5
Coleta e Publicação de Métricas

Para fornecer as informações necessárias para a monitoração de um sistema de componentes distribuídos, precisamos planejar como extrair essas informações. Ao acrescentar um mecanismo de coleta de dados, tipicamente é introduzida uma sobrecarga de processamento na aplicação. Logo, nosso maior cuidado será em obter e disponibilizar essas métricas coletadas de maneira eficiente e da forma mais transparente possível para o usuário da aplicação.

Primeiramente é preciso definir quais informações que irão auxiliar o desenvolvedor a monitorar o ambiente de execução da aplicação. As informações que devem ser coletadas são muito dependentes do escopo para o qual o sistema foi criado. Como neste trabalho lidamos com aplicações distribuídas baseadas em componentes de software, tentamos escolher algumas métricas que serão interessantes para monitoração desse tipo de sistema.

Durante o desenvolvimento deste trabalho lidamos com o uso de duas linguagens distintas Lua e Java. Nessa dissertação utilizamos a linguagem Java como principal para o desenvolvimento dos testes, no entanto a linguagem Lua é utilizada em diversos exemplos e durante o texto iremos abordar características dessa linguagem importantes para implementação da arquitetura proposta.

O mecanismo de interceptadores oferecido pela arquitetura CORBA [4] pode ser de grande valia para coleta de informações. Na seção 5.1 temos uma explicação sobre o uso de interceptadores, e também características específicas encontradas no seu uso nas linguagens Java e Lua [15]. Em seguida, na seção 5.2 apresentamos algumas métricas que serão coletadas pela arquitetura. Na seção 5.3 mostramos os detalhes da instrumentação do SCS. Finalmente na seção 5.4 temos a descrição do mecanismo de publicação criado e algumas ferramentas disponibilizadas para facilitar o acesso às informações obtidas.

5.1 Interceptadores

A arquitetura CORBA oferece um mecanismo, chamado interceptador, capaz de desviar o fluxo normal de uma requisição, fazendo com que ela passe por determinados métodos antes de ser efetivamente entregue para o objeto alvo. Existem diversos pontos de uma requisição onde podem ocorrer esses desvios, de modo que podemos realizar medições em diferentes etapas da requisição, tanto no lado do cliente como no lado do servidor.

Um aspecto interessante no uso de interceptadores é a transparência que eles oferecem ao desenvolvedor do componente. Dessa forma é possível executar um componente com ou sem interceptador, sem nenhuma alteração no seu código. Porém, o uso de interceptadores pode onerar o desempenho da aplicação, pois incluímos código para ser executado a cada operação que é invocada. Por isso, é de extrema importância que o código executado pelo objeto interceptador seja o mais simples possível evitando assim grandes perdas de desempenho.

A figura 5.1 mostra os intervalos de medição que podemos obter com as interceptações. No exemplo da figura temos um cliente realizando uma chamada a um objeto servidor, que por sua vez irá realizar outras duas chamadas, agora como cliente, a outros componentes. Observando a figura podemos extrair as seguintes medições:

- a: Tempo de resposta do servidor 1, que inclui o tempo de execução no servidor 1 mais o tempo de rede;
- b: Tempo de resposta do servidor 2, que inclui o tempo de execução no servidor 2 mais o tempo de rede;
- c: Tempo de execução do servidor 1, que inclui o tempo de execução no servidor 2 e 3 mais o tempo de rede;
- d: Tempo de execução do servidor 2;
- e: Tempo de execução do servidor 3;
- f: Tempo de resposta do servidor 3, que inclui o tempo de execução no servidor 3 mais o tempo de rede.

Outra medida que podemos obter seria o tempo de rede gasto durante cada chamada. Para obtê-lo podemos subtrair D de B que teremos o tempo gasto com troca de mensagens entre o nó do servidor 1 e o nó do servidor 2.

