UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GISELE PINHEIRO SOUZA

Tuplebiz: Um espaço de tuplas distribuído e com suporte a transações resilientes a falhas bizantinas

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, fevereiro de 2012.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Souza, Gisele Pinheiro

Tuplebiz: Um Espaço de Tuplas Distribuído Resiliente a Falhas Bizantinas [manuscrito] / Gisele Pinheiro Souza. – 2012.

82 f.:il.

Orientador: Cláudio Fernando Resin Geyer.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 20012.

1.Espaço de Tuplas. 2. Falhas Bizantinas 3.Transação. I. Geyer, Claúdio Fernando Resin. II. Título

AGRADECIMENTOS

Agradeço ao meu orientador, Cláudio Geyer, pelo suporte prestado nesses anos de trabalho. Agradeço também a turma da sala 205 da UFRGS pelas ideias e por disponibilizar as máquinas para validar minha dissertação.

Agradeço também ao Terje Iversen, meu chefe, que foi compreensível e me disponibilizou um dos itens mais importante para completar a dissertação: tempo.

Agradeço à minha família e ao meu namorado, Eduardo Castro. O apoio por eles prestado foi essencial. Muito obrigada.

E por fim, agradeço a Deus por me dar perseverança.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, consistência, isolamento e durabilidade
API	Interface de Programação de Aplicativos
CIM	Modelo independente de computação
DSL	Linguagem específica de domínio
EAI	Integração de aplicações corporativas
ET	Espaço de Tuplas
GT	Gerente de Transações
HTTP	Protocolo de Transferência de Hipertexto
IDE	Integrated Development Environment
MDA	Arquitetura dirigida a modelos
MOM	*Middleware* Orientado a Mensagens
OMG	*Object Management Group*
PIM	Modelo independente de plataforma
PSM	Modelo dependente de plataforma
RPC	Chamada remota de procedimento
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Os modelos de coordenação de comunicação possibilitam a cooperação entre os diversos processos que fazem parte de um sistema distribuído. O modelo de coordenação de espaço de dados compartilhado, o qual é representado pelo espaço de tuplas, permite que a comunicação tenha tanto desacoplamento referencial quanto temporal. Devido essas características, o espaço de tuplas é frequentemente usado em aplicações pervasivas e paralelas. A habilidade de tolerar a falhas é importante para ambos os tipos de aplicações. Para aplicações pervasivas na área médica, uma falha pode custar vidas. Nesse contexto, esse trabalho propõe o Tuplebiz, um espaço de tuplas distribuído que suporta transações em um ambiente sujeito a falhas bizantinas. As falhas bizantinas encapsulam uma variedade de comportamentos faltosos que podem ocorrer no sistema.

O Tuplebiz é dividido em partições de dados para facilitar a distribuição entre diferentes servidores. Cada partição garante tolerância a falhas por meio de replicação de máquina de estados. Adicionalmente, o Tuplebiz também provê transações que possuem as propriedades ACID, isto é, as propriedades de atomicidade, consistência, isolamento e durabilidade. O gerente de transações é responsável por garantir o isolamento das transações.

Testes de desempenho e injeção de falhas foram realizados. A latência do Tuplebiz sem falhas é aproximadamente 2,8 vezes maior que a latência de um sistema não replicado. Os testes de injeção tiveram como base um *framework* de testes de injeção de falhas para sistemas tolerantes a falhas bizantinas. Os testes avaliaram os seguintes tipos de falha: mensagens perdidas, atrasos de envio de mensagens, corrupção de mensagens, suspensão do sistema e *crash*. A latência no caso de falhas foi maior que no caso sem falhas, mas todas as falhas foram suportadas pelo Tuplebiz.

Como estudo de caso, é revisada a integração do Tuplebiz com a Guaraná, uma linguagem específica de domínio usada para modelar soluções de integração de sistemas. As tarefas de uma solução de integração na Guaraná são centralizadas atualmente. A proposta de integração prevê a distribuição das tarefas entre diferentes servidores.

Palavras-Chaves: espaço de tuplas, falha bizantina, transação.

Tuplebiz: a distributed Tuple space Resilient to Byzantine Faults

ABSTRACT

The coordination models enable the communication among the process in a distributed system. The shared data model is time and referential decoupled, which is represented by tuple spaces. For this reason, the tuple space is used by parallel and pervasive applications. The fault tolerance is very important for both type of application. For healthcare applications, the fault can cost a life. In this context, this work introduces the Tuplebiz, a distributed tuple space that supports transactions in environment where byzantine faults can occur. Byzantine faults include many types of system faults.

The Tuplebiz is spitted in partitions. The main idea behind it is to distribute the tuple space among servers. Each partition guarantees the fault tolerance by using state machine replication. Furthermore, Tuplebiz has transaction support, which follows the ACID properties (atomicity, consistency, isolation, durability). The transaction manager is responsible for maintaining the isolation.

Performance and fault injection tests were made in order to evaluate the Tuplebiz. The Tuplebiz latency is approximately 2.8 times bigger than the one for a non replicated system. The injection tests were based on an injection fault framework for byzantine faults. The tests applied were: lost message, delay message, corrupted message, system suspension and crash. The latency was worst on those cases; however the Tuplebiz was able to deal with all of them.

Also, a case is presented. This case shows the integration between Tuplebiz and Guaraná, which is a domain specific language, used for designing Enterprise Application Integration applications. The solution integration tasks are centralized nowadays. The integration approach aims to distribute the tasks among servers.

Keywords: tuple space, byzantine fault, transaction.

1 INTRODUÇÃO

Esse trabalho apresenta o Tuplebiz, o qual é um espaço de tuplas distribuído que possui suporte a transações num ambiente sujeito a falhas bizantinas. A união de espaço de tuplas distribuído com suporte a falhas bizantinas e transações é um problema ainda não tratado na literatura atual.

Tuplebiz pode ser usado como *middleware* em diferentes tipos de aplicações, tais como, aplicações pervasivas e paralelas. O modelo de transação proposto para o Tuplebiz garante as propriedades ACID, isto é, as propriedades de atomicidade, consistência, isolamento e durabilidade.

Como estudo de caso foi desenvolvido um modelo da integração do Tuplebiz com a Guaraná, uma linguagem específica de domínio para integração de aplicações desenvolvida pela Universidade de Sevilha. As vantagens trazidas nessa integração são a possibilidade de diminuir a granularidade de distribuição de soluções desenvolvidas para a Guaraná e o aumento de confiabilidade da Guaraná como um todo.

1.1 Motivação

A coordenação de comunicação consiste em um sistema que manipula a comunicação e a cooperação entre diversos processos envolvidos (SOUZA, 2009). Os modelos de coordenação são classificados em três grandes grupos: Modelos de Coordenação Direta, Modelos de Coordenação Baseados em Eventos e Modelo de Coordenação de Espaço de Dados Compartilhados (MAMEI apud SOUZA, 2009, p.21). Dentre esses grupos apresentados, o de coordenação de espaços de dados compartilhados é o único que apresenta desacoplamento temporal e referencial. O modelo de coordenação de espaços de dados compartilhados é representado pelo espaço de tuplas (ET), o qual se baseia no modelo Linda (CARRIERO; GELERNTER, 1992). Outros autores (BAKKEN, 2011) e (FUMMI ET. AL., 2007) classificam o espaço de tuplas como um *middleware*. Nesse caso, os *middlewares* orientados a mensagens (MOM) também garantem desacoplamento referencial e temporal. Porém, os MOMs apresentam limitações quanto ao armazenamento (BAKKEN, 2011).

As características de desacoplamento temporal e referencial permitem que diferentes tipos de aplicações façam uso do espaço de tuplas como modelo de coordenação. As aplicações que utilizam espaço de tuplas mais frequentemente são *middlewares* para computação pervasiva (MAMEI; ZAMBONELLI; LEONARDI, 2003), (HENRICKSEN; ROBINSON, 2006), (COSTA ET. AL., 2006), (CERIOTTI ET.AL; 2009), (MATTHEWS; CHALMERS; WAKEMAN, 2011) e aplicações paralelas (ATKINSON, 2008), (SCHUMACHER; SHULLER, 2009) e (GHASEMIGOL ET. AL., 2009).

Para os tipos de aplicações aqui enumerados, o tratamento de falhas se faz necessário. Falhas em *middlewares* para computação pervasiva podem levar à detecção incorreta do contexto, falhas de segurança, privacidade e mal uso de recursos (CHETAN; RANGANATHAN; CAMPBELL, 2005). Adicionalmente, esse tipo de sistema vem sendo usado para aplicações médicas e monitoramento de idosos, o que pode custar vidas em caso de falha (CHETAN; RANGANATHAN; CAMPBELL, 2005). Aplicações paralelas, por sua vez, também estão sujeitas a falhas (ATKINSON, 2010). Existe um esforço para aumentar a dependabilidade desse tipo de aplicação.

Tendo em vista os diferentes ambientes onde o espaço de tuplas pode ser utilizado, aumentar a dependabilidade é crucial para um correto funcionamento da aplicação. Dentre os diferentes tipos de falhas que podem ser tratadas, as falhas bizantinas são as mais gerais. Falha bizantina é um conceito que captura uma grande variedade de "outros" comportamentos faltosos, os quais incluem corrupção de dados, programas que não seguem o protocolo correto, e até mesmo, comportamentos maliciosos que buscam violar a confiabilidade do sistema (BIRMAM, 1996). Logo, ao tratar falhas bizantinas, tratam-se falhas de *crash*, omissão, tempo, entre outras.

Ainda com o intuito de acrescer a confiabilidade do sistema, o uso de transações se mostra bastante eficiente. O termo transação se refere a uma coleção de operações que formam uma única unidade lógica de trabalho (SILBERSCHATZ; KORTH; SUDARSHAN, 2006). Desse modo, garante-se que um bloco de operações irá ser inteiramente executado ou nenhuma das operações será executada. (PERICH ET AL., 2003) resalta a importância do uso de transações em *middlewares* pervasivos e propõe um gerenciador de transações para esse tipo de ambiente. (SATYANARAYANAN, 2001) enumera transação como um problema que deve ser tratado tanto em sistemas distribuídos em geral quanto em sistemas pervasivos.

Nesse cenário, faz-se necessário definir um espaço de tuplas que suporte transações num ambiente sujeito a falhas bizantinas. Atualmente na literatura, não foram difundidos trabalhos nesse sentido. (BESSANI, 2006) apresenta espaços de tuplas que suportam falhas bizantinas e indica o uso de transações nesse tipo de aplicação como trabalho futuro. Gigaspace (GIGASPACE, 2011), Javaspace (SUN MICROSYSTEMS, 1999) e TSpaces (LEHMAN; MCLAUGHRY; WYCKO, 1999) apresentam suporte a transações, mas não possuem outras técnicas de tolerância falhas. PLinda 2.0 (JEONG; SHASHA, 1994) suporta transação, mas suporta apenas falhas de *crash*. (VANDIVER; BALAKRISHNAN; LISKOV, 2007) e (LUIZ; LUNG; CORREIA, 2011) apresentam suporte a transações e a falhas bizantinas, mas no contexto de banco de dados.

Paralelamente a tolerância a falhas, o espaço de tuplas deve suportar distribuição, uma vez que as aplicações que farão uso desse espaço de tuplas são distribuídas. Existem diversos trabalhos que versam sobre espaços de tuplas e distribuição, entre eles (COSTA ET. AL, 2006), (PICCO; MURPHY; ROMAN, 1999), (CHARLES; MENEZES; TOLKSDORF, 2004) e (SARIGÖL; RIVA; ALONSO, 2010).

Como estudo de caso é apresentado a integração do Tuplebiz com a Guaraná. (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010). Essa é uma linguagem específica de domínio que possui um motor de execução orientado a mensagens. O intuito da Guaraná é modelar e executar aplicações de integração.

1.2 Objetivos da dissertação

O objetivo principal desse trabalho é definir um espaço de tuplas distribuído que possua suporte a falhas bizantinas e transações. As transações devem garantir as propriedades ACID, isto é, as propriedades de atomicidade, consistência, isolamento e durabilidade. O nível de isolamento deve ser pelo menos similar ao encontrado na literatura.

Outro objetivo do trabalho é apresentar um estudo de caso, o qual é modelar o *middleware* de comunicação do motor de execução da Guaraná. Hoje o motor de execução da Guaraná usa filas locais em memória para comunicação entre tarefas. A integração do espaço de tuplas nesse motor de execução muda o paradigma de funcionamento do mesmo.

1.3 Contribuições da dissertação

Em resumo, as principais contribuições desse trabalho são:

- A definição de um espaço de tuplas tolerante a falhas bizantinas e que suporte transações;
- Definição de um espaço de tuplas particionado;
- Definir um acesso transparente às partições do espaço de tuplas;
- O uso de espaços de tuplas como *middleware* para a Guaraná.

1.4 Organização da dissertação

Esse trabalho possui seis capítulos, os quais apresentam análise bibliográfica do tema e as etapas realizadas para cumprir os objetivos apresentados nessa dissertação.

O capítulo 2 dá uma visão geral do que existe hoje em relação à tolerância a falhas bizantinas e mecanismos de tolerância para os espaços de tuplas. Primeiramente são apresentados conceitos básicos de tolerância a falhas. Logo após são apresentados espaços de tuplas que possuem mecanismos de tolerância a falhas hoje na literatura. Espaços de tuplas distribuídos também são apresentados.

O capítulo 3 disserta sobre o Tuplebiz, isto é, mostra a arquitetura do sistema e como as questões de tolerância a falhas são endereçadas. Nesse capítulo estão as definições do protocolo de transação, o modo de endereçar o não determinismo inerente ao espaço de tuplas e as demais definições.

O capítulo 4 descreve o protótipo, a metodologia de experimentos e resultados dos mesmos. Nesse capítulo são apresentados detalhes do protótipo como a linguagem de programação utilizada, as ferramentas auxiliares e as bibliotecas. Além disso, há uma comparação com os trabalhos existentes.

O capítulo 5 mostra o funcionamento do motor de execução da Guaraná , bem como, as técnicas atuais de monitoramento a falhas existentes na Guaraná. Também, é apresentada a integração com a Guaraná. Esse capítulo mostra como é possível tornar o motor de execução da Guaraná distribuído ao usar o Tuplebiz.

O capítulo 6 é a conclusão. Nesse capítulo são revisadas as contribuições do Tuplebiz. Além disso, os trabalhos futuros são enumerados.

2 CONCEITOS BÁSICOS

Esse capítulo versa sobre estudos que endereçam tolerância a falhas e distribuição nos espaço de tuplas. Tendo em vista as falhas que podem ocorrer, este capítulo tem na sua primeira sessão uma definição dos tipos de falhas. Os diferentes tipos de falhas são definidos, mas a ênfase maior é dada para falhas bizantinas, as quais são o cerne desse trabalho.

Primeiramente, é realizada uma revisão do modelo Linda, o qual introduziu o conceito de espaço de tuplas. Existem diversos trabalhos que visam aumentar a dependabilidade de espaço de tuplas. Esses trabalhos abordam diferentes tipos de falhas e possuem soluções distintas para aumentar a dependabilidade. Uma comparação almejando descrever as distintas soluções é feita.

Considerando as soluções de distribuição para espaços de tuplas, este capítulo mostra os espaços de tuplas que possuem algum suporte à distribuição. Interessante salientar que o conjunto de espaços de tuplas que possuem algum tipo de tolerância a falhas não é distribuído.

2.1 Tipos de falhas em sistemas distribuídos

Antes de falar de tipos de falhas, é necessário introduzir os conceitos relativos à falha, ao erro e ao defeito. Define-se como falha uma causa física ou algorítmica que leva ao erro. O erro acontece quando o processamento posterior a ele gera um defeito. Entende-se como defeito o desvio da especificação. A Figura 2.1 mostra a relação causal entre falhas, erro e defeito (WEBER, 2002). Como exemplo dessa relação pode-se usar uma falha na memória que mude o estado de um bit de 1 para 0. Essa falha gera um erro na aplicação de gerenciamento de e-mail que por sua vez se manifesta como um defeito para o usuário, o qual não pode ler suas mensagens da caixa de entrada.

Figura 2.1: Relação entre falha, erro e defeito

Os modelos de falhas em sistemas distribuídos podem ser vistos na Figura 2.2. O diagrama indica a abrangência de cada tipo de falha. Por exemplo, falha por omissão inclui falhas por *crash*. Segue a definição de cada um dos tipos de falhas enumerados na figura.

- ***Crash***: se caracteriza pela parada do sistema ou perda do seu estado interno;
- **Omissão:** se caracteriza pela falta de resposta para algumas entradas;

- **Temporização**: se caracteriza pela resposta adiantada, ou tardia, a uma entrada;
- **Resposta:** se caracteriza por valores incorretos de saída devido à execução errada de algumas entradas;
- **Bizantina**: também conhecida como arbitrária, é um conceito que captura uma grande variedade de "outros" comportamentos faltosos, os quais incluem corrupção de dados, programas que não seguem o protocolo correto, e até mesmo, comportamentos maliciosos que buscam violar a confiabilidade do sistema (BIRMAM, 1996). A falha bizantina engloba um grande conjunto de falhas.

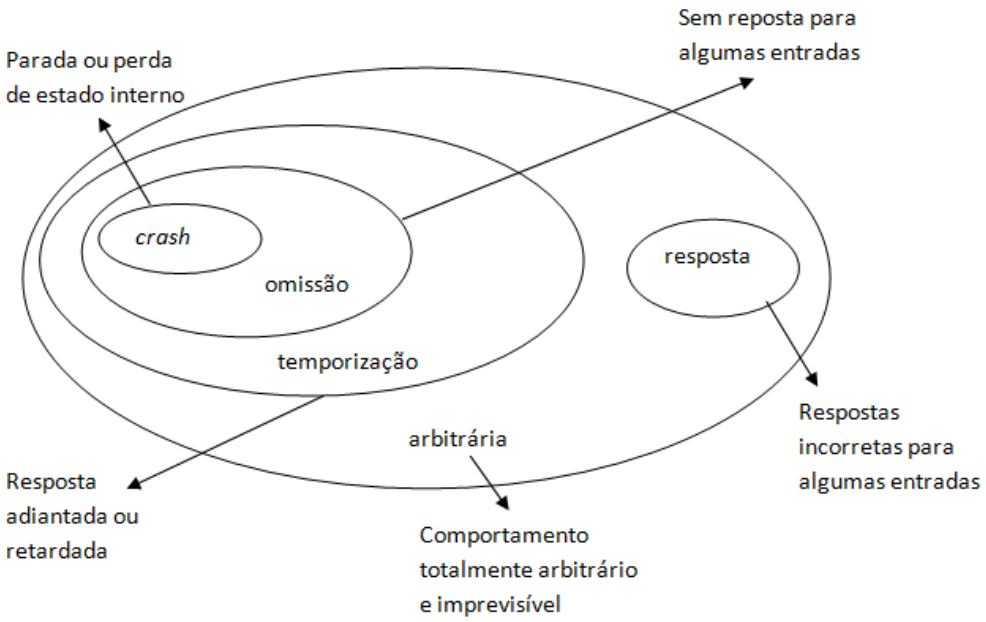


Figura 2.2: Modelo de falhas em sistemas distribuídos retirado de (WEBER, 2002)

2.2 Técnicas de tolerância a falhas bizantinas

Tolerância a falhas é a capacidade de um sistema fornecer o serviço esperado mesmo na presença de falhas O objetivo de tolerância a falhas é alcançar dependabilidade, a qual indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido (WEBER, 2002).

Todas as técnicas de tolerância a falhas envolvem alguma forma de redundância (WEBER, 2002). A replicação baseada em *software* vem sendo estudada como uma solução barata de tolerância a falhas se comparada à replicação de *hardware* (GUERRAOUI; SCHIPER, 1997).

O modelo de replicação mais comum para tolerância a falhas em sistemas sujeitos a falhas bizantinas é a replicação por máquina de estados (LAMPORT apud BESSANI, 2006, p. 23). Existem três maneiras de tratar falhas bizantinas utilizando máquina de estado (COWLING ET AL., 2006):

- Máquina de estado baseada em réplicas;
- Máquina de estado baseada em quórum;

- Híbrido que se baseia em réplicas e quórum.

A abordagem de máquina de estado baseada em réplicas usa a comunicação entre as réplicas para acordar a ordem de requisição. Nessa abordagem o número mínimo teórico de réplicas é visto na equação (2.1), onde *f* o número de nodos falhos suportados (COWLING ET. AL, 2006). Os trabalhos de (CASTRO; LISKOV, 2002) e (KOTLA ET. AL., 2009) utilizam esse tipo de abordagem.

$$n = 3f + 1 \qquad (2.1)$$

Na abordagem máquina de estado baseada em quórum, os clientes se comunicam diretamente com as réplicas para executar as operações de forma otimista. Essa abordagem tem como número mínimo de réplicas a equação (2.2). Essa abordagem é usada em (ABD-EL-MALEK ET AL., 2005).

$$n = 5f + 1 \qquad (2.2)$$

O modelo híbrido se baseia em réplicas e quórum faz uso do quórum quando não há controvérsia e usa réplicas para quando há controvérsia. Essa abordagem, assim como a baseada em réplicas, tem o número mínimo teórico de réplicas a equação (2.1). Essa abordagem é usada em (COWLING ET. AL., 2006).

2.3 Modelo Linda (Espaço de tuplas)

Linda é um modelo de memória (CARRIERO; GELERNTER, 1992), no qual a memória se chama espaço de tuplas. É importante salientar que Linda é um modelo e não uma ferramenta, logo se pode ter as mais diversas implementações, dependendo em qual contexto o modelo será usado. Esse modelo define quatro operações básicas: out, in, rd e eval, e mais duas formas variantes inp e rdp.

A operação out(t) insere o elemento *t* no espaço de tuplas. O processo em execução continua imediatamente (CARRIERO; GELERNTER, 1992). Por exemplo, a operação out(“linda string”, 2, “outro texto”, 5.3) está inserindo uma tupla com quatro campos, onde o primeiro e o terceiro são uma cadeia de caracteres, o segundo um número inteiro e o quarto um número real.

A operação in(s) busca e removida uma tupla *t* que case com a anti-tupla *s*. Uma anti-tupla é uma série de tipos de campos; alguns são valores (ou verdadeiros), outros são tipos com valores não definidos (ou formais) (CARRIERO; GELERNTER, 1992). Valores formais possuem um símbolo de exclamação no início. Essa operação é bloqueante, isto é, o processo fica esperando até haver uma tupla que coincida com a anti-tupla *s*. Se houver mais de uma tupla que coincida com *s*, qualquer uma será retornada de forma arbitrária. Por exemplo, in(“linda string”, ?x, ?y, 5.3) contém dois campos verdadeiros “linda string” e 5.3 e, dois campos formais x e y. Nesse exemplo, as seguintes tuplas coincidem com a anti-tupla e qualquer uma delas do espaço de tuplas: (“linda string”, 2, “outro texto”, 5.3) ou (“linda string”, “algum texto”, 6.8, 5.3).

A operação rd(s) é similar à operação in(s), o que as difere é que a rd(s) não retira a tupla do espaço de tuplas. As operações rdp e inp são similares às correspondentes rd e in, entretanto retornam 0 se falham, senão retornam 1. Essas são versões não bloqueantes das operações rd e in.

A operação eval(t) funciona de forma similar a out(t) se distinguindo pelo fato da operação eval(t) ser avaliada somente após a inserção. Por exemplo, a operação

eval(“Linda String”, 5, sqrt(9)) vai criar 3 processos que vão avaliar cada um dos campos da tupla, o primeiro e o segundo são triviais, mas o terceiro exige um processamento (ele executará a raiz quadrada de 9). Logo após ser processada, a tupla pode ser buscada por uma operação de in ou rd. Seguindo o exemplo, a operação rd(“Linda String”, 5, 3) retornaria a tupla inserida na operação eval.

O espaço de tuplas pode ser utilizado em diferentes contextos. (LEHMAN; MCLAUGHRY; WYCKO, 1999) cita o espaço de tuplas como um mecanismo robusto e confiável para aplicações paralelas e distribuídas. Isso se deve ao fato de vários agentes poderem trabalhar em conjunto em uma tarefa.

Sistemas baseados no modelo Linda foram implementados em um expressivo número de plataformas (NIXON ET AL., 2008). O espaço de tuplas se mostrou uma ótima ferramenta para paralelizar aplicações de alta granularidade (ATKINSON, 2008). O aparecimento de diferentes plataformas mostrou que o modelo Linda poderia ser estendido e suportar funcionalidades além das quais ele foi concebido (NIXON ET AL., 2008). Por exemplo, Tupleware (ATKINSON, 2008) foi desenvolvido para uso em aplicações paralelas, Lime (PICCO; MURPHY; ROMAN, 1999) e TeenyLime (COSTA ET AL., 2006) para redes de sensores e WCL (ROWSTRON, 1998) para sistemas geograficamente distribuídos com alta latência de banda.

2.4 Espaço de tuplas resilientes a falhas

Dentre essa gama de implementações disponíveis, as implementações relevantes para esse trabalho são as que apresentam algum suporte para tolerância a falhas ou distribuição. Dois aspectos importantes a serem observados, quando se trata de falhas, são o tipo de falha a ser tratada e a solução de tolerância a falhas adotada. Primeiramente podem ser enumerados espaços de tuplas em que a ideia principal é garantir que sejam executadas operações atômicas. Nesse grupo estão os espaços de tuplas com suporte a transações. Outro tipo de espaço de tuplas são aqueles que visam suportar falhas de *crash*. Um terceiro tipo seria o grupo dos espaços de tuplas que suporta falhas bizantinas.

Antes de falar sobre os espaços de tuplas com suporte à transações, uma revisão do conceito de transação se faz necessária. O termo transação se refere a uma coleção de operações que formam uma única unidade lógica de trabalho (SILBERSCHATZ; KORTH; SUDARSHAN, 2006). Uma transação segue as propriedades ACID, isto é, atomicidade, consistência, isolamento e durabilidade. Segue a definição das propriedades como descrito em (SILBERSCHATZ; KORTH; SUDARSHAN, 2006):

- **Atomicidade**: todas as ações da transação são executadas ou nenhuma delas é executada. O sistema não pode ficar em um estado onde a transação é parcialmente executada;
- **Consistência:** o sistema que inicialmente está consistente, após a execução da transação, deve permanecer consistente;
- **Isolamento:** uma transação pode executar concorrentemente a outra transação, no entanto, cada transação tem a impressão de ser a única a estar executando;
- **Durabilidade:** após ser realizada a confirmação de uma transação, os dados não são perdidos, mesmo que haja uma falha no sistema.

A propriedade de isolamento pode ser classificada em quatro tipos de acordo com ANSI/ISO SQL-92 (SIMON; MELTON, 1993): *read uncommitted*, *read committed*, *repetable read* e serializável. O nível de isolamento é definido por três fenômenos que podem ocorrer durante a transação. O primeiro deles é a leitura suja (*dirty read*) que acontece quando uma transação vê mudanças realizadas por outra transação antes delas realizarem uma confirmação. O segundo fenômeno é a leitura não repetível (*unrepeatable read*) que acontece quando uma transação lê um dado e quando ela tenta lê-lo novamente o dado foi mudado ou removido por outra transação. E por fim, o terceiro fenômeno é a leitura fantasma (*phantom read*), a qual ocorre quando uma transação lê um dado e quando ela tenta lê-lo novamente novas informações são adicionadas por outras transações. A Tabela 2.1 mostra a relação entre os fenômenos e os níveis de isolamento.

Tabela 2.1: Relação entre nível de isolamento e um fenômeno, adaptado de (SIMON; MELTON, 1993).

