Explorando a Abstração Espaço de Tuplas no Escalonamento em Grades Computacionais*

Fábio Favarim[1]†, Joni Fraga[1]†, Lau C.Lung[2]†, Miguel Correia[3], João F.Santos[1]

[1]DAS, Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

[2]PPGIA, Pontifícia Universidade Católica do Paraná – Curitiba – PR – Brasil

[3]LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal

{fabio,fraga,jfsantos}@das.ufsc.br, lau@ppgia.pucpr.br, mpc@di.fc.ul.pt

Abstract. *One of the problems in grid environments is how to effectively use the resources. Generally, the scheduling in this environment needs to make use of some information about resources. In this paper we propose a grid infrastructure (GridTS) in which the resources select the tasks they execute, instead of a traditional schedulers, where they find resources for the tasks. Moreover, GridTS provides fault-tolerant scheduling by combining a set of fault tolerance techniques. The core of the solution is a tuple space, which supports the communication e provides support for the fault tolerance mechanisms.*

Resumo. *Um dos problemas em ambientes de grades é como fazer o uso efetivo dos recursos. Geralmente, o escalonamento neste ambiente necessita de algumas informações sobre os recursos, porém é difícil de se obter. Neste artigo propomos uma infra-estrutura de grade (GridTS) onde os recursos selecionam as tarefas que eles executam, ao contrário dos escalonadores tradicionais, onde buscam por recursos para executar as tarefas. Além disso, o GridTS provê um escalonamento tolerante a faltas através da combinação de um conjunto de técnicas de tolerância a faltas. O núcleo desta solução é o espaço de tuplas, o qual provê o suporte à comunicação e aos mecanismos de tolerância a faltas.*

1. Introdução

Um objetivo importante de uma infra-estrutura de grade computacional é fazer o uso eficiente dos recursos, isto é, maximizar a utilização dos recursos enquanto tenta minimizar o tempo total de execução de uma aplicação (também chamado de *makespan*) [Fujimoto and Hagihara 2003]. O desempenho da grade depende fortemente da eficiência do escalonamento. Um escalonamento é uma associação das tarefas de uma aplicação a um conjunto de recursos. Cada aplicação consiste de um conjunto de tarefas, que são atribuídas a um conjunto de recursos da grade para suas execuções. Muitos escalonadores necessitam de informações sobre recursos e tarefas para efetuar o escalonamento. Geralmente estas informações são fornecidas por um serviço de informação.

O processo de agregar informações sobre recursos de uma grade é conhecido como obter o *snapshot* do mesmo. Isto é, corresponde a obter um estado de ocupação da grade o mais próximo possível da realidade da mesma em um dado instante. Esta operação é custosa e a "fotografia" dificilmente é completamente atual devido à complexidade destes sistemas formados por grandes quantidades de recursos heterogêneos, não dedicados e amplamente dispersos na grade. Sabemos da teoria de sistemas distribuídos que problemas

*Trabalho apoiado pelo CNPq (processos 550114/2005-0 e 506639/04-5).
†Bolsista CNPq.

de obtenção de *snapshots* precisos (estados globais) em sistemas distribuídos assíncronos (a Internet é um exemplo destes) não possuem solução [Chandy and Lamport 1985]. Portanto, o escalonamento baseado nestas informações de ocupação obtidas através de um serviço de informação corre um forte risco de definir atribuições que não se efetivam devido a desatualização das informações usadas pelo escalonador.

Este artigo propõe o ***GridTS***, uma infra-estrutura que provê uma nova solução para o escalonamento, onde os recursos selecionam as tarefas mais apropriadas para suas condições de execução. Ou seja, é invertida a ordem tradicional onde os escalonadores é que buscam os recursos para as tarefas disponíveis. A solução proposta não faz uso de um serviço de informação e mesmo assim, permite decisões do escalonamento serem feitas com informações atualizadas. Os recursos conhecem suas limitações e devem procurar tarefas mais adequadas às suas características. Portanto, acreditamos que a nossa solução supera os problemas da obtenção de informações atualizadas, normalmente apontados pelos escalonadores que necessitam de tais informações.

O *GridTS* é baseado no modelo de coordenação generativa, onde os processos (*brokers* e recursos) interagem através de um objeto de memória compartilhada, chamado de **Espaço de Tuplas** (TS) [Gelernter 1985]. Este modelo de coordenação provê suporte para comunicações desacopladas tanto no tempo quanto no espaço, ou seja, processos comunicantes não precisam estar ativos ao mesmo tempo e não precisam saber a localização ou endereço um dos outros [Cabri et al. 2000]. Isto faz deste modelo de coordenação particularmente adequado para sistemas altamente dinâmicos como a grade.

Em sistemas de larga escala, como uma grade, a probabilidade de falhas[1] de componentes acontecer é alta. Muitas das infra-estruturas de grades atuais possuem pontos únicos de falha, ou seja, nem todos seus componentes são tolerantes a falhas. O *GridTS* suporta estas falhas parciais no sentido que todos seus componentes podem falhar por parada (*crash*) e o sistema continua se comportando como esperado.

Isto é garantido através de combinação de técnicas de tolerância a faltas. A disponibilidade do espaço de tuplas é garantida pela replicação do mesmo. A consistência neste espaço, diante da concorrência e de possíveis falhas dos processos comunicantes (*brokers* e recursos), é mantida através de um mecanismo de transações. E, finalmente, um mecanismo de *checkpoint* é usado para limitar a quantidade de processamento perdido na falha de um recurso durante a execução de tarefas longas.

Em resumo, este trabalho apresenta duas contribuições principais: apresenta uma infra-estrutura para a computação em grade que permite recursos buscarem por tarefas adequadas a suas características, mesmo que estas características mudem com o tempo; a infra-estrutura provê um escalonamento tolerante a faltas através da combinação de um conjunto de técnicas técnicas de tolerância a faltas. Através de simulações mostramos que o *GridTS* e suas contribuições são viáveis na computação em grade.

---

[1] Os termos usados neste texto para as ditas imperfeições em sistemas informáticos são:

- Falta: causa de possíveis interrupções de um serviço.
- Erro: manifestação interna provocada pela ativação de uma falta.
- Falha : manifestação externa da ocorrência de uma falta resultando na "parada" (crash) de fornecimento do serviço correto.

