X Simpósio Brasileiro de Arquitetura de Computadores

A Implementação de uma Arquitetura de Suporte à Alta Disponibilidade de Objetos

Raimundo da Silva Barreto[1] Claudionor Nunes Coelho Jr[2]
Antônio Otávio Fernandes[2]

[1]Depto de Ciência da Computação
Universidade do Amazonas
Av. Rodrigo Ramos, 3000 Aleixo
69077-000 Manaus-AM
e-mail: barreto@dcc.fua.br

[2]Depto de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702
31270-010 Belo Horizonte - MG
e-mail: {coelho, otavio}@dcc.ufmg.br

Resumo

Este artigo apresenta a implementação de uma arquitetura que objetiva é a provisão de alta disponibilidade aos objetos distribuídos com transparência e flexibilidade. Esta arquitetura utiliza os conceitos de reflexão computacional e sua implementação foi realizada usando um *protocolo de meta-objetos*. A arquitetura utiliza meta-objetos tanto para impor aos objetos o grau de disponibilidade desejado quanto para o próprio acesso a estes objetos. A localização do objeto ficou totalmente transparente, fazendo com que o acesso parecesse local. Também apresentamos resultados experimentais, mostrando o *overhead* de chamadas de métodos quando incorporando disponibilidade.

Abstract

This paper presents the implementation of an architecture whose goal is the high availability provision to distributed objects with transparency and flexibility. This architecture is based on computational reflection concept and its implementation was made using a meta-object protocol. The architecture uses meta-objects to incorporate the availability level desired to the objects as long as the access to these objects. The object localization is completely transparent to the clients, where it can be seen as local and not remote. We also present early experimental results showing the method call overhead when incorporating availability in objects.

1 Introdução

Os ambientes distribuídos têm sido muito utilizados na implementação de aplicações que requerem alta disponibilidade, provavelmente porque quando se fala em distribuição a redundância, que é sempre requerida na incorporação de disponibilidade, é uma característica natural. Nesses ambientes, o modelo de clientes e servidores é o mais utilizado. Este modelo tem dois níveis de abstração: os *servidores* que oferecem os serviços e os *clientes* que utilizam desses serviços para realizar suas aplicações.

O modelo de objetos distribuídos pode ser visto como uma extensão do modelo cliente/servidor, onde os servidores são os próprios objetos, os serviços são o acesso a seus métodos e os clientes são os usuários do objeto. Existem objetos distribuídos que são críticos para um grande número de aplicações clientes. Estes objetos possuem, portanto, o requisito de serem altamente disponíveis, ou seja, não podem parar de prestar seus serviços mesmo que certas falhas venham a ocorrer.

Reflexão computacional pode ser definida como uma computação (chamada de computação reflexiva) que atua sobre uma outra computação (neste contexto chamada de computação regular). Este conceito pode ser usado para garantir uma clara separação entre os requisitos funcionais e os não-funcionais de um sistema [10]. Os requisitos funcionais estão relacionados com a solução do problema em si. Entretanto, existem outros requisitos que dizem respeito, por exemplo, à distribuição, segurança, disponibilidade, dentre outros, que também precisam ser incorporados em um sistema. Estes são chamados de requisitos não-funcionais porque independem da funcionalidade da aplicação.

O objetivo deste trabalho é o de apresentar uma arquitetura que dê suporte à alta disponibilidade de objetos, onde a localização destes objetos é desconhecida dos clientes. O acesso aos métodos de um objeto é feito como se fosse local e não remoto e foi usado replicação de processos para a o aumento de disponibilidade. Algumas modificações dinâmicas, como o aumento ou diminuição do número de réplicas, pode ser feito sempre que necessário, mesmo que o objeto já esteja em execução.

Nossa proposta de solução consiste em utilizar os conceitos de reflexão computacional, como uma forma de incorporar disponibilidade de forma transparente aos objetos distribuídos. Neste caso, tanto a geração dos objetos replicados quanto o protocolo de consistência entre réplicas são escondidos do objeto.

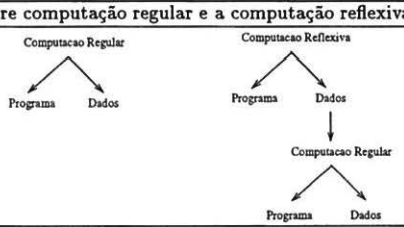
Este artigo está organizado da seguinte forma. Na Seção seguinte apresentaremos a definição de reflexão computacional. Na Seção 3 será visto os detalhes da arquitetura proposta. A Seção 4 examina os aspectos implementacionais deste trabalho. Na Seção 5 serão avaliados os resultados experimentais. Finalmente na última seção serão apresentadas as conclusões e os trabalhos futuros.

