# 2
# Trabalhos Relacionados

Este capítulo é dedicado à análise de trabalhos relacionados ao tema desta dissertação, que foram escolhidos por apresentarem mecanismos de monitoração de sistemas distribuídos, ou por utilizarem técnicas que foram empregadas nesta dissertação. A análise desses trabalhos ajudou a identificar aspectos importantes para monitoração de sistemas distribuídos, servindo de base para esta dissertação e para a implementação do sistema de monitoração.



## 2.1
## LuaMonitor

Em [1] é apresentado um ambiente que oferece suporte para o desenvolvimento de aplicações distribuídas auto-adaptativas. As ferramentas criadas utilizam LuaCorba que é responsável pela integração entre CORBA e a linguagem Lua [15]. Esse trabalho possui três objetivos principais:

- Encontrar dinamicamente componentes que melhor se encaixem aos requisitos de qualidade de serviço (QoS) da aplicação, onde esses requisitos são expressos como atributos não funcionais do componente.
- Monitorar as propriedades associadas a esses requisitos, de forma que se possa garantir o atendimento desses requisitos durante o ciclo de vida da aplicação.
- Reagir a mudanças dessas propriedades ativando as estratégias de adaptação adequadas.

Para realizar as adaptações dinâmicas de forma automática foi usado o conceito de *smart proxy*. Nessa abordagem o cliente não se conecta diretamente ao servidor e sim ao *smart proxy*. Ele mantém um registro de serviços oferecidos

por diversos componentes e oferece esses serviços realizando o tratamento dos requisitos não funcionais demandados pela aplicação. O *smart proxy* possui estratégias para seleção dinâmica do componente, monitoração e adaptação, permitindo que o código da aplicação fique voltado somente para os requisitos funcionais. O *smart proxy* pode utilizar diferentes componentes durante a execução da aplicação, com o objetivo de atender aos requisitos exigidos (Figura 2.1).

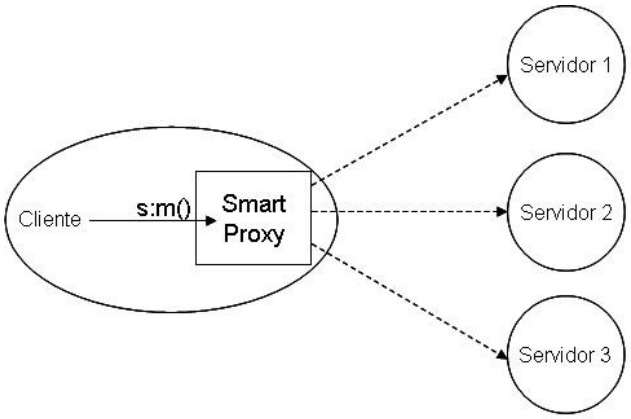


Figura 2.1: *Smart Proxy* [1]

Para implementar o serviço de localização dinâmica dos componentes o *smart proxy* utiliza o serviço de *trading*[7] oferecido pela arquitetura CORBA. Utilizando o serviço de *trading* o usuário pode obter informações dinâmicas sobre os serviços oferecidos e escolher o mais adequado de acordo com suas características ou propriedades não funcionais. Para garantir que o serviço de *trading* possui informações atualizadas sobre os serviços disponíveis existe o *service agent* que é o responsável por anunciar as ofertas de serviços ao *trader*.

Além de garantir a localização de serviços de acordo com os requisitos desejados, o *smart proxy* deve garantir que os serviços utilizados estejam cumprindo os requisitos durante a execução da aplicação. Para coletar essas informações relativas ao estado dos recursos, garantindo a qualidade do serviço prestado, foram implementados os *LuaMonitors*.

Como as características que devem ser monitoradas podem variar de acordo com o sistema, os *LuaMonitors* são monitores genéricos, capazes de fornecer ao administrador da aplicação uma forma dinâmica para especificação de quais e como serão as métricas coletadas. Dessa forma, ele é preparado para receber *scripts* de configuração que irão conter a forma de coletar as métricas desejadas. De acordo com o valor dessas métricas, ele irá gerar avisos sobre a ocorrência de uma determinada situação.

