ao término da aplicação. Em contraste, BDs precisam armazenar os dados no disco antes da transação confirmar (*commit*) (LARUS, 2006).

Em memória transacional o termo granularidade refere-se à unidade de armazenamento na qual o sistema de memória transacional detecta conflitos. A maioria dos sistemas STM estendem uma linguagem baseada em objetos e implementam granularidade de objeto (LARUS, 2006). A granularidade de objeto detecta conflito de acesso aos objetos mesmo se as transações referenciam diferentes campos. Existem outras granularidades como *word* e *block*, onde a granularidade *word* detecta conflitos em acessos conflitantes em palavras (*word*) ou um grupo fixo de palavras. A granularidade *block* detecta conflitos em blocos de espaços de memória. Do ponto de vista do desenvolvedor, abordagens que utilizam granularidade em objeto são mais simples de implementar do que granularidade por bloco (*block*) (LARUS, 2006). Mas granularidade em palavra ou bloco permite ajustes mais precisos no dado compartilhado do que por objeto.

Todo dado atualizado por uma transação deve de alguma maneira refletir sua atualização ao resto da aplicação ou aplicações em um dado momento. Os modelos mais conhecidos e utilizados são: atualização direta (*Direct-Update*) e tardia (*Deferred-Update*).

- *Direct-Update*: na atualização direta a transação modi-

[11]Somente *Hard Disk Drives - HDD* sem considerar *Solid State Disk - SSD*.

fica imediatamente os dados por ela acessados. Caso a transação venha a ser abortada é necessário o uso de técnicas para restaurar (*roll-back*) os dados alterados. Esse método utiliza sincronização explícita para prevenir que outras transações leiam ou escrevam dados modificados (LARUS, 2006).

• *Deferred-Update*: na atualização tardia os dados são alterados utilizando uma cópia privada do mesmo e, somente quando a transação é confirmada, (*commit*), é que os dados são atualizados e propagados para outras transações (LARUS, 2006). A atualização tardia pode aumentar o uso de memória pelo uso de cópia privada dos objetos.

Quando um sistema executa mais de uma transação ao mesmo tempo (concorrente), é necessário algum mecanismo de sincronização para intermediar o acesso aos dados manipulados por transações diferentes mas, usando os mesmos dados (LARUS, 2006). Um conflito ocorre quando duas transações concorrentes executam operações no mesmo dado como detalhado na Tabela 1. Quando isso ocorre, somente uma transação deve poder modificar o dado de forma visível permanetemente para outras transações. As duas formas de controle de concorrência mais conhecidos são otimista e pessimista.

- No controle de concorrência otimista, a detecção e resolução de conflitos podem acontecer após o conflito ocorrer, devendo ser detectado antes da confirmação (*commit*). Permite assim que múltiplas transações acessem um objeto concorrentemente.

- No controle de concorrência pessimista, tão logo um conflito é detectado ele é resolvido. Permite que uma transação tenha acesso exclusivo a um dado (bloqueio), deste modo pre-

venindo que outras transações o acessem. Acesso exclusivo a um dado pode levar a *deadlocks*, mas existem maneiras de evitá-los.

A Tabela 1 identifica quando ocorre conflito entre dados compartilhados.

| Operações em transações diferentes utilizando os mesmos dados. | Conflito? | Motivo. |
|---|---|---|
| Leitura - Leitura | Não | A ordem em operações de (leitura - leitura) não interfere no resultado. |
| Leitura - Escrita | Sim | A ordem em operações de (leitura - escrita) afeta o resultado. |
| Escrita - Escrita | Sim | A ordem em operações de (escrita - escrita) afeta o resultado. |

Tabela 1: Operações em transações concorrentes.

A propriedade isolação, como dito anteriormente, implica que transações não acessem dados de outras transações não confirmadas. Por exemplo, caso a transação $T_1$ leia o dado $x$ e a transação $T_2$ escreva no dado $x$, sendo que $T_1 \prec T_2$, temos uma leitura-suja (*Dirty read*). Para contornar o problema de leitura-suja, é necessário detectar os conflitos inerentes as leituras e escritas. A detecção de conflitos entre transações pode ser de três formas:

• *detected on open*: acontece quando a transação declara a intenção de acessar um objeto (*opening* ou *acquiring*) ou na primeira referência do objeto na transação.

• *detected on validation*: a validação pode ocorrer a qualquer momento da execução da transação, e consiste no exame dos

dados que foram lidos ou atualizados por outra transação durante sua execução.

• *detected on commit*: acontece na fase de confirmação da transação, consiste na verificação de conflitos com os dados lidos ou atualizados por outras transações com a atual.

As duas abordagens de detecção de conflito mais conhecidas são *Early* e *Late*. Na detecção de conflitos *Early* são utilizadas as duas primeiras formas citadas acima *detected on open* e *detected on validation*; a detecção de conflitos *Late* utiliza a última forma *detected on commit*.

O mecanismo de gerenciamento de contenção é usado para assegurar progresso e evitar *livelock*, *starvation* e promover justiça em todo o sistema (SPEAR et al., 2009). A política utilizada no gerenciador de contenção serve para resolver os conflitos em transações, e sua política indica qual transação continuará, qual abortará ou aguardará (SPEAR et al., 2009). Tipicamente é definido de qual forma serão alcançados atomicidade e isolação dos dados, sendo elas *eager acquire* e *lazy acquire* as mais comuns. As políticas mais populares são:

• *Passive* (SPEAR et al., 2009): a transação se auto-aborta caso encontre conflito de escrita com uma transação concorrente.

• *Agressive* (III; SCOTT, 2004): o gerenciador de contenção opta por ignorar todas notificações dos métodos e sempre escolhe por abortar uma transação conflitante. Embora essa abordagem possa ser propensa a ocorrência de *livelock*, ela forma uma base para comparação com outras políticas de contenção.

• *Polite* (HERLIHY; LUCHANGCO, 2003): utiliza um exponencial *backoff* para solucionar conflitos, proporcional a $2^n$ nano segundos, sendo que $n$ é o número de tentativas que tentou confirmar a transação sem sucesso. Depois de 8 tentativas, a política

*Polite* aborta a transação que está competindo com ela.

• *Karma* (III; SCOTT, 2004): essa política verifica a quantidade de trabalho já realizado pela transação e, com base nesses dados, opta ou não por abortar a transação. A verificação pode ser feita contando a quantidade de blocos abertos pela mesma como uma estimativa do trabalho realizado. Nessa política é preferível abortar uma transação que acabou de iniciar do que uma mais antiga. Um rastreio dos blocos abertos é mantido pelo gerenciador, a cada *commit* a prioridade da transação é setada para zero, e a cada bloco aberto é incrementada a prioridade. Quando um conflito é encontrado, é verificada a prioridade das transações em questão e abortada a de menor prioridade, mas a cada transação abortada, a transação ganha um ponto, dando a ela maior chance de ser concluída ao ser re-executada.

• *Randomized* (III; SCOTT, 2004): o método mais simples de contenção utiliza probabilidade 50/50, ou seja, entre duas transações conflitantes, não levam-se em conta qualquer critério de desempate; todas transações tem 50% de chance de confirmar. E antes de ser reiniciada, a transação aguarda um tempo randômico variando em valores fixos.

• *Greedy* (GUERRAOUI; HERLIHY; POCHON, 2005): cada transação recebe um *timestamp* o qual retém mesmo se a transação vier a abortar e reiniciar. Transações com *timestamp* mais velhos têm prioridade sobre transações com *timestamp* mais novos. Cada transação possui um campo contendo o estado da transação: *active, committed* ou *aborted*. Uma transação muda seu estado de duas formas, de (*active* → *committed*) ou de (*active* → *aborted*) utilizando *CAS - (Compare-And-Swap)*.

• *Eruption* (III; SCOTT, 2004): semelhante ao gerenciador de contenção *Karma*, utiliza o número de blocos abertos como

uma medida grosseira de progresso. A cada bloco aberto com sucesso, a transação ganha um ponto *momentum* (prioridade). Consoante o número de blocos abertos aumenta, maior a probabilidade de ela não abortar.

• *KillBlocked* (III; SCOTT, 2004): o gerenciador marca uma transação como bloqueada quando notificado pela primeira vez da tentativa de abrir um bloco sem sucesso. O gerenciador aborta uma transação concorrente quando; (a) a transação concorrente também está bloqueada, ou (b) o tempo máximo de espera expirou.

• *KinderGarten* (III; SCOTT, 2004): baseia-se no revezamento de acesso aos blocos compartilhados. Para cada transação $T$ o gerenciador mantém uma lista (inicialmente vazia) das transações conflitantes nas quais $T$ já havia abortado. Em tempo de verificação de conflito, o gerenciador checa a transação conflitante e aborta caso esteja na lista. Caso contrário, ele adiciona a transação conflitante na lista e aguarda (*backoff*) por um período de tempo. Se depois de um número fixo de *backoffs* a transação conflitante ainda estiver aguardando, o gerenciador abortará a transação $T$.

• *TimeStamp* (III; SCOTT, 2004): o gerenciador que utiliza a política de *timestamp* tenta ser o mais justo possível com as transações. O gerenciador registra a hora de início de cada transação. Quando o gerenciador encontra contenção entre as transações $T$ e $Z$, o *timestamp* é comparado como critério de desempate. Se o *timestamp* de $T$ é anterior ao de $Z$, $Z$ é abortada. Após ser abortada, o gerenciador aguarda por uma série de intervalos fixos, e após metade desse tempo, a transação $Z$ é setada como um possível defunto. Se depois de um número máximo de intervalo a transação continuar sendo setada como defunto, o

gerenciador abortará as outras transações e deixará a transação setada como defunto ser confirmada. O objetivo do *timestamp* é evitar abortar transações antigas. Também é possível distinguir através da *flag defunct* as transações que estão vivas das que já foram abortadas.

# 3 TRABALHOS RELACIONADOS

Esse capítulo sintetiza os trabalhos relevantes para a construção dessa dissertação. A Tabela 2 encontra-se ao final desta seção e resume as principais características dos trabalhos relevantes em STM. Os pontos aqui abordados inicialmente focam nos trabalhos individuais existentes e posteriormente retratam as semelhanças entre a proposta aqui apresentada e a literatura existente. Os trabalhos que mais se assemelham à abordagem que será apresentada serão tratados mais profundamente.

## 3.1 JVSTM

*JVSTM* (CACHOPO; RITO-SILVA, 2006) consiste em uma biblioteca que implementa Memória Transacional em Software utilizando o conceito de caixas de versão. O conceito de caixas de versões, ou *snapshot*, permite que transações concorrentes acessem os mesmos itens do banco de dados, e conflitos sejam resolvidos[1] somente na fase de confirmação. Cada caixa de versão contém os dados compartilhados e é etiquetada utilizando um identificador[2], como pode ser visto na Figura 2. As caixas de versão podem ser vistas como um histórico de valores que foram atualizados por transações. Cada caixa pode conter mais de um item de dados, sendo que o último e mais atual é retornado ao usuário, sendo que que o número de versão identifica o item e não a caixa.

O protocolo executado pela *JVSTM* captura o início e

[1] Deferred-update, a verificação dos conflitos apenas é verificada na fase de *commit*.

[2] Inteiro incrementado a cada transação confirmada.

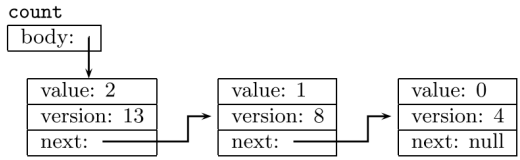


Figura 2: JVSTM caixas de versão (CACHOPO; RITO-SILVA, 2006).

fim de uma transação através das primitivas *Transaction.Begin()* e *Transaction.Commit()*. Cada nova transação cria em seu espaço privado de memória uma cópia do dado que será utilizado em sua transação; o dado copiado é o da última transação confirmada. Ao fazer a cópia dos dados, um novo identificador é definido para demarcar a nova caixa e esse identificador é armazenado no campo *version*.

Após a execução da transação é necessário fazer a validação dos dados que foram copiados para que a confirmação (*commit*) possa ocorrer. Essa validação é feita utilizando CAS (*Compare-And-Swap*), que por sua vez, compara a caixa de versão copiada na memória privada com a caixa existente na memória pública. Caso a transação venha a ser confirmada[3], o identificador em questão é setado com o da transação confirmada, e novas transações buscarão os dados consoante o identificador definido.

Para a execução de transações aninhadas, as atualizações feitas pela transação pai são propagadas para as transações filhas, e assim sucessivamente. Transações aninhadas são sempre confirmadas, e seu conjunto de escrita é agregado ao da transação pai. Somente com a confirmação da transação pai é que os dados modificados por transações aninhadas são propagados. Caso

[3]Não existem conflitos, ou seja, o identificador é o mesmo da caixa copiada com o existente.

uma transação pai seja abortada, todas as transações aninhadas também o serão.

A vantagem de utilizar a $JVSTM$ está no alto desempenho em operações somente leitura, sendo que as mesmas não abortam. Transações somente leitura não abortam visto que os objetos lidos são capturados em um estado consistente dos dados. A biblioteca $JVSTM$ foi implementada em Java, não é distribuída e trata concorrência entre transações que estejam executando na mesma JVM.

## 3.2 D2STM

$D2STM$, (COUCEIRO; ROMANO, 2009), implementa um sistema de Memória Transacional em Software Distribuído tolerante a faltas de parada, utiliza difusão atômica, *1-copy serializability*[4] e é baseado no JVSTM. A Figura 3 mostra a arquitetura do $D^2STM$: a camada principal é o gestor de replicação, que faz a coordenação entre as réplicas para manter a semântica de seriabilidade. A interceptação das indicações para iniciar e terminar uma transação (*begin* e *commit*) é feita pelo gestor de replicação também.

Cada início de transação é capturado através das primitivas *Transaction.Begin* e *Transaction.Commit* e são executadas localmente. Ao capturar o início de uma transação (*Begin*), uma cópia dos dados é feita conforme a lógica da $JVSTM$ e sua execução é iniciada. Após a execução local, a etapa de *commit* se inicia, com isso a transação é verificada localmente. Caso a verificação local se confirme, a etapa de verificação global se inicia.

[4]Uma cópia do objeto por nodo e os nodos corretos executam as mesmas sequências de operações.

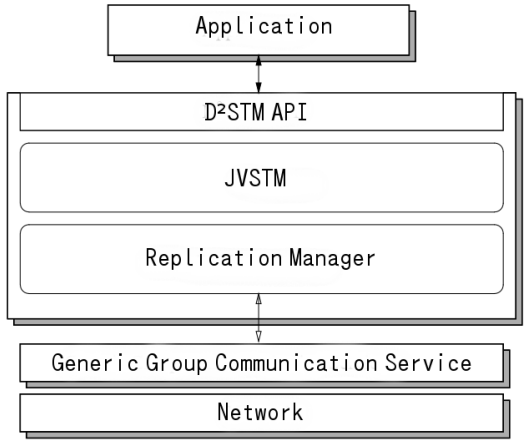


Figura 3: Arquitetura de uma réplica D2STM (COUCEIRO; ROMANO, 2009).

A fase de certificação global é feita utilizando filtro de *Bloom*, detalhes sobre filtros de *Bloom* podem ser vistos no Capítulo 5 Seção 5.4. Após a validação local, o gestor de replicação codifica o conjunto de leitura em um filtro de *Bloom*, juntamente com o conjunto de escrita, esse não codificado e os envia por difusão atômica para as réplicas.

Nos protocolos de certificação sem votação[5] clássicos (PEDONE; GUERRAOUI; SCHIPER, 2003), as transações de escrita são validadas em função da sua entrega através de difusão atômica juntamente com seus conjuntos de leitura e escrita. Na $D^2STM$, um mecanismo semelhante é utilizado; com o recebimento das transações por difusão atômica, cada réplica verifica se o filtro de *Bloom* de uma transação contém algum item atualizado por transações com um identificador de versão maior. Caso isso

[5]Na certificação sem votação cada réplica tem informação suficiente para validar ou não a transação.

ocorra, a transação deve ser abortada; caso contrário, a mesma pode ser confirmada. Essa verificacão utilizando filtro de *Bloom* é feita transação a transação no *buffer* atômico de transações. Lembrando que a execução de uma transação se dá em apenas uma réplica, e caso a transação venha a ser confirmada, seu conjunto de escrita é repassado para as réplicas através de difusão atômica. Importante ressaltar que a *thread* que executou a transação é bloqueada na fase de confirmação e só é desbloqueada ao final da mesma.

As principais contribuições da $D^2STM$ estão no modelo utilizado para fazer a verificação de conflitos entre réplicas. Nos modelos tradicionais, os conjuntos de leitura e escrita são propagados junto com a solicitação de confirmação. Na $D^2STM$ isso é feito através da combinação da técnica de atualização tardia [6] com a utilização de filtro de *Bloom* como meio para detectar conflitos de escrita/leitura. A utilização de filtros de *Bloom* pode ocasionar falsos positivos, ou seja, existe uma taxa de erro em que a verificação pode resultar em *aborts* que não deveriam ocorrer. Os falsos positivos apenas geram uma taxa maior de *aborts* mas não gera inconsistência. Uma característica pertinente da $D^2STM$ é que a mesma herda algumas peculiaridades da $JVSTM$, como a capacidade de não abortar transações somente leitura, *weak-atomicity* e *opacity*.

## 3.3 SPECULA

$SPECULA$, (PELUSO et al., 2012), utiliza o conceito de predição para execução de transações. A predição se baseia em

[6] Deferred update: a verificação dos conflitos apenas é verificada na fase de *commit*.

uma possível entrega ordenada de mensagens determinada pela rede. O *SPECULA* tolera faltas de parada e necessita que o protocolo atômico de entrega de mensagens chegue ao seu fim para que transações sejam definitivamente confirmadas. A arquitetura de uma réplica *SPECULA* pode ser vista na Figura 4.

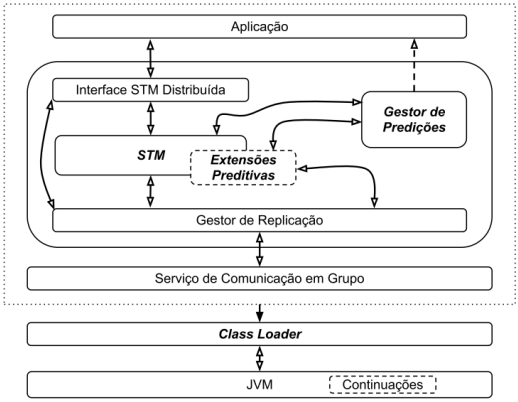


Figura 4: Arquitetura de uma réplica SPECULA (PELUSO et al., 2012).

A réplica *SPECULA* ao receber uma transação a executa imediatamente. Logo após executar a transação, a certificação local se inicia, com isso as alterações se tornam visíveis para outras transações preditivas[7] sob a hipótese de que a validação final levará a confirmação das transações preditivas. Com isso, é possível encadear transações preditivas, porém, caso a verificação global não tenha sucesso, todas as transações preditivas devem ser canceladas e o estado deve ser reposto para antes da predição.