2 Reflexão Computacional

Maes [7] definiu reflexão computacional como a atividade executada por um sistema computacional quando computando sobre, e possivelmente afetando, sua própria computação. Neste sentido, um sistema reflexivo pode modificar a si próprio usando sua propria computação.

Podemos distinguir entre uma computação regular e uma computação reflexiva. Uma computação regular incorpora dados e programas. Uma computação reflexiva também incorpora dados e programas, entretanto, os dados de uma computação reflexiva são os dados e o programa de uma computação regular. Na Figura 1 podemos ver um contraste entre computação regular e reflexiva.

Figura 1 Contraste entre computação regular e a computação reflexiva.



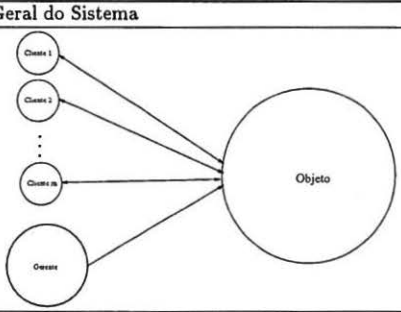
Em linguagens orientadas por objetos, a reflexão computacional é realizada através de dois níveis de abstração: nível base e nível meta. No nível base encontram-se os objetos e no nível meta encontram-se as extensões deste objeto. Existem duas formas de implementarmos estas extensões: através de meta-classes e através de meta-objetos [6]. Meta-classes são classes especiais cujas instâncias são também classes. São as chamadas classes de classes. Os meta-objetos

são objetos de nível meta que podem estar associados a objetos de nível base. Neste artigo iremos utilizar a associação por meta-objetos.

3 Arquitetura Proposta

A arquitetura de uma solução para o problema levantado pode ser sumarizada pela Figura 2, onde temos três entidades principais: clientes, objetos e gerente. O papel do *cliente* é solicitar que métodos sejam executados e aguardar pelo resultado. O *objeto* aguarda por pedidos, executa o método requerido e devolve o resultado ao chamador. O *gerente* atua na alteração da política de disponibilidade, ou seja, quantos objetos (servidores) devem existir para um certo número de métodos (serviços).

Figura 2 Arquitetura Geral do Sistema



Do ponto de vista do usuário, o acesso ao objeto tem que parecer local e a somente um objeto (embora além de poder não ser local, pode-se ter um grupo de objetos replicados). A disponibilidade será garantida através da replicação dos objetos e esta arquitetura levará isto em consideração. Algumas questões como: qual a política de replicação inicial de um objeto, qual o protocolo de sincronização entre as réplicas, como alcançar acordo em modificações no grupo de objetos (*agreement on a new membership*), como supervisionar falhas de membros, como se dará a confiabilidade da comunicação dentre outros é o assunto desta Seção.

3.1 Vigilância contra Falhas de Membros

O modelo de falhas usado neste trabalho é o de *parada de processo*. Neste, um processo simplesmente pára de trabalhar sem dar qualquer aviso. Uma restrição, é que a confiabilidade da comunicação será de responsabilidade dos protocolos de comunicação. Já estamos trabalhando em um outro projeto cujo objetivo é a implementação de difusão confiável [1].

Com o propósito de detecção de falhas, as réplicas formarão um *anel virtual* com passagem de um *token*. A confiabilidade da comunicação garante que o *token* nunca vai se perder ou ser replicado. As seguintes definições tem que ser sempre válidas.

Definição 3.1: Um anel virtual é um grafo direcionado $G = (V, E)$, onde $V = \{v_0, v_1, \ldots, v_{n-1}\}$ é o conjunto de nodos, $E = \{e_{0,1}, e_{1,2}, e_{2,3}, \ldots, e_{n-1,0}\}$ é o conjunto de arestas e n é o número de nodos.

