4
Estudo de Caso: Middleware CORBA

Neste capítulo será feita uma análise do modelo proposto no capítulo 3 aplicado a um *middleware*, dentro do contexto de chamadas remotas de procedimento utilizando o padrão CORBA.

4.1
Remote Procedure Call

O RPC ou *Remote Procedure Call* é baseado na extensão do conceito das tradicionais chamadas locais de procedimentos. Com o RPC, pode-se construir de maneira transparente aplicações cliente/servidor, de forma que a função chamada possa existir em um espaço de endereçamento remoto. O RPC permite que toda a complexidade da troca de informações com a camada de comunicação de rede seja tratada de forma transparente, permitindo que os desenvolvedores se preocupem apenas com a lógica da aplicação.

Quando se executa uma chamada remota de método, os parâmetros são enviados atráves de um protocolo conhecido pelas duas partes envolvidas na comunicação. Normalmente, após a chamada de uma função, o processo permanece bloqueado até que a resposta seja retornada pelo servidor que executa o método. Após o retorno, o cliente continua o procedimento de execução[30, 33].

4.2
Common Object Request Broker Architecture

Com o conceito de chamada remota de procedimento, criou-se a possibilidade de existirem ambientes distribuídos operando através de plataformas bastante diferentes em termos de hardware e software. Tornou-se então evidente a necessidade de padronizar os métodos utilizados para a comunicação nesse espaço heterogêneo. O padrão CORBA foi projetado

PUC-Rio - Certificação Digital Nº 0321108/CA

para assegurar que as chamadas remotas de procedimento se tornassem tão simples de utilizar quanto o modelo de programação tradicional.

CORBA foi projetado com o objetivo de oferecer um ambiente de desenvolvimento com transparência de localização, o que permite que o acesso aos métodos oferecidos pela interface de um objeto sejam invocados da mesma maneira, não importando se o objeto CORBA se encontra no espaço de endereçamento local ou em um ambiente remoto. Além disso, o padrão CORBA pode interoperar com múltiplas linguagens de programação, permitindo que clientes e servidores sejam desenvolvidos em linguagens diferentes [5].

Dentre as linguagens que atualmente apresentam um mapeamento CORBA, podemos citar C, C++, Java, COBOL, Ada, Smalltalk, Lisp e Lua. O primeiro mapeamento para a linguagem Lua surgiu através do LuaOrb[16], uma ferramenta baseada nas propriedades reflexivas de CORBA e na natureza dinâmica de Lua. O uso de uma linguagem de *script* permite a rápida prototipagem, configuração dinâmica e um melhor suporte para testes, fazendo com que o sistema desenvolvido possa ser modificado e estendido sem precisar de compilação ou processos de amarração de código [16].

Embora se beneficie das vantagens de uma linguagem altamente flexível, o LuaOrb ainda depende de um OBR C++ oferecendo suporte à interface de invocação dinâmica e ao repositório de interfaces. O suporte CORBA na linguagem Lua evoluiu para uma implementação CORBA totalmente desenvolvida utilizando as ferramentas oferecidas pela linguagem Lua. Na seção 4.3 será apresentado o *ORB in Lua*, ou simplesmente OiL.

PUC-Rio - Certificação Digital Nº 0321108/CA

4.3 ORB in Lua

O OiL[24] é uma implementação de CORBA completamente escrita em Lua, desenvolvido com o objetivo de ser portável, flexível, e intuitivo de ser utilizado. Havia também a necessidade de manter ao máximo a compatibilidade com os padrões definidos pelo LuaOrb, permitindo que desenvolvedores já familiarizados com essa ferramenta e com a linguagem Lua pudessem desenvolver aplicações distribuídas em muito pouco tempo, ou acessar interfaces já expostas por outros servidores de forma interativa e simples.

A implementação do *middleware* em Lua permitiu que todo o poder de descrição de dados da linguagem pudesse ser aplicado de maneira a

simplificar a modelagem e a construção de aplicações CORBA, com os seguintes objetivos principais:

- **Portabilidade** - O sistema deve se aproximar do modelo de portabilidade do Lua, possibilitando sua execução em qualquer um dos sistemas operacionais principais como Windows e Unix, além de ser facilmente portável para outras arquiteturas que disponibilizem rotinas de comunicação com *sockets* e compatibilidade com o padrão ANSI C.
- **Simplicidade** - O uso do OiL deve ser simples para desenvolvedores com alguma experiência em programação Lua e acostumados com conceitos de objetos distribuídos.
- **Flexibilidade** - O *middleware* desenvolvido precisa contemplar não somente flexibilidade durante o desenvolvimento, mas também deve permitir métodos simples de configuração e disponibilização dos serviços, bem como atualizações em tempo de execução, sem que haja a necessidade de reiniciá-los.
- **Uso de recursos otimizado** - A plataforma de *middleware* deve poder ser executada utilizando os recursos disponíveis no sistema de maneira eficiente. Isso ajuda a melhorar o desempenho quando executado em PCs de médio porte e ao mesmo tempo viabiliza a execução em ambientes com limitações mais acentuadas de recursos, como PDAs e telefones celulares.