Nível de isolamento	**Leitura suja**	**Leitura não repetível**	**Leitura fantasma**
read uncommitted	possível	possível	possível
read committed	impossível	possível	possível
repetable read	impossível	impossível	possível
serializável	impossível	impossível	impossível

Outro conceito importante referente às transações distribuídas é o protocolo de *commit* em duas fases. Esse é o mais simples e popular protocolo de *commit* atômico (BERNSTEIN; HADZILACOS; GOODIMAN, 1987). O protocolo possui duas fases: a fase de votação e a fase de decisão. Na fase de votação o coordenador envia uma mensagem que requer o voto de *commit* para cada participante. Cada participante responde com o seu voto de sim ou não para o *commit* da transação. Na fase de decisão, se pelo menos um participante decide pelo aborto da transação (responde não), a transação deve ser abortada. Então, o coordenador envia para todos os participantes que votaram sim uma mensagem indicando que a transação deve ser confirmada ou abortada.

TSpaces (LEHMAN; MCLAUGHRY; WYCKO, 1999), Gigaspaces (GIGASPACE, 2011) e Javaspace (SUN MICROSYSTEMS, 1999) implementam a transação de forma similar. A transação é definida de forma explícita, ou seja, o desenvolvedor deve indicar o inicio e o final da transação. As premissas para garantir as propriedades da transação também são similares nessas ferramentas. A primeira premissa diz que as operações de out(t) são visíveis apenas quando as transações são finalizadas. A segunda premissa diz que a operação de rdp(s) é bloqueada para somente leitura dentro do contexto transacional. Uma vez que a transação executa a operação rdp(s), a tupla retornada fica disponível apenas para leitura. E por fim, a terceira premissa diz que a operação de inp(s) remove a tupla do Tuplebiz, entretanto se a transação é abortada, a tupla retorna ao espaço de tuplas. Apenas considerando essas premissas, o nível de isolamento da transação não é serializável (BUSI, 2001). Algumas modificações necessitam ser realizadas para que o nível seja serializável.

Para PLinda 2.0 (JEONG; SHASHA, 1994) toda operação é executada dentro de um contexto transacional. Não há um detalhamento na literatura de como o bloqueio das operações é realizado. Além do suporte à transação, o PLinda 2.0 possui um mecanismo de *checkpoint* para suportar falhas de *crash*.

Mobile Co-ordination (ROWSTRON, 1999) não possui suporte a transação propriamente dita. Não há isolamento entre transações. Essa ferramenta tem o conceito de execução de sacola de tarefas. O usuário do espaço de tuplas envia um saco de tarefas para o espaço. O espaço de tuplas executa cada operação da sacola e essa alteração já é visível para outras transações. A vantagem, dada pelo autor para essa abordagem, é evitar o *deadlock* e o suporte a falhas caso ocorra uma falha de comunicação entre o cliente e o espaço de tuplas.

Em (PATTERSON ET AL., 1993) é apresentado um espaço de tuplas que suporta falhas de *crash*. Réplicas são usadas para garantir a tolerância a falhas. Porções do espaço de tuplas, chamadas de subspaces, são replicadas. O número de réplicas depende do número de usuários do subspace. Mais precisamente, se houver dez usuários de um subspace, haverá dez réplicas de uma mesma tupla.

O FT-Linda (BAKKEN, 1995) tem como objetivo disponibilizar um espaço de tuplas estável e com operações atômicas. Para manter o espaço de tuplas estável é utilizada a técnica de replicação de máquina de estados. Para garantir a atomicidade, que nesse caso significa executar as operações no modelo tudo-ou-nada, são usadas operações atômicas ao invés de utilizar um protocolo de *commit* em duas fases.

O BTS (BESSANI, 2006) tem como objetivo suportar falhas bizantinas e não possui qualquer tratamento a transações. As operações suportadas são out(t), rdp(s) e inp(s). Cada uma das operações possui um algoritmo distinto.

No BTS, durante a operação de out(t), o cliente do espaço de tuplas envia uma mensagem para todas as réplicas com a operação de out(t) e não espera confirmação. Nesse caso, o desempenho é ótimo, pois só há um passo de comunicação.

Ainda no BTS, a operação de rdp(s) requer dois passos de comunicação. O cliente envia uma mensagem com a operação de rdp(t) para todas as réplicas e essas, por sua vez, enviam uma mensagem de resposta que contém uma lista com todas as tuplas que casam com a anti-tupla s.

A operação de inp(s) no BTS requer mais passos de comunicação. Nesse, o algoritmo de Paxos bizantino[1] apresentado em (CASTRO; LISKOV, 2002) é executado duas vezes. Primeiramente, o cliente requer o bloqueio do espaço de tuplas. Para tanto, é usado um algoritmo de exclusão mútua que utiliza Paxos. Uma vez que a exclusão mútua é garantida, o cliente executa uma operação de rdp(s) que retorna a tuplas t. Ao obter t, o algoritmo de Paxos é executado para remover a tupla do espaço de tuplas.

O algoritmo de LBTS (BESSANI, 2006) é linearizável, ou seja, as operações são confirmáveis. Esse comportamento é diferente do BTS, onde as operações não são confirmáveis. A operação de out(t) é similar ao do BTS. A única diferença é uma mensagem de ack recebida ao final da inserção da tupla.

A operação de rdp(s) no LBTS leva dois passos de comunicação para realizar a operação e no pior caso leva seis passos. Nessa operação, o cliente escuta as mudanças

[1] Algoritmo usado para suportar falhas bizantinas baseado em replicação de máquina de estados.

nas réplicas e é informado de cada mudança que ocorra. A operação só é confirmada quando o número de tuplas removidas é igual ao menos f + 1 réplicas.

A operação de inp(s) no LBTS leva quatro passos de comunicação para realizar a operação e no pior caso leva sete passos. Nessa operação o algoritmo de Paxos é modificado e o uso de exclusão mútua não é necessário.

2.4.1 Comparação entre os espaços de tuplas

Quando se fala de tolerância a falhas em espaço de tuplas, encontram-se normalmente trabalhos que abordam operações atômicas, tolerância a falhas de *crash* e tolerância a falhas bizantinas. As operações atômicas nem sempre satisfazem as propriedades das transações.

A Tabela 3.1 tem um resumo das características suportadas pelas diferentes implementações de espaço de tuplas apresentadas na seção anterior. Nota-se que apenas um terço desses espaços de tuplas apresentados tolera algum tipo de falha e suporta transações e, mesmo nesse grupo, apenas um possui transações que satisfazem as propriedades ACID.

Somente dois dos espaços de tuplas apresentados suportam a categoria mais geral de falha, que é a falha bizantina. Dentre desses, nenhum possui qualquer suporte a operações atômicas, apesar de se salientar a importância das mesmas em (BESSANI, 2006).

Tabela 2.2: Comparação de espaços de tuplas que possuem tolerância a falhas

Espaço de Tuplas	Tipo de falhas toleradas	Suporta transações
TSpaces	Não suporta falhas	Sim
Gigaspace	Não suporta falhas	Sim
Javaspace	Não suporta falhas	Sim
Mobile Co-ordination	Falha de comunicação	Sim[2]
(PATTERSON ET AL., 1993)	*Crash*	Não
Plinda 2.0	*Crash*	Sim
FT-Linda	*Crash*	Sim[2]
BTS	Bizantina	Não
LBTS	Bizantina	Não

2.5 Espaços de Tuplas distribuídos

Considerando as diferentes abordagens de implementação de um espaço de tuplas distribuído, existem algumas características peculiares a esse tipo de implementação, as quais se fazem importantes para comparar as diferentes propostas. Alguns pontos foram escolhidos a fim de salientar as contribuições de cada aplicação, os quais se encontram abaixo.

- **Distribuição das tuplas**: indica a maneira utilizada para organizar e distribuir tuplas entre os diferentes nodos do sistema;
- **Aplicação destino**: indica a aplicação para qual o espaço de tuplas foi desenvolvido;

[2] Não suporta todas as propriedades ACID.

- **Transparência no acesso**: indica a capacidade de adicionar e remover elementos no espaço de tuplas distribuído, sem, no entanto, indicar o nodo onde a operação será realizada. Por exemplo, se existe um espaço de tuplas local e um global, o desenvolvedor não precisa indicar seu desejo de acessar o espaço local ou global.

Jada (CIANCARINI; ROSSI, 1997) foi desenvolvido com o intuito de atender aplicações web em plataforma Java, podendo ser inserido em Jada applets (uma API similar ao Java Applet). Os espaços de tuplas são disjuntos, não havendo comunicação entre os diferentes espaços. O desenvolvedor deve indicar de forma explícita qual espaço de tupla deve ser acessado. Cada espaço de tuplas é um ObjectServer que é acessado pela rede através de um ObjectClient. Um espaço de tuplas no Jada pode ser explicitamente criado e manipulado por processos Jada.

WCL (ROWSTRON, 1998) foi desenvolvido para atender sistemas geograficamente distribuídos com alta latência de banda. Na sua arquitetura um servidor possui um espaço de tuplas inteiro, e as tuplas são migradas entre os servidores conforme a posição geográfica do servidor e do agente que está utilizando as tuplas. O controle das operações é feito de forma centralizada, por um componente, o qual tenta otimizar o desempenho movendo a tupla para o espaço mais próximo do usuário da mesma.

LIME (PICCO; MURPHY; ROMAN, 1999) tem como objetivo obter alta disponibilidade em ambientes pervasivos. O *host* nesse ambiente pode ser uma estação de trabalho estática ou um componente móvel. Seu modelo considera um espaço de tuplas local, ao invés de um global, formando uma federação de espaço de tuplas. O *host*, quando se move fisicamente, passa a fazer parte do espaço de tuplas da localidade de forma transparente. Essa aplicação considera o compartilhamento de tuplas dos *hosts* vizinhos. Possui uma operação chamada de *upon*, a qual permite que um cliente seja notificado quando é escrita no espaço de tuplas uma tupla que contenha as características desejadas.

SwarmLinda (CHARLES; MENEZES; TOLKSDORF, 2004) faz uso de inteligência coletiva para realizar busca no espaço de tuplas. O mecanismo empregado na implementação é o uso de uma colônia de formigas para realizar a busca em um sistema distribuído aberto. As tuplas comunicam-se entre si diretamente, entretanto a comunicação não é gerenciada pelo nodo, e sim através de "formigas". O sistema funciona da seguinte forma: a formiga recebe a tupla que deve ser armazenada e percorre os nodos até achar tuplas similares. Se dentro de certo tempo de expiração um nodo compatível com a tuplas a ser armazenada não é encontrado, então a formiga fica "cansada" e armazena a tupla no nodo no qual se encontra. A formiga deixa um sinal para as demais para encontrar o caminho.

Espaço de tuplas para redes sociais em redes ad-hoc (SARIGÖL; RIVA; ALONSO, 2010) é uma implementação do espaço de tuplas que visa prover uma solução para redes sociais em redes ad-hoc. Uma tupla para essa ferramenta possui alguns campos não usuais a um espaço de tuplas: escopo, tempo de vida e versão. O escopo indica por quantos nodos a tupla pode passar na rede. O tempo de vida indica quanto tempo a tupla deve ser armazenada localmente em um nodo. E, por fim, a versão que é usada na substituição da tupla no nodo, o qual deve sempre a versão mais atual. Essa implementação não faz replicação de todas as tuplas para garantir tolerância a falhas. A replicação que existe se deve ao campo de escopo, que indica quantos nodos podem

armazenar a tupla. Não possui o conceito de transação, logo a atomicidade, segundo o autor, é atingida associando o tempo de vida e a versão da tupla.

O Tupleware (ATKINSON, 2008) foi desenvolvido para suportar aplicações paralelas, onde vários acessos são feitos de modo simultâneo. Essa ferramenta usa uma tabela *hash* para armazenar as tuplas, tendo um custo da ordem de O(1) para acessar uma tupla. Para realizar a busca de uma tupla, o motor de execução, primeiramente, realiza a busca na tupla local, e se não encontra a tupla desejada, o motor executa um algoritmo de busca heurístico para encontrar a tupla em outros espaços de tuplas.

Quanto ao modelo de distribuição das tuplas, cada implementação tem seu próprio modo de organizar as tuplas. As ferramentas Jada, SwarmLinda, WCL, Tupleware e a de rede social permitem que todos os nodos vejam todos os espaços de tuplas disponíveis, entretanto somente algumas dessas ferramentas permitem que o acesso aos diferentes espaços de tuplas seja transparente, isto é, não é preciso que o usuário indique explicitamente de onde ele deseja recuperar/inserir a tupla. Já Lime permite que só os espaços de tuplas adjacentes sejam visualizados, mas sempre de forma transparente para o usuário da tupla. A Tabela 2.3 sintetiza essa comparação.

Tabela 2.3: Comparação entre espaços de tuplas distribuídos

Ferramenta	Modelo de Distribuição	Transparência de acesso (Porção do espaço de tuplas deve ser especificada explicitamente?)	Aplicação destino
Jada	Existem vários espaços de tuplas que são disjuntos e não se comunicam entre si.	NÃO	Internet em plataformas Java
Lime	As tuplas são federadas, um nodo tem visão apenas do espaço de tuplas dos nodos vizinhos.	SIM	Redes de Sensores
SwarmLinda	As tuplas são vistas por todos os nodos, e são organizadas por similaridade de forma dinâmica.	SIM	Redes de Sensores
WCL	As tuplas são migradas entre os espaços de tuplas de acordo com a localização dos usuários da tupla.	SIM	Ambientes geograficamente distribuídos
Tupleware	As tuplas são vistas por todos os nodos.	NÃO	Aplicações paralelas
Rede Social	As tuplas são vistas por todos os nodos, as tuplas são distribuídas entre os usuários das tuplas.	NÃO	Redes ad-hoc

2.6 Considerações finais

Apesar de tolerância a falhas ser uma preocupação em implementações de espaços de tuplas desde a década de noventa, poucos trabalhos versam sobre falhas bizantinas e espaços de tuplas. Nota-se que muitas soluções existentes unem o uso de transações em espaços de tuplas e suporte a falhas de *crash*.

A distribuição também é o foco para diferentes implementações de espaços de tuplas. Entretanto, é difícil encontrar aplicações que unam as duas abordagens: distribuição e tolerância a falhas.

Inserido nesse contexto, o próximo capítulo versa sobre o Tuplebiz, um espaço de tuplas distribuído que possui suporte a transações e falhas bizantinas. Essa união de distribuição, suporte a falhas bizantinas e transações em um espaço de tuplas é única ao conhecimento do autor na literatura até então.

3 TUPLEBIZ

O Tuplebiz é um espaço de tuplas distribuído que suporta transações num ambiente sujeito a falhas bizantinas. Esse conceito ainda é único na literatura.

Inicialmente uma visão geral do sistema é dada e é demonstrado como as propriedades de vivacidade (*liveness*) e segurança (*safety*) são satisfeitas. Adicionalmente, é vista a forma como o Tuplebiz endereça problemas inerentes ao tratamento de falhas bizantinas e transações.

Na sequência, a arquitetura do Tuplebiz é definida. Nessa seção são definidos os algoritmos implementados nas operações suportadas pelo Tuplebiz. O mecanismo usado pelo Tuplebiz para garantir a distribuição também é definido nesse capítulo.

3.1 Visão Geral

O Tuplebiz é um espaço de tuplas que suporta transações num ambiente sujeito a falhas bizantinas. O termo transação se refere uma coleção de operações que formam uma única unidade lógica de trabalho (SILBERSCHATZ; KORTH; SUDARSHAN, 2006). Falha bizantina é um conceito que captura uma grande variedade de "outros" comportamentos faltosos, os quais incluem corrupção de dados, programas que não seguem o protocolo correto, e até mesmo, comportamentos maliciosos que buscam violar a confiabilidade do sistema (BIRMAM, 1996). Mais detalhes sobre falhas bizantinas podem ser vistos no capítulo 2.

A técnica utilizada para garantir tolerância a falhas é a replicação através de máquina de estados. Zyzzyva (KOTLA ET AL., 2009) foi o protocolo de consenso escolhido para ser usado como base do Tuplebiz. A escolha do protocolo em questão se deve ao fato do bom desempenho que esse apresenta quando comparado aos demais (KOTLA ET AL., 2009). Apesar do Tuplebiz ser construído no topo do Zyzzyva, muitos dos conceitos aqui apresentados podem ser usados em conjunto com outros protocolos de consenso. As falhas bizantinas podem ser de natureza acidental ou maliciosa (BESSANI, 2006). Esse trabalho trata ambas as falhas, entretanto dá uma maior ênfase às falhas acidentais. Não é tratada confidencialidade de tuplas, ou seja, os dados inseridos nas tuplas não são criptografados de tal modo que as réplicas não possuam um modo de decifrá-los. Essa funcionalidade pode ser encontrada no trabalho de (BESSANI, 2006) e pode ser adaptada ao Tuplebiz.

As operações realizadas pelo Tuplebiz são as operações de out(t), inp(s) e rdp(s). Isso se deve ao fato do Tuplebiz satisfazer a propriedade de vivacidade (*liveness*). Operações bloqueantes (rd(s) e in(s)) não satisfazem a propriedade de terminação livre de espera (*wait-free termination*) e vivacidade (*liveness*) (BESSANI, 2006). Além da propriedade de vivacidade, o sistema também satisfaz a propriedade de segurança (*safety*).

Além disso, para aumentar a escalabilidade, o Tuplebiz foi dividido em partições. As partições são completamente independentes e não têm conhecimento uma das outras. Elas podem ser distribuídas entre diferentes servidores e ficarem geograficamente mais próxima dos clientes que os utilizam.

3.2 O protocolo Zyzzyva

Entre os trabalhos de (CASTRO; LISKOV, 2002), (ABD-AL-MALEK ET AL., 2005) e (COWLING ET AL., 2006), o Zyzzyva (KOTLA ET. AL., 2009) se apresentou como o protocolo mais eficiente (KOTLA ET. AL., 2009). Uma das diferenças entre o Zyzzyva e os demais é a ideia de resposta especulativa. Se todas as réplicas estiverem corretas, se tem a resposta certa com menos passos de comunicação. Se o cliente obtiver uma resposta correta de $3f$ +1 nodos, então se tem uma resposta em três passos. Se o número de respostas corretas for entre $2f$ + 1 e $3f$+1, a resposta é obtida em seis passos. Por apresentar um melhor desempenho, o Zyzzyva foi escolhido como base para o Tuplebiz.

Zyzzyva faz uso de resposta especulativa para aumentar o desempenho de replicação em sistemas que usam a abordagem de máquina de estado baseada em réplicas para suportar falhas bizantinas (KOTLA ET. AL., 2009).

O protocolo tem como participantes, o nodo primário e os nodos secundários que garantem a resposta correta e o cliente que faz requisições ao sistema.

Zyzzyva possui um nodo primário que garante a ordem geral das operações. O cliente faz uso de *timestamps* para garantir a ordem das mensagens enviadas. O nodo primário ao receber uma requisição do cliente, só a aceita se o *timestamp* da mensagem for maior que o último *timestamp* armazenado para o cliente em questão. Se a requisição for aceita, então o primário adiciona um número indicando a ordem dentro da *view* atual. Uma *view* é o período no qual um nodo é considerado primário. Um nodo primário é destituído da função quando há suspeita das réplicas quanto o funcionamento do mesmo. Quando um primário é destituído, o número da *view* é incrementado em um.

Zyzzyva possui três diferentes sub-protocolos:

- Sub-protocolo de *agreement*: responsável por proporcionar ao cliente a execução das operações de forma a garantir uma resposta correta;
- Sub-protocolo de *view change*: responsável por apontar um novo primário em caso de falha do mesmo;
- Sub-protocolo de *checkpoint*: responsável por salvar periodicamente o estado das réplicas para futura recuperação.

Dentre os sub-protocolos do Zyzzyva, o único que será apresentado com mais detalhes é o de *agreement*, pois uma melhor compreensão dele é necessária para entender as modificações realizadas pelo Tuplebiz. Para detalhes dos demais protocolos ver (KOTLA ET. AL., 2009).

O sub-protocolo de *agreement* possui um grupo de mensagens cujos parâmetros podem ser vistos na Tabela 3.1. Os próximos parágrafos irão abordar o comportamento de cada mensagem segundo a execução padrão. Execuções alternativas podem ser vistas em (KOTLA ET. AL., 2009). As mensagens são:

- *Request*: mensagem enviada pela cliente para a réplica primária;
- *Order-request*: mensagem enviada pela réplica primária para as réplicas secundárias;
- *Spec-response*: resposta enviada pelas réplicas para os clientes;
- *Fill-hole*: mensagem enviada por uma réplica secundária para a réplica primária quando a réplica não possui todas as mensagens de *order-request*;
- *Local-commit*: resposta da réplica que indica para o cliente que recebeu a mensagem de *commit*;
- *Commit*: mensagem do cliente para as réplicas, a qual indica que o cliente recebeu a resposta correta.

Tabela 3.1: Siglas usadas nas mensagens do Zyzzyva

Sigla	Operação
c	Identificador do cliente
CC	Certificado de Confirmação
d	Resumo da mensagem requisitada pelo cliente d = H(m)
i	Identificador dos servidores
n	Número de sequência
o	Operação requisitada pelo cliente
r	Resposta da aplicação para o cliente
t	*Timestamp* indicada pelo cliente para a operação
v	Número da v*iew*
ND	Dado adicional que possa ser requerido pela aplicação
$< >_{\sigma i}$	Indica que a mensagem é assinada por i

Uma operação começa quando um cliente envia uma mensagem m = ‹ REQUEST, t, o, c› $_{\sigma c}$ para a réplica primária. Essa, por sua vez, ao receber a mensagem m, envia para as demais réplicas a mensagem OR = ‹‹ORDER-REQ, v, n, h_n, d, ND› $_{\sigma p}$, m›.

Ao receber a mensagem OR, cada réplica faz as verificações necessárias para garantir que a validade da mensagem, como por exemplo, se o n é igual ao máximo n armazenado ($máx_n$) acrescido de uma unidade. Após as verificações, a réplica manda uma reposta para o cliente R = ‹‹SPEC-RESPONSE, v, n, h_n, H(r), c, t› $_{\sigma i}$, i, r, OR›. Se o n recebido pela réplica, for maior que o $máx_n$ + 1, então a réplica envia uma mensagem F = ‹‹FILL-HOLE, v, k, n, i› $_{\sigma i}$ para a primária. A réplica primária irá reenviar todas as mensagens de ORDER-REQ do intervalo k (k é igual $máx_n$ armazenado pela réplica) até o atual n.

O cliente ao receber as respostas R das réplicas procede de três maneiras distintas, dependendo da quantidade de respostas iguais.

No primeiro caso são recebidas 3f + 1 respostas iguais e o cliente retorna a reposta para a aplicação.

No segundo caso são recebidas entre 2f + 1 e 3f respostas iguais, o que provoca a troca de algumas mensagens. O cliente manda para todas as réplicas que enviaram a mesma resposta a mensagem C = ‹ COMMIT, c, CC› $_{\sigma c}$. Cada uma das réplicas recebe a mensagem C, salva o CC enviado e responde para o cliente com a mensagem RC = ‹LOCAL-COMMIT, v, d, h, i, c› $_{\sigma i}$. Ao receber 2f + 1 mensagens de RC, o cliente retorna o resultado para a aplicação.

No terceiro e último caso, o cliente recebe menos de 2f + 1 respostas, o que provoca o reenvio por parte do cliente da mensagem de REQUEST, só que dessa vez, a mensagem é enviada para todas as réplicas. Cada réplica ao receber a mensagem de REQUEST, envia uma mensagem CR = ‹CONFIRM-REQ, v, m, i› $_{\sigma i}$ para o primário. Esse, por sua vez, irá reenviar a ORDER-REQ se já tiver recebido esse REQUEST anteriormente, ou enviará um novo ORDER-REQ se a mensagem não foi processada em outro período.

3.3 Não determinismo

Todas as réplicas devem atualizar seu estado de modo determinístico para obterem-se as mesmas respostas (RODRIGUES; CASTRO; LISKOV, 2001). Entretanto, as repostas de algumas operações no espaço de tuplas são não-determinísticas. Se um conjunto de tuplas combina com a anti-tupla s para as operações de in(s), rd(s), inp(s) e rdp(s), quaisquer tuplas do conjunto pode ser retornada de forma não determinística (CARRIERO; GELERNTER, 1992).

Dada uma anti-tupla e um conjunto de tuplas que casam com ela, o espaço de tuplas pode retornar qualquer uma das tuplas do conjunto de forma não determinística. Contudo, para execução de um protocolo tolerante a falhas bizantinas, é necessário que todas as réplicas retornem a mesma tupla para um anti-tupla.

O Tuplebiz define alguns mecanismos cuja função é garantir que todas as réplicas não falhas retornem a mesma resposta para uma mesma operação. Esses mecanismos são:

- Tuplas únicas (BESSANI, 2006);
- Consulta especializada;
- Réplicas idênticas.

Uma tupla no Tuplebiz é única. Mesmo que duas tuplas forem inseridas com campos idênticos, elas serão consideradas tuplas distintas. Uma vez inserida a tupla recebe um identificador. Esse identificador inicia em zero e é incrementado em uma unidade a cada inserção no espaço.

Uma consulta de uma tupla no Tuplebiz sempre retorna a tupla que case com a anti-tupla e possua o menor identificador. Todas as réplicas são idênticas e executam todas as operações na mesma ordem, logo se garante que (1) as tuplas inseridas são identificadas na mesma ordem; (2) as tuplas retiradas possuam o mesmo identificador em cada réplica.

O espaço de tuplas, em todas as réplicas, é idêntico e possui o mesmo estado inicial. Isso garante que o mecanismo usado para busca, bem como, o estado inicial, seja mais facilmente controlado. Por exemplo, garante-se que o valor inicial de identificação seja zero. O uso de réplicas distintas seria uma abordagem interessante para aumentar a confiabilidade, pois cada réplica pode ter uma implementação distinta e teria a

vantagem de utilizar programação diversitária (WEBER, 2002). A adaptação para uso de réplicas distintas é um trabalho futuro.

3.4 Transações no Tuplebiz

A transação no Tuplebiz deve ser definida de forma explícita pelo cliente do espaço de tuplas. O cliente deve indicar o início e o final da transação, assim sendo, nem toda a operação acontece dentro de um contexto transacional. Essa abordagem foi utilizada em virtude do desempenho, pois a execução no contexto transacional é um pouco mais lenta do que fora dele e nem sempre o cliente tem a necessidade de usar uma transação. O desempenho é menor porque deve ser mantida uma lista de leitura escrita, um gerente de bloqueio e garantir que todas as partições estejam sincronizadas com o final da transação seja abortando seja realizando a confirmação.

A indicação de início e fim da transação se dá pelas operações de *transaction_begin* e *transaction_commit* ou *transaction_abort*. Uma transação pode englobar mais de uma partição. A operação *transaction_begin* indica que a transação deve ser iniciada. A operação de *transaction_commit* indica que a transação deve ser completada. E, por fim, a operação de *transaction_abort* indica que a transação deve ser abortada.

Não são permitidas transações aninhadas. As transações aninhadas são sub-transações que podem ser executadas concorrentemente. As transações e subtransações podem ser representadas por uma árvore, onde a raiz é a transação e as folhas são as sub-transações (PU, 1986). Esse tipo de abordagem aumenta a complexidade do modelo e é sugerida como trabalho futuro. Cada cliente pode ter apenas uma transação ativa por vez. Quando um cliente, que já tem uma transação aberta, tenta abrir uma nova transação, ele recebe uma mensagem de erro.

As transações no Tuplebiz respeitam as propriedades ACID. A atomicidade é garantida pelas listas de leitura e escrita. Cada transação possui suas próprias listas que são mantidas ativas durante todo o contexto transacional. Quando uma operação de inp(*s*) é realizada, a tupla fica armazenada em uma lista de leitura. Quando a operação de out(*s*) é realizada, a tupla fica armazenada na lista de escrita. As operações de rdp(*s*) são marcadas como somente leitura no gerente de bloqueio, o que será melhor detalhado posteriormente. Ao final da transação, se a transação é abortada, as tuplas na lista de leitura retornam para o espaço de tuplas e as que se encontram na lista de escrita são descartadas. Ao final da transação, se é realizada uma confirmação, as tuplas da lista de leitura são descartadas e as tuplas que se encontram na lista de escrita são inseridas no espaço de tuplas. Desse modo, as operações no espaço de tuplas são executadas no modelo do tudo-ou-nada, isto é, todas as operações são refletidas no espaço de tuplas ou nenhuma delas é.