Definição 3.2: O vizinho da esquerda do nodo i, para $0 \leq i < n$, é:

$$vizinho\_esq(i) = (i + 1) \ mod \ n$$

Definição 3.3: O vizinho da direita do nodo i, para $0 \leq i < n$, é:

$$vizinho\_dir(i) = ((i - 1) + n) \ mod \ n$$

Cada aresta $e_{i,j}$ representa um *canal de comunicação simplex* entre nodos i e j. Cada nodo i (para $0 \leq i < n$) é conectado a somente duas arestas: (i) $e_{i,vizinho\_esq(i)}$, o qual é usado pelo nodo i para transmitir o token para o nodo $vizinho\_esq(i)$, e (ii) $e_{vizinho\_dir(i),i}$ que é usado para o nodo i receber o *token* do nodo $vizinho\_dir(i)$. Neste caso, em cada nodo i é possivel (a) enviar o *token* usando o comando $send(token)_{\ vizinho\_esq(i)}$ e (b) receber o *token* usando o comando $receive(token)_{\ vizinho\_dir(i)}$. A partir desses conceitos, a forma de detecão de falhas de membros passa a ser uma atividade simples: se após ter passado um certo período de tempo e o token não tiver chegado a uma estação, esta saberá que uma estação falhou.

3.2 Configuração Inicial do Serviço

A configuração inical do serviço tem duas funções: (a) criação de todas as réplicas e (b) formação do anel virtual. Da mesma forma que o trabalho de Bernard e Conan [3], estamos supondo que o ambiente de execução possui um sistema de arquivos confiável. Neste caso, o código executável do objeto distribuído estará sempre disponível. O envio e recepção de *token* requer uma comunicação também confiável, portanto, a comunicação será com conexão e com confirmação.

Algoritimo 3.1 Algoritmo de configuração.

usuario inicia a $replica_0$
$replica_i$ inicia a $replica_{i+1}$, para $0 \leq i < n-1$
$replica_{n-1}$ estabeleçe uma conexão com $replica_0$
$replica_i$ estabeleçe uma conexão com $replica_{i+1}$, for $0 \leq i < n-1$
$replica_{n-1}$ envia o primeiro token

Como pode ser visto no Algoritmo 3.1, quando a $replica_0$ começa a ser executada, ela dá inicio ao processo de configuração do serviço. De uma forma geral, a $replica_i$ (para $0 \leq i < n-1$) inicia a $replica_{i+1}$ e aguarda por um pedido de conexão.

Após a criação da $replica_{n-1}$, esta inicia o processo de estabelecimento de conexão, pedindo uma conexão com a $replica_0$. O algoritmo prossegue com a $replica_i$ (para $0 \leq i < n-1$) pedindo uma conexão com a $replica_{i+1}$. Após receber e aceitar um pedido de conexão, a $replica_{n-1}$ começa a passagem do token.

3.3 Reconfiguração do Serviço

Neste trabalho as reconfigurações podem ocorrer em duas situações: (i) quando falhas tiverem ocorrido; (ii) quando o gerente requisitar a alteração no número de réplicas.

No caso de adicionar uma nova réplica, tanto para trocar uma réplica falha quanto para incrementar a disponibilidade, esta nova réplica deve ter o estado da computação atual, onde dependendo do tamanho deste estado pode ser gasto um tempo muito grande na transmissão. Schneider [8] propôs uma forma de integrar um novo elemento em um sistema replicado sem ter que parar o serviço. Nossa implementação utiliza este mecanismo.

No caso de reconfiguração por falha é preciso distinguir se o número de réplicas tem que ser mantido ou não. Se é para ser mantido, deve ser iniciado uma nova réplica em um nodo diferente para substituir a réplica falha. O estado da computação atual deve ser enviado para a nova réplica e o anel virtual deve ser refeito. Se o número de réplicas não é para ser mantido é necessário somente refazer o anel virtual. Existem dois algoritmos, um para a réplica que descobriu a falha (Algoritmo 3.2) e outro para as outras réplicas.

3.4 Políticas de Sincronização

Existem duas políticas principais de sincronização entre réplicas. A primeira é chamada de *loose synchronization*, que pressupõe a existência de uma réplica primária que mantem o estado do

