REPEATS - Uma Arquitetura para Replicação Tolerante a Faltas Bizantinas baseada em Espaço de Tuplas

Aldelir Fernando Luiz[1], **Alysson Neves Bessani**[3],
Lau Cheuk Lung[2], **Tatiana Filgueiras**[1]

[1]PPGIA - Pontifícia Universidade Católica do Paraná - Curitiba - Paraná

[2]INE - Departamento de Informática e Estatística - UFSC - Santa Catarina

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

aldelir@ppgia.pucpr.br, bessani@di.fc.ul.pt, lau.lung@inf.ufsc.br

Abstract. *One of the greatest challenges of the modern computing is to build systems more robust, reliable and safe. An approach that has been exploited to achieve that is to employ models and protocols that provide byzantine fault tolerant guarantees to the services. The main advantage of this approach is that it provides means to implements reliable services. This papers describes* REPEATS*, an architecture for state machine replication based on tuple space coordination paradigm. In the* REPEATS *architecture, the tuple space is used to provide the communication support and a stable storage system for shared data used by the replicas.*

Resumo. *Um dos grandes desafios da computação moderna é tornar os sistemas mais robustos, confiáveis e seguros. Uma abordagem para enfrentar este desafio que tem sido explorada nos últimos anos é o uso de protocolos e modelos que permitam aos serviços tolerarem faltas arbitrárias, também chamadas bizantinas. Este trabalho apresenta uma arquitetura para replicação tolerante a faltas bizantinas em sistemas distribuídos baseado no modelo de coordenação generativa. Esta arquitetura, chamada* REPEATS*, é construída sobre um espaço de tuplas com segurança de funcionamento que fornece um suporte de comunicação e meio de armazenamento para informações compartilhadas entre as réplicas do serviço confiável.*

1. Introdução

A construção de aplicações distribuídas tolerantes a faltas utilizando técnicas de redundância tem sido investigada por mais de vinte cinco anos [Lamport et al. 1982, Schneider 1990]. Esta técnica é bastante atrativa por permitir implementar serviços redundantes capazes de atender requisitos de confiabilidade, integridade e disponibilidade. Na prática, uma solução que vem sendo empregada na construção de sistemas distribuídos com segurança de funcionamento é a adaptação de técnicas de redundância na aplicação, e em seu ambiente de execução. Esta redundância é implementada através da abordagem de Replicação Máquina de Estado (RME) [Lamport 1978, Schneider 1990], que combina uma série de mecanismos que contribuem para a manutenção da disponibilidade e integridade das aplicações, bem como dos ambientes de execução. Recentemente, muitos trabalhos têm produzido soluções práticas para replicação Máquina de Estado com foco à tolerância a faltas bizantinas [Castro and Liskov 2002, Castro et al. 2003, Yin et al. 2003],

visando tornar as aplicações resistentes inclusive a ataques maliciosos bem sucedidos contra o sistema (também chamados intrusões), tornando assim os serviços **tolerantes a intrusões** [Veríssimo et al. 2003]. Algumas destas pesquisas têm demonstrado inclusive a viabilidade de se utilizar estas soluções para aumentar a confiabilidade de serviços reais [Castro and Liskov 2002, Castro et al. 2003, Yin et al. 2003].

No entanto, as características dinâmicas dos ambientes de comunicação e computação, principalmente em sistemas distribuídos de larga escala, introduzem alguns problemas que prejudicam a segurança e a integridade das aplicações. Tornando assim, o desenvolvimento de mecanismos para RME para esses sistemas muito mais complexo. Por outro lado, a comunidade vem investigando modelos de programação alternativos [Cabri et al. 2000], que permitam e/ou facilitem o desenvolvimento de aplicações complexas, capazes de se adaptar às condições adversas destes tipos de ambientes. O paradigma de **coordenação** [Gelernter and Carriero 1992] tem como premissa a separação das atividades dos sistemas em coordenação e computação. Neste contexto, o **modelo de coordenação generativa** (ou coordenação por espaço de tuplas) [Gelernter 1985] tem se mostrado como uma alternativa ao modelo de comunicação por passagem de mensagens devido ao seu poder e simplicidade: poucas e simples operações fornecem um modelo de comunicação desacoplado tanto no tempo quanto no espaço [Cabri et al. 2000], i.e., os processos comunicantes não precisam estar ativos ao mesmo tempo nem tampouco conhecer os endereços uns dos outros para se comunicarem.

Uma das grandes dificuldades em se implementar uma arquitetura para RME tolerante a faltas bizantinas está no suporte que implementa os protocolos de acordo responsáveis por definir, para os servidores replicados, uma ordem atômica de execução das requisições clientes. As soluções usualmente propostas na literatura [Schneider 1990, Castro and Liskov 2002] envolvem um alto custo de hardware e software, pois os algoritmos impõem a restrição de que para se tolerar f servidores faltosos, o número mínimo de réplicas é $3f + 1$. Em [Yin et al. 2003] é proposta uma nova abordagem para reduzir esse custo para $2f + 1$ réplicas do serviço. A abordagem proposta consiste em separar a camada de acordo (que implementa o protocolo de difusão com ordem total tolerante a faltas bizantinas) da camada de execução de requisições em conjuntos de servidores distintos. Ou seja, utilizar um conjunto de servidores para estabelecer a ordem das mensagens (executando os protocolos de acordo e portanto requerendo $3f + 1$ servidores) e um outro conjunto para executar as requisições clientes (requer apenas $2f + 1$ servidores).