As listas de leitura e escrita também garantem a consistência. O banco de dados sempre tem um estado válido antes e depois da transação. O Tuplebiz garante a durabilidade através do uso de banco de dados para armazenar as tuplas.

A transação mantém o mesmo nível de isolamento presente no Javaspace (FREEMAN, 1999). Apesar de (FREEMAN, 1999) enunciar que o nível de isolamento é serializável, (BUSI, 2001) prova que mais algumas restrições devem ser aplicadas para que o nível de isolamento seja serializável. O Tuplebiz segue as mesmas premissas de bloqueio e visibilidade do Javaspace (FREEMAN, 1999) e do Gigaspace (GIGASPACE, 2011). As premissas de bloqueio e visibilidade são:

1) As operações de out(t) são visíveis quando as transações são finalizadas.
2) A operação de rdp(s) é bloqueada para somente leitura dentro do contexto transacional. Uma vez que a transação executa a operação rdp(s), a tupla retornada fica disponível apenas para leitura.
3) A operação de inp(s) remove a tupla do espaço de tuplas, entretanto se a transação é abortada, a tupla retorna ao espaço de tuplas.

Três exemplos foram criados para demonstrar cada uma dessas premissas. Eles são tratados nos parágrafos seguintes. Para todos eles, considera-se que no instante t0, o espaço de tuplas está vazio e que nenhum cliente possui transação aberta.

A Figura 3.1 mostra uma operação de out(t) dentro de um contexto transacional. No instante de tempo t1, o cliente 1 inicia a transação; e no tempo t2, o mesmo cliente realiza a operação de out(“a”, 2). Devido à premissa de bloqueio e visibilidade 1, ao realizar a operação de rdp (“a”, ?), o cliente 2 recebe uma indicação que não existe tupla que casa com a anti-tupla. Entretanto, após a transação ser encerrada pelo cliente 1 no momento t4, o cliente 2, ao refazer a operação rdp(“a”, 2), recebe a tupla (“a”, 2) no tempo t5.

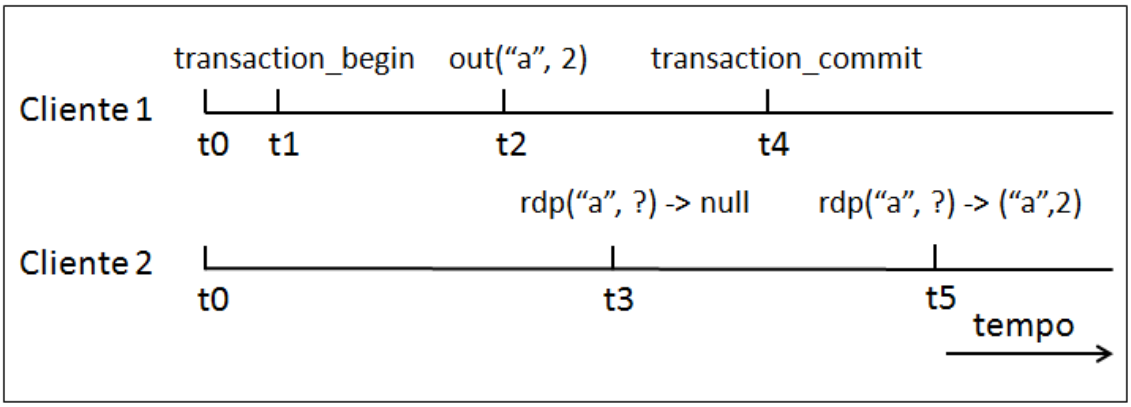


Figura 3.1: Exemplo da operação de out(t) dentro do contexto transacional

Um exemplo de funcionamento da premissa de bloqueio e visibilidade 2 pode ser visto na Figura 3.2. No instante de tempo t1, o cliente 1 inicia a transação. Logo em seguida, no momento t2 uma tupla é inserida no espaço. No tempo t3, ao realizar a operação de rdp(“a”, ?), o cliente 1 realiza o bloqueio para leitura da tupla (“a”, 2). Esse bloqueio quer dizer, em outras palavras, que a tupla estará visível fora da transação somente para operações de rdp(s). Por conseguinte, para o cliente 2, a tupla está disponível para a operação de rdp(“a”, ?) no instante t4, mas não está disponível para a operação de inp(“a”, ?) no momento t5. Ao finalizar a transação, todos os bloqueios adquiridos são liberados e, por esse motivo, a operação inp(“a”, ?), realizada no tempo t7, retorna a tupla (“a”, 2).

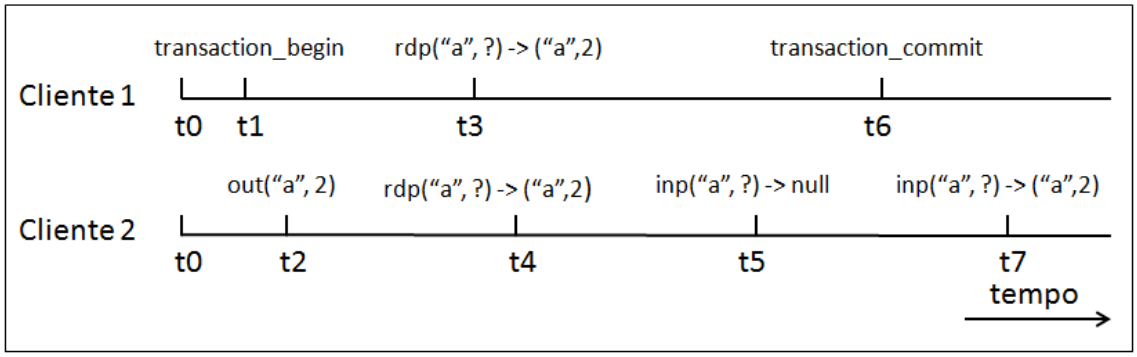


Figura 3.2: Exemplo da operação de rdp(s) dentro do contexto transacional

A Figura 3.3 exibe um exemplo de funcionamento da premissa de bloqueio e visibilidade 3. No instante de tempo t1, o cliente 1 inicia a transação. Logo em seguida, no momento t2 uma tupla é inserida no espaço. A tupla é retirada do espaço de tuplas no momento t3 pelo cliente 1. Durante os tempos t3 e t5, ocorre uma falha e a transação é abortada no momento t5. Verifica-se que a tupla não é visível para o cliente 2 antes de t5 (momento em que a transação é abortada), mas a mesma tupla é visível no momento t6. Isso se deve ao fato de que uma transação, quando abortada, deve retornar ao espaço todas as tuplas retiradas.

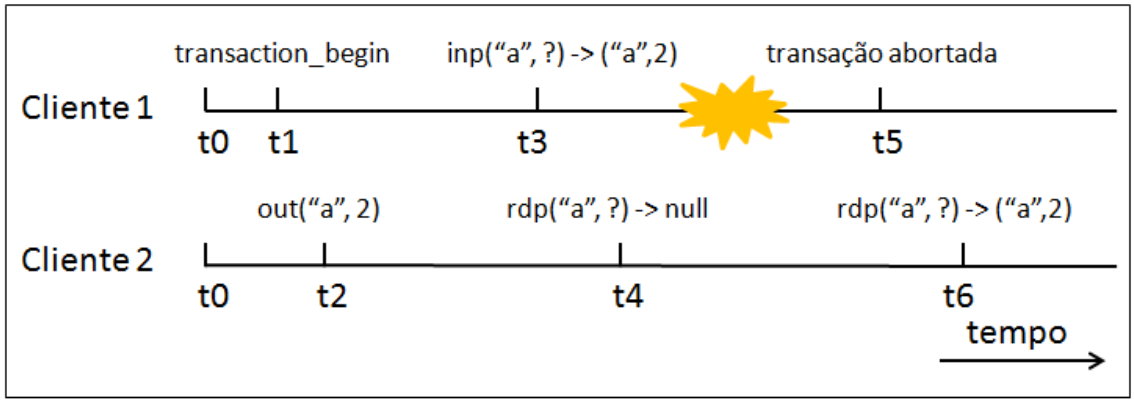


Figura 3.3: Exemplo da operação de inp(s) dentro do contexto transacional

O bloqueio das transações é gerenciado pelo Gerente de Bloqueio. Esse módulo é melhor detalhado na seção que descreve a arquitetura da réplica.

3.5 Arquitetura do sistema

A arquitetura do Tuplebiz pode ser vista na Figura 3.4. O processo (P) é uma aplicação cliente que desconhece o protocolo de replicação. O processo se comunica com o espaço de tuplas através de um cliente (C) que conhece o protocolo de replicação.

O Gerente de Transação (GT), como o próprio nome diz, gerencia as transações entre as diferentes partições. Para cada cliente, existe um Gerente de Transação (GT) que se comunica com os as partições. Esse é responsável pela orquestração da transação entre as diferentes partições.

As partições contêm o espaço de tuplas. Cada uma delas é uma porção do espaço como um todo.

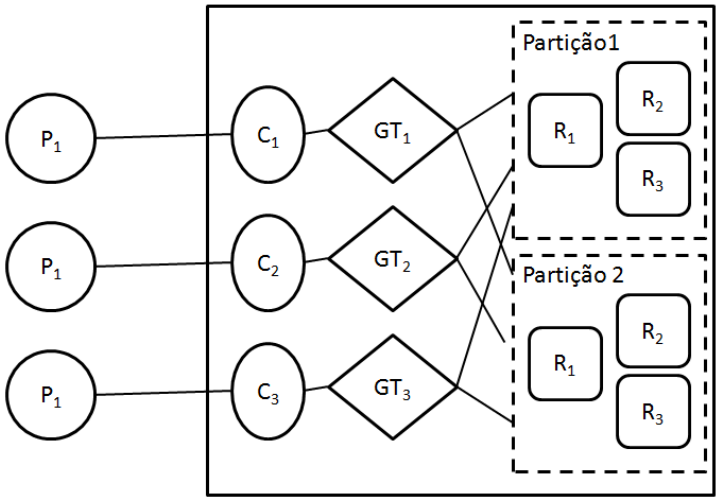


Figura 3.4: Arquitetura do Sistema

3.5.1 Partições

Um problema comum em uma aplicação distribuída é a escalabilidade. Esses problemas têm, normalmente, como causa raiz a capacidade limitada de servidores e redes (TANENBAUM; VAN STEEN, 2002). Uma abordagem para solucionar esse problema é a distribuição que consiste em pegar um componente, dividir em pedaços menores e distribuí-lo pelo sistema (TANENBAUM; VAN STEEN, 2002).

Para ilustrar os problemas enumerados, será usado o exemplo da Figura 3.5. Esse exemplo mostra uma solução de integração que usa o Tuplebiz. As aplicações se encontram geograficamente distribuídas: as aplicações A e B estão em Porto Alegre e as aplicações C e D estão em São Paulo. Considera-se nesse cenário que a aplicação A e B, bem como as aplicações C e D se comunicam constantemente entre si e as aplicações A e B se comunicam com as aplicações C e D esporadicamente. Adicionalmente, considera-se que a solução de integração está sendo executada em Porto Alegre e que cada filial tem sua própria rede interna.

Tendo em vista o exemplo enunciado no parágrafo anterior, as aplicações C e D apesar de estarem na mesma rede, precisam acessar a rede de Porto Alegre para se comunicar. Tem-se aqui um caso de aumento de latência. Ainda focado nesse exemplo, todas as aplicações estão acessando o mesmo espaço de tuplas, logo, dependendo do número de clientes, o espaço de tuplas não poderá responder com uma taxa de transferência aceitável.

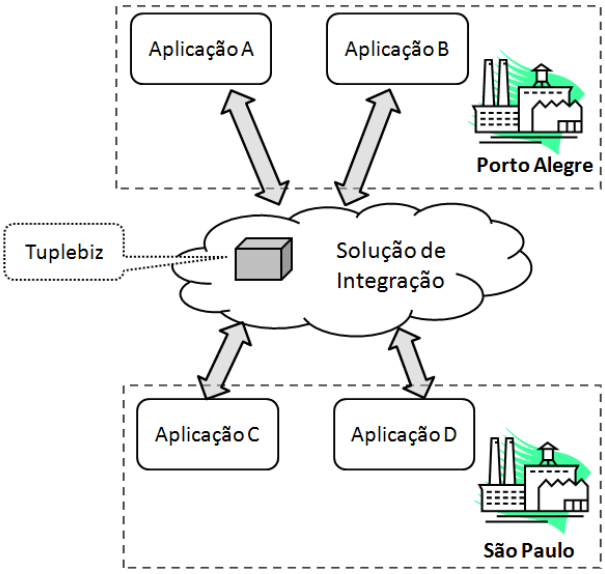


Figura 3.5: Exemplo de solução de integração

Com essas limitações em mente, o Tuplebiz possui o conceito de partições para aumentar a escalabilidade. Esse modo de operação permite que o espaço de tuplas seja geograficamente distribuído. A forma escolhida para atingir essa capacidade é o uso de partições.

No Tuplebiz uma partição segue a definição de (MENEZES, 2004): partição de um conjunto A é um conjunto de subconjuntos não-vazios e mutuamente disjuntos de A e a união de todos esses subconjuntos resultam em A. Em outras palavras, cada partição gerencia apenas um subconjunto de dados e não há intersecção entre partições.

O Gerenciador de Transações – que será melhor detalhado posteriormente - conhece a relação de equivalência aplicada à tupla que induz para a partição correta. A relação de equivalência é definida de acordo com a aplicação que utiliza o Tuplebiz. A Figura 3.6 ilustra a divisão do Tuplebiz em partições. A parte “a” da figura mostra uma única partição com todos os clientes acessando a mesma. A parte “b” da figura mostra duas partições que podem estar geograficamente distribuídas, isto é, em servidores distintos.

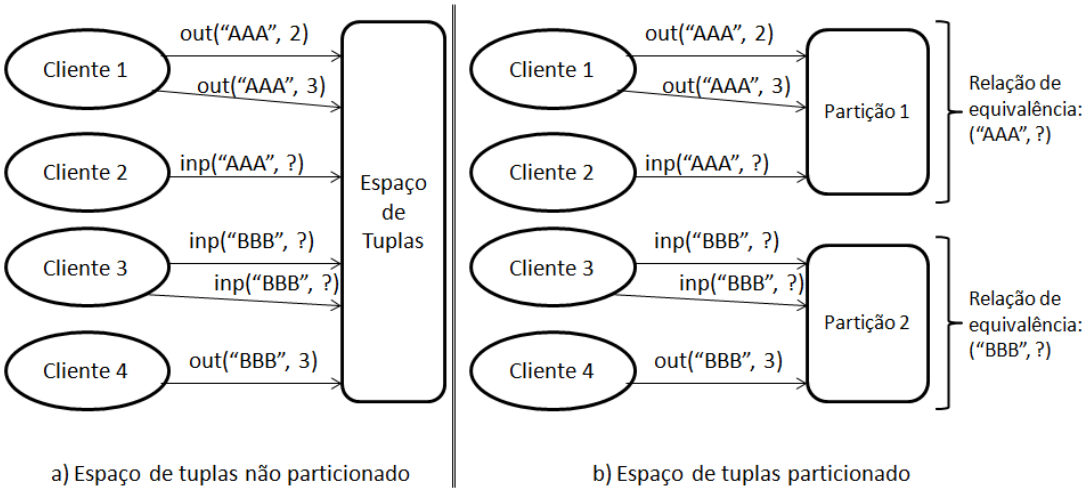


Figura 3.6: Espaço de tuplas particionado

Cada partição do Tuplebiz é um espaço de tuplas a parte, logo cada partição tem seu próprio mecanismo de replicação. Uma partição desconhece as demais. Somente o Gerenciador de Transações conhece todas as partições pertencentes ao espaço de tuplas. A Figura 3.7 mostra como são definidas as réplicas para cada partição. Os blocos T_1, T_2 e T_3 são as partições do espaço de tuplas e os blocos de T_{xm} são as réplicas, onde x indica a partição e m indica o número da réplica.

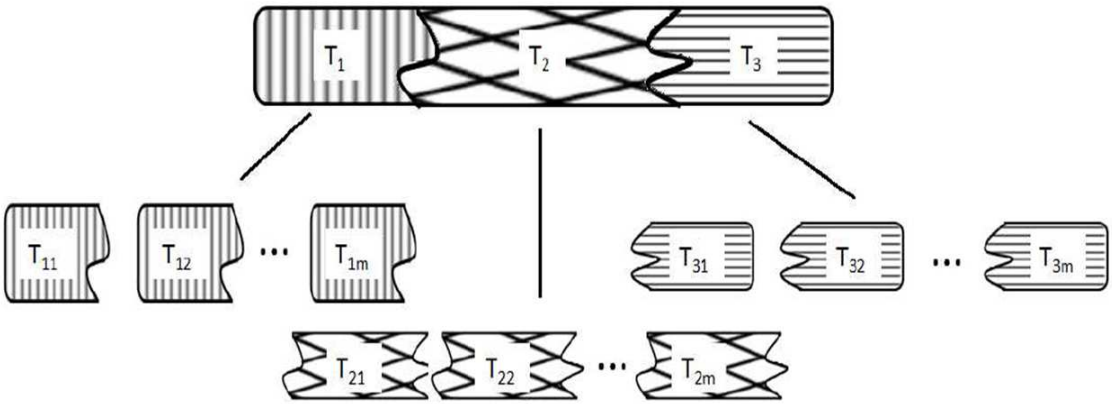


Figura 3.7: Replicação das partições do Tuplebiz

A principal ideia em torno dessa abordagem é que clientes que se comunicam mais constantemente fiquem mais próximos de uma mesma partição e, assim, diminua a latência. Porções menores têm um menor número de clientes acessando o espaço simultaneamente e, por conseguinte, há um aumento da taxa de transferência.

Ainda considerando o exemplo da Figura 3.5, tem-se as mesmas premissas apresentadas anteriormente, salvo uma exceção: o Tuplebiz está dividido em duas partições uma em Porto Alegre e outra em São Paulo, ao invés de todo o espaço estar em Porto Alegre. Nesse caso há uma diminuição da latência, pois as aplicações que se comunicam com mais frequência estão na mesma rede.

Todavia, essa abordagem traz, também, um novo problema para ser endereçado: como tratar uma transação que possui tuplas em mais de uma partição. A solução desse problema envolve o Gerente de Transações e na seção referente a ele há uma explicação mais detalhada de como esse problema foi solucionado nessa arquitetura.

3.5.2 Gerente de Transação

O Gerente de Transações (GT) é responsável por gerenciar as transações entre as diferentes partições do sistema. A lógica do GT é simples, ele apenas encaminha as requisições para as partições. O Gerente de Transação é centralizado. Essa é a mesma abordagem utilizada por (VANDIVER; BALAKRISHNAN; LISKOV, 2007). Entretanto, como trabalho futuro prevê-se a também a replicação do Gerente de Transação.

O estado do GT pode ser definido através de uma máquina de estados. A Figura 3.8 ilustra os possíveis estados do GT. No início o estado é fechado, o qual permanece até a transação ser criada. Uma fez que o cliente tenta executar uma transação de *transaction_begin*, o gerente de transações fica no estado de “esperando resposta de início” e nesse estado permanece até receber as respostas das réplicas. Uma vez que a transação é aberta, o GT permanece no estado de “aberto” até que o cliente realize uma operação de *transaction_commit* ou de *transaction_abort*. Durante a confirmação ou

aborto o GT realiza o protocolo de *commit* de duas fases que é representado pelos estados de “fase de preparação”, “esperando ack de aborto” e “esperando ack de confirmação”.

A relação das mudanças de estados e o algoritmo do GT são mostrados no Algoritmo 3.1. O algoritmo introduz novas operações ao Tuplebiz, sendo elas: *transaction_vote*, *ack_commit* e *ack_abort*. Pode-se dividir o algoritmo em duas grandes partes: a primeira que são as mensagens recebidas pelo cliente (da linha 3 até 22) e as mensagens recebidas das réplicas (da linha 23 até 48). É importante salientar, que quando se fala em resposta da partição no algoritmo, está se considerando que o protocolo de falhas bizantinas foi executado e que a resposta já é o consenso de todas as réplicas da partição.

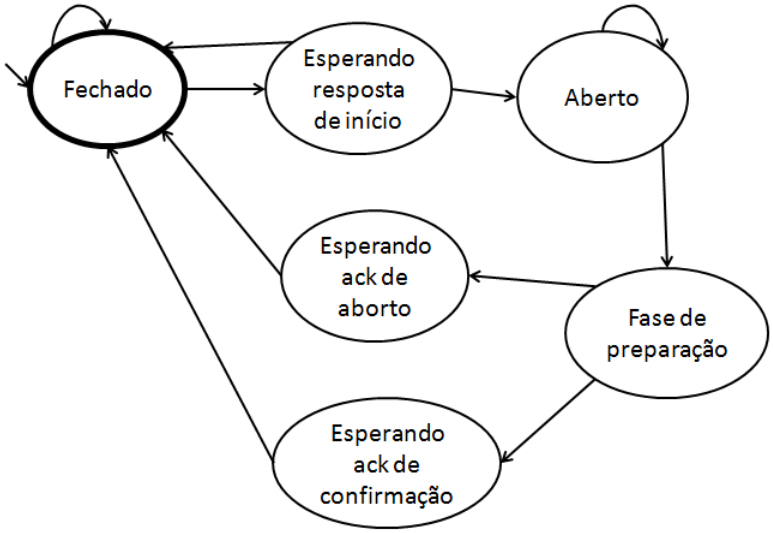


Figura 3.8: Estados do gerente de transação

No início da execução do algoritmo, o estado do GT é fechado, pois não há transações abertas. Quando ele recebe uma mensagem relativa ao espaço de tuplas do cliente (leia-se inp(s), rdp(s) ou out(t)) o estado permanece o mesmo, seja aberto, seja fechado. A mensagem é somente encaminhada para a partição correspondente. Esse código se encontra nas linhas 4 até 6.

Quando o cliente inicia uma transação (o cliente chama uma operação de *transaction_begin*, na linha 7), primeiramente é verificado o estado atual do gerente de transações. Se o estado for “aberto”, então uma mensagem de erro é enviada para o cliente. Esse erro é enviado, pois transações aninhadas não são permitidas. Se o estado for fechado, o GT envia para todas as partições uma mensagem de início de transação (linhas 26 até 31). Nesse ponto não é realizado um protocolo de *commit* de duas fases para identificar se a transação pode ser iniciada. Se por alguma eventualidade uma partição não puder iniciar a transação, essa usará seu voto para que transação seja abortada durante a confirmação.

Quando o cliente envia uma operação de *transaction_commit* (linha 15) ou *transaction_abort* (linha 19), o GT inicia um protocolo de *commit* em duas fases. Primeiramente é enviada para todas as partições uma requisição de confirmação ou aborto e o estado muda para “Fase de preparação”. Logo após, o GT fica no aguardo dos votos da partição (linha 33). Se pelo menos uma partição indica que não pode realizar a *confirmação*, a transação como um todo é abortada. Uma vez recebidas respostas de

todas as partições, o resultado da operação é enviado para as partições (linhas 35 a 38) e o estado muda para “esperando ack de aborto” ou “esperando ack de confirmação” de acordo com o resultado. Por fim, ao receber a resposta do ack (linha 45) a resposta é enviada para o Cliente.

Algoritmo 3.1: Gerente de Transações

```
1.  Inicialização:
2.      estado <- “fechado”;
3.  RecebeMensagemDoCliente(m):
4.    caso (m é Operação do Espaço de Tuplas)
5.       p <- avalia(m);
6.       envia Requisição(m) para p ∈ Partições;
7.    caso (m é transaction_begin)
8.       se (estado é “aberto”)
9.          envia Resposta(“Erro”) para Cliente
10.      senão
11.         para todo p ∈ Partições
12.            envia Requisição(transaction_begin) para p;
13.         estado <- “esperando resposta de início”;
14.      fechaSe
15.   caso (m é transaction_commit)
16.      ∀ p ∈ Partição
17.         envia Requisição(transaction_commit) para p;
18.         estado <- “Fase de preparação”;
19.   caso (m é transaction_abort)
20.      ∀ p ∈ Partição
21.         envia Requisição(transaction_abort) para p;
22.         estado <- “Fase de preparação”;
23. RecebidaMensagemDaRéplica(m):
24.   caso (m é resposta de operação do espaço de tuplas)
25.      envia Resposta(m) para Cliente;
26.   caso (m é reposta para transaction_begin)
27.      adiciona m.RespostaTransação no
28.          p. RespostaTransação ∈ Partições;
29.      se (∀ p ∈ Partição tem RespostaTransação)
30.        enviar Resposta(RespostaTransação) para Cliente;
31.        estado <-“Aberto”
32.      fechaSe
33.   caso (m é reposta para transaction_commit ou transaction_abort)
34.      adiciona m.Voto em p.Voto ∈ Partições;
35.      se ( ∀ p ∈ Partições tem Voto)
36.         voto <- pegarVotoSelecionado()
37.         ∀ p ∈ Partição
38.            envia Resposta(transaction_vote(voto)) para p;
39.         se(voto é Aborto)
40.            estado <- “esperando ack de aborto”
41.         senão
42.            estado <- “esperando ack de confirmação”
43.         fechaSe
44.      fechaSe
45.   caso ( m é ack_abort ou m é ack_commit)
```

```
46.      se ( ∀p ∈ Partições tem ack)
47.          enviar Resposta(voto) para Cliente;
48.          estado <-fechado.
```

3.6 Arquitetura das réplicas

A arquitetura da réplica pode ser vista Figura 3.9. Os blocos em cinza são contribuição do Tuplebiz, os blocos em branco são contribuição do Zyzzyva e o bloco com listras é parte do protocolo Zyzzyva que foi alterado para suportar as necessidades do Tuplebiz. Todas as réplicas são idênticas quando comparada a arquitetura. O que as diferencia é o papel de cada uma no protocolo. A réplica primária executa alguns passos a mais quando comparada a outras réplicas.

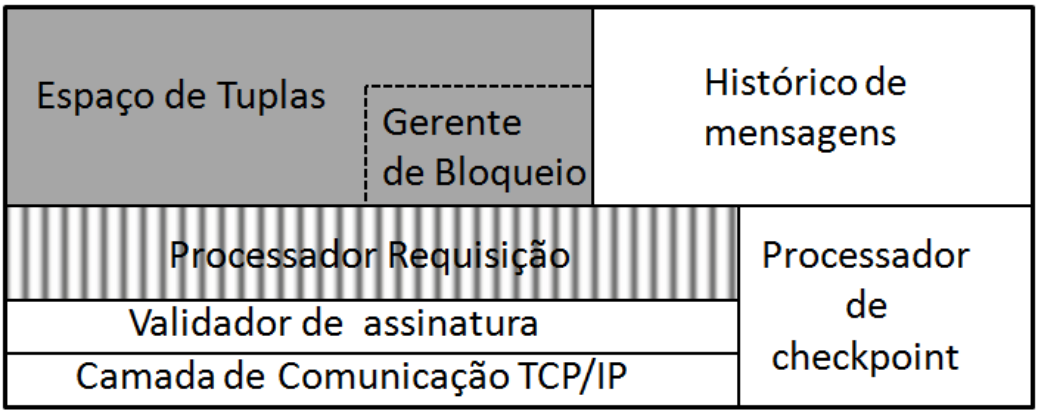


Figura 3.9: Arquitetura de uma réplica

A camada de comunicação é responsável por enviar as mensagens entre as réplicas e o gerente de transações. Não possui qualquer tipo de criptografia, pois isso é tratado na camada superior. O validador de assinatura assina as mensagens enviadas e verifica se as assinaturas das mensagens recebidas estão corretas.