No *SPECULA* apenas uma réplica executa as transações, e faz difusão atômica da sequência executada. Enquanto isso

[7]Transações preditivas são previsões das ordens de confirmação das transações. Então as transações são postas em diferentes ordens de execução aumentando a previsibilidade de uma dessas possíveis ordens se confirmar.

acontece, outras réplicas também executam as transações, e solicitam validações à réplica primária onde a transação é executada. As outras réplicas seguem executando transações na ordem em que chegam, e solicitam a certificação global à réplica principal. As réplicas secundárias ao receberem as respostas da réplica principal sobre a confirmação global, verificam se as predições estão corretas ou precisarão seguir a ordem atômica estabelecida pelo *Atomic Broadcast*. Caso a predição não seja a mesma definida pelo protocolo atômico, as transações precisam ser abortadas.

Os trabalhos $D^2STM$ e $SCert$ (Seção 3.7) ao iniciar a fase de certificação bloqueiam a *thread* de execução até o término da certificação local e global. Esse bloqueio não acontece no $SPECULA$ e logo após a certificação local a *thread* é desbloqueada[8]. O $SPECULA$ só tenta sincronizar transações quando entra na fase de commit, ou seja, não realiza análise de transações para verificar conflitos.

## 3.4 AGGRO

*AGGRO*, (PALMIERI; QUAGLIA; ROMANO, 2010), conta com a entrega otimista de mensagens para execução de transações, semelhante ao $SPECULA$. O otimismo encontrado no *AGGRO* supõe que a ordem de execução local de transações será igual à ordem final de execução. A partir dessa suposição, a ideia chave por trás do *AGGRO* é buscar um limiar entre a execução de transações e a coordenação entre as réplicas utilizando protocolos de ordem total juntamente com protocolo otimista. *AGGRO* tolera faltas de parada. A arquitetura de uma réplica *AGGRO* pode ser vista na Figura 5.

[8]Logo após o desbloqueio da *thread*, novas transações podem ser iniciadas.

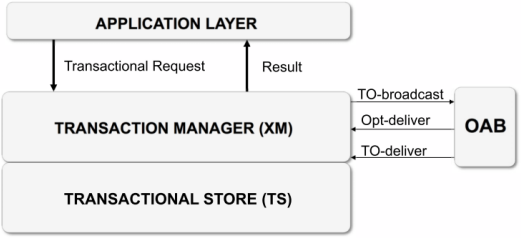


Figura 5: Arquitetura de uma réplica AGGRO (PALMIERI; QUAGLIA; ROMANO, 2010).

Toda transação é repassada da camada de aplicação para o gerenciador de transações (XM da Figura 5). O gerenciador de transações utiliza dois *buffers* de transações: OPT-Delivered e TO-Delivered, entrega otimista e total ordem respectivamente, As transações ao serem entregues pelo protocolo otimista logo são executadas. Os dados alterados pelas transações no *AGGRO* podem estar em um de dois estados: *Work-In-Progress*[9] (WiP) ou *Complete*[10]. Dados são definidos como *Work-In-Progress* quando atualizados pela primeira vez e que já não estejam nesse estado. Quando um dado de escrita é marcado como *WiP* uma verificação é feita com as outras transações no sistema. A verificação se dá para eliminar leituras sujas, sendo que, transações de leitura que precedem transações de escrita no *buffer* de transações otimistas são abortadas. Ao término de cada transação, uma chamada a função *Complete()* é gerada, a transação é desmarcada de *WiP* e definida como *Complete*.

Na fase de confirmação, a *thread* que executou a transação fica bloqueada até seu término (*abort* ou *commit*). Ao entrar na

[9]Transações no estado *Work-In-Progress* significam que estão sendo executadas.

[10]Transações no estado *Complete* significam que foram totalmente executadas, mas ainda não confirmada.

fase de confirmação o estado da transação se altera para *Complete* e com isso suas alterações são propagadas para outras transações. Com isso é possível encadear a execução de transações de forma semelhante aos modelos de atualização direta (*direct-update*). Porém, a transação só termina quando o protocolo de ordem total chega ao seu fim, sendo que a mesma ainda pode ser abortada.

Importante ressaltar que o *AGGRO* utiliza controle de concorrência baseado em *locks*, o que permite apenas uma linha de execução por transação. Na confirmação da suposição[11] acima, não haverá necessidade de re-execução, caso contrário, as réplicas em que a ordem não é a mesma terão de re-executar as transações na ordem imposta pelo protocolo atômico.

## 3.5 OSARE

OSARE, (PALMIERI; QUAGLIA; ROMANO, 2011), executa de forma especulativa um sub-conjunto das possíveis ordens plausíveis seriais das transações. Assemelha-se ao modelo proposto por *AGGRO*, porém ao invés de utilizar apenas uma linha de execução o *OSARE* utiliza algumas possíveis ordens de execução. Da mesma forma que no *AGGRO*, o *OSARE* conta com dois *buffers*, um para o recebimento especulativo de transações e outro que contém o recebimento atômico. As possíveis ordens de execução são definidas a partir do *buffer* especulativo. Após o término do protocolo atômico as ordens são comparadas, e caso a ordem especulativa combine com a ordem final definida pelo protocolo atômico, essa ordem é confirmada. *OSARE* tolera faltas de parada e necessita do término do protocolo atômico para confirmar transações.

[11] Ordem final do protocolo atômico é igual a ordem otimista / inicial.

O cliente emite uma transação como no *AGGRO*. Toda escrita feita durante a execução da transação é armazenada em um *buffer* que identifica os objetos que foram alterados. Antes da leitura, o objeto é verificado se existe conflito de escrita-leitura, isso é feito usando um *buffer* como identificador. A cada novo conflito uma nova cópia do objeto é gerada, onde as alterações são feitas. As alterações feitas por uma transação ao seu término são propagadas, caso estejam marcadas como completas, semelhantemente a atualização direta. A verificação de conflitos é feita a cada leitura, e caso conflitos existam uma nova linha de execução é criada para operar com o dado conflitante. A divisão faz com que uma linha siga utilizando os objetos do *snapshot* confirmado e outra siga pela linha que utiliza os objetos alterados pela transação conflitante, simulando atualização direta. A cada nova linha de execução, uma função de reordenação é chamada para que o *buffer* de transações otimistas se assemelhe ao de ordem total. Após o término do protocolo atômico é possível fazer a validação e confirmação das transações. A confirmação e validação das transações seguem o modelo tradicional de verificação dos conjuntos de leituras e escritas.

## 3.6 STR

STR, (ROMANO et al., 2010), é baseado na exploração mínima das possíveis ordens distintas e serializáveis das transações recebidas de forma otimista. Sua abordagem segue na mesma linha dos trabalhos *SPECULA* e *OSARE* onde transações são executadas logo que são recebidas, diferenciando-se na forma como são geradas as possíveis ordens especulativas por não utilizar permutação.

A ideia chave é que ao final do protocolo atômico haverá uma ordem especulativa que combinará com a ordem definida pelo protocolo atômico. Sendo assim, somente há a necessidade de confirmar essa ordem, sem necessidade de re-executar. *STR* tolera faltas de parada e utiliza o conceito de *Speculative Polygraph* na fase de validação de uma transação.

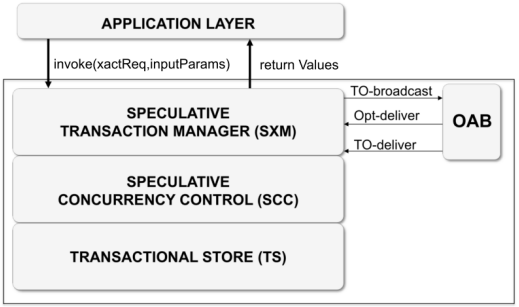


Figura 6: Arquitetura de uma réplica STR (ROMANO et al., 2010).

O cliente interage com a camada do gerenciador de transações especulativas (SXM), conforme a Figura 6, através da primitiva *invoke*. Então o cliente emite sua transação, semelhante aos modelos anteriores, onde existem dois *buffers*, um para a entrega otimista e outro para ordem total. Quando o protocolo de ordem total começa a entregar suas mensagens, com isso o *buffer* otimista é preenchido e as transações iniciam sua execução. A cada leitura efetuada pela transação, uma verificação de conflitos é disparada e caso existam conflitos, uma nova linha de execução é iniciada. Toda transação inicia em um *snapshot* correto, mas a cada nova linha de execução criada, um novo *snapshot* relativo é gerado. O *snapshot* relativo contém os dados conflitantes atualizados, semelhante aos modelos onde atualização direta é utilizado. Então assim a transação conflitante possui duas linhas

de execução, uma com o *snapshot* original e outra com o relativo. Com a divisão da linha de execução, o mínimo possível de ordens distintas é alcançado, pois a cada nova leitura conflitante é gerada outra linha de execução para tratar o conflito encontrado. Após a execução da transação é necessário confirmar a mesma, sendo a confirmação feita somente com o término do protocolo atômico. Com o fim do protocolo atômico, haverá uma ordem serial que combinará com a ordem otimista definida durante a execução das transações. Com isso as transações são validadas e confirmadas, consoante a ordem final imposta pelo protocolo atômico. Após a confirmação da transação a resposta é encaminhada ao cliente.

Diferente das abordagens que utilizam permutação para tentar definir qual será a ordem final através da entrega otimista, no *STR*, a busca pela ordem final é construída em tempo de execução, através da verificação de conflitos entre as transações. Isso faz com que sempre exista uma ordem especulativa que se encaixe com a ordem final.

## 3.7 SCERT

SCert, (CARVALHO; ROMANO; RODRIGUES, 2011), é um protocolo de replicação que permite certificar transações de forma otimista. Sua abordagem utiliza mecanismo semelhante às abordagens anteriores para execução otimista de transações, onde somente após o término do protocolo atômico é que as transações são finalmente confirmadas. Utiliza detecção de conflitos adiantada[12] e propaga atualizações de transações que passaram a fase de validação. Sua arquitetura está representada na Figura 7.

Semelhante as propostas de *STR*, *OSARE* e *AGGRO* o

[12]Conflito é detectado logo que ocorre (*early detection*).

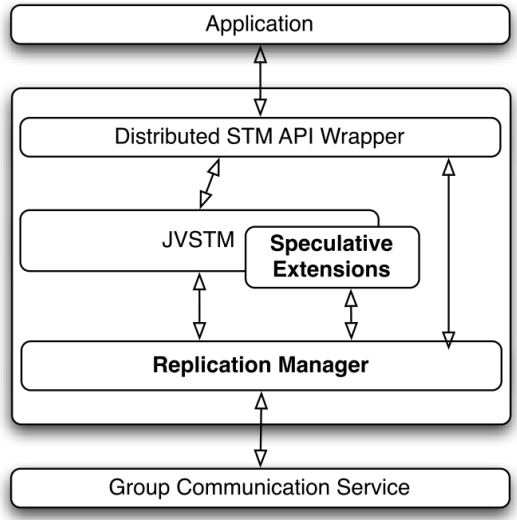


Figura 7: Arquitetura de uma réplica SCert (CARVALHO; ROMANO; RODRIGUES, 2011).

*SCert* inicia de forma especulativa, onde transações são iniciadas logo que recebidas. A execução de transações acontece sem a interação entre réplicas. Após a execução da transação, a mesma é validada localmente e caso a validação tenha sucesso os conjuntos de leitura e escrita são disseminados para as outras réplicas para que a validação global se efetive. Com a certificação global da transação, os dados alterados pela mesma são feitos visíveis para as outras transações. Com isso novas transações no sistema iniciam sua execução utilizando o último *snapshot* confirmado especulativamente. Após a execução da transação, confirmação local e global, a transação é marcada como confirmada especulativamente. Transações que precedem transações que estejam marcadas como confirmadas especulativamente e possuem conflitos de dados são abortadas imediatamente. Ao término do protocolo

atômico, as transações são reordenadas conforme a ordem final caso difiram. Por fim, as transações são confirmadas consoante a ordem final.

*SCert* utiliza as primitivas *begin*, *commit* e *abort* para iniciar, confirmar e abortar uma transação. No *SCert* transações podem ser bloqueadas temporariamente.

## 3.8 GRANOLA

*Granola*, (COWLING; LISKOV, 2012), é uma infra-estrutura para coordenação de transações que permite construir e executar aplicações de forma confiável e distribuída. O *Granola* provê consistência forte através da serialização e coordenação das transações que serão executadas. O modelo de transações encontrado no *Granola* segue a mesma linha de transações pré-declaradas, onde não existe interação com o cliente após o envio da transação; defini-se que o modelo realiza transações de uma rodada (*one-round*[13]). Transações podem ser de três tipos: *single-repository*, *coordinated*, e *independent transactions*. A contribuição primária do *Granola* está na execução de transações independentes.

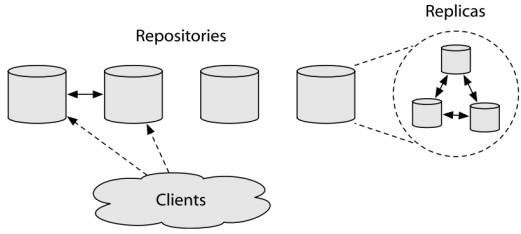


Figura 8: Granola Topologia (COWLING; LISKOV, 2012).

*Granola* possui clientes, repositórios e nodos, como pode

[13]Uma única rodada de comunicação entre o cliente e os repositórios.

ser visto na Figura 8 os repositórios comunicam-se uns com os outros para coordenar suas ações. Cada repositório é composto por nodos que provêm as garantias necessárias ao protocolo. A aplicação do cliente é composta por um *proxy* que direciona para quais repositórios determinada transação deve ser emitida. Cada transação é etiquetada com um identificador único. Após o recebimento da transação, cada repositório executa a transação de forma isolada, ou seja, não há comunicação entre repositórios nesse momento. Cada repositório pode estar executando em um de dois modos: *timestamp mode* e *locking mode*. No modo *timestamp* as transações são ordenadas em função da hora de chegada e, para confirmar a mesma, os repositórios trocam os *timestamps* antes de confirmar a transação para garantir que a ordem será a mesma em todos repositórios. No *locking mode* as transações só podem ser confirmadas se receberem confirmação positiva de todos outros participantes (semelhante à certificação total).

Logo que uma transação é confirmada, as transações que possuem dados dependentes são todas abortadas, diferente dos esquemas de certificação convencional onde somente com o término do protocolo atômico a verificação de conflitos é feita. A abordagem utilizada nessa dissertação não permite que um cliente emita mais de uma transação por vez; no *Granola* essa restrição não existe.

## 3.9 DISTM

*DiSTM* (KOTSELIDIS et al., 2008) é um sistema de Memória Transacional em Software Distribuída que executa transações de acordo com o número de *threads* que foram definidas em cada réplica. Segundo os autores é um mecanismo onde pode-se fa-

cilmente conectar diferentes métodos para controle de contenção remoto e local.

Clientes emitem transações, cada réplica processa localmente suas transações e quando a mesma tenta modificar um objeto, uma cópia local é feita e as modificações são feitas na cópia. A fase de validação é feita através dos métodos *validation()* e *commit()*. Com o primeiro método (*validation()*) os conjuntos de leitura e escrita são disseminados para todos os participantes, e caso não surjam conflitos, a transação é alterada de *active* para *commit*. Com isso existem duas possibilidades, a primeira é disseminar para os participantes os conjuntos de leitura e escrita para validação global logo que a fase de validação ocorreu. Outro método é primeiro adquirir a permissão para confirmar e só então disseminar os conjuntos de leitura e escrita. A permissão vem do nodo responsável por manter a ordem serial de execução das transações. Logo, o nodo responsável pode se tornar um gargalo do sistema.

*DiSTM* utiliza mecanismo centralizado para controle de contenção. Não tolera faltas de parada e não utiliza meios otimistas para o processamento de transações.

## 3.10 DMV

*Distributed MultiVersioning* (MANASSIEV; MIHAILESCU; AMZA, 2006), é um algoritmo de controle de concorrência distribuído em nível de página. O *DMV* permite diferentes nodos manipular estruturas de dados compartilhados em memória de uma maneira atômica e serial. O *DMV* executa as transações em cima do sistema de Memória Compartilhada Distribuída *DSM - (Distributed Shared Memory)*, através de modificações na camada de memória

compartilhada.

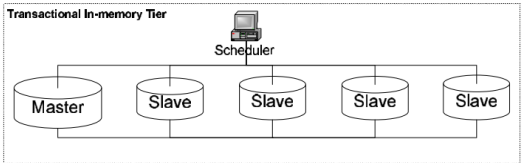


Figura 9: *DMV* visão geral (MANASSIEV; MIHAILESCU; AMZA, 2006).

As principais modificações na camada de memória compartilhada distribuída feitas pelo *DMV* são modificações que capturam as primitivas de leitura e escrita. Com isso é possível verificar conflitos inerentes a acessos concorrentes. Caso conflitos aconteçam e para que somente um objeto modificado seja confirmado em todos os nodos, é necessário obter um *token* único para que a escrita a esse objeto aconteça. O *token* é definido pelo nodo mestre, como pode ser visto na Figura 9. O *DMV* ao invés de utilizar *locks* distribuídos como controle de concorrência, usa o modelo transacional para serializar as escritas de forma global. Todas as escritas a dados são transformadas em transações.

O modelo utiliza um mecanismo de atualização centralizado chamado de escalonador para manter as atualizações consistentes, semelhante ao trabalho de Kotselidis (KOTSELIDIS et al., 2008). Porém, mecanismos centralizados incorrem em perda de desempenho quando muitos nodos fazem parte do sistema. É importante ressaltar que o *DMV* apenas mantém uma cópia do objeto por nodo, diferente do que aqui será proposto, onde mais de uma cópia pode haver, sendo que uma e somente uma será confirmada.

## 3.11 ANACONDA

Anaconda (KOTSELIDIS; LUJáN, 2010) tem como objetivo principal investigar a implementação de primitivas de sincronização em *clusters* baseado em Memória Transacional. Utiliza RMI - *Remote Method Invocation* para recuperação de objetos inter-nodo, utiliza o protocolo *lazy* para validação remota e *eager* para validação local, onde uma transação é abortada logo que detecta conflito, não havendo necessidade de esperar pela fase de *commit*. Implementa STM usando uma JVM - *Java Virtual Machine* por nodo, e cada nodo pode executar várias *threads*. A comunicação entre os nodos é feita através do *ProActive framework*.

Basicamente, a proposta altera as primitivas de sincronização do Java (*locks*) para que funcionem como Memória Transacional em Software, ou seja, não são mais utilizados *locks* para garantir acesso a dados concorrentes, ao invés disso são utilizadas transações como controle de concorrência.

## 3.12 CLUSTER-STM

$Cluster - STM$ (BOCCHINO; ADVE; CHAMBERLAIN, 2008) foca em como particionar um conjunto de dados em uma grande e escalável Memória Transacional em Software. $Cluster - STM$ não utiliza uma cópia do objeto por nodo (*1-copy-serializability*), o que faz é justamente o contrário, particiona os objetos de tal forma que cada nodo deve manter o objeto. O particionamento é feito através da atribuição a cada objeto de um nodo *home* ao qual é responsável por manter o dado atualizado. Semelhante a abordagem que será apresentada, $Cluster - STM$ garante *weak-atomicity* e difere na utilização de mecanismos otimistas no pro-

cessamento de transações.