Diante dos termos usados, a disciplina que garante por meio de redundância o serviço mesmo em presença de faltas toma o nome de "tolerância a faltas" neste texto.

2. Espaço de Tuplas

O modelo de coordenação generativa, introduzido no contexto da linguagem de programação LINDA, onde os processos distribuídos interagem através de um espaço de memória compartilhado, chamado de **espaço de tuplas**, que pode ser implementando em uma rede usando um ou mais servidores [Gelernter 1985]. Neste espaço, estruturas de dados genéricas, chamadas **tuplas**, podem ser inseridas, lidas e removidas.

Uma tupla é uma seqüência de campos tipados. Dado uma tupla $t = \langle f_1, f_2, ..., f_n \rangle$, cada campo f_i pode ser: **real**, i.e., ser um valor; **formal**, i.e., um nome de variável precedido por um sinal de "?"; ou um **símbolo especial**, como "*" representando qualquer valor. Uma tupla onde todos os campos são reais é chamada de **entrada** (*entry*) e representada por t. Uma tupla com pelo menos um campo formal ou "*" é chamada de **molde** (*template*) e é representada por $\bar{t}$. Um espaço de tuplas somente pode armazenar entradas, nunca moldes. Os moldes são usados para ler as tuplas dos espaço.

Uma importante característica do modelo de coordenação generativa é a natureza associativa da comunicação: as tuplas não são acessadas por um endereço ou identificador, mas através de seu conteúdo. Uma entrada t e um molde $\bar{t}$ combinam se: (i) ambos têm o mesmo número de campos, (ii) os campos correspondentes tem o mesmo tipo, e (iii) os campos reais correspondentes tem o mesmo valor.

Um espaço de tuplas provê três operações básicas [Gelernter 1985]: $out(t)$ que insere a tupla (entrada) t no espaço de tuplas (escrita); $in(\bar{t})$ que lê e remove, do espaço de tuplas, uma tupla t que combine com $\bar{t}$ (leitura destrutiva). Se nenhuma tupla que combine com $\bar{t}$ está disponível, o processo fica bloqueado até estas esteja disponível; $rd(\bar{t})$ tem um comportamento similar a operação in, mas somente faz a leitura da tupla que combina com $\bar{t}$, sem removê-la do espaço (leitura não destrutiva).

Uma extensão típica deste modelo, é a provisão de variantes não bloqueantes das operações de leitura: inp e rdp. Estas operações funcionam da mesma forma que suas versões bloqueantes, mas retorna um valor "tupla não encontrada"se nenhuma tupla que combine com o molde está disponível no espaço de tuplas. Todas as operações de leitura são não-deterministas, pois se existe mais de uma tupla disponível combinando com o molde, uma delas é escolhida aleatoriamente. Neste artigo, também fazemos o uso da operação $copy_collect(\bar{t})$, que é uma variação da operação rd que faz a leitura de todas a tuplas que combinam com o molde $\bar{t}$ [Rowstron and Wood 1998].

3. Visão Geral do *GridTS*

Em nossa proposta, o *GridTS*, usamos o espaço de tuplas para dar suporte ao escalonamento de tarefas na grade. Resumidamente, a idéia é a seguinte: tuplas descrevendo as tarefas que compõem a aplicação do usuário são colocadas, através do *broker*, no espaço de tuplas. Os recursos computacionais da grade recuperam do espaço as tuplas que descrevem tarefas que eles são capazes de executar. Um recurso, ao término de uma tarefa, deve colocar no espaço de tuplas o resultado deste processamento. Este resultado fica disponível, através do *broker*, para o usuário que submeteu a aplicação a grade.

A descrição da tarefa contém todas as informações necessárias para sua execução, como: a identificação da tarefa, os requisitos para sua execução (ex: carga e velocidade do processador, memória disponível), o código e os parâmetros (dados de entrada) para a execução da mesma. Os usuários não precisam saber quais recursos irão executar as tarefas, suas localizações ou quando estes recursos estarão disponíveis. Portanto, o escalonamento é descentralizado e não existe a necessidade de um serviço de informação.

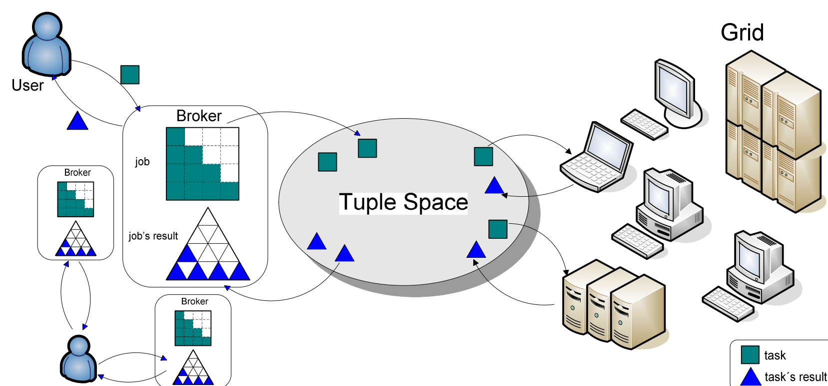


Figura 1. A infra-estrutura *GridTS*

A Figura 1 apresenta a infra-estrutura proposta do *GridTS*. O escalonamento de tarefas é baseado no padrão de projeto *replicated-worker*, também conhecido como padrão *master-workers* [Carriero and Gelernter 1989]. Este padrão possui dois tipos de entidades: um mestre e diversos escravos. O mestre submete as tarefas aos escravos que as executam e retornam os correspondentes resultado ao mestre.

No *GridTS* não existe somente um, mas diversos mestres – chamados de ***brokers*** – que pegam as aplicações de seus **usuários**, divide estas aplicações em **tarefas** tornando-as disponíveis aos **recursos** da grade. Os *brokers* são geralmente específicos para um tipo de aplicação, ou seja, eles sabem como decompor em tarefas uma determinada aplicação. Por exemplo, se a aplicação trata do processamento de imagens de satélite, o *broker* decompõe a imagem (aplicação) em diversas parte menores (tarefas), que podem ser analisadas por diferentes recursos individualmente. As comunicações entre *brokers* e recursos são feitas feitas exclusivamente através do espaço de tuplas.