Como parte do protocolo, o processador armazena um histórico de mensagens. O histórico de mensagens existe para reenviar mensagens perdidas e também para recuperação de dados em caso de *crash*. As mensagens podem ser reenviadas em duas situações: (1) quando a mensagem falha ao ser enviada ao cliente e (2) quando uma réplica retorna depois de um *crash* e a réplica primária precisa reenviar as mensagens perdidas. Detalhes do armazenamento se encontram em (KOTLA ET AL., 2009).

O processador de *checkpoint* está relacionado com o histórico de mensagens. Ele armazena as mensagens de tempos em tempos para realizar uma recuperação de dados em caso de falhas.

O processador de requisição é a parte mais importanteda réplica. Ele executa o protocolo tolerante a falhas bizantinas. Esse processador realiza as operações definidas pelo Zyzzyva (KOTLA ET AL., 2009). A execução dos algoritmos, que realizam a transação, também ocorre nesse módulo. Anteriormente, foi visto o funcionamento dos algoritmos relacionados às operações de *transaction_begin*, *transaction_commit* e *transaction_abort* no Gerente de Transações e, agora, nos algoritmos a seguir temos as mesmas operações vistas do ângulo da réplica.

O Algoritmo 3.2 mostra como é a execução da operação de *transaction_begin*. Primeiramente, o Gerente de Bloqueio é requisitado a abrir uma transação (linha 2). Isso

significa criar as filas de leitura e escrita da transação. Se já houver uma transação aberta para o cliente em questão, a operação retorna falso, caso contrário a resposta é verdadeiro. A réplica também mantém um status da transação (linha 3) o qual é usado para indicar o voto de aborto ou confirmação no final da transação.

Algoritmo 3.2: Processador de Requisição – Operação de transaction_begin

```
1. RecebeOperação(transaction_begin)
2.    transaçãoOK <- GerenteDeBloqueio.AbrirTransação(transaction_begin.ClientId);
3.    listaTransação.Adiciona(transaction_begin.ClientId, transaçãoOK);
4.    Resposta(transaçãoOK)
```

O Algoritmo 3.3 mostra como é a execução da operação de *transaction_commit* na réplica. Primeiramente, a transação é congelada para evitar que o cliente continue a acessar a mesma transação. Nesse passo, é verificado o status da transação, o qual é o indicador se a transação deve ou não ser finalizada com sucesso. Se o status for falso, o voto é favorável ao aborto da transação (linhas 3 e 4). A transação pode ter status falso quando já existir transação aberta no momento da abertura da transação ou quando ocorrer um timeout da transação.

Algoritmo 3.3: Processador de Requisição – Operação de transaction_commit

```
1. RecebeOperação(transaction_commit)
2.    GerenteDeBloqueio.CongelaTransação(transaction_commit.ClientId);
3.    se (listaTransação.Retira(transaction_commit.ClientId) == false)
4.      Resposta(Aborta_Transação);
5.    senão
6.      Resposta(Commit_Transação);
7.    endSe
```

O sistema de *timeout* de transação é utilizado para evitar que clientes maliciosos mantenham o sistema em *deadlock*. Um cliente malicioso pode abrir uma transação, realizar várias operações de inp(s) e rdp(s) e nunca realizar a confirmação. Nessa situação, diversas tuplas ficarão bloqueadas e outros clientes poderão esperar indefinidamente para finalizar as suas transações. Com o intuito de evitar esse comportamento, um sistema de *timeout* foi adicionado. Um cliente pode manter o bloqueio de uma tupla por tempo indefinido; porém se a tupla for necessária em outra transação, um contador é iniciado. O contador indica o tempo máximo que a transação a qual possui o bloqueio deve completar. Se ela não for completada no tempo esperado, todos os bloqueios são liberados e a transação é marcada como falha. Se o cliente possuidor da transação falha tentar realizar a confirmação, a réplica votará como aborto. Uso de sistemas de timeout é uma abordagem usada em outros sistemas que toleram falhas bizantinas, como o (VANDIVER; BALAKRISHNAN; LISKOV, 2007).

O Algoritmo 3.4 demonstra como funciona uma operação de *transaction_abort*. Primeiramente, é requisitado ao gerente de bloqueio que congele a transação e, consequentemente, não permita mais alterações na mesma. Logo após, é enviado o voto de aborto.

Algoritmo 3.4: Processador de Requisição – Operação de transaction_abort

```
1. RecebeOperação(transaction_abort)
2.       GerenteDeBloqueio.CongelaTransação(transaction_commit.ClientId);
3.       Resposta(Aborta_Transação);
```

O Algoritmo 3.5 mostra como funciona a operação de *transaction_vote* do lado da réplica. Dependendo da resposta recebida uma ação é tomada. Se a resposta for uma confirmação, é requisitado ao gerente de bloqueio que finalize a transação. Se a resposta for um aborto, então é requisitado um *rollback* da transação.

Algoritmo 3.5: Processador de Requisição – Operação de transaction_vote

```
1. RecebeOperação(transaction_vote)
2.    se (vote.VotoSelecionado == confirmação)
3.       GerenteDeBloqueio.FinalizaTransação(vote.ClientId);
4.      Enviar Resposta()
5.    senão
6.       GerenteDeBloqueio.RollbackTransação(vote.ClientId);
7.    endSe
```

O espaço de tuplas na Figura 3.9 é onde as operações sobre o espaço de tuplas são realizadas. Diferentemente da definição original de espaços de tuplas, ele possui internamente um gerente de bloqueio. Esse gerente é similar aos encontrados em banco de dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2006). O Gerente de Bloqueio tem como principal objetivo garantir que a transação possua isolamento. Dentre as funções por ele realizadas estão a gerência da fila de bloqueio, armazenamento da lista de escrita e armazenamento de lista de leitura. Cada operação tem um conjunto de passos a ser executados quando estão dentro do contexto transacional.

As operações disponíveis no gerente de bloqueio são:

- **AbrirTransação**: essa operação cria as listas de leitura e escrita para o cliente em questão. Se já houver uma transação aberta para o cliente, a operação apenas retorna falso;
- **CongelaTransação**: essa operação adiciona uma marcação na transação para indicar que não poderá mais ser alterada. Qualquer operação em uma transação congelada gera um erro;
- **FinalizaTransação**: essa operação escreve as tuplas da lista de escrita no espaço de tuplas e retira as tuplas da lista de leitura do espaço. Libera o bloqueio de leitura das tuplas lidas por essa transação;
- **RollbackTransação**: essa operação descarta todas as tuplas que estão na lista de escrita. Além disso, todos os bloqueios realizados nessa transação são liberados;
- **Operações de out(t), inp(s) e rdp(s):** essas operações são melhor detalhadas nos parágrafos seguintes.

O Gerente de Bloqueio é responsável por garantir que as regras de bloqueio definidas na seção 3.4 sejam respeitadas. Ao realizar um bloqueio, deve haver um

cuidado para evitar que um cliente aguarde indefinidamente pelo desbloqueio de uma tupla. Como o bloqueio de tuplas é tratado para cada uma das operações, é descrito nos parágrafos seguintes.

A Figura 3.10 mostra como funciona o gerente de bloqueio quando recebe uma operação de out(tupla). Primeiramente, é gerado um identificador temporário para a tupla. O identificador é iniciado em zero para cada transação e é incrementado em uma unidade. A tupla é inserida na lista de escrita da transação. O espaço de tuplas propriamente dito não tem ainda conhecimento dessa tupla.

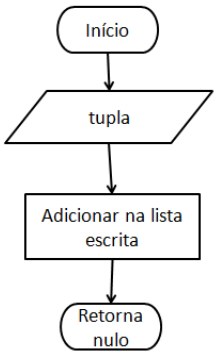


Figura 3.10: Operação de out(tupla) no Gerente de Bloqueio

A Figura 3.11 mostra como funciona o gerente de bloqueio quando recebe uma operação de inp(anti-tupla). Primeiramente, é verificado se o cliente já inseriu uma tupla que casa com a anti-tupla nessa transação. Caso, isso ocorra, a tupla com o menor identificador é retornada. Essa abordagem é usada para minimizar o número de tuplas bloqueadas, uma vez que as outras transações só terão ciência dessa tupla após a confirmação. Se a primeira tentativa não tiver sucesso, então é procurada uma tupla que ainda não foi bloqueada por outra transação. E por fim, se só houver tuplas bloqueadas que casam com a anti-tupla, o gerente de bloqueio inicia a contagem de um *timeout* para a transação que possui o bloqueio. Se a transação não terminar no tempo esperado, essa é abortada e a tupla é liberada para o cliente que a aguarda.

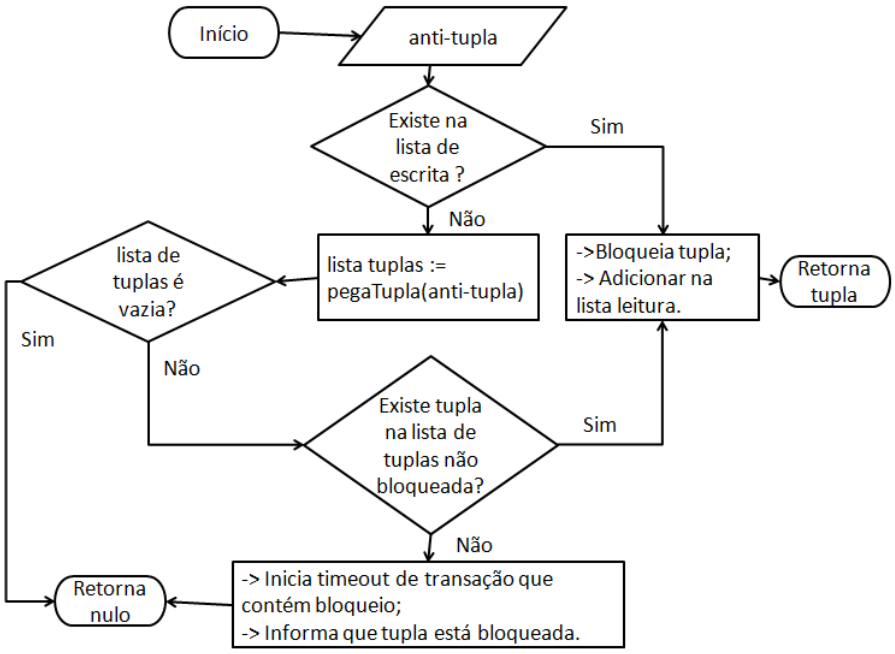


Figura 3.11: Operação de inp(anti-tupla) no Gerente de Bloqueio

A Figura 3.12 mostra como funciona o gerente de bloqueio quando recebe uma operação de rdp(anti-tupla). O funcionamento é bem similar ao da operação de inp(anti-tupla). A diferença entre as operações é que a tupla retornada pela operação de rdp(anti-tupla) fica disponível para outras transações realizarem a mesma operação (rdp(anti-tupla)).

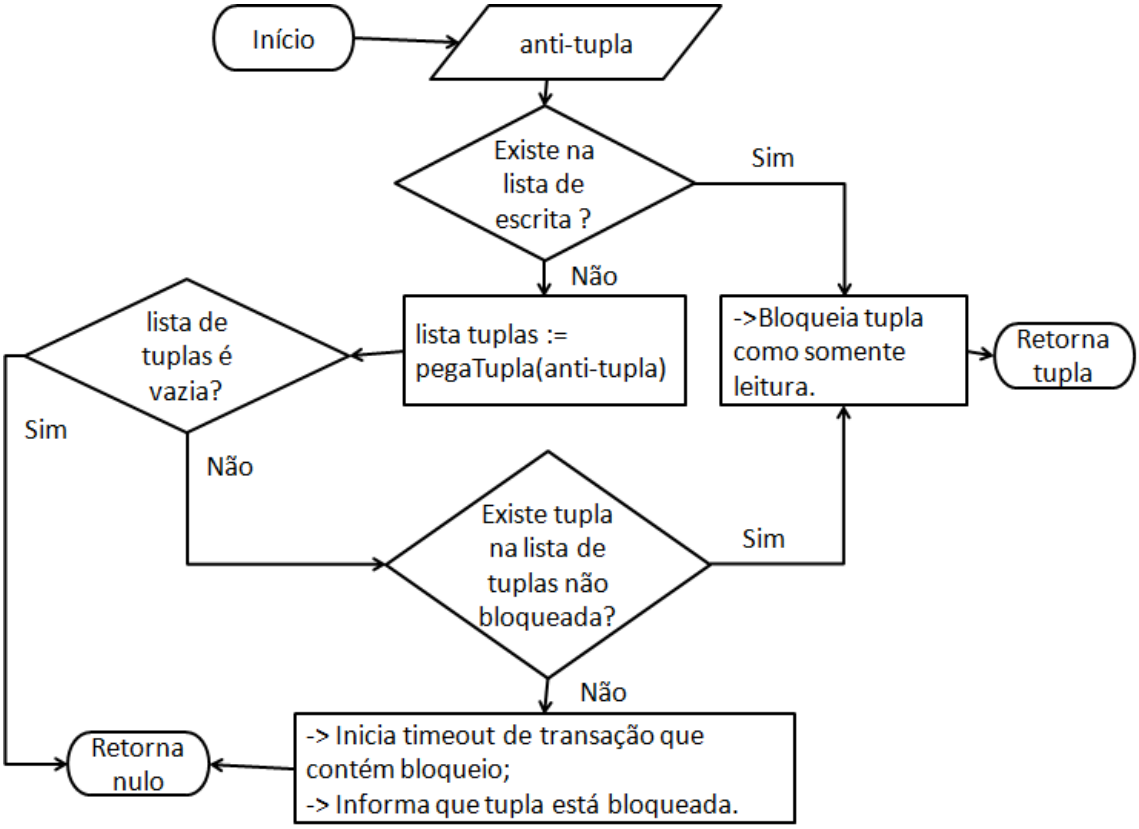


Figura 3.12: Operação de rdp(anti-tupla) no Gerente de Bloqueio

O Tuplebiz respeita as propriedades de vivacidade e segurança. Zyzzyva, o protocolo base do Tuplebiz, já possui essas propriedades (KOTLA ET AL., 2009). Logo, se faz necessário garantir que as mudanças realizadas no mesmo ainda mantêm as mesmas propriedades.

As operações realizadas no Tuplebiz fora do contexto transacional não afetam as propriedades já garantidas pelo Zyzzyva. Isso ocorre porque essas operações não afetam o comportamento padrão do Zyzzyva que já espera receber uma operação como parâmetro e não está relacionada à qualquer estado da réplica.

Entretanto, operações dentro do contexto transacional precisam passar pelo gerente de bloqueio, o qual pode realizar o bloqueio de uma tupla. Esse comportamento é similar ao encontrado em um bloco de exclusão mútua. Um bloco de exclusão mútua pode ser dividido em três partes: a parte de entrada, onde o cliente espera para acessar a seção crítica; a seção crítica e a parte de saída, onde o cliente sai da seção crítica. A primeira parte, no caso do gerente de bloqueio, acontece quando o cliente deseja acessar uma tupla do espaço de tuplas dentro de uma transação. Se só houverem tuplas bloqueadas que casem com a anti-tupla, o cliente espera uma resposta. Se não houver, o cliente entra na seção crítica, onde apenas ele terá acesso à tupla. A segunda parte é

quando a tupla pertence à transação. A terceira parte ocorre quando a transação é finalizada e o bloqueio é liberado.

O trabalho de (BESSANI, 2006) indica que a principal propriedade que o sistema deve possuir para garantir segurança é a exclusão mútua, a qual significa que não existe estado no sistema onde dois processos estão em uma seção crítica simultaneamente. O gerente de bloqueio garante essa propriedade mantendo a lista de bloqueio das tuplas em uso.

Ainda em (BESSANI, 2006), as propriedades importantes para atingir a vivacidade nesse tipo de sistema é o Progresso (*Deadlock-freedom*) e o Progresso justo (*Starvation-freedom*). Progresso significa que se um processo está na seção de entrada, algum processo termina por executar a sua seção crítica. Progresso justo significa se o processo está na seção de entrada, ele termina de executar sua seção crítica.

O progresso é garantido no Tuplebiz através do uso de *timeouts*. Se um cliente A mantém o bloqueio de uma tupla e outro cliente B também necessita da mesma tupla, um *timeout* é iniciado indicando o tempo que o cliente A ainda possui para finalizar a transação. Se dentro desse período a transação não for concluída, a mesma sofre um aborto.

O progresso justo é garantido pelo Tuplebiz através do uso de *timeouts* e filas de espera. Com base no exemplo do parágrafo anterior (onde um cliente A mantém o bloqueio de uma tupla e outro cliente B também necessita da mesma tupla), o cliente B recebe uma mensagem para tentar o acesso posteriormente. O tempo indicado para espera é superior ao *timeout* dado ao cliente A. O cliente B possui uma janela de tempo no qual tem preferência ao acesso. Após essa janela, a tupla fica disponibilizada para outros clientes que desejam acessá-la.

3.7 Considerações finais

O Tuplebiz se mostra um sistema poderoso para tratar falhas bizantinas e transações. Baseia-se em um protocolo de tolerância a falhas bem definido e com bom desempenho quando comparado aos demais da literatura. Ainda respeita as propriedades de vivacidade (*liveness*) e segurança (*safety*).

Visando um melhor desempenho para aplicações distribuídas, o Tuplebiz é configurado em partições. O sistema pode ter diferentes partições distribuídas, ao longo de vários servidores.

Como praticamente todos os sistemas que usam consenso de réplicas como peça chave, o Tuplebiz apresenta uma queda de desempenho quando comparado a um sistema não replicado. Isso se deve ao maior número de passos para realizar uma operação. Por outro lado, o fato de tolerar falhas bizantinas compensa a queda de desempenho do sistema. Com o intuito de mensurar essa queda de desempenho, no capítulo 4 é feita uma comparação de desempenho entre o Tuplebiz e um espaço de tuplas não replicado sem qualquer tipo de tolerância a falhas. Além disso, são realizados testes de injeção de falhas com o intuito de validar o suporte a falhas provido pelo Tuplebiz.

Tendo em vista as diferentes aplicações que podem fazer uso do Tuplebiz, o capítulo 5 mostra como uma delas, a Guaraná, pode ser beneficiada com o uso do Tuplebiz para comunicação entre as tarefas.

4 AVALIAÇÃO

Nesse capítulo o Tuplebiz é avaliado em relação à quantidade de passos de execução, ao desempenho, à injeção de falhas e aos trabalhos relacionados. Primeiramente, são avaliados quantos passos de comunicação são necessários para executar cada uma das operações do Tuplebiz.

A avaliação de desempenho é feita comparando um sistema não replicado e o Tuplebiz. A replicação, certamente, afeta o desempenho, logo, a ideia é mensurar esse impacto no desempenho.

Na avaliação de injeção de falhas são inseridas falhas no Tuplebiz durante sua operação. Nesse tipo de teste são avaliadas duas propriedades: (1) se o sistema consegue tolerar a falha e (2) o desempenho em caso de falhas.

E por fim, o Tuplebiz é comparado com trabalhos existentes na literatura. Não foi publicado até então um trabalho que possua todas as características existentes no Tuplebiz (um espaço de tuplas com suporte a transações em um ambiente sujeito a falhas bizantinas) pelo conhecimento do autor. Então, as comparações são realizadas com espaços de tuplas que suportam falhas bizantinas, espaços de tuplas que suportam transações e outros tipos de sistemas (que não são relacionados a espaço de tuplas) que suportam tanto transações quanto falhas bizantinas.

4.1 Avaliação da troca de mensagens

A avaliação é feita considerando os passos de comunicação, ou seja, a quantidade de mensagens trocadas entre os nodos. O Tuplebiz executa diferentes números de passos de acordo com a operação a ser realizada e da existência de falhas. Todas as operações, exceto a de *transaction_commit* terminam após cinco passos quando se trata de uma execução otimista (sem presença de falhas). As mensagens podem ser vistas na Figura 4.1. O exemplo da figura mostra uma instanciação do Tuplebiz com apenas uma partição. Se forem utilizadas mais partições, o número de mensagens será proporcional ao número de réplicas de cada partição, ou seja, o número de mensagens deve ser acrescido de duas vezes o número de novas réplicas. O número de passos permanece igual, pois as mensagens são enviadas paralelamente para cada partição. Em caso de falhas é executado o protocolo de mudança de *view*, o qual pertence ao Zyzzyva (KOTLA ET AL., 2009).

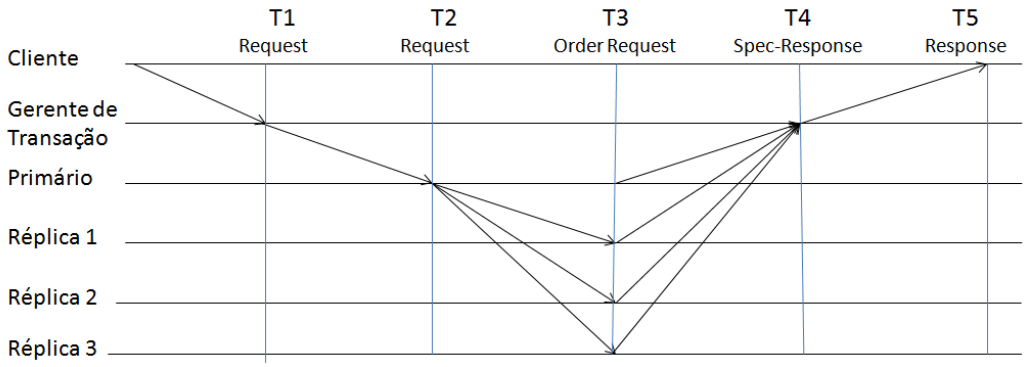


Figura 4.1: Mensagens trocadas no Tuplebiz para operações (exceto *transaction_commit*)

A operação de *transaction_commit* necessita de pelo menos dez passos para ser executada. A Figura 4.2 mostra os passos de comunicação. O cliente envia a mensagem para o gerente de transações e esse por sua vez, envia uma mensagem para cada partição. Dentro de cada partição o nodo primário repassa a mensagem para os nodos secundários. A comunicação entre o primário e os secundários é vista com detalhes na Figura 4.2.

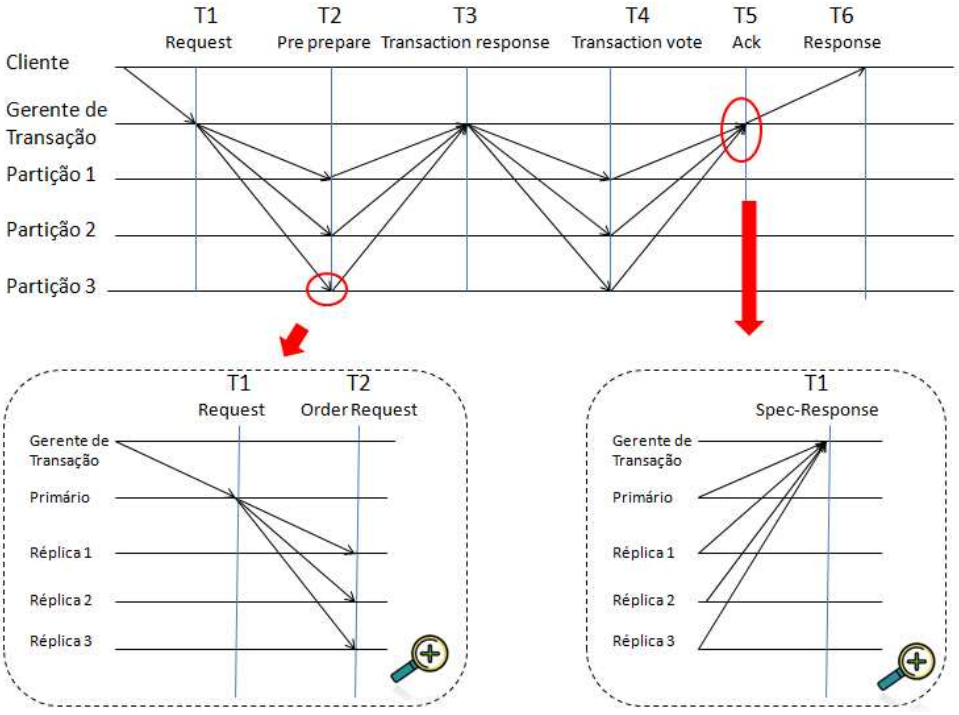


Figura 4.2: Mensagens trocadas no Tuplebiz para operação de *transaction_commit*

4.2 Protótipo

O protótipo foi desenvolvido em Java. A escolha dessa linguagem se deve ao fato da Guaraná também ser desenvolvida em Java e por ter bibliotecas de segurança e banco de dados.

Os sub-protocolos de *agreement* e *view change* Zyzzyva foram implementados no protótipo. O histórico das mensagens é salvo periodicamente em disco. O protótipo tem aproximadamente nove mil linhas de código desconsiderando os comentários.

Para a comunicação são usados *sockets* TCP. A escolha desse meio de comunicação se deve pela sua flexibilidade de uso. Para que seja executada uma aplicação que faz uso de *sockets*, apenas a *Java virtual machine* (JVM) precisa estar disponível no computador em questão e nenhuma configuração extra é necessária. O uso de outras ferramentas como *Remote Method Invocation* (RMI) e *Enterprise Java Beans* (EJB) exige instalações e/ou configurações extras. Para ser utilizado, EJB, por exemplo, é necessário que seja utilizado um servidor de aplicações (exemplo JBoss). O TCP foi utilizado ao invés do UDP por garantir entrega e ordem.

O cálculo do resumo de mensagem, bem como a assinatura da mensagem, é feito utilizando o pacote *java.security*. A assinatura das mensagens usa o método Digital Signature Algorithm (DSA) é um padrão do governo federal norte-americano para assinatura digital (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2009). O cálculo do resumo das mensagens utiliza *Message-Digest algorithm 5* (MD5) definido pela RFC 1321.

O histórico das mensagens, bem como as tuplas, é armazenado em um banco de dados em memória, o HSQL (HSQL, 2011). Os dados contidos no banco de dados são periodicamente armazenados em disco. A vantagem de usar um banco de dados para armazenar as tuplas e o histórico é retirar a responsabilidade de armazenamento do código. Manter toda estrutura de dados em memória pode causar estouro de memória, logo, é necessário manter uma estrutura para gerenciar dados. Com o uso de banco de dados em memória, a responsabilidade de gerenciar os dados é do próprio banco de dados. Além disso, HSQL já tem uma implementação *multi-thread* para prover melhor desempenho.

O protótipo da réplica segue o mesmo modelo descrito no capítulo 3. Não houve nenhuma otimização ou mudança.

Um protótipo não replicado foi desenvolvido para realização dos testes de desempenho. Esse usa exatamente o mesmo espaço de tuplas do Tuplebiz. Isso é possível porque o protótipo do Tuplebiz é modular. A principal diferença entre os protótipos é que enquanto o Tuplebiz tem uma instância de espaço de tuplas em cada réplica, o espaço de tuplas não replicado contém apenas uma instância para todo o sistema.

A comunicação entre os nodos pode ser vista na Figura 4.3. A comunicação entre o processo e o cliente é feita através de uma chamada local de método. Nessa situação não é calculado o resumo da mensagem e nem a mensagem é assinada digitalmente. Já a comunicação entre o cliente e a réplica é feita através de *socket* TCP. Todas as mensagens passadas através dos *sockets* têm seu resumo calculado, bem como são assinadas digitalmente. O formato das mensagens é o mesmo para o Tuplebiz e o Tuplebiz NoRep. O Tuplebiz NoRep não armazena histórico de mensagens e, portanto, não faz *checkpoint* do mesmo.