Este artigo apresenta o REPEATS (*Replication over Policy-Enforced Augmented Tuple Space*), uma nova arquitetura para RME tolerante a faltas bizantinas fundamentado no modelo PEATS (*Policy-Enforced Augmented Tuple Space*) [Bessani et al. 2006], onde os processos (tanto clientes quanto as réplicas do serviço) coordenam-se através de uma abstração de alto nível: um espaço de tuplas resistente à faltas bizantinas. Da mesma forma que o trabalho de [Yin et al. 2003], o uso desta abstração nos permite também separar as entidades responsáveis pelo acordo, implementado pelo espaço de tuplas, daquelas responsáveis pela execução das requisições enviadas pelos clientes, com a vantagem de se ter algoritmos de replicação modulares e muito mais simples. O modelo requer apenas $2f + 1$ réplicas para um serviço, e é genérico o suficiente para comportar diversos conjuntos de serviços (com diferentes aplicações) compartilhando o mesmo suporte de comunicação e coordenação (o espaço de tuplas), de modo que as particularidades de um serviço não interferem nas outras. O REPEATS é, até onde sabemos, o primeiro trabalho

a propor a utilização de abstrações de mais alto nível, neste caso um espaço de tuplas, para a construção de serviços replicados.

A fim de validar a arquitetura proposta, foi implementado um protótipo do REPEATS tendo como base o DEPSPACE, um middleware de espaço de tuplas tolerante a faltas bizantinas [Bessani et al. 2007]. Da mesma forma que em outros trabalhos na área (ex. [Castro and Liskov 2002, Yin et al. 2003]), este protótipo foi utilizado para implementar um serviço NFS tolerante a faltas bizantinas, e os resultados preliminares de nosso experimentos demonstram que o uso do REPEATS incorre em um custo moderado em termos de desempenho, quando comparado com um NFS não replicado (e não tolerantes a faltas).

2. Espaço de Tuplas Aumentado e Protegido por Políticas

O espaço de tuplas é uma abstração de memória compartilhada útil para a coordenação de processos, bem como para o armazenamento de dados. Esta abstração é oriunda do modelo de **coordenação generativa** e teve sua primeira implementação na linguagem LINDA [Gelernter 1985]. No espaço de tuplas é possível realizar o armazenamento e a recuperação de estruturas de dados genéricas sob a forma de tuplas. Uma **tupla** $t = \langle f_1, f_2, ..., f_n \rangle$ é composta por uma seqüência de campos. Um campo f_i de uma tupla pode conter um valor definido, um formal (variável) "?" ou ainda um símbolo especial "*". Um campo formal é usado para extrair conteúdos individuais dos campos de uma tupla, já os símbolos especiais são usados para representar campos sem valor definido. Uma tupla t cujos campos têm valores definidos é denominada **entrada**. Uma tupla que possui algum campo formal "?" e/ou um campo especial "*" é denominada **molde**, e é representada por $\bar{t}$. Um molde $\bar{t}$ **combina** com uma entrada t se ambas as tuplas têm o mesmo número de campos e todos os campos com valores definidos de $\bar{t}$ contém o mesmo valor do campo correspondente em t. Por exemplo, uma tupla $\langle \text{“}RePEATS\text{”}, 2008 \rangle$ combina com os moldes $\langle \text{“}RePEATS\text{”}, * \rangle$, $\langle *, 2008 \rangle$ e $\langle *, * \rangle$, mas não com $\langle *, 2007 \rangle$.

As manipulações no espaço de tuplas são realizadas por meio de invocações de três operações [Gelernter 1985]: $out(t)$ para inserção de uma tupla t no espaço de tuplas; $in(\bar{t})$ para a retirada da tupla t, que combina com o molde $\bar{t}$, do espaço de tuplas; e $rd(\bar{t})$ para a leitura da tupla t (que combina com $\bar{t}$) sem retirá-la do espaço. As operações de leitura e remoção são não-deterministas, isto é, caso exista um conjunto de tuplas que combine com o molde especificado, qualquer uma delas pode ser escolhida como resposta da operação. As operações *in* e *rd* são bloqueantes, e caso não exista nenhuma tupla t que corresponda ao molde $\bar{t}$ no espaço o processo permanece bloqueado até que uma tupla que combine com o molde seja inserida para que a operação seja completada. Além destas operações básicas, variantes não bloqueantes das operações de leitura são também suportadas. Estas operações são denominadas inp e rdp, e possuem o mesmo comportamento de suas versões bloqueantes, exceto pelo fato delas retornarem mesmo que não exista nenhuma tupla que combine com o molde fornecido (elas retornam um valor booleano que sinaliza se uma tupla foi encontrada ou não).

Além das operações descritas, é util incluir na especificação do espaço de tuplas uma operação de inserção condicional denominada *Conditional Atomic Swap*, denotada por $cas(\bar{t}, t)$ [Segall 1995]. Esta operação recebe como argumentos um molde $\bar{t}$ e uma entrada t, e executa de forma indivisível o código: **if** $\neg rdp(\bar{t})$ **then** $out(t)$. Isto significa que, se a leitura do molde $\bar{t}$ falhar, então a inserção da tupla t é realizada no espaço,

caso contrário, a tupla que combina com o molde $\bar{t}$ é retornada ao processo que invocou a operação. O uso da operação cas permite que o espaço de tuplas seja capaz de resolver o problema de consenso em ambientes assíncronos [Segall 1995].

O modelo clássico de coordenação por espaço de tuplas não provê mecanismos capazes de lidar com processos maliciosos acessando o espaço de tuplas. Este problema foi resolvido com a introdução do PEATS (*Policy-Enforced Augmented Tuple Space*) [Bessani et al. 2006], que consiste em um espaço de tuplas onde as interações entre os processos são reguladas por **políticas de acesso de granularidade fina**. Estas políticas, que são usadas como mecanismo de controle de acesso ao espaço de tuplas, são compostas por um conjunto de regras que são definidas por meio da especificação de padrões de invocação para operações no espaço de tuplas e condições que devem ser satisfeitas para que estas invocações possam ser executadas, ou de outra forma negadas. Para tanto, o PEATS considera os dados oriundos da invocação (o processo invocador e os parâmetros da invocação) e o estado do atual do espaço.

3. A Arquitetura REPEATS

3.1. Visão Geral do REPEATS

O REPEATS consiste em uma concretização de replicação **Máquina de Estados** [Schneider 1990] tendo como elemento de comunicação entre os processos envolvidos um PEATS, dando origem então a uma arquitetura de suporte a replicação tolerante a faltas bizantinas.