A arquitetura do *GridTS* tem o benefício imediato de não exigir um serviço de informação para indicar a ocupação de um recurso. Ao contrário, a mesma garante uma forma natural de balanceamento de carga desde que os recursos pegam tarefas adequadas a suas condições momentâneas. Um recurso só busca uma nova tarefa quando tiver terminado de executar a anterior. No entanto, existem alguns desafios. O primeiro, diante da concorrência de diversos *brokers* colocando tarefas de diferentes aplicações no espaço de tuplas, é a garantia de *fairness* na execução destas aplicações e suas tarefas. E o segundo destes desafios está relacionado à tolerância a faltas. O *GridTS* tem que suportar falhas de *brokers* mantendo a consistência das tuplas no espaço de tuplas e, principalmente, tratar eficientemente as falhas de recursos.

4. Principais Aspectos do *GridTS*

Esta seção apresenta aspectos referentes a infra-estrutura *GridTS*. Iniciamos através da apresentação do modelo de sistema e das propriedades que o *GridTS* deve satisfazer.

4.1. Modelo do Sistema

O espaço de tuplas (TS) é definido para o acesso um número indeterminado de processos clientes. Neste modelo, um conjunto de n servidores $U = \{s_1, s_2, ..., s_n\}$ implementa o espaço de tuplas tolerante a faltas. A tolerância a faltas é garantida através da replicação do espaço de tuplas em todos os servidores [Xu and Liskov 1989, Bakken and Schlichting 1995]. Assumimos que um número arbitrário de clientes e um número f de servidores podem falhar e a correção dos serviços da grade é ainda mantida.

No sistema, cada processo (cliente) que usa o espaço de tuplas, comporta-se de acordo com sua especificação até que falhe ou pare de participar da grade por alguma

razão. As falhas de parada (*crash failures*), consideradas neste contexto, podem ser acidentais (recurso deixa de funcionar) ou forçadas por proprietários que desejam remover da grade seus recursos. Não consideramos a possibilidade dos recursos devolverem resultados que não correspondam à execução correta de tarefas. Um processo que nunca falha é dito ser **correto**, enquanto um processo que falhe é dito ser **faltoso**.

Os processos clientes são divididos em dois subconjuntos: o conjunto $R = \{r_1, r_2, r_3...\}$ de **recursos** e o conjunto $B = \{b_1, b_2, b_3, ...\}$ de **brokers**. Os processos clientes podem somente se comunicarem através do espaço de tuplas (TS).

Assumimos o modelo de computação ***bag-of-tasks*** [Smith and Shrivastava 1996] onde tarefas são executadas de forma independente, ou seja, não há nenhuma comunicação ou sincronização entre os recursos que as processam[2]. A grade pode ser composta por recursos heterogêneos, em diferentes domínios administrativos, e/ou recursos voluntários dispersos sobre a Internet.

Embora o sistema apresentado se enquadre mais com modelos de assíncronos de processamento distribuído[3], consideramos a existência de um serviço de transação que não pode ser implementado em um sistema estritamente assíncrono. Ou seja, teremos que ter premissas adicionais no sistema considerando o tempo.

É importante deixar claro que o término anormal de um recurso não é a única razão para uma tarefa parar de ser executada. Se um recurso se torna indisponível por qualquer razão, por exemplo, seu proprietário precisa utilizá-lo, então também consideramos como uma falha de parada do recurso (*crash* do recurso).

4.2. Propriedades do *GridTS*

Sistemas distribuídos são especificados levando em consideração as noções de *safety* e *liveness* [Alpern and Schneider 1984]. Informalmente, *safety* indica que "alguma coisa ruim não irá acontecer" durante a execução do sistema, enquanto *liveness* indica que "alguma coisa boa deve acabar por acontecer". Neste sentido, na nossa proposta, uma **tarefa pronta para execução** deve ser associada a uma tupla que a descreve no espaço de tuplas. Duas propriedades precisam ser satisfeitas pelo *GridTS*:

- *Partial correctness*: se um recurso que está executando uma tarefa falhar, então a tarefa torna-se novamente pronta para ser executada.
- *Starvation freedom*: se existe alguma tarefa para ser executada e um recuso correto capaz de executá-la, então esta tarefa acabará por ser executada.

A primeira propriedade (ligada a *safety*) diz que uma tarefa não desaparece do espaço TS mesmo que o recurso que a esteja executando venha a falhar. A segunda propriedade (*liveness*) diz que toda tarefa será executada se existe pelo menos um recurso correto capaz de executá-la, ou seja, nenhuma tarefa ficará esperando eternamente sua execução. Estas são as principais propriedades que o *GridTS* tem que garantir para que todas as tarefas sejam executadas na grade.

4.3. Base Algorítmica do *GridTS*

Nesta seção definimos o comportamento dos *brokers* e recursos através dos algoritmos que os mesmos executam. A Tabela 1 descreve a estrutura definida para as tuplas usadas

---

[2]Muitas aplicações importantes são classificadas nesta categoria: *data mining*, simulações de Monte Carlo, buscas massivas e manipulação de imagem.

[3]Sistemas Assíncronos [Fischer et al. 1985] são aqueles em que não são verificados limites em seus atrasos de comunicação ou de processamento

nos algoritmos. Em todas as tuplas, o primeiro campo corresponde ao identificador da mesma. A maioria das tuplas contém entre os seus campos $jobId$ e $taskId$, os quais identificam a aplicação e a tarefa, respectivamente:

Tabela 1. Estrutura definida para as tuplas do GridTS

$\langle$"*TICKET*", *ticket*$\rangle$ - define um seqüenciador *ticket* usado para garantir a ordem na execução das tarefas. O objetivo é garantir o escalonamento justo, ou seja, garantir *starvation freedom*. O campo *ticket* contém o valor ainda não alocado do contador *ticket*. O espaço de tuplas é inicializado com uma tupla $\langle$"*TICKET*", *0*$\rangle$.
$\langle$"*JOB*", *jobId*, *numberTasks*, *ticket*, *information*, *code*$\rangle$ - representa informações comuns à todas tarefas da mesma aplicação. O campo *numberTasks* contém o número de tarefas que compõem a aplicação, *ticket* é o valor do *ticket* associado à aplicação, e *information* indica os atributos para a execução da aplicação (ex., a velocidade do processador, memória, sistema operacional). O campo *code* pode conter ou o código a ser executado ou a sua localização (ex.: uma URL).
$\langle$"*TASK*", *jobId*, *taskId*, *information*, *parameters*$\rangle$ - corresponde à tarefa a ser executada. O campo *information* tem atributos para a execução da tarefa, **parameters** contém os dados de entrada para a execução da tarefa ou a sua localização. Uma tarefa é identificada através dos campos *jobId* e *taskId*.
$\langle$"*RESULT*", *jobId*, *taskId*, *result*$\rangle$ - descreve o resultado da execução da tarefa. O campo *result* contém o resultado ou a referência para sua localização.
$\langle$"*CHECKPOINT*", *jobId*, *taskId*, *checkpoint*$\rangle$ - representa o estado da tarefa após uma execução parcial, ou seja, um *checkpoint*. Se um recurso falhar durante a execução de um tarefa, este *checkpoint* é usado por outro recurso para continuar a execução da tarefa. O campo *checkpoint* contém o estado parcial da computação ou uma referência.
$\langle$"*TRANS*", *transId*, *ticket*, *jobId*$\rangle$ - indica a última transação executada pelo *broker*, visto que os *brokers* executam uma seqüência de duas transações. O objetivo é evitar que *broker* reexecute a mesma transação já confirmada quando for reiniciado após uma falha do mesmo. O campo *transId* identifica a última transação que foi confirmada com sucesso. Os campos *ticket* e *jobId* são usados para especificar o que o *broker* estava fazendo quando falhou.

Podem haver pequenas variações das tuplas de aplicação e de tarefas. As tuplas de aplicação e de tarefas descritas acima foram projetadas para aplicações cujas tarefas executam o mesmo código e somente os dados de entrada são diferentes para cada tarefa. Modificações triviais são necessárias, por exemplo, para aplicações cujas tarefas executam códigos diferentes, mas tem os mesmos dados de entrada, ou aplicações com código e dados de entrada diferentes para todas as tarefas.

4.3.1. Broker

O algoritmo executado pelos *brokers* está apresentado no Algoritmo 1. O mesmo é baseado em transações atômicas jeong:1994 que são construídas no sentido de fazer o espaço de tuplas se manter consistente mesmo na presença de falhas de *brokers*. A primeira transação é usada basicamente para garantir que as tarefas da aplicação são inseridas atomicamente no espaço, ou seja, ou todas são inseridas ou nenhuma quando o *broker* falhar durante a inserção. A segunda transação é usada para pegar, atomicamente, os resultados das tarefas do espaço de tuplas. Estas transações permitem também que o *broker* deixe o sistema após ter inserido as tarefas no espaço e voltar posteriormente para pegar os resultados, assim não fica bloqueado enquanto todas as tarefas são executadas.

O algoritmo começa verificando se o *broker* foi reiniciado devido a uma falha. Isto é feito através da operação $rdp()$ (linha 2). Se esta operação não retornar uma tupla, então $transId = 1$ (linha 1), a aplicação a aplicação é dividida em tarefas (linha 4) e a primeira transação é executada (linhas 5-14). Caso contrário, no retorno de uma tupla na

Algoritmo 1 Broker b_i

procedure broker($jobId, information, parameters, code$)

```
1:  transId = 1;
2:  rdp("TRANS", ?transId, ?ticket, jobId)
3:  if (transId = 1) then
4:      tasks ← generateTasks(parameters);
5:      begin transaction                      // transação 1 – insere as tarefas no espaço de tuplas
6:          in("TICKET", ?ticket);
7:          out("TICKET", ++ticket);
8:          out("JOB", jobId, numberTasks, ticket, information, code);
9:          for i ← 1 to numberTasks do
10:             out("TASK", jobId, tasks[i].id, task[i].information, tasks[i].parameters);
11:         end for
12:         transId = 2;
13:         out("TRANS", transId, ticket, jobId);
14:     commit transaction
15: end if
16: if (transId = 2) then
17:     begin transaction                      // transação 2 – pega os resultados do espaço de tuplas
18:         for taskId ← 1 to numberTasks do
19:             inp("RESULT", b_i, jobId, taskId, ?r);
20:             result ← result ∪ {r};
21:         end for
22:         in("TRANS", 2, ticket, jobId)
23:         in("JOB", jobId, numberTasks, ticket, information, code);
24:         deliverToUser(result);
25:     commit transaction
26: end if
```

linha 2, a variável $transId$ recebe o valor 2 do campo correspondente da tupla. Este valor faz com a segunda transação seja executada (linhas 17-25). A última situação é resultado da confirmação da primeira transação como bem sucedida (a operação out na linha 13 foi executada), e da segunda transação sendo interrompida pela falha do *broker* (a operação in na linha 22 não foi executada). O objetivo do controle feito com o identificador de transação ($transId$) é evitar que tarefas sejam inseridas mais de uma vez no espaço de tuplas devido a um falha e reinicialização correspondente do *broker*.

A primeira transação começa obtendo, incrementando e escrevendo novamente a tupla de *ticket* no espaço de tuplas (linhas 6-7). Estas operações devem ser feitas dentro do contexto de transação, pois se a tupla *ticket* é removida do espaço é não é reinserida, então nenhuma outra aplicação será capaz de inserir suas tarefas no espaço. Após manipular o *ticket*, a transação 1 insere a tupla descrevendo a aplicação no espaço TS, e também suas tuplas de tarefas correspondentes (linhas 8-11). A transação 1 termina com a inserção da tupla de transação ($trans$) no espaço, indicando que a mesma foi concluída (linhas 12-13). A segunda transação obtém os resultados da execução das tarefas do espaço (linhas 18-21). Após isso, as tuplas de transação e da aplicação são removidas do espaço (linhas 22-23). Por fim, o resultado é entregue para o usuário de maneira confiável (linha 24).