Algoritimo 3.2 Máquina de estado para reconfiguração pela réplica que descobriu a falha.

```
reconfiguration_by_fault_1 : state_machine
    handle_node_failed : command
        multicast ("Node Failed", replica_f) to all replicas
        if (number of replicas should be maintained) →
            start_process (replica_new)
            send computational_state to replica_new
            send clients_requests to replica_new
            multicast ("New Replica Ready", replica_new) to all replicas
            if (left_neighbor(replica_new) = i) →
                accept a new connection from replica_new
            □ (right_neighbor(replica_new) = i) →
                connect a new connection from replica_new
            fi
        □ (number of replicas should not be maintained) →
            accept a new connection from right_neighbor(replica_f)
        fi
    end handle_node_failed
end reconfiguration_by_fault_1
```

serviço corrente. Nesta política há a necessidade de que esta réplica primária envie às outras réplicas mensagens de *checkpoint* passando o estado da computação atual. Na segunda política, *close synchronization*, os servidores são parceiros e interpretam todos os pedidos dos clientes em paralelo e cada um mantem seu respectivo estado interno que deve ser igual a todas as réplicas.

Algumas técnicas de replicação, que implementam uma ou outra política, foram propostas. Elas foram classificadas em três tipos: passiva, semi-ativa e ativa [2, 5].

Na *replicação passiva* existe uma réplica primária e $n-1$ réplicas *backups*. A réplica primária é quem presta o serviço e esta deve passar a todas as réplicas *backups* o seu estado da computação. Na *replicação semi-ativa*, o pedido do cliente é difundido a todas as réplicas e todas o processam. No entanto, quem devolve o resultado é sempre a réplica primária. Não há a necessidade de transmitir o estado da computação, uma vez que todas as réplicas executam o mesmo pedido. Na *replicação ativa* todas as réplicas também recebem e processam todos os pedidos dos clientes. Este protocolo pode ser usado de duas formas diferentes, conforme a aplicação seja determinística ou não. Se a aplicação for determinística pode-se usar um protocolo conhecido por *"replicação ativa competitiva"* onde a primeira réplica a terminar o serviço é quem responde ao cliente. Se a aplicação for não-determinística, os resultados gerados por cada réplica devem passar por um processo de votação majoritária.

4 Aspectos Implementacionais

A implementação da arquitetura proposta foi feita a partir dos conceitos de reflexão computacional, tendo em vista a necessidade de que a disponibilidade seja incorporada de forma transparente e flexível aos objetos. Nossa implementação foi realizada em um ambiente que executa o sistema operacional UNIX, a comunicação foi feita utilizando a interface *sockets* [9] e foi utilizado a ferramenta Open C++ [4] como suporte para reflexão.

Open C++ é uma variante da linguagem C++ onde chamadas de métodos e acesso à atributos são extensíveis. Desta forma, um programador pode implementar uma nova funcionalidade, através do nível meta, de forma transparente às aplicações no nível base. A única atividade que o programador da aplicação deve fazer é a associação do objeto no nível base com o meta-objeto no nível meta.

4.1 Vigilância e Controle de Falhas

Quando um nodo envia o *token*, um temporizador é iniciado. Desta forma, o vizinho da direita é dito estar falho quando um certo limite de tempo tenha expirado e o *token* não tenha chegado. A réplica que detectou a falha deve difundir uma mensagem a todas as outras réplicas, informando que seu vizinho da direita falhou.

Um servidor não pode ficar *bloqueado* esperando para receber o *token*. Uma forma de resolver este problema é usando *comunicação baseada em interrupção* [9], onde, quando uma mensagem chega, o processo é interrompido para tratar a mensagem.

Algoritimo 4.1 Máquina de estado para vigilância de falhas.

```
/* Esta maquina de estado deve ser executada para cada replica_i (para 0 ≤ i ≤ n − 1) */
fault_detection : state_machine
    var token : char;
    receive_token : command
        receive(token)_neighbor_right(i);
        stop schedule <reconfiguration_by_fault_1.handle_node_failed>
        schedule <fault_detection.send_token> for +WTST
    end receive_token
    send_token : command
        send(token)_neighbor_left(i);
        <fault_detection.wait_token>
    end send_token
    wait_token : command
        schedule
            <reconfiguration_by_fault_1.handle_node_failed>
        for +TDNF
    end wait_token
end fault_detection
```

A máquina de estado do Algoritmo 4.1 implementa um mecanismo de detecção de falhas, que deve ser executado por cada réplica, e foi construída baseada no trabalho de Schneider [8]. Nesta máquina de estado tem se dois limites superiores de tempo: *WTST* (*Wait Time to Send Token* [Tempo de Espera para o Envio do Token]) e TDNF (*Time to Detect Node Faulty* [Tempo para Detecção de um Nodo Faultoso]). O WTST é um tempo de espera para evitar um tráfico intensivo na rede de comunicação, implicando que após recebido o token, este só será reenvidado após serem transcorridos *WTST* unidades de tempo. O TDNF é baseado no número de réplicas atual e deve ser escolhido de tal forma que seja maior do que o tempo de propagação de uma mensagem na rede em uso. Após enviado o *token*, se o TDNF estiver expirado, isto significa que o vizinho da direita, da réplica em execução, falhou.