Em nossa abordagem, o PEATS fornece o suporte de comunicação desacoplado para as interações entre os clientes e as réplicas de um serviço, além de um ambiente de armazenamento estável e seguro para as informações compartilhadas entre as réplicas. Na figura 1, os clientes inserem suas requisições na forma de tuplas no PEATS (passo 1) e as réplicas do serviço que está sendo acessado lêem estas tuplas do PEATS para obter as requisições a serem executadas (passo 2). Em seguida, os serviços replicados processam as requisições e enviam os resultados também na forma de tuplas no PEATS (passo 3), para que os clientes possam obter as respostas (passo 4).

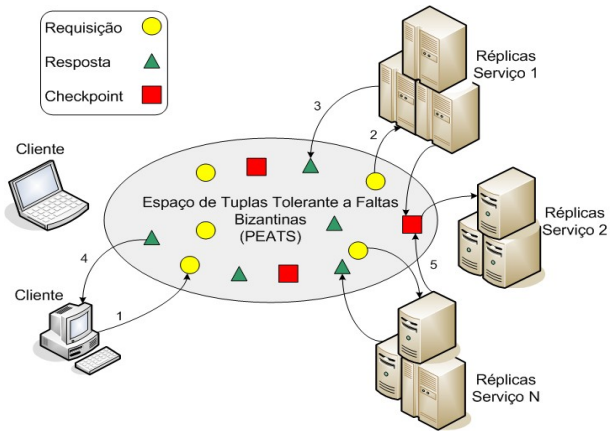


Figura 1. Modelo de execução do REPEATS

Um premissa do REPEATS é o **determinismo de réplica** [Schneider 1990]. Este requisito define que réplicas partindo de um mesmo estado inicial e sujeitas a execução de

uma mesma seqüência de operações, devem chegar ao mesmo estado final. Assim, em um sistema onde as réplicas implementam um serviço determinista, esta propriedade é implementada por meio do uso de protocolos de **difusão com ordem total** [Défago et al. 2004], que garantem que todas as operações enviadas ao sistema são processadas por todas as réplicas (acordo) em uma mesma ordem (ordem total). A partir daí, cada réplica executa a operação, atualiza seu estado (quando há necessidade) e envia ao cliente o resultado obtido. O cliente aceita o resultado da operação caso receba $f + 1$ respostas iguais de diferentes réplicas, considerando f o número máximo de servidores que podem sofrer faltas bizantinas. No REPEATS, o algoritmo de difusão com ordem total é implementado através do PEATS.

3.2. Modelo de Sistema

O modelo de sistema adotado consiste em um **sistema assíncrono** onde existem dois conjuntos de processos: $C = \{c_1, c_2, ...\}$ com um número arbitrário de clientes e $S = \{s_1, s_2, ..., s_n\}$ com n servidores que implementam réplicas do serviço replicado[1].

Em relação ao modelo de falhas de processos, admitimos que um número arbitrário de clientes e até $f \leq \lfloor \frac{n-1}{2} \rfloor$ servidores podem falhar em suas especificações apresentando **faltas bizantinas** [Lamport et al. 1982]: os processos faltosos se desviam de suas especificações arbitrariamente podendo parar, omitir envio ou entrega de mensagens, enviar respostas incorretas, etc. No entanto, assume-se **independência de faltas**, i.e., a probabilidade da ocorrência de uma falta em determinada réplica de um serviço é independente da ocorrência de falta em outra réplica. Esta premissa pode ser obtida na prática através do uso extensivo de diversidade (hardware, sistemas operacionais, máquinas virtuais, bancos de dados, etc) [Castro et al. 2003, Obelheiro et al. 2005].

Os processos (clientes e servidores) se comunicam através de um PEATS[2], i.e., compartilhamento de informações em tuplas armazenadas em um espaço de tuplas que fornece as operações especificadas na seção 2. Este espaço de tuplas é **confiável**, não estando portanto sucetível a falhas. Esta forte premissa pode ser substanciada na prática através do uso de um espaço de tuplas tolerante a faltas bizantinas, como o DEPSPACE [Bessani et al. 2007], utilizado neste trabalho.

Assumimos que os canais de comunicação que ligam os processos ao PEATS, bem como entre os servidores e clientes (quando necessário) são confiáveis e autenticados. Este canais são implementado a partir do protocolo TCP e do uso de MACs (Códigos de Autenticação de Mensagens) juntamente com chaves de sessão, além de ser empregados mecanismos de retransmissão e reconexão quando necessário. Finalmente, admitimos que todos os processos têm relógios locais, e que estes relógios não são sincronizados, tendo como única premissa o progresso.

3.3. Propriedades

Em geral, os sistemas distribuídos são especificados considerando segurança (*safety*) e vivacidade (*liveness*). No REPEATS, algumas propriedades devem ser satisfeitas para que

[1]Apesar do sistema suportar múltiplos serviços replicados partilhando da mesma infra-estrutura de coordenação, para fins de simplificação dos algoritmos, assumimos apenas um único serviço replicado neste artigo.

[2]Se o PEATS for replicado, o modelo de sincronismo necessário para implementá-lo é o de **sincronismo parcial** [Dwork et al. 1988].

o sistema seja vivo e seguro, são elas:

- **Ordem total:** as requisições são executadas com ordem total por todas as réplicas corretas do sistema;
- **Liberdade de bloqueio (*lock-freedom*):** em qualquer execução do sistema, se um processo correto envia uma requisição para execução onde existam requisições pendentes de execução, alguma requisição será executada.

Além das propriedades descritas, um serviço replicado usando o REPEATS apresenta um comportamento equivalente a sua implementação em um sistema não replicado, isto é, satisfaz o modelo de consistência conhecido por **linearização** [Herlihy and Wing 1990].