A comunicação entre os objetos distribuídos no OiL foi projetada para utilizar *sockets* síncronos, o que simplifica o processo de desenvolvimento do *middleware*, pois o tratamento das requisições segue sempre um fluxo linear. Porém, essa abordagem apresenta problemas críticos de desempenho quando se pensa em servidores tratando diversas conexões ou requisições de processamentos de longa duração.

Para minimizar esse problema, incorporamos o modelo de escalonamento de eventos proposto no capítulo 3 ao *middleware* OiL, fazendo com que as chamadas aos métodos bloqueantes sejam encapsuladas em um demultiplexador e sejam tratadas como corrotinas com processamento colaborativo concorrente.

PUC-Rio - Certificação Digital Nº 0321108/CA

4.3.1
Análise de Requisitos

O modelo de tratamento de requisições original do OiL baseava-se no processamento síncrono das requisições, fazendo com que ele se tornasse extremamente ineficiente no tratamento de múltiplos clientes ou de métodos demorados. Além disso, esse modelo não permitia o uso de objetos de *callback*, criando uma forte restrição à aplicabilidade do *middleware*.

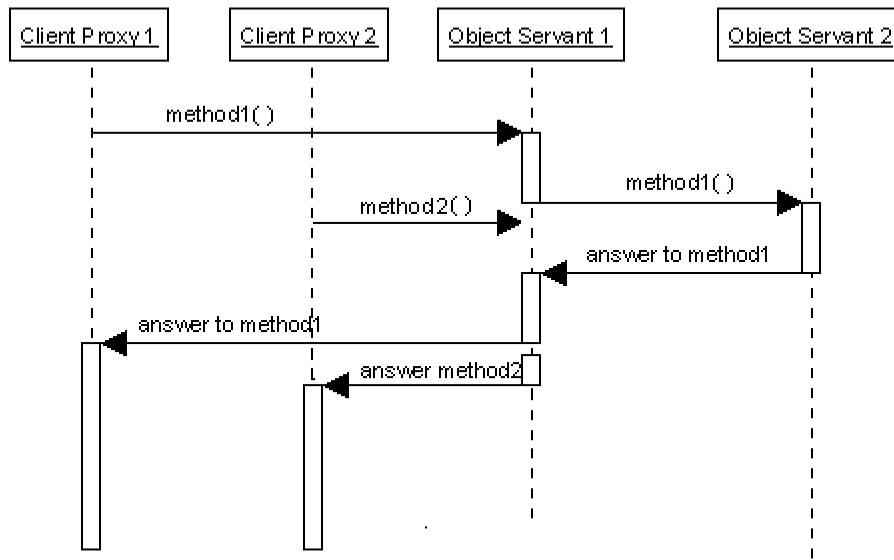

Figura 4.1: Seqüência de processamento original do *OiL*.

O diagrama da figura 4.1 mostra dois *proxyies* clientes efetuando chamadas remotas em um servidor. O método `method1()` é delegado para outro servidor, enquanto o método `method2()` é processado inteiramente pelo primeiro servidor. No modelo original, a requisição de processamento do método `method2()` fica postergada até o final do envio da resposta para o método `method1()`.

O objetivo da adaptação do modelo de escalonamento proposto para o *middleware* OiL é permitir que ocorra o processamento de diversas conexões de maneira simultânea, otimizando o tempo disponível de CPU no sistema e permitindo ganhos de escala no sistema. Com o escalonador, deve ser possível o tratamento concorrente de requisições, permitindo também o registro de eventos temporais, introduzindo o conceito de temporizadores no *middleware*.

A integração do modelo proposto deve ocorrer com a implementação de módulos externos ao núcleo do *OiL*, garantindo a simplicidade de incorporação no *middleware* ainda em fase de desenvolvimento, minimizando as

PUC-Rio - Certificação Digital Nº 0321108/CA

alterações no código do núcleo e permitindo uma manutenção mais estruturada, além de permitir que, por questões de otimização, essa funcionalidade possa ser removida em ambientes que não necessitem desse processamento concorrente.

Em sistemas onde os papéis de clientes e servidores não estão claramente distinguidos, esse processamento concorrente permite que os papéis se invertam, tornando possível que um servidor tome a iniciativa de invocar um método em um objeto localizado em um processo "cliente". Nesse tipo de sistema, a aplicação cliente precisa implementar e instanciar um objeto de *callback* CORBA e assumir algumas características de um servidor, precisando receber chamadas de métodos pela face servidor mesmo enquanto aguarda a finalização de um método invocado pela face cliente. Essa funcionalidade deve ser garantida na integração do modelo de escalonamento colaborativo para processamento concorrente[5].