4.2 Protocolos de Replicação

Foram implementados, através de classes de meta-objetos, três protocolos de replicação: passiva, semi-ativa e ativa competitiva. Os protocolos de configuração e reconfiguração foram os mesmos para todos os protocolos de replicação.

4.2.1 Replicação Passiva

O envio do estado da computação pela réplica primária às réplicas *backups* se dá sempre após a chamada de um método reflexivo. Isto implica que o programador de aplicação deve tomar o cuidado de especificar como reflexivo, todos os métodos que atualizam o estado da computação. A mensagem contendo este estado é automaticamente gerada pelo meta-objeto da réplica primária de forma completamente transparente ao programa de nível base. A classe `PassiveReplication` foi definida na linguagem Open C++ [4] como:

```
#include "metaobj.h"
class PassiveReplication : public MetaObj {
public:
    void Meta_StartUp();
    void Meta_CleanUp();
    void Meta_ReceiveMethodCall(Id method_, Id category, ArgPac& args);
    void Meta_SendResultMethodCall(Id method_, Id category, ArgPac& reply);
protected:
    void Meta_MethodCall(Id methodid, Id category, ArgPac& args. ArgPac& reply);
    void Meta_SendCheckpoint(ArgPac& checkpoint);
    void Meta_GenerateCheckpoint(ArgPac& checkpoint);
    void Meta_ReceiveCheckpoint(ArgPac& checkpoint);
    void Meta_UpdateStateFromCheckpoint();
    Configuracao Config; /* configuracao/reconfiguracao */
    Datagram datagram_checkpoint; /* envio/recepcao checkpoint */
    Datagram datagram_service; /* envio/recepcao pedido e resposta */
    :
};
```

Estes métodos têm a seguinte função:

(a) `Meta_StartUp()`: Neste método são criados os objetos responsáveis pela configuração e reconfiguração dos objetos distribuídos. Este método é também o responsável pela inicialização de todos os aspectos de comunicação utilizando sockets.

(b) `Meta_CleanUp()`: Este método é o responsável pelo fechamento da interface de comunicação.

(c) `Meta_ReceiveMethodCall(...)`: Este método recebe, via interface de comunicação, uma chamada de método remoto. Este método está sempre aguardando a uma chamada de métodos.

(d) `Meta_SendResultMethodCall(...)`: Este método devolve o resultado da execução do método chamado ao chamador.

(e) `Meta_MethodCall(...)`: Este método implementa chamada real de métodos reflexivos. Após o retorno do método chamado, é gerado e enviado uma mensagem contendo o estado atual da computação.

(f) `Meta_SendMsgCheckpoint(...)`: Este método é usado pela réplica primária para enviar a todas as réplicas *backup* uma mensagem contendo o seu estado de computação. Este método chama o método `Meta_GenerateCheckpoint` e é retornado um objeto `ArgPac` contendo os valores dos atributos reflexivos.

(g) `Meta_GenerateCheckpoint(...)`: Este método gera um objeto `ArgPac` contendo todos os valores dos atributos reflexivos. Isto é feito através da chamada do método pré-definido pelo Open C++, chamado de `Meta_refl_Copy`.

(h)`Meta_ReceiveCheckpoint(...)`: Este método é usado pelas réplicas *backup* para receber uma cadeia de caracteres contendo o estado da computação da réplica primária. Esta cadeia é transformada em um objeto `ArgPac`.

(i) `UpdateStateFromCheckpoint(...)`: Este método é usado pelas réplicas *backup* para atualizar seu estado a partir do estado recebido da réplica primária. É utilizado um método definido pelo Open C++, chamado de `Meta_refl_Init`.

4.2.2 Replicação Semi-Ativa

Após o envio do resultado para o cliente pela réplica primária, deve haver um sincronismo entre as réplicas para evitar que, após uma falha, as réplicas possuam diferentes estados. A classe de nível meta `SemiActiveReplication` foi definida com os seguintes métodos: Meta_StartUp,

Meta_CleanUp, Meta_ReceiveMethodCall, Meta_SendResultMethodCall, Meta_MethodCall e Meta_Synchonization.