Para promover adaptações de acordo com as métricas coletadas, os *smart proxies* podem ser configurados dinamicamente e realizam as estratégias definidas de acordo com as métricas coletadas, fazendo com que os componentes se adaptem automaticamente a variações no ambiente onde estão executando. Para obter as informações do ambiente de execução, os *smart proxies* se registram nos *LuaMonitors* e assim passam a receber mensagens com as informações que foram coletadas. A arquitetura básica proposta pelo artigo está apresentada na Figura 2.2.

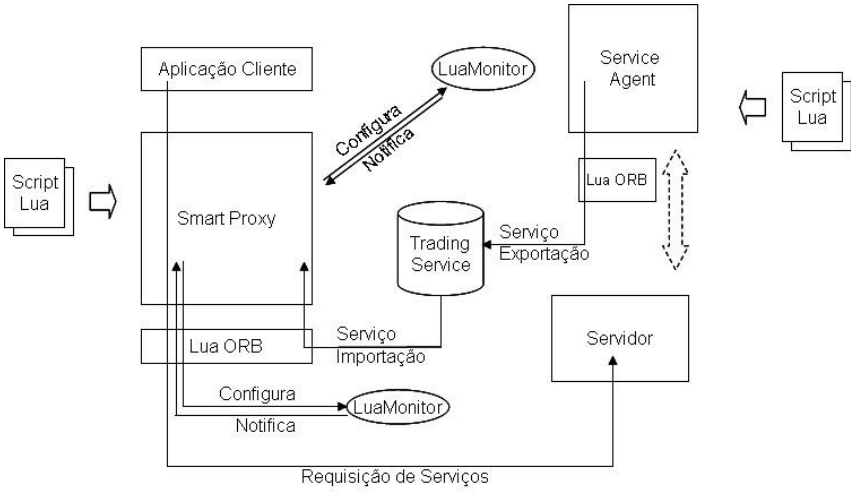


Figura 2.2: Arquitetura para Adaptação Dinâmica de Aplicações Distribuídas [1]

O usuário pode ter interesse em manipular as informações imediatamente após serem coletadas com o objetivo de gerar valores mais elaborados, como estatísticas, ou mesmo atribuir semântica aos resultados obtidos. Para isso a arquitetura oferece uma forma do usuário associar aspectos aos coletores, que definirão o tratamento que deve ser realizado com o valor que foi obtido. O código Lua da Figura 2.1 mostra um exemplo de função que será atrelada a um coletor de quantidade de memória livre da máquina. Ele verifica se existe uma tendência de crescimento no uso desse recurso.

```
1
2  lmon:defineAspect("Increasing",
3    [[function(self,currval)
4      if currval[1] > currval[2] then
5        return "yes"
6      else
7        return "no"
8      end
9    end]])
10   return lmon
11 end
```

Listagem 2.1: Pseudo-Código do Aspecto de Tendência de Uso

## 2.2
## Chung e Chang

Em [2] é apresentado um mecanismo de monitoração de recursos em grades computacionais, denominado *Grid Resource Information Monitoring* (GRIM) que é um mecanismo para obtenção de informações dos diversos nós da grade baseado no modelo *push* para publicação das informações.

O GRIM é dividido em três camadas principais (figura 2.3): Consulta, Mediação e Provedora de Informações. A primeira é onde estão localizados os usuários, que desejam obter informações sobre o estado de determinados recursos da grade. A segunda camada consiste em um conjunto de máquinas que armazenam os valores atualizados dos recursos que estão sendo monitorados. A terceira é formada pelas máquinas que de fato compõem a grade, onde os coletores de informações sobre os recursos são implantados.