## 3.13 SNAKE-DSTM

No $Snake - DSTM$, (SAAD; RAVINDRAN, 2011), é implementado um sistema distribuído utilizando o controle de fluxo do Java RMI para conceber a capacidade de funcionar como uma memória transacional em software. No Snake-DSTM cada objeto encontra-se no seu nó de origem e, quando outro nodo necessita acessá-lo, faz uma chamada remota ao dono do objeto. O nodo por sua vez pode fazer outra chamada remota, até encontrar o objeto. As chamadas são armazenadas em um grafo chamado grafo de chamada e é semelhante ao Cluster-STM onde apenas existe um processo em cada nodo por motivos de estabilidade.

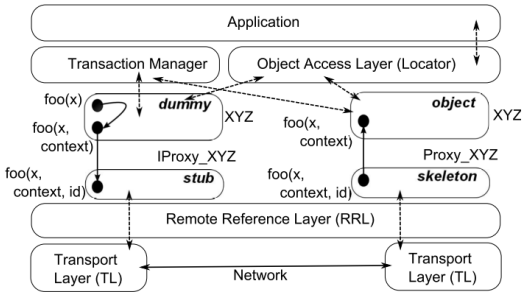


Figura 10: Arquitetura de uma réplica Snake DSTM (SAAD; RAVINDRAN, 2011).

O cliente emite uma transação ao nodo que irá inicialmente receber a transação, como mostrado na Figura 10. Logo após o recebimento o nodo inicia a execução da transação caso possua autoridade sobre o objeto. Caso não possua autoridade o dado é buscado remotamente. Ao final da execução da transação, uma mensagem de validação da mesma é encaminhada para todos par-

ticipantes. Junto com a mensagem de validação da transação é anexado o conjunto de escrita, com isso, é possível validar globalmente a transação. Se todos nodos retornam positivamente, a transação é confirmada. Após a confirmação é retornado ao cliente o resultado.

O protocolo de certificação consiste de votação, onde para uma transação ser confirmada, é necessário que todos os nodos retornem positivamente a solicitação de *commit*. O protocolo padrão[14] utilizado pelo $Snake - DSTM$ para confirmação de uma transação não suporta faltas de parada. $Snake - DSTM$ está disponível como parte do projeto *HyFlow*[15].

## 3.14 RAM-DUR

RAM-DUR (SCIASCIA; PEDONE, 2012), Figura 11, é um mecanismo de *cache* distribuído que permite um alto desempenho e forte consistência dos dados. No modelo tradicional de bases de dados distribuída, os nodos apenas mantém uma parte da base de dados em memória e caso um dado que não esteja em *cache* seja requerido, é necessário fazer acesso a disco para recuperá-lo. Na implementação de (SCIASCIA; PEDONE, 2012) a base de dados é inteiramente dividida entre os nodos que compõem o sistema, e caso um dado que não esteja em um nodo seja requisitado, ao invés de fazer acesso a disco, é feita a recuperação do dado pela rede. Cada nodo executa otimisticamente as distintas operações de cada transação emitida pelo cliente e na fase de confirmação, é enviado através de *broadcast* para os outros nodos no sistema o conjunto de escritas. Com isso é possível fazer a certificação

[14] D2PC - Protocolo de confirmação de duas fases
[15] http://www.hyflow.org/

da transação globalmente, somente necessário para transações de escrita. Apenas um nodo executa a transação, e caso ela venha a ser confirmada, apenas as atualizações são enviadas para os outros nodos. O modelo tolera faltas de parada.

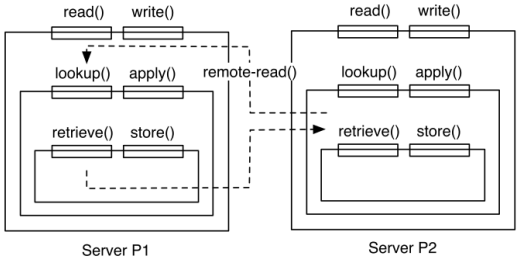


Figura 11: RAM-DUR abstração de nodos (SCIASCIA; PEDONE, 2012).

Os clientes têm acesso aos dados através de primitivas *read* e *write* que são implementadas em cima das operações *lookup* e *apply* respectivamente. A operação *lookup* é convertida para *retrieve*[16] ou *remote-read*[17]. O servidor ao receber uma operação de leitura faz busca local para verificar se o dado em questão está em memória. Caso o dado não esteja em memória a operação é convertida para *remote-read* e o dado é recuperado do servidor que possui o dado. Com isso o cliente pode fazer as alterações que desejar no dado recuperado. Após a execução da transação uma operação de *commit* é emitida para que a transação seja confirmada. O mecanismo utilizado para verificar a possibilidade de confirmação de uma transação segue o modelo *deferred-update*, que por sua vez, somente em tempo de *commit* é que conflitos são resolvidos. O protocolo $RAM - DUR$ utiliza certificação através da interseção do conjunto de escrita da transação que quer con-

[16] O dado está em memória e não necessita ser recuperado de outro servidor.
[17] O dado não está em memória e precisa ser recuperado remotamente.

firmar com o conjunto de leitura de todas outras transações no sistema. Se o resultado da interseção for vazio, então a transação pode ser confirmada, as alterações são aplicadas ,(através da primitiva *store*), à base de dados e a resposta é encaminhada ao cliente.

$RAM-DUR$ é focado em desempenho, recuperar um dado que está na memória principal pela rede é algumas ordens de magnitude mais rápido do que recuperá-los do disco. Os itens que são frequentemente recuperados são armazenados em *cache*, deste modo reduzindo tráfego de rede. O mecanismo utilizado no protocolo de terminação é baseado no Paxos (LAMPORT, 1998). O conteúdo e ordem das transações são seguramente armazenado pelo Paxos de forma durável, sendo que, não serão perdidos mesmo na presença de faltas.

## 3.15 ZHANG

Zhang (ZHANG; ZHAO, 2012), Figura 12, utiliza separação do acordo da execução. O modelo não permite transações aninhadas e todas as transações passam pelo *cluster* de acordo, no qual faz a ordenação das requisições e encaminha para o *cluster* que faz a execução da transação STM, não permitindo a execução em paralelo de transações não conflitantes. Necessita de *3f+1* réplicas para alcançar tolerância a faltas Bizantinas, devido a separação da execução do acordo, sendo que o *cluster* que executa as transações não tolera faltas Bizantinas. Foi implementado utilizando a biblioteca *LSA-STM - A Lazy Snapshot Algorithm with Eager Validation* de (RIEGEL; FELBER; FETZER, 2006), que por sua vez aborta transações somente leitura e utiliza *Compare And Swap* na fase de validação. O algoritmo de sincronização descrito

no trabalho de Zhang (ZHANG; ZHAO, 2012) utiliza a propriedade *lock-freedom* garantindo a ausência de *deadlock*, mas permitindo a ocorrência de inanição.

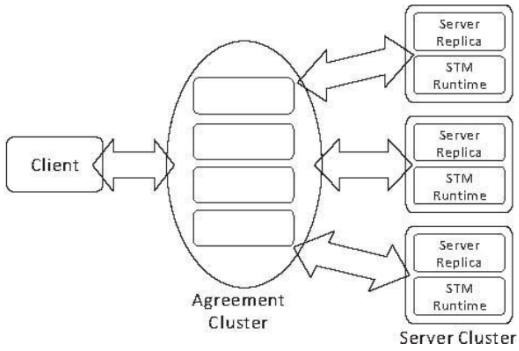


Figura 12: Zhang arquitetura (ZHANG; ZHAO, 2012).

O funcionamento consiste da seguinte forma: o cliente envia uma requisição pré-declarada ao *cluster* de acordo, que recebe, ordena e envia as requisições totalmente ordenadas para o *cluster* de execução STM, que por sua vez executa a transação e retorna ao *cluster* de acordo, no qual valida ou aborta ou reinicia a transação.

A abordagem considerada por (ZHANG; ZHAO, 2012), assemelha-se à aqui apresentada pela parte bizantina do protocolo mas se distingue pelos quesitos otimista onde não faz suposição alguma sobre. Aborto de transações somente leitura, execução paralela de transações, transações interativas e inanição.

## 3.16 CONSIDERAÇÕES GERAIS

As abordagens $Cluster - STM$, $D^2STM$, $RAM - DUR$, 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 proposta nessa dissertação 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 e sem réplicas maliciosas no sistema.

Nos trabalhos *AGGRO*, *STR*, *SPECULA*, *OSARE*, e *SCert* existe execução otimista das transações, mas diferente da proposta dessa dissertação, é necessário que a ordem final de entrega das mensagens seja definida para que as transações possam ser confirmadas. Mais especificamente, o otimismo utilizado em *STR* é baseado na completa exploração de todas as ordens distintas plausíveis recebidas pelo *Optimistic Atomic Broadcast* (OAB), e *OSARE* utiliza um subconjunto das ordens distintas.

O Mesobi utiliza abordagem semelhante aos trabalhos *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 $D^2STM$, *SCert*, *STR*, *AGGRO* e *OSARE* bloqueiam a *thread* que executou a transação até o final da fase de certificação. No *SCert* e nessa dissertação, novas transações podem ser bloqueadas temporariamente antes de serem executadas e liberada após a confirmação de outra transação ou pelo final do protocolo Bizantino.

A semântica de transação utilizada é *weak atomicity*, trabalhos que seguem nessa mesma linha são $Cluster - STM$ e $D^2STM$. O critério de consistência para replicação semelhante é encontrado nos trabalhos $D^2STM$, *DMV*, *AGGRO*, *STR*, *SPECULA*, *SCert* e *OSARE* onde *1-copy serializability* é utilizado. Em específico, os trabalhos que seguem *AGGRO* como base

(ver Tabela 2 coluna A-B) utilizam o conceito de ordens alternativas serializáveis, onde várias ordens seriais existem no sistema, e somente uma e a mesma é confirmada em cada réplica. Essa dissertação utiliza técnica semelhante como mecanismo otimista no processamento de transações. A técnica aqui utilizada permite que mais de um histórico de execução exista no sistema, mas históricos não confirmados não são visíveis à outras transações.

Os trabalhos *Anaconda*, $D^2STM$, $RAM-DUR$ utilizam verificação global de conflito tardia com filtro de *Bloom* (BLOOM, 1970). Somente após a detecção global de conflitos é que o histórico de execução pode ou não ser confirmado. Essa dissertação difere na fase de verificação de conflitos. A verificação de conflitos local é feita através do PTA - *Parallel Transactional Analyzer* e a verificação de conflitos global é feita utilizando protocolo de certificação total através do uso da técnica *Compare and Swap*.

Diferente de $DiSTM$ (KOTSELIDIS et al., 2008) que utiliza mecanismos de exclusão mútua e *tickets* na fase de confirmação (*commit*) da transação, a abordagem aqui utilizada consiste de um mecanismo de confirmação total semelhante ao encontrado em *Granola*. A utilização de exclusão mútua na fase de confirmação pode se tornar um gargalo para o sistema conforme o número de clientes aumenta. $DiSTM$ utiliza um *ticket* global que faz o papel de gerenciador de contenção para detectar transações conflitantes em diferentes nodos. Contenção aqui é verificado através do PTA e será explicado na Seção 4.4.4.

Semelhante à proposta apresentada, $STR$, $AGGRO$, $OSARE$, $RAM-DUR$ e $SCert$ não necessitam do conhecimento *a-priori* dos dados acessados por uma transação para sua execução (transações interativas, ver Seção 4.1). Utiliza-se para transações interativas,

mecanismo semelhante ao $RAM - DUR$ na fase de verificação de conflitos entre transações, onde os conjuntos de leitura e escrita são enviados para cada réplica. Diferente de $RAM - DUR$, ao invés de utilizar *atomic broadcast* como mecanismo de validação, um modelo de certificação total é empregado. Adicionalmente, a proposta trata da execução simultânea de transações pré-declaradas e interativas, sendo o cliente quem define qual utilizar. Mecanismo semelhante ao *Granola* para certificação entre transações é utilizado. No *Granola* e aqui é necessário que todos repositórios respondam positivamente a uma requisição de *commit*. Utiliza-se mecanismo semelhante quando transações interativas necessitam confirmar em paralelo: o desempate é feito priorizando a transação com menor identificador (*timestamp*).

A grande maioria dos trabalhos sobre Memória Transacional em Software na literatura, trata de faltas de parada *crash* e, até onde temos conhecimento, apenas o trabalho de Zhang (ZHANG; ZHAO, 2012) lida com faltas Bizantinas. No entanto sua estrutura é separada, onde o acordo é feito por um *cluster* utilizando *3f+1* réplicas tolerante a faltas Bizantinas e outro *cluster* utilizando *2f+1* réplicas para tratar da execução das transações na Memória Transacional em Software. No trabalho de Zhang, transações somente leitura podem ser abortadas, não utiliza a alta probabilidade em que as mensagens são entregues pela rede utilizando IP-Multicast e o conceito de transações interativas não é considerado.

Na proposta aqui apresentada, somente transações declaradas que possuem conflitos com outras transações declaradas ou interativas são bloqueadas, sendo que, transações interati-

[18] Precisa esperar pela ordem final de entrega de mensagens para terminar.
[19] Existem 2f+1 réplicas dentro de cada repositório, onde $c$ é o número de repositórios.

| | A-B | Crash | Byzantine | Réplica. | A-L | Modelo | Validação |
|---|---|---|---|---|---|---|---|
| DMV | Não | Não | Não | >= 1 | Sim | n/a | Centr. |
| Cluster-STM | Sim | Não | Não | >= 1 | n/a | n/a | n/a |
| DiSTM | n/a | Não | Não | >= 1 | Sim | n/a | Centr. |
| Anaconda | n/a | Não | Não | n/a | n/a | n/a | F. Bloom |
| Snake-DSTM | n/a | Não | Não | >= 1 | n/a | Pré-declarado | Eleição |
| D2STM | Sim | Sim | Não | >1 | Não | Pré-declarado | F. Bloom |
| RAM-DUR | Sim | Sim | Não | >1 | Não | Interativo | F. Bloom |
| AGGRO | Sim[18] | Sim | Não | >1 | Não | Interativo | CAS |
| STR | Idem AG-GRO | Sim | Não | >1 | Não | Interativo | CAS |
| SPECULA | Idem AG-GRO | Sim | Não | >1 | Sim | Interativo | CAS |
| SCert | Idem AG-GRO | Sim | Não | >1 | Sim | Interativo | CAS |
| OSARE | Idem AG-GRO | Sim | Não | >1 | Não | Interativo | CAS |
| Granola | Não | Sim | Não | $(2f+1)*c_{19}$ | Não | Pré-declarado | Eleição |
| Zhang | Sim | Sim | Sim | 3f+1; 2f+1 | Sim | Pré-declarado | CAS |

Tabela 2: Trabalhos relacionados. (A-B *Atomic Broadcast*, A-L Aborta Leitura)

vas são temporariamente bloqueadas caso possuam conflito com transações declaradas em execução. A proposta se beneficia da alta probabilidade das mensagens serem entregues em ordem pela rede usando IP-Multicast (KEMME et al., 2003; PEDONE; SCHIPER, 1998); somente em caso de mensagens fora de ordem ou réplicas maliciosas é que o protocolo Bizantino inicia, não necessitando que a ordem final seja definida para a execução das transações no caso otimista. O protocolo desenvolvido nessa dissertação e apresentado mais adiante foi construído utilizando a biblioteca JVSTM e preserva suas propriedades: *weak-atomicity* e *opacity*. A *JVSTM* é uma biblioteca que suporta controle de concorrência com múltiplas versões *(MVCC - MultiVersioning Concurrency*

*Control)*, foi criada por (CACHOPO; RITO-SILVA, 2006) e oferece um excelente desempenho em operações somente leitura. Porém a *JVSTM* não é distribuída, sendo que, a coordenação e gerência das transações será feita pelo protocolo que será apresentado.

Algumas características pertinentes a esta dissertação:

- O critério de desempate utilizado é temporal, ou seja, *timestamp*;
- As características implícitas encontradas na JVSTM serão preservadas, a saber, *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 garantia *opacity* é alcançada através do uso de *snapshots*;
- A técnica de atualização tardia (*deferred-update*) juntamente com verificação local antecipada de conflitos será empregada; A verificação de conflitos antecipada será tratada pelo Analizador de Transações Paralelas - PTA. A verificação global (*deferred-update*) utilizará mecanismo de certificação total, onde todos os participantes devem concordar para que uma transação seja confirmada de forma otimista; caso contrário o protocolo Bizantino precisa ser iniciado. A verificacão tardia (*deferred-update*) aqui empregada é semelhante ao encontrado no trabalho de (LUIZ; LUNG; CORREIA, 2011), onde os conjuntos de leitura e escrita são encaminhados na fase de certificação.
- Na proposta aqui apresentada, somente transações declaradas que possuem conflitos com outras transações declaradas ou interativas são bloqueadas, sendo que, transações

interativas são temporariamente bloqueadas caso possuam conflito com transações declaradas em execução.

# 4 PROPOSTA

Este capítulo descreve as definições básicas do sistema, premissas consideradas para o protocolo e sua arquitetura. O protocolo inicialmente conta apenas com a execução de transações pré-declaradas e é apresentado na Seção 4.3. O protocolo completo permite a execução de transações pré-declaradas e interativas e pode ser visto na Seção 4.4. A arquitetura aqui apresentada será do modelo completo, a saber, Mesobi.

## 4.1 DEFINIÇÕES BÁSICAS DO SISTEMA E PREMISSAS

É considerado 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 conjunto finito de réplicas R = $\{r_1, r_2, ..., r_n\}$. É assumido no modelo um nível parcial de sincronia, onde o sistema se comporta de forma assíncrona em grande parte do tempo mas existem períodos de estabilidade (DWORK; LYNCH; STOCKMEYER, 1988). As réplicas se comunicam via passagem de mensagem e podem falhar de acordo com o modelo de faltas Bizantinas (LAMPORT; FISCHER, 1982) 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 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 suporta transações pré-declaradas e interativas. Em transações pré-declaradas, o cliente necessita enviar todo código transacional para as réplicas[1]. 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 prevalecerá a confirmação das transações interativas; as transações pré-declaradas serão abortadas e terão a execução reagendada. A prevalência das transações interativas sobre as pré-declaradas deve-se pela facilidade de re-execução das transações pré-declaradas, onde não existe a necessidade de interação com o cliente, sendo assim, mais "barato" re-executá-las.

## 4.2 ARQUITETURA

A Figura 13 provê uma visão em alto nível da arquitetura composta em cada réplica Mesobi. O componente comunicação confiável do 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*. Os Gerentes TD e TI fazem a verificação de conflitos inerentes às transações em que estão vinculados. Após serem analisadas, e caso não possuam conflitos as transações são encaminhadas para os executores de transações (JVSTM). Mais detalhes sobre conflitos podem ser vistos na Seção 4.4. Os exe-

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

cutores JVSTM são compostos por *threads* e cada transação é vinculada a uma *thread*. A certificação otimista é acoplada a JVSTM e faz a captura dos métodos de *commit* utilizado na verificação global de conflitos ou certificação total. Em caso de não sucesso na verificação global o controle é passado para o protocolo Bizantino. O protocolo Bizantino utilizado nesse trabalho é o PBFT de (CASTRO; LISKOV, 1998) mas pode-se utilizar uma implementação mais robusta, como o BFT-Smart de (BESSANI; SOUSA; ALCHIERI, 2013). A biblioteca JVSTM permite bom desempenho em operações somente leitura através da utilização de *Multi Versioning Concurrence Control* e não é distribuída.