3.4. Controle de Acesso

Um mecanismo fundamental para que os algoritmos do REPEATS tolerem comportamento malicioso dos processos é o controle de acesso provido pelas políticas de granularidade fina suportadas pelo PEATS [Bessani et al. 2006]. Este controle acontece através da implementação de políticas de segurança associadas ao espaço de tuplas. Quando uma operação é invocada no PEATS, as regras especificadas nestas políticas são verificadas tomando como base o identificador do processo que invoca a operação, a operação que está sendo invocada (e seus argumentos) e o estado atual do espaço de tuplas para negar ou permitir a execução da operação. Um exemplo de regra que pode ser associada a um espaço de tuplas seria "uma operação $out(\langle REQ, p, op, seq\rangle)$ só pode ser executada se o processo invocador é p **e** não existe nenhuma tupla $\langle REQ, q, x, seq\rangle$ no espaço **e** existe uma tupla $\langle REQ, r, y, seq-1\rangle$". Uma característica fundamental do modelo de controle de acesso do PEATS é que as operações não especificadas na política tem sua execução negada por padrão. Neste trabalho apresentamos as políticas de acesso seguindo o mesmo formato usado em [Bessani et al. 2006].

3.5. Algoritmo de Ordenação de Requisições

Nesta seção é apresentado o algoritmo de replicação do REPEATS. Este algoritmo, bem como os outros que suportam esta arquitetura, são executados sobre um PEATS, tendo como alicerce as políticas de controle de acesso que impedem processos maliciosos de afetar a corretude do nosso esquema de replicação.

Conforme já discutido, a difusão com ordem total é um requisito fundamental para replicação Máquina de Estados para que sejam asseguradas as propriedades que concernem ao determinismo de réplicas. Uma vez que todas as comunicações ocorrem através do PEATS, as mensagens depositadas no espaço ficam disponíveis no mesmo instante para todos os processos que têm acesso ao espaço. Deste modo, a ordenação total das requisições é feita a partir de um algoritmo de sequenciamento de mensagens implementado sobre o espaço de tuplas. Neste algoritmo, as requisições são inseridas no espaço juntamente com um número de seqüencia único, em tuplas. Estes números de seqüencia estabelecem a ordem na qual essas requisições devem ser executadas por todas as réplicas de um serviço. A formalização deste protocolo é apresentada no algoritmo 1.

O funcionamento do algoritmo executado pelo cliente (linhas 3-7) é bastante simples. Quando o cliente tem de enviar um comando C, ele começa a tentar colocar uma tupla REQUEST no espaço com um número de seqüencia uma unidade maior que seu último

Algoritmo 1 Algoritmo de Ordenação de Requisições (cliente p_i e servidor s_i).

Variávies compartilhadas:
1: $ts = \emptyset$ {espaço de tuplas}

{Parte Cliente}
Variáveis locais:
2: $reqnum = 0$ {número da ultima requisição conhecida}
procedure $execute(C)$
3: **repeat**
4: $reqnum \leftarrow inspect\_ts(reqnum)$
5: $reqnum \leftarrow reqnum + 1$
6: **until** $ts.cas(\langle REQUEST, reqnum, *, * \rangle, \langle REQUEST, reqnum, p_i, C \rangle)$
7: **return** $wait\_response(reqnum)$

{Parte Servidor}
Variáveis locais:
8: $reqnum = 0$ {número da req. enviada}
tarefa principal
9: **loop**
10: $reqnum \leftarrow reqnum + 1$
11: $ts.rd(\langle REQUEST, reqnum, ?client, ?command \rangle)$
12: $response \leftarrow local\_execute(command)$
13: $send\_response(client, response)$
14: **end loop**

número de seqüencia conhecido (inicialmente 0 – linha 2), através de uma invocação a operação cas, que verifica se existe uma tupla REQUEST com o número de seqüencia em questão no espaço e, em não havendo, insere a tupla pretendida (linha 6). Quando não é bem sucedido nesta inserção, o cliente incrementa o número de seqüencia e tenta novamente (linhas 3-6). Ao ser bem sucedido na inserção da tupla, o cliente é bloqueado na função *wait_response* (linha 7), onde aguardará por $f + 1$ respostas iguais advindas de diferentes servidores. A função *inspect_ts* (linha 4) serve para evitar que um cliente que tenha iniciado sua execução após já haver um número grande de requisições no espaço tenha de iterar muitas vezes pelo laço das linhas 3-6 (cada vez invocando uma operação no espaço de tuplas). Esta função implementa uma estratégia que inspeciona o espaço de tuplas e tenta encontrar um número de seqüencia mais próximo daquele presente na última requisição inserida no espaço. A existência desta função não interfere na correção do algoritmo, mas melhora em muito o desempenho de clientes "atrasados". Uma possível estratégia para implementação dessa função seria verificar qual o *checkpoint* mais novo do sistema e retornar o número da última requisição contida neste *checkpoint* (ver seção 3.6.). Outras possiveis estratégias para a implementação desta função podem ser utilizadas tirando-se proveito de funcionalidades especiais que uma determinada implementação do PEATS pode implementar (ex. suporte a operações que lêem um conjunto de tuplas ou a capacidade de se especificar se o resultado de uma leitura deve pegar a tupla mais nova ou mais velha do espaço que combina com o molde fornecido).

O algoritmo dos servidores (linhas 9-14) também é inicializado com o número de seqüencia da requisição igual 0 (linha 8). As réplicas processam os comandos contidos nas tuplas REQUEST em ordem ascendente primeiro obtendo a tupla com o número de seqüencia uma unidade maior que a última requisição executada (linhas 10-11), executando o comando contido nesta tuplas através da função *local_execute* (linha 12), e enviando a resposta ao cliente, cuja identidade está indicada na tupla REQUEST, através da função *send_response* (linha 13). Por razões de desempenho, esta função envia uma mensagem com a resposta **diretamente ao cliente**, sem utilizar o PEATS.

A política de acesso do PEATS para o algoritmo 1 é descrita na figura 2. A principal finalidade das regras desta política é evitar que processos maliciosos quebrem a ordem total, inserindo tuplas REQUEST fora do intervalo de seqüencia no espaço. Para isso, a inclusão de requisições (tupla REQUEST) só pode ser feita através da instrução cas, na condição de que a tupla REQUEST com número seqüencial anterior ao que está sendo incluído esteja presente no espaço. A operação rd é permitida somente para tuplas REQUEST, desde que os campos 3 e 4 do molde sejam formais.

