

Mecanismo de Tolerância a Falhas para a Linguagem Distribuída DPC++¹

Maurício Lima Pilla² Marcos Ennes Barreto³
Rafael R. dos Santos⁴ Gerson G. H. Cavalheiro⁵
Philippe O. A. Navaux⁶

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Caixa Postal 15064, Porto Alegre, RS

Resumo

Este trabalho apresenta um mecanismo e algoritmo para criar *checkpoints* e aumentar a confiabilidade de DPC++. O mecanismo proposto permite a criação de *checkpoints* distribuídos e a recuperação de objetos DPC++ na ocorrência de falhas. Um ambiente de testes foi criado para validar o modelo em questão. Ao final, é apresentada uma análise dos resultados obtidos e de futuros trabalhos.

Abstract

This work presents a mechanism and algorithm to create checkpoints and increase the reliability of DPC++. This mechanism allow the creation of distributed checkpoints and the recovery of DPC++ objects, when a fault occurs. A test environment was generated to validate this model. Finally, an analysis of the results is presented and the future works are introduced.

¹Projeto financiado pelo CNPq e FINEP através do Programa PAD

²Bolsista IC-CNPq, bacharelado, Instituto de Informática, UFRGS, email:pilla@inf.ufrgs.br

³Bolsista IC-CNPq, bacharelado, Departamento de Informática, ULBRA, email:barreto@inf.ufrgs.br

⁴Mestrando, CPGCC, UFRGS, email:rrsantos@inf.ufrgs.br

⁵Doutorando INPG-IMAG-França, email: Gerson.Cavalheiro@imag.fr

⁶Professor Doutor (Grenoble, 1979), Instituto de Informática, UFRGS, email:navaux@inf.ufrgs.br

1 Introdução

Um dos serviços mais comuns em tolerância a falhas é a recuperação do sistema de software para um estado consistente [JAL94]. A recuperação de erros, que restaura o sistema para um estado consistente, é um passo essencial para suportar tolerância a falhas.

Em sistemas uniprocessados, quando um erro é detectado, a recuperação deste não é uma tarefa difícil. Uma abordagem possível para este problema é o uso de técnicas de *recuperação de erros por retrocesso* [SIN94]. *Checkpoints* (ou pontos de recuperação) são estabelecidos periodicamente durante a execução normal do processo através do salvamento de todas as informações necessárias para recomeçar o processo, armazenado-as em um meio estável. A informação salva inclui o valor das variáveis do processo, seu ambiente, informações de controle, valor dos registradores, etc.

Quando um erro é detectado, o processo é recuperado para um estado, previamente salvo, através da recuperação do último *checkpoint* armazenado do processo que falhou. No caso do nodo falhar, a restauração do estado pode ocorrer depois que o nodo falho tenha sido reparado ou através da realocação (migração) do objeto para um nodo apto a continuar sua execução.

Em sistemas distribuídos, quando existem múltiplos processos em comunicação, o estabelecimento de *checkpoints* e *rollback* do sistema não é uma tarefa trivial. A principal dificuldade se deve ao fato de que o estado do sistema agora inclui os estados de diferentes processos que estão executando em diferentes nodos. E o evento de estabelecer um *checkpoint* é uma ação que pode ser executada em cada nodo somente para seus dados locais (para sistemas distribuídos, considera-se o *checkpoint* de um nodo como todas as informações de todos os processos que estão executando no nodo). Cada nodo pode estabelecer um *checkpoint* localmente em determinado instante de tempo, mas não existe um método direto para estabelecer um *checkpoint* ao mesmo tempo de todos os diferentes nodos para capturar o estado atual do sistema em uma instância de tempo. Esta necessidade de uma visão global requer que algum outro método seja empregado para estabelecer um *checkpoint* global do sistema.

Um método simples de criação de *checkpoints* assíncronos [SIN94] poderia fazer com que cada nodo estabelecesse um *checkpoint* independentemente e a coleção destes *checkpoints* seria tida como um *checkpoint* global do sistema. Intuitivamente, pode-se ver que num sistema com processos em comunicação isto não pode ser considerado como um estado válido para que o sistema possa ser restaurado, pois as mensagens trocadas entre os processos podem não estar apropriadamente refletidas neste estado.

Outro método para a criação de *checkpoints* seria manter um conjunto consistente de *checkpoints* [SIN94], sendo que cada processo tomaria seu *checkpoint* após cada envio de mensagem. Desta forma, o conjunto dos *checkpoints* mais recentes estará sempre consistente. Porém, este método pode resultar em mensagens órfãs e, conseqüentemente, em um estado inconsistente do sistema. Além disto, este esquema necessita que a operação de enviar ou receber uma mensagem e a criação de *checkpoints* constituam uma operação atômica.