4.3.2 Solução

A introdução de *multithreading* preemptivo em um sistema remete a uma solução onde pode-se utilizar uma linha de execução para processar cada requisição, atingindo o objetivo de processa-las de maneira concorrente. Como visto anteriormente, essa solução traz consigo um conjunto de dificuldades para o desenvolvimento que ferem os conceitos básicos da implementação do *ORB in Lua* e que podem ser evitadas com sistemas sem preempção. Embora existam bibliotecas de extensão que garantem suporte a *threads* na linguagem Lua[18], não existe suporte nativo a esse modelo. Lua oferece uma implementação de corrotinas assimétricas e *stackfull*, que aliada à modelos de demultiplexação de eventos garante um modelo para processamento colaborativo simples e eficiente.

Embora em Lua não exista o conceito de classe, através do uso de tabelas pode-se criar estruturas com forma e comportamento bem definidos emulando o conceito de classe. Esse conceito permite simplificar a tarefa de modularização do código, fazendo com que a implementação da estrutura proposta siga bastante próxima do modelo estruturado de desenvolvimento do *OiL*.

PUC-Rio - Certificação Digital Nº 0321108/CA

Integração com LuaSocket

O primeiro passo para a adaptação do modelo proposto foi desenvolver uma camada que permitisse o funcionamento assíncrono das chamadas tradicionais de *sockets* da biblioteca LuaSocket[17]. Essa camada cria uma interface com um método público `scheduler.tcp()`, responsável por criar os *sockets* “assíncronos”, e dois métodos de comunicação `scheduler.send()` e `scheduler.receive()`, que substituem os método análogos dos *sockets* tradicionais. A implementação desses métodos pode ser encontrada na listagem 4.1.

```
function tcp()
  ...
  local skt = socket.tcp()
  skt:settimeout(0)
  ...
  return skt
end

function receive(client, pattern)
  ...
  repeat
    s, err, part = client:receive(pattern, part)
    if s or err ~= "timeout" then return s, err end
    if is_running then coroutine.yield("read", client) end
  until false
  ...
end

function send(client, data)
  ...
  repeat
    s, err, sent = client:send(data)
    if s or err ~= "timeout" then return s, err end
    if is_running then coroutine.yield("write", client) end
  until false
  ..
end
```

Listing 4.1: Camada para utilização de *sockets* não bloqueantes.

No método `scheduler.tcp()`, cria-se um objeto *socket* com *timeout* de transmissão igual a zero, fazendo com que as operações bloqueantes executadas nesse objeto retornem quando houver indisponibilidade do dado desejado. Dessa maneira, permite-se que nos métodos `scheduler.send()` e `scheduler.receive()` utilize-se o mecanismo de corrotinas para retornar o controle ao escalonador sem que ocorra a perda do contexto da comunicação. Essa implementação, em conjunto com o método `select()` padrão, cria uma camada de acoplamento sobre os *sockets* tradicionais do *LuaSocket* que permite a sua utilização como uma *Fonte de Eventos* integrada com o *Observador de Eventos*.

PUC-Rio - Certificação Digital Nº 0321108/CA

O Mapeamento do Modelo de Escalonamento

A flexibilidade das tabelas em Lua também permite a implementação das filas do *Repositório de Tratadores de Eventos* e do controlador de eventos temporizados, usado como fonte produtora de eventos temporais. Este último é implementado com uma fila de objetos, onde guarda-se o tempo absoluto do primeiro evento (o que está mais próximo de expirar) e o tempo dos demais é armazenado como tempo relativo ao primeiro elemento da fila.

O modelo do *Escalonador* utilizando corrotinas permitirá desacoplar três partes fundamentais do processamento de chamadas remotas de métodos. Os métodos de comunicação de rede serão tratados pelo *Observador de Eventos* em conjunto com os *sockets* assíncronos encapsulados da implementação *LuaSocket* tradicional; o mapeamento dos tratadores de requisições remotas pelo *Repositório de Tratadores de Eventos* e a execução dos tratadores concretos de eventos, de responsabilidade do próprio escalonador, que permitem a decodificação e implementação do protocolo de comunicação pelos componentes do *OiL*.

No *OiL*, o módulo `Broker` implementa o tratador de eventos concreto para as chamadas remotas de método recebidas pelo escalonador. Esse tratador recebe as informações do cabeçalho da chamada e faz a análise e a requisição dos dados necessários para o despacho do método. Após o processamento, essa mesma rotina retorna o controle para o escalonador e termina sua execução. o diagrama 4.2 ilustra o processamento de uma requisição no *OiL* utilizando o escalonador colaborativo.

Ativação do Escalonador

Enquanto que em um processo “servidor” o escalonador inicia seu processamento através da chamada de método explícita, em um processo “cliente” essa integração deve ser feita de forma transparente para permitir a possibilidade de receber requisições de processamento reentrante mesmo enquanto o cliente está bloqueado aguardando a finalização de um método invocado. Para acessar um objeto remoto, o *middleware* oferece um *proxy* local, onde cada primeiro acesso a um método cria uma função encapsulando as chamadas dos métodos de comunicação com o objeto. Nessa função os *sockets* utilizados na chamada devem ser registrados no *Escalonador* juntamente com os objetos de *callback* implementados pelo cliente. O ciclo de processamento no escalonador é iniciado de forma transparente e terminado juntamente com a finalização do método invocado.