Object State TS
R_{rd}: $execute(rd(\langle REQUEST, reqnum, x, y\rangle)) \ :-$
$\quad invoke(p, rd(\langle REQUEST, reqnum, x, y\rangle)) \wedge formal(x) \wedge formal(y)$
R_{cas}: $execute(cas(\langle REQUEST, reqnum, x, y\rangle, \langle REQUEST, reqnum, p, inv\rangle)) \ :-$
$\quad invoke(p, cas(\langle REQUEST, reqnum, x, y\rangle, \langle REQUEST, reqnum, p, inv\rangle)) \wedge$
$\quad (reqnum = 1 \vee \exists w, z : \langle REQUEST, reqnum - 1, w, z\rangle) \in TS)$

Figura 2. Política de acesso para o PEATS usado no algoritmo 1.

Para a correção do protocolo algumas premissas são admitidas: (*i*) cada requisição do cliente tem um identificador único e crescente; (*ii*) o cliente só envia uma requisição após ter recebido a resposta da requisição anterior[3]; (*iii*) um temporizador é associado a cada requisição enviada, e caso ocorra um *timeout* e a resposta da requisição ainda não tenha sido obtida, o cliente reenvia a requisição. Maiores detalhes sobre o algoritmo 1, bem como as provas de que ele satisfaz as propriedades definidas na seção 3.3. são omitidas por razões de espaço e podem ser encontradas na versão estendida deste artigo [Luiz et al. 2007].

3.6. Algoritmo de *Checkpoint*

O REPEATS emprega *checkpoints* para permitir a recuperação de réplicas faltosas e também para estabilizar o serviço para fins de coleta de lixo (ver próxima seção), o que é necessário devido a memória finita do espaço de tuplas (note que o algoritmo da seção anterior mantém todas as requisições dos clientes no espaço de tuplas). O algoritmo descrito nesta seção pode ser usado na produção de *checkpoints* em diferentes níveis (completos, incrementais e acumulativos).

A idéia fundamental do algoritmo de *checkpointing* é guardar o estado[4] das réplicas corretas do serviço (que será o mesmo) no espaço de tuplas a cada N requisições executadas, sendo N um parâmetro configurável e igual em todas as réplicas corretas do serviço. No entanto, alguns cuidados devem ser tomados para se evitar que réplicas maliciosas criem *checkpoints* com estados incorretos do serviço. O protocolo executado para tanto é descrito no algoritmo 2.

Quando iniciado, o algoritmo recebe como argumentos o estado a ser armazenado através do *checkpoint* e o número da última requisição executada pelo servidor. Primeiramente, a réplica insere sua proposta de *checkpoint* (uma tupla CKPTPROP), que reflete o seu estado atual (linha 5). A tupla da proposta contém o número do *checkpoint* em questão, o número da última requisição e o resumo criptográfico do estado da réplica (calculado na linha 4). A partir daí os servidores inspecionam o espaço de tuplas até coletarem f propostas deste *checkpoint* feitas por outros servidores (linhas 6-13). Finalmente, o último

[3]Note que esta limitação pode ser relaxada para *k* requisições caso os servidores tenham a capacidade de armazenar as últimas *k* respostas de cada cliente.

[4]O estado pode ser transiente ou persistente, dependendo das características do serviço.

Algoritmo 2 Algoritmo de *Checkpoint* (processo servidor s_i)

{Servidor} **Variávies compartilhadas:**
1: $ts = \emptyset$ {espaço de tuplas}
Variávies locais:
2: $ckptnum = 0$ {número de seqüência do checkpoint}
procedure checkpoint($state, reqnum$)
3: $ckptnum \leftarrow ckptnum + 1$
4: $hash \leftarrow D(state)$ {D é a função que computa o Digest (Hash)}
5: $ts.out(\langle CKPTPROP, p_i, ckptnum, reqnum, hash\rangle)$
6: $RemainingServers \leftarrow S \setminus \{s_i\}$ {Inicialmente $2f$ servidores}
7: **while** $|RemainingServers| > f$ **do**
8: **for all** $s_j \in RemainingServers$ **do**
9: **if** $ts.rdp(\langle CKPTPROP, s_j, ckptnum, reqnum, hash\rangle)$ **then**
10: $RemainingServers \leftarrow RemainingServers \setminus \{s_j\}$
11: **end if**
12: **end for**
13: **end while**
14: $ts.cas(\langle CKPT, ckptnum, reqnum, *, hash\rangle, \langle CKPT, ckptnum, reqnum, state, hash\rangle)$

passo do algoritmo é inserir o *checkpoint* no espaço. Para evitar que todas as réplicas insiram um *checkpoint*, a operação cas é utilizada nesta inserção (linha 14), i.e., um servidor só insere a tupla CKPT se esta não tiver sido inserida previamente.

A política de acesso para o algoritmo 2 é descrita na figura 2. Nesta política descrevem-se em que situações as operações rdp, cas e out têm sua execução permitida. A operação out pode ser usada para depositar no espaço tuplas CKPTPROP que representam as propostas de *checkpoint*, desde que um servidor não tenha já depositado uma proposta de *checkpoint* com esse número. A operação rdp é permitida desde que usada para coletar as tuplas CKPTPROP. Por fim, a consolidação do *checkpoint* (tupla CKPT) é feita somente através da operação *cas*, sendo que para tal deve haver no mínimo $f + 1$ propostas no espaço de tuplas com o mesmo número, número de requisição atual e resumo. Além disso a tupla que corresponde ao *checkpoint* anterior deve estar contida no espaço, caso este *checkpoint* não seja o primeiro. Note que da forma como a regra para a operação cas funciona, é impossível que as réplicas maliciosas (no máximo f) insiram *checkpoints* forjados no espaço (não podem fazer $f + 1$ propostas com o resumo desse *checkpoint*).