Atualmente, sistemas distribuídos têm sido amplamente utilizados nos sistemas de computação em geral. Este fato deve-se, principalmente, à grande difusão do uso de redes de computadores. O objetivo do uso destes sistemas é prover um aumento considerável no poder de processamento. Sua estrutura é basicamente a seguinte: vários processos executam em processadores diferentes (ou iguais) e comunicam-se através de mecanismos como, por

exemplo, a troca de mensagens, buscando resolver algum problema em conjunto. Como o uso dos recursos do sistema torna-se mais intenso, também aumenta a possibilidade de ocorrência de falhas. Além do aumento do poder de processamento e da possibilidade de falhas, o uso de processamento distribuído, com vários nodos executando ao mesmo tempo, cria uma situação de redundância, o que pode ser utilizado para aumentar a confiabilidade do sistema. Um nó da rede pode ser substituído por outro em caso de *crash*, bem como processos que estejam executando neste nó falho podem ser disparados e recuperados em outros nós.

Técnicas de tolerância a falhas são utilizadas, neste sentido, visando detectar erros produzidos por falhas e recuperá-los, levando o sistema para um estado anterior consistente, restaurando seu contexto. Para isso, faz-se uso de técnicas de recuperação de erros, que consistem de mecanismos que garantem uma recuperação transparente, finita e consistente, cujo objetivo é recuperar a computação feita por processos que falharem.

O presente artigo aborda na seção 2 a linguagem *DPC++*, explanando suas principais características. A seção 3 consiste do modelo distribuído utilizado pelo *DPC++*. A seção 4 traz a definição de *estado global consistente de um sistema distribuído*. A seção 5 consiste da *descrição do modelo de checkpoints*. Na seção 6 é abordada a *proposta do ambiente de simulação/monitoração*. A seção 7 apresenta os *resultados da simulação* e a seção 8 traz uma *conclusão* sobre o artigo.

2 A Linguagem DPC++

Processamento Distribuído em C++ (*DPC++*) é uma linguagem para programação distribuída orientada a objetos baseada em C++. É uma linguagem de propósito geral, que dispõe de recursos que visam facilitar a programação de grandes sistemas. No entanto, sua principal área de atuação são as aplicações que necessitam de concorrência para melhorar seu desempenho, pois possui recursos para distribuição de tarefas.

As características de orientação a objetos de *DPC++* são as herdadas do C++. Inclusive a sintaxe dos programas é a mesma. Porém, em *DPC++*, foi introduzido um novo tipo de classe: a classe dos objetos distribuídos. Em [CAV93], são encontradas outras considerações e restrições da linguagem *DPC++*.

A simplicidade da escrita de aplicações distribuídas é garantida pelo uso de recursos de manipulação do ambiente distribuído, provido pelo pré-processador *DPC++*. Este gera o código que será submetido ao compilador C++.

O mecanismo de comunicação utilizado é o de *sockets* (datagramas - UDP). Este mecanismo não oferece controle de fluxo, controle de erros e não é orientado à conexão. No entanto, o uso deste protocolo de comunicação é mais simplificado e permite um desempenho melhor do sistema com confiabilidade aceitável quando empregado em redes locais [MID90, SAN93].

Segundo [CAV94], uma das principais vantagens desta linguagem é que ela concilia as facilidades da programação orientada a objetos com a obtenção de melhores índices de desempenho na execução do programa, o que é possível através da execução distribuída da aplicação.

Os conceitos básicos de orientação a objetos são aplicados ao modelo *DPC++*, onde existem objetos que encapsulam todas as suas propriedades: dados (memória interna) e funções (métodos). A execução de programas é feita invocando-se métodos dos objetos,

através do envio de mensagens. Ainda, no modelo DPC++ é explorada a concorrência de execução entre objetos, utilizando-os em sistemas distribuídos.

3 O Modelo Distribuído

A linguagem DPC++ utiliza como modelo base de objetos distribuídos a execução da função destes em uma rede de processadores homogêneos, onde cada nodo pode suportar n objetos distribuídos executando, sendo este número limitado apenas pela capacidade de memória local disponível. Um escalonador é responsável por compartilhar o uso do processador entre os objetos.

3.1 Objetos Distribuídos

O objeto distribuído é, no modelo proposto, a unidade básica de execução. O esquema deste objeto é apresentado na figura 1. Neste esquema, é possível distinguir os métodos e o estado interno do objeto, como é encontrado nos objetos implementados pelas linguagens seqüenciais tradicionais, consistindo respectivamente nos serviços prestados pelo objeto e nos seu conjuntos de dados privados. O novo elemento introduzido é a interface de acesso, sendo sua função possibilitar que o objeto distribuído receba invocações de serviços e retorne os resultados da execução destes serviços.