PUC-Rio - Certificação Digital Nº 0321108/CA

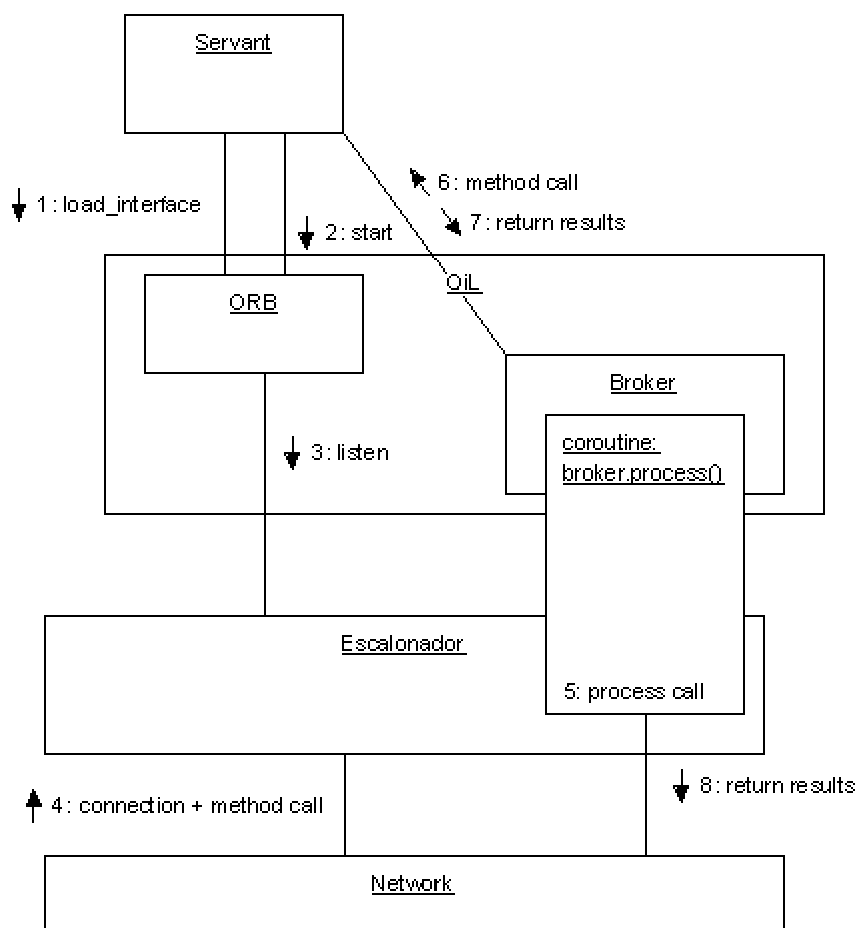

Figura 4.2: Processamento de requisições no *OiL* utilizando o escalonador.

PUC-Rio - Certificação Digital Nº 0321108/CA

Compartilhamento de Conexões CORBA

A implementação original do ORB executava todo o processamento do protocolo e despacho do método para o servidor através de uma única rotina, fazendo com que o *socket* usado para o recebimento de dados ficasse bloqueado para leitura e escrita até que o método solicitado terminasse sua execução, prejudicando o processamento de múltiplas requisições em um mesmo canal de comunicação, cenário observado no reaproveitamento de conexões CORBA[1]. Um cenário onde se observa o reaproveitamento de conexões é ilustrado na figura 4.3.

Neste caso, o método invocado pelo primeiro *proxy* (P1) é invocado em (S1) e delegado para o servidor S2, abrindo o canal de comunicação entre os servidores. Caso esse método entre em espera em S1, o controle seria devolvido ao escalonador, que aguardaria sinalizações do *socket* utilizado na comunicação para retomar o processo de leitura da resposta ao método. Se um outro método fosse enviado pelo mesmo canal de comunicação, para S2, na chegada de uma notificação de recebimento de dados, não haveria meios

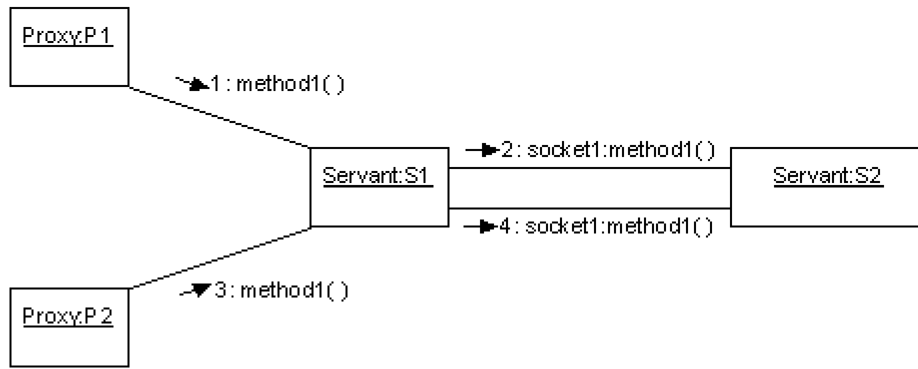

Figura 4.3: Reaproveitamento de conexões CORBA.

de associar a resposta recebida a somente uma das requisições enviadas. A maneira mais simples de solucionar o problema seria bloquear o canal de comunicação até que a resposta ao primeiro método invocado fosse completamente recebida.