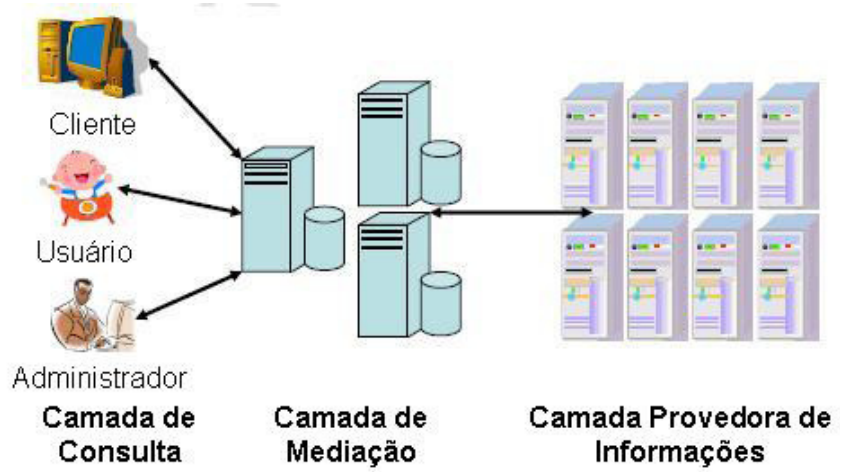


Figura 2.3: Camadas do GRIM [2]

O uso dos mediadores é feito para evitar que somente no momento da requisição das informações, pelo cliente, os valores do estado dos recursos sejam coletados. Se isso ocorresse, tanto o usuário poderia ter que esperar mais para ter sua resposta, como as máquinas do cluster poderiam ter uma grande sobrecarga com um elevado número de pedidos.

Com o uso dos mediadores, o desafio passa a ser descobrir com qual frequência as máquinas do *cluster* devem informar as mudanças de estado ocorridas aos mediadores. Um intervalo muito pequeno pode onerar a rede com o excesso de mensagens trocadas, e intervalos grandes podem prejudicar a acurácia dos valores coletados. Para lidar com esses problemas é proposto o *Grid Resource Information Retrieving* (GRIR), que é um protocolo para

publicação dos dados coletados que procura identificar dinamicamente quando o valor necessita ser anunciado ao mediador. Três abordagens são apresentadas para esse protocolo: sensíveis a variação de tempo, sensíveis a variação de valor e híbrida. Nesta dissertação implementamos o uso dessas políticas como mecanismo de publicação dos valores coletados, logo, maiores detalhes sobre essas abordagens estão descritas na Seção 5.4.4.

Os autores também apresentam três métricas que podem ser usadas para medir o desempenho das políticas propostas: taxa de atualização (frequência com que são geradas mensagens para os mediadores), taxa de atualizações perdidas (frequência que mudanças nos valores coletados não foram informadas), porcentagem de informações redundantes (porcentagem de valores idênticos que foram informados em sequência). Com base nessas três métricas, eles avaliam o uso da política híbrida proposta contra outros trabalhos que lidam com o mesmo tema. Em todos os testes feitos a abordagem híbrida leva vantagem sobre as apresentadas nos demais trabalhos.



## 2.3 Debusmann et al

Em [3] é apresentada uma arquitetura de monitoração de aplicações baseadas em CORBA utilizando uma instrumentação genérica. Ele apresenta um projeto chamado *AppMan* que se concentra em aspectos como: gerenciamento de desempenho, gerenciamento de eventos e falhas, e gerenciamento de configurações. Esse projeto decompõe a aplicação que está sendo monitorada em quatro camadas: aplicação, CORBA, sistema e rede. Cada uma delas possui uma forma específica de ser monitorada. Todas as informações coletadas são expostas através do uso de um serviço de eventos. A arquitetura ainda contempla agentes para realizar a manipulação dos dados coletados, um gerente para realização de automações, e um serviço encarregado de manter informações sobre a topologia da aplicação que está sendo monitorada. A figura 2.4 apresenta a arquitetura do *AppMan*. O artigo [3] é focado nas áreas cinza escuro da figura.