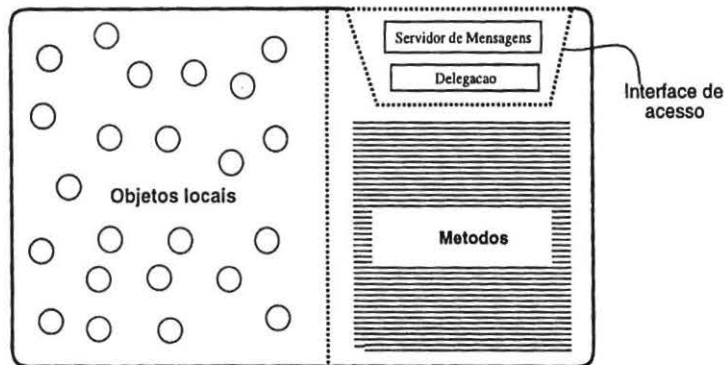


Figura 1 Objeto Distribuído.

A interface de acesso possui um *servidor de mensagens* e um elemento responsável pela ativação dos serviços solicitados, o elemento de *Delegação*. O servidor de mensagens é responsável pela comunicação do objeto distribuído, realizando a manipulação de mensagens que chegam ao objeto e enviando mensagens contendo respostas. As mensagens recebidas consistem em invocações de métodos e são armazenadas em uma fila para atendimento. Este servidor identifica, através da mensagem, o objeto originador, possibilitando o envio de respostas.

O elemento de delegação ativa os métodos invocados pelas mensagens recebidas, retirando-as da fila de mensagens. As invocações são tratadas na sua ordem de chegada e apenas uma invocação é tratada em determinado instante.

Um objeto distribuído é composto basicamente por três partes:

- Métodos: o conjunto de métodos corresponde à implementação de todos os serviços oferecidos pelo objeto.
- Estado Interno: o estado interno (ou memória) é composto pelo conjunto de dados manipulados pelo objeto. Estes dados não podem ser referenciados diretamente por outros objetos distribuídos, pois são considerados locais.
- Interface de Comunicação: é através desta que o objeto recebe as mensagens de invocação dos seus métodos. A interface é ligada diretamente à rede de comunicação e é endereçada pelo identificador do objeto. Sua função é organizar as mensagens recebidas, por ordem de chegada, ativar os métodos correspondentes e, se necessário, enviar mensagens com respostas às solicitações recebidas.

3.2 Diretório

Sua estrutura é semelhante à dos objetos distribuídos. Sua tarefa é realizar o controle dos objetos do programa do usuário e da carga de processamento de cada nodo. Nele são centralizados os pedidos de criação de objetos distribuídos e é decidido onde instanciar objetos, de acordo com as taxas de processamento de cada nodo.

Existem objetos auxiliares para o controle do processamento, os chamados *objetos espíões*. Em cada nodo da rede é instanciado um objeto deste tipo, cuja estrutura é igualmente semelhante à dos objetos distribuídos. Sua função é contabilizar a carga computacional do nodo onde se encontra.

O Diretório mantém uma tabela da carga computacional dos nodos, que é atualizada regularmente através de solicitações aos objetos espíões sobre a carga de processamento de cada nodo. Então, no momento da criação dos objetos, esta tabela é consultada e o nodo que apresenta a menor carga é selecionado para instanciá-lo. Por outro lado, o programador pode definir um nodo específico para a instanciação de um objeto distribuído. Neste caso, o sistema ativa o objeto sem consultar a carga de trabalho do nodo.

O objeto Diretório manipula ainda uma tabela de controle dos objetos distribuídos. Nesta tabela encontram-se informações relativas a cada objeto, como: identificação dos objetos criados (através da qual são endereçadas as mensagens), identificação do objeto que requisitou sua criação e o nodo onde está instanciado.

Apesar de ser um serviço centralizado, o Diretório não é requisitado freqüentemente pelos demais objetos, os quais mantêm maior interação entre si depois de criados. Portanto, a centralização destes serviços não é proibitiva em relação à performance do sistema.

4 Estado Global Consistente de um Sistema Distribuído

Em sistemas distribuídos, um evento pode ser uma transição espontânea de estado, ou o envio/recebimento de uma mensagem feitos por um processo [BEL93]. Podemos portanto representar um dado evento ε como $\varepsilon(p, t, e)$, onde p e t designam respectivamente o processo e a data de execução de uma dada instrução e . Entre dois eventos ε e ε' , o evento ε possui uma ordem temporal inferior à ε' ($\varepsilon < \varepsilon'$) caso:

- $p = p'$ e $t < t'$; ou,
- e' é o evento de recebimento de uma mensagem originada por e , sendo $p \neq p'$.