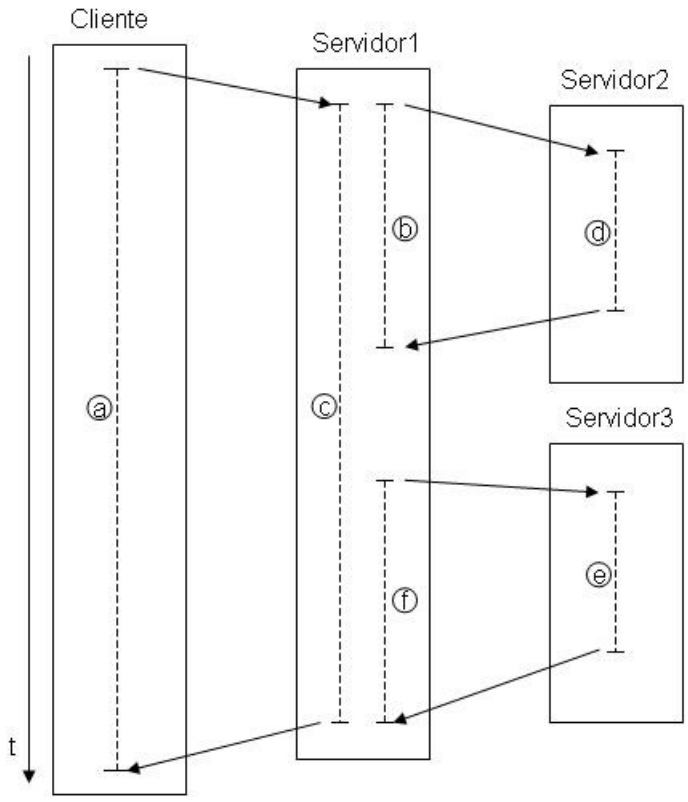

Figura 5.1: Medidas Retiradas com Interceptadores

Essa operação pode ser repetida com as demais informações coletadas, obtendo os tempos gastos nas diversas comunicações realizadas.

Nesta dissertação utilizaremos os interceptadores para coletar informações referentes às trocas de mensagens entre os componentes. Como o trabalho desenvolvido realizou implementações em duas linguagens, Lua e Java, temos alguns resultados sobre o uso desse tipo de recurso nessas duas tecnologias.

5.1.1 Detalhes de Interceptadores Java e Lua

Em Java os interceptadores devem ser definidos antes da execução do processo, ou seja, quando inicializamos o processo devemos passar as informações dos interceptadores que serão utilizados. Isso impossibilita a retirada ou inclusão desses interceptadores durante a execução do processo, após o ORB ter sido inicializado. Dessa forma não podemos remover completamente e de forma dinâmica o interceptador. O que podemos fazer é colocar um condicionante no código dos métodos interceptadores para que esses não realizem a

coleta efetiva das métricas. Assim, todo ponto de interceptação foi inicializado com um teste onde é verificado se a coleta deve ser realizada ou não.

Em cada uma das funções de interceptação são oferecidos objetos das classes *ClientRequestInfo* e *ServerRequestInfo*, que fornecem informações sobre aquela chamada. Dentre as principais funções oferecidas por esses objetos destacamos:

- target_most_derived_interface(): Obtém a interface que implementa o método que está sendo invocado;
- operation(): Obtém o nome do método que está sendo invocado;
- add_request_service_context() , get_request_service_context(), add_reply_service_context(), get_reply_service_context(): Permite adicionar e obter informações na requisição, com isso podemos colocar informações coletadas no cliente e enviá-las junto com a requisição para o servidor, e vice-versa;
- arguments(): Obtém as lista dos argumentos que foram passados para o método.

A especificação de interceptadores impõem algumas limitações no uso dessas funções. No mapeamento Java dos *PortableInterceptors* os parâmetros utilizados na requisição não estão disponíveis para aplicações que não utilizem invocação dinâmica (DSI/DII)[7].Ou seja, uma chamada ao método *arguments()* resultará na exceção *NO_RESOURCES*. Isso porque nos objetos clientes o *marshaling* dos parâmetros que serão passados é feito antes do interceptador do cliente receber a informação. Já no caso do servidor acontece o oposto, o *unmarshaling* dos parâmetros só é executado após a passagem da requisição pelo interceptador. A operação de *unmarshaling* só poderia ser realizada com objetos auxiliares que não estão disponíveis no contexto de execução dos interceptadores [25][26]. Dessa forma, nos pontos de interceptação não conseguimos as informações sobre os argumentos que estão sendo passados. O artigo [4], apresentado na seção 2.4, apresenta uma solução para esse tipo de problema.