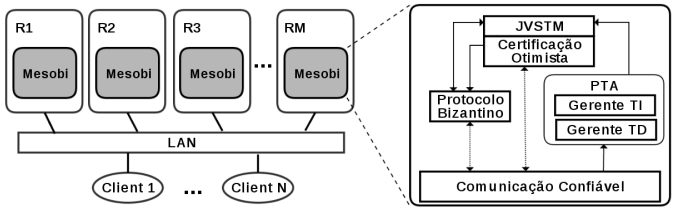


Figura 13: Arquitetura de uma réplica Mesobi.

## 4.3 PROTOCOLO OB-STM

Para melhor compreensão, será explicado o fluxo de operação (Figura 14) junto com o algoritmo OB-STM (Seção 4.3.1).

Inicialmente o cliente envia uma transação às réplicas (etapa T.Begin Figura 14). O protocolo OB-STM suporta o recebimento de transações declaradas que são definidas como $T_i$. Cada cliente pode enviar apenas uma transação declarada por vez. Será tratada primeiramente a execução otimista do protocolo OB-STM e por conseguinte a execução não otimista. Com o recebimento

da transação pelas réplicas (linha 1 do Algoritmo 1) a transação é anexada ao final da fila de transações. O conjunto de leitura e escrita da transação é recuperado pela função *GetWxRx*. Logo após a recuperação do conjunto de leitura e escrita a função PTA faz verificação de possibilidade de execução (etapa PTA Figura 14), formalmente definida na linha 13 do Algoritmo 1. Caso conflitos não surjam ou sejam detectados a transação pode ser executada (etapa EXEC Figura 14, linha 17 do Algoritmo 1). Após a execução da transação é necessário confirmar a transação, isso é feito através do protocolo de certificação total (etapa *AskForCommit* linha 46 do Algoritmo 1). Na etapa *AskForCommit* uma mensagem de confirmação da transação é enviada para todas as réplicas. Junto com a mensagem os identificadores (timestamp, conjuntos de leitura e escrita, ordem.) da transação são enviados. Cada réplica ao receber uma mensagem *AskForCommit* verifica utilizando as informações recebidas se a mesma pode ser confirmada. A resposta de confirmação só é positiva se todas informações contidas na transação solicitante condizerem com as informações da transação na réplica alvo (réplica alvo entende-se a réplica para onde a solicitação de confirmação foi enviada.). Caso a fase de confirmação tenha alcançado sucesso a transação é confirmada, com isso, os conjuntos de leitura, escrita e a transação em si são removidos dos *buffers* (linhas 49, 50 e 51 do Algoritmo 1). Ao final do processo com sucesso a resposta é enviada ao cliente (etapa Reply Figura 14, linha 52 do Algoritmo 1). Com o envio para o cliente a próxima transação em espera é colocada em execução (linha 53 do Algoritmo 1).

Na execução não otimista, todos passos são idênticos a execução otimista até a fase de validação da transação, que é a fase *AskForCommit*. Com o recebimento de uma negação da

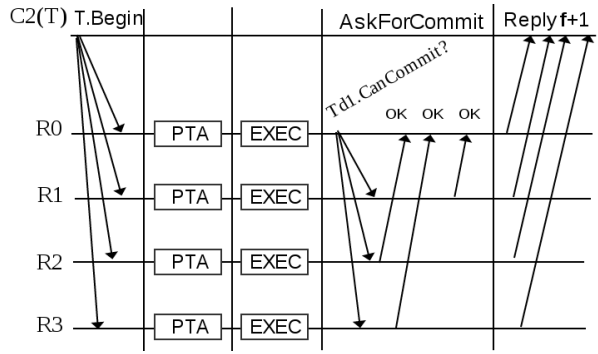


Figura 14: Transações declaradas, execução otimista.

solicitação de confirmação de uma transação (etapa *AskForCommit* Figura 15), o protocolo Bizantino precisa ser iniciado (etapa *Pre-Prepare* Figura 15, linha 39 do Algoritmo 1). O protocolo Bizantino utilizado no OB-STM e Mesobi é o mesmo, a saber, PBFT. Na mensagem de inicialização do protocolo Bizantino é anexada a mensagem de negação da transação e os históricos parciais de transações confirmadas e não confirmadas. Com isso é possível saber a veracidade do pedido e sincronizar os históricos inter-réplicas. O método utilizado para sincronizar os históricos pode ser visto na Seção 5.3.2. Após a geração da ordem pelo protocolo Bizantino (linha 23 do Algoritmo 2) as transações são re-executadas caso necessário e confirmadas (etapa RE-EXEC Figura 15, linha 26 do Algoritmo 2). A re-execução somente acontece se a ordem imposta pelo protocolo Bizantino diferir da ordem de execução na réplica. Logo que uma transação é confirmada a próxima na lista de execução é iniciada caso satisfaça as premissas definidas na Seção 4.4.2 (linha 53 do Algoritmo 1).

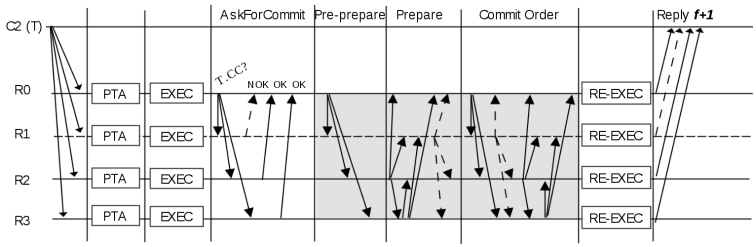


Figura 15: Transações declaradas, execução não otimista.

### 4.3.1 Algoritmo OB-STM

A Tabela 3 detalha as variáveis utilizadas no algoritmo OB-STM.

Tabela 3: Variáveis OBSTM.

| | |
|---|---|
| 1: $\prod^{R}$ | ▷ Replica Set |
| 2: $\prod^{Reply}$ | ▷ Reply Set |
| 3: $\prod^{T}$ | ▷ Transaction Set, Pre-Declared |
| 4: $\prod^{C}$ | ▷ UUID-Client Set: unique identifier |
| 5: $\prod^{WS}$ | ▷ Write Set of Transactions |
| 6: $\prod^{RS}$ | ▷ Read Set of Transactions |
| 7: BCO | ▷ Buffer Commit Order |
| 8: f | ▷ Number tolerated faults |
| 9: timeout | ▷ Timeout of wait replica response |

## 4.4 PROTOCOLO MESOBI

Semelhante ao feito na Seção anterior para o protocolo com transações declaradas, será explicado o fluxo de operação (Figura 16) junto com o algoritmo Mesobi (Seção 4.4.1).

A Figura 16 detalha a interação cliente-réplicas e réplica-réplica. No Mesobi 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 $TX = \{TX_i.B, TX_i.Op_1, ..., TX_i.Op_n, (TX_i.C \vee TX_i.A)\}$ para as interativas, onde $B \equiv$ Begin, $Op \equiv$ Operação, $C \equiv$ Commit e $A \equiv$ Abort.

A execução otimista e não otimista das transações pré-declaradas, na ausência de transações interativas, segue fluxo idêntico ao protocolo OB-STM (ver Seção 4.3) (RIBEIRO; LUNG; NETTO, 2013). O Mesobi é uma evolução do protocolo OB-STM onde transações interativas e pré-declaradas concorrem pela execução, sendo que as transações interativas têm prioridade sobre as pré-declaradas. Como o funcionamento do Mesobi na presença de apenas transações pré-declaradas é semelhante ao do OB-STM, este funcionamento não será novamente explicado (Seção 4.3). O Mesobi utiliza o conceito *WiP - WorkInProgress*, semelhante ao utilizado por (PALMIERI; QUAGLIA; ROMANO, 2010), diferindo no quesito propagação de atualização de objetos (ver Capítulo 3). Uma transação está em andamento (WiP) se a mesma já foi iniciada e ainda não foi concluída (entende-se por concluída a execução completa da transação e sua confirmação com sucesso). Uma transação entra no estado WiP quando inicia sua execução, a ativação do modo WiP é definido pelo analisador de transações paralelas (PTA). A cada nova operação recebida, seus objetos são acrescidos à lista de transações em progresso se a operação foi posta em execução. Os impasses ocorridos devido à entrada no modo WiP pelas transações são resolvidos pelos protocolos de confirmação otimista e Bizantino em caso de insucesso. A área sombreada na Figura 16b representa o PBFT - *Pratical Byzantine Fault Tolerance* (CASTRO; LISKOV, 1998), utilizado

na fase de resolução de conflitos (mensagens fora de ordem ou réplicas maliciosas).

No caso de transações interativas, o cliente necessita efetuar alguns passos a mais. Inicialmente o cliente requisita às réplicas o início da transação (etapa TX.Begin, Figura 16). Cada réplica ao receber a solicitação de início concatena a transação ao seu referido *buffer* e retorna ao cliente o seu *timestamp* ($ts$) (linha 1 do Algoritmo 4). A transação é definida como em execução (WiP), mas como ainda não existem operações para essa transação, não influenciará outras transações pois seu conjunto de leitura e escrita estão vazios. 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. O $ts$ utilizado segue o modelo proposto por (BAZZI; DING, 2004), para evitar o crescimento arbitrário do espaço de endereçamento por réplicas maliciosas. Após a definição do $ts$ pelo cliente (etapa TX.Begin, Figura 16), a primeira operação ($TX_i.Op$) pode ser enviada às réplicas. O recebimento da operação na réplica acontece na etapa TX.Op, Figura 16, linha 6 do Algoritmo 4. Com o recebimento da operação, o conjunto de leitura e o conjunto de escrita são postos nos seus referidos *buffers* (linhas 7, 8 e 9 do Algoritmo 4). A operação somente é executada se não houver conflitos com outras transações em execução ou esperando para ser executada (linha 12 do Algoritmo 4). A cada operação executada pelas réplicas um novo $ts$ é retornado ao cliente para determinar a ordem das operações; o gerenciador de contenção utiliza o $ts$ da primeira operação como critério de desempate.

Após o cliente terminar o envio de operações, é necessário enviar o comando de término da transação ($TX_i.C$) (etapa TX.Commit

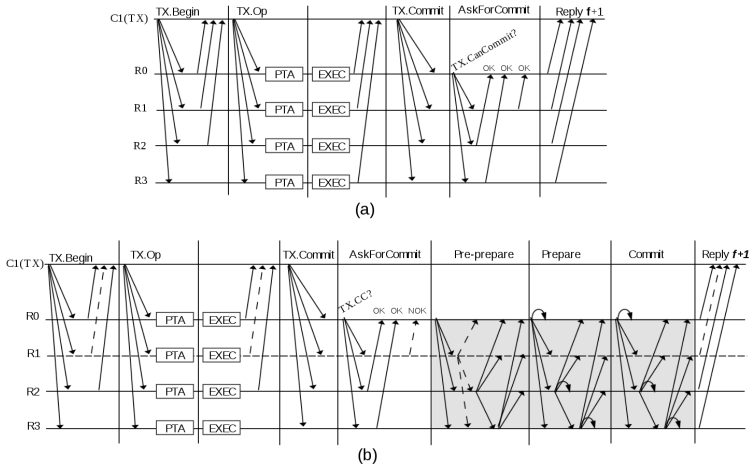


Figura 16: Transações interativas: fluxo de operação otimista (a) e não otimista (b).

Figura 16). Ao receber do cliente a solicitação de confirmação (linha 20 do Algoritmo 4) cada réplica envia para o conjunto de réplicas no sistema ($\prod^{R}$) uma mensagem de solicitação de permissão de confirmação (A.F.C.) da transação em questão (etapa AskForCommit, Figura 16, linha 21 do Algoritmo 4). Cada réplica necessita receber a confirmação de todas as outras réplicas (certificação total), a saber $3f$ mensagens OK! (linha 39 do Algoritmo 5) para confirmar a transação de forma otimista. Caso receba uma não-permissão (mensagem NOK!, etapa AskForCommit, Figura 16b) devido a uma réplica maliciosa ou transações fora de ordem, o protocolo Bizantino precisa ser iniciado (etapa *Pre-prepare*, Figura 16b, linha 42 do Algoritmo 5). 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 $\prod^{TX}$. Após a definição da ordem as transações são re-executadas caso necessário e a res-

posta é encaminhada ao cliente. Os critérios definidos para qual histórico parcial deve ser confirmado, quais transações devem ser ou não abortadas/confirmadas, serão explicados nas Seções 4.4.2 e 5.3.

### 4.4.1 Algoritmo Mesobi

Essa seção contém o algoritmo do protocolo Mesobi que é executado em cada réplica. Os algoritmos foram divididos para melhor enquadramento no texto e melhor compreensão. A Tabela 4 detalha as variáveis utilizadas. No Algoritmo 3 é feita análise de conflitos entre transações interativas e pré-declaradas e execução das transações pré-declaradas. O Algoritmo 4 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 5. Por fim, o Algoritmo 6 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. O algoritmo Mesobi é citado na Seção 4.4 onde é explicado em paralelo com a Figura 16.

### 4.4.2 Histórico, premissas e *snapshot*

O histórico de execução, premissas e captura do *snapshot* serão explicados nessa seção. Para as abordagens otimista e pessimista, as definições abaixo se mantêm.

O princípio da teoria de seriabilidade (BERNSTEIN; GOODMAN, 1985) garante que a execução do histórico $\mathcal{H}$ de transações em todos os processos replicados é equivalente à execução de

Tabela 4: Variáveis Mesobi.

| | |
|---|---|
| 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 |

$\mathcal{H}$ 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 devem 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) $\Sigma_T = \cup_{i=1}^{n} \Sigma_i$. (2) $\prec_\mathcal{H} \supseteq \cup_{i=1}^{n} \prec T_i$. (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.

No Mesobi a execução das operações de uma transação são

respeitadas atavés das premissas de execução abaixo. Transações não conflitantes podem ter seus históricos de execução diferenciados no sistema (visão global), mas o resultado final será o mesmo (transações sem conflito a ordem de execução não é importante). Para melhor elucidar, é suposto que duas transações não conflitantes estejam executando em ordem distinta nas réplicas. Mesmo quando há diferença no histórico global de execução das réplicas: $R_1 = \mathcal{H}^c_T = \{T_1, T_2\}$ e $R_{2,3,4} = \mathcal{H}^c_T = \{T_2, T_1\}$ o resultado é o mesmo. O resultado final independe da ordem, pois não há conflitos. Transações conflitantes devem ter sua ordem de operações preservadas no sistema global.

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;

- ($ii$) 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 conflitarem com transações pré-declaradas em execução ou que estejam marcadas como *WiP*.

As verificações das premissas ($i$) e ($ii$) encontram-se no Algoritmo 3, na função PTA (linha 1). A premissa ($iii$) é verificada no Algoritmo 4, 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 17, onde o símbolo ↓ deve ser interpretado como a operação $z = z + 1$.

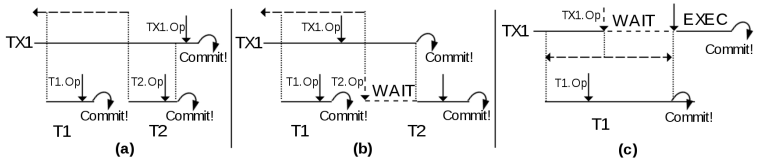


Figura 17: Transações interativas e pré-declaradas: (a) sem conflito. (b) com conflito ($TX_1.Op \prec T_2.B$). (c) com conflito ($T_1.Op \prec TX_1.Op$).

O caso otimista é representado na Figura 17 e detalha as relações de conflitos entre transações interativas e declaradas. Na Figura 17 quando visualizado *commit* deve ser entendido que a transação passou por todos os passos da certificação global. No detalhe (a), não existem conflitos entre as transações, pois não existem operações entrelaçadas (visão local). Formalmente, $\mathcal{H}_T^c = \{T_1.Op, T_1.C, T_2.Op, T_2.C, TX_1.Op, TX_1.C\}$[2]. 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 $T_2.Op$, sendo assim, não será permitido que a transação $T_2$ execute de

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

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 $TX_1.Op$, sendo assim, 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 momento em que a operação $TX_1.Op$ foi recebida pela réplica havia conflito (seta pontilhada), e com isso o *snapshot* não pôde ser capturado. Somente após o (*commit*|*abort*) da transação $T_1$ é que o *snapshot* realmente será capturado e a operação da transação $TX_1.Op$ será executada. Após a execução das operações emitidas pelo cliente, o mesmo envia o comando para término da transação e segue lógica da Seção 4.4 para confirmação de uma transação, a lembrar, certificação total através de *AskForCommit*.

O caso não otimista é retratado conforme as Figuras 18, 19, e 20; suponha que a ordem de chegada das operações nas réplicas seja: $R_{0,1} = \{T_1.B, T_2.B, TX_1.Op\}$, (note a precedência de chegada das transações declaradas) e $R_{2,3} = \{T_2.B, T_1.B, TX_1.Op\}$[3]. 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 à premissa (i). Os históricos parciais de operações não confirmadas das operações nas 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

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

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. A ordem imposta nesse exemplo, levando em conta o líder como sendo $R_{0,1}$ é $T_1, T_2$ e logo em seguida a operação da transação $TX$, resumindo: $\{T_1, T_2, TX.Op\}$. Como o comando de término (TX.Commit) ainda não havia sido recebido e a transação interativa estava sem modificações as transações pré-declaradas tiveram sua execução completa e logo em seguida a operação da transação interativa pôde ser executada. Com o término as respostas são encaminhas aos clientes e novas transações podem ser enviadas ao sistema.

A Figura 18 representa as transações pré-declaradas e interativas, com conflito de dados antes do protocolo Bizantino para as réplicas $R_{0,1}$ e a Figura 19 para as réplicas $R_{2,3}$.

A Figura 20 representa as transações pré-declaradas e interativas, com conflito de dados após o protocolo Bizantino para as réplicas $R_{0,1,2,3}$. O *batch* gerado pelo protocolo Bizantino será definido em função das transações declaradas, pois nesse caso, somente é conhecido a intenção de confirmar das transações declaradas. Com isso todas transações declaradas que estão esperando para executar e confirmar serão executadas caso necessário e confirmadas. Após a confirmação das transações declaradas pelo protocolo Bizantino, a operação da transação interativa po-

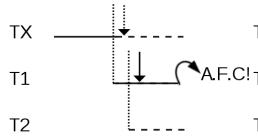


Figura 18: $R_{0,1}$

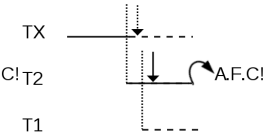


Figura 19: $R_{2,3}$

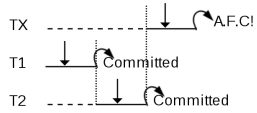


Figura 20: $R_{0,1,2,3}$

derá ser executada.