Em um dado instante de tempo i , o estado local L de um processo p é definido como a soma do seu estado inicial com o somatório dos eventos $\varepsilon(p, t, e)$ pertencentes ao processo, tal que $t \leq i$: $L_i^j = E_i^j + \sum_{j=0}^i \varepsilon(p, t, e)$. O estado global do sistema com n processos num instante de tempo i é definido pelo conjunto G_i de estados locais dos processos: $G_i = \{L_i^0, \dots, L_i^n\}$. O estado dos canais para um estado global G_i é o conjunto de mensagens enviadas mas não recebidas em G_i .

A figura 2 mostra a ocorrência de eventos na linha de tempo de dois processos, denotados por pontos. As setas representam mensagens trocadas entre os processos, e o estado global em determinado instante é representado pelos cortes verticais c e c'

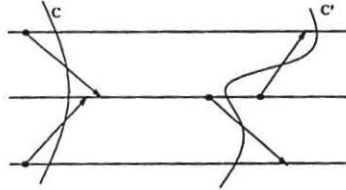


Figura 2 Estado global consistente e inconsistente.

Informalmente, um corte (estado global) é consistente se nenhuma seta começa à direita do corte e termina à esquerda deste. Esta noção de consistência induz à observação de que uma mensagem não pode ser recebida antes de ser enviada. Por exemplo, os cortes c e c' na figura 2 são estados consistente e inconsistente, respectivamente.

5 Descrição do Modelo de Checkpoints

A facilidade oferecida pelo ambiente de programação DPC++ para criação de aplicações distribuídas leva a uma situação onde mais recursos do sistema são utilizados simultaneamente, o que eleva a possibilidade de ocorrência de falhas. Portanto, é totalmente justificável, e até desejável, que haja a inclusão de algum mecanismo de tolerância a falhas que aumente a confiabilidade dos programas. Um exemplo de aplicação onde há necessidade de maior confiabilidade são os sistemas de longo tempo de execução, nos quais uma falha pode fazer com que o sistema tenha que ser executado novamente do início, desperdiçando tempo de processamento.

Em aplicações científicas e comerciais, a computação de um programa que foi interrompido por uma falha tem de ser refeita do início novamente. Como resultado, as aplicações são terminadas apenas se há um intervalo de tempo sem falhas longo o suficiente para que elas processem até o final [ZOM96]. Foi demonstrado em outros estudos que o tempo médio de execução de um programa na presença de falhas cresce exponencialmente com o tamanho do programa. A utilização de *checkpoints* faz com que o tempo médio de execução de um programa cresça linearmente com o tamanho do programa [ZOM96].

Por isto, como mecanismo de tolerância a falhas para aplicações DPC++ foi escolhida a técnica de *checkpoints* e *roll-back recovering*, por sua simplicidade de implementação e pela generalidade que oferece. A biblioteca utilizada para a criação dos checkpoints é a *libckpt*, descrita com detalhes em [PLA95].

Como simplesmente a criação de *checkpoints* não garante o estabelecimento de um estado global consistente, definiu-se um protocolo para a criação de *checkpoints*. Este

método consiste em um algoritmo que garante que nunca um ponto de recuperação criado possa ocasionar a perda de mensagens. Como o comportamento dos programas DPC++ é bem definido e a troca de mensagens restrita (objetos distribuídos podem invocar métodos de outros objetos ou receberem pedidos de métodos e respondê-los, sem que seja possível alterar o estado de um objeto de outra forma, só trocando mensagens nestas ocasiões), é possível a criação de *checkpoints* consistentes entre objetos distribuídos criando a cada mensagem enviada um ponto de recuperação no objeto emissor e no objeto receptor.

Considera-se que o meio de transmissão é confiável o suficiente para garantir que todas as mensagens enviadas pelo nodo origem são corretamente recebidas pelo nodo destino [MID90]. O problema de confiabilidade do meio de transmissão não é considerado inicialmente.

A detecção de falhas ocorre através de mecanismos de *timeout*, utilizados na troca de mensagens entre objetos, sendo que o Diretório verifica o *status* dos objetos mediante requisição e os recupera, se necessário.