Esse gerenciamento de métodos invocados através da reutilização de conexões não poderia ser gerenciado diretamente pelo *Escalonador* sem que este conhecesse o protocolo que está sendo tratado. Para oferecer uma maior flexibilidade no compartilhamento de conexões, foi modelado um *proxy* para o *OiL*, responsável por aguardar respostas às requisições iniciadas através de uma conexão e demultiplexar os valores de retorno de acordo com o *RequestID*. O método `orb.invoke()` foi então dividido em duas partes. A primeira é responsável por enviar o cabeçalho e os parâmetros da requisição, registrar o *socket* e o *RequestID* no *proxy* de conexões e utilizar a segunda parte do método original como uma corrotina associada para o tratamento do par *socket*/*RequestID* registrado no *proxy*.

Ao receber a notificação da camada de rede indicando um novo estabelecimento de conexão, o *Escalonador* cria uma nova corrotina usando o método `broker.process()` como tratador do evento. Os parâmetros são recebidos e uma nova corrotina é criada encapsulando o processamento do método. Essa corrotina é registrada no *proxy* de conexões para permitir que a corrotina original, associada ao canal de comunicação utilizado no recebimento dos dados pudesse finalizar sua execução, liberando o canal para novas requisições.

O registro no *proxy* de conexões é feito associando o *RequestID* correspondente à requisição enviada ao *socket* de comunicação, permitindo a correta demultiplexação dos dados retornados por uma conexão utilizada por diferentes linhas de processamento. Ao receber a resposta de um método, o escalonador encaminha os dados recebidos diretamente para o *proxy* de

PUC-Rio - Certificação Digital Nº 0321108/CA

conexões, que, a partir de uma análise destes dados, recupera o *RequestID* e encaminha para a corrotina associada. Com essa pequena alteração no fluxo de processamento do OiL, é possivel termos um reaproveitamento de conexões CORBA mais flexível.

4.4
Ambiente de Testes e Análise de desempenho

O modelo de escalonamento apresentado foi incorporado na estrutura do *OiL* através de um módulo carregado pelo próprio ORB durante sua inicialização. Esse módulo não altera a API de desenvolvimento original do *OiL* e portanto a integração se mostrou transparente para os desenvolvedores de aplicação.

Para validar o modelo aplicado, iniciou-se uma fase de teste composta por diferentes serviços e arquiteturas de máquinas com o objetivo de analisar o funcionamento e o desempenho da ferramenta completa, composta pelo *OiL* original integrado com o Escalonador proposto. O primeiro ambiente de teste foi organizado de maneira a validar a necessidade de modelos de escalonamento para serviços cuja execução dependa da possibilidade de ganhar-se escala sem impactar o tempo médio de resposta.

Para isso, utilizou-se uma máquina servidora[1] executando um serviço que fornece uma interface com dois métodos disponíveis: o `delegate()` e o `process()`. Enquanto as chamadas ao método `process()` são processados localmente, ao se invocar o método `delegate()`, a execução é delegada para outro servidor, implicando comunicação através da rede e ativando com maior freqüência o ciclo de escalonamento. Essa arquitetura está ilustrada na figura 4.4.

O segundo servidor é executado em outra máquina servidora[1] e o processamento do método delegado será apenas retornar uma *string* pequena. Para os processos clientes, teremos dois possíveis comportamentos: Teremos os clientes que chamarão o método `delegate()` e outros clientes que chamarão o método `process()`. Nesse cenário, o principal objetivo é analisar o desempenho do serviço medindo o tempo médio necessário para se processar 200 chamadas de métodos, variando-se o número de clientes simultâneos utilizados. O serviço foi executado utilizando inicialmente o *OiL* sem suporte ao escalonamento colaborativo de eventos e posteriormente

[1] As máquinas servidoras são micros *Pentium* IV com processador 2.8GHz e 1Gb de memória RAM executando sistema operacional *SuSE* 9.1 com *kernel* 2.6. O sistema está montado em uma rede local de 100 Mbits