O trabalho divide as métricas coletadas em dois grupos: simples e complexas. As simples são aquelas coletadas pontualmente pelos interceptadores como o número de chamadas realizadas a um determinado método, as complexas são o resultado da coleta em diferentes pontos da requisição. Para realizar a obtenção das métricas complexas é utilizado uma API chamada Ap-

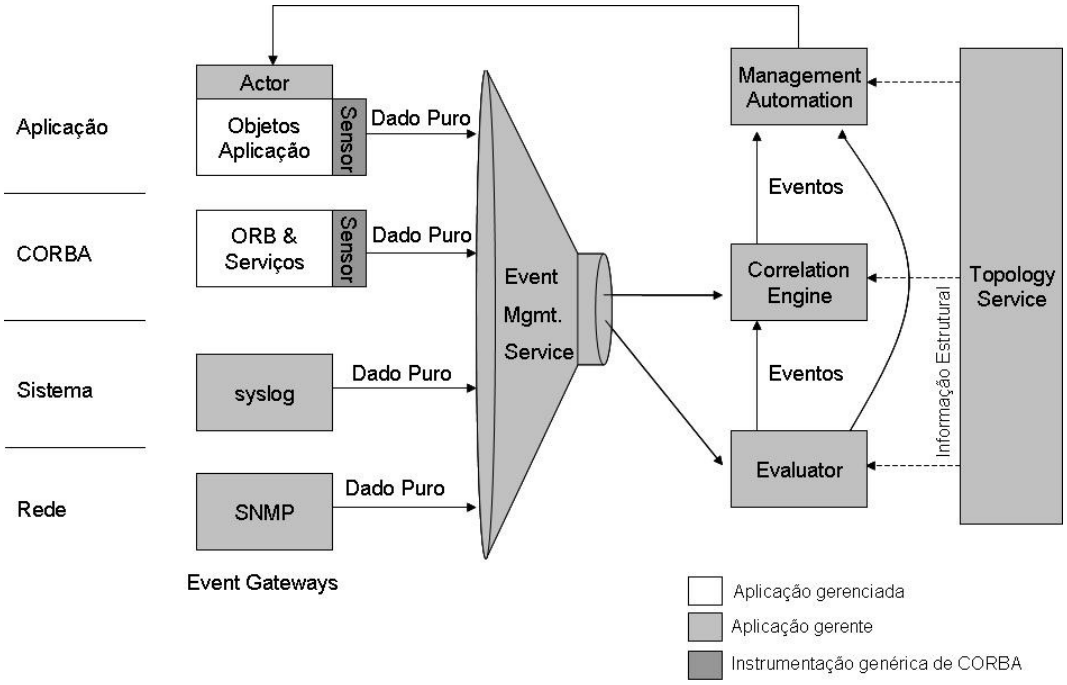


Figura 2.4: Arquitetura do AppMan [3]



plication Response Measurement (ARM) [16] que oferece funções para coleta de informações.

No cliente, antes da chamada ser enviada para o servidor ela é interceptada. Nesse ponto é feita uma chamada ao método *arm_start* na biblioteca do ARM. Essa função retorna um objeto com as informações obtidas pela biblioteca. Esse objeto é inserido na requisição e enviado para o servidor. Antes da requisição ser tratada ela é enviada para o interceptador do servidor, que extrai o objeto que foi inserido no cliente e executa novamente o método *arm_start*. Em seguida a requisição é tratada e antes dela ser retornada ao cliente é novamente interceptada no servidor, que executa o método *arm_stop* e coloca o objeto resultante na resposta que será enviada para o cliente. Finalmente o cliente ao receber a resposta, extrai o objeto ARM, e executa o método *arm_stop*. A biblioteca ARM é preparada para coletar informações aninhadas, logo, ao final da execução da requisição é possível inspecionar o objeto ARM que transitou junto com a requisição e obter informações como o tempo total de espera do cliente e do servidor para requisições bem e mal sucedidas. A figura 2.5 mostra o fluxo da chamada realizada e as funções dos interceptadores responsáveis por tratá-las em cada ponto a requisição.