4.3.2. Recurso

O algoritmo 2 descreve o funcionamento de um recurso r_i. O algoritmo inicia obtendo todas as tuplas de aplicação (*job*) disponíveis no espaço, através da operação $copy_collect$ e são armazenadas em um conjunto chamado de $jobList$ (linha 2). A função $chooseJob$

(linha 3), a qual não especificamos totalmente, é usada para escolher uma das tuplas de aplicação contidas em $jobList$ de acordo com algum critério. Para garantir um escalonamento justo, ou seja, onde a propriedade de *starvation freedom* é verificada, o critério de seleção da aplicação é baseado na tupla de menor valor de *ticket*.

Algoritmo 2 Resource r_i

```
procedure resource()
 1: loop
 2:   jobList ← copy_collect("JOB", *, *, *, *, *, *);
 3:   job ← chooseJob(jobList);
 4:   if (job ≠ ⊥) then
 5:     taskId ← chooseTask(job);
 6:     if (taskId ≠ ⊥) then
 7:       begin transaction                                   // gets and executes a task
 8:         inp("TASK", job.jobId, taskId, *, ?parameters);
 9:         result ← executeTask(job.jobId, taskId, parameters, job.code);
10:         out("RESULT", job.jobId, taskId, result);
11:       commit transaction
12:     end if
13:   end if
14: end loop
function executeTask(jobId, taskId, parameters, code)
15: repeat
16:   begin transaction                                       // executa parcialmente uma tarefa
17:     inp("CHECKPOINT", jobId, taskId, ?checkpoint)
18:     result, checkpoint, taskFinished ← partialExecute(code, parameters, checkpoint);
19:     if not (taskFinished) then
20:       out("CHECKPOINT", jobId, taskId, checkpoint)
21:     end if
22:   commit transaction
23: until (taskFinished)
24: return result
```

Depois da seleção da aplicação, o recurso seleciona a tarefa para executar de acordo com uma heurística – operação $chooseTask$ (linha 5). Recursos lentos executando tarefas muito grandes podem levar muito tempo em seus processamentos prejudicando o desempenho da conclusão da aplicação correspondente na grade.

Para minimizar a má combinação entre tamanho de tarefa e velocidade de recurso, distribuímos tanto tarefas como recursos em **classes**. Os recursos são classificados de acordo com suas velocidades em classes $\mathcal{CR} = \{r_1, \ldots, r_{nc}\}$. Por exemplo, três classes ($nc = 3$) podem ser distinguidas: classe com recursos até 1GHz; classe de recursos com velocidade de 1 a 3GHz; e a terceira classe com valores de velocidade acima de 3GHz. As tarefas são distribuídas em classes $\mathcal{CT} = \{t_1, \ldots, t_{nc}\}$ tomando como base, por exemplo, seus respectivos tamanhos. O *broker* que é responsável por distribuir as tarefas em classes, coloca informações sobre estas distribuições nas tuplas correspondentes de cada tarefa (campo $information$). O mesmo número (nc) de classes de tarefas e de recursos determina a correspondência e, portanto, a atribuição de elementos entre as mesmas: a classe r_i deve incluir os recursos mais lentos e a classe t_i as tarefas menores; a classe r_nc, por sua vez, inclui os recursos mais rápidos e a classe t_nc as maiores tarefas.

Os recursos começam pegando tarefas de classes correspondentes, ou seja, um recurso que pertence a classe r_i deve tentar executar uma tarefa da classe t_i. Se não existirem tarefas da classe de seu nível (nível i), o recurso passa a procurar tarefas em classes

superiores (tarefas maiores). Esta procura começa na classe t_{i+1} e se repete até a classe t_{nc}; se não existirem mais tarefas nesta classe t_{nc}, então o recurso começa a tentar obter tarefas em classes inferiores (tarefas menores), começando por t_{i-1} e, a cada insucesso em classe, passa a uma ordem inferior até atingir t_1. Se não existirem mais tarefas, significa que todas as tarefas da aplicação foram (ou estão sendo) executadas. Forçando recursos rápidos executarem tarefas maiores primeiro, e recursos lentos executarem tarefas menores primeiro, a probabilidade de uma tarefa grande ser executada por um recurso lento é reduzida e o tempo de execução da aplicação tende a se tornar também menor.

Depois que uma tarefa é escolhida, uma transação é iniciada (linhas 7-11). Nesta transação, a tarefa escolhida é removida do espaço de tuplas (linha 8), executada (operação $executeTask$ - linha 9) e o resultado é inserido no espaço (linha 10). A transação garante que a seqüência destas três operações seja executada de forma atômica. Se o recurso falhar durante a transação, a tupla descrevendo a tarefa é devolvida ao espaço e ficará disponível para outro recurso.

Em um ambiente de grade, com centenas, milhares, ou até mesmo milhões de recursos, entradas, saídas e falhas de recursos são muito freqüentes. Tarefas podem levar muito tempo para ser executadas (horas ou mesmo dias). Não seria, neste caso, conveniente retomar sempre do início a execução de tarefas quando recursos falham ou saem da grade. Para minimizar este problema, o *GridTS* usa um mecanismo de *checkpointing*, que consiste em salvar periodicamente o estado de um recurso (um *checkpoint*) em um meio de armazenamento estável [Koo and Toueg 1987]. Assim, se o recurso falhar, então outro recurso pode continuar a execução da tarefa a partir do último *checkpointing* realizado.

A execução de uma tarefa é descrita pelo Algoritmo 2 (linhas 15-24). O *GridTS* usa o próprio espaço de tuplas como meio de armazenamento estável. Portanto, quando um recurso está executando uma tarefa, uma tupla de *checkpoint* é inserido no espaço periodicamente. Antes de iniciar a execução da tarefa, o recurso deve procurar por uma tupla de *checkpoint* no espaço de tuplas (linha 17). Se existir, a tarefa inicia a sua execução do estado descrito neste *checkpoint*.

A execução de uma tarefa segundo o algoritmo 2 envolve o uso de uma transação de alto nível aninhada [Liskov and Scheifler 1983] linhas 16-22. Se o recurso falhar quando estiver executando na transação aninhada, as duas transações do algoritmo 2 são abortadas. Porém, se a tupla *checkpoint* for inserida dentro do contexto de uma transação aninhada confirmada (linha 20), esta deve permanecer no espaço de tuplas. Ou seja, o retrocesso da transação pai não deve remover esta tupla. Este é o propósito do uso do conceito de transações de alto nível aninhadas.

