# Uma Infraestrutura para Suporte de Memória Transacional Distribuída

Tiago M. Vale, Ricardo J. Dias, and João M. Lourenço*

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
t.vale@campus.fct.unl.pt ricardo.dias@campus.fct.unl.pt
joao.lourenco@fct.unl.pt

**Resumo** As técnicas e algoritmos desenvolvidos sobre diferentes infraestruturas específicas dificilmente podem ser comparados entre si. Este princípio também se aplica às infraestruturas para execução de Memória Transacional Distribuída (MTD), pois não só são muito escassas aquelas que permitem o desenvolvimento, teste e comparação de vários algoritmos e técnicas de implementação, como fornecem uma interface intrusiva para o programador. Sem uma comparação justa, não é possível aferir quais as técnicas e algoritmos mais apropriados em cada contexto de utilização (*workload*). Neste artigo propomos uma infraestrutura generalista, muito flexível, que possibilita a experimentação de várias estratégias de MTD, permitindo o desenvolvimento de uma grande variedade de algoritmos e de técnicas de implementação eficientes e otimizadas. Através da sua utilização, é agora possível a comparação de técnicas e algoritmos em diferentes contextos de utilização (*workloads*), recorrendo a uma única infraestrutura e com implicações mínimas no código da aplicação.

**Keywords:** Memória transacional distribuída; Replicação; Concorrência, Arquitetura de *software*, Java

## 1 Introdução

A Memória Transacional (MT) está a ganhar ímpeto como modelo de controlo de concorrência, não só devido à complexidade incorrida no uso de trincos, que é propenso a erros dada a sua natureza de baixo nível, como à falta de capacidade de composição que carateriza o uso destes. No coração deste modelo reside o conceito de *transação*, uma sequência de operações que executam mediante uma semântica de *tudo-ou-nada*, *i.e.*, ou todas as operações executam com sucesso e como se fossem uma única ação indivisível e instantânea, ou nenhuma executa. Deste modo o programador pode embrulhar o acesso à memória partilhada numa transação, efetivamente delegando o controlo dos acessos para o sistema de execução.

* Este trabalho foi parcialmente financiado pela União Europeia, no contexto da COST Action IC1001 (Euro-TM), e pela Fundação para a Ciência e Tecnologia (FCT/MCTES), no contexto dos projetos de investigação PTDC/EIA-EIA/108963/2008 e PTDC/EIA-EIA/113613/2009 e bolsa de investigação SFRH/BD/41765/2007.

Tem havido muita investigação visando a utilização de MT para controlo de concorrência num sistema centralizado, no entanto a utilização deste paradigma num contexto distribuído está ainda muito incipiente. Sistemas distribuídos são amplamente utilizados, quer por questões de performance, fiabilidade e/ou qualidade de serviço. Como a utilização de trincos neste contexto é inviável, a MT apresenta-se como uma alternativa natural.

Das várias infraestruturas de MT centralizada para a linguagem Java, a TribuSTM [7], uma extensão à Deuce [10], permite a implementação de vários algoritmos de MT. Esta é caracterizada por ser muito eficiente e disponibilizar uma interface não intrusiva para o programador de aplicações. Ambas são propriedades desejáveis e nenhuma das poucas infraestruturas existentes para Memória Transacional Distribuída (MTD) [4, 11, 13] as reúne em simultâneo. Cada uma destas infraestruturas está construída de modo a permitir exclusivamente a estratégia de memória distribuída para a qual foi desenhada, nomeadamente memória distribuída pura [11, 13] e replicação total [4]. Juntando *APIs* heterogéneas com infraestruturas diferentes, torna-se inexequível o desenvolvimento, teste e comparação justa de vários algoritmos, ou o desenvolvimento de aplicações sobre um modelo de MTD adaptável.