Como nesta dissertação não estamos interessados em acessar os valores passados entre as chamadas remotas, não foi implementado nenhum recurso para contornar essa limitação. Porém em uma possível extensão desse trabalho esses mecanismos poderão ser utilizados.

Em Lua as funções e pontos de interceptação são os mesmos que os apresentados em Java. As diferenças ficam na parte de instanciação dos elementos que interceptarão essas chamadas. Nesse caso, basta implementarmos as funções de interceptação e utilizarmos as funções *setclientinterceptor* e *setserverinterceptor*, definidas pelo *OiL (ORB In Lua)* [23] para indicarmos os objetos que deverão interceptar as requisições do cliente e do servidor.

Desse modo podemos em qualquer ponto do código adicionar ou remover o interceptador sem necessidade de parar o processo. Essa característica será uma vantagem dessa implementação, pois, em Java, se solicitarmos o encerramento da coleta de informações não podemos retirar o interceptador por completo do processo, como vimos anteriormente.

Em contra-partida o OiL ainda não está preparado para receber mais de um interceptador no cliente nem no servidor, como neste trabalho não precisaremos desse recurso, isso não representou um entrave para o desenvolvimento do projeto. Porém no caso de termos uma aplicação utilizando interceptadores, isso poderá gerar um problema no uso do OiL. A tese [27] apresenta um gerenciador de interceptadores implementado para o OiL que busca resolver essa limitação.

5.2 Descrição de Métricas

As métricas escolhidas obtêm informações sobre o tempo gasto na comunicação entre os componentes, facilitando a identificação de gargalos de rede. Também obtém dados sobre o uso de CPU e memória nos processos que hospedam os componentes da solução, tornando possível uma visão global da carga de cada nó participante da aplicação. Além de informações sobre o tempo de resposta dos métodos invocados.

A coleta das informações propostas neste trabalho foi realizada de três formas distintas, cada uma utilizando um tipo de recurso, são elas:

- *Daemon* no *Container*: Cada *container* irá inicializar uma *thread* que será responsável por coletar periodicamente informações sobre a execução daquele processo;
- Por interceptadores: Informações que envolvam a cascata de chamadas de métodos remotos realizadas durante a execução da aplicação, essas

métricas são coletadas a partir de medições realizadas em diferentes pontos de interceptação;

– Metadados providos pela estrutura de execução do SCS: Informações sobre a distribuição (topologia) dos componentes nos diversos nós da rede.

Na figura 5.2 estão as definições das métricas que podem ser obtidas pela ferramenta, e um exemplo de como essas métricas poderiam ajudar no acompanhamento do sistema.

	Métrica	Descrição	Exemplo de Uso
Coletadas no Container	Tempo total de uso da CPU	Tempo total em que o processo utilizou a CPU.	O processo pode executar componentes que são críticos para a aplicação, e mesmo assim ganhar poucas vezes acesso a CPU(starvation). Com isso o usuário poderá aumentar a prioridade do processo que executa esse container para tentar melhorar seu desempenho.
	Tempo de execução do processo (Container)	Tempo total de execução do container desde a inicialização da coleta de informações.	Como algumas informações coletadas podem ser armazenadas de forma cumulativa este tempo pode ser utilizado para relativizar outras métricas. Exemplo: o tempo total de uso da CPU de um processo é de 10 segundos, com a informação do tempo de execução do processo podemos perceber o quanto isso reperesenta do total em que o processo ficou ativo.
	Uso de CPU	Porcentagem de ocupação da CPU no nó onde está executando o container em um determinado instante.	O usuário pode desejar operar com um determinado nível de uso da CPU. Caso os limites definidos sejam excedidos o usuário pode descarregar alguns componentes de um nó e colocá-los em máquinas menos sobrecarregadas.
	Uso de memória	Memoria alocada para o processo.	O usuário pode desejar operar com um determinado nível de carga na máquina.
Coletadas nos Interceptadores	Tempo de Resposta	Tempo que um determinado método servidor levou para responder a uma requisição.	Com esta métrica o desenvolvedor pode verificar quais métodos estão demorando excessivamente para retornar, e tentar otimizá-los em uma nova versão do componente.
	Tempo de CPU	Tempo de uso da CPU utilizado pelo processo durante o tratamento da requisição.	Assim como o tempo de uso de CPU coletado no container, com essa informação podemos observar quanto tempo o processo teve acesso a CPU durante o tratamento de um determinado método.
	Destino das Requisições	Quais métodos e de quais componentes foram chamados a partir de um determinado container.	Em uma aplicação distribuída em vários nós pode ser difícil identificar entre quais máquinas estão ocorrendo trocas de mensagens. Com essa métrica teremos essa informação.
	Tempo de Rede	Tempo gasto em tráfego de rede durante o tratamento de uma determinada requisição.	Gargalos de rede podem estar onerando o desempenho do sistema, o desenvolvedor, baseado nesta métrica, poderá reorganizar os componentes de forma a aliviar o gargalo formado.
Obtidas pelo SCS	Lista de Containers	Listagem dos containers presentes em cada nó.	Auxilia na visualização da distribuição dos componentes.
	Lista de Componentes	Listagem de componentes presentes em cada container	