Para que o problema da perda de mensagens não ocorra, pois o protocolo UDP/IP não o garantirá, serão criados pontos de recuperação sempre que dois objetos trocarem mensagens para invocação de métodos e para retorno de resultados. O procedimento ocorrerá da seguinte forma, sendo que p e p' são dois objetos distribuídos:

1. p envia mensagem de solicitação/retorno para p'
2. p cria seu *checkpoint*
3. p espera pela confirmação de p'
4. se p não recebe confirmação em determinado tempo (*timeout*), então
 - (a) solicita ao Diretório *status* do objeto p'
 - (b) se p' está "morto" é então recriado pelo Diretório e restaurado a partir do seu último ponto de recuperação
 - i. o Diretório retorna o novo endereço de p' para p , mais uma informação sobre o último *checkpoint* de p'
 - ii. a partir desta informação recebida, p determina qual a próxima ação a ser executada
 - (c) senão, p volta para 3
5. senão
 - (a) p envia confirmação para p'
 - (b) p prossegue execução normalmente

Paralelo a este procedimento, no objeto p' :

1. p' espera por uma mensagem de solicitação/retorno de p
2. p' recebe uma mensagem de solicitação/retorno e cria seu *checkpoint*
3. p' envia uma mensagem de confirmação para p
4. p' espera por uma mensagem de confirmação de p

5. se p' não recebe confirmação em determinado tempo (*timeout*), então
 - (a) solicita ao Diretório *status* do objeto p
 - (b) se p está "morto" é então recriado pelo Diretório e restaurado a partir do seu último ponto de recuperação
 - i. o Diretório retorna o novo endereço de p para p' que está aguardando, mais uma informação sobre o último *checkpoint* de p
 - ii. a partir desta informação recebida, p' determina qual a próxima ação a ser executada
 - (c) senão, p' volta para 4
6. senão
 - (a) p' prossegue execução normalmente

A informação sobre o último ponto de recuperação de um objeto é um indicativo do *checkpoint* criado imediatamente antes da falha deste objeto, significando que o *checkpoint* foi criado após o envio ou o recebimento de uma mensagem, as únicas duas possibilidades quando utilizado o mecanismo acima. Com esta informação é possível determinar se:

- há necessidade de retransmitir a última mensagem ou;
- é preciso esperar pela retransmissão de alguma mensagem.

Através desta técnica garantimos a criação de pontos de recuperação consistentes e mantemos o estado global do sistema consistente. O número de mensagens aumenta, pois para cada mensagem enviada são necessárias duas outras de confirmação. As mensagens de confirmação (*ACK*) são mensagens com tamanho de 1 byte.

O algoritmo de criação de *checkpoints* apresentado por [KOO87] exige um número maior de mensagens, pois quando um processo deseja criar um *checkpoint* envia mensagens a todos os processos de quem recebeu mensagens desde a criação de seu último *checkpoint*. Estes processos, por sua vez, executam o mesmo procedimento, fazendo com que o número de mensagens seja bastante grande em função do número de processos (objetos). O outro problema associado à criação de *checkpoints* deste modo é que um grande número de processos criam *checkpoints* ao mesmo tempo, o que pode diminuir o desempenho do sistema em certos momentos.

A figura 3 ilustra o fluxo de mensagens entre dois objetos, p e p' . Inicialmente, o objeto distribuído p envia uma mensagem $m1$ ao objeto distribuído p' e ambos criam seus *checkpoints*. Depois, o objeto p' envia uma mensagem de confirmação (*ACK*) ao objeto p , que então envia uma mensagem de confirmação para p' e prossegue normalmente.

Após enviar a mensagem de confirmação para p , p' falha. Quando p tenta enviar uma mensagem $m2$ para p' , cria seu *checkpoint* e aguarda pela confirmação de p' . Como p' não responde em um determinado *timeout*, p envia uma mensagem ao Diretório, pedindo o *status* do objeto p' . O Diretório descobre que p' falhou e dispara um novo objeto da mesma classe ($rsh(p')$). Este novo objeto utiliza as informações armazenadas no último estado armazenado do objeto que falhou para restaurar seu contexto e continuar a execução.

O Diretório devolve para p o novo endereço de p' . O objeto p envia novamente a mensagem, sem, contudo, criar um novo *checkpoint*, e espera pela confirmação. p' envia a

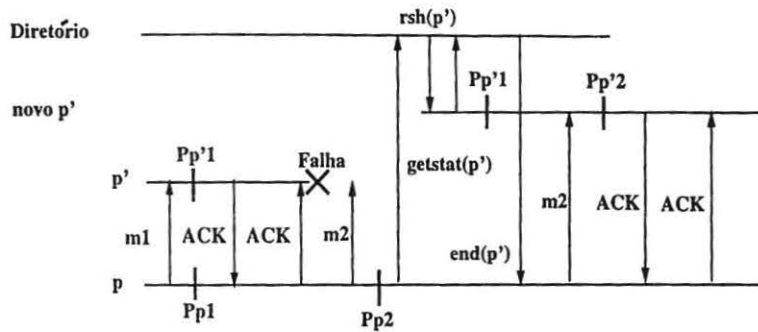


Figura 3 Fluxo de mensagens entre dois objetos distribuídos

confirmação e espera pela confirmação de p . Depois disto, ambos os objetos prosseguem normalmente.