PUC-Rio - Certificação Digital Nº 0321108/CA

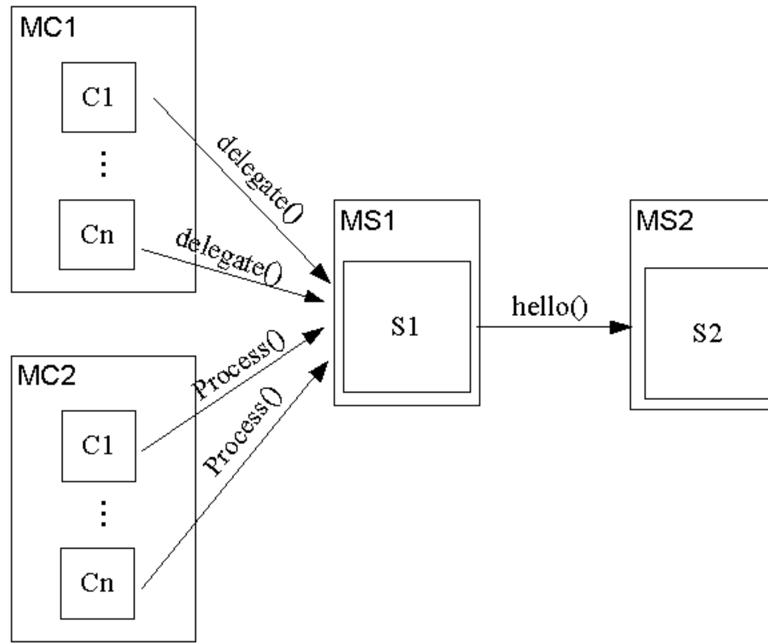

Figura 4.4: Arquitetura do caso de teste 1.

PUC-Rio - Certificação Digital Nº 0321108/CA

habilitando o módulo do escalonador. O resultado obtido é mostrado na figura 4.5.

Através da análise desse gráfico pode-se perceber que até mesmo para um pequeno número de clientes simultâneos, o tempo de resposta do serviço executando na ferramenta sem o módulo de escalonamento é maior do que o obtido com a inserção das rotinas para processamento colaborativo das requisições. Conforme o número de clientes simultaneamente ativos aumenta, o tempo de resposta no modelo sem escalonamento fica maior, indicando claramente a impossibilidade de tratar as requisições simultaneamente resultando em degradação do serviço executado. No modelo com escalonador, o aumento do número de clientes ativos significa um aumento do volume de chamadas, fazendo com que mais requisições sejam processadas e assim reduzindo o tempo médio de processamento por chamada.

O segundo caso de teste foi montado com o objetivo de testar o desempenho do *OiL* em comparação com outros ORBs em uma arquitetura um pouco mais complexa. Nesse ambiente, foram utilizadas no total 11 máquinas[2] organizadas como mostra a figura 4.6. Em cada uma das máquinas clientes, executou-se um total de 100 clientes, cuja implementação consistia de um ciclo infinito de chamadas a um único método remoto. Es-

[2]As máquinas MCx utilizadas como clientes na arquitetura são micros *Athlon XP* 2600+ com 256Mb de memória RAM executando sistema operacional Windows XP. As máquinas do *proxy* seguem a configuração descrita em [1]. O sistema está montado em uma rede local de 100 Mbits

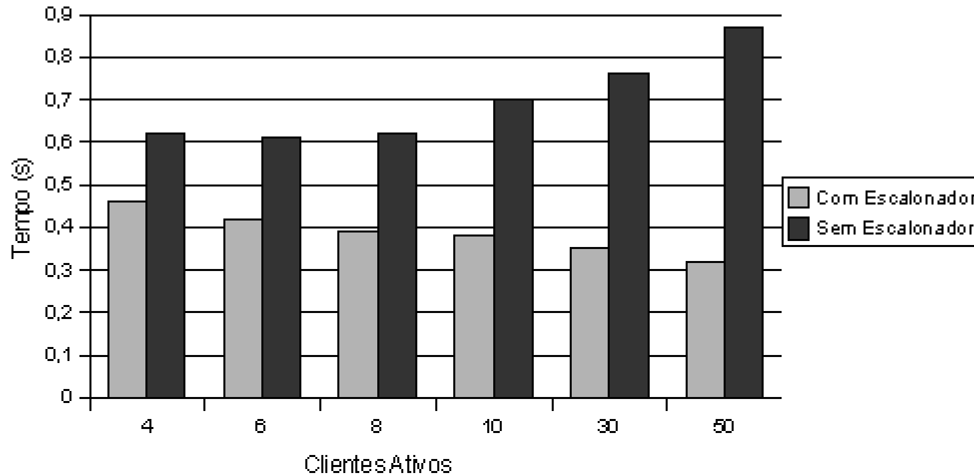

Figura 4.5: Tempo de resposta médio por 200 chamadas de método.

sas chamadas eram feitas diretamente a um *proxy* que se encarregava de distribuir a carga os 5 servidores disponíveis para o tratamento do método. Dessa maneira, configura-se um cenário onde o gargalo de comunicação e processamento se concentra em um elemento central (*proxy*) de onde pode-se extrair medidas de desempenho, como o número de métodos processados e a quantidade de memória utilizada durante a execução dos testes nessa arquitetura.

Nesse teste utilizou-se uma única implementação para os servidores e para os clientes, utilizando o ORB OiL. Para analisar o desempenho do *proxy*, implementou-se uma versão utilizando o *OiL*, e uma versão equivalente em C++, utilizando o ORBACUS 4.3.0[25] e variando-se o modelo de concorrência utilizado pelo ORBACUS.