Figura 5.2: Métricas Básicas Coletadas

As métricas apresentadas estão prontas para serem utilizadas pelo desenvolvedor. No entanto, é razoável que ele possa escolher quais dessas métricas são realmente interessantes para sua aplicação, ou mesmo definir novas métricas que ainda não tenham sido implementadas. A arquitetura proposta oferece alguns tipos de configurações de coleta que podem ser realizadas durante a execução do sistema. Assim, o usuário tem uma forma de evitar a coleta de informações desnecessárias, como também de definir novas métricas. A Seção 5.3 irá abordar esses recursos em maiores detalhes.

5.3 Instrumentando o SCS

Como usamos o SCS para implementar nossa estrutura de componentes, precisamos realizar algumas alterações em seu código para que o mesmo fosse capaz de coletar as informações desejadas. Essas alterações não irão impactar o desenvolvedor dos componentes SCS, somente irão fornecer novos recursos para que a arquitetura passe a ser instrumentada.

Para configurar a coleta de informações adicionamos ao *container* uma nova faceta, a *InstrumentationManager*. Os itens abaixo apresentam brevemente cada um dos métodos implementados por essa faceta. Detalhes da implementação de cada um deles são encontrados nas próximas seções. A IDL dessa faceta é apresentada na listagem 5.1.

- *setDataCollector*: Inicializa ou interrompe a coleta de dados;
- *setEventChannel*: Informa o canal de eventos onde as informações devem ser publicadas;
- *setCollectInterval*: Indica o intervalo entre as coletas realizadas no *daemon* do *container*;
- *getInterceptedMethods*: Recupera os métodos que são interceptados;
- *addInterceptedMethod*: Adiciona um método na lista de métodos interceptados;
- *removeInterceptedMethod*: Remove um método da lista de métodos interceptados;
- *getMetrics*: Recupera as métricas que estão sendo coletadas;
- *addMetric*: Adiciona uma métrica para coleta no *daemon* do *container*;
- *removeMetric*: Remove uma métrica que está sendo coletada no *daemon* do *container*;
- *setPolicy*: Indica a política de publicação que deve ser usada em uma determinada métrica.

```
1   interface InstrumentationManager {
2
3       void setDataCollector(in boolean status);
4       void setEventChannel(EventChannel channel)
5        raises (EventChannelNotFound);
6
7       void setCollectInterval(in unsigned long long time)
8        raises (InstrumentationNotStartup);
```

```
9
10          StringSeq getInterceptedMethods();
11          void addInterceptedMethod(in string methodName);
12          void removeInterceptedMethod(in string methodName);
13
14          MetricsSeq getMetrics();
15          void addMetric(Metrics metric);
16          void removeMetric(Metrics metric);
17
18          void setPolicy(in Metrics metric, in PublishPolicy policy);
19
20      };
```