5. Avaliação

Esta seção apresenta uma comparação, baseada em simulações, de diferentes algoritmos de escalonamento. A métrica de desempenho utilizada em todas simulações foi o tempo total de execução de todas as tarefas de uma aplicação, também conhecido como *makespan*. A seção a seguir apresenta, por motivos de espaço, somente três algoritmos de escalonamento para aplicações *bag-of-tasks* de vários presentes na literatura.

5.1. Algoritmos de Escalonamento

O MFTF (Most Fit Task First) [Wang et al. 2005] é um algoritmo de escalonamento que usa informações de tamanho da tarefa e a carga e a velocidade do recurso para construir a alocação das tarefas. O algoritmo escalona a tarefa "mais adequada" para um dado

recurso. O cálculo para saber o quão adequada uma tarefa é para um recurso é definido como: $fitness(i, j) = \frac{100000}{1+|W_i/S_j - E_i|}$. W_i corresponde ao tamanho da tarefa i. E_i é o tempo de execução esperado para a tarefa i. S_j é a velocidade do recurso j obtida pelo escalonador através do serviço de informação. W_i/S_j é o tempo de execução estimado para a tarefa i usando o recurso j. $(W_i/S_j) - E_i$ é a diferença entre o tempo de execução estimado e o tempo de execução esperado. Quanto menor esta diferença, mais adequada é a tarefa é ao recurso considerado. Determinar o E_i é muito importante no método de escalonamento.

O Workqueue [Cirne et al. 2003] é um algoritmo de escalonamento que não usa nenhuma informação sobre os recursos para fazer o escalonamento das tarefas. A primeira tarefa que estiver aguardando para ser escalonada é associada a um recurso de maneira aleatória. Este procedimento é executado até que todas as tarefas sejam escalonadas. Depois que uma tarefa é executada, o escalonador associa outra tarefa para o recurso. O problema desta abordagem é quando uma tarefa grande é associada a um recurso pequeno, a execução de toda a aplicação pode ser atrasada até o término da execução desta tarefa.

O WQR (*Workqueue with Replication*) [Silva et al. 2003] impõe o mesmo procedimento de escalonamento do Workqueue. No entanto, quando não existem mais tarefas para serem executadas e restam recursos disponíveis, as tarefas que ainda estão sendo executadas são replicadas nestes recursos disponíveis. Quando uma réplica de uma tarefa termina de ser executada, todas as outras réplicas têm suas execuções canceladas. A idéia por trás dessa abordagem é que quando uma tarefa é replicada, existe maior chance da tarefa ser associada para um recurso mais rápido. As simulações mostraram que o WQR tem um bom desempenho se sempre existirem recursos disponíveis para a replicação das tarefas, mas este não é o caso de ambientes de grades onde, geralmente, existem diversos usuários competindo por recursos para a execução de suas aplicações.

5.2. Ambiente de Simulação

Foram executadas 2490 simulações e repetidas cada uma 10 vezes de maneira a avaliar o *GridTS* em relação a outros três algoritmos de escalonamento: Workqueue, WQR e MFTF. O *GridTS* foi simulado usando uma, três e cinco classes de recursos e de tarefas (denotados por *GridTS*1, *GridTS*3 e *GridTS*5, respectivamente) e WQR usando somente duas réplicas (e por isto referenciado nos testes por WQR2x). Para o MFTF, nos testes realizados, foram usadas informações "atualizadas" sobre recursos e tarefas, algo como já dito neste texto, muito difícil de se obter em ambiente de grade.

As simulações foram efetuadas usando o simulador GridSim [Buyya and Murshed 2002]. Todas as simulações usam o mesmo valor para a velocidade da grade, ou seja, a soma da velocidade de todos os recursos: 1000. A **velocidade do recurso** representa o desempenho de um recurso ao executar uma tarefa. Por exemplo, um recurso com velocidade 5 pode executar uma tarefa com tamanho 100 em 20 unidades de tempo. Consideramos que que o tamanho da aplicação é 6000000. Em um mundo ideal, o *makespan* desta aplicação seria 6000 unidades de tempo. Fixando a velocidade da grade e o tamanho da aplicação, a variação do *makespan* deve-se somente pelas diferenças dos algoritmos de escalonamento. Relacionado com as comunicações, assumimos que os tempos de transmissão são negligíveis, uma vez que, pelo modelo *bag-of-tasks*, tarefas não interagem entre si e possuem poucos dados de entrada/saída.

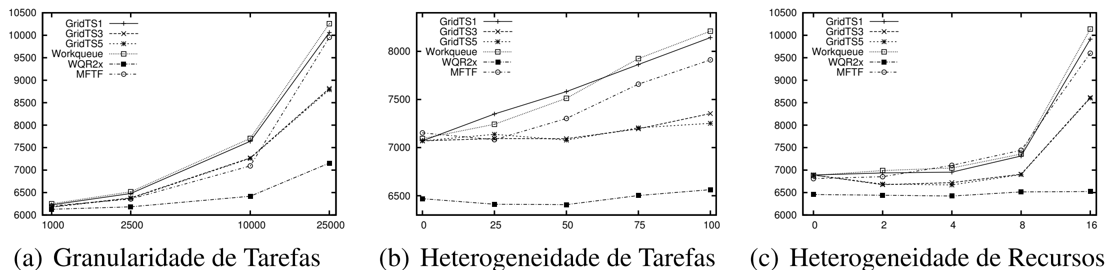
Em um ambiente de grade, o *makespan* depende de diversos parâmetros, como o número de recursos e tarefas, a **granularidade da tarefa** (tamanho da tarefa), a **heteroge-**

neidade das tarefas (variação do tamanho das tarefas) e a **heterogeneidade dos recursos** (variação da velocidade dos recursos) e a **carga de falhas** (número de falhas de recursos). A combinação desses parâmetros definem os ambientes de execução (cenários).