Os modelos de concorrência [26] descrevem a maneira com que o ORB deverá tratar a comunicação e execução das requisições. Nesse teste, o ORBACUS foi executado utilizando três diferentes modelos, cobrindo a execução em ambientes *single-threaded* e *multi-threaded*:

– Reactive - O modelo Reativo faz com que servidores utilizem chamadas ao método `select()` de maneira a simultaneamente aceitar requisições de conexão, chamadas de métodos provenientes de múltiplos clientes e devolver as respostas. O modelo Reativo é executado em uma única frente de execução, o que torna esse ambiente bastante próximo do modelo de escalonamento proposto para o *OiL*.

– Thread per Client - O modelo de *thread-per-client* executa um ambiente onde cada novo cliente que se conecta ao servidor dispara o

PUC-Rio - Certificação Digital Nº 0321108/CA

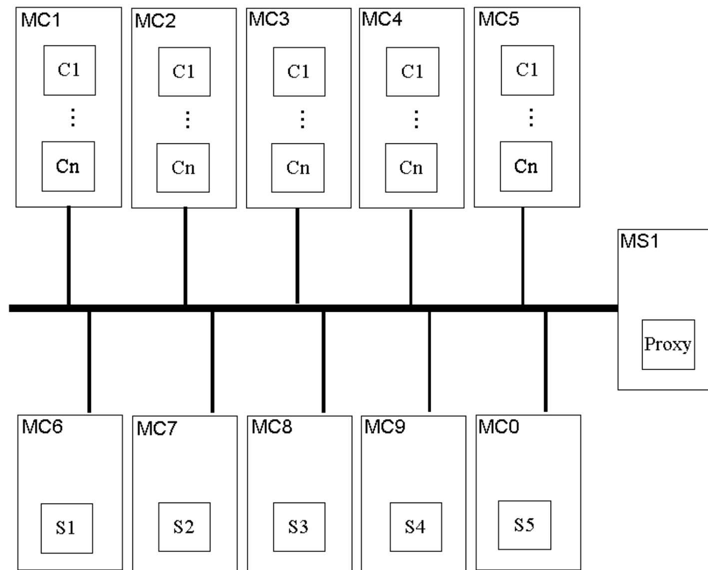

Figura 4.6: Arquitetura do caso de teste 2.

PUC-Rio - Certificação Digital Nº 0321108/CA

início de uma nova linha de execução. Esse modelo consegue aliar a capacidade de processar requisições de maneira concorrente com um baixo custo de criação de *threads*, visto que estes são criados somente no momento de novas conexões.

– Thread per Request - Se o modelo de concorrência utilizado for o de *thread-per-request*, o ORB deverá criar uma nova linha de execução para cada requisição a ser processada. Esse modelo permite que as chamadas de métodos possam sempre ser executadas instantaneamente, porém traz consigo um alto custo de processamento e maior uso de memória devido ao grande volume de *threads* criados e destruídos em seqüência.

O teste se inicia com a execução do *proxy*, que publica seu IOR em um sistema de arquivos acessível por todas as outras máquinas da arquitetura. Em seguida iniciam-se os servidores, que se registram diretamente no *proxy* através do método `registerServant()`. Durante o teste, os clientes são iniciados em blocos de 100, em intervalos de 150 segundos, permitindo uma análise da evolução do uso de memória e do tempo necessário para o repasse dos métodos. Cada cliente fica em um ciclo infinito fazendo chamadas ao método `hello()` do *proxy*, que por sua vez utiliza um política de fila circular para decidir qual o servidor que irá processar as chamadas de métodos recebidas. A memória utilizada por cada um dos processos foi medida em intervalos regulares de cinco segundos diretamente do sistema operacional utilizando as informações contidas no arquivo `/proc/(PROCESSID)/status`.

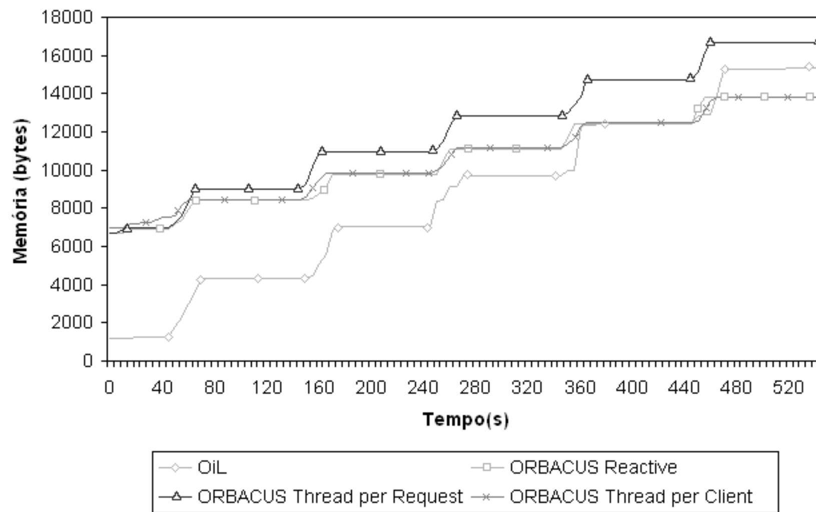

Figura 4.7: Uso de memória.