Object State TS
R_{out}: $execute(out(\langle CKPTPROP, p, ckptnum, x, y, z\rangle)) \ :-$
$invoke(p, out(\langle CKPTPROP, p, ckptnum, x, y, z\rangle)) \wedge$
$(\nexists i, j, k : \langle CKPTPROP, p, ckptnum, i, j, k\rangle) \in TS)$
R_{rdp}: $execute(rdp(\langle CKPTPROP, p, ckptnum, x, y, z\rangle)) \ :-$
$invoke(p, rdp(\langle CKPTPROP, p, ckptnum, x, y, z\rangle))$
R_{cas}: $execute(cas(\langle CKPT, ckptnum, rn, *, h\rangle, \langle CKPT, ckptnum, rn, state, h\rangle)) \ :-$
$invoke(p, cas(\langle CKPT, ckptnum, rn, *, h\rangle, \langle CKPT, ckptnum, rn, state, h\rangle)) \wedge$
$\exists X \subset S : (|X| \geq f + 1 \wedge \forall s \in X : \langle CKPTPROP, s, ckptnum, rn, h\rangle) \wedge$
$(ckptnum = 1 \vee \exists x, y, z : \langle CKPT, ckptnum - 1, x, y, z\rangle) \in TS)$

Figura 3. Política de acesso para o PEATS usado no algoritmo 2.

3.7. Coleta de Lixo

Para evitar a saturação do espaço de tuplas (com muitas tuplas REQUEST, por exemplo) utilizamos um mecanismo para coleta de lixo. Este mecanismo é responsável por limpar as requisições e *checkpoints* não mais usados pelo sistema garantindo que as tuplas candidatas à exclusão não estão sendo utilizadas por servidores corretos.

A coleta de lixo no REPEATS é feita logo após a criação de um *checkpoint*. Todos

os servidores tentam remover tuplas REQUEST, CKPTPROP e CKPT (criadas nos algoritmos 1 e 2) que tenham se tornadas obsoletas devido ao último *checkpoint* criado. Por exemplo, quando se cria um *checkpoint* de número k e com o número de última requisição executada r, potencialmente é possível apagar todas tuplas REQUEST com número de seqüencia menor que r e todas as tuplas CKPT e CKPTPROP com número menor que k. No entanto, é preciso ter cuidado para não apagar requisições que ainda não foram processadas por réplicas "atrasadas", pois neste caso, em havendo pelo menos uma réplica maliciosa no sistema, é possível que os clientes que enviaram as requisições apagadas fiquem bloqueados indefinidamente a espera de $f+1$ respostas. A resolução deste problema requer pequenas modificações no protocolo de replicação (algoritmo 1), as quais podem ser resumidas da seguinte forma: inclui-se no estado a ser armazenado no *checkpoint* uma tabela contendo um resumo criptográfico da última requisição processadas de cada cliente e suas respostas (ou k requisições conforme descrito na seção 3.5), desta forma todo servidor (inclusive os atrasados e recuperados, que estabeleceram seu estado a partir de um *checkpoint* no sistema) sabe o que responder a um cliente em caso de requisições repetidas. O cliente, por sua vez, deve fazer com que cada uma de suas requisições seja única (ex., incluindo um número de seqüencia no comando) e, se uma requisição colocada no espaço demora muito a ser respondida, ela é reenviada (através do algoritmo de replicação).

3.8. Mecanismo de Logging e Recuperação de Réplicas

Como as requisições são armazenadas no espaço de tuplas (PEATS), exploramos esta facilidade para fins de definição de um mecanismo de *logging*. Este mecanismo é necessário para a recuperação das réplicas que possam falhar. As requisições enviadas perduram no espaço até o momento da gravação de um *checkpoint* posterior, que sinaliza que as requisições anteriores a ele não são mais necessárias durante o processo de recuperação de réplicas. Deste modo, para a recuperação pontual de uma réplica, o processo restaura os *checkpoints* necessários e se houverem requisições após o último *checkpoint*, estas são recuperadas diretamente do espaço[5].

4. Implementação do REPEATS

Os algoritmos e mecanismos descritos nas seções anteriores foram implementados usando como base o espaço de tuplas tolerante a faltas bizantinas DEPSPACE[6] [Bessani et al. 2007]. Todos os componentes foram implementados na linguagem Java, usando o JDK 1.6.0_3 da SUN.

O REPEATS consiste basicamente em uma biblioteca com algumas classes, implementando uma camada de abstração (*stub*) entre as entidades (clientes e servidores) e o espaço de tuplas. Processos clientes e servidores acessam estas classes, que implementam os algoritmos descritos na seção anterior, que por sua vez, acessam o DEPSPACE. Este acesso é feito através de canais confiáveis e autenticados, suportados por *sockets* TCP e MACs. Este mesmo tipo de canal é usado para envio de resposta dos servidores aos clientes.

Em relação a quantidade de máquinas utilizadas em nosso protótipo, são necessárias $n_s = 2f_s + 1$ réplicas do serviço e $n_{ts} = 3f_{ts} + 1$ réplicas do espaço de tu-

[5]Caso o estado transiente/persistente seja muito grande inviabilizando seu armazanamento no PEATS, além do *checkpoint* uma rotina auxiliar de recuperação de estado é usada.

[6]Disponível em `http://www.navigators.di.fc.ul.pt/software/depspace/`.

plas (um requisito do modelo de replicação adotado pelo DEPSPACE [Bessani et al. 2007]), tolerando assim f_s réplicas faltosas do serviço e f_{ts} réplicas faltosas do espaço de tuplas. No entanto, há de se considerar que o serviço fornecido pelas mesmas n_{ts} pode ser utilizado concorrentemente por diferentes conjuntos de serviços replicados (diferentes aplicações), o que reduz consideravelmente o custo em se replicar serviços distribuídos.