Para dar resposta a estes problemas, este artigo apresenta uma infraestrutura para suporte de MTD na linguagem Java. Esta foi desenhada para permitir integrar de modo eficiente e flexível vários algoritmos, estratégias ou implementações diferentes dos vários componentes que compõem a arquitetura da infraestrutura, através de interfaces bem definidas entre as várias camadas da infraestrutura e do uso de padrões de desenho como *Observer*, *Memento* e *Mediator*/*Facade* [9]. A infraestrutura proposta fornece uma interface não intrusiva para o programador de aplicações, é eficiente e, tanto quanto sabemos, é a primeira infraestrutura para MTD que permite várias estratégias de memória distribuída. Essas estratégias são obtidas através da utilização de uma técnica de injeção de metadados nos objetos em conjunto com a funcionalidade de serialização de objetos fornecida pela plataforma Java.

Na continuação deste artigo, apresentamos a base inicial do nosso trabalho na §2. Na §3 descrevemos a arquitetura do sistema, pormenorizamos a técnica conjunta de metadados e serialização bem como o suporte para transações distribuídas. Apresentamos os resultados experimentais na §4, reforçando a necessidade de uma plataforma comum, e discutimos trabalho relacionado na §5. Concluímos com alguns comentários e direções futuras na §6.

## 2 TribuSTM

Vários algoritmos de Memória Transacional (MT) foram propostos nos últimos anos. Todos eles associam informação à memória gerida (*metadados*), mas a natureza dessa informação varia consoante o algoritmo.

A Deuce [10] é uma infraestrutura de Memória Transacional (MT) para a linguagem Java que permite implementar vários algoritmos de MT, e que se destaca pela sua interface muito pouco intrusiva para o programador de aplicações. Este

apenas necessita de adicionar a anotação `@Atomic` aos métodos a executar como transações e a Deuce, de forma completamente transparente para o programador da aplicação, reescreve o *bytecode* do programa, intercetando e transferindo o controlo para a Deuce no início e o fim de transações e nos acessos à memória realizados num contexto transacional.

Contudo, a Deuce apenas permite implementar algoritmos de MT utilizando um mapeamento externo entre os campos dos objetos e os respetivos metadados transacionais mantidos pelos algoritmos. Este mapeamento é normalmente realizado através duma tabela de dispersão que, por questões de desempenho, não trata colisões, levando a uma relação N–1 entre os campos e os metadados associados. Esta é uma opção válida para, por exemplo, o algoritmo TL2 [8], em que a informação guardada nos metadados são trincos. Quando os metadados dependem dos campos, como por exemplo na JVSTM [2] que utiliza listas de versões, exige-se uma relação 1–1, pelo que a tabela de mapeamento necessita de tratar as colisões, o que não permite a implementação eficiente destes algoritmos [7].

A TribuSTM [7] estende a Deuce para endereçar os problemas identificados, adicionando ao mapeamento externo já existente, a possibilidade de inclusão (*inlining*) dos metadados na classes, juntamente com os campos a que estão associados, sem qualquer alteração na interface disponibilizada ao programador de aplicações. Deste modo obtém-se uma relação 1–1 entre os campos e os metadados, permitindo a implementação eficiente de algoritmos de MT cujos metadados dependam da localização de memória a que estão associados.

## 3 Suporte de Memória Transacional Distribuída com a TribuSTM

Suportar Memória Transacional Distribuída (MTD) sobre a infraestrutura descrita na §2 levanta novas questões, nomeadamente: (*i*) como *validar* e *aplicar* transações num contexto distribuído; (*ii*) onde guardar os dados e o metadados e como lhes aceder; e eventualmente (*iii*) considerar a migração de dados e/ou transações. Apesar destas novas dimensões, o princípio de fornecer uma interface não intrusiva ao programador é um dos principais requisitos deste trabalho.

Um dos conceitos realçados no trabalho apresentado na §2 é o de que diferentes algoritmos de MT associam diferentes tipos de *metadados* à memória gerida. Do mesmo modo podemos idealizar que diferentes estratégias de memória distribuída (puramente distribuída, total ou parcialmente replicada) podem também ser implementadas com recurso a metadados sobre as localizações de memória (ou objetos). Por exemplo, num ambiente de replicação total, cada localização de memória (ou objeto) pode ter associada um identificador único que permite reconhecer essa localização na réplica local. E num contexto de distribuição pura, esses metadados podem conter todas as informações necessárias à execução de uma chamada de procedimento remoto, dirigida ao "dono" da localização de memória associada, para realizar uma leitura ou escrita transacional.