Os métodos da classe de nível meta `SemiActiveReplication` são parecidos com os métodos do protocolo de replicação passiva. A diferença é que neste protocolo há um método que faz a sincronização das réplicas após o envio da resposta pela réplica primária. Outra diferença é que não há necessidade de envio/recebimento de mensagens de estado da computação.

4.2.3 Replicação Ativa

Uma forma de garantir que somente uma réplica responda ao cliente pode ser a utilização de um protocolo de comunicação que garanta uma ordem de envio da mensagem. A classe de nível meta `ActiveCompetitiveReplication` foi definida com os seguintes métodos: Meta_StartUp, Meta_CleanUp, Meta_ReceiveMethodCall, Meta_SendResultMethodCall, Meta_MethodCall e Meta_DiscoverWinner.

Esta classe é parecida com os protocolos anteriores. A diferença está em quem deve responder ao cliente. Para tanto nesta classe existe um método específico para descobrir que réplica venceu nesta disputa (Meta_DiscoverWinner).

4.3 Acesso a Objetos Distribuídos

Foi provido, ainda, uma classe de nível meta que facilita o acesso remoto aos objetos distribuídos. Desta forma é escondido do programador os aspectos tanto de localização e ativação quanto de comunicação com objetos remotos. O programador de aplicação deve associar a classe do objetos que deseja acessar com a classe `MetaDistributedObjectAccess`.

Tendo em vista a necessidade de automatizar a localização de um objeto, o meta-objeto deve consultar a um outro objeto chamado de `PortMapper`. O objeto PortMapper é acessado da mesma forma que qualquer outro objeto. A única diferença é que sua localização é bem conhecida.

O objeto PortMapper é um ponto crítico de falha. Portanto, este também deve incorporar algum aumento de disponibilidade, cuja implementação é igual aos protocolos apresentados anteriormente.

A classe `MetaDistributedObjectAccess` foi definida da seguinte forma:

```
#include "metaobj.h"
class MetaDistributedObjectAccess : public MetaObj {
public:
    void Meta_StartUp();
    void Meta_MethodCall(Id methodid, Id category, ArgPac& args. ArgPac& reply);
protected:
    void Meta_SendMethodCall(Id methodid, Id category, ArgPac& args);
    void Meta_ReceiveResultMethodCall(Id methodid, Id category, ArgPac& reply);
    :
};
```

O método `Meta_MethodCall(...)` chama o método `Meta_SendMethodCall` para enviar uma chamada de método remoto para o respectivo objeto e recebe deste o resultado através do método `Meta_ReceiveResultMethodCall`.

5 Resultados Experimentais

Nos experimentos foi usado GNU G++ versão 2.6.2 e Open C++ version 1.2 como linguagens de programação básicas. O ambiente consiste de estações de trabalho executando o sistema operacional UNIX, conectada através do protocolo TCP/IP em uma rede local *Ethernet*.

Os experimentos foram executados sempre com três réplicas. O *WTST* (*Wait Time to Send the Token*) foi sempre de um segundo. Usamos cinco objetos com diferentes tamanhos de estado da computação (512b até 15Kb) e tempo de execuão de métodos (de 0,5s at'e 8s). Os *overheads* de tempo foram calculados, para cada protocolo implementado, a partir da comparação entre os objetos com e sem disponibilidade. A Tabela 1 apresenta os resultados.

Tabela 1 *Overhead* de tempo de chamada de métodos.

	$objeto_1$	$objeto_2$	$objeto_3$	$objeto_4$	$objeto_5$
Tamanho Estado	512B	1KB	5KB	10KB	15KB
Passiva	46.7%	207%	270%	302%	374%
Semi-Ativa	52.7%	185%	213%	208%	210%
Ativa	15.6%	111%	105%	109%	112%

Três fatores influenciaram neste *overhead*: (1) o mecanismo de detecção de falhas; (2) o protocolo entre réplicas; e (3) o mecanismo de reflexão computacional.

O tempo de execução do método do $objeto_1$ foi relativamente curto, neste caso, o mecanismo de detecção de falhas influenciou muito pouco. Por isso o *overhead* medido foi o menor. De uma forma geral, o protocolo de replicação passiva possui um fraco desempenho conforme há um aumento no tamanho do estado da computação, como pode ser visto do $objeto_1$ até o $objeto_5$.