A Figura 21 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. Como anteriormente comentado, transações interativas são priorizadas em relação as pré-declaradas. Talvez não perceba-se na figura, mas a transação $T_1$ será confirmada devido ao seu início anterior às demais. Logo após a confirmação da transação $T_1$ as

transações $TX_1$ e $TX_2$ capturam o *snapshot* para execução de sua operação. Por fim, apenas uma transação será confirmada, a que emitir a ordem de confirmação primeiro (*commit*).

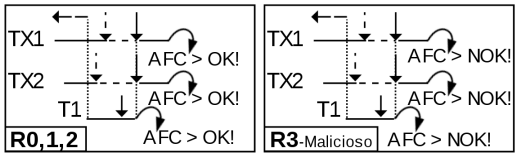


Figura 21: Transações pré-declaradas e interativas, com réplica maliciosa.

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.

### 4.4.3 Ciclo de vida das transações

O ciclo de vida das transações é caracterizado pelo autômato representado na figura 22. As transações interativas são retratadas como $TI$ e transações pré-declaradas são retratadas como $TD$. Inicialmente as transações são recebidas pelas réplicas e podem ser de dois tipos, $TI$ ou $TD$. Após o recebimento das transações pelas réplicas a verificação de conflitos é iniciada (etapa $PTA$). A verificação de conflitos é feita utilizando os conjuntos de leitura e escrita e pode ser melhor compreendido no algoritmo 3 linhas 4 e 7. Importante frisar que a verificação de conflitos é distinta entre transações interativas e pré-declaradas. Para as transações interativas apenas é necessário fazer análise de con-

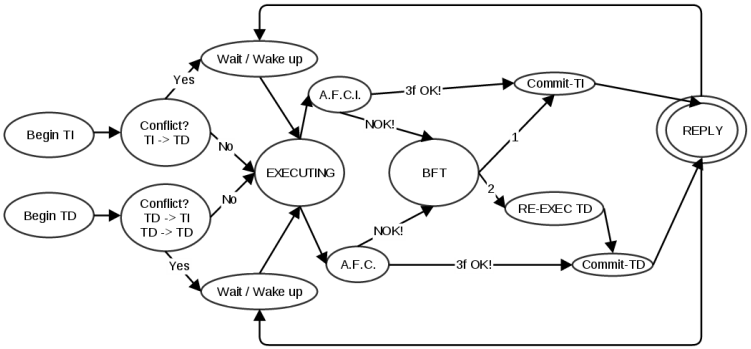


Figura 22: Ciclo de vida de transações interativas e pré-declaradas.

flitos com transações pré-declaradas. Com as transações pré-declaradas, a análise de conflitos é feita entre os dois tipo de transações, a saber, interativa e pré-declarada. Caso exista conflito com alguma transação existente nos *buffers*, a transação é colocada em espera (etapa *wait/wake up*). Na ausência de conflitos, a transação pode ser executada. Após a execução da transação, para que suas alterações sejam propagadas, é necessário fazer o *commit*. Ao tentar fazer o *commit*, uma mensagem *AskForCommit* ou *AskForCommitInteractive* é enviada (etapas *A.F.C.* ou *A.F.C.I.*). Cada réplica ao receber uma mensagem *A.F.C.I.* ou *A.F.C.* inicia etapa de certificação. Na certificação, uma transação é verificada sobre a possibilidade de ser confirmada, caso seja possível, uma mensagem (*OK!*) é retornada, caso contrário uma mensagem (*NOK!*) é retornada. Com o recebimento de $3f$ mensagens *OK!*, a transação pode ser confirmada (etapas *Commit-TI* ou *Commit-TD*). Ao receber uma mensagem *NOK!* o protocolo Bizantino precisa ser iniciado. O protocolo Bizantino define a ordem das transações existentes nos *buffers* de transações

interativas e pré-declaradas. Após a execução do protocolo Bizantino a confirmação das transações interativas devem ser efetivadas primeiro (etapa *Commit-TI*). Com o término da confirmação das transações interativas, as transações pré-declaradas podem ser re-executadas caso necessário e confirmadas (etapas *Re-Exec-TD* e *Commit-TD*). Por fim a resposta é enviada ao cliente (etapa *Reply*) e as transações que estão em espera são postas em execução caso satisfaçam as premissas do sistema.

### 4.4.4 Analisador de Transações Paralelas - PTA

Cada nova transação que é recebida por uma réplica deve ser verificada em relação as outras transações que estão em execução e nos *buffers* $\prod^{T}$ e $\prod^{TX}$. Caso não existam conflitos a transação recém recebida pode ser executada. Caso conflitos existam, a transação é colocada em espera.

Duas transações são ditas conflitantes se no seu conjunto de escrita existe ao menos uma variável em comum com o conjunto de escrita ou leitura de outra transação. A verificação de conflitos pode ser definida como segue: A transação $t_i$ com seu conjunto de escrita $\mathrm{WS}(t_i)$ e leitura $\mathrm{RS}(t_i)$, e a transação $t_j$ com seu conjunto de escrita $\mathrm{WS}(t_j)$ e leitura $\mathrm{RS}(t_j)$, são ditas conflitantes se qualquer uma das condições forem verdadeiras:

- (1) $\mathrm{WS}(t_i) \cap \mathrm{RS}(t_j) \neq \emptyset$.
- (2) $\mathrm{WS}(t_j) \cap \mathrm{RS}(t_i) \neq \emptyset$.
- (3) $\mathrm{WS}(t_i) \cap \mathrm{WS}(t_j) \neq \emptyset$.

Essa verificação pode ser vista mais formalmente nos algoritmos 1 e 3, em específico na função PTA linhas 8 e 1 respectivamente para OB-STM e Mesobi.

**Algoritmo 1** Replica OB-STM

```
1: upon: receive(⟨ REQUEST, Ti, Ci ⟩) from client
2:     ∏^T ← enqueue(Ti, Ci)
3:     Call GetWxRx(Ti, Ci)
4:     Call PTA(Ti, Ci, false)
5: function GetWxRx(Ti, Ci)
6:     ∏^WS ← ∏^WS ∪ (getWriteSet(Ti), Ci)
7:     ∏^RS ← ∏^RS ∪ (getReadSet(Ti), Ci)
8: function PTA(Ti, Ci, Analyzed)
9:     if (size(∏^T) >1 ) then
10:        if (Analyzed = false) then
11:            initExec = true
12:            for all Tj ∈ ∏^T i ≠ j do
13:                if (WS(Ti) ∩ RS(Tj) ≠ ∅) ∨ (WS(Tj) ∩ RS(Ti) ≠ ∅) ∨ (WS(Ti)
    ∩ WS(Tj) ≠ ∅) then
14:                    initExec = false
15:                    break for all
16:            if (initExec = true) then
17:                Call EXEC(Ti, Ci)
18:        else
19:            if (size(BCO) >0 ) then
20:                Call RE-EXEC(getFirstElement(BCO), Ci)
21:            else
22:                Call EXEC(getFirstElement(∏^T), Ci)
23:    else
24:        Call EXEC(Ti, Ci)
25:
26: upon: receive(⟨ASKFORCOMMIT⟩, Ti, Ri, Ci) from replica
27:    if (Ti can commit) then
28:        ReliableUnicast(⟨R-ASKFORCOMMIT⟩,Ti, Ri, Ci, OK!)
29:    else
30:        ReliableUnicast(⟨R-ASKFORCOMMIT⟩, Ti, Ri, Ci, NOK!)
31:    end if
32:
33: upon: receive(⟨R-ASKFORCOMMIT⟩, Ti, Ri, Ci, Reply) from replica
34:    acceptCommit ← WaitForAcceptance(timeout)
35:    if (acceptCommit = 3f) then
36:        Call CommitT (Ti, Ci)
37:    else
38:        //Begin Byzantine protocol.
39:        TO-Deliver(⟨PRE-PREPARE⟩, Ti, Ri, Ci, ORDER)
40:    end if
41: function Exec(Ti, Ci)
42:    set Ti as running
43:    begin execution
44:    if (Ti to try commit) then
45:        set Ti as committing
46:        ReliableMulticast(⟨ASKFORCOMMIT⟩, Ti, ∏^R, Ci)
47: function CommitT(Ti, Ci)
48:    Commit transaction Ti!
49:    ∏^T ← dequeue(Ti, Ci)
50:    ∏^WS ← ∏^WS \ (getWriteSet(Ti), Ci)
51:    ∏^RS ← ∏^RS \ (getReadSet(Ti), Ci)
52:    SendReliable(⟨REPLY⟩, Ri, Reply, Ci)          ▷ Send reply to client
53:    Call PTA(getFirstElement(∏^T), Ci, true)
```

**Algoritmo 2** Byzantine Replica OB-STM

```
1: upon: receive(⟨PRE-PREPARE⟩, Ti, Ri, Ci, ORDER) from replicas
2:     if (PRE-PREPARE was accept) then
3:         for all (R ∈ ∏^R minus his own replica) do
4:             TO-Deliver(⟨PREPARE⟩, T, Ri, Ci)
5:         end for
6:     end if
7: upon: receive(⟨PREPARE⟩, Ti, Ri, Ci, ORDER) from replicas
8:     if (PREPARE was accept) then
9:         for all (R ∈ ∏^R minus his own replica) do
10:            TO-Deliver(⟨COMMIT-ORDER⟩, Ti, Ri, Ci)
11:        end for
12:    end if
13:
14:
15: function RE-EXEC(Ti, Ci)
16:    set Ti as running
17:    begin execution
18:    if (Ti to try commit) then
19:        BCO ← dequeue(Ti,Ci)
20:        Call CommitT (Ti, Ci)
21: upon: receive(⟨COMMIT-ORDER⟩, Ti, Ri, Ci, ORDER) from replicas
22:    if (COMMIT-ORDER was defined) then
23:        BCO ← enqueue(ORDER)
24:        Tx ← getFirst(BCO)
25:        if ((Tx ≠ Ti) then
26:            Call RE-EXEC (Ti, Ci)
27:        else
28:            if (Ti is running) then
29:                wait Ti is committing
30:            end if
31:            Call CommitT (Ti, Ci)
32:        end if
33:    end if
```

**Algoritmo 3** Transações pré-declaradas - Mesobi.

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** $T_j \in \prod^{T} i \neq j$ **do**
4: **if** (¬ ( WS($T_i$) ∩ RS($T_j$) = ∅ ∧ WS($T_j$) ∩ RS($T_i$) = ∅ ∧ WS($T_i$) ∩ WS($T_j$) = ∅ )) **then**
5: return false
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: return false
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 4** Transações interativas - Mesobi.

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 $\leftarrow$ true
11: **for all** $T_j \in \prod^{T} \wedge$ WiP($T_j$) **then**
12: **if** ($\neg$ ( (WS($T_j$) $\cap$ XRS($TX_i.Op$) = $\emptyset$) $\wedge$ (RS($T_j$) $\cap$ XWS($TX_i.Op$) = $\emptyset$) $\wedge$ (WS($T_j$) $\cap$ XWS($TX_i.Op$) = $\emptyset$) ) ) **then**
13: initExec $\leftarrow$ 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 5** Certificação e confirmação - Mesobi.

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 3
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 3
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($\langle PRE - PREPARE, R_i, Order, OrderTX, v, n, d \rangle \sigma$)

**Algoritmo 6** Protocolo Bizantino - PBFT, Mesobi.

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 ← 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 ← 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($\prod^T$))

# 5 AVALIAÇÕES E RESULTADOS

Neste capítulo serão explanadas as avaliações e resultados obtidos, otimizações que podem ser feitas e a arquitetura utilizada nos testes. As avaliações e resultados consistem de duas implementações, inicialmente OB-STM onde o primeiro protótipo foi criado. O protótipo OB-STM somente executa transações pré-declaradas. No protótipo Mesobi foi introduzida a capacidade de processamento de transações interativas e algumas otimizações foram feitas para melhor desempenho. As alterações básicas na implementação consistem de melhorias no código, tamanho dos *buffers* que são enviados na fase de *pre-prepare*, não utilização de espera ocupada por *threads*, utilização de classes *singleton* e jGroups como biblioteca de comunicação de grupo.

A Figura 23 representa o cenário utilizado nos testes do algoritmo Mesobi. Em particular, os testes do protocolo OB-STM foram feitos utilizando apenas um computador e os clientes foram simulados através do uso de *threads*. Para o protocolo Mesobi foram utilizados cinco computadores físicos, dentre estes, quatro representam réplicas e um representa os clientes, os clientes foram simulados através do uso de *threads*. Os testes foram feitos utilizando placas de rede (LAN) 10/100 e a interconexão entre os módulos *ethernet* se fez utilizando um *switch* 10/100.

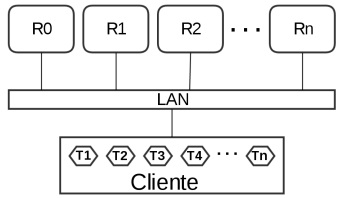


Figura 23: Representação do ambiente Mesobi para testes (4 réplicas). Tolerando uma falta (f=1).

## 5.1 AVALIAÇÃO E RESULTADOS OBSTM

Aqui será avaliado o desempenho alcançado pelo protocolo OB-STM. Os testes foram feitos usando um computador com sistema operacional Slackware Linux 2.6.37.6 SMP i686 Intel(R) Core(TM)2 Quad CPU Q9650 3.00GHz GenuineIntel GNU/Linux com 3GB memória. Foram utilizadas assinaturas digitais com chave de 512 *bits* para comunicação entre cliente e réplicas. Importante lembrar que os testes aqui mostrados foram realizados com a versão do protocolo OB-STM submetido ao SBESC. Alguns testes somente com transações declaradas foram feitos posteriormente e demonstram um melhor desempenho, os testes podem ser vistos nas figuras 41 e 42.

As Figuras 24, 25, 26 e 27 representam os resultados de 20 segundos de execução. Para a aplicação, foi utilizado um contador compartilhado e os resultados obtidos provêm da média de três execuções. Os clientes foram simulados utilizando *threads*, variando de 1 até 32. O OB-STM não foi comparado com os trabalhos de (COUCEIRO; ROMANO, 2009) e (ZHANG; ZHAO, 2012) por falta de acesso ao código.

A Figura 24 compara o OB-STM e Tazio, em um ambiente otimista, considerando somente um cliente fazendo requisições. No caso com até sete réplicas, o desempenho do OB-STM é melhor em relação ao Tazio, o que se inverte ao aumentar o número de réplicas. Com o aumento no número de réplicas o consenso acaba sendo penalizado, o que por sua vez, incorre em perda de desempenho.

A escalabilidade é mostrada na Figura 25, onde um cenário com alta contenção é avaliado. Quanto maior o número de clientes, menor as chances de haver ordenação espontânea pela

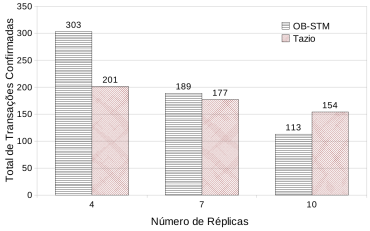


Figura 24: Número transações confirmadas.

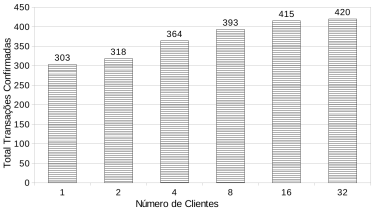


Figura 25: Escalabilidade.

rede utilizando IP-Multicast. Neste cenário a quantidade de transações aumenta uma vez que o PTA faz análise prévia de transações e permite uma boa ação a ser tomada. Por exemplo, é melhor fazer execução de transações em lote do que submeter uma a uma para execução, isso reduz a sobrecarga do protocolo de acordo (KOTLA; DAHLIN, 2004). A latência melhora com a execução de transações em lote devido a diminuição no número de mensagens trocadas para alcançar o acordo. O protocolo está tolerando uma falta, o que requer quatro réplicas.

A Figura 26 apresenta a média de transações confirmadas por segundo. A média de transações confirmadas (*committed*) alcança ponto de saturação em 32 clientes na implementação de melhor esforço.

A Figura 27 retrata o número de transações confirmadas por cliente. Embora a curva decline com o aumento no número de clientes, o resultado final de transações confirmadas é maior com mais clientes. Por exemplo, o número de transações confirmadas com dois clientes é 318 (média de 159.17 por cliente); com quatro clientes, 364 (média de 91.08 por cliente) transações confirmadas no total. O número total de transações confirmadas aumenta em 46 neste caso, apesar do aumento de contenção por parte do

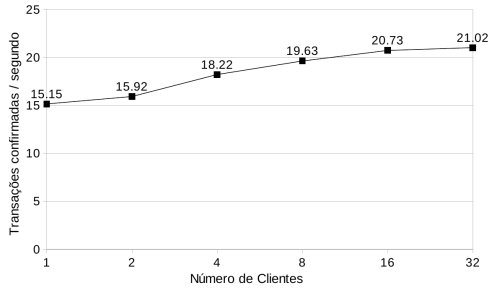


Figura 26: Transações confirmadas / segundo.

aumento do número de clientes.

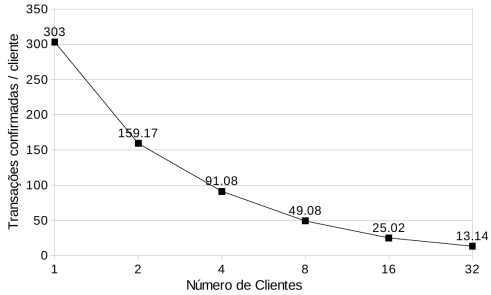


Figura 27: Transações confirmadas / cliente.

## 5.2 AVALIAÇÃO E RESULTADOS MESOBI

As avaliações foram feitas utilizando cinco computadores conectados por uma rede Local (LAN) 10/100MiB. Dentre estes, quatro foram utilizados para processamento nas réplicas, com sistema operacional Linux 3.2.0-4-amd64 SMP Debian x86_64 GNU/Linux Intel(R) Core I7 com oito núcleos de 1.6Ghz, 12GB de memória e capacidade de 2 *threads* por núcleo. Um computador para simular os clientes, rodando 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 simulação dos clientes no sistema se fez através de *threads*.

Os testes foram executados utilizando dez (10) requisições por cliente, sendo que cada requisição contém apenas uma operação, que consiste em ler ou alterar os dados contidos em uma lista encadeada que é composta por dez objetos (valores/dados). Quando dito sem conflitos nos gráficos, 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 28 representa a taxa média de commits por segundo. O fator de escrita consiste em 10, 30, 50 e 90%.