5. Estudo de Caso: Sistema de Arquivos NFS Tolerante a Faltas Bizantinas

5.1. Aspectos Gerais de Implementação

A fim de avaliar o REPEATS, implementamos um serviço NFS replicado, nos mesmos moldes de trabalhos anteriores como [Castro and Liskov 2002, Yin et al. 2003]. Nossa implementação foi inspirada nas especificações do WebNFS [Callaghan 1996a, Callaghan 1996b], que consiste em uma evolução do NFS onde é possível obter o acesso aos objetos NFS para aplicações sem a necessidade de realizar a montagem do sistema de arquivos remoto no VFS (*Virtual File System*) do sistema operacional do cliente. Todo o acesso a arquivos, diretórios e ligações é feito através de uma biblioteca de classes (feita em Java) carregada pela aplicação cliente. O servidor NFS usado foi o "jnfs"[7], que é uma implementação aberta do NFS versão 2 escrita em Java, sendo necessária a modificação do núcleo do "jnfs" para prover algumas funcionalidades requeridas para o WebNFS. Este servidor foi modificado para receber requisições através do REPEATS, i.e., ao invés de esperar requisições RPC/XDR, ele usa a biblioteca do REPEATS para ler tuplas de requisições do espaço de tuplas.

5.2. Avaliação de Desempenho

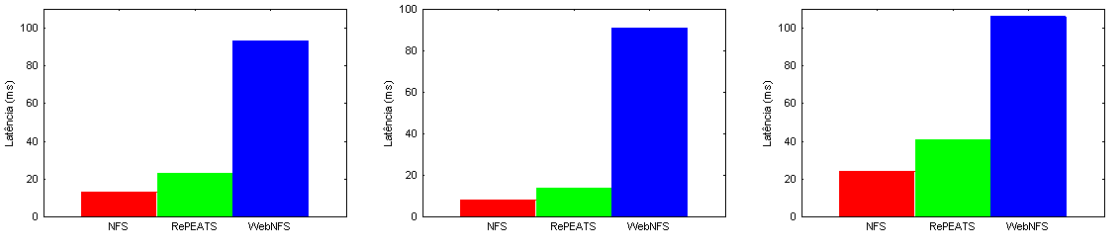
Esta seção apresenta alguns resultados experimentais comparando o NFS tolerante a faltas bizantinas construído com base no REPEATS com um serviço NFS não replicado e com um cliente WebNFS[8]. A métrica utilizada para medir o desempenho foi o tempo total de execução de várias operações comuns sobre arquivos.

O ambiente de testes é constituído por quatro máquinas para o DEPSPACE (constituindo um PEATS tolerante a um servidor faltoso), todas com processador Pentium IV 2.8 Dual, 2 GB de memória e interface de rede ethernet Gigabit 1000Mb/s, tendo instalado o Suse Linux 10 EL. As três máquinas usadas para o serviço NFS replicado (tolerando um servidor falho) têm as seguintes configurações: *(i)* uma máquina com 2GB de memória, dois processadores Intel Xeon 3.2, interfaces ethernet Gigabit e Linux RedHat AS 4.0; *(ii)* uma máquina com 2GB de memória, processador Pentium IV Dual, interfaces Ethernet Gigabit e Suse Linux 9 EL; *(iii)* uma máquina com 2GB de memória, dois processadores Pentium III Xeon 2.0, interfaces Ethernet Gigabit e Suse Linux 10 EL. A máquina *(i)* é usada como único servidor NFS nos experimentos que não envolvem o REPEATS. Todas as máquinas foram conectadas através de um switch Gigabit 3Com SuperStack3. Para os clientes foram usadas máquinas com configurações homogêneas (processador Intel Pentium VI 2.8 HT, 512MB de RAM, interface Ethernet 100Mb/s).

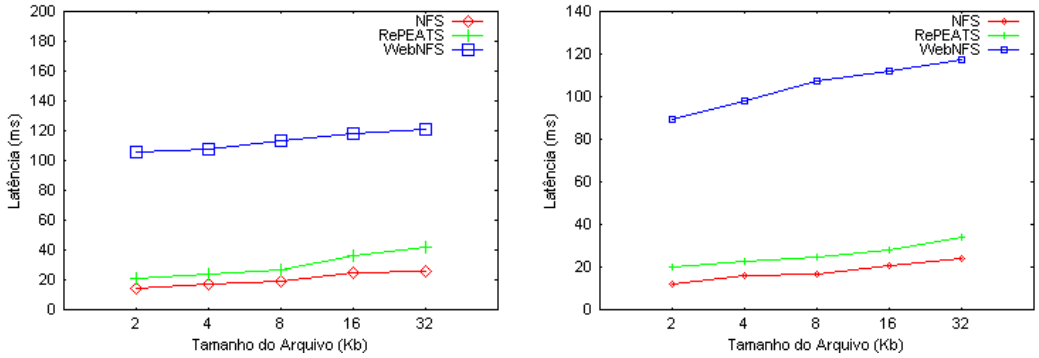
Nos experimentos foram analisadas as seguintes operações sobre o NFS: (1) criação de arquivos e diretórios (figura 4(a)), (2) exclusão de arquivos e diretórios (figura 4(b)), (3) listagem do conteúdo de diretórios (figura 4(c)), (4) escrita de 2k, 4k, 8k,

[7]Disponível em `http://www.void.org/~steven/jnfs/`.
[8]Disponível em `https://yanfs.dev.java.net/`.

16k e 32k em arquivos remotos (figura 4(d)), e (5) leitura dos dados de arquivos com 2k, 4k, 8k, 16k e 32k (figura 4(e)). Para cada um destes experimentos foram realizadas 1000 execuções da operação em questão, onde a latência reportada compreende o tempo médio necessário para a execução da operação, excluindo-se 1% dos valores com maior desvio, conforme percebida pelo cliente. Assim como em trabalhos similares [Castro and Liskov 2002, Yin et al. 2003], nossos experimentos consideram condições mais freqüentes um sistema de arquivo distribuído: ausência de concorrência e réplicas faltosas.



(a) Criação de arquivos. (b) Exclusão de arquivos. (c) Listagem de diretórios.