No DPC++, o objeto distribuído que recebe uma mensagem cria seu ponto de recuperação armazenando seu estado atual e a mensagem recebida. Caso ocorra uma falha, não é necessário que o objeto emissor da última mensagem retorne a um ponto anterior ao do envio da mensagem. No pior caso, o objeto emissor apenas terá que reenviar uma mensagem, sem que seja necessário restaurar seu estado a partir do seu último *checkpoint*. Assim, a recuperação de objetos é facilitada pois na maioria dos casos somente o objeto que falhou é recuperado e restaurado.

Para garantir que foram criados *checkpoints* tanto no objeto que envia quanto no que recebe a mensagem, foi utilizado um esquema de "commit" onde um objeto só prossegue depois de receber a confirmação de que o outro criou um *checkpoint*. A utilização das mensagens de confirmação garante que os *checkpoints* foram realmente criados, o que evita uma recuperação posterior do sistema para um estado inconsistente.

6 Proposta do Ambiente de Simulação/Monitoração

Para validar o modelo de *checkpoints* em DPC++, foi desenvolvida uma aplicação de teste baseada no gerador de fractais de Mandelbrot em DPC++ [CAV94], a qual possui os seguintes objetos:

- Objeto MAIN : é o objeto que inicia a execução do gerador de fractais, criando o Diretório e solicitando a este a criação dos objetos de cálculo e do objeto de saída. Também é o encarregado da distribuição das áreas do fractal para os objetos de cálculo, bem como da finalização do objeto.
- Objetos de Cálculo : são responsáveis pelo cálculo de áreas parciais do fractal, distribuídas pelo MAIN, e pelo envio deste para o objeto de saída.
- Objeto de Saída : consiste no objeto que recebe as partes do fractal, já calculadas, bem como a posição destas partes na tela, plotando cada um dos pontos.
- Diretório : é o objeto responsável pelas solicitações de criação de novos objetos distribuídos, pelo armazenamento dos endereços destes e pela recuperação, quando

necessário.

Para facilitar a análise foi adotado um ambiente simplificado. A criação de *checkpoints* ficou resumida à solicitação e ao retorno da distribuição das áreas do fractal a serem calculadas. Os demais métodos, como construtores e destrutores, bem como o envio das áreas calculadas para o objeto de saída, foram deliberadamente excluídos da análise e deixados sem a criação de *checkpoints*. A abordagem também não levou em conta possíveis falhas do Diretório. Este estudo será apresentado em trabalhos futuros.

O teste do algoritmo de criação de *checkpoints* consistiu na injeção de falhas em diversas partes dos objetos *MAIN* e *Cálculo*, fazendo com que os processos correspondentes falhassem em determinados momentos da execução, como antes e depois do envio ou recebimento de uma mensagem ou da criação de *checkpoints*. A partir disto, foram analisadas as saídas ocasionadas por comandos de depuração posicionados antes e depois das trocas de mensagens e criação de *checkpoints*. O processamento prosseguiu normalmente para falhas de um objeto de cada vez. Outros tipos de falhas, como falhas múltiplas do objeto *MAIN* e dos objetos *Cálculo*, ainda estão sendo analisadas.

7 Resultados da Simulação

A simulação da aplicação proposta foi executada utilizando-se a variação do número de objetos de Cálculo e a criação de *checkpoints* ou não. As *workstations* utilizadas foram uma *SPARCstation 2* e uma *SPARCstation IPC*, ambas da Sun Microsystems, com o sistema operacional SunOs 4.1. O compilador utilizado foi o g++ versão 2.5.8, da GNU.

Na avaliação do tempo necessário para a execução não foi considerada a ocorrência de falhas, o que certamente aumentaria o tempo de execução.

7.1 Tamanho dos checkpoints

O tamanho dos *checkpoints* da simulação ficou entre 180 Kbytes e 200 Kbytes para o objeto *MAIN* e 160 Kbytes para os objetos de Cálculo.

Um programa de teste foi utilizado para medir o tamanho dos *checkpoints* em relação à área de dados dos objetos. Este programa é formado por todas as bibliotecas utilizadas normalmente por aplicações DPC++, como bibliotecas de comunicação, mais o estritamente essencial para que o objeto distribuído resultante pudesse criar *checkpoints* e se comunicar com um Diretório hipotético, ou seja, um procurador para Diretório e um objeto do tipo Comunicação.

A variação do tamanho da área de dados foi obtida através de uma matriz, a qual era dimensionada para simular uma área de dados de tamanho menor ou maior. A posição 0 do eixo X da figura 4 corresponde a um objeto distribuído sem nenhum tipo de estrutura de dados ou funções definidos pelo usuário, apenas aqueles criados pelo DPC++.