A infraestrutura que propomos para suportar MTD utiliza a TribuSTM para controlo de concorrência interno, e dois componentes adicionais: o Gestor de Dis-

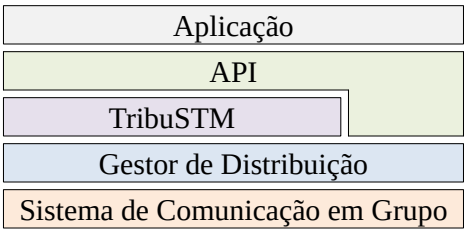



Figura 1: Componentes da arquitetura.

tribuição (GD) e o Sistema de Comunicação em Grupo (SCG), tal como ilustrado na Fig. 1. O GD implementa um sistema de memória distribuída de acordo com a política de replicação desejada (distribuição pura, replicação total ou parcial) recorrendo a metadados, bem como os protocolos de suporte à execução de transações distribuídas. No decorrer deste artigo, apenas consideraremos um GD que dê suporte a replicação total. O SCG encapsula as primitivas de comunicação entre os vários constituintes do sistema distribuído necessárias à implementação do GD, nomeadamente a difusão atómica, fornecendo-lhe uma interface uniforme independentemente do sistema utilizado em concreto. Atualmente suportamos JGroups [1] e Appia [12].

### 3.1 Metadados e Memória Distribuída

Por questões de performance, fiabilidade e/ou resistência a falhas, os dados da aplicação acedidos no contexto de transações podem estar puramente distribuídos, ou total ou parcialmente replicados. A gestão desta distribuição é feita com recurso a metadados associados às localizações de memória distribuída. Por exemplo, num ambiente de replicação total, cada localização de memória (ou objeto) necessita de ser reconhecida globalmente no contexto do sistema distribuído, podendo-se associar-lhe um identificador único (definido pelo metadado).

A infraestrutura proposta neste artigo tem por base a TribuSTM e, como tal, está implementada na linguagem Java. Uma particularidade desta consiste na sua habilidade para *serializar* e *de-serializar* objetos. Serializar um objeto consiste em transformar o seu estado numa sequência de *bytes*. O processo contrário, deserialização, resume-se a recriar um objeto a partir de uma dessas sequências. Este mecanismo pode ser usado para enviar objetos de um computador para outro através de uma rede que os interliga.

Vejamos como é que estes dois conceitos nos permitem implementar objetos distribuídos, e em particular, replicados.

**Metadados** Independentemente da estratégia de objetos distribuídos que se pretende implementar, é sempre necessário associar-lhes metadados. Esses metadados são definidos por uma qualquer subclasse de `Metadata` definida pelo programador do GD. As classes da aplicação cujos objetos são distribuídos devem disponibilizar acesso aos metadados associados, portanto definimos uma interface `DistributedObject` com os métodos `getMetadata` e

`setMetadata(Metadata)` a ser implementada pelas classes dos objetos distribuídos.

Para implementar um ambiente de replicação total, em que todos os elementos do sistema distribuído (réplicas) mantêm uma cópia local de todos os dados, cada objeto deve ser identificável no contexto global do sistema. Num sistema centralizado um objeto pode ser univocamente identificado pelo seu endereço, mas tal não se aplica num contexto distribuído. Neste caso, o metadado do objeto em vez do endereço contém um identificador global e único para cada objeto.

**Serialização** A plataforma Java permite um controlo fino por parte do programador sobre o processo de (de-)serialização. Em particular, autoriza que a classe de um objeto $O$ nomeie outro objeto $S$, potencialmente até de uma classe distinta, como seu substituto antes da transformação numa série de *bytes* *i.e.*, a serialização de $O$ consiste em transformar não o seu estado numa série de *bytes*, mas sim o de $S$. No caso da de-serialização, a plataforma Java permite trocar o objeto $O$ recriado a partir da série de *bytes* por outro objeto $S$, *i.e.*, a de-serialização de $O$ não resulta na recriação de $O$, mas sim em $S$. Para esse efeito, basta o objeto $O$ implementar os métodos `writeReplace` e `readResolve`, sendo o objeto devolvido por estes ($S$) serializado ou de-serializado em vez de $O$, respetivamente.