O trabalho realiza testes para mensurar a sobrecarga imposta pela infraestrutura de coleta utilizando os interceptadores e a biblioteca ARM. No exemplo usado foi montada uma aplicação em CORBA com um cliente e um

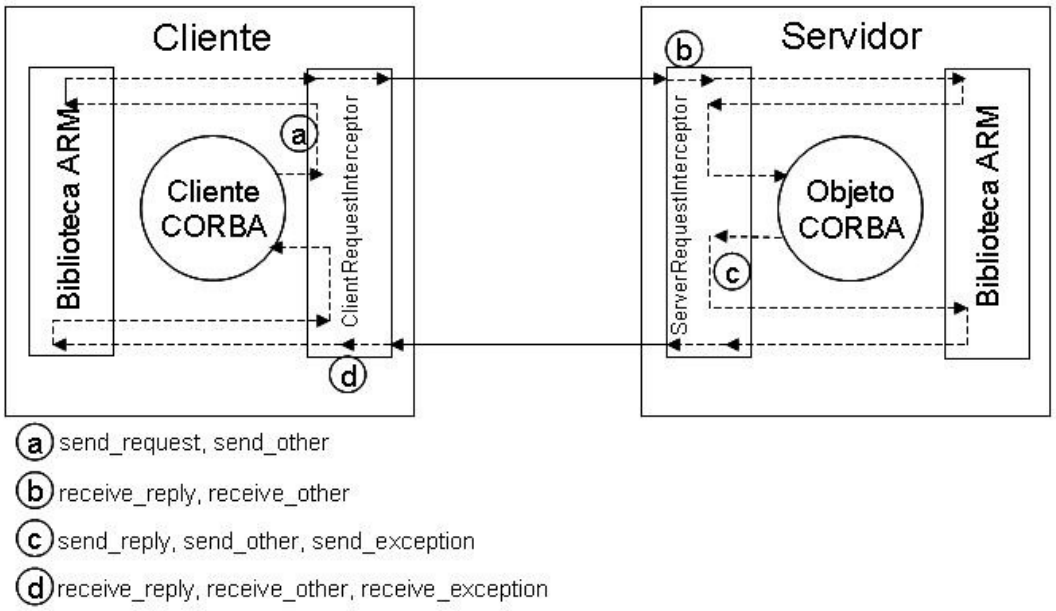


Figura 2.5: Pontos de Interceptação Utilizando ARM [3]



servidor, foram medidos os tempos de execução para diferentes configurações, dentre elas: sem nenhuma infraestrutura de monitoração, com o uso de interceptadores vazios, coletando somente as métricas simples, utilizando somente o ARM e finalmente com toda a infraestrutura. Nesse último caso a sobrecarga imposta ao sistema chegou a 643,29%.

## 2.4 Marchetti et al

Em [4] é feito um estudo sobre o uso de interceptadores em três implementações de CORBA para linguagem Java: ORBACUS [17], JACORB [18] e ORBIX [19]. Esse artigo foi de grande valia para suprir a falta de informações sobre o uso de interceptadores em Java. Em [4] são implementados três serviços que fazem uso de interceptadores, são eles: redirecionamento, *piggybacking* e *proxy*. O trabalho desenvolvido nesta dissertação utilizou amplamente um recurso definido na arquitetura CORBA conhecido como interceptador, Seção 5.1. No entanto as implementações de CORBA disponíveis oferecem poucas informações [18] sobre o uso desses recursos.