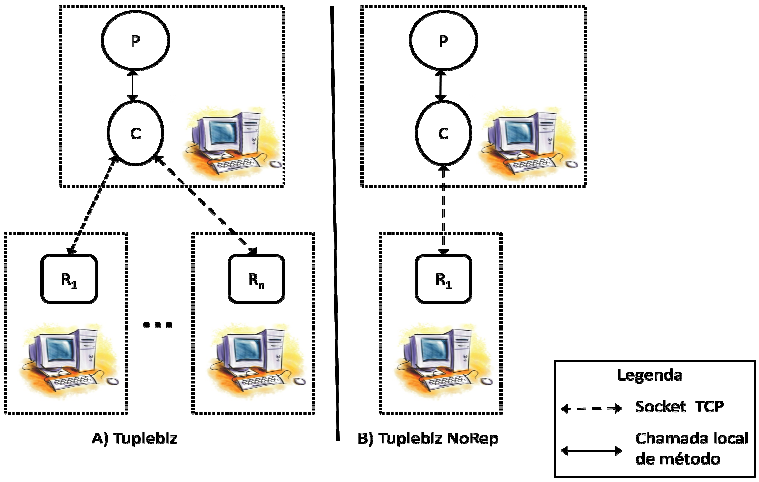


Figura 4.3: Comunicação entre os nodos durante o teste. a) Tuplebiz Replicado b) Tuplebiz não replicado

4.3 Avaliação de desempenho

Certamente a arquitetura resiliente a falhas diminui o desempenho geral do sistema, entretanto, aumenta a confiabilidade. Nesse caso, os testes de desempenho servem para mensurar quanto se perde em desempenho para garantir tolerância a falhas. Como a arquitetura do Tuplebiz permite que seja utilizado o espaço de tuplas sem replicação, a comparação de desempenho é feita usando um protótipo do Tuplebiz não replicado (NoRep) e um protótipo do Tuplebiz replicado (Tuplebiz). As réplicas no Tuplebiz são idênticas e executam as mesmas operações em todas elas. Logo, existe um custo de sincronização do algoritmo a fim que a resposta correta seja devidamente entregue ao cliente. No caso do sistema não replicado, só um servidor recebe a requisição, a executa e envia a resposta para o cliente. Não há qualquer algoritmo para sincronizar respostas, o qual é custoso quando se trata de troca de mensagens.

Para definir os testes realizados para a avaliação de desempenho, foram pesquisados *benchmarks* de diferentes áreas. Dentre os *benchmarks* avaliados estão os relacionados aos espaços de tuplas, transações de banco de dados e protocolos de consenso para falhas bizantinas. Esses tipos de *benchmarks* foram escolhidos por se tratarem de tópicos utilizados nessa dissertação: espaço de tuplas, transações e protocolo de consenso.

Em (NOBLE; ZLATEVA, 2001) é especificado um *benchmark* que avalia o JavaSpace (SUN MICROSYSTEMS, 1999) para computação paralela. Esse trabalho pode ser dividido em duas abordagens: um *microbenchmark* das operações básicas do JavaSpace e testes de desempenho para computação paralela. O grupo de testes para computação paralela não é considerado, pois essa não é a aplicação alvo desse trabalho. O *microbenchmark* que trata apenas de operações básicas é usado para definição dos testes.

O SETTLE (FIEDLER ET AL., 2005) é um *benchmark* para avaliação de desempenho em espaço de tuplas. Ele estressa o uso de operações simples como *in(s)* e *out(t)* na sua avaliação. Entretanto, ele não pôde ser diretamente usado nos testes por não avaliar transações. Algumas ideias de validação presentes nele, como por exemplo, as fases de execução foram usadas na definição dos testes.

Os trabalhos de (VANDIVER; BALAKRISHNAN; LISKOV, 2007) e (LUIZ; LUNG; CORREIA, 2011) versam sobre transações de banco de dados num ambiente sujeito a falhas bizantinas. Ambos fazem uso do TPC-C (TPC, 2011) como *benchmark* para avaliação de desempenho. O TPC-C faz uso de aplicações que estressam banco de dados. Esse *benchmark* não pode ser utilizado diretamente no contexto do Tuplebiz, pois é especialmente desenvolvido para banco de dados.

Os testes realizados em (CASTRO; LISKOV, 2002) e (KOTLA ET AL., 2009) avaliam o desempenho de algoritmos de consenso. Neles são avaliados os seguintes itens: latência, *taxa de transferência*, configuração aumentando o número de réplicas, gerenciamento de *checkpoint*, mudanças de *view* e *microbenchmarks* de avaliação de sistema de arquivos. Nem todas as medições por eles apresentadas são interessantes para avaliar o Tuplebiz. Gerenciamento de *checkpoint*, bem como mudança de *view*, é ligado diretamente ao protocolo de consenso e, como o Tuplebiz é baseado no protocolo Zyzzyva (KOTLA ET AL., 2009), esses testes não se fazem necessários. Por não se tratar de um sistema de arquivos, os *microbenchmarks* de sistemas de arquivos não são avaliados.

Considerando os tipos de teste que podem ser realizados, os mais relevantes para esse trabalho são os testes de latência, os quais foram aplicados tanto no sistema replicado quanto o não replicado.

4.3.1 Metodologia

O ciclo de teste, que pode ser visto na Figura 4.4, prevê um tempo de ação e um tempo de limpeza de ação, como o do (FIEDLER ET AL., 2005). O tempo de ação é utilizado para pré-popular o espaço de tuplas. Esse tempo de ação é necessário para operações como in(s), a qual necessita que a tupla já esteja presente no espaço de tuplas. Para cada teste realizado são definidas, então, algumas operações no período de ação e limpeza de ação.

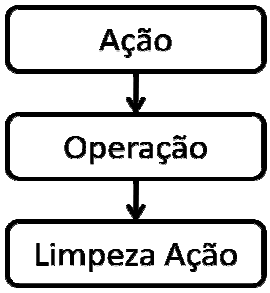


Figura 4.4: Passos do ciclo de teste

A quantidade de repetições é baseada no modelo estatístico definido em (LILJA, 2004), o qual considera que os erros possuem um comportamento gaussiano para um número grande de medidas (para número de medidas maior ou igual a 30), isso porque as medidas têm a tendência de estarem centradas em uma mesma média.

A Figura 4.5 mostra como a medida é definida. Assume-se que a média $\bar{X}$ é a melhor aproximação de n medidas. O intervalo entre c1 e c2 é o intervalo de confiança e α é o nível de significância. A equação (4.1) é usada para se definir o número de medidas necessárias para atingir o intervalo de confiança requerido, onde *n* é o número de testes, *s* o desvio padrão, *e* o erro, $Z_{1-\alpha/2}$, valor unidade padrão da distribuição normal que tem uma área $1-\alpha/2$ à esquerda de $Z_{1-\alpha/2}$. É necessário saber a média ($\bar{X}$) e o desvio padrão (s) para definir o número de medidas, logo, faz-se necessário realizar um pequeno

número de amostras para estimar a média e o desvio padrão. Para os testes aqui realizados foram pegas 30 amostras para estimar a média e desvio padrão. Nesse trabalho, o intervalo de confidência escolhido foi de 95% e o erro de 1%, logo α é 0,05 e *e* é 0,01.

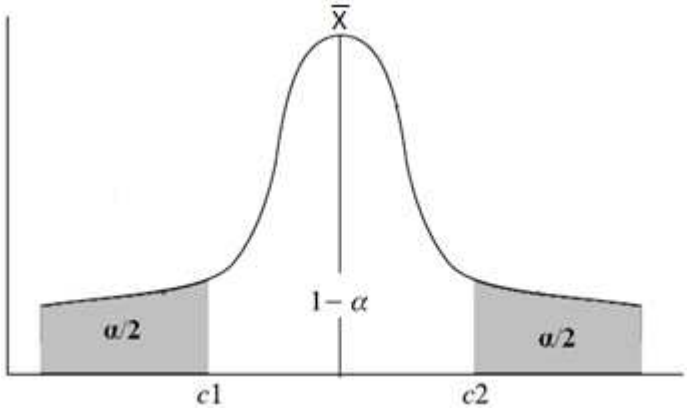


Figura 4.5: Probabilidade de o verdadeiro valor medido estar no intervalo c1 e c2. Adaptado de (LILJA, 2004).

$$n = \left(\frac{Z_{1-\alpha/2}s}{e\bar{X}}\right)^2 \quad (4.1)$$

No caso do Tuplebiz a latência é medida verificando o mesmo conjunto de operações, dentro e fora do contexto transacional. As operações realizadas são out(a), rdp(a?) e inp(a?). Cada uma dessas operações é executada separadamente completando um ciclo inteiro de teste, isso é, são executadas as etapas de ação, operação e limpeza de ação (ver cada ciclo em Tabela 4.1). O volume de dados usados nos testes é definido em (KOTLA ET AL., 2009). Os argumentos "a" e "a?" vão variar de tamanho de acordo com os argumentos de entrada e resultado requeridos na Tabela 4.2, a qual mostra um ciclo de teste em cada linha.

Tabela 4.1: Ciclos de teste para avaliação de latência

Id	Etapa de Ação	Operação	Etapa de Limpeza
1	Sem ação	out(a)	Excluir todas as tuplas que casam com a anti-tupla "a?"
2	Inserir n tuplas que casam com "a?", onde n é o número de repetições que devem ser feitas.	inp(a?)	Sem ação
3	Sem ação	inp(a?)	Sem ação
4	Inserir uma tupla que casa com "a?"	rdp(a?)	Sem ação
5	Sem ação	rdp(a?)	Sem ação

Tabela 4.2: Dados para teste de latência

Número de Clientes	Tamanho do argumento de entrada (bytes)	Tamanho do argumento de saída (bytes)
1	0	0
1	0	2000
1	0	4000
1	0	6000
1	0	8000
1	0	0
1	2000	0
1	4000	0
1	6000	0
1	8000	0

4.3.2 Ambiente de teste

Um conjunto de seis computadores foi usado durante os testes. A configuração das máquinas usadas se encontra na Tabela 4.3. Todas as máquinas possuem a mesma versão de Java (Java 1.6.0_27) e a comunicação entre as máquinas usa um link de 100 Mbps, em uma LAN com um *hop*. Testes de latência para espaços de tuplas tolerantes a falhas bizantinas possuem configurações similares.

Tabela 4.3: Descrição das máquinas de teste

Plataforma	Memória	Processador
Linux - Ubuntu 10.04.1 LTS	4GB	Intel Xeon 2GHz
Linux - Ubuntu 11.04	4GB	Intel Core2 Quad CPU 2.33GHz
Linux - Ubuntu 11.04	3GB	Intel Core2 Quad CPU 2.40GHz
Linux - Ubuntu 11.04	4GB	Intel Core2 Quad CPU 2.33GHz
Linux - Ubuntu 11.04	4GB	Intel Core2 Quad CPU 2.33GHz
Linux - Ubuntu 11.04	4GB	Intel Core2 Quad CPU 2.33GHz

4.3.3 Resultados

Foram realizados 46 casos de testes de latência. Para cada um dos cinco ciclos de testes da Tabela 4.1 foram variadas as entradas e saídas conforme definido na Tabela 4.2. A configuração do Tuplebiz utilizada foi de uma partição com quatro nodos. Os testes foram aplicados fora e dentro do contexto transacional a fim de medir o impacto do gerente de bloqueio sobre as operações.

Os resultados apresentados são comparados com o espaço de tuplas não replicado. Também, é feita uma comparação com o DepSpace (BESSANI ET AL., 2008) que apresenta um espaço de tuplas com tolerância a falhas bizantinas. Importante salientar, que o tamanho dos dados usados nos testes, bem como o espaço de tuplas não replicado, é diferente do apresentado neste trabalho e no trabalho do DepSpace. A comparação se dá através da comparação de quantas vezes o Tuplebiz é mais lento que o NoRep contra quantas vezes DepSpace é mais lento que o Gigaspace.

As Tabela 4.4 e Tabela 4.5 mostram os resultados dos testes de out(t) fora do contexto transacional e dentro do contexto transacional, respectivamente. O número de repetições é o número de iterações indicado após a execução da amostragem de 30 iterações.

Tabela 4.4: Resultados dos testes de out(a) fora do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
out	0	0	3567	24.08	6.12	4019	8.58	2.50
out	2000	0	1885	25.09	6.85	3303	8.72	2.51
out	4000	0	2591	25.05	6.79	3382	8.77	2.73
out	6000	0	1819	25.62	7.60	3686	8.83	2.57
out	8000	0	1595	26.13	7.38	4915	8.76	2.47

Tabela 4.5: Resultados dos testes de out(a) dentro do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
out	0	0	104	25.74	5.54	210	9.25	2.14
out	2000	0	206	24.86	4.35	229	9.33	1.95
out	4000	0	166	25.34	5.12	255	9.33	2.26
out	6000	0	217	25.47	5.44	264	9.50	2.22
out	8000	0	249	25.43	4.51	298	9.52	2.66

A Figura 4.6 mostra a comparação entre o Tuplebiz e o sistema não replicado para operação de out(t). O Tuplebiz leva aproximadamente 2,8 vezes mais tempo que o modelo não replicado para executar. Para o sistema não replicado o tamanho do argumento tem um efeito mínimo sobre o desempenho. Para o Tuplebiz, nota-se que a latência aumenta com o aumento do argumento usado. O DepSpace (BESANNI, 2008) que apresenta um espaço de tuplas com tolerância a falhas bizantinas, a operação de out(t) é aproximadamente 4 vezes mais lenta que em um espaço não replicado.

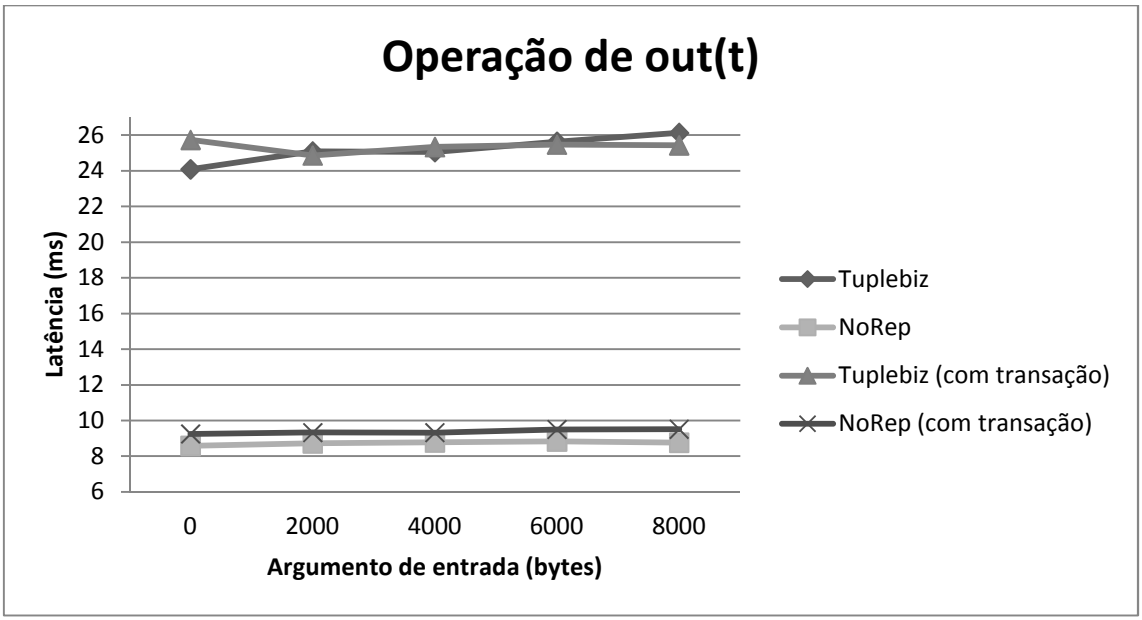


Figura 4.6: Operação de out(t)

Os resultados dos testes de inp(s) são similares ao da operação de out(t). A Tabela 4.6 e a Tabela 4.7 mostram os resultados para as operações de inp(s) fora e dentro do contexto transacional respectivamente. O formato da tabela é o mesmo utilizado nas operações de out(t).

Tabela 4.6: Operações de inp(s) fora do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
inp(s)	0	0	3263	24.43	6.42	4588	8.62	2.55
inp(s)	2000	0	1772	25.37	7.38	2606	9.07	2.61
inp(s)	4000	0	2093	25.59	7.45	3731	8.82	2.72
inp(s)	6000	0	2128	25.78	7.21	2555	9.16	2.58
inp(s)	8000	0	1990	25.83	8.08	3906	8.94	2.50
inp(s)	0	2000	572	25.86	5.71	320	10.16	3.10
inp(s)	0	4000	1083	24.97	6.36	1575	9.01	2.19
inp(s)	0	6000	858	25.38	5.03	514	9.45	1.98
inp(s)	0	8000	3071	24.47	6.91	4207	8.63	2.53

Tabela 4.7: Operações de inp(s) dentro do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
inp(s)	0	0	308	25.64	4.38	265	10.76	2.52
inp(s)	2000	0	204	26.98	6.66	215	11.17	2.52
inp(s)	4000	0	197	27.27	6.22	208	10.99	2.31
inp(s)	6000	0	265	26.51	4.77	314	10.57	2.07
inp(s)	8000	0	140	28.67	7.36	378	10.59	2.33
inp(s)	0	2000	153	26.95	6.56	202	10.52	1.94
inp(s)	0	4000	160	26.69	5.95	173	11.36	2.80
inp(s)	0	6000	217	26.02	5.27	175	10.93	2.41
inp(s)	0	8000	191	26.04	5.39	153	11.25	2.42

A Figura 4.7 mostra a operação de inp(s) que não retorna uma tupla, mas que tem anti-tuplas de tamanhos que variam de 0bytes até 8Kbytes. A Figura 4.8 mostra a operação de inp(s) que retorna tuplas que variam de 0bytes até 8Kbytes e em que a anti-tupla é vazia. Os valores aqui apresentados são bem similares aos vistos na operação de out(t). O Tuplebiz é em média 2,8 vezes mais lento que o modelo não replicado. No DepSpace a operação de inp(s) é aproximadamente 4 vezes mais lenta que no Gigaspace.

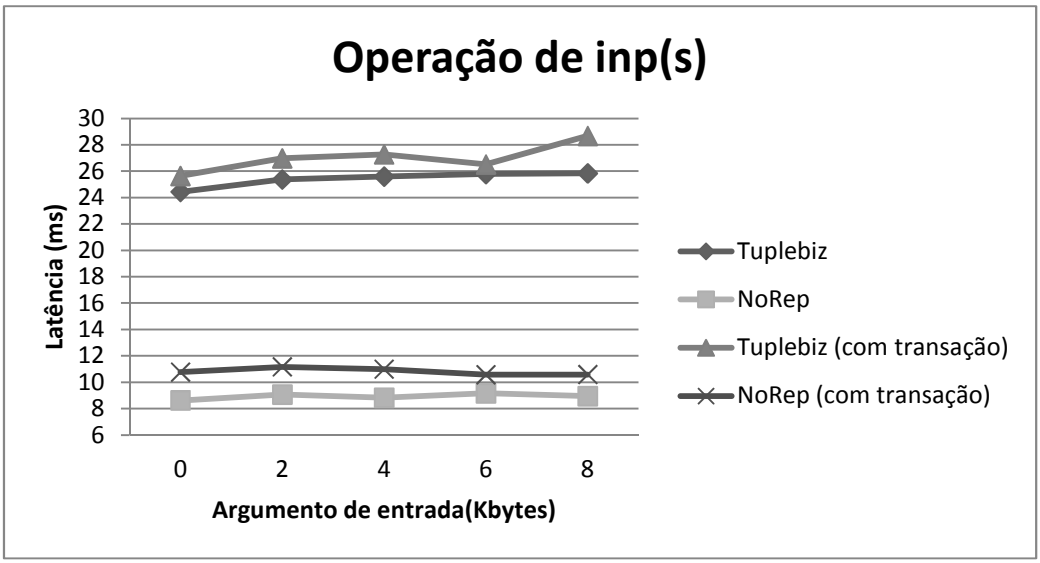


Figura 4.7: Operação de inp(s) variando o argumento de entrada

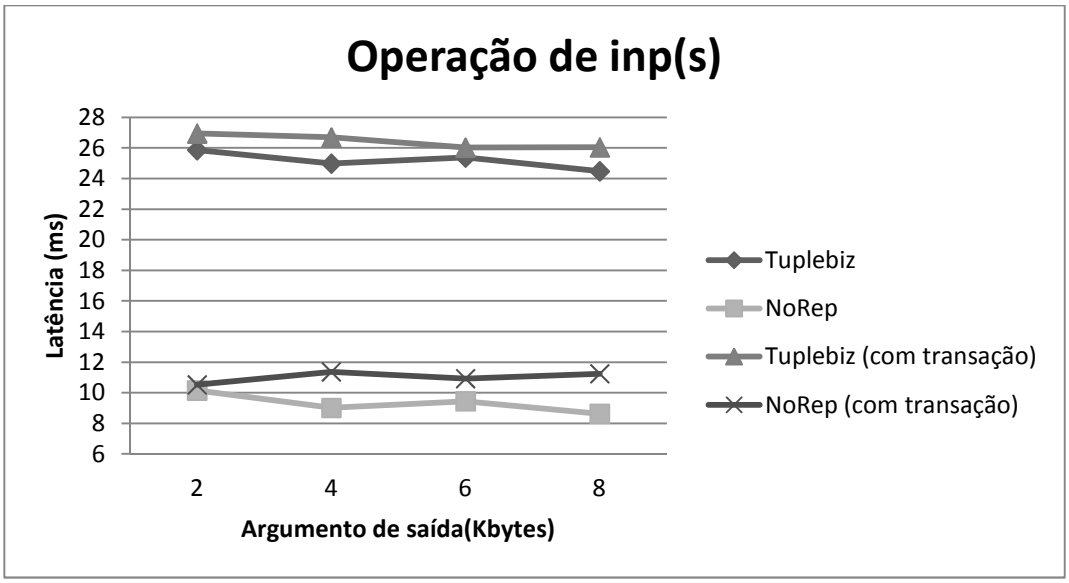


Figura 4.8: Operação de inp(s) variando o argumento de saída

As Tabela 4.8 e Tabela 4.9 mostram os resultados dos testes para a operação de rdp(s). Essa operação apresenta valores bem similares às demais operações. Esse resultado é esperado, pois não há implementações distintas para cada de tipo de operação. O número de mensagens é o mesmo independente da operação a ser realizada.

Tabela 4.8 : Operações de rdp(s) fora do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
rdp(s)	0	0	2708	24.68	6.86	4646	8.59	2.42
rdp(s)	2000	0	1873	25.33	6.33	3502	8.80	2.46
rdp(s)	4000	0	2071	25.52	7.75	3442	8.89	2.70
rdp(s)	6000	0	2165	25.65	7.39	2494	9.23	2.86
rdp(s)	8000	0	1728	26.15	8.14	3671	8.91	2.52
rdp(s)	0	2000	381	26.65	6.42	4207	8.63	2.53
rdp(s)	0	4000	966	25.16	6.54	319	10.06	2.56
rdp(s)	0	6000	518	25.86	6.80	1252	9.13	2.31
rdp(s)	0	8000	3484	24.43	8.85	330	9.94	2.16

Tabela 4.9: Operações de rdp(s) dentro do contexto transacional

Descrição			Resultado					
			Tuplebiz			NoRep		
Operação	Entrada (bytes)	Saída (bytes)	Número de repetições	Média	Desvio Padrão	Número de repetições	Média	Desvio Padrão
rdp(s)	0	0	150	27.52	4.95	241	11.20	2.40
rdp(s)	2000	0	204	27.51	4.67	221	11.56	2.86
rdp(s)	4000	0	211	27.61	5.37	304	11.20	2.80
rdp(s)	6000	0	217	27.78	5.60	205	12.08	3.57
rdp(s)	8000	0	104	29.71	7.35	218	12.32	3.22
rdp(s)	0	2000	150	27.52	4.95	241	11.20	2.40
rdp(s)	0	4000	204	27.51	4.67	221	11.56	2.86
rdp(s)	0	6000	211	27.61	5.37	304	11.20	2.80
rdp(s)	0	8000	217	27.78	5.60	205	12.08	3.57

A Figura 4.9 mostra a operação de rdp(s) que não retorna uma tupla, mas que tem anti-tuplas de tamanhos que variam de 0bytes até 8Kbytes. A Figura 4.10 mostra a operação de rdp(s) que retorna tuplas que variam de 0bytes até 8Kbytes e em que a anti-tupla é vazia. Assim como nas demais operações o Tuplebiz é 2,8 vezes mais lento que o espaço não replicado. O DepSpace possui um melhor desempenho nesse caso. O DepSpace possui a mesma latência que o Gigaspace. Isso se deve ao fato do DepSpace ter um algoritmo diferente para cada operação.

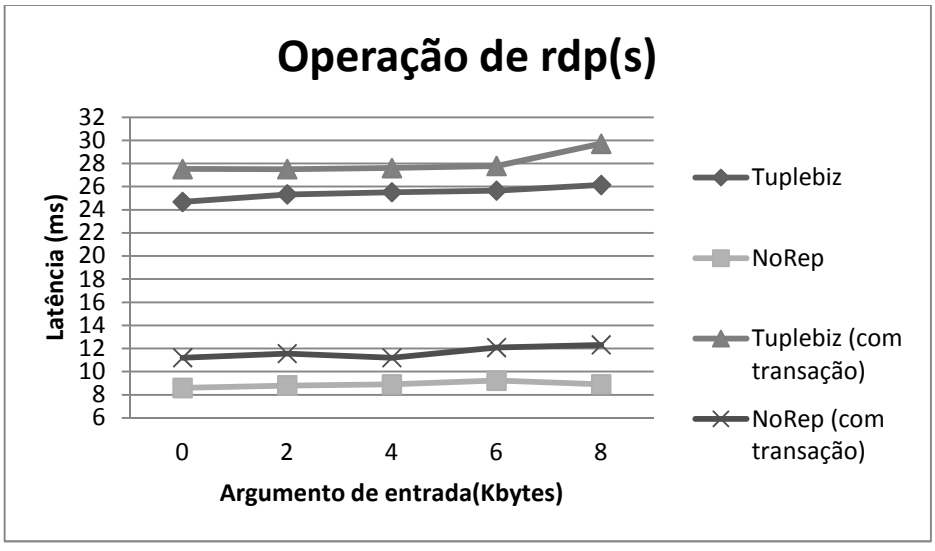


Figura 4.9: Operação de rdp(s) variando o argumento de entrada

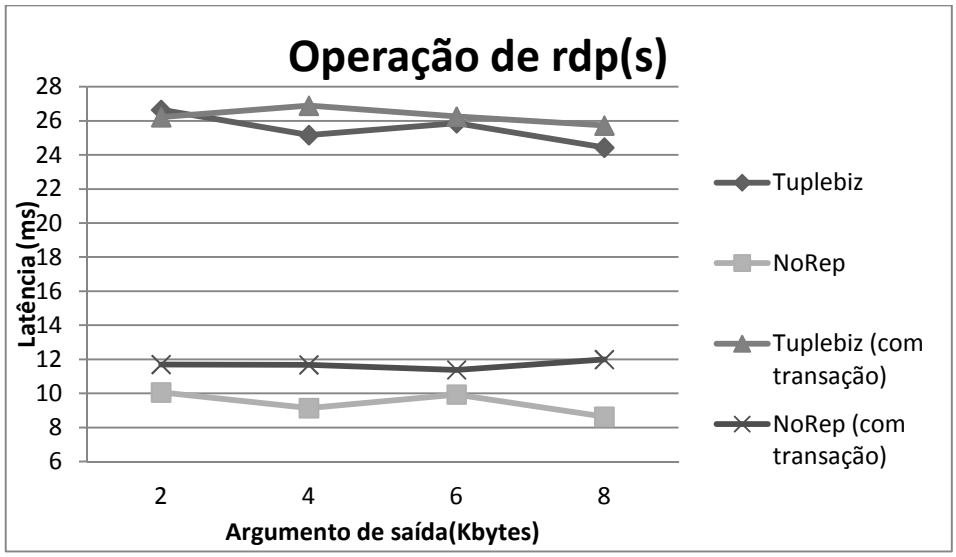


Figura 4.10 : Operação de rdp(s) variando o argumento de saída

4.4 Teste de injeção de falhas

Esse tipo de teste visa verificar a robustez do sistema na presença de falhas. Apesar de existirem diversos trabalhos nessa área, esse tipo de teste não é muito praticado em sistemas que toleram falhas bizantinas. Em (WIERMAN; NARASIMHAN, 2008) são listados diversos sistemas que toleram falhas bizantinas, mas que não realizam qualquer teste de injeção para avaliação do sistema.