Na Figura 29 pode ser visto a taxa de *commits* BFT efetuados em função do número total de requisições emitidas pelos clientes. Nota-se que consoante o número de clientes aumenta, a taxa de *commits* BFT aumenta, esse comportamento era esperado, pois menor é a probabilidade de ordenação espontânea pela rede. Sem ordenação espontânea, o protocolo Bizantino precisa ser iniciado para resolver conflitos inerente a transações confli-

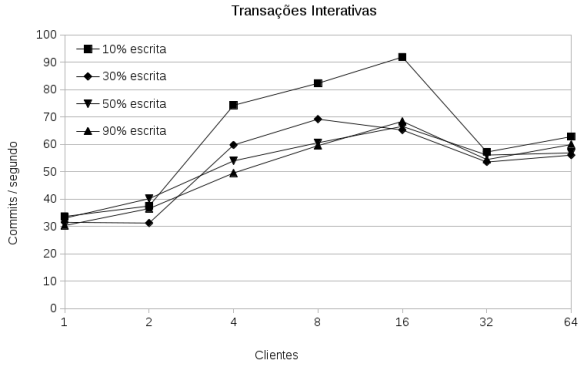


Figura 28: Transações interativas (commits / segundo).

tantes fora de ordem.

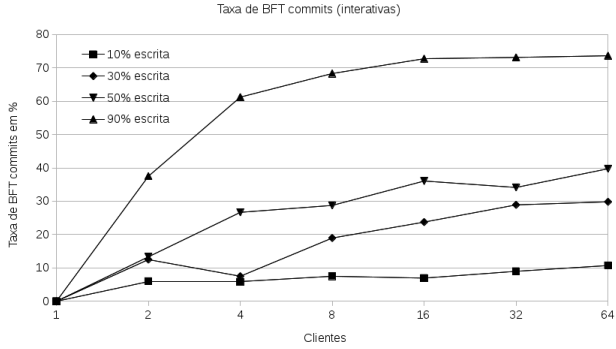


Figura 29: Taxa de *commits* BFT, transações interativas.

A Figura 30 apresenta a taxa de *aborts* percebido pelos clientes interativos na execução de suas transações. Nota-se também que conforme o aumento no número de clientes, maior a taxa de *aborts* percebido pelos clientes. Percebe-se certa estabilização em torno de dezesseis (16) clientes.

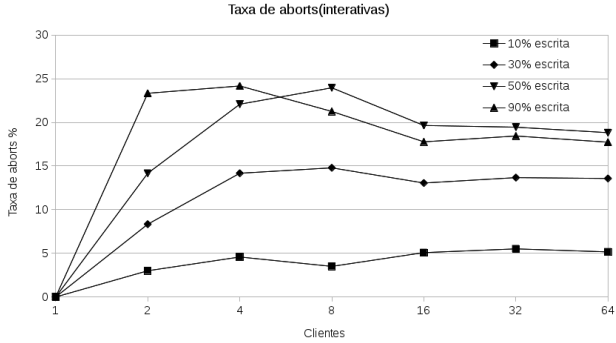


Figura 30: Taxa de *aborts*, transações interativas.

As Figuras 31, 32, 33 e 34 representam a execução de transações interativas juntamente com declaradas sem conflitos. As taxas de atualização variam em 10, 30 50 e 90%. As transações declaradas são representadas por sua primeira letra assim como as interativas.

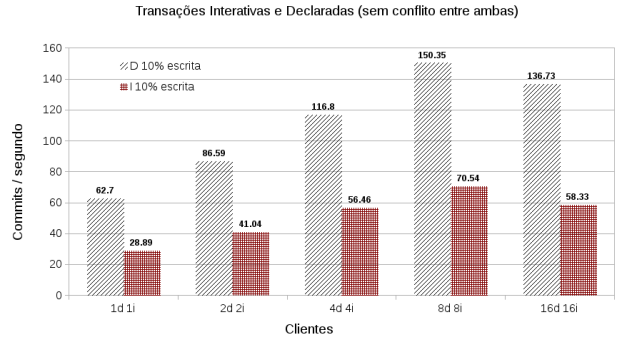


Figura 31: Transações pré-declaradas (D) e interativas (I), sem conflito de dados e 10% de escrita.

A Figura 35 apresenta um resumo consolidado das Figuras 31, 32, 33 e 34. As transações não conflitam entre si e percebe-se

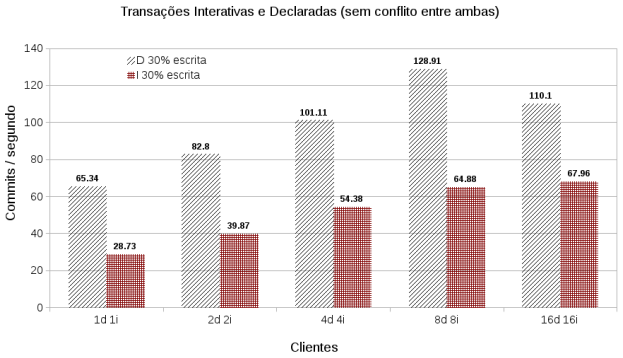


Figura 32: Transações pré-declaradas (D) e interativas (I), sem conflito de dados e 30% de escrita.

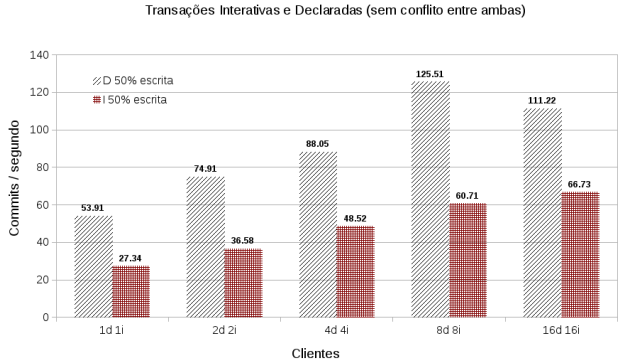


Figura 33: Transações pré-declaradas (D) e interativas (I), sem conflito de dados e 50% de escrita.

um número maior de confirmações de transações pré-declaradas. Mesmo comparando transações interativas com taxa de escrita em 10% e transações pré-declaradas com taxa de escrita em 90%, as transações pré-declaradas levam a melhor.

As Figuras 36, 37, 38 e 39 representam a execução de transações interativas juntamente com declaradas com conflitos.

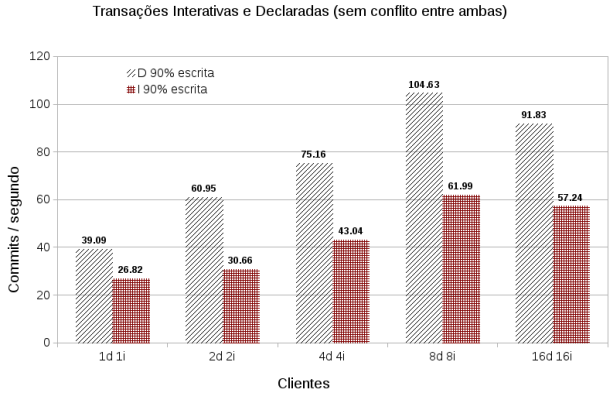


Figura 34: Transações pré-declaradas (D) e interativas (I), sem conflito de dados e 90% de escrita.

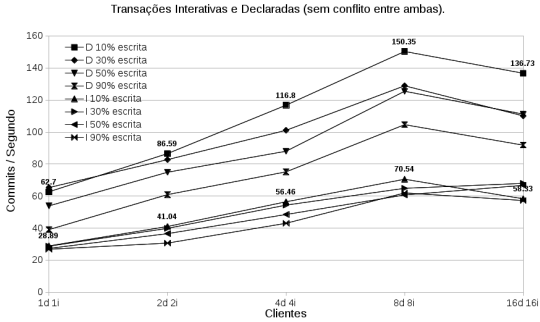


Figura 35: Consolidado transações pré-declaradas (D) e interativas (I), sem conflito de dados.

As taxas de atualização variam em 10, 30 50 e 90% e transações de tipos diferentes são conflitantes.

A Figura 40 apresenta um resumo consolidado das Figuras 36, 37, 38 e 39. As transações são conflitantes entre os diferentes tipos, percebe-se um número maior de confirmações de transações pré-declaradas novamente. As inversões que ocorrem são um

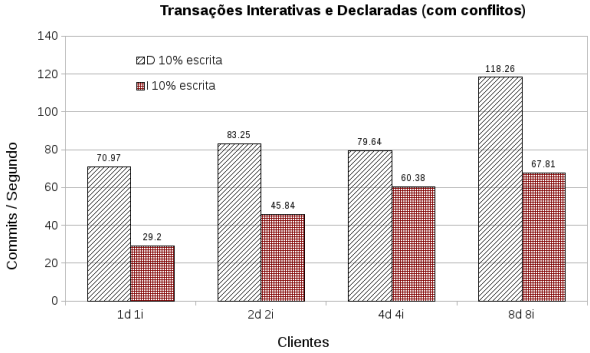


Figura 36: Transações pré-declaradas (D) e interativas (I), com conflito de dados e 10% de escrita.

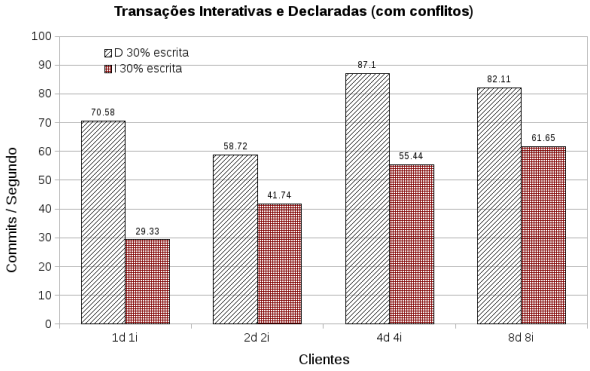


Figura 37: Transações pré-declaradas (D) e interativas (I), com conflito de dados e 30% de escrita.

ponto interessante de se analisar. Partindo do princípio que transações interativas têm prioridades sobre as pré-declaradas. As inversões são frutos da necessidade de espera de transações pré-declaradas pelas transações interativas. O que acaba por comprometer o desempenho das mesmas.

A Figura 41 apresenta testes de execução de transações

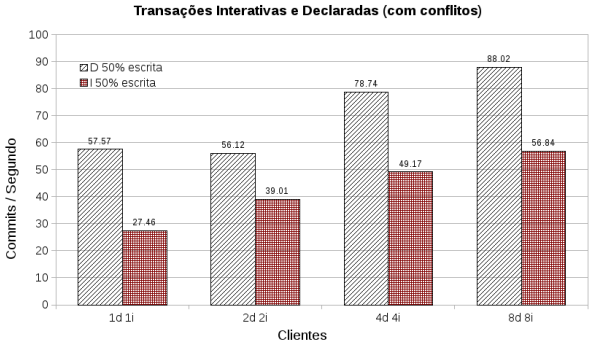


Figura 38: Transações pré-declaradas (D) e interativas (I), com conflito de dados e 50% de escrita.

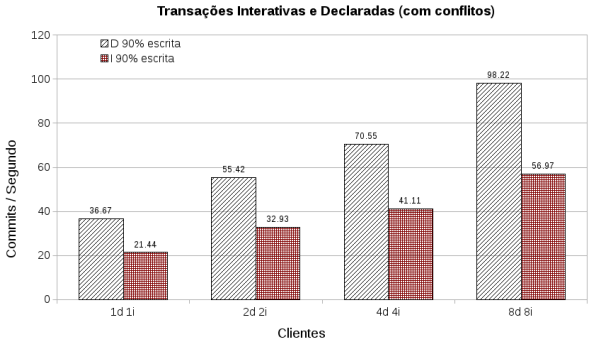


Figura 39: Transações pré-declaradas (D) e interativas (I), com conflito de dados e 90% de escrita.

pré-declaradas feitos em uma lista encadeada. As taxas de atualização foram definidas em: 10, 30, 50 e 100%. Pode-se concluir que o ponto de saturação do sistema está em torno de dezesseis clientes. Consoante o número de clientes aumenta, menor é a taxa de confirmação, isso era esperado devido a não ordenação das mensagens pela rede. O que implica na execução do protocolo Bizantino para definição da ordem que deve ser seguida.

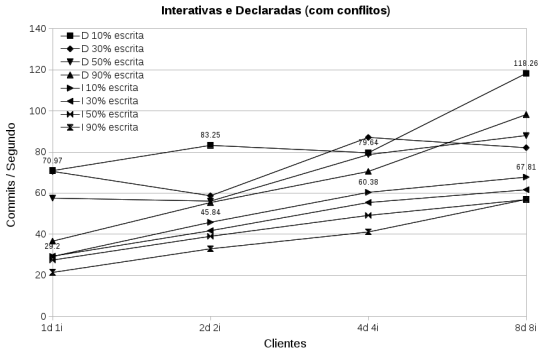


Figura 40: Consolidado transações pré-declaradas (D) e interativas (I), sem conflito de dados.

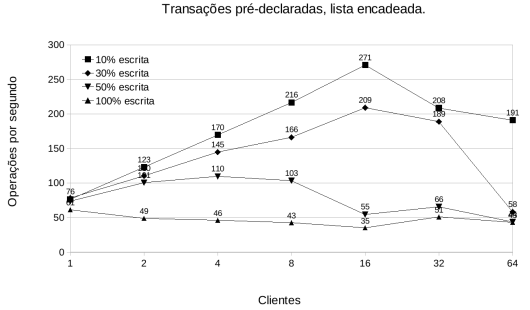


Figura 41: Transações pré-declaradas: operações / segundo.

A Figura 42 apresenta a relação entre confirmação otimista e confirmações que necessitaram do protocolo Bizantino para resolução de conflitos inerente a mensagens fora de ordem. Pode-se perceber que consoante a quantidade de clientes aumenta, menor a taxa de confirmações otimistas. Quando o número de clientes ultrapassa os dezesseis, praticamente todas as transações que são confirmadas são através do protocolo Bizantino. Conforme citado anteriormente, era esperado a saturação de *commits* otimistas com o aumento do número de clientes.

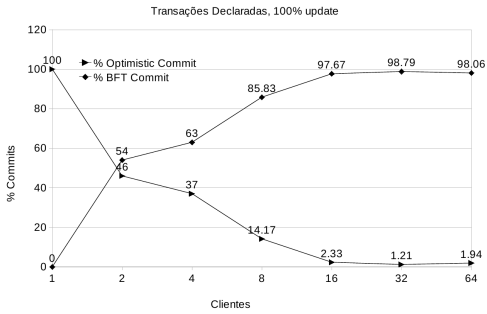


Figura 42: Transações pré-declaradas: % Confirmação otimista e Bizantino, 100% atualização.

## 5.3 CORRETUDE

Nessa seção será feita a análise de corretude para o caso base e pior caso. O caso base consiste apenas de um cliente no sistema. Para o pior caso, é utilizado apenas o primeiro nível (ver Figura 43) na fase de certificação total, é necessário que o número de clientes ultrapasse a resiliência de $3f+1$ para que cada réplica receba uma mensagem diferente. As análises nesta seção serão feitas utilizando transações conflitantes (100% atualização) e a sequência abaixo será realizada:

- (1) análise somente de transações pré-declaradas no sistema.
- (2) análise somente de transações interativas no sistema.
- (3) análise de transações pré-declaradas juntamente com transações interativas no sistema (com e sem réplicas maliciosas).

Cada transação recebida por uma réplica é armazenada em um *buffer* como visto nas Figuras 43 e 44. A etapa de certificação total é feita utilizando o primeiro elemento de cada *buffer* (nível 1). A etapa de certificação total poderia ser feita utilizando mais níveis, mas a análise de corretude seria outra. A análise para os três casos será feita apenas com o primeiro nível.

#### *5.3.0.0.1 Transações pré-declaradas*

A Tabela 5 retrata o *buffer* $\prod^T$ de transações pré-declaradas recebidas pelo sistema em uma execução normal e os itens $a$, $b$, $c$ e $d$ representam a ordem de chegada das mensagens pelas réplicas.

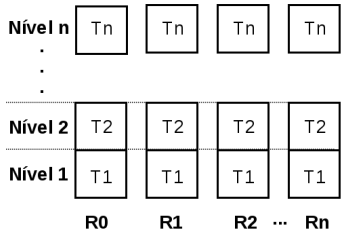


Figura 43: Recebimento ordenado.

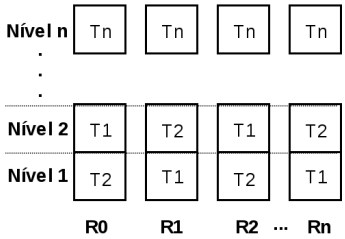


Figura 44: Recebimento fora de ordem.

Seguindo o ciclo de vida das transações retratado na Figura 22 a transação $t_1$ nas réplicas $R_{0,2}$ poderá ser executada, pois quando foram recebidas não haviam outras transações concorrentes, visão local (item *a* da Tabela 5). Nos itens *b*, *c* e *d* podem ser vistos que a ordem de recebimento das mensagens foram iguais em todas as réplicas. Quando a ordem é a mesma em todas réplicas, o protocolo confirmará de forma otimista as transações, pois na fase de certificação total, todas réplicas retornarão "*OK!*" para a mensagem *AskForCommit*, pois não há mensagens fora de ordem nem réplicas maliciosas. Quando a ordem de recebimento é igual em todas as réplicas, o sistema se comporta como se houvesse somente um cliente.

| | |
|---|---|
| $R_0$ | $t_1$ |
| $R_1$ | |
| $R_2$ | $t_1$ |
| $R_3$ | |

a

| | |
|---|---|
| $R_0$ | $t_1$ |
| $R_1$ | $t_1$ |
| $R_2$ | $t_1$ |
| $R_3$ | $t_1$ |

b

| | | |
|---|---|---|
| $R_0$ | $t_1$ | $t_2$ |
| $R_1$ | $t_1$ | |
| $R_2$ | $t_1$ | $t_2$ |
| $R_3$ | $t_1$ | |

c

| | | |
|---|---|---|
| $R_0$ | $t_1$ | $t_2$ |
| $R_1$ | $t_1$ | $t_2$ |
| $R_2$ | $t_1$ | $t_2$ |
| $R_3$ | $t_1$ | $t_2$ |

d

Tabela 5: Visão global do *buffer* de transações pré-declaradas (em ordem).

*5.3.0.0.2 Requisições fora de ordem*

O pior caso para requisições fora de ordem está em função da cardinalidade do número de réplicas, $\mid R \mid \geq 3f + 1$. Como é utilizado somente o primeiro nível na certificação, a análise do pior caso independe do número de clientes quando maior que $3f + 1$. A análise será feita utilizando $f = 1$, o que resulta em quatro réplicas. Para essa análise quatro clientes serão utilizados e o conjunto de transações $\prod^{T} = \{t_1, t_2, t_3, t_4\}$ será utilizado. A Tabela 6 representa a ordem de chegada de cada transação na réplica. A ordem de chegada consiste da esquerda para direita, sendo assim, as transações $\{t_1, t_4, t_3, t_2\}$ são as primeiras nas réplicas $R_{0,1\ ,2\ ,3}$ respectivamente. Como pode ser visto no destaque na Tabela 6.

| | | | | |
|---|---|---|---|---|
| $R_0$ | $\boldsymbol{t_1}$ | $t_2$ | $t_3$ | $t_4$ |
| $R_1$ | $\boldsymbol{t_4}$ | $t_1$ | $t_2$ | $t_3$ |
| $R_2$ | $\boldsymbol{t_3}$ | $t_4$ | $t_1$ | $t_2$ |
| $R_3$ | $\boldsymbol{t_2}$ | $t_3$ | $t_4$ | $t_1$ |

Tabela 6: Visão global do *buffer* de transações pré-declaradas (fora de ordem).