Vejamos como podemos explorar este mecanismo para implementar a replicação de objetos. Considere-se que um objeto replicado $O$ pode estar em dois estados, nomeadamente (*i*) *publicado*; ou (*ii*) *privado*. No primeiro caso já lhe foi atribuído um identificador *oid* (metadado), o que não se verifica no segundo. A serialização de um objeto privado $O$ consiste em gerar o respetivo *oid* e efetivamente serializar $O$. Se este já estiver publicado, basta-nos nomear *oid* como seu representante, serializando *oid* em vez de $O$. Nesse caso, a de-serialização de *oid* devolve o objeto $O'$ tal que $id(O') = oid$, *i.e.*, devolve o objeto correspondente à réplica local com identificador *oid*. A de-serialização de um objeto privado $O$ devolve o próprio $O$. Isto significa que o um objeto já publicado nunca é enviado pela rede, apenas o seu identificador, resultando numa diminuição potencialmente grande no tamanho das mensagens.

Sendo o nosso objetivo não só fornecer uma infraestrutura flexível como uma interface não intrusiva para o programador, apresentamos de seguida todas as transformações aplicadas automaticamente a qualquer classe da aplicação para suportar objetos distribuídos. Seja $C^I = \{c_1, \ldots, c_n, m_1, \ldots, m_k\}$ a classe $C$ que implementa as interfaces $I$ onde $c_i$ e $m_j$, com $i \leq n$ e $j \leq k$, representam os campos e métodos da classe, respetivamente. A transformação efetuada a qualquer classe $C$ da aplicação consiste em torná-la uma implementação de `DistributedObject` e ser-lhe adicionada um novo campo $c^m$ do tipo `Metadata`,

bem como os métodos `writeReplace` e `readResolve`. Ou seja:

$$C^{I \cup \texttt{DistributedObject}} = \left\{ \begin{array}{c} c^m \\ c_1, \ldots, c_n \\ m_1, \ldots, m_k \\ \texttt{getMetadata}, \texttt{setMetadata} \\ \texttt{writeReplace}, \texttt{readResolve} \end{array} \right\}$$

### 3.2 Transações Distribuídas

Os protocolos baseados em certificação afiguram-se como interessantes a serem usados no contexto de MTD totalmente replicada, por não necessitarem de sincronização entre as réplicas durante a execução de transações. Dependendo do protocolo ser com ou sem votação, a coordenação entre réplicas é apenas necessária durante a fase de confirmação de uma transação, e consiste numa difusão atómica, acrescida de mais uma difusão (não atómica) no caso do protocolo sem votação. Em qualquer das variantes é garantido que todas as réplicas irão processar eventuais transações concorrentes pela ordem total imposta pela difusão atómica, sendo sempre tomada a mesma decisão (confirmar ou abortar) em todas as réplicas para cada uma das transações.

Para permitir a integração flexível de vários protocolos com a camada de MT, esta e o GD interagem através de duas interfaces distintas e bem definidas. A primeira destina-se a propagar eventos provocados pela aplicação na camada de MT através do padrão *Observer* [9]. Por exemplo, quando um fluxo de execução da aplicação desencadeia a *confirmação* de uma transação, esse evento pode fluir para o GD que despoleta o protocolo de certificação. Ou se pensarmos num ambiente onde não existe replicação total, uma *leitura* de um campo de um determinado objeto pode fazer com que o GD execute um procedimento remoto ao "dono" do objeto para obter o valor. A segunda interface entre a camada de MT e o GD têm dois objetivos. O primeiro é permitir ao GD obter uma representação opaca do estado de uma transação (padrão *Memento* [9]) que torne possível a reconstrução desta nas outras réplicas do sistema. A opacidade desta representação é fulcral para permitir a interoperabilidade do GD com variados algoritmos de MT, dos quais o estado de uma transação depende. O segundo objetivo é disponibilizar ao GD a possibilidade de interagir com a memória gerida pela camada de MT, e.g., através da aplicação das escritas de uma transação no decorrer do protocolo de certificação.

Em concreto, a interface bidirecional utilizada para interação entre o componente de MT e o GD, bem como a interface utilizada entre o GD e o SCG, estão descritas nas Tabelas 1 e 2, respetivamente.