As falhas bizantinas englobam uma variedade de falhas, logo é necessário definir um conjunto de falhas que devem ser injetadas. O trabalho de (WIERMAN; NARASIMHAN, 2008) sugere a injeção das seguintes falhas para teste de sistemas que toleram falhas bizantinas:

- Mensagem perdida – consiste em não enviar a mensagem, mas enviar confirmação de envio para remetente;
- Atraso de envio de mensagem – consiste em enviar a mensagem mais tarde, mas retornar sucesso;
- Corrupção de mensagem – consiste em modificar conteúdo da mensagem;
- Suspensão do sistema – o sistema parar de responder;
- *Crash* – o sistema interrompe a execução;
- Estouro de memória – o sistema ocupa mais memória que a disponível para uso.

O Tuplebiz foi testado usando todas as falhas injetadas acima, exceto a de estouro de memória. Essa última foi excluída por causar um efeito similar à falha de *crash* em Java. Quando o sistema estoura a memória, uma exceção é lançada e a execução do sistema é interrompida.

Foi avaliado o desempenho na presença das falhas e comparado com os testes sem presença de falhas. Os casos de testes são os mesmos enumerados na seção de teste de desempenho.

Para realizar a injeção das falhas, foram adicionados interceptores na camada de comunicação, com exceção da falha de *crash* que simplesmente para de executar o nodo. A Tabela 4.10 mostra a ação de cada interceptor. Ao iniciar a execução do sistema são definidos quando os gatilhos devem ser disparados para realizar a ação do interceptor. Esses gatilhos são executados periodicamente, ou seja, os intervalos onde as falhas devem ocorrer são previamente definidos.

Tabela 4.10: Relação entre a falha injetada e a ação do interceptor

Falha injetada	Ação do interceptor
Mensagem perdida	Remove a mensagem do buffer de saída.
Atraso de envio de mensagem	Armazena a mensagem por um tempo pré-definido e a envia depois.
Corrupção de mensagem	Altera o resumo da mensagem.
Suspensão do sistema	Mantém o sistema aguardando o envio da mensagem por um tempo pré-definido.

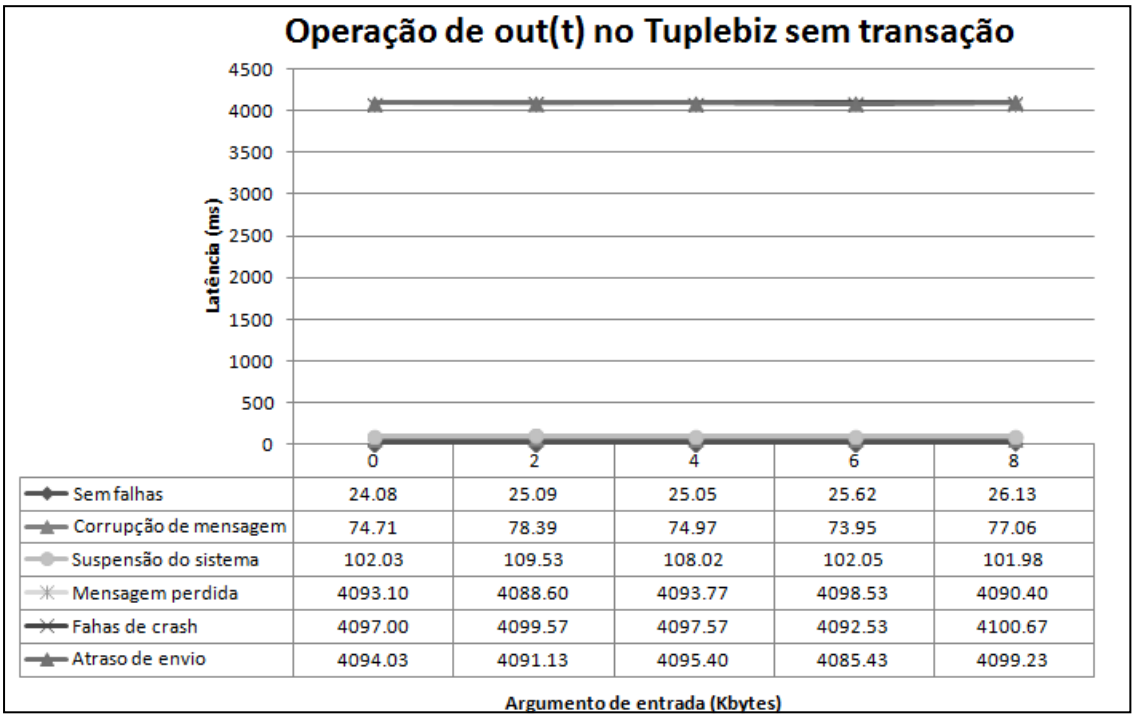
4.4.1 Resultados

Para todos os testes de injeção foram utilizadas uma partição com quatro réplicas. Com esse número de réplicas é suportado a falha de um nodo.

Para os testes de corrupção de mensagem, suspensão do sistema e atraso no envio foram inseridas falhas a cada 15ms em todas as réplicas. No teste de suspensão do sistema, o sistema ficava bloqueado por 20ms. No teste de *crash*, uma réplica do sistema foi desligada. No teste de mensagem perdida as mensagens foram excluídas do *buffer* a cada 400ms para uma réplica.

A Figura 4.11 mostra o resultado para a operação de out(t) fora da transação. Nota-se que para os testes de mensagem perdida e *crash* o tempo de processamento foi praticamente os mesmos em todas as execuções. Esse tempo é diretamente ligado ao *timeout* de espera de mensagem. Como a resposta especulativa não acontece, pois sempre há um nodo que não responde, o cliente executa o protocolo de *local_commit*. O cliente manda uma mensagem de *commit* com o certificado para as réplicas e essas por sua vez respondem com uma resposta de *local_commit*.

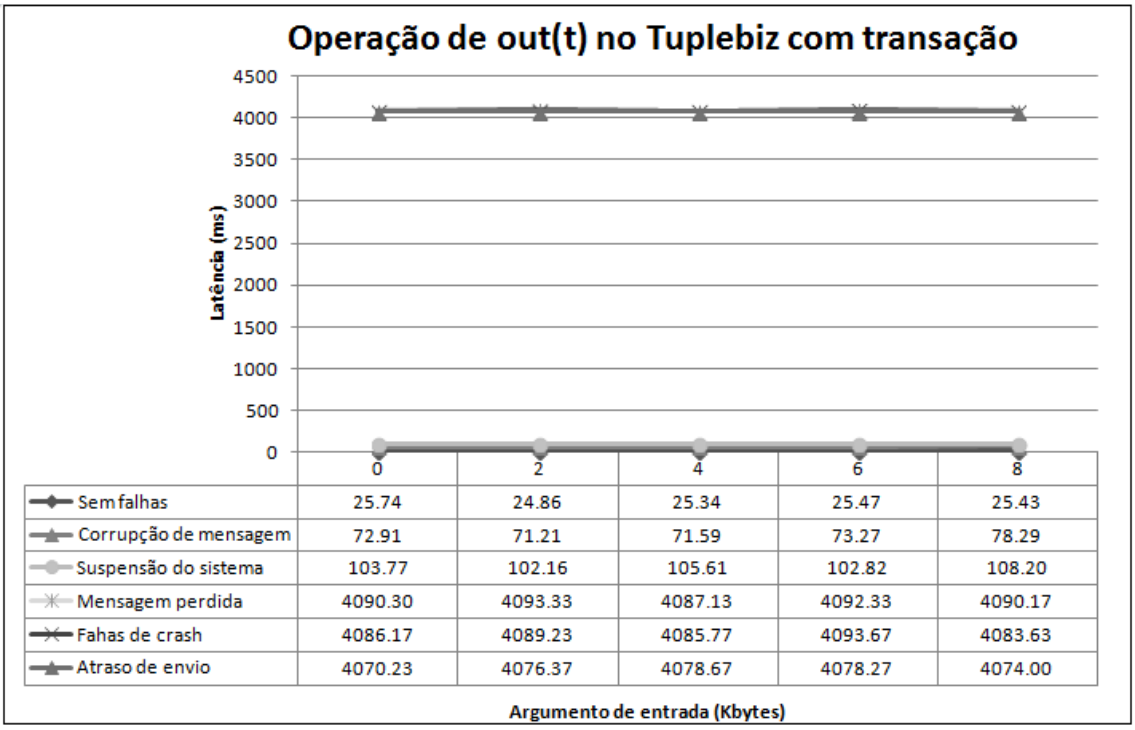
Para os testes de corrupção de mensagem e suspensão do sistema, o tempo foi bem superior ao da execução normal. A causa provável foi o reenvio de mensagens após verificação errônea da assinatura e dos parâmetros da mensagem.



	0	2	4	6	8
Sem falhas	24.08	25.09	25.05	25.62	26.13
Corrupção de mensagem	74.71	78.39	74.97	73.95	77.06
Suspensão do sistema	102.03	109.53	108.02	102.05	101.98
Mensagem perdida	4093.10	4088.60	4093.77	4098.53	4090.40
Fahas de crash	4097.00	4099.57	4097.57	4092.53	4100.67
Atraso de envio	4094.03	4091.13	4095.40	4085.43	4099.23

Figura 4.11: Injeção de falhas na operação de out(t) sem contexto transacional

A Figura 4.12 mostra os resultados para os testes dentro do contexto transacional. Esses foram similares os resultados obtidos fora do contexto transacional. Para as demais operações, inp(s) e rdp(s) os resultados foram bastante similares. Para evitar duplicações, os gráficos obtidos nos demais testes se encontram no apêndice.



	0	2	4	6	8
Sem falhas	25.74	24.86	25.34	25.47	25.43
Corrupção de mensagem	72.91	71.21	71.59	73.27	78.29
Suspensão do sistema	103.77	102.16	105.61	102.82	108.20
Mensagem perdida	4090.30	4093.33	4087.13	4092.33	4090.17
Fahas de crash	4086.17	4089.23	4085.77	4093.67	4083.63
Atraso de envio	4070.23	4076.37	4078.67	4078.27	4074.00

Figura 4.12: Injeção de falhas na operação de out(t) com contexto transacional

4.5 Trabalhos relacionados

Os trabalhos relacionados podem ser divididos em três grupos: trabalhos que tratam falhas bizantinas, trabalhos que tratam transação e trabalhos que tratam distribuição. Isso se deve ao fato da combinação única de tratamento a falhas bizantinas, transações e distribuição no Tuplebiz.

Em (BESSANI, 2006) são apresentados três espaços de tuplas distintos que toleram falhas bizantinas. Nenhum deles possui suporte a transações, apesar do autor salientar a importância das mesmas. A comparação descrita nos parágrafos seguintes irá considerar apenas operações fora do contexto transacional. Para as equações apresentadas posteriormente, *n* indica o número total de nodos e *f* o número de nodos falhos.

O WSMR-TS (Weak State Machine Replication Tuple Space) descrito por (BESSANI, 2006) utiliza uma implementação de máquina de estados como base. Necessita de $n \geq 3f + 1$ nodos para garantir o seu funcionamento correto. Todas as operações executam de forma similar em quatro passos, com exceção da operação de rdp(s) que funciona de forma especulativa. Se o cliente receber $n - f$ respostas iguais, então considera como resposta da leitura correta.

O BTS (Byzantine Tuple Space) apresentado por (BESSANI, 2006) necessita, assim como o Tuplebiz, $n \geq 3f + 1$ para garantir o seu funcionamento. A operação de out(t) é não confirmável e o número de passos para completar essa operação é um. Esse número é inferior ao Tuplebiz, entretanto não há qualquer garantia de ordem. O fato de ser não ordenado faz a operação de out(t) não ser *Turing-powerful* (BUSI apud BESSANI, 2006, p. 52). A operação de rdp(s) executa em dois passos ao invés dos cinco desse trabalho. Contudo, a operação de inp(s) é bem mais complexa. Ela necessita de seis passos para ser completada. Primeiramente, para realizar a operação de inp(s), o BTS precisa garantir exclusão mútua ao espaço de tuplas e logo após executar o protocolo de consenso. Assim como o out(s), a operação de inp(t) no BTS é não confirmável.

O LBTS (Linearizable Byzantine Tuple Space) apresentado em (BESSANI, 2006) necessita de $n \geq 4f + 1$ para garantir seu funcionamento. Cada operação possui um número de passos de comunicação diferente. A operação mais custosa precisa de quatro passos de comunicação sendo é menor que o Tuplebiz.

A Tabela 4.11 mostra uma comparação entre as diferentes implementações de espaço de tuplas que toleram falhas bizantinas. Dentre todos elas, o LBTS é o espaço de tuplas que apresenta o menor número de passos em uma execução otimista. O Tuplebiz apresenta um número constante de cinco passos para qualquer operação e possui todas as operações confirmáveis. Apesar de o Tuplebiz em média necessitar mais passos de comunicação, ele apresenta a vantagem de suportar transações. O trabalho de (BESSANI, 2006) não apresenta análise de desempenho, logo não podemos compara-los nesse ponto de vista.

Tabela 4.11: Comparação entre espaços de tuplas que suportam falhas bizantinas

	WSMR-TS		**BTS**		**LBTS**		**Tuplebiz**	
	Passos	Confir-mável	Passos	Confir-mável	Passos	Confir-mável	Passos	Confir-mável
out(t)	4	Sim	1	Não	2	Sim	5	Sim
inp(s)	4	Sim	6	Sim	4	Sim	5	Sim
rdp(s)	2/6	Sim	2	Não	2	Sim	5	Sim

Considerando os trabalhos relacionados a transações em espaço de tuplas, tem-se o Gigaspace (GIGASPACE, 2012), Javaspace (SUN MICROSYSTEMS, 1999), Plinda 2.0 (JEONG; SHASHA, 1994) e o (ROWSTRON, 1999). Nenhum deles tem suporte a transações em um ambiente sujeito a falhas bizantinas.

O Gigaspace possui dois tipos de bloqueio: otimista e pessimista. Por padrão a aplicação usa bloqueio otimista, o qual não é aplicável ao Tuplebiz. Isso se deve ao fato de o Gigaspace possuir uma operação extra ao modelo Linda, a operação de *update*, que atualiza uma tupla no espaço de tuplas. No caso do bloqueio otimista, se dois processos tentam realizar uma atualização na mesma tupla ao mesmo tempo, aquele que possuir a cópia da tupla com o identificador de versão que se encontra no espaço de tuplas irá completar a transação e o outro receberá uma exceção. O bloqueio pessimista tem como nível de isolamento padrão o de leitura repetida, entretanto pode ser definido o nível de isolamento como serializável. Assim como o Tuplebiz, o Gigaspace permite que operações sejam feitas fora de transações.

O Javaspace é similar ao Gigaspace, entretanto só possui bloqueio pessimista. Também permite que operações sejam feitas fora de um contexto transacional.

O PLinda 2.0 provê transações com nível de isolamento serializável e possui mecanismos para recuperar o espaço de tuplas em caso de falhas. Não permite que operações possam ser feitas fora do contexto transacional, o que gera certo ônus para o sistema, pois nem sempre uma transação é requerida.

(ROWSTRON, 1999) apresenta um modelo diferente para transações. Nesse modelo considera que o espaço de tuplas é resiliente e as operações são do tipo “tudo ou nada”, mas não tem isolamento entre si. Também permite que operações sejam feitas fora de um contexto transacional.

Existem vários trabalhos que relacionam espaço de tuplas e distribuição. Para compara-los com o Tuplebiz foram utilizados os mesmos critérios da seção 2.5. A Tabela 4.12 mostra a comparação entre o Tuplebiz e os demais espaços de tuplas. Dentre os espaços de tuplas enumerados, apenas o Lime, o SwarnLinda e o Tuplebiz possuem transparência de acesso.

Tabela 4.12: Comparação entre Tuplebiz e espaços de tuplas distribuídos

Ferramenta	Modelo de Distribuição	Transparência de acesso (Porção do espaço de tuplas deve ser especificada explicitamente?)	Aplicação destino
Jada	Existem vários espaços de tuplas que são disjuntos e não se comunicam entre si.	NÃO	Internet em plataformas Java
Lime	As tuplas são federadas, um nodo tem visão apenas do espaço de tuplas dos nodos vizinhos.	SIM	Redes Sensores
SwarnLinda	As tuplas são vistas por todos os nodos, e são organizadas por similaridade de forma dinâmica.	SIM	Redes Sensores
WCL	As tuplas são migradas entre os espaços de tuplas de acordo com o a localização dos usuários da tupla.	SIM	Ambientes geograficamente distribuídos
Tupleware	As tuplas são vistas por todos os nodos.	NÃO	Aplicações paralelas
Rede Social	As tuplas são vistas por todos os nodos; as tuplas são distribuídas entre os usuários das tuplas.	NÃO	Redes ad-hoc
Tuplebiz	O espaço de tuplas é dividido em partições.	SIM	Aplicações pervasivas e Guaraná

E por fim, existem o Commit Barrier Schedule (VANDIVER; BALAKRISHNAN; LISKOV, 2007) e o Byzantine Fault-Tolerant Transaction Processing for Replicated Databases (LUIZ; LUNG; CORREIA, 2011) que propõem transações em um ambiente sujeito a falhas bizantinas. Entretanto ambos possuem banco de dados nas suas réplicas. Por esse motivo, os modelos por eles apresentados transferem todo o cuidado em relação a bloqueio e chamadas concorrentes para as implementações de banco de dados, as quais constituem as suas réplicas.

4.6 Considerações finais

O Tuplebiz apresenta uma combinação única de suporte a transações em um espaço de tuplas sujeito a falhas bizantinas. Existem trabalhos que suportam falhas bizantinas e que em média executam em menos passos que o Tuplebiz. Entretanto, esses não suportam transações. Outros trabalhos suportam transações, mas não possuem suporte a falhas bizantinas.

O desempenho do Tuplebiz é inferior quando comparado a um sistema não replicado. Todavia, ele garante tolerância a falhas o que é crítico para muitos sistemas.

5 TUPLEBIZ INTEGRADO COM A GUARANÁ

Esse capítulo descreve um estudo de caso, o uso da Guaraná com o Tuplebiz. São apresentadas as vantagens de utilizar o Tuplebiz com a Guaraná e como se dá essa integração. A integração prove duas novas propriedades para a Guaraná: possibilidade de distribuição de tarefas entre diferentes motores de execução e uma abordagem de tolerância a falhas. Para melhor ilustrar essa integração, são descritas abordagens de distribuição de tarefas e tolerância a falhas em diferentes níveis na Guaraná.

Primeiramente, tem-se uma visão geral da Guaraná e de como unir os modelos de coordenação com a mesma. Ainda é mostrada uma comparação entre as abordagens de comunicação.

Esse capítulo apresenta características básicas que abordagens de tolerância a falhas para Guaraná devem possuir. E por fim, é apresentado como o Tuplebiz pode ser integrado com a Guaraná para aumentar a dependabilidade e escalabilidade.

5.1 Motivação

Uma solução Guaraná consiste em um conjunto de processos, que são formados por tarefas. As tarefas, por sua vez, são combinadas pelo desenvolvedor para modelar uma solução de integração. Por exemplo, o desenvolvedor pode usar uma tarefa de filtro para remover um campo da mensagem e, acoplada a essa, uma tarefa de distribuição cuja função é replicar a mensagem. As tarefas, bem como os processos, se comunicam através de mensagens. O motor de execução apresentado em (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010) realiza a comunicação entre processos de forma distribuída, todavia, a comunicação entre tarefas é feita em memória, através de *buffers*. Nesse contexto, o espaço de tuplas pode ser usado como ferramenta para comunicação entre as tarefas, o que permite diminuir a granularidade de distribuição do motor de execução. Em outras palavras, podem ser distribuídas tarefas ao invés de apenas processos. Para processos com muitas tarefas, o uso de memória pode ser um gargalo.

Em (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010) é apresentada uma arquitetura para identificar falhas que podem ocorrer na Guaraná. Essa arquitetura trata as seguintes falhas: omissão, resposta, tempo e processamento de mensagens. Falha de omissão diz respeito às falhas que impedem o envio e recebimento de mensagens. Falha de resposta é causada por repostas incorretas enviadas pela aplicação ou canal de comunicação. Falha de tempo acontece quando uma aplicação leva mais tempo que o esperado para enviar uma resposta. Falha de processamento de mensagens acontece quando unidades de processamento não conseguem realizar a operação. Ainda relacionado à tolerância a falhas e à Guaraná, (FRANTZ; CORCHUELO; MOLINA-JIMENEZ, 2012a) e (FRANTZ; CORCHUELO; MOLINA-JIMENEZ, 2009)

apresentam um monitor de erros que pode ser usado para prover uma solução de integração de aplicações corporativas (EAI em inglês) com tolerância a falhas.

Para a Guaraná, a transação também é interessante. Comumente, soluções de integração integram aplicações de forma exógena, ou seja, as aplicações não têm conhecimento da integração. Se uma falha ocorre durante a execução da solução de integração, nem sempre é possível manter as aplicações que estão sendo integradas consistentes. Além desses trabalhos que expressam a necessidade do uso de transações, (BESSANI, 2006) e (PATTERSON ET AL., 1993) salientam a importância do uso de transações num espaço de tuplas.

5.2 Guaraná

A Guaraná (FRANTZ; REINA-QUINTERO; CORCHUELO, 2011) surgiu como uma proposta de Integração de Aplicações Corporativas. A Guaraná, na sua origem, é uma linguagem específica de domínio (DSL), cuja função é permitir a modelagem de uma solução de integração independentemente da aplicação à qual a solução será implantada. Porém, a Guaraná foi agregando mais funcionalidades de tal forma que hoje possui uma biblioteca para desenvolvimento e um motor de execução.

A Guaraná, como DSL, é baseada na Arquitetura Baseada em Modelos (MDA) (OMG, 2001) suportada pela Object Management Group (OMG). Segundo o MDA uma aplicação é especificada através de modelos que sofrem transformações gerando outros modelos. Essas transformações são ora manuais, ora automáticas. A Figura 5.1 ilustra os níveis e as transformações realizadas para que se obtenha, a partir de documentos informais, a solução de integração.

O modelo de mais alto nível é o Modelo Independente de Computação (CIM), o qual é descrito por pessoas da área de negócio. O modelo seguinte é o Modelo Independente de Plataforma (PIM) que não possui qualquer informação sobre a plataforma onde será executado. Nesse modelo se enquadra a Guaraná DSL. O modelo PIM é desenvolvido por pessoas da área de computação. Seguindo, o próximo modelo é o Modelo Específico de Plataforma (PSM), o qual contém dados específicos da plataforma na qual será executado e, normalmente, é gerado de forma automática a partir do modelo PIM. Por fim, tem-se o código executável.

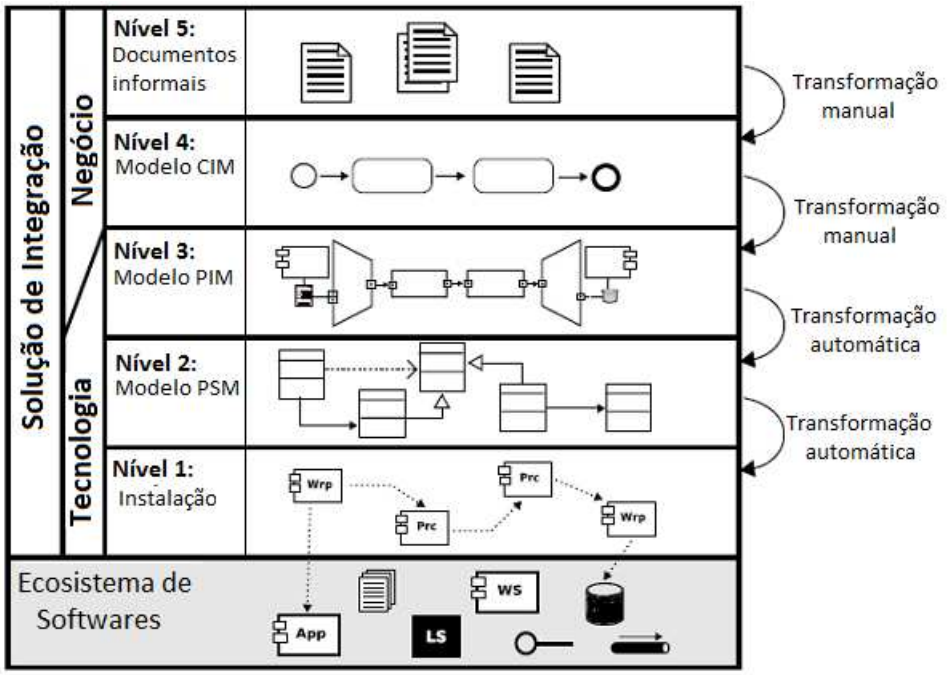


Figura 5.1: Níveis do MDA adaptado de (FRANTZ; REINA-QUINTERO; CORCHUELO, 2011)

Uma solução de integração modelada na Guaraná DSL pode ser vista na Figura 5.2. Essa figura aponta as diferentes entidades que fazem parte uma solução Guaraná: *task* (tarefa), *process* (processo), *wrapper*, *communication port*, *slot* e *integration link*.

Existe um conjunto de tarefas pré-determinado para ser usado na solução de integração. As tarefas se comunicam entre si através de troca de mensagens em um canal de comunicação chamado de *slot*. Um exemplo de tarefa é o *filter* cuja função é filtrar trechos de mensagens que não são necessários para as demais tarefas.

O processo, bem como o *wrapper*, é formado por tarefas. O *wrapper* é na verdade uma especialização de processo, isto é, o *wrapper* é um processo cuja função é se comunicar com a aplicação legada. Os processos usam portas para se comunicar com os outros e com as aplicações. O mecanismo de comunicação pode variar de um protocolo RPC sobre HTTP até um protocolo documentado desenvolvido sobre um banco de dados. Diferentes soluções podem compartilhar um mesmo processo (FRANTZ, 2012b).

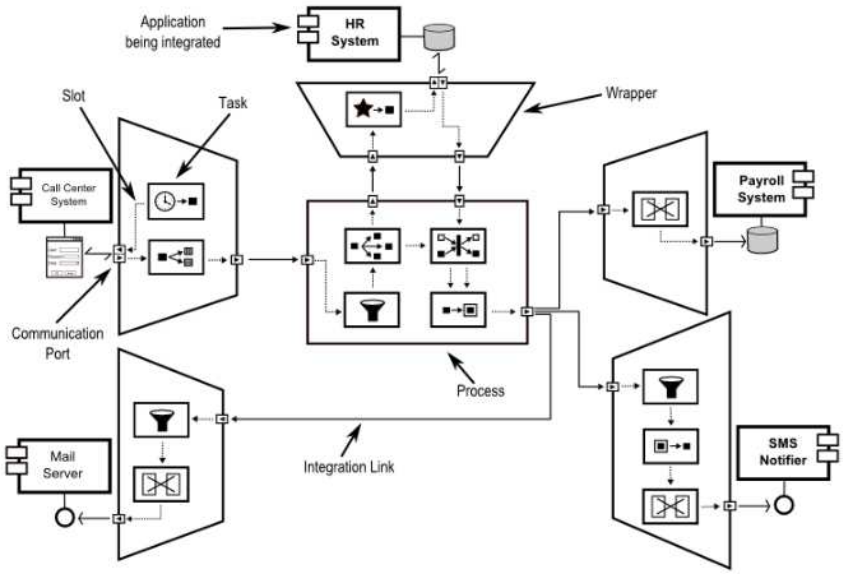


Figura 5.2: Exemplo de integração com a Guaraná retirado de (FRANTZ, 2012b)

A Guaraná, como biblioteca de desenvolvimento, permite que todas as entidades sejam implementadas. Desenvolvida em Java, a biblioteca possui classes de *slots*, processos, *integration links* e tarefas. Existe uma implementação para cada tipo de tarefa. O papel do desenvolvedor é definir o comportamento de cada tarefa de acordo com as regras de negócio e uni-las através dos *slots*.