Listagem 5.1: IDL do InstrumentationManager

5.3.1 Com Interceptadores

Para realizar a coleta de métricas nos interceptadores criamos um interceptador chamado *StatsServiceInterceptor*, preparado para interceptar tanto chamadas de cliente como de servidor. A arquitetura CORBA realiza diversas chamadas remotas durante a execução da aplicação. Tais chamadas não se restringem unicamente às realizadas explicitamente pelo desenvolvedor. Dessa forma, se interceptarmos todas as chamadas que cheguem ao *container* estaremos interceptando métodos utilizados internamente pela arquitetura CORBA, além dos próprios métodos utilizados pelo mecanismo de instrumentação, por exemplo, quando as métricas são encaminhadas para o canal de eventos. Isso geraria uma grande perda de desempenho na aplicação.

Além disso, uma aplicação distribuída pode realizar um número grande de chamadas remotas de forma que o administrador da aplicação poderá ter interesse em apenas uma grupo dessas chamadas, evitando a interceptação de todas as mensagens trocadas. Para contornar essas dificuldades, na arquitetura proposta o usuário poderá escolher quais métodos ele deseja interceptar, utilizando o método *addInterceptedMethod* e *removeInterceptedMethod* definidos na faceta *InstrumentationManager*.

A interceptação das chamadas é realizada em quatro pontos distintos. A seguir apresentamos esses pontos e as informações obtidas em cada um deles.

- Envio da requisição (*send_request*): Grava-se uma marca de tempo colocando-a junto do contexto da requisição. Essa marca será usada futuramente para cálculo do tempo de rede.

- Recebimento da requisição (*receive_request*): Grava-se uma marca de tempo para o cálculo do gasto com processamento no servidor. Verifica-se qual componente está recebendo a requisição, na implementação Java essa verificação não é trivial e a subseção (*Identificando o Componente*) aborda os detalhes de como isso é resolvido. Utilizando uma biblioteca específica, subseção *Biblioteca do Sistema*, são coletadas informações sobre o estado atual da CPU. Finalmente, as informações coletadas são inseridas em uma estrutura indexada pelo identificador da requisição.
- Envio da resposta (*send_reply*): São coletadas as informações sobre o estado atual da CPU. São acessados os dados armazenados no item anterior. Confronta-se os dados obtendo o tempo médio de uso de CPU do processo e o tempo total para tratamento da requisição. Armazena-se essa última informação e a marca de tempo que foi enviada do cliente no contexto da resposta. As informações obtidas são publicadas.
- Recebimento da resposta (*receive_reply*): Uma nova marca de tempo é coletada. Retira-se do contexto informações sobre o tempo de processamento no servidor e marca de tempo que foi enviada pelo próprio cliente. Subtraindo a marca de tempo atual da que foi enviada é obtêm-se o tempo decorrido para recebimento da resposta no cliente. Subtraindo esse valor do tempo de processamento no servidor chega-se o tempo de rede da requisição.

Caso ocorra uma exceção no servidor durante o tratamento da requisição o método invocado no interceptador não será o *send_reply* e sim o *send_exception*. O mesmo ocorrerá no interceptador do cliente onde o método invocado deixará de ser o *receive_reply* e passará a ser o *receive_exception*. Para que possamos tratar também as requisições que geraram exceções essas duas novas funções também seguiram os passos definidos nos itens acima.

Biblioteca do Sistema

Como precisamos coletar algumas informações do sistema operacional da máquina que executa os componentes, criamos uma biblioteca que deve ser implementada para cada tipo de sistema operacional que se deseje monitorar. No caso deste trabalho implementamos essa biblioteca para as plataformas Linux e Windows.

Ela possui métodos que coletam algumas formações relativas ao processo, como: a carga da CPU em um determinado instante, uso de memória, uso de *IO* e o identificador do processo. Essa biblioteca está implementada em C. Para utilizar a biblioteca a partir do código Java utilizamos *Java Native Interface*(JNI) [28], fornecendo uma camada para que o código Java possa acessar a biblioteca C. Dessa forma, quando implementamos a biblioteca para um segundo sistema operacional alteramos o código em C porém não precisamos realizar alterações no código Java.