Na Fig. 2 podemos ver o pseudocódigo do protocolo de certificação sem votação, implementado através das interfaces definidas anteriormente, que apresentamos de seguida descrevendo informalmente o fluxo de execução de uma transação por parte da aplicação.

Um fluxo de execução da aplicação inicia uma transação através da invocação do método BEGIN na camada de MT (Fig. 2a). Depois de efetuadas todas as

Tabela 1: Interface de interação entre o componente de MT e o GD.

| Função | Descrição |
|---|---|
| createState($T$) | O componente de MT devolve uma representação do estado da transação $T$, constituída pelos conjuntos de escritas e leituras, um identificador local *id*, bem como outra informação dependente do algoritmo de MT. |
| recreateTx($S$) | O componente de MT devolve uma transação recriada a partir de um estado $S$. |
| validate($T$) | O componente de MT valida a transação $T$, devolvendo a informação de sucesso ou insucesso. |
| applyWS($T$) | O componente de MT aplica todas as atualizações efetuadas pela transação $T$. |
| processed($T$, $r$) | O componente de MT é notificado que o processamento distribuído da transação $T$, por parte do GD, terminou com o resultado $r$. |
| onBegin($T$) | O GD é notificado do início da transação $T$. |
| onCommit($T$) | O GD é notificado do início da fase de confirmação da transação $T$. Pode ser usado para desencadear o protocolo de certificação. |

Tabela 2: Interface de interação entre o GD e o SCG.

| Função | Descrição |
|---|---|
| ABcast($M$) | Disponibiliza ao GD a difusão atómica de uma mensagem $M$. |
| onABDeliver($M$, $E$) | O GD é notificado da recepção de uma mensagem $M$ difundida atomicamente pelo emissor $E$. |
| self($E$) | Permite ao GD saber se o emissor $E$ é ele próprio ou outra réplica. |

inicializações necessárias, o fluxo de execução da aplicação executa a transação localmente. Quando chega ao fim, o fluxo de execução da aplicação solicita a confirmação da transação ao componente de MT (commit), que desencadeia o protocolo de certificação da transação (onCommit, Fig. 2b). Neste ponto o fluxo de execução da aplicação espera pelo término da certificação, que irá validar e confirmar a transação ou abortá-la.

A difusão atómica efetuada pela certificação é recebida por todas as réplicas (onABDeliver). A réplica emissora processa a transação local correspondente, enquanto que as outras réplicas recriam a transação através do estado recebido na difusão atómica. Deste modo, não há distinção entre o processamento de transações locais e remotas, simplificando a implementação do componente de MT. Todas as réplicas procedem à validação da transação e subsequente aplicação em caso de confirmação. No caso da réplica à qual a transação é local (onde foi executada), a notificação de que a transação foi processada (processed) desbloqueia o fluxo de execução da aplicação que esperava pela decisão do protocolo de certificação.

```
committed ← false

procedure BEGIN(T)
    ...
    onBEGIN(T)

function COMMIT(T)
    onCOMMIT(T)
    esperar até T ser processada
    return committed

procedure PROCESSED(T, r)
    committed ← r
    T foi processada
```

(a) Componente de Memória Transacional.

```
localTxs ← ∅

procedure onCOMMIT(T)
    localTxs ← localTxs[T.id ↦ T]
    S ← createSTATE(T)
    ABCAST(S)

procedure onABDELIVER(S, E)
    if SELF(E) then
        T ← localTxs(S.id)
        localTxs ← localTxs \ T
    else
        T ← RECREATETX(S)
    valid ← VALIDATE(T)
    if valid then
        APPLYWS(T)
    PROCESSED(T, valid)
```

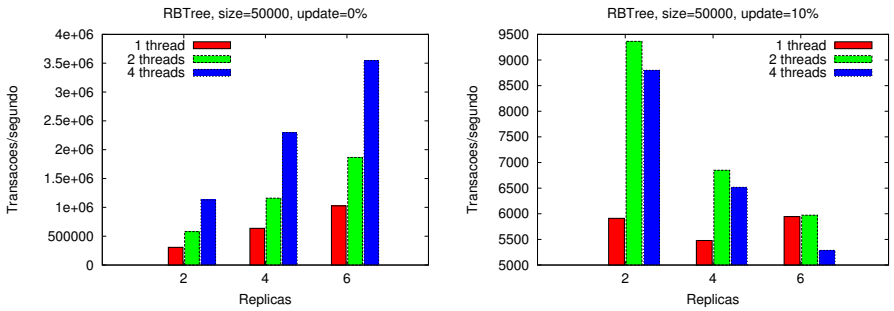
(b) Gestor de Distribuição.