A **velocidade dos recursos** da grade tem uma distribuição $U(10 - hm/2, 10 + hm/2)$, onde $U(a, b)$ representa uma distribuição uniforme de a até b e os valores usados para hm foram 0, 2, 4, 8 e 16. Quanto ao **tamanho das tarefas**, foram considerado quatro tamanhos de tarefas, com média de 1000, 2500, 10000 e 25000. Quanto maior o tamanho das tarefas, menor é o número de tarefas por recurso. A **variação do tamanho das tarefas** foi de de 0%, 25%, 50%, 75% e 100 %. Por exemplo, uma variação de 0% significa que todas as tarefas tem o mesmo tamanho (tarefas homogêneas), enquanto uma variação de 50% significa que tarefas possuem tamanhos correspondendo a uma distribuição uniforme $U(7500, 12500)$. A **carga de falhas** define a porcentagem dos recursos que falham (por parada) durante um experimento da simulação. Quando não existe carga de falhas, significa que todos os recursos da grade se comportam corretamente. Consideramos também que recursos que falham não retornam à grade.

5.3. Simulação sem Falhas

Os resultados apresentados nesta seção mostram o desempenho dos algoritmos de escalonamento em um ambiente sem carga de falhas. Os parâmetros que foram variados nas simulações são: a granularidade das tarefas e as heterogeneidades dos recursos e das tarefas. A Figura 2 mostra os resultados das simulações com variação destes parâmetros.



(a) Granularidade de Tarefas (b) Heterogeneidade de Tarefas (c) Heterogeneidade de Recursos

Figura 2. Média do *makespan* variando, (a) granularidade das tarefas, (b) heterogeneidade das tarefas e (c) heterogeneidade dos recursos (sem falhas)

Granularidade das Tarefas. A Figura 2(a) mostra o *makespan* médio com diferentes granularidades de tarefas (1000, 2500, 10000, 25000). Cada ponto foi obtido através da média de todos os níveis de heterogeneidade de tarefas e de recursos. Pode ser observado que quando as tarefas são menores, os escalonadores tendem a ter desempenhos similares. Isto deve-se ao fato de existir muitas tarefas por recurso, de modo que todos os recursos tendem a ficar ocupados a maior parte do tempo. Porém, conforme o tamanho das tarefas crescem, a diferença do *makespan* para os diferentes escalonadores também cresce.

Como esperávamos, o *GridTS1* tem desempenho similar ao Workqueue. Com tarefa grandes, ambos têm o maior *makespan*, visto que tarefas grandes podem ser escalonadas para recursos lentos no final da execução, levando ao aumento do *makespan*. A figura 2(a) mostra também que o uso de classes no *GridTS* (*GridTS3*, *GridTS5*) minimiza este efeito, forçando os recursos a executar tarefas mais adequadas a suas características, a probabilidade de uma tarefa grande ser executada por um recurso lento torna-se menor. O *GridTS* é melhor quando o número de tarefas por recurso é grande.

O WQR tem melhor desempenho que outros escalonadores porque o mesmo replica tarefas quando os recursos se tornam disponíveis. Porém, esta abordagem perde

a sua característica de desempenho na simultaneidade de várias aplicações que determinam uma maior ocupação dos recursos. Quando as tarefas começam a se tornar muito grandes e, portanto, teremos menos tarefas por recurso, o desempenho do WQR também começa a piorar. A razão para isto, é que o efeito de recursos rápidos começa a ser menos significativos em replicações de tarefas grandes.

O MFTF tem bom desempenho somente quando as tarefas são pequenas. A justificativa para isso é que o MFTF escalona uma tarefa para o recurso mais adequado, mas este pode não ser o recurso mais rápido. Portanto, a solução escolhida pelo escalonador pode não levar ao melhor *makespan*, mas pode conseguir um tempo de execução estável similar ao tempo de execução esperado (E_i) para cada tarefa. O cálculo do E_i é crucial para obter o melhor *makespan* possível, mas na prática isto é difícil de se obter. Nas simulações, configuramos o E_i sendo o tamanho médio da tarefa dividido pela média da velocidade dos recursos.

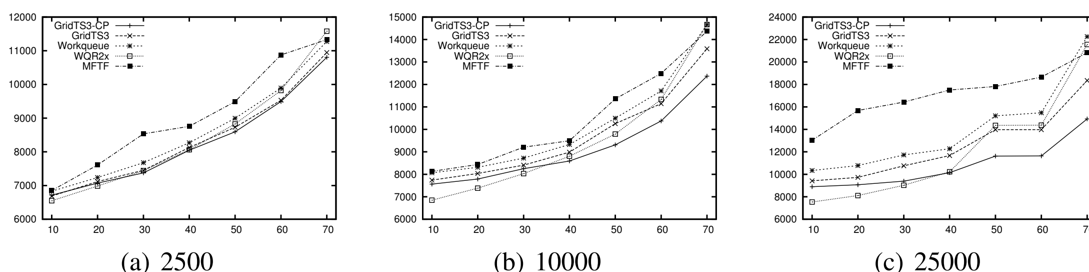
Heterogeneidade das Tarefas. A Figura 2(b) avalia o comportamento de cada escalonador com diferentes níveis de heteroneidade de tarefas. O desempenho do WQR permanece quase inalterado, em todos os casos, devido a seu esquema de replicação. Usando classes no *GridTS* (*GridTS3* e *GridTS5*) faz seu desempenho também permanecer quase inalterado. O desempenho alcançado pelo *GridTS3* e *GridTS5* pode ser creditado pela habilidade de um recurso mais rápido escolher uma tarefa maior para executar. O desempenho do MFTF torna-se pior quando a heterogeneidade das tarefas aumenta: quanto maior a diferença entre os tamanhos de tarefas, menores serão os valores de adequação de uma tarefa a um recurso, prejudicando o *makespan*.

Heterogeneidade dos Recursos. A Figura 2(c) mostra o desempenho dos escalonadores com diferentes níveis de heteroneidade de recursos. Novamente, o *GridTS*1 tem desempenho similar ao Workqueue e o desempenho do WQR permanece quase inalterado em todos os casos. Os desempenhos do *GridTS*3 e *GridTS*5 permanecem quase inalterados quando o nível de heterogeneidade dos recursos é menor ou igual a 8 e com nível 16 seu desempenho diminui. MFTF apresenta o mesmo desempenho já citado acima.