O primeiro tipo é utilizado para redirecionar, no objeto cliente, o destino da requisição, fazendo com que um novo objeto seja o alvo da chamada. Interceptadores CORBA permitem que o usuário defina se este redirecionamento

ocorrerá de forma permanente ou não, levando em consideração as próximas chamadas que serão realizadas àquele método. As figuras 2.6 e 2.7 ilustram como esses dois tipos de redirecionamentos ocorrem. Na figura 2.6, o cliente deseja invocar um método no *server2*, no entanto a requisição é interceptada e redirecionada para o *server1*. Como nesse caso está sendo utilizado um redirecionamento permanente as próximas requisições são enviadas diretamente para o *server1*.

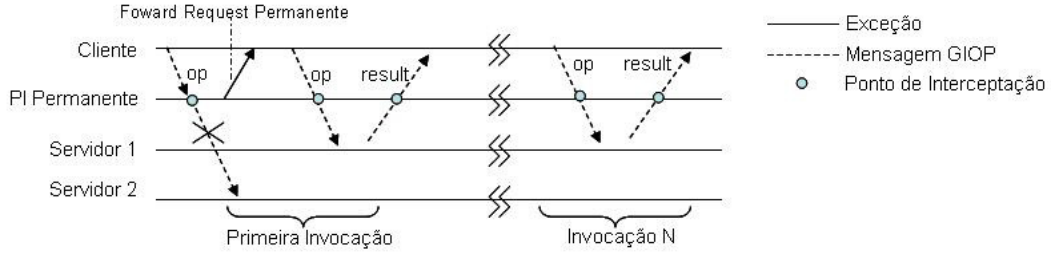


Figura 2.6: Redirecionamento Permanente [4]

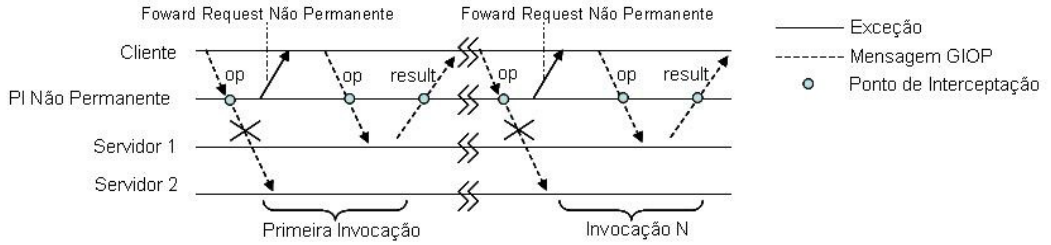


Figura 2.7: Redirecionamento Não Permanente [4]

O segundo tipo consegue inserir informações, a partir do interceptador, na requisição que será enviada para o servidor. Dessa forma, essas informações poderão ser obtidas no contexto dos interceptadores do objeto servidor. A figura 2.8 ilustra um cliente adicionando algumas informações na requisição e o servidor recuperando as mesmas. O objeto com formato quadrado da figura representa a informação que é trocada entre os interceptadores do cliente e do servidor.

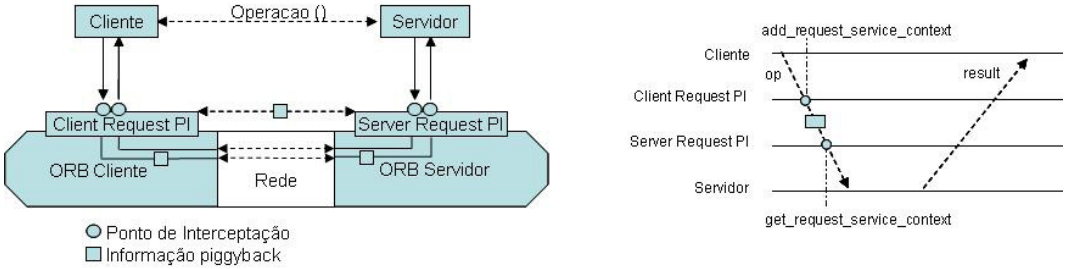


Figura 2.8: Piggbacking [4]