Figura 2: Pseudocódigo do protocolo de certificação sem votação.

## 4 Resultados Experimentais

Os testes foram efetuados num sistema com seis computadores com 2×Quad-Core AMD Opteron 2376, cada um a 2.3Ghz e com 4×512KB *cache* L2. Os computadores executam Linux 2.6.26-2-amd64, Debian 5.0.8, e estão ligados em rede via Ethernet Gigabit privada. A versão da plataforma Java utilizada foi a 6, em concreto OpenJDK Runtime Environment (IcedTea6 1.8.3) (6b18-1.8.3-2 lenny1). Na nossa infraestrutura, ao nível da camada de MT foi escolhido o algoritmo TL2. O protocolo de certificação sem votação (Fig. 2) foi utilizado para manter a consistência entre réplicas. A implementação ao nível do SCG foi o JGroups 3.1.Beta1. Este foi configurado com o protocolo SEQUENCER para ordenação total das mensagens, e os *buffers* de envio e receção *unicast* e *multicast* configurados com capacidade 100K e 64K, respetivamente.

Apresentam-se agora resultados para o micro-teste Red Black Tree (incluído na Deuce). O teste é composto por três tipos diferentes de transações, nomeadamente inserções, remoções e pesquisas. O teste foi configurado para um tamanho inicial de 50000 elementos no intervalo $[0; 200000]$ sendo a probabilidade de contenção baixa. A Fig. 3b mostra o débito de transações do sistema (eixo das ordenadas) variando o número de réplicas entre 2 e 6 (eixo das abcissas) e fluxos de execução em cada uma entre 1 e 4, numa configuração com 10% de transações de escrita. Podemos observar que o desempenho do sistema se degrada com o aumento absoluto de fluxos de execução, especialmente quando o aumento é feito à custa de novas réplicas. Em particular, ao passarmos de 2 para 4 réplicas observa-se um decréscimo de desempenho. A Fig. 3a mostra que o sistema numa utilização centralizada escala (configuração com transações apenas de leitura, que executam exclusivamente na réplica) pelo que a contenção se encontra, como é intuitivo, na comunicação em rede. Cremos que o decrescimento de desempenho assenta essencialmente nas duas premissas seguintes.




(a) Sem acesso à rede.

(b) Com acesso à rede.

Figura 3: Débito do sistema.

**Injustiça** O protocolo SEQUENCER baseia-se num coordenador centralizado (*sequenciador*) que implementa ordem total reencaminhando todas as mensagens para este, que as difunde para o grupo em nome do emissor original. Neste esquema, as mensagens oriundas do coordenador são difundidas utilizando um passo de comunicação a menos relativamente a todos os outros membros do grupo.

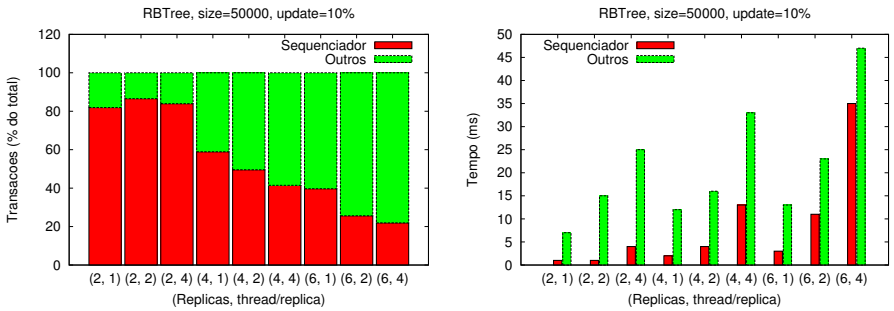
**Desproporcionalidade** A duração da execução de uma transação é uma ordem de grandeza mais baixa que a de comunicação na rede.