O tamanho dos *checkpoints* varia linearmente em função do tamanho da área de dados, sendo que o tamanho de um *checkpoint* para a aplicação mínima DPC++ foi de 75 Kbytes.

7.2 Tempo para Gravação de Checkpoints

O tempo médio para gravação de um *checkpoint* na aplicação simulada foi de 1,2 segundos. O programa de testes da seção anterior foi adaptado para medir o tempo de criação de

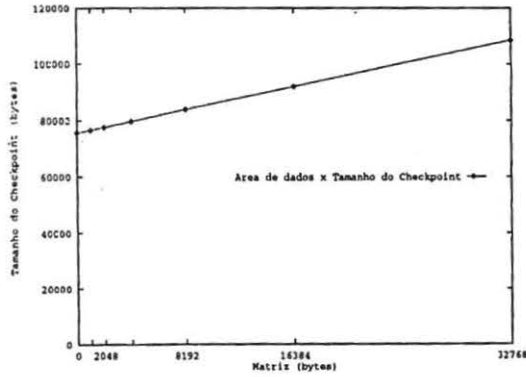


Figura 4 Tamanho do checkpoint em relação à área de dados

checkpoints para diversos tamanhos de matrizes. Com isto, foi possível determinar o tempo gasto na criação de *checkpoints*.

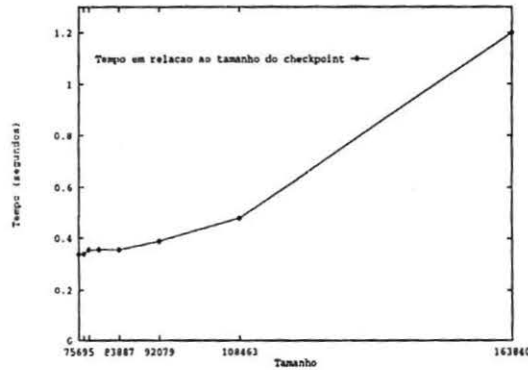


Figura 5 Tempo para gravação do checkpoint em relação à área de dados

7.3 Tempo de Execução

Para avaliar a perda de desempenho decorrente da utilização de *checkpoints*, a simulação foi executada variando-se o número de objetos de cálculo e mantendo-se a mesma carga de trabalho (um fractal de 100 x 100, dividido em 50 partes).

Além de haver um grande aumento no tempo necessário para a execução quando criados *checkpoints*, ocorreu um aumento no tempo de execução quando aumentado o número de objetos distribuídos. A explicação para isto está na própria estrutura da simulação e no aumento de mensagens trocadas. O objeto MAIN, que atuava como distribuidor das áreas do fractal para os objetos de Cálculo, recebia várias mensagens ao mesmo tempo, de objetos diferentes, podendo interpretar requisições de métodos como confirmações. Se

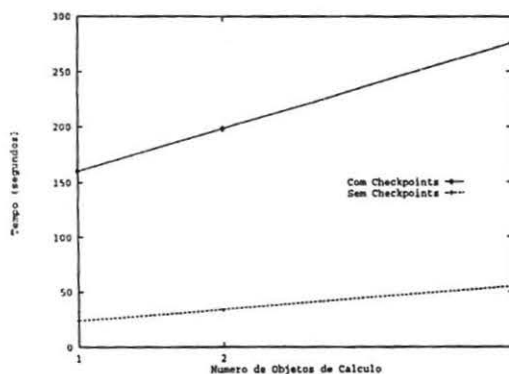


Figura 6 Tempo para execução da simulação com e sem checkpoints

este problema não fosse tratado, poderia acontecer que mensagens de confirmação fossem interpretadas como respostas e vice-versa, ou até mesmo casos em que um objeto ficasse esperando por uma confirmação que nunca chegaria.

A figura 7 mostra um caso onde ocorre o problema de mensagens trocadas. Um objeto *CALC1* envia uma requisição de método (*MSG1*) para *MAIN*, o qual envia a mensagem de confirmação (*ACK1*) para *CALC1*. Então, *CALC2* envia uma mensagem de requisição de método (*MSG2*) para *MAIN*, o qual interpreta a mensagem como sendo uma confirmação de *CALC1*.