Um resultado importante foi obtido a partir dos objetos $objeto_3$ até o $objeto_5$. Como os tempos de execução dos métodos são de 2,3 s até 7,6 s, o mecanismo de detecção de falhas passou a interferir no tempo de resposta e os resultados apontam para um valor constante de *overhead*, ficando em aproximadamente 210% (em média) para o protocolo semi-ativo e 108% para o protocolo ativo competitivo. O protocolo com melhor desempenho foi o protocolo ativo competitivo, onde a primeira réplica a terminar a execução do método é quem devolve o resultado para o cliente.

6 Conclusões

Neste artigo apresentamos uma implementação de disponibilidade com algumas características dinâmicas, em objetos distribuídos. Os objetos distribuídos ofereçem uma forma de se criar sistemas flexíveis uma vez que, estando os dados e operações sobre esses dados encapsulados dentro de um objeto, é perfeitamente possivel alterar a implemetação de um método sem alterar a interface do objeto. A forma de implementação de disponibilidade foi utilizando replicação de processos. Apresentamos, ainda, como as réplicas devem ser configuradas a fim de manusear a ocorrência de falhas e o processo de reconfiguração quando uma réplica falha tenha sido detectada.

Na fase de configuração todas as réplicas devem ser criadas e um anel virtual deve ser formado entre elas. A fase de reconfiguração é iniciada ou por causa de uma falha ou porque o gerente, que é um cliente especial, requisita adição/remoção de réplicas. A abordagem proposta foi baseada na utilização do conceito de reflexão computacional. Usando estes conceitos provemos transparência e flexibilidade ao programador de aplicação quando incorporando disponibilidade em objetos distribuídos.

Mostramos alguns resultados experimentais que apresentaram o *overhead* de tempo para a incorporação de disponibilidade usando três protocolos de replicação: passiva, semi-ativa e ativa competitiva. Os resultados apontaram um melhor desempenho do protocolo ativo competitivo.

Todo o acesso aos objetos distribuídos foi feito através de um meta-objeto. Tendo em vista a necessidade de facilitar ainda mais o acesso aos objetos distribuídos, estamos propondo, como trabalho futuro, a utilização de uma arquitetura CORBA e a incorporação de comunicação confiável através de difusão atômica. A abordagem usando Open C++ é estática, ou seja, depois de compilado não há como associar o objeto a um outro meta-objeto. Nossos estudos recentes buscam utilizarmos uma outra plataforma onde uma "migração" entre meta-objetos seja possível, de forma a privilegiar mudanças de políticas de replicação em tempo de execução.

Agradecimentos

Os autores agradecem a Denilson de Moura Barbosa, Ronaldo Duarte Campos, Fernando Fernandes Nunes Pereira e Bruno César Bernardes, pelas valorosas discussões sobre este trabalho. Este trabalho foi suportado pela CAPES, CNPQ, FINEP e FAPEMIG.

Referências

[1] R. Barreto. Um estudo e proposta de implementação de difusão confiável. Relatório técnico 98-1, DCC/UA, jan 1998.

[2] T. Becker. Application-transparent fault tolerance in distributed systems. In *Proceedings of the 2nd International Workshop on Configurable Distributed Computing*, pages 36–45, Pittsburg, março 1994.

[3] G. Bernard and D. Conan. Making distributed applications fault-tolerant in networks of unix workstations. In *Proceedings of the I2U Convention'93*, Milano, Italy, maio 1993.

[4] S. Chiba. Open c++ programmer's guide. Relatório técnico 93-3, Department of Information Science, University of Tokio, 1993.

[5] J-C. Fabre, V. Nicomette, T. Pérennou, R. Stroud, and Z. Wu. Implementing fault-tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, pages 489–498, Pasadena, CA, junho 1995. IEEE.

[6] M. L. Lisbôa. Reflexão computacional. *Minicurso, II Simpósio Brasileiro de Linguagens de Programação*, setembro 1997.

[7] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Object-Oriented Programming, Languages and Applications*, pages 147–155. ACM, outubro 1987.

[8] F. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, dezembro 1990.

[9] W. R. Stevens. *Unix Network Programming*. Prentice-Hall, 1990.

[10] R. Stroud and Z. Wu. Using metaobject protocols to separate functional and non-functional concerns: An example. Relatório técnico DeVa 14, Department of Computer Science, University of Newcastle upon Tyne, 1996.