A Guaraná, como motor de execução, permite que sejam executadas as soluções de integração de forma distribuída através do uso de vários motores de execução paralelamente. A unidade mínima de execução é um processo, isto é, um motor de execução recebe um processo para ser processado. O desenvolvedor deve explicitamente indicar em qual motor de execução ele pretende que o processo seja executado. Um mesmo motor de execução pode executar tarefas de diferentes soluções.

De acordo com (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010) o motor de execução é composto dos seguintes elementos:

- Pelo menos dois *BindingComponent* são necessários. Eles fazem a ligação entre a solução de integração e aplicação. Mesmo que exista somente uma aplicação, dois desses componentes são necessários, pois um escreve e outro lê a informação;
- *ThreadPool* é um conjunto de *threads* que executam as tarefas disponíveis. Uma tarefa é executada quando essa possui um número de mensagens suficiente na sua entrada para ser executada. O tamanho do *threadPool* é definido pelo desenvolvedor em um arquivo de configuração;
- *ReadyQueue* contém as tarefas que estão disponíveis para serem executadas e aguardam uma thread livre para fazê-lo;
- *ChannelBoard* gerencia os canais de comunicação. As mensagens não são armazenadas em ordem, pois a ordem não é importante para a aplicação (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010).

5.2.1 Desempenho do motor de execução da Guaraná

Uma aplicação da Guaraná foi utilizada para verificar como se comportam a alocação de memória e número de *threads* no motor de execução da Guaraná. A aplicação utilizada foi a café-solution que se encontra em (FRANTZ, 2012b). A Figura 5.3 mostra o modelo da solução. O código fonte utilizado sofreu algumas poucas alterações. A aplicação *orders* que no código fonte original é executada remotamente, aqui é executada na mesma máquina. Além disso, foram adicionados tempos de espera nas aplicações *orders*, *barista cold drink* e *barista hot drink.*

As medidas de memória e número de *threads* foram feitas usando o Profiler disponível na IDE Netbeans. A versão de Netbeans utilizada foi a NetBeans IDE 7.0 (Build 201107282000). O tamanho de ThreadPool usado foi 4. O número de *threads* criados foi 10, pois são criadas algumas *threads* adicionais além das *threads* que processam as tarefas.

No primeiro teste, a aplicação de *orders* envia um pedido a cada 10ms e as aplicações *baristas* levam 100ms para responder ao pedido. Nesse caso de testes o uso de memória aumentou progressivamente conforme o uso. A Figura 5.4 mostra como o uso da memória aumentou com o passar do tempo. No primeiro *snapshot* o uso de memória era de aproximadamente 12MB e após 17 minutos o uso passou para aproximadamente 250MB.

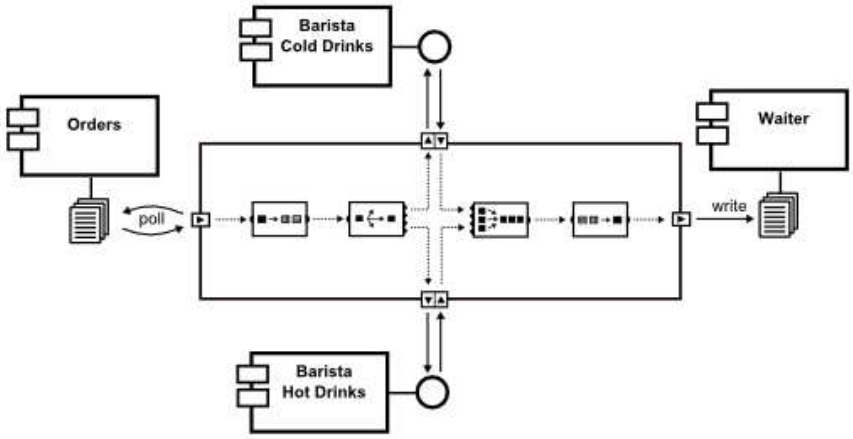


Figura 5.3: Cafe-solution, solução modelada em Guaraná. Retirada de (FRANTZ, 2012b)

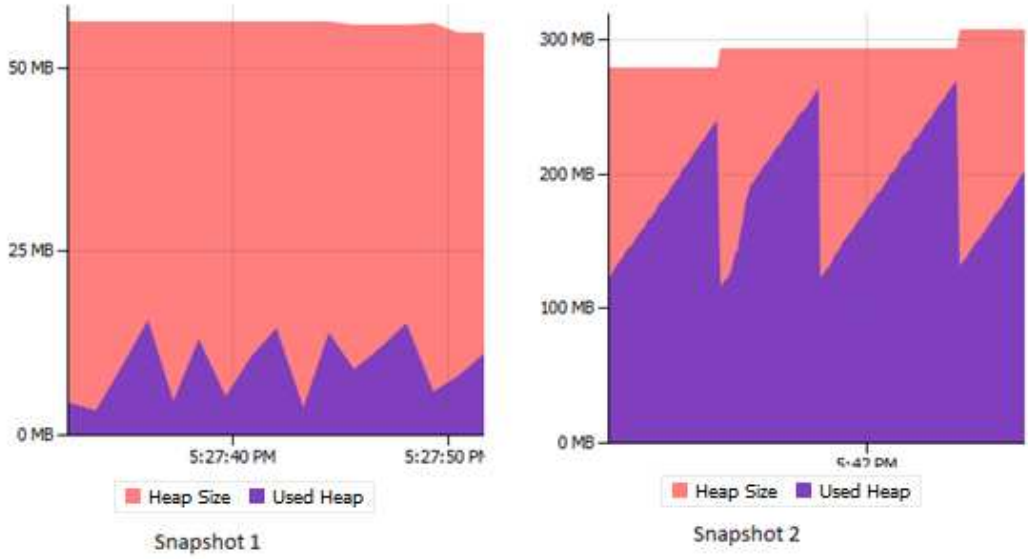


Figura 5.4: Uso de memória da Guaraná com envio de pedido a cada 10ms e resposta a cada 100ms

No segundo teste, a aplicação de *order* envia um pedido a cada 20ms e as aplicações *baristas* levam 100ms para responder ao pedido. A Figura 5.5 mostra o uso da memória com o passar do tempo. No primeiro *snapshot* o uso de memória era de aproximadamente 12MB e após 4 minutos o uso passou para aproximadamente 50MB. A quantidade de *threads* não é o fator limitante, mas sim o uso de memória. Se os tempos de processamento das tarefas não forem proporcionais, os *slots* ficarão repletos de mensagens e ocuparão muita memória.

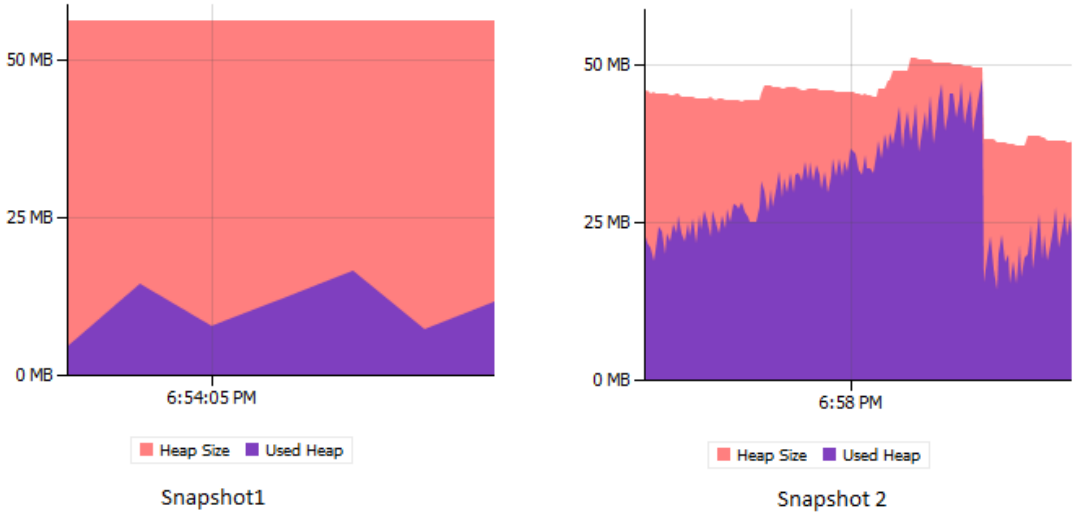


Figura 5.5: Uso de memória da Guaraná com envio de pedido a cada 20ms e resposta a cada 100ms

5.2.2 Tolerância a falhas na Guaraná

A Guaraná não possui uma arquitetura tolerante a falhas. Os trabalhos de tolerância a falhas que existem hoje na Guaraná estão relacionados ao monitoramento de falhas e não no tratamento das mesmas. Segundo (FRANTZ; CORCHUELO; MOLINA-JIMENEZ, 2012a) no estágio de monitoramento de falhas, o evento que indica o erro é armazenado para que se ache uma correlação entre o erro e as mensagens que o geraram. Essa correlação será usada no estágio de recuperação de falhas.

Em (FRANTZ; CORCHUELO; MOLINA-JIMENEZ, 2009) é apresentada uma solução de detecção que se enquadra ao Guaraná. A ideia principal desse trabalho é armazenar em um *log* todas as execuções, tanto as falhas e as bem sucedidas. Os processos e as portas são responsáveis por alimentar o *log*. As mensagens são relacionadas aos fluxos de execução. São armazenadas as relações entre as mensagens, isto é, qual a sequência de mensagens que foram enviadas para gerar a mensagem em questão. Quando informada uma execução falha, uma ação de compensação deve ser executada. Em (FRANTZ; CORCHUELO; MOLINA-JIMENEZ, 2012a) é apresentada uma solução de monitoramento de erro. Entretanto, não há publicações sobre tolerância a falhas na Guaraná.

5.3 Coordenação de comunicação e Guaraná

O motor de execução da Guaraná permite que apenas processos sejam distribuídos. Consequentemente, todas as tarefas de um processo devem ser executadas em apenas um motor de execução. A comunicação entre tarefas é feita em memória. A Figura 5.6 mostra um exemplo de integração distribuída entre dois motores de execução.

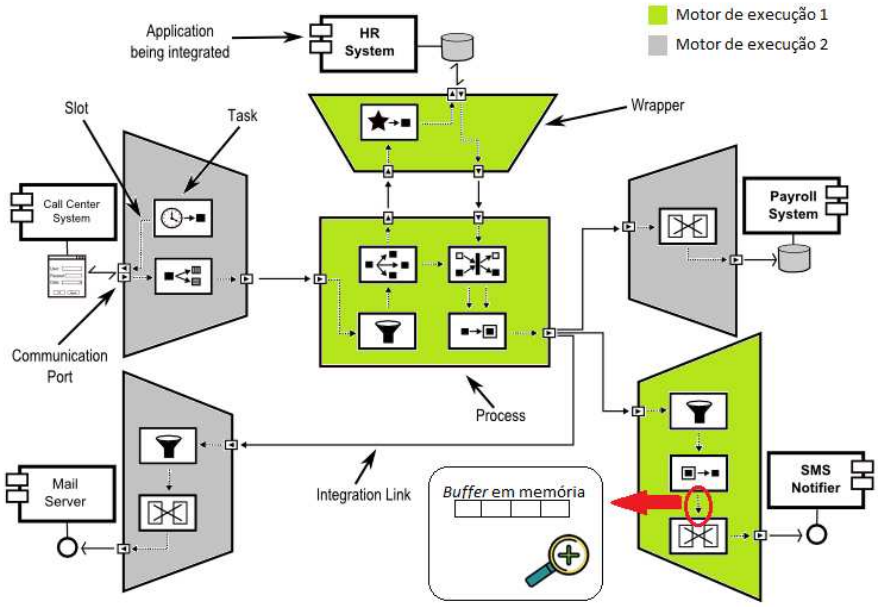


Figura 5.6: Exemplo de distribuição de processos entre os motores de execução da Guaraná adaptado de (FRANTZ, 2012b)

Um processo poder ser compartilhado entre diferentes soluções (FRANTZ, 2012b) e não há uma limitação de quantidades de tarefas em um processo, ou seja, um processo pode ser complexo e conter muitas tarefas. Adicionalmente, um processo pode se comunicar com aplicações em localidades distintas. No caso mais crítico, onde o processo com muitas tarefas é compartilhado entre aplicações, a execução centralizada de tarefas pode ser uma limitação conforme seção 5.2.1, a qual indica que o uso de memória aumenta consideravelmente se o tempo usado para a execução das tarefas não é uniforme.

Nesse contexto, essa seção propõe a execução distribuída das tarefas. A maior vantagem dessa abordagem é um melhor desempenho, mas também serve de base para distribuição dinâmica das aplicações entre os motores de execução. Os modelos de coordenação disponíveis são: direto, baseado em eventos e espaço de dados compartilhados (SOUZA, 2009).

O motor de execução da Guaraná contém um *ThreadPool*, o qual é formado por um grupo de *threads* que executam as tarefas quando essas estiverem prontas para serem executadas. Uma tarefa está pronta para ser executada quando possui no seu *buffer* de entrada todas as mensagens que precisa para executar. Para utilizar a Guaraná com endereçamento direto, cada tarefa deverá ter seu próprio endereço e a tarefa de destino deve estar em execução quando a mensagem for enviada. Então, uma nova arquitetura de comunicação precisa ser desenvolvida como mostra a Figura 5.7. Um *buffer* em memória deve ser criado para cada tarefa. Desse modo a tarefa só será executada quando houver alguma mensagem nesse *buffer* e o motor de execução funcionará de modo similar ao atual. Adicionalmente, dois processos precisam ser criados para receber e enviar as mensagens.

A vantagen desse modelo é poder distribuir as tarefas ao longo dos servidores e usar um modelo simples de comunicação, como por exemplo, *sockets*. Uma das desvantagens é a grande quantidade de endereços que devem ser administrados. Tem-se um endereço distinto para cada tarefa. Outra desvantagem se deve ao fato que se um servidor não estiver disponível, as mensagens enviadas a ele são perdidas.

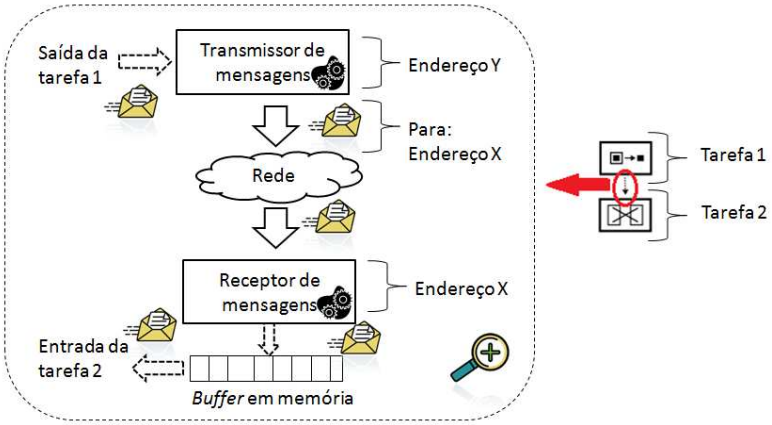


Figura 5.7: Comunicação entre tarefas da Guaraná usando coordenação direta

Para se utilizar a comunicação baseado em eventos, cada tarefa deve se inscrever no tópico e deve estar em execução quando a mensagem for enviada. Assim como no caso da comunicação direta, esse funcionamento não condiz com o esperado pelo motor de execução. A Figura 5.8 mostra uma abordagem de integração da Guaraná com um *middleware* baseado em eventos. Assim como no modelo de comunicação direta, um

buffer em memória deve ser criado. Cada tarefa deve ter dois processos adicionais: o *publisher* e o *subscriber*. O *publisher* enviará a mensagem para um tópico pré-definido e o *middleware* enviará a mensagens para todos o *subscribers* que estão esperando mensagens com esse tópico. Na prática, cada tarefa deverá ter um tópico associado com o intuito de garantir que apenas uma tarefa consuma uma mensagem por vez.

As vantagens dessa abordagem são a distribuição das tarefas entre os servidores e o desacoplamento referencial, isto é, não é necessário que cada tarefa seja endereçada individualmente. Uma desvantagem é a necessidade de muitos tópicos o que acabará funcionando de modo similar a ter vários endereços. Outra desvantagem é a falta de garantia de entrega, isto é, se um servidor estiver fora do ar no momento do envio, a mensagem enviada nunca chegará ao destino.

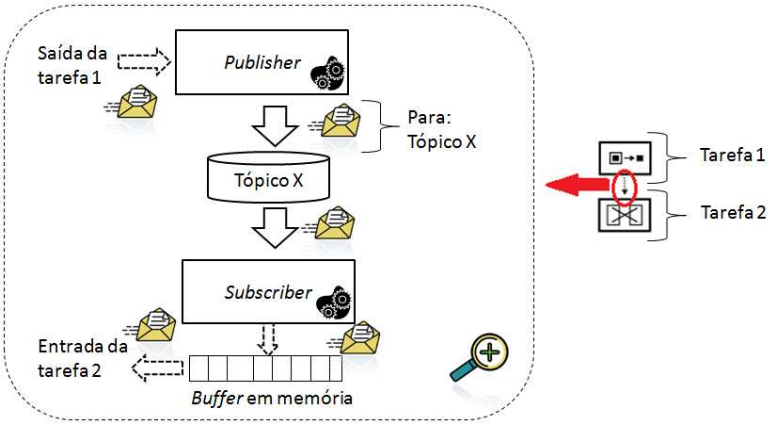


Figura 5.8: Comunicação entre tarefas da Guaraná usando coordenação baseada em eventos

Para utilizar a Guaraná com o modelo de coordenação baseado em espaço de dados (representado pelo espaço de tuplas), cada tarefa deve conhecer o espaço de tuplas. Nessa abordagem as mensagens não são ordenadas o que não é um problema para a Guaraná (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010). A Figura 5.9 mostra como fica a Guaraná integrada com um espaço de tuplas. Nesse caso não é necessário um *buffer* em memória, pois as mensagens ficam armazenadas no espaço de tuplas. Também, não são necessários processos adicionais, porque várias implementações de espaço de tuplas tem um evento de notificação para indicar quando uma mensagem com os padrões esperado é recebida.

Uma das vantagens desse modelo é o fato de não necessitar de um *buffer* de memória local e nem de processos extras. Outra vantagem é que mesmo se um servidor estiver fora do ar, a mensagem não é perdida. A mensagem fica disponível para ser processada posteriormente. Uma desvantagem é o desempenho do sistema que pode ser prejudicado pelo acesso simultâneo de várias tarefas ao mesmo espaço de tuplas. Esse tipo de problema pode ocorrer quando as tarefas tiverem tempo similar de execução e acessarem quase que simultaneamente o espaço de tuplas.

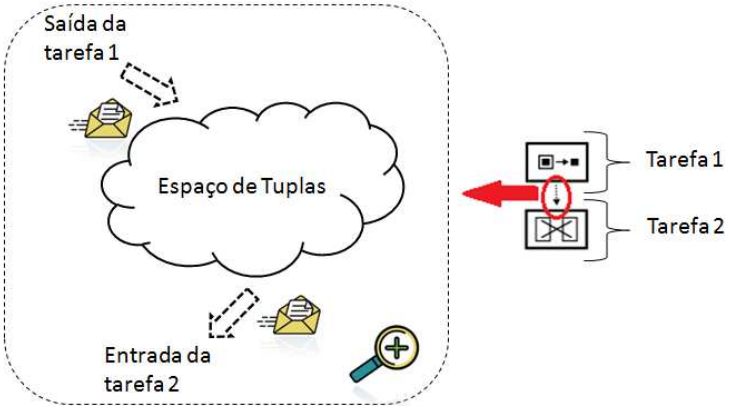


Figura 5.9: Comunicação entre tarefas da Guaraná usando espaço de dados compartilhado

A Tabela 5.1 mostra uma comparação entre os diferentes métodos de coordenação que podem ser utilizados com a Guaraná. O modelo de coordenação com espaço de dados compartilhados é o que apresenta mais vantagens em relação aos demais. A principal delas é o fato das mensagens não serem perdidas quando o servidor de destino fica fora do ar. Essa característica é importante como um mecanismo para suportar falhas de *crash* dos servidores onde estão os motores de execução.

Tabela 5.1: Comparação entre as diferentes abordagens para distribuir tarefas da Guaraná

Modelo De Coordenação	**Vantagens**	**Desvantagens**
Direto	• Permite distribuição; • Simples.	• Cada tarefa deve ter seu próprio endereço; • Acoplamento temporal.
Baseado em eventos	• Permite distribuição; • Não é necessário dar um endereço para cada tarefa.	• Cada tarefa deve ter seu próprio tópico; • Acoplamento temporal.
Espaço de dados compartilhado	• Permite distribuição; • Desacoplamento temporal; • Não é necessário dar um endereço para cada tarefa.	• Pode ser o gargalo de desempenho no processo

5.4 Guaraná integrada ao Tuplebiz

Existem hoje algumas abordagens para detecção de falhas para Guaraná, mas nenhuma para tolerar falhas. Os tipos de falhas que podem ocorrer na Guaraná estão enumeradas na sessão 5.1, as quais são falhas de omissão, resposta, tempo e processamento de mensagens (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010). Pensando em uma abordagem para tolerância a falhas para a Guaraná, duas características devem estar presentes na abordagem proposta: (1) garantia de que as mensagens são corretamente recebidas pelo destino; (2) garantia que a execução completa aconteça de forma atômica. A característica (1) garante que as falhas de

omissão, resposta e tempo serão tratadas, pois a entrega de mensagem correta da mensagem ao destino é garantida. Como especificado na seção 2.1, uma abordagem de tolerância a falhas que suporta falhas bizantinas atenderia a característica (1). A característica (2) garante que falhas no processamento de mensagens não deixam o sistema inconsistente. Se uma tarefa ou processo (depende do nível de granularidade da abordagem) garante que sempre ao consumir uma mensagem, irá processa-la e gerar uma ou mais mensagens de saída, o sistema não fica em um estado inconsistente onde apenas parte do processamento é executado.

O Tuplebiz provê para a Guaraná um motor de execução que pode executar tarefas de modo distribuído e uma abordagem de tolerância a falhas no nível de tarefas. O Tuplebiz é um espaço de tuplas, logo sua integração funciona como mostra a Figura 5.10.

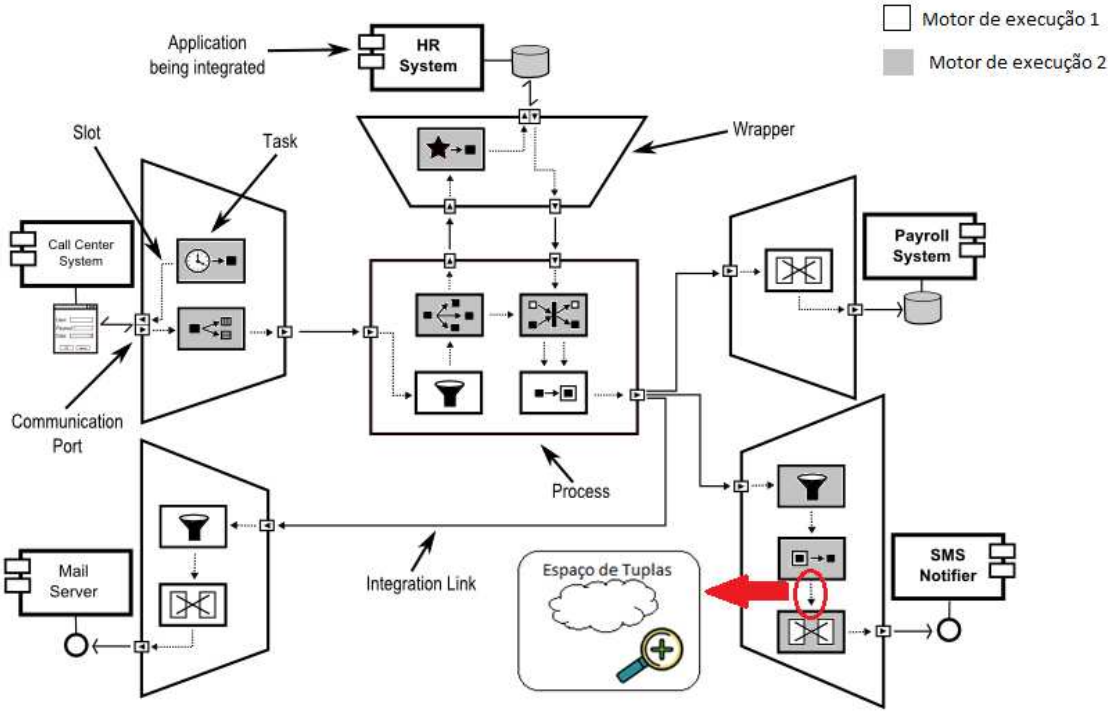


Figura 5.10: Exemplo de distribuição de tarefas entre os motores de execução da Guaraná integrado com Tuplebiz adaptado de (FRANTZ, 2012b)

A execução de tarefas distribuídas é realizada da mesma forma que no modelo de coordenação de espaços de dados compartilhado descrito na seção 5.1. O único ponto, que deve ser inserido no Guaraná para que a integração seja totalmente satisfatória, é a implementação do evento que notifica a inserção da tupla com um determinado valor. Esse mecanismo é encontrado em implementações como Javaspace e Gigaspace. Essa característica ainda não está presente no Tuplebiz, entretanto mecanismos alternativos podem ser usados para o mesmo, como por exemplo, o uso de uma tarefa que verifica de tempos em tempos se existe uma tupla com as características desejadas no espaço de tuplas.

O conceito de particionamento do Tuplebiz pode garantir um melhor desempenho para a Guaraná. As tarefas sempre recebem e enviam mensagens para os mesmos pares, logo se pode definir uma partição para um grupo de tarefas que se comunicam com mais

frequência. Esse modelo evita o gargalo de várias tarefas acessando o espaço de tuplas ao mesmo tempo.

As características necessárias de uma abordagem de tolerância a falhas na Guaraná, as quais foram citadas no início da seção, são atendidas pelo Tuplebiz. A característica (1) - a garantia que as mensagens são corretamente recebidas pelo o destino – é provida pela capacidade de tolerar falhas bizantinas do Tuplebiz. A característica (2) - garantia que a execução completa aconteça de forma atômica - é provida pelo suporte à transação.

5.5 Considerações finais

A extensão da Guaraná para suportar distribuição de tarefas entre motores de execução é interessante para aumentar a escalabilidade. Ao levar em consideração os modelos de comunicação existentes, o modelo baseado em espaço de dados se mostrou o mais adequado.

A integração do motor de execução da Guaraná e o Tuplebiz é muito promissora. Além de ser um modelo de comunicação bastante adequado, a proposta de tolerância a falhas e o suporte a transações fazem do Tuplebiz uma ótima proposta de *middleware* para a Guaraná. O modo que o Tuplebiz trata as falhas atende as possíveis falhas da Guaraná enumeradas por (FRANTZ; MOLINA-JIMENEZ; CORCHUELO, 2010).

Como trabalho futuro, a distribuição dinâmica das tarefas da Guaraná é sugerida. Outro trabalho futuro é avaliar, através de protótipos, a integração entre a Guaraná e o Tuplebiz.

6 CONCLUSÃO

Esse trabalho define o Tuplebiz, um espaço de tuplas distribuído e resiliente a falhas bizantinas e com suporte a transações. O estudo de caso apresentado é a Guaraná, uma linguagem específica de domínio, a qual é utilizada para integrar aplicações corporativas.

O modelo compreende os mecanismos usados na distribuição, tolerância a falhas e suporte a transação. O modelo do sistema, bem como a arquitetura, foi detalhada.

O estudo de caso da Guaraná mostrou abordagens para distribuir as tarefas de uma solução entre os diferentes motores de execução. Essas abordagens levaram em conta os distintos modelos de coordenação. A seção que descreve a integração da Guaraná com o Tuplebiz mostrou as vantagens e desvantagens dessa integração no que diz respeito à escalabilidade e tolerância a falhas.