Já Lua oferece uma API que permite o acesso a código em C a partir de scripts Lua. Dessa forma, podemos tornar essa biblioteca acessível nas duas linguagens, e seguindo a interface da biblioteca podemos implementá-la para diversas plataformas como Linux, Windows e outras.

Identificando o Componente

Quando o interceptador do objeto servidor recebe a requisição, o objeto *ServerRequestInfo* contém informações sobre a chamada. Com o método *target_most_derived_interface()* obtemos a interface que irá tratar aquele método. Porém, podemos ter várias instâncias de um determinado componente carregado naquele *container* e a informação da interface não diz qual instância irá tratá-la.

Para contornar esse problema alteramos o código interno do *container* SCS. A estrutura que possui a listagem dos componentes é o *container*, logo, o interceptador precisará conhecer o seu *container*. Poderíamos passar o *container* para o interceptador quando o mesmo fosse criado, porém nesse momento o *container* ainda não existe. Dessa forma o *container*, quando é criado, informa para o interceptador a sua referência.

Agora quando recebemos uma requisição podemos obter a faceta *ComponentCollection* do *container* e com ela obter quais componentes estão executando no *container*. O objeto *ServerRequestInfo* oferece o *object_id* do objeto que está sendo invocado, com ele através do POA podemos obter uma referência ao objeto CORBA que está sendo chamado.

Esse objeto estará apontando para a faceta que implementa o serviço desejado pelo cliente, esse tipo de objeto, de acordo com a implementação JACORB [18] de CORBA, implementa o método *_get_component()* que retorna o componente onde essa faceta é implementada. Com ele podemos

obter no *ComponentCollection* qual instância estamos chamando. Com essa adaptação feita no SCS conseguimos identificar a instância exata que está tratando a requisição. A listagem 5.2 mostra o código Java comentado que realiza essa operação.

```
1   ...
2
3  // Obtem a interface ComponentCollection a partir da
4  // referencia do container.
5  ComponentCollectionServant compColl =
6      (ComponentCollectionServant)container.getFacets().
7                      get(ContainerApp.FACET_COMPCOLLECTION);
8
9  org.jacorb.orb.Reference ic = null;
10 try {
11   // Obtem uma referencia para o CORBA Object do IComponent
12   // que implementa a faceta.
13   ic = (org.jacorb.orb.Reference) poa.reference_to_servant(
14             poa.id_to_reference(ri.object_id())).
15 } catch (WrongAdapter e) {
16   e.printStackTrace();
17 }
18 // Obtem o objeto IComponent
19 IComponent icHandle = IComponentHelper.narrow(ic);
20 // Busca na ComponentCollection o componentHandle daquele componente.
21 ch = compColl.getComponent(icHandle);
22
23 ...
```

Listagem 5.2: Código Java que Obtém a Instância do Componente

5.3.2 Com *Threads*

Na estrutura instrumentada, o *container* pode disparar uma *thread* que irá coletar de tempos em tempos dados sobre a carga do processo. Essa *thread* irá varrer um vetor contendo todas as métricas que foram adicionadas ao *container* e assim irá executar a coleta de dados. Assim como no caso dos interceptadores, logo que as métricas são coletadas elas são publicadas no canal de eventos. Caso o usuário desative a coleta de informações usando o método *setDataCollector* (listagem 5.1), a execução dessa *thread* é interrompida.

O intervalo entre as coletas pode ser ajustado com o método *setCollectInterval* (listagem 5.1), porém vale ressaltar que um intervalo muito curto entre as coletas pode resultar um uma maior perda de desempenho da aplicação, da mesma forma que intervalos grandes pode levar a perda de informações. Esse intervalo deve ser ajustado de acordo com os requisitos de monitoração de cada aplicação.

Caso julgue que apenas algumas métricas são necessárias para o acompanhamento da aplicação, o usuário poderá escolher, em tempo de execução, quais métricas ele deseja que sejam coletadas. Para isso ele poderá usar os métodos *addMetric* e *removeMetric*.

5.3.3 Definição de Novas Métricas

Neste trabalho implementamos algumas métricas que julgamos ser úteis para monitoração de sistemas de componentes distribuídos. Porém, muitas outras métricas podem ser úteis para o administrador da aplicação, e quais serão essas métricas dependerá de qual aspecto do sistema deseja-se obter maiores informações.