A combinação da injustiça do protocolo com a rapidez da execução de transações, em contraste com a difusão de uma transação e subsequente confirmação, faz com que a quantidade de mensagens do sequenciador que são difundidas, e consequentemente o número de transações executadas, seja muito superior à dos restantes nós. Com consequência, o pico de desempenho que o sistema apresenta com 2 réplicas é fruto de este se comportar de maneira próxima ao de um centralizado. Como podemos ver na Fig. 4a, no caso em que temos 2 réplicas (eixo das abcissas) o sequenciador é responsável por cerca de 80% das transações do sistema (eixo das ordenadas). Isto realça a importância da utilização de uma implementação de ordem total que seja justa na difusão.

O aumento do número de fluxos de execução em absoluto no sistema é diretamente proporcional à taxa de transações que necessitam de ser certificadas. Dada a premissa da desproporcionalidade, uma maior taxa de transações para certificar aumenta a latência (Fig. 4b) entre a invocação da difusão de uma transação e a sua receção, resultando num decréscimo de desempenho pelo aumento de tempo de computação desperdiçado.[1]

A avaliação efetuada reforça a necessidade de uma plataforma comum que permita a recriação dos ambientes de teste, para o desenvolvimento e aferição justa de novos algoritmos. Os resultados indicam também o impacto que diferentes implementações dos mesmos serviços, neste caso o suporte para ordem

[1] Este problema é endereçado em [5].




(a) Contribuição de cada réplica para o total de transações do sistema.

(b) Lâtencia na receção de uma transação difundida.

Figura 4: Estatísticas de execução.

total utilizado, podem ter no desempenho do sistema e consequentemente na comparação de diferentes contribuições.

## 5 Trabalho Relacionado

Em [11] propõe-se a plataforma DiSTM para facilitar o teste de protocolos de coerência para MTD não replicada. No modelo de execução deste sistema existe um elemento mestre onde os dados globais estão concentrados. Os outros elementos atuam como trabalhores, *i.e.* executam transações, e mantêm uma *cache* dos dados globais. As transações executam localmente mas as atualizações são efetuadas no mestre, que imediatamente atualiza todas as *caches* abortando quaisquer transações que não validem contra a atualização. São testados três protocolos de coerência, dois centralizados que usam um esquema de permissões sobre os dados, e um distribuído. A nossa infraestrutura distingue-se desta por oferecer suporte a replicação e não impor um modelo mestre-escravo, mas a sua flexibilidade permite implementar um sistema equivalente bem como os protocolos de coerência testados. A interface que expomos ao programador para definir transações consiste numa anotação `@Atomic`, enquanto que na DiSTM é necessário utilizar uma classe `Thread` específica da plataforma, ao qual se passam `Callable`s cujo código é executado como uma transação. Na DiSTM, classes cujos objetos são distribuídos têm que ser programadas com recurso a interfaces anotadas e a instanciação de tais objetos usando o padrão *Factory* [9] em vez da palavra-chave `new`. Na nossa abordagem nada tem que ser feito, pois as modificações necessárias são efetuadas automaticamente pela infraestrutura.

A plataforma em [13] fornece suporte à execução de MTD não replicada, e também permite a utilização de vários protocolos de coerência. Adicionalmente é possível empregar diversos protocolos para pesquisa dos objetos distribuídos através de um serviço de diretório, em contraste com a abordagem centralizada da infraestrutura anterior. Esta infraestrutura permite a execução de transações

distribuídas sob um modelo de migração de dados ou de fluxo de controlo, *i.e.* os dados migram para junto do fluxo de execução, ou o fluxo de execução (a transação) migra para junto dos dados para lhes aceder. O modelo de programação oferecido delega no programador o suporte para identificação unívoca de objetos no sistema distribuído e a própria geração dos identificadores. O programador passa agora a manipular identificadores em vez de referências Java normais, de tal modo que para aceder aos objetos tem que explicitamente utilizar a *API* de serviço de diretório da plataforma que, dado o identificador de um objeto, retorna a referência para o objeto. Na nossa solução, o modelo de programação não é alterado. Em relação a migração de objetos e/ou transações, apesar de fora do contexto deste artigo, cremos que a técnica conjunta de metadados e serialização é flexível ao ponto de permitir a sua implementação na nossa infraestrutura.