O capítulo de avaliação mostra a quantidade de passos trocados para executar as operações do Tuplebiz. Além disso, são realizados os testes de desempenho e injeção de falhas. A latência do Tuplebiz sem falhas é aproximadamente 2,8 vezes maior que a latência de um sistema não replicado. Os testes de injeção tiveram como base o Vajra (WIERMAN; NARASIMHAN, 2008), um *framework* de testes de injeção de falhas para sistemas tolerantes a falhas bizantinas. O Vajra compreende os seguintes tipos de falha: mensagens perdidas, atrasos de envio de mensagens, corrupção de mensagens, suspensão do sistema e *crash*. A latência no caso de falhas foi maior que no caso sem falhas, mas todas as falhas foram suportadas pelo Tuplebiz.

6.1 Revisão dos objetivos

Essa seção ilustra as abordagens utilizadas nessa dissertação para atingir os objetivos citados na seção1 .1.

- Definir um espaço de tuplas distribuído que possua suporte a falhas bizantinas e transações.

A distribuição é provida pelo uso das partições. As partições permitem que o espaço de tuplas seja distribuído de forma transparente para o cliente entre os diferentes nodos do sistema. Cada partição tem uma função de equivalência que define as características das tuplas que devem estar presentes em cada partição. Essa abordagem evita o acesso centralizado ao espaço de tuplas.

As falhas são suportadas por um protocolo de tolerância a falhas bizantinas. O Tuplebiz utiliza como base o protocolo Zyzzyva. Algumas características peculiares ao espaço de tuplas precisaram ser endereçadas para atingir os objetivos. Uma delas é o

não determinismo. Um mecanismo de identificação foi posto em prática para garantir respostas determinísticas.

As transações no Tuplebiz devem ser explicitamente definidas pelo cliente. O cliente pode executar operações tanto dentro quanto fora de um contexto transacional. Transações aninhadas não são suportadas.

- As transações devem garantir as propriedades ACID, isto é, as propriedades de atomicidade, consistência, isolamento e durabilidade. O nível de isolamento deve ser pelo menos similar ao encontrado na literatura.

As transações providas pelo Tuplebiz possuem o mesmo nível de isolamento do Javaspace e Gigaspace. As propriedades ACID são garantidas através de bloqueios de leitura e escrita. Os bloqueios são gerenciados pelo Gerente de Bloqueio, o qual possui mecanismos para minimizar o número de tuplas bloqueadas. Esse mecanismo unido ao mecanismo de *timeout* evita o ataque de clientes maliciosos.

- Outro objetivo desse trabalho é modelar o *middleware* de comunicação do motor de execução da Guaraná distribuído e propor um mecanismo de tolerância a falhas.

A integração com a Guaraná foi usada como estudo de caso. Foi verificado como integrar a Guaraná com os diferentes modelos de coordenação. A integração do Tuplebiz com a Guaraná permite que sejam distribuídas tarefas ao invés de processos. Além disso, essa integração provê um mecanismo de tolerância a falhas para a Guaraná.

6.2 Contribuições e resultados

O Tuplebiz apresenta um conceito único até então na literatura que é um espaço de tuplas distribuído, resiliente a falhas bizantinas e com suporte a transações. As transações possuem as propriedades ACID.

O uso de técnicas de tolerância a falhas fazem o Tuplebiz ter um pior desempenho quando comparado a um sistema que não possui qualquer tolerância a falhas. No que diz respeito à latência, o Tuplebiz é aproximadamente três vezes mais lento que o modelo não replicado. A latência dentro e fora do contexto transacional é praticamente a mesma. Testes de injeção de falhas foram realizados para verificar a capacidade de Tuplebiz de suportar falhas. A latência no caso de falhas foi maior que no caso sem falhas, mas todas as falhas injetadas foram suportadas pelo Tuplebiz.

A integração com a Guaraná garante a mesma a distribuição de tarefas e tolerância a falhas. O mecanismo de tolerância a falhas age no nível de tarefas. Não foi definida uma proposta de tolerância a falhas para uma solução de integração como um todo.

Quando comparado a outros espaços de tuplas que suportam falhas bizantinas, o Tuplebiz necessita em média mais passos de execução. Entretanto, dentre esses é o único que possui suporte a transações. Dentre os espaços de tuplas com suporte a transações ou distribuídos, o Tuplebiz é o único que apresenta suporte a falhas bizantinas.

6.3 Trabalho futuros

Existe uma série de trabalhos futuros que podem ser derivados do Tuplebiz e da integração com a Guaraná.

Uma sugestão de trabalho futuro é o uso de diferentes implementações de espaços de tuplas, um em cada réplica. Essa abordagem pode aprimorar as técnicas de tolerância a falhas utilizadas fazendo o uso de programação diversitária (WEBER, 2002). Esse tipo de abordagem facilita a descoberta de erros de implementação que estariam presentes em todas as réplicas. Em (VANDIVER; BALAKRISHNAN; LISKOV, 2007) foi encontrado um erro em uma versão do banco de dados MySQL devido à programação diversitária.

Outro trabalho futuro diz respeito ao uso de transações. Definições mais restritas de bloqueio podem ser aplicadas às transações para garantir um nível de isolamento serializável.

No ponto de vista de desempenho, seria interessante um trabalho que visasse eliminar gargalos do modelo. O próprio Zyzzyva (KOTLA ET AL., 2009) propõe algumas técnicas que podem ser aplicadas para melhorar o desempenho.

REFERÊNCIAS

ABD-EL-MALEK, M., ET AL. Fault-scalable Byzantine fault-tolerant services. In: ACM symposium on Operating systems principles, SOSP, 20, 2005, Brighton, Inglaterra. **Proceedings**, New York, Estados Unidos, 2005, p. 59-74.

ATKINSON, A., Tupleware: A Distributed Tuple Space for Cluster Computing. In: International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT, 9, 2008, Otago, Nova Zelândia. **Proceedings**,p. 121- 126.

ATKINSON, A. **Tupleware: a distributed tuple space for the development and execution of array-based applications in a cluster computing environment.** Tese de doutorado**,** University of Tasmania, 178f. University of Tasmania, 2010.

BAKKEN, D.E. Supporting fault-tolerant parallel programming in Linda**. IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.6,n.3, p. 287 – 302, março, 1995.

BAKKEN, D.E. 2011. **Middleware**. Disponível em http://www.eecs.wsu.edu/~bakken/, acesso abril de 2011.

BESSANI, A.N., **Coordenação Desacoplada Tolerante a Faltas Bizantinas**. 144f.Tese (Doutorado em Engenharia Elétrica). UFSC, 2006.

BESSANI, A.N., ET AL. DepSpace: a byzantine fault-tolerant coordination service. In: **ACM SIGOPS/EuroSys European Conference on Computer Systems**. EuroSys, 3, 2008, Glasgow, Escócia. ACM SIGOPS Operating Systems Review - EuroSys '08, New York, Estados Unidos, v. 42, n. 4, p. 163-176.

BERNSTEIN, P. A., HADZILACOS, V., GOODIMAN, N., **Concurrency Control and Recovery in Distributed Database Systems**, Boston: Addison Wesley Publishing Company, 1987.

BIRMAN, K. **Building Secure and Reliable Network Applications**.1 ed. Nova York. Manning Publications Co., 1996.

CARRIERO, N., GELERNTER, D. **How to write parallel programs. A first course**.3 ed. [S.l.], MIT Press, 1992.

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

CERIOTTI, M., ET AL. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In: International Conference on Information Processing in Sensor Networks, IPSN, 2009, São Francisco, Estados Unidos. **Proceedings**, [S.l.], 2009, p. 277 -288.

CHARLES, A., MENEZES, R., TOLKSDORF, R. On the Implementation of SwarmLinda**.** In: ACM Southeast regional conference, ACM SE, 42, 2004, Huntsville, Estados Unidos. **Proceedings**, New York, Estados Unidos, 2004, p. 297-298.

CHETAN, S., RANGANATHAN, A., CAMPBELL, R. Towards fault tolerance pervasive computing**. IEEE, Technology and Society Magazine**, [S. l.], v. 24, n. 1, p.38-45, mar. 2005.

CIANCARINI, P. ROSSI, D. Jada: Coordination and communication for Java agents. **Mobile Object Systems Towards the Programmable Internet**, Berlin, Alemanha, v. 1222, p. 213-226, 1997.

COSTA, P., ET AL. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In: International workshop on Middleware for sensor networks, MidSens, 2006, Melbourne, Austrália. **Proceedings**, Nova York, 2006, p. 43-48.

COWLING, J., ET AL. HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In: Symposium on Operating systems design and implementation, OSDI, 7, Seattle, Estados Unidos. **Proceedings**, [S.l.], 2006, p. 177-190.

FREEMAN, E. **JavaSpaces : principles, patterns, and practice**.1 ed. Essex, Addison-Wesley Longman Ltd., 1999.

FUMMI F., ET AL. A Middleware-centric Design Flow for Networked Embedded Systems. In Design, Automation & Test in Europe Conference & Exhibition, DATE, Nice, France, 2007. **Proceedings**, [S.l.], 2006, p. 1-6.

FRANTZ, R. Z., CORCHUELO, R., MOLINA-JIMENEZ, C., Towards a Fault-Tolerant Architecture for Enterprise Application Integration Solutions. **Lecture Notes in Computer Science**, v. 5872, p. 294-303, 2009.

FRANTZ, R. Z., MOLINA-JIMENEZ, C.,CORCHUELO, R., On the Design of a Domain Specific Language for Enterprise Application Integration Solutions. In: **International Workshop on Model-Driven Engineering** , MOSE, 2010, Málaga, Espanha. **Proceedings**, [S.l.], 2006, p. 19-30.

FRANTZ, R. Z., REINA-QUINTERO, A. M., CORCHUELO, R., A Domain-Specific Language to Design Enterprise Application Integration Solutions**. International Journal of Cooperative Information Systems (IJCIS)**, [S.l.], v.20, n.2, p.143-176, maio 2011.

FRANTZ, R. Z., CORCHUELO, R., MOLINA-JIMENEZ, C., A proposal to detect errors in Enterprise Application Integration solutions. **Journal of Systems and Software**, New York, Estados Unidos, v.85, n. 3, p.480-497. 2012a.

FRANTZ, R. Z., **Guaraná DSL,** Disponível em http://www.tdg-seville.info/rzfrantz/Guaran%C3%A1+DSL, 2012b, acesso em março de 2012.

GUERRAOUI, R., SCHIPER, A. Software-Based Replication for Fault Tolerance**. Computer** , Los Alamitos, Estados Unidos, v. 30 , n. 4, p. 68-74, abril 1997.

GHASEMIGOL, M. ET AL. A Linda-based Hierarchical Master-Worker Model. **International Journal of Computer Theory and Engineering**, [S.l.], v. 1, n. 5, p. 1793-8201, dez. 2009.

GIGASPACE . **Gigaspace**.2011. Disponível em http://www.gigaspaces.com, acesso

maio 2011.

HENRICKSEN, K., ROBINSON, R. A survey of middleware for sensor networks: state-of-the-art and future directions. In: International workshop on Middleware for sensor networks, MidSens, 2006, Melbourne, Austrália. **Proceedings**, Nova York, 2006, p. 60-65.

HSQL, 2011, **HSQL**. Disponível em http://hsqldb.org/, acesso maio 2011

IBM, 2012, **TSpace**. Disponível em https://www.almaden.ibm.com/cs/TSpaces/, acesso fevereiro 2012.

JEONG, K., SHASHA, D., PLinda 2.0: a transactional/checkpointing approach to fault tolerant Linda. In: Symposium on Reliable Distributed Systems, 13, 1994, Dana Point, Estados Unidos. **Proceedings**, [S.l], 1994, p. 96-105.

KOTLA, R. ET AL. Zyzzyva: Speculative Byzantine fault tolerance**.** In: ACM SIGOPS symposium on Operating systems principles, SIGOPS, 21, 2007, Washington, Estados Unidos. **Proceedings**, Nova York, 2007, p. 45-58.

LEHMAN, T.J, MCLAUGHRY, S. W., WYCKO, P**.** T Spaces: The Next Wave. In: Annual Hawaii International Conference on System Sciences, HICSS, 32, 1999, Mauai, Estados Unidos. **Proceedings**, [S.l], 1999, p. 8037.

LILJA, D. J. **Measuring computer performance. A practitioner's guide.** 1 ed., Cambridge, Cambridge University Press, 2004.

LUIZ, A. F., LUNG, L. C., CORREIA, M. Byzantine Fault-Tolerant Transaction Processing for Replicated Databases. In: IEEE International Symposium on Network Computing and Applications , NCA, 10, 2011, Cambridge, Estados Unidos. **Proceedings**, [S.l], 2011, p. 83-90.

MAMEI, M., ZAMBONELLI, F., LEONARDI, L., Tuples on the air: a middleware for context-aware computing in dynamic networks. In: International Conference Distributed Computing Systems Workshops, 23, 2003, Rhode Island, Estados Unidos. **Proceedings**, [S.l], 2003, p. 342-347.

MATTHEWS, D., CHALMERS, D., WAKEMAN, I. MediateSpace: decentralised contextual mediation using tuple spaces. In: International Workshop on Middleware for Pervasive Mobile and Embedded Computing, M-MPAC, 3, 2011, Lisboa, Portugal. **Proceedings**, [S.l], 2003, artigo 5.

MENEZES, P. B., **Matemática Discreta para Computação e Informática**. 1. Ed. Porto Alegre, Editora Sagra Luzzatto, 2004.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **Digital signature standard (DSS)**. Estados Unidos, 2009. Disponível em http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf, acesso outubro de 2012.

NIXON, L.J.B. ET AL., Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art. **The Knowledge Engineering Review**, New York, Estados Unidos, v. 23, n.2, p. 181–212, 2008.

NOBLE, M. S., ZLATEVA, S., Scientific Computation With JavaSpaces. In: International Conference on High-Performance Computing and Networking, HPCN, 9, Amsterdam, Holanda. 2001. **Proceedings**, Londres, Inglaterra, 2001, p. 657 – 666

OMG, 2001, **Model Driven Architecture (MDA).** Disponível em http://www.omg.org/mda/specs.htm, acesso em abril de 2011.

PATTERSON, L., ET AL. Construction of a Fault-Tolerant Distributed Tuple-Space. In: ACM/SIGAPP symposium on Applied computing: states of the art and practice, SAC, 1993, Indianapolis, Estados Unidos. **Proceedings**, New York, Estados Unidos, 1993, p. 279 - 285

PERICH, F. ET AL. Neighborhood-Consistent Transaction Management for Pervasive Computing Environments. **Database and Expert Systems Applications**, Berlim, Alemanha, v. 2736, p. 276-286, 2003.

PICCO, G.P., MURPHY, A.L., ROMAN, G. LIME: Linda meets mobility. In: International conference on Software engineering, ICSE, 21, 1999, Los Angeles, Estados Unidos. **Proceedings**, New York, Estados Unidos, 1999, p. 368-377.

PU, C. **Replication and Nested Transactions in the Eden Distributed System**. Technical Report. Department of Computer Science, Columbia University, New York, Estados Unidos, 1986.

ROWSTRON, A. WCL: A Coordination Language to Geographically Distributed Agents. **World Wide Web**, [S.l.] , Kluwer Academic Publishers, v.1, n. 3, p. 167-179. 1998

ROWSTRON, A. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages. **Coordination Languages and Models**, Berlim, Alemanha, v. 1594, p. 196- 210, 1999.

RODRIGUES, R. CASTRO, M. LISKOV.B. BASE: using abstraction to improve fault tolerance. **ACM Transactions on Computer Systems (TOCS)** , New York, Estados Unidos, v. 21, n .3, p. 236 – 269, ago. 2003.

SARIGÖL, E., RIVA, O., ALONSO, G. A Tuple Space for Social Networking on Mobile Phones. In: IEEE International Conference on Data Engineering , ICDE, 26, 2010. Long Beach, Estados Unidos. **Proceedings**, [S.l], 2010, p. 988 – 991.

SATYANARAYANAN, M. Pervasive computing: vision and challenges. **Personal Communications**, IEEE, [S.l.], v. 8, n.4, p. 10-17 , 2001.

SERAFINI, M., SURI, N. **Reducing the Costs of Large-Scale BFT Replication,** disponível em http://www.cs.cornell.edu/projects/ladis2008/materials/serafini-LADIS-talk-web.pdf, acessado em novembro 2011, 2008.

SIMON, A., MELTON, J. **Understanding the new SQL: a complete guide**.1 ed. [S.l.], Morgan Kaufmann, 1993.

SCHUMACHER, M., SHULLER, B., Space-Based Approach to High-Throughput Computations in UNICORE 6 Grids. **Euro-Par 2008 Workshops - Parallel Processing**, [S.l], v. 5415, p. 75- 83, 2009.

SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S. **Database System Concepts**. 5 ed. Nova York. McGrawhill, 2006.

SOUZA, R. **Uma Contribuição à Coordenação na Computação Pervasiva com Aplicações na Área Médica**. Dissertação de Mestrado. Pelotas. Brasil, 2009.

SUN MICROSYSTEMS. **JavaSpaces™ Specification : Especificação.** Palo Alto, Estados Unidos, 1999

TANENBAUM, A. S., VAN STEEN, M. **Distributed System. Principles and Paradigms.** 1 ed., [S.l.], Pearson Prentice Hall. 2002.

TPC. Disponível em http://www.tpc.org/tpcc/, acesso: julho de 2011.

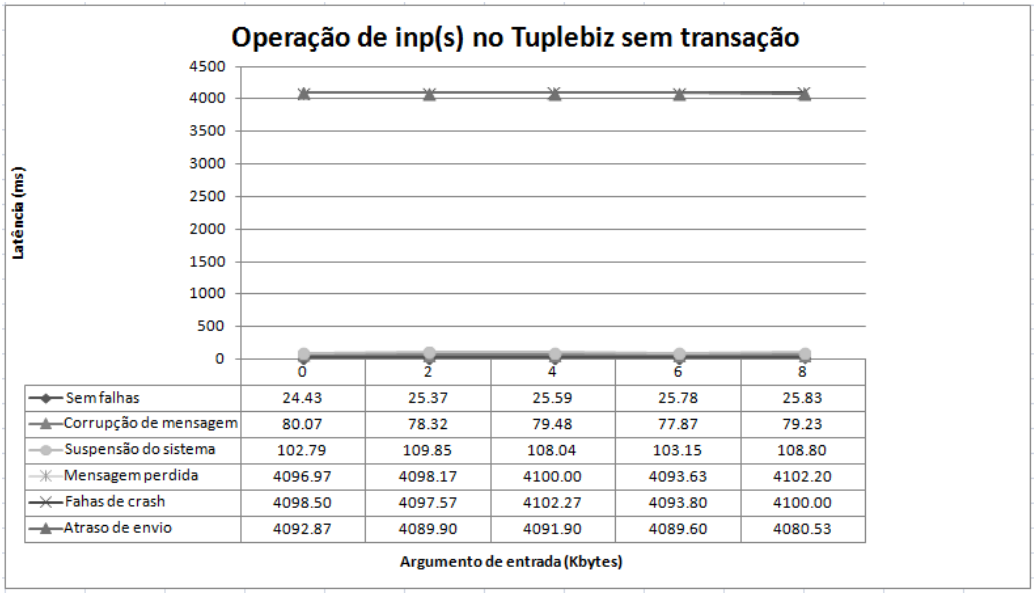
VANDIVER, B., BALAKRISHNAN, H., LISKOV, B. Tolerating Byzantine faults in transaction proceeding system using commit barrier scheduling. In: ACM SIGOPS, symposium on Operating systems principles, SOSP, 21, 2007, Washington, Estados Unidos. **Proceedings**, New York, Estados Unidos, 2007, p. 59-72.

FIEDLER, D. ET. AL. Towards the Measurement of Tuple Space Performance. ACM **SIGMETRICS Performance Evaluation Review**, New York, Estados Unidos, v. 33, n.3, p. 51-62, dez. 2005.

WEBER, T. S. **Um roteiro para exploração dos conceitos básicos de tolerância a falhas**. 2002. Disponível em http://www.inf.ufrgs.br/~taisy/disciplinas/textos/index.html, acesso dezembro 2011.

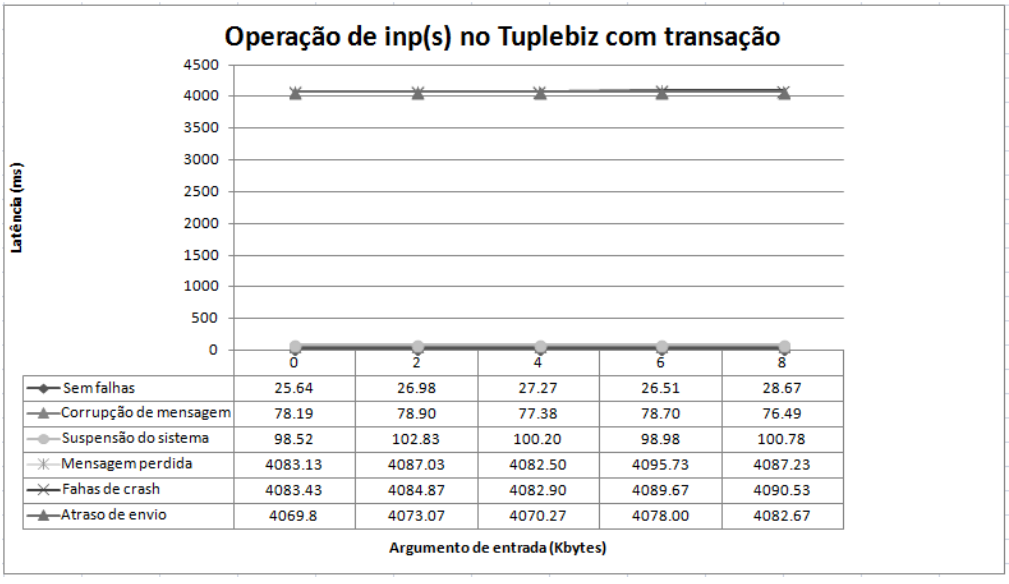
WIERMAN, S., J., NARASIMHAN, P., **Vajra: Evaluating Byzantine-Fault-Tolerant Distributed Systems.** In. KANOUN, K., SPAINHOWER, L., Dependability Benchmarking for computer systems. [S.l.], Wiley-IEEE Computer Society, 2008, p. 163-183.

APÊNDICE



	0	2	4	6	8
Sem falhas	24.43	25.37	25.59	25.78	25.83
Corrupção de mensagem	80.07	78.32	79.48	77.87	79.23
Suspensão do sistema	102.79	109.85	108.04	103.15	108.80
Mensagem perdida	4096.97	4098.17	4100.00	4093.63	4102.20
Fahas de crash	4098.50	4097.57	4102.27	4093.80	4100.00
Atraso de envio	4092.87	4089.90	4091.90	4089.60	4080.53

Figura 6.1: Injeção de falhas na operação de inp(s) fora do contexto transacional



	0	2	4	6	8
Sem falhas	25.64	26.98	27.27	26.51	28.67
Corrupção de mensagem	78.19	78.90	77.38	78.70	76.49
Suspensão do sistema	98.52	102.83	100.20	98.98	100.78
Mensagem perdida	4083.13	4087.03	4082.50	4095.73	4087.23
Fahas de crash	4083.43	4084.87	4082.90	4089.67	4090.53
Atraso de envio	4069.8	4073.07	4070.27	4078.00	4082.67

Figura 6.2: Injeção de falhas na operação de inp(s) dentro do contexto transacional

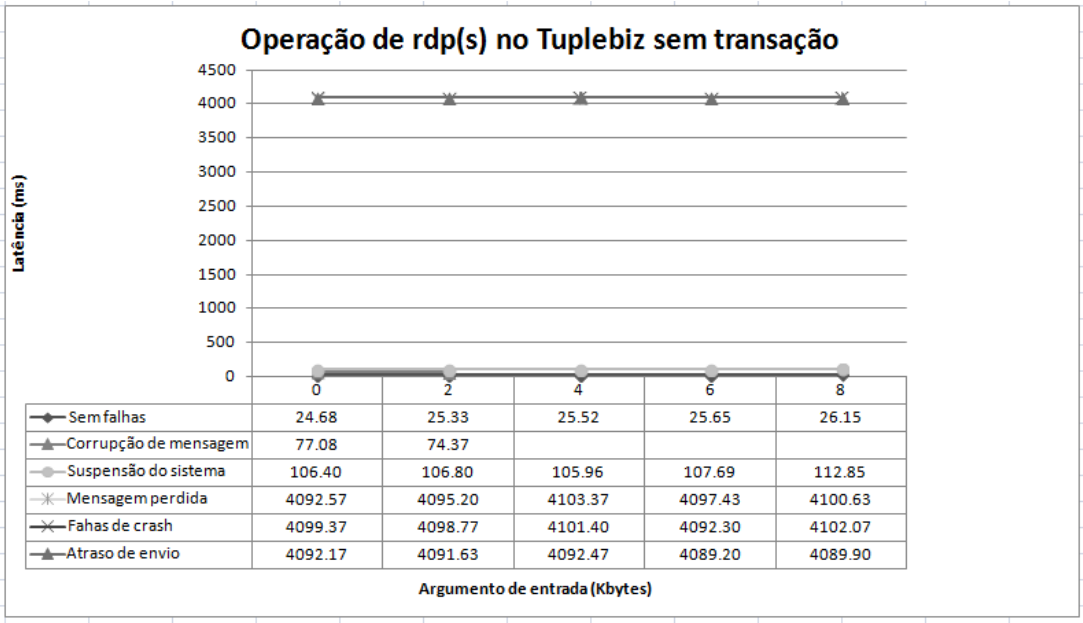


Figura 6.3: Injeção de falhas na operação de rdp(s) fora do contexto transacional

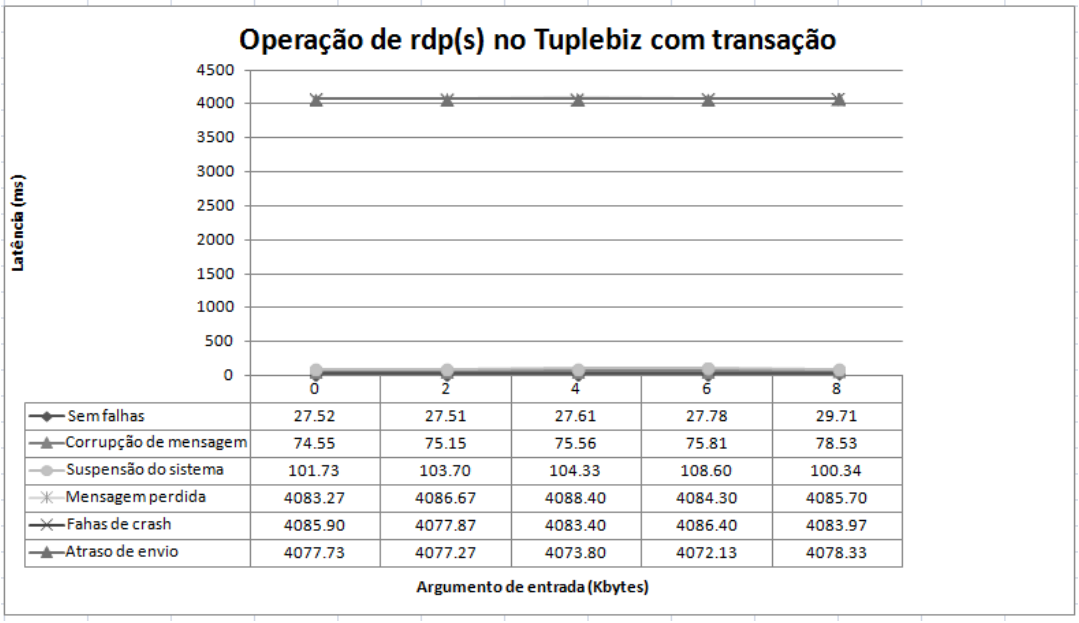


Figura 6.4: Injeção de falhas na operação de rdp(s) dentro do contexto transacional