A necessidade de observação dessas novas métricas pode surgir quando o sistema monitorado já está em operação. Dessa forma, é importante que a adição de métricas seja realizada sem que o sistema tenha que ser paralisado. Para lidar com a necessidade de inserirmos novas métricas dinamicamente na arquitetura de monitoração, disponibilizamos alguns métodos para facilitar a adição.

Para definir uma nova métrica o usuário deverá estender a classe *MetricCollector* listagem 5.3 definida no pacote *scs.instrumentation.metrics*. Essa classe possui um atributo do tipo *Metrics* (listagem 5.4) que contém a métrica coletada, e um método abstrato chamado *collect()* que deve ser implementado pelo usuário com o algoritmo de coleta desejado.

```
1
2  public abstract class MetricCollector {
3
4    private Metrics metric = new Metrics();
5
6    ...
7
8    public Metrics getMetric()
9    {
10     return metric;
11   }
12
13   public abstract Metrics collect();
14
15 }
```

Listagem 5.3: Código Parcial da Classe MetricCollector

Em seguida o usuário deve compilar a classe e colocar o arquivo *.class*

gerado junto com as demais métricas que são coletadas pelo *container*. Para iniciar a coleta da nova métrica o usuário deve utilizar o método *addMetric* da interface *InstrumentationManager*, passando como parâmetro um objeto do tipo *Metrics* que terá no campo *className* o nome completo da nova classe implementada, por exemplo *scs.instrumentation.metrics.NovaMetrica*.

```
1     struct Metrics {
2        string name;
3        string description;
4        any    value;
5        string className;
6        PublishPolicy   policy;
7     };
```

Listagem 5.4: Metrica

5.4 Publicação

Coletar a métrica é um dos passos para monitorar o ambiente de execução da aplicação, porém, de nada adiantará termos a métrica se não existir uma forma eficiente dessa informação ser entregue ao administrador do sistema. Sendo assim, o objetivo agora passa ser oferecer uma forma de disponibilizar os dados coletados. Como a coleta de informações invariavelmente irá onerar o desempenho da aplicação, procuramos desenvolver uma forma de distribuir essas informações gerando um baixo custo.

Uma das formas de publicar as informações obtidas seria salvando-as em arquivos, no entanto, o custo de escrita em disco é alto, e em um ambiente com diversas métricas sendo coletadas teríamos muita escrita em disco. Além disso, disponibilizar essas informações em arquivos poderá dificultar a recuperação desses dados em um ambiente distribuído.

Outra solução seria armazenar os dados coletados em estruturas de dados, deixando as informações em memória, o que diminuiria o tempo para armazenamento e acesso. No entanto, como existe um alto potencial de crescimento das informações armazenadas, poderemos aumentar muito o consumo de memória da máquina, de forma que essas estruturas passariam a disputar esse recurso com a aplicação.

Qualquer solução que mantenha as métricas coletadas nos nós onde ocorreu a coleta irá prejudicar a aplicação, seja na hora de armazenar as informações, seja na hora em que os interessados por elas tentem acessá-las.

Para contornar essa questão optamos por usar um serviço de eventos para publicação dessas informações. Nesse caso, assim que a informação é obtida, ela é enviada para o serviço de eventos, desse modo a máquina que está executando a aplicação não fica responsável por gerenciar esse dado. Cada usuário interessado em receber informações poderá se conectar ao serviço de eventos como consumidor e dessa forma receber os valores obtidos. Assim, fica a cargo do consumidor do canal de eventos como manter as informações.

Outra vantagem dessa arquitetura é que podemos utilizar mais do que um canal de eventos para publicação de informações. Assim, caso um canal esteja recebendo dados de muitos *containers*, podemos criar um outro canal e reconfigurar a aplicação de forma que ela passe a usar o novo canal criado, aliviando o primeiro. Ainda sobre os benefícios do uso de canal de eventos, ele permite que diversos clientes recebam as informações.