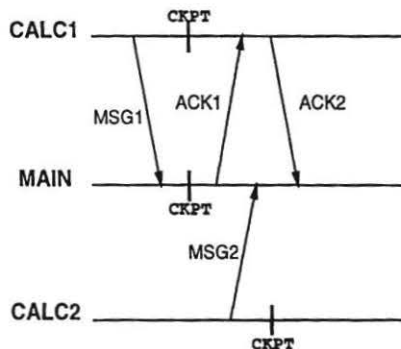


Figura 7 Mistura de mensagens de confirmação com outras mensagens

Para evitar esta situação, foi utilizado um esquema que testa as mensagens de confirmação (*ACK*) e rejeita aquelas que não contém um valor pré-determinado. Contudo, os objetos de Cálculo têm que retransmitir estas mensagens rejeitadas. Do contrário, as mensagens que fossem rejeitadas seriam perdidas. Ficou definido que cada dois *timeouts* com o objeto *MAIN* ainda em execução seriam considerados como uma mensagem rejeitada, ocorrendo a retransmissão da mesma. Um novo método para resolver este problema, mais otimizado e consistente, deverá ser apresentado posteriormente.

8 Conclusão

O mecanismo apresentado para a criação de *checkpoints* em DPC++ demonstrou ter várias características interessantes, pois linguagens de programação distribuída com técnicas de tolerância a falhas implementadas não são comuns. No DPC++, a utilização deste mecanismo é totalmente transparente ao usuário, bastando que seja indicado, no momento da compilação, que é necessário o emprego do mesmo. A utilização do mecanismo conseguiu fazer com que a aplicação suportasse falhas simples, aumentando a confiabilidade da mesma.

A troca de mensagens e a criação de *checkpoints* são otimizados em relação a outros algoritmos mais genéricos, os quais não levam em conta particularidades das aplicações. O mecanismo proposto somente cria *checkpoints* após uma troca de mensagens entre dois objetos, e, assim mesmo, apenas entre estes dois objetos. Quando é necessário realizar a recuperação de um objeto distribuído, apenas o objeto que falha é recuperado a partir de seu último *checkpoint*. Os demais objetos distribuídos não precisam retornar a um ponto anterior do processamento, podendo continuar normalmente a execução e evitando o *overhead* da recuperação de objetos que não falharam.

Futuras pesquisas podem ser realizadas nas questões da validação das mensagens, como canais exclusivos para confirmação (*ACK*) ou o uso de mensagens identificadas, no problema da confiabilidade do Diretório, no uso de objetos espíões para aumentar a eficiência da detecção de erros, na redução do tamanho dos *checkpoints* e nas otimizações para uso com aplicações de menor granularidade.

Referências

- [BEL93] BELMONTE, Valdir Rossi, WEBER, Raul Fernando. *Gerindo Tolerância a Falhas em Sistemas Distribuídos*. São José dos Campos: V Simpósio de Computadores Tolerantes a Falhas, *anais...*, outubro, 1993.
- [CAV93] CAVALHEIRO, Gerson G. H., NAVAUX, P. O. A.. *DPC++: Uma Linguagem para Processamento Distribuído*. Florianópolis: V SBAC-PAD, *anais...*, outubro, 1993.
- [CAV94] CAVALHEIRO, Gerson G. H., SANTOS, Rafael R., NAVAUX, Philippe O. A.. *Análise de Desempenho de um Protótipo da Linguagem DPC++*. Ca-xambu : XXI SEMISH, *anais...*, 1994.
- [JAL94] JALOTE, Pankaj. *Fault Tolerance in Distributed Systems*. P T R Pretince Hall. 1994.
- [KOO87] KOO, Richard, TOUEG, Sam. *Checkpointing and Rollback-Recovery for Distributed Systems*. IEEE Transactions on Software Engineering, vol. SE-13, no.1, January, 1987.
- [MID90] MIDKIFF, S. F. e VAIDYANATHAN, P. Performance evaluation of communication protocols for distributed processing. *Computer Communications*, 13:(5) junho 1990.

- [PLA95] PLANK, James S. et alli. *Libckpt: Transparent Checkpointing under Unix*. USENIX Winter 95 Technical Conference. 1995.
- [SAN93] SANTOS, Rafael R. dos; CAVALHEIRO, Gerson G. H.; NAVAU, Philippe O. A. *Mecanismo de Transporte para Comunicação entre Objetos Distribuídos*.
Simpósio Nacional de Redes de Computadores e suas Aplicações. Porto Alegre. SUCESU-RS. s.n, Agosto 1993.
- [SIN94] SINGHAL, Mukesh; SHIVARATRI, Niranjan G. *Advanced Concepts in Operating Systems*. McGraw-Hill. 1994.
- [YAU92] YAU, S. S., JIA, X., BAE, D. H.. *Software Design Methods for Distributed Computing Systems*. Computer Communications, 15(4):213-224, May, 1992.
- [ZOM96] ZOMAYA, Albert Y.H. *Parallel and Distributed Computing Handbook*. McGraw Hill. 1996.