O terceiro tipo foi criado como uma alternativa para uma limitação dos interceptadores, que não conseguem obter os argumentos que estão sendo passados na requisição. Para contornar essa dificuldade foi criado um *proxy* que é invocado a cada chamada de um servidor utilizando o mecanismo de redirecionamento, assim dentro do *proxy* pode ser feita a inspeção dos argumentos. Existem duas formas de implementar esse *proxy*. Em uma delas, o *proxy* executa no mesmo espaço de endereçamento do objeto cliente (*Collocated*). E na outra forma executa em um espaço de endereçamento diferente. A figura 2.9 mostra o caminho percorrido pela requisição em cada um desses casos.

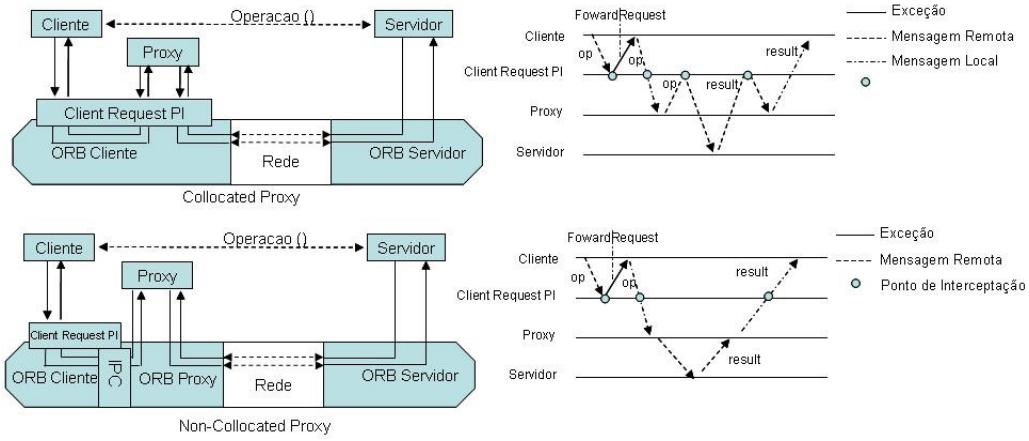


Figura 2.9: Uso de *Proxy* [4]

São realizados testes das técnicas sugeridas implementadas em diferentes ORBs. Durante a execução desses testes, *bugs* foram encontrados no uso do JACORB para redirecionamento não permanente, e no uso do ORBIX com *proxy* de redirecionamento permanente. No primeiro caso a sobrecarga imposta pelo redirecionamento chega a 780%, e no segundo caso a aplicação entra em um *looping* infinito. Esses tipos de *bug* em implementações amplamente utilizadas, como o caso das analisadas aqui, mostram como esse recurso ainda é pouco explorado.

## 2.5 Considerações Finais

Os trabalhos apresentados nesse capítulo possuem características distintas. Dentre elas, destacam-se o uso de objetos monitores para coleta de informações, descrição de políticas para publicação de informações, uso de interceptadores CORBA para coleta de dados, e dificuldades e soluções para o uso de interceptadores.

Esses trabalhos influenciaram a construção dos mecanismos que são descritos nesta dissertação. Porém, nesses artigos não encontramos uma única solução que faça a monitoração dos nós participantes da aplicação e ao mesmo tempo se preocupe com as interações entre os diversos objetos remotos que compõem a aplicação, e que também forneça um mecanismo robusto para publicação das informações obtidas.

Em nosso trabalho não lidamos com algumas questões abordadas nesses artigos, como auto adaptação. Porém, construímos uma arquitetura que tanto monitora a carga das máquinas participantes da aplicação, como monitora as interações entre os diversos objetos remotos participantes. Também oferecemos um ambiente baseado em eventos para publicação das informações que pode ser adaptado por políticas definidas pelo usuário. Com isso pretendemos obter uma solução de monitoração robusta e flexível que poderá ser usada por outros trabalhos.