5.4. Simulações com Falhas

Esta seção apresenta o comportamento dos algoritmos quando estão sujeitos a diferentes cargas de falhas e a influência do uso do mecanismo de *checkpointing* para o *GridTS*.

As simulações foram feitas considerando diferentes porcentagens do total de recursos falhos, em um tempo aleatório, durante a execução das tarefas. Para o *GridTS* somente é mostrado seu desempenho para três classes. Nos experimentos, cada ponto foi obtido através da média de todos os níveis de heterogeneidade de recursos. Na Figura 3 três níveis de granularidade de tarefas são mostrados (2500, 10000, 25000), com variação de 50% entre os tamanhos das tarefas de cada nível.



(a) 2500 (b) 10000 (c) 25000

Figura 3. Média do *makespan* considerando falha dos recursos.

Como pode ser observado, quando há mais do que 50% dos recursos sujeitos a falhas, o desempenho do *GridTS3* torna-se melhor do que o WQR. A razão para isto é que quando muito recursos falham, a chance de um recurso estar disponível para replicar as tarefas é reduzida, assim o WQR acaba funcionado como o Worqueue Assim como em ambientes não sujeitos a falhas, o MTFT também não tem bom desempenho em ambientes sujeitos a falhas. Novamente, isto é devido a dificuldade de calcular um bom valor para E_i. Este foi calculado sem considerar faltas no sistema, visto que não sabemos como esta informação poderia ser incorporada no cálculo do mesmo.

A Figura 3 também nos permite avaliar que o uso do *checkpointing* no *GridTS* permite aumentar o seu desempenho, principalmente quando as tarefas são maiores, devido a quantidade de processamento perdido que pode ser evitado é maior. Quando mais do que 30% dos recursos estão sujeitos a falhas, o WQR torna-se pior que *GridTS3*.

5.5. Resumo da Avaliação

As simulações nos levam a diversas conclusões interessantes. A primeira é que o *GridTS* com 3 ou 5 classes apresenta um *makespan* melhor que a maioria dos outros algoritmos, com exceção do WQR quando o número de falhas de recursos não é muito alto (neste caso o *GridTS* é melhor que o WQR). No entanto, nas simulações, o WQR se beneficia do fato que cada simulação foi feita para uma única aplicação, assim o WQR tem a oportunidade de usar recursos adicionais (recursos mais rápidos que finalizam os seus processamentos) na replicação de tarefas e reduzir o *makespan*. Porém, em grades que estão permanentemente executando diversas aplicações simultaneamente este cenário de replicação não é muito fácil de se caracterizar.

É especialmente interessante observar que o *GridTS* tem melhor desempenho que o MFTF; isto é devido a não trivialidade em definir o parâmetro (E_i). O *GridTS* não está sujeito a esta dificuldade. Outra conclusão interessante, é que o desempenho do *GridTS*, com classes de tarefas e de recursos, tanto para três níveis como para cinco classes, comporta-se de maneira similar. Aparentemente, melhorias de desempenho no GridTS não parecem ser muito significativas com o uso de mais de três níveis de classes.

Finalmente, as simulações confirmam o resultado esperado de que o mecanismo de *checkpointing* tem um efeito positivo no *makespan* quando há falhas nos recursos, mais ainda quando a carga de falhas definida é alta.

6. Conclusões

Este artigo apresentou o *GridTS*, uma infra-estrutura de grade descentralizada e tolerante a faltas, na qual são os recursos que devem procurar as tarefas para executar. Nesta abordagem o escalonador perde a forma centralizadora das outras abordagens no escalonamento. A comunicação é feita através de um espaço de tuplas, beneficiando-se do seu desacoplamento tanto no tempo quanto no espaço. O *GridTS* combina um conjunto de diferentes técnicas de tolerância a faltas – *checkpointing*, transações e replicação – para prover o escalonamento tolerante a faltas.

Comparamos o desempenho do *GridTS* usando um grande número de simulações no GridSim. O mecanismo de escalonamento descentralizado provou ser bastante eficiente quando comparado com outros da literatura, além de não necessitar de um serviço centralizado para obter informações atualizadas da grade.

O artigo apresentou o *GridTS* em detalhes, incluindo os algoritmos executados tanto pelos *brokers* quanto pelos recursos. A abordagem GridTS pode ser facilmente implementada e integrada em plataformas de grade existentes.

Referências

Alpern, B. and Schneider, F. (1984). Defining Liveness. Technical report, Department of Computer Science, Cornell University.

Bakken, D. E. and Schlichting, R. D. (1995). Supporting fault-tolerant parallel programming in Linda. *Transactions on Parallel and Distributed Systems*, 06(3):287–302.

Buyya, R. and Murshed, M. (2002). GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *The Journal of Concurrency and Computation: Practice and Experience*, 14(13–15):1175–1220.

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

Carriero, N. and Gelernter, D. (1989). How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions Computer Systems*, 3(1):63–75.

Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauvé, J., Silva, F., Barros, C. O., and Silveira, C. (2003). Running bag-of-tasks applications on computational grids: The MyGrid approach. In *International Conference on Parallel Processing (ICCP-03)*, pages 407–416.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Fujimoto, N. and Hagihara, K. (2003). Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *International Conference on Parallel Processing (ICPP-03)*, pages 391–398.

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

Koo, R. and Toueg, S. (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31.

Liskov, B. and Scheifler, R. (1983). Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Programing Languages and Systems*, 5(3):381–404.

Rowstron, A. I. T. and Wood, A. (1998). Solving the Linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358.

Silva, D., Cirne, W., and Brasileiro, F. V. (2003). Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *9th International Euro-Par Conference (EURO-PAR'03)*, pages 169–180.

Smith, J. A. and Shrivastava, S. K. (1996). A system for fault-tolerant execution of data and compute intensive programs over a network of workstations. In *2nd International Euro-Par Conference (EURO-PAR'96)*, pages 487–495.

Wang, S.-D., Hsu, I.-T., and Huang, Z. Y. (2005). Dynamic scheduling methods for computational grid environments. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 22–28.

Xu, A. and Liskov, B. (1989). A design for a fault-tolerant, distributed implementation of Linda. In *19th Symposium on Fault-Tolerant Computing (FTCS'89)*, pages 199–206.