Inicialmente cada réplica executará uma transação diferente. Ao término da transação $t_1$ pela réplica $R_0$, uma mensagem *AskForCommit* é gerada e encaminhada para cada réplica solicitando permissão para confirmar a transação $t_1$. As réplicas $R_{1\ ,2\ ,3}$ (nesse caso) retornariam uma negação ao pedido de confirmação ("*NOK!*"). Com a negação do pedido de confirmação, a réplica $R_0$, se for a réplica primária inicia o protocolo Bizantino, se não, faz solicitação para início do protocolo Bizantino. Ao fazer a solicitação para início do protocolo Bizantino, a réplica

solicitante encaminha junto à mensagem de negação recebida e seu histórico parcial de execução para todas réplicas. Com isso é possível iniciar a troca de visão caso o líder seja malicioso.

O protocolo Bizantino definirá a sequência a ser executada em função do seu conjunto de transações, nesse caso representado na Tabela 6. Supõe-se que o líder seja a réplica $R_1$ e que a ordem definida é o conjunto de transações locais: $\prod^{Ordem} = \{t_4, t_1, t_2, t_3\}$. Após o consenso executado pelo protocolo Bizantino, uma nova ordem é estabelecida. A ordem definida é representada na Tabela 7.

| | | | | |
|---|---|---|---|---|
| $R_0$ | $\boldsymbol{t_4}$ | $t_1$ | $t_2$ | $t_3$ |
| $R_1$ | $\boldsymbol{t_4}$ | $t_1$ | $t_2$ | $t_3$ |
| $R_2$ | $\boldsymbol{t_4}$ | $t_1$ | $t_2$ | $t_3$ |
| $R_3$ | $\boldsymbol{t_4}$ | $t_1$ | $t_2$ | $t_3$ |

Tabela 7: Visão global do *buffer* de transações pré-declaradas (após ordenação).

Ao final do protocolo todas as réplicas possuem a mesma sequência a ser executada. Com a definição da ordem, todas as réplicas passam para o estado *RE-EXEC-TD* mostrado na Figura 22 e re-executam as transações na mesma ordem. Nesse caso, apenas a réplica $R_1$ não necessitará re-executar a transação $t_4$, pois a mesma já foi executada, sendo assim, somente faz a confirmação de $t_4$ e retorna a resposta ao cliente. Importante lembrar que a ordem é definida utilizando as transações existentes nos *buffers* $\prod^{T}$ e $\prod^{TX}$

*5.3.0.0.3 Transações interativas*

Para as transações interativas, as operações são executadas logo que são recebidas, se não houver transações pré-declaradas

conflitantes. O conjunto de transações que será analisado nesse exemplo é $\prod^{TX} = \{t_{x1}, t_{x2}, t_{x3}, t_{x4}\}$. Cada transação opera sobre três operações que variam de $[a...i]$. Ao receber uma operação para ser executada, por mais que existam conflitos entre as operações de outras transações interativas, a operação é executada (visão local). Os conflitos entre transações interativas só são tratados na fase de confirmação da transação (transações interativas com interativas). A Figura 45 retrata quatro linhas de execução para cada réplica. Pode-se perceber que existem conflitos entre as operações $c, e$ e $g$ (transações em réplicas distintas executam operações conflitantes, visão global). Os conflitos nesse caso só seriam capturados em tempo de confirmação quando a verificação global ocorresse. A execução segue até que o cliente envie uma mensagem de *abort* ou *commit*. Supõe-se que o recebimento da mensagem de *commit* aconteceu de forma diferente em algumas réplicas, como retratado na Figura 45. Ao receber a mensagem de *commit*, cada réplica imediatamente faz solicitação de confirmação da transação interativa em questão (estado A.F.C.I. da Figura 22 e linha 21 do algoritmo 4). As réplicas $R_{2,3}$ primeiramente solicitam confirmação para a transação $t_{x3}$ pois receberam a mensagem de confirmação primeiro, e as réplicas $R_{0,1}$ primeiramente solicitam confirmação para as transações $t_{x1}$ e $t_{x4}$ respectivamente.

Logo em seguida, outras mensagens de confirmação chegam ao sistema. A chegada das operações de *commit* enviada pelos clientes consiste da seguinte ordem para as réplicas:

- $R_0$: $\prod^{TX} = \{t_{x1.Commit}, t_{x2.Commit}, t_{x3.Commit}, t_{x4.Commit}\}$.
- $R_1$: $\prod^{TX} = \{t_{x4.Commit}, t_{x1.Commit}, t_{x2.Commit}, t_{x3.Commit}\}$.
- $R_2 \wedge R_3$: $\prod^{TX} = \{t_{x3.Commit}, t_{x4.Commit}, t_{x1.Commit}, t_{x2.Commit}\}$.

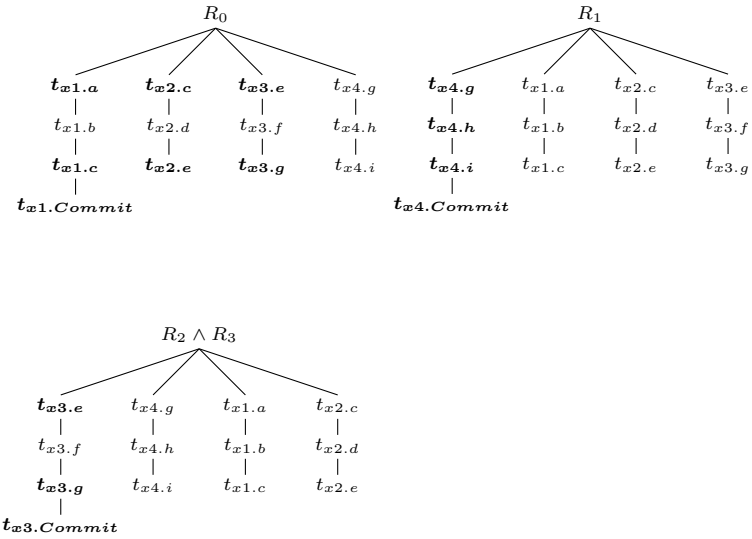


Figura 45: Transações interativas, requisição de commit fora de ordem.

Será feito análise apenas para uma réplica, e para as conseguintes a ideia segue a mesma. A réplica $R_1$ ao receber a mensagem $t_{x4.Commit}$ passa para o estado *A.F.C.I.* e inicia a fase de confirmação de transação interativa. Nessa fase cada réplica envia uma mensagem chamada *AskForCommitInteractive*, a mensagem contém o *timestamp* do início ($t_{xi.Begin}$) e de cada operação ($t_{xj.Op}$) recebida pela réplica. Com isso é possível verificar qual transação começou primeiro e utilizar o critério temporal como meio de desempate. Mais especificamente, pelas definições básicas do sistema e premissas na Seção 4.1, os clientes se comportam corretamente e com isso é garantido que os *timestamp's* utilizados pelas réplicas serão os mesmos.

A réplica $R_1$ será definida como primária. A ordem de recebimento da réplica $R_1$ difere das demais como pode ser visto na Figura 45. Como a ordem de recebimento é distinta entre a réplica $R_1$ e as outras, as respostas na fase de confirmação serão negativas ("*NOK!*") e com isso o protocolo Bizantino precisa ser iniciado para definir qual ordem seguir, ou melhor, qual das quatro linhas de execução deve ser confirmada (*committed*). A Figura 46 retrata as quatro linhas de execução da réplica $R_1$ e em negrito a ordem escolhida pela réplica primária para ser confirmada. Todas as réplicas confirmam $t_{x4}$ e abortam as demais transações. Com a confirmação da transação a resposta é enviada ao cliente.

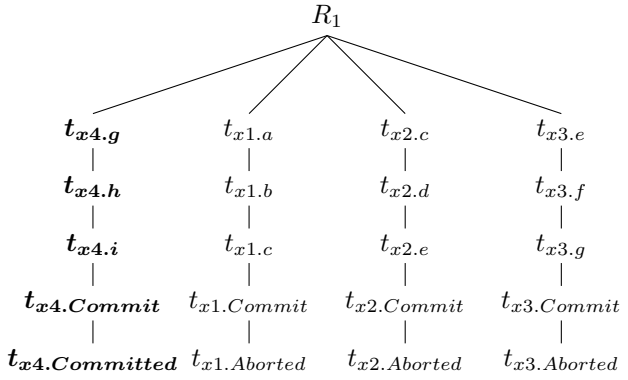


Figura 46: Transações interativas, visão local da réplica $R_1$.

*5.3.0.0.4 Transações interativas e pré-declaradas*

A análise entre transações interativas e declaradas necessita que o número de clientes ultrapasse $3f + 1$, nesse caso, a análise será feita apenas para $f = 1$; o número de clientes é quatro. Sendo o pior caso onde cada réplica possui uma transação diferente, como representado na Tabela 8.

| | | | | |
|---|---|---|---|---|
| $R_0$ | $\boldsymbol{t_{x1}}$ | $t_{x2}$ | $t_1$ | $t_2$ |
| $R_1$ | $\boldsymbol{t_2}$ | $t_{x1}$ | $t_{x2}$ | $t_1$ |
| $R_2$ | $\boldsymbol{t_1}$ | $t_2$ | $t_{x1}$ | $t_{x2}$ |
| $R_3$ | $\boldsymbol{t_{x2}}$ | $t_1$ | $t_2$ | $t_{x1}$ |

Tabela 8: Visão global do *buffer* de transações pré-declaradas e interativas (pior caso).

Transações que estão em destaque[1] na Tabela 8 representam a primeira na fila de execução. É suposto que a réplica $R_3$ é a réplica primária e é a primeira a entrar na fase de certificação. Ao término das transações $t_{x1}, t_2, t_1$ e $t_{x2}$ pelas réplicas $R_0, R_1, R_2$ e $R_3$ respectivamente, a fase de certificação total deve entrar em ação. Ao solicitar permissão de confirmação da transação $t_{x2}$ pela réplica $R_3$, uma mensagem de negação de confirmação (NOK!) será retornada pelas outras réplicas no sistema. A mensagem de negação é gerada pois a ordem entre $R_3$ e as outras réplicas são distintas. Com o recebimento da mensagem NOK! a réplica primária propõe a ordem e inicia o protocolo de consenso. Após o consenso, a ordem final de execução é gerada e as réplicas podem atuar na ordem imposta pelo protocolo Bizantino. A ordem estabelecida pode ser vista na Tabela 9.

Seguindo o protocolo, transações interativas podem sem

[1] Primeira coluna.

| | | | | |
|---|---|---|---|---|
| $R_0$ | $\boldsymbol{t_{x2}}$ | $t_{x1.Abort}$ | $t_1$ | $t_2$ |
| $R_1$ | $\boldsymbol{t_{x2}}$ | $t_{x1.Abort}$ | $t_1$ | $t_2$ |
| $R_2$ | $\boldsymbol{t_{x2}}$ | $t_{x1.Abort}$ | $t_1$ | $t_2$ |
| $R_3$ | $\boldsymbol{t_{x2}}$ | $t_{x1.Abort}$ | $t_1$ | $t_2$ |

Tabela 9: Visão global do *buffer* de transações pré-declaradas e interativas, após ordenação.

abortadas, pois suas operações podem ser dependentes de operações no cliente. As transações $t_{x2}, t_1$ e $t_2$ são confirmadas e $t_{x1}$ é abortada.

*5.3.0.0.5 Transações na presença de réplica(s) maliciosa(s)*

A análise para transações na presença de réplicas maliciosas é explanada abaixo. A Tabela 10 retrata a ordem de chegada das transações nas réplicas. A réplica $R_3$ é maliciosa, então pode ou não se comportar de forma correta, ou seja, a réplica maliciosa pode retornar tanto uma permissão de confirmação como uma negação.

| | | | |
|---|---|---|---|
| $R_0$ | $t_{x1}$ | $t_1$ | |
| $R_1$ | $t_{x1}$ | $t_1$ | |
| $R_2$ | $t_{x1}$ | $t_1$ | |
| $R_3$ | $t_1$ | $t_{x1}$ | Maliciosa |

Tabela 10: Visão global do *buffer* de transações pré-declaradas e interativas (com réplica maliciosa).

Na fase de certificação as réplicas $R_0, R_1$ e $R_2$ solicitam permissão para confirmar a transação $t_{x1}$. Como a réplica $R_3$ é maliciosa, ela tenta corromper o sistema enviando mensagens diferentes para as réplicas. As réplicas corretas, a saber $R_0, R_1$ e $R_2$ retornam OK! uma às outras. No entanto, a réplica $R_3$ retorna

OK! para as réplicas $R_0$ e $R_1$ e NOK! para $R_2$ como pode ser visto na Tabela 11. Dessa forma as réplicas $R_0$ e $R_1$ confirmam a transação $t_{x1}$. Porém a réplica $R_2$ recebeu uma mensagem NOK! para a confirmação da transação $t_{x1}$, com isso o protocolo Bizantino precisa ser iniciado. Para o início do protocolo Bizantino a réplica $R_2$ solicita ao primário que inicie o protocolo. Para iniciar o protocolo Bizantino a mensagem de negação enviada por $R_3$ é encaminhada junto com a solicitação de início e o histórico de execução[2] da réplica.

| Recebe | R0 | R1 | R2 | R3 |
|---|---|---|---|---|
| R0 | - | OK! | OK! | OK! |
| R1 | OK! | - | OK! | OK! |
| R2 | OK! | OK! | - | **NOK!** |
| R3 | OK! | OK! | OK! | - |

Tabela 11: Visão término fase AskForCommit de cada réplica.

A garantia de consistência é alcançada através da comparação dos históricos executados pelas réplicas. Dessa forma os históricos parciais de cada réplica pode ser visto logo abaixo:

- $\mathcal{H}^{p}_{R_{0,1}} = \{t_{x1.Committed}\}$
- $\mathcal{H}^{p}_{R_2} = \{t_{x1}\}$
- $\mathcal{H}^{p}_{R_3} = \{***\}$

O histórico de réplicas maliciosas não é considerado, uma réplica é considerada maliciosa também, se responder OK! a uma mensagem A.F.C. e "*NOK!*"para a mesma transação em execução. A réplica primária em posse do histórico de execução da réplica

[2]Apenas o histórico parcial é enviado.

solicitante, no caso $R_2$, faz análise de seriabilidade[3] e a partir disso o protocolo Bizantino entra em execução.

Importante salientar que para transações declaradas, apenas uma *thread* executa a mesma (quando há conflito) e para transações interativas múltiplas *threads* podem estar concorrentemente em execução. Isso acontece pois transações interativas não verificam conflitos entre transações interativas. Múltiplas *threads* podem estar em execução para transações pré-declaradas se não existirem conflitos entre as transações.

### 5.3.1 *Vivacidade*

A cada remoção de um elemento do *buffer* de transações declaradas $\prod^T$ ou interativas $\prod^{TX}$, as transações em espera são verificadas sobre a possibilidade de executar. Essas remoções são efetuadas pelo gerente de transações através das primitivas *commit* e *abort*. Quando transações interativas sofrem *abort*, são logo retiradas do *buffer* de transações interativas, e transações no *buffer* de transações declaradas são novamente verificadas sobre a possibilidade de executar. Transações declaradas são removidas após sua execução e confirmação.

As propriedades se mantêm para todas transações, garantindo assim que o sistema não estagne (*liveness*).

- Definições:
  - $\prod^T = \{t_1, t_2, t_3, ...t_n\}$ conjunto de transações declaradas.

[3] Verifica se a ordem é igual e define qual deve ser a ordem seguida levando em conta os históricos de execução, mais detalhes podem ser vistos na Seção 5.3.2.

- $\prod^{TX} = \{t_{x1}, t_{x2}, t_{x3}, ...t_{xn}\}$ conjunto de transações interativas.
- *commit*($T_{xi}$): ao confirmar $T_{xi}$, $T_{xi}$ é retirado do conjunto $Tx$. O conjunto de transações $T$ é verificado sobre a possibilidade de iniciar outra transação que não possua conflitos. Pode ser que transações declaradas não executem devido a transações interativas no sistema, no entanto, logo após que o *commit* ou *abort* ocorram, uma nova varredura é feita.
- *commit*($Ti$): ao confirmar $T_i$, $T_i$ sai do conjunto de transações $T$. $Tx$ é varrido sobre a possibilidade de liberar operações que estejam paradas. Logo após a busca por operações interativas paradas, o conjunto $T$ é verificado sobre a possibilidade de executar transações.
- *abort*($T_{xi}$): $T_{xi}$ é removido do conjunto $Tx$. $T$ é varrido novamente sobre possibilidade de executar transações.
- *abort*($T_i$): $T_i$ não é retirado do conjunto até sua confirmação. *Aborts* em transações pré-declaradas levam a execução do protocolo Bizantino, com isso a ordem das transações é definida e sua execução e confirmação segue a ordem imposta. Sendo assim, todas transações pré-declaradas que se encontram no conjunto $T$ que não foram confirmadas de forma otimista, serão confirmadas através do protocolo Bizantino. Logo após a execução e confirmação das transações, uma nova varredura é feita.

- Considerando:
  - $Td \rightarrow$ transação declarada.

  - $Tx \rightarrow$ transação interativa.
  - wait $\rightarrow$ transação em espera.
  - exec $\rightarrow$ transação em execução.
  - a...z $\rightarrow$ transações.

- Premissas:
  - $\forall Td^{a}_{wait}\ \exists\, Td^{b}_{exec}$ s.t. $Td^{b}_{exec} \rightarrow \mathit{commit}(Td_b) \vee \mathit{abort}(Td_b)$
  - $\forall Td^{a}_{wait}\ \exists\, Tx^{a}_{exec}$ s.t. $Tx^{a}_{exec} \rightarrow \mathit{commit}(Tx_a) \vee \mathit{abort}(Tx_a)$
  - $\forall Tx^{a}_{wait}\ \exists\, Td^{b}_{exec}$ s.t. $Td^{b}_{exec} \rightarrow \mathit{commit}(Td_b) \vee \mathit{abort}(Td_b)$

- Então:
  - $\forall (Td^{a}_{wait} \vee Tx^{a}_{wait})\ \exists\ (Td^{b}_{exec} \vee Tx^{b}_{exec})$ s.t. $Td^{b}_{exec} \vee Tx^{b}_{exec} \rightarrow \mathit{commit}(Td^{b} \vee Tx^{b}) \vee \mathit{abort}(Td^{b} \vee Tx^{b})$
  - $\mathit{commit}(Td^{b} \vee Tx^{b}) \vee \mathit{abort}(Tx^{b}) \rightarrow Td^{a}_{exec} \vee Tx^{a}_{exec}$

### 5.3.2 *Consistência*

Considerando:

- $R_x\mathcal{H}^{p}_{c} \rightarrow$ Histórico parcial confirmado (*committed*) da réplica x.
- $R_x\mathcal{H}^{p}_{nc} \rightarrow$ Histórico parcial não confirmado (*not committed*) da réplica x.
- 1...n $\rightarrow$ transações.