(d) Escrita de arquivo. (e) Leitura de arquivo.

Figura 4. Latência de diversas operações sobre arquivos nos três serviços NFS comparados: REPEATS NFS, NFS nativo (API *java.io* + cliente NFS Linux), WebNFS. O servidor NFS usado em todos os experimentos é o jnfs.

A partir dos gráficos da figura 4 podemos observar a viabilidade da proposta, onde observa-se que o custo adicional do uso do REPEATS fica entre 30 a 60% em relação ao acesso a um serviço não tolerante a faltas (NFS). Este custo adicional se deve basicamente a latência extra em se acessar o DEPSPACE, que é por volta de 4 ms em operações de escrita e 1 ms em operações de leitura. Em nossa opinião este custo é moderado se considerarmos os benefícios em termos de confiabilidade e segurança que nossa solução replicada oferece. Também é digno de nota o fato da latência apresentar um crescimento leve, quando o tamanho da resposta do serviço é aumentado. Isto ocorre pelo fato das respostas serem enviadas diretamente aos clientes e não passarem pelo DEPSPACE. Finalmente, nosso experimentos mostraram que a implementação WebNFS (não tolerantes a faltas) tem um desempenho muito inferior ao NFS baseado no REPEATS. Isto se explica pelo fato de que, nesta implementação, a cada operação invocada no WebNFS, todos os passos requeridos no servidor NFS são re-executados (i.e., montagem, verificação de permissões, acesso, etc).

6. Trabalhos Relacionados

Tomando a literatura recente sobre replicação tolerante a faltas, existem pelo menos dois trabalhos que estão intimamente relacionados aos nosso esforços com o REPEATS: *(i)* o

CL-BFT/BASE [Castro and Liskov 2002, Castro et al. 2003], que foi o primeiro trabalho a propor um esquema completo para replicação tolerante a faltas bizantinas com viabilidade prática; e (*ii*) o modelo arquitetural introduzido em [Yin et al. 2003], que propôs a "separação" das atividades de um serviço replicado em duas camadas (acordo e execução), além de agregar confidencialidade (através de uma terceira camada entreposta às camadas de acordo e execução, que funciona como uma espécie de *firewall* replicado) no esquema de RME. Do ponto de vista arquitetural, o REPEATS, prove uma separação entre as atividades de estabelecimento de ordem total nas requisições dos clientes e execução do serviço em camadas diferentes, da mesma forma que [Yin et al. 2003]. No entanto, o REPEATS usa como "camada de acordo" o PEATS, o que também possibilita a redução do número de réplicas de um serviço para $2f + 1$. Vale citar que em nossa proposta, o PEATS, além de ser usado para definir a ordem total das requisições dos clientes, provê algum nível de confidencialidade nas comunicações (através do controle de acesso provido pelas políticas), e, sendo uma abstração muito mais poderosa que uma simples "camada de acordo", pode ser usado para tarefas tão distintas como definição de *checkpointing*, armazenamento de *logging* confiável e resolução de não-determinismos. Um outra vantagem fundamental do REPEATS em relação aos trabalhos anteriores é que o fato de se basear em uma abstração muito mais poderosa que primitivas de difusão com ordem total, permite que seus algoritmos de replicação sejam muito mais simples.

Convém citar que recentemente tem havido um grande esforço por parte da comunidade, no que concerne a investigação de modelos de coordenação/comunicação alternativos para sistemas distribuídos [Cabri et al. 2000]. O PEATS [Bessani et al. 2006] é um exemplo deste esforço, introduzindo uma abstração de alto nível que permite a construção de algoritmos tolerantes a faltas bizantinas simples e eficientes, tendo como base o modelo de coordenação por espaço de tuplas. O presente trabalho pode ser visto como uma primeira exploração desta abstração com finalidades mais práticas, neste caso, implementar replicação máquina de estados.

7. Conclusões

Este artigo apresentou uma arquitetura para replicação tolerante a faltas bizantinas baseado no modelo de coordenação por espaço de tuplas. A proposta foi testada sobre uma aplicação real e medidas de desempenho mostraram que o custo para de acesso a um NFS tolerante a faltas bizantinas baseado no REPEATS é entre 30-60% maior em termos de latência que o custo de acesso a um NFS não replicado, o que é bastante satisfatório dado o elevado nível de qualidade de serviço oferecido por nossa arquitetura.

Os trabalhos atuais se concentram no refinamento dos mecanismos de coleta de lixo e otimizações para clientes "atrasados" e na implementação de outros modelos de replicação tolerante a faltas bizantinas [Castro and Liskov 2002, Yin et al. 2003] para melhor mensurar os benefícios e custos associados a proposta apresentada neste artigo. Esses resultados serão reportados em trabalhos futuros.

Referências

Bessani, A. N., Alchieri, E., Correia, M. P., da Silva Fraga, J., and Lung, L. C. (2007). DEPSPACE: Um middleware para coordenação em ambientes dinâmicos e não confiáveis. In *Salão de Ferramentas do XXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. SBC.

Bessani, A. N., Correia, M., da Silva Fraga, J., and Lung, L. C. (2006). Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

Cabri, G., Leonardi, L., and Zambonelli, F. (2000). Mobile-agent coordination models for Internet applications. *Computer*, 33(2):82–89.

Callaghan, B. (1996a). WebNFS Client Specification (RFC 2054). IETF Request For Comments.

Callaghan, B. (1996b). WebNFS Server Specification (RFC 2055). IETF Request For Comments.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461.

Castro, M., Rodrigues, R., and Liskov, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269.

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

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of ACM*, 35(2):96–107.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

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

Luiz, A. F., Bessani, A. N., Lung, L. C., and Filgueiras, T. (2007). REPEATS - uma arquitetura para replicação tolerante a faltas bizantinas baseada em espaço de tuplas. Disponível em `http://www.di.fc.ul.pt/~bessani/repeats.pdf`.

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*.

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

Segall, E. J. (1995). Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, pages 320–327.

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, pages 253–267.