O gráfico da figura 4.7 mostra os valores coletados para as 4 implementações de *proxys* testados. Neste gráfico, os patamares horizontais de memória representam os valores estabilizados após a entrada de um novo lote de clientes. Observa-se que o consumo de memória inicial do *OiL* é muito inferior ao consumo médio inicial do ORBACUS porém, quando executa-se a primeira bateria de clientes, percebe-se um aumento de 3028 *Kbytes* no uso de memória no *OiL* (embora grande parte disso esteja relacionada à reserva do coletor de lixo), Esse aumento foi maior que o aumento médio de 1530 *Kbytes* observado no ORBACUS, o que faz com que ao final dos 500 clientes o consumo de memória do *OiL* medido pelo sistema operacional supere em 550 *Kbytes* ou 4% o consumo médio do ORBACUS.

Através de uma análise mais aprofundada do uso de memória do *OiL*, percebe-se que o valor de memória reservado no sistema operacional representa em média o dobro do valor fixo utilizado pelo *middleware*. Durante a execução dos testes, observamos que o valor de uso real de memória obtido através do método `gcinfo()` oscilava entre os ciclos de coleta de lixo executados pela máquina virtual Lua devido às estruturas criadas no processo de recebimento e tratamento das requisições que perdem todas as suas referências ao término da chamada. Em ambientes onde existam maiores restrições de memória esse consumo pode ser ajustado através dos métodos de coleta de lixo oferecidos pela própria linguagem *Lua*, fazendo com que o total de memória reservado no sistema operacional seja menor.

Mediu-se também o tempo médio necessário para o *proxy* processar 100 chamadas de método. Nesse teste não observou-se variação no tempo de

PUC-Rio - Certificação Digital Nº 0321108/CA

resposta durante a execução das baterias de clientes, mantendo-se constante em um mesmo ORB. Os tempos de resposta de cada ORB (e seus diferentes modelos de concorrência) estão apresentados na figura 4.8.

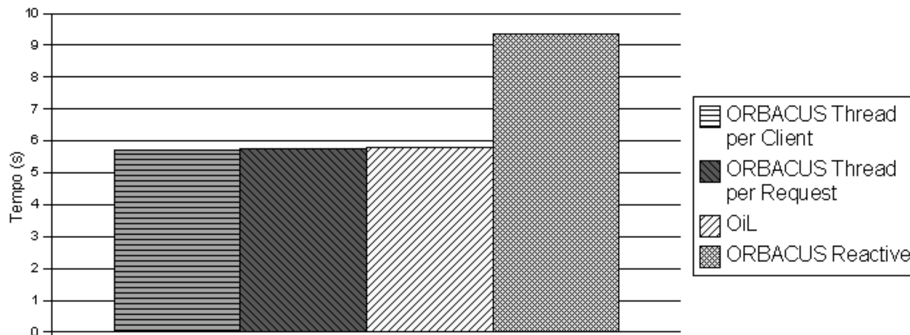

Figura 4.8: Resultados do caso de teste 2.

Pela análise do gráfico apresentado na figura 4.8, observa-se que o teste executado utilizando um Proxy com o ORBACUS em modo reativo apresentou um tempo médio de 9.35 segundos para processar as 100 requisições contra um tempo médio de 5.79 segundos usando o modelo de escalonador proposto no OiL. Essa grande diferença de desempenho pode ser explicada pelo fato do modelo reativo o ORBACUS não permitir que outras requisições sejam processadas enquanto o Proxy aguarda a resposta de alguma outra requisição.

PUC-Rio - Certificação Digital Nº 0321108/CA

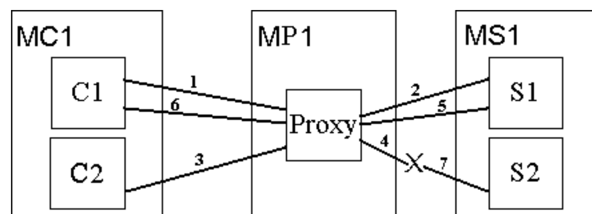

Figura 4.9: Exemplo ilustrativo para análise de desempenho do modelo Reativo.

A figura 4.9 mostra um sub-conjunto da estrutura montada no caso de teste 2 e pode ser usada para ilustrar melhor o problema. Nesse diagrama, o Proxy recebe uma chamada de método do Cliente 1 e repassa para o primeiro Servant. Em seguida o Proxy recebe uma chamada de método do Cliente 2, porém essa chamada não é repassada até o recebimento da resposta do primeiro Servant. Esse atraso, em um modelo de maior escala como o usado no caso de teste 2, é o responsável pelo baixo desempenho do modelo reativo do ORBACUS.

Esse comportamento não é observado nos modelos *multithread*, onde obteve-se um tempo médio de 5.73 segundos para o mesmo processamento. Os resultados desse teste mostram que o *OiL* com as rotinas de escalonamento apresenta um desempenho dentro dos padrões observados em ORBs C++ comerciais *multithreaded*, mantendo em níveis aceitáveis o uso de memória, mesmo com a configuração padrão de coleta de lixo utilizada pela máquina virtual Lua.

PUC-Rio - Certificação Digital Nº 0321108/CA