Partindo do princípio que toda transação que não foi definida para ser executada nem confirmada pelo protocolo Bizantino, a transação deve fazer a solicitação de confirmação usando o

protocolo otimista. Com isso, cada réplica ao receber uma mensagem de negação, anexa seu histórico parcial confirmado e não confirmado juntamente na mensagem de solicitação de início do protocolo Bizantino, como pode ser visto abaixo.

- initBFT(($\mathcal{H}_c^p$ + $\mathcal{H}_{nc}^p$), Mensagem de negação recebida)

A réplica primária faz verificação de autenticidade da mensagem de negação usando a mensagem de negação anexa.

A réplica primária[4] ao receber a mensagem de início do protocolo Bizantino, define a mensagem de *propose* com base nos históricos parciais recebidos da réplica solicitante. A mensagem de *propose* é composta da seguinte forma, sendo que, $R_1$ é a réplica solicitante do início do protocolo Bizantino:

- $Propose = (R_1\mathcal{H}_c^p \cup R_1\mathcal{H}_{nc}^p) \setminus R_0\mathcal{H}_{nc}^p$

É suposto que dois clientes $C_1$ e $C_2$ emitam uma transação cada, sendo elas: $T_1$ e $T_2$ respectivamente. A ordem das transações é $T_1, T_2$ e é a mesma em todas as réplicas, sendo que uma réplica é maliciosa. A réplica maliciosa tentará corromper uma das réplicas (não a primária, quando a réplica primária recebe uma mensagem de negação a análise é outra, abordada logo mais abaixo nesse texto.). As réplicas corretas $R_0$ e $R_1$ receberam a mensagem de confirmação da transação, sendo que, $R_2$ recebeu a mensagem de negação. Com isso as réplicas $R_0$ e $R_1$ confirmam a transação $T_1$ e $R_2$ solicita o início do protocolo Bizantino. A mensagem de *Propose* é formada da seguinte forma:

- $Propose = (((R_2\mathcal{H}_c^p \cup R_2\mathcal{H}_{nc}^p) \setminus R_0\mathcal{H}_{nc}^p) \cup \mathcal{H}_{nc}^p)$
- $Propose = ((((\{\emptyset\} \cup \{T_1, T_2\}) \setminus \{T_2\}) \cup \{T_2\})$

[4]Para esse exemplo a réplica $R_0$ será definida como primária.

- $Propose = \{T_1, T_2\}$

Dessa forma a réplica $R_2$ vai confirmar a transação $T_1$ e em seguida executar a transação $T_2$. As réplicas $R_0$ e $R_1$ como já confirmaram a transação $T_1$, apenas executarão a transação $T_2$. Importante lembrar que em caso de mensagens fora de ordem, não haverá confirmações de transações sem a execução do protocolo Bizantino. A confirmação da transação $T_1$ pelas réplicas $R_0$ e $R_1$ só aconteceram porque $R_3$ é uma réplica maliciosa. Sendo assim, existe uma janela[5] em que as réplicas estão desalinhadas. O desalinhamento entre as réplicas não é percebido pelo cliente, pois o mesmo necessita de $f + 1$ respostas iguais para suas operações. Logo que o protocolo Bizantino termina, as réplicas se alinharão novamente.

Quando a réplica primária é a própria que recebe a mensagem de negação de confirmação da transação a proposta da ordem (*propose*) é feita de forma diferenciada. A mensagem de *propose* é criada em função do histórico não confirmado da réplica primária e é formada da seguinte forma:

- $Propose = R_0\mathcal{H}^p_{nc}$
- $Propose = \{T_1, T_2\}$

Importante lembrar que com ordens distintas, não haverá confirmações de transações sem uso do protocolo Bizantino. Isso acontece porque réplicas corretas sempre retornarão uma mensagem "*NOK*"para transações fora de ordem. No modelo implementado, novas transações conflitantes precisarão esperar pelo fim do protocolo Bizantino para serem iniciadas.

[5] A janela de desalinhamento acontece entre a confirmação de uma transação por uma réplica e o término e execução das transações definidas no *propose* do protocolo Bizantino.

## 5.4 OTIMIZAÇÃO: OBSTM E MESOBI

Existem algumas otimizações que podem ser feitas para melhorar o desempenho e diminuir o número de mensagens trafegadas pela rede. A fase de certificação de uma transação (linha 21 do Algoritmo 3) 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 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.

Outra otimização que pode ser feita é utilizar filtros de *Bloom* na fase de validação / certificação. Filtro de *Bloom* pode ser definido como uma estrutura de dados probabilística que é utilizado para verificar se um elemento está ou não em um conjunto de elementos. Essa verificação por sua vez pode incorrer em falsos positivos, mas falsos negativos não ocorrem. Os filtros de *Bloom* são preenchidos utilizando funções *hash*. A verificação de conflitos consistiria em verificar se um dado de escrita ou leitura se encontra nas estruturas utilizadas por outras transações, caso essa verificação resulte em positivo, um conflito existe (com uma taxa de erro ajustável). Caso a verificação retorne negativa, significa que não existem conflitos (sem taxa de erros). Essa otimização economizaria a utilização da rede, pois diminuiria o tamanho dos pacotes na fase de certificação.

# 6 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 seguir lógicas diferentes do ponto de vista da aplicação com base nos resultados obtidos, além de tornar possível a realização de operações confidenciais; ao mesmo tempo, transações pré-declaradas são úteis pois permitem certa flexibilidade em relação ao momento em que serão executadas. Esta dissertação apresenta o OB-STM e Mesobi. O OB-STM permite execução de transações pré-declaradas apenas, já com o Mesobi é possível a execução de transações interativas e pré-declaradas no mesmo ambiente.

O Mesobi permite execução de transações utilizando verificação antecipada de conflitos (PTA) junto com validação tardia (*deferred update*) para que transações fora de ordem possam ser corretamente executadas. Permite sincronização entre réplicas utilizando os históricos de transações não confirmadas e confirmadas. O modelo apresentado é tolerante a faltas arbitrárias no que diz respeito às réplicas. 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.

Um dos pontos cruciais no trabalho aqui apresentado está

na integração entre a certificação otimista e o protocolo Bizantino. A maior dificuldade está em manter a serialização do histórico na presença de réplicas maliciosas.

## 6.1 TRABALHOS FUTUROS

Implementar novos modelos de testes para verificar desempenho utilizando *benchmarks* padronizados (*Array-list, B-Tree, ...*). Pôr em prática as otimizações e efetuar novos testes.

## 6.2 CONTRIBUIÇÃO

Esta dissertação trata de tolerância a faltas Bizantinas em um ambiente de Memória Transacional em Software. Como pode ser visto na Tabela 2 do Capítulo 3, somente o trabalho de Zhang (ZHANG; ZHAO, 2012) trata tolerância a faltas Bizantinas, sendo que seu modelo aborta transações somente leitura e utiliza um total de $(3f+1)+(2f+1)$ para alcançar tolerância a faltas Bizantinas. O modelo proposto aqui utiliza $3f+1$ réplicas para alcançar o mesmo fim e não aborta transações somente leitura, sendo que a execução de transações inicia de forma otimista. O modelo de Zhang utiliza somente transações pré-declaradas, a proposta dessa dissertação contempla os dois modelos de transações: pré-declaradas e interativas.

Durante o mestrado foram realizadas publicações com o objetivo de validar as propostas desenvolvidas. Foram publicados os seguintes artigos:

- Ribeiro, Tulio Alberton; Lung, Lau Cheuk; Netto, Hylson Vescovi. OB-STM: An Optimistic Approach for Byzantine Fault Tolerance in Software Transactional Memory. SBESC

2013.

- Ribeiro, Tulio Alberton; Lung, Lau Cheuk; Netto, Hylson Vescovi. Lidando com Transações Interativas em um STM Tolerante a Faltas Bizantinas. SBRC 2014.

# REFERÊNCIAS

AVIZIENIS, A.; KELLY, J. P. J. Fault tolerance by design diversity: Concepts and experiments. **Computer**, v. 17, n. 8, p. 67–80, 1984.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **Dependable and Secure Computing, IEEE Transactions on**, IEEE, v. 1, n. 1, p. 11–33, 2004.

BAZZI, R. A.; DING, Y. Non-skipping timestamps for byzantine data storage systems. In: **Distributed Computing**. [S.l.]: Springer, 2004. p. 405–419.

BERNSTEIN, P. A.; GOODMAN, N. Serializability theory for replicated databases. **Journal of Computer and System Sciences**, Elsevier, v. 31, n. 3, p. 355–374, 1985.

BESSANI, A.; SOUSA, J.; ALCHIERI, E. State machine replication for the masses with bft-smart. **DI/FCUL, Tech. Rep**, 2013.

BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. **Communications of the ACM**, v. 13, p. 422–426, 1970.

BOCCHINO, R.; ADVE, V.; CHAMBERLAIN, B. Software transactional memory for large scale clusters. **Proceedings of the 13th ACM . . .**, n. 3, p. 247–257, 2008.

CACHOPO, J.; RITO-SILVA, A. Versioned boxes as the basis for memory transactions. **Science of Computer Programming**, 2006.

CARVALHO, N.; ROMANO, P.; RODRIGUES, L. Scert: Speculative certification in replicated software transactional memories. In: ACM. **Proceedings of the 4th Annual International Conference on Systems and Storage**. [S.l.], 2011. p. 10.

CASTRO, M.; LISKOV, B. Practical Byzantine fault tolerance. **Operating Systems Review**, n. February, p. 1–14, 1998.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 20, n. 4, p. 398–461, 2002.

CORREIA, M. et al. Practical Hardening of Crash-Tolerant Systems. **USENIX Annual Technical Conference**, p. 453–466, 2012.

CORREIA, M.; NEVES, N.; VERISSIMO, P. How to tolerate half less one Byzantine nodes in practical distributed systems. In: **Proceedings of the Symposium on Reliable Distributed Systems**. [S.l.: s.n.], 2004. p. 174–183.

CORREIA, M.; VERONESE, G. S.; LUNG, L. C. Asynchronous byzantine consensus with 2f+ 1 processes. In: ACM. **Proceedings of the 2010 ACM Symposium on Applied Computing**. [S.l.], 2010. p. 475–480.

COSTA, P. et al. On the Performance of Byzantine Fault-Tolerant MapReduce. **IEEE Transactions on Dependable and Secure Computing**, v. 10, n. 5, p. 301–313, set. 2013. ISSN 1545-5971.

COUCEIRO, M.; ROMANO, P. D2STM: Dependable distributed software transactional memory. **Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on. IEEE, 2009.**, 2009.

COULOURIS, G. et al. **Distributed Systems: Concepts and Design, 5/e**. [S.l.: s.n.], 2012. ISBN 9780132143011.

COWLING, J.; LISKOV, B. Granola: low-overhead distributed transaction coordination. In: USENIX ASSOCIATION. **Proceedings of the 2012 USENIX conference on Annual Technical Conference**. [S.l.], 2012. p. 21–21.

DEFAGO, X.; SCHIPER, A.; URBAN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys (CSUR)**, v. 36, n. 4, p. 372–421, 2004.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM (JACM)**, ACM, v. 35, n. 2, p. 288–323, 1988.

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Toward a theory of transactional contention managers. **Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '05**, ACM Press, New York, New York, USA, p. 258, 2005.

GUERRAOUI, R.; RODRIGUES, L. **Reliable Distributed Programming**. [S.l.]: Springer, 2006.

HADZILACOS, V.; TOUEG, S. A modular approach to fault-tolerant broadcasts and related problems. p. 1–84, 1994.

HARRIS, T. et al. TRANSACTIONAL MEMORY: AN OVERVIEW. p. 8–29, 2007.

HERLIHY, M. Wait-free synchronization. **ACM Transactions on Programming Languages and ...**, v. 11, n. 1, p. 124–149, 1991.

HERLIHY, M.; LUCHANGCO, V. Software transactional memory for dynamic-sized data structures. **Proceedings of the twenty- ...**, 2003.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M. Obstruction-free synchronization: Double-ended queues as an example. **... Computing Systems, 2003. ...**, 2003.

HERLIHY, M.; MOSS, J. **Transactional memory: Architectural support for lock-free data structures**. [S.l.]: IEEE Comput. Soc. Press, 1993. 289–300 p. ISBN 0-8186-3810-9.

III, W. S.; SCOTT, M. Contention management in dynamic software transactional memory. **PODC Workshop on Concurrency ...**, 2004.

KEMME, B. et al. Using optimistic atomic broadcast in transaction processing systems. **Knowledge and Data Engineering, IEEE Transactions on**, IEEE, n. 4, 2003.

KOTLA, R. et al. Zyzzyva: speculative byzantine fault tolerance. In: **Proceedings of ACM SIGOPS Symposium on Operating Systems Principles**. [S.l.: s.n.], 2007.

KOTLA, R.; DAHLIN, M. High throughput Byzantine fault tolerance. **International Conference on Dependable Systems and Networks, 2004**, Ieee, p. 575–584, 2004.

KOTSELIDIS, C. et al. DiSTM: A Software Transactional Memory Framework for Clusters. **37th International Conference on Parallel Processing**, Ieee, p. 51–58, set. 2008.

KOTSELIDIS, C.; LUJáN, M. Clustering JVMs with software transactional memory support. **Parallel & Distributed . . .**, 2010.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, 1978.

LAMPORT, L. The part-time parliament. **ACM Transactions on Computer Systems (TOCS)**, ACM, v. 16, n. 2, p. 133–169, 1998.

LAMPORT, L.; FISCHER, M. **Byzantine Generals and Transaction Commit Protocols**. [S.l.], 1982.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 4, n. 3, p. 382–401, 1982.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The Byzantine Generals Problem. **ACM Transactions on Programming Languages and Systems**, v. 4, n. 3, p. 382–401, jul. 1982. ISSN 01640925.

LARUS, J. R. **Transactional Memory**. [S.l.: s.n.], 2006. 1–226 p. ISSN 1935-3235. ISBN 9781598291247.

LI, K.; HUDAK, P. Memory coherence in shared virtual memory systems. **ACM Transactions on Computer Systems (TOCS)**, v. 7, n. 4, p. 321–359, 1989.

LUIZ, A. F.; LUNG, L. C.; CORREIA, M. Byzantine fault-tolerant transaction processing for replicated databases. In: IEEE. **Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on**. [S.l.], 2011. p. 83–90.

LUIZ, A. F.; LUNG, L. C.; RECH, L. d. O. On the practicality to implement byzantine fault tolerant services based on tuple space. In: **Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications**. Washington, DC, USA: IEEE Computer Society, 2014. (AINA '14), p. 1041–1048.

MANASSIEV, K.; MIHAILESCU, M.; AMZA, C. Exploiting distributed version concurrency in a transactional memory cluster. **Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '06**, ACM Press, New York, New York, USA, p. 198, 2006.

MARATHE, V.; SCOTT, M. A qualitative survey of modern software transactional memory systems. **University of Rochester Computer Science . . .**, p. 1–20, 2004.

MASSALIN, H.; PU, C. A lock-free multiprocessor os kernel. **ACM SIGOPS Operating Systems Review**, v. 26, n. 2, p. 108, 1992.

NAEHRIG, M.; LAUTER, K.; VAIKUNTANATHAN, V. Can homomorphic encryption be practical? **Proceedings of the 3rd ACM workshop on Cloud computing security workshop - CCSW '11**, ACM Press, New York, New York, USA, p. 113, 2011.

NIGHTINGALE, E. B.; DOUCEUR, J. R.; ORGOVAN, V. Cycles , Cells and Platters : An Empirical Analysis of Hardware Failures on a Million Consumer PCs. 2009.

PALMIERI, R.; QUAGLIA, F.; ROMANO, P. AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing. **2010 Ninth IEEE International Symposium on Network Computing and Applications**, Ieee, p. 20–27, jul. 2010.

PALMIERI, R.; QUAGLIA, F.; ROMANO, P. OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems. **SRDS**, Ieee, n. 257784, p. 59–64, out. 2011.

PEDONE, F.; GUERRAOUI, R.; SCHIPER, A. The database state machine approach. **Distributed and Parallel Databases**, 2003.

PEDONE, F.; SCHIPER, A. Optimistic Atomic Broadcast. **Distributed Computing. Springer Berlin Heidelberg**, n. 95, p. 318–332, 1998.

PELUSO, S. et al. SPECULA: Speculative Replication of Software Transactional Memory. **2012 IEEE 31st Symposium on Reliable Distributed Systems**, Ieee, p. 91–100, 2012.

PROTIC, J.; TOMASEVIC, M.; MILUTINOVIC, V. Distributed shared memory: Concepts and systems. **Parallel & Distributed Technology: Systems & Applications, IEEE**, IEEE, v. 4, n. 2, p. 63–71, 1996.

RIBEIRO, T. A.; LUNG, L. C.; NETTO, H. V. OB-STM: An Optimistic Approach for Byzantine Fault Tolerance in Software Transactional Memory. **Simpósio Brasileiro de Engenharia de Sistemas Computacionais - SBESC**, 2013.

RIEGEL, T.; FELBER, P.; FETZER, C. A lazy snapshot algorithm with eager validation. In: **Distributed Computing**. [S.l.]: Springer, 2006. p. 284–298.

ROMANO, P. et al. An Optimal Speculative Transactional Replication Protocol. **International Symposium on Parallel and Distributed Processing with Applications**, Ieee, p. 449–457, set. 2010.

SAAD, M.; RAVINDRAN, B. RMI-DSTM : Control Flow Distributed Software Transactional Memory [ Technical Report ]. 2011.

SCHLICHTING, R. D.; SCHNEIDER, F. B. Fail-stop processors: an approach to designing fault-tolerant computing systems. **ACM Transactions on Computer Systems**, v. 1, n. 3, p. 222–238, ago. 1983. ISSN 07342071.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. **ACM Computing Surveys (CSUR)**, ACM Press New York, NY, USA, v. 22, n. 4, p. 299–319, 1990.

SCHROEDER, B.; GIBSON, G. a. Understanding failures in petascale computers. **Journal of Physics: Conference Series**, v. 78, p. 012022, jul. 2007. ISSN 1742-6588.

SCHROEDER, B.; PINHEIRO, E.; WEBER, W.-D. Dram errors in the wild: a large-scale field study. In: ACM. **ACM SIGMETRICS Performance Evaluation Review**. [S.l.], 2009. v. 37, n. 1, p. 193–204.

SCIASCIA, D.; PEDONE, F. RAM-DUR: In-Memory Deferred Update Replication. **2012 IEEE 31st Symposium on Reliable Distributed Systems**, Ieee, p. 81–90, 2012.

SHAVIT, N.; TOUITOU, D. Software transactional memory. **Proceedings of the fourteenth annual ACM ...**, p. 204–213, 1995.

SPEAR, M. F. et al. A comprehensive strategy for contention management in software transactional memory. **ACM SIGPLAN Notices**, v. 44, p. 141, 2009. ISSN 03621340.

STUMM, V. et al. Intrusion tolerant services through virtualization: A shared memory approach. In: **Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on**. [S.l.: s.n.], 2010. p. 768–774. ISSN 1550-445X.

ZHANG, H.; ZHAO, W. Concurrent Byzantine Fault Tolerance for Software-Transaction-Memory Based Applications. **ijfcc.org**, v. 1, n. 1, p. 1–4, 2012.