A faceta *InstrumentationManager* (listagem 5.5) disponibiliza o método *setEventChannel*, que recebe uma referência para o canal de eventos onde o *container* deverá enviar as informações coletadas.

```
1    interface InstrumentationManager {
2       ...
3       void setEventChannel(EventChannel channel)
4         raises (EventChannelNotFound);
5
6       void setPolicy(in Metrics metric, in PublishPolicy policy);
7
8       PolicySeq getPolicys();
9    };
```

Listagem 5.5: InstrumentationManager

Nesse canal, tanto métricas coletadas a partir dos interceptadores, como as métricas coletadas a partir dos *containers* são publicadas. Cada *container* poderá publicar suas informações em um determinado canal de eventos, a escolha do administrador. Na figura 5.3 temos representado o *ExecutionNode* responsável pela criação do *container*, que por sua vez é responsável por carregar os componentes da aplicação. O objeto *StatsServiceInterceptor* é responsável pela interceptação das chamadas. As linhas pontilhadas indicam a direção e o destino das informações coletadas. No caso apresentado introduzimos um novo elemento chamado *StatsCollection* que será apresentado na seção 5.4.2.

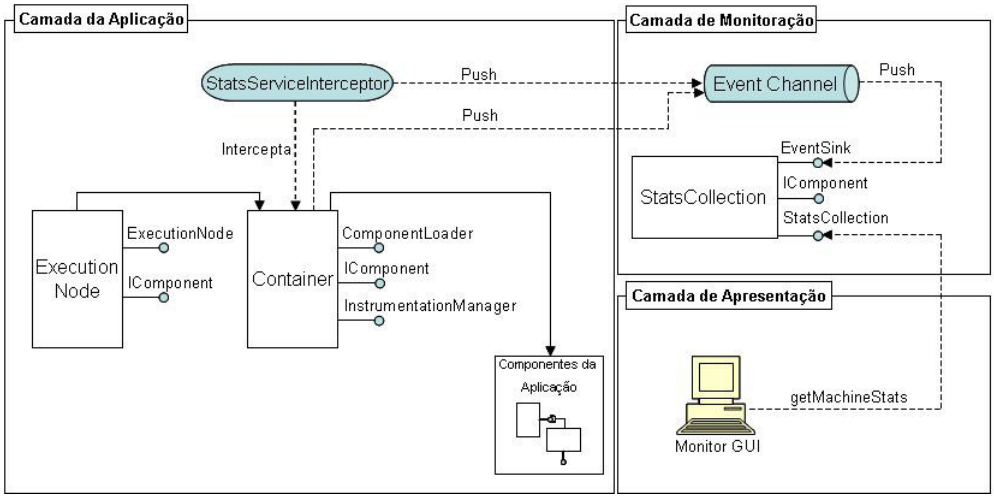

Figura 5.3: Arquitetura SCS Instrumentada

5.4.1
Estrutura de Dados

Após coletar as informações optamos por agrupá-las em uma estrutura de dados, antes que seja feito o envio para o canal de eventos. Assim podemos reduzir o número de mensagens enviadas, evitando aumentar a sobrecarga do sistema. Dessa forma definimos uma estrutura que poderá agregar todas as informações coletadas.

O diagrama de classe da figura 5.4 ilustra a estrutura criada. Nela definimos o objeto *ContainerStats*, que possui o nome da máquina *machineName* onde o *container* está, o nome do *container*, *containerName*, uma coleção com as interfaces que estão carregadas nele *InterfaceStats*, uma coleção com as métricas que foram coletadas no *daemon* do *container*, e finalmente uma coleção contendo os *containers* para onde foram realizadas chamadas, *targetContainersStats*.

Cada componente possui uma coleção contendo os métodos que foram chamados. Cada objeto dessa coleção de métodos possui o nome do método chamado e uma coleção das métricas coletadas pelos interceptadores durante o processamento daquele método.

Cada métrica possui o nome, a descrição, o valor e a política de publicação associada a essa métrica, a seção 5.4.4 irá explicar em detalhes o que são essas políticas. A listagem 5.4 mostra a idl da estrutura métrica.

A coleção *targetContainersStats* é formada por objetos do tipo *containerStats*, navegando pela estrutura que já foi definida, chegaremos aos métodos