Por último, a GenRSTM [4] é uma infraestrutura para MTD totalmente replicada. Esta é composta por três principais componentes, nomeadamente (*i*) camada de Memória Transacional (MT); (*ii*) Gestor de Replicação (GR); e (*iii*) Sistema de Comunicação em Grupo (SCG). A camada de MT é baseada na JVSTM [2], embora permita outros algoritmos de MT desde que implementados sob a mesma abstração de *versioned boxes*. O GR implementa um protocolo para manter a consistência das réplicas utilizando as primitivas de comunicação fornecidas pelo SCG. Vários protocolos recentemente propostos para serem utilizados no contexto de MTD [3, 5, 6] foram implementados nesta infraestrutura. Objetos replicados são explicitamente definidos pelo programador, tendo as classes respetivas que estender uma fornecida pela plataforma, `GenRSTMObject`. Das três infraestruturas apresentadas, esta é aquela da qual a nossa se aproxima mais em termos arquiteturais. Contudo, a nossa camada de MT baseia-se na Deuce pelo que a interface fornecida pela GenRSTM é bastante intrusiva comparada com a nossa. As interfaces de interação do GR com a camada de MT e SCG são na sua generalidade equivalentes às nossas. A diferença encontra-se na técnica usada para implementar objetos replicados. Na GenRSTM isso está encapsulado em `GenRSTMObject` e apenas permite replicação total. Os protocolos para gestão da coerência que foram implementados na GenRSTM podem ser implementados na nossa infraestrutura.

## 6 Conclusões

A infraestrutura apresentada neste artigo é, tanto quanto sabemos, a primeira para MTD na plataforma Java que fornece uma interface não intrusiva ao programador e permite várias estratégias de distribuição. Isso é alcançado através da conjunção entre uma técnica de metadados associados a objetos que são injetados automaticamente e o mecanismo de serialização do Java.

No seu estado atual, a infraestrutura permite integrar diferentes protocolos de consistência de réplicas e algoritmos de MT, de forma completamente transparente para a aplicação. A sua arquitetura simplifica o desenvolvimento e avaliação de algoritmos de suporte à execução num contexto de MTD replicada ou não, através do encapsulamento dos três principais componentes que comuni-

cam através de interfaces bem definidas. Direções futuras, no seguimento desta contribuição, incluem a migração de transações e/ou objetos, replicação parcial e ambientes *cloud*.

A infraestrutura será disponibilizada como *software* de código aberto com o objetivo de ajudar a comunidade a prototipar e testar com facilidade novos algoritmos para MTD, bem como a compará-los de forma justa. Para isso é imperativo a existência de uma plataforma comum, tese essa que os resultados apresentados apoiam.

## Referências


1. Bela Ban. JGroups - A Toolkit for Reliable Multicast Communication. `http://www.jgroups.org`, June 2012.
2. João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, December 2006.
3. Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous Lease-Based Replication of Software Transactional Memory. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, pages 376–396, 2010.
4. Nuno Carvalho, Paolo Romano, and Luís Rodrigues. A Generic Framework for Replicated Software Transactional Memories. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, pages 271–274, 2011.
5. Nuno Carvalho, Paolo Romano, and Luís Rodrigues. SCert: Speculative certification in replicated software transactional memories. In *Proceedings of the Annual International Systems and Storage Conference (SYSTOR)*, 2011.
6. Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. $D^2$STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 307–313, 2009.
7. Ricardo J. Dias, Tiago M. Vale, and João M. Lourenço. Efficient Support for In-Place Metadata in Transactional Memory. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2012.
8. David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
10. Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.
11. Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 51–58, 2008.
12. Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Poster on the International Conference on Distributed Computing Systems (ICDCS)*, pages 707–710, 2001.

13. Mohamed Saad and Binoy Ravindran. HyFlow: a high performance distributed software transactional memory framework. In *Student Research Posters of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 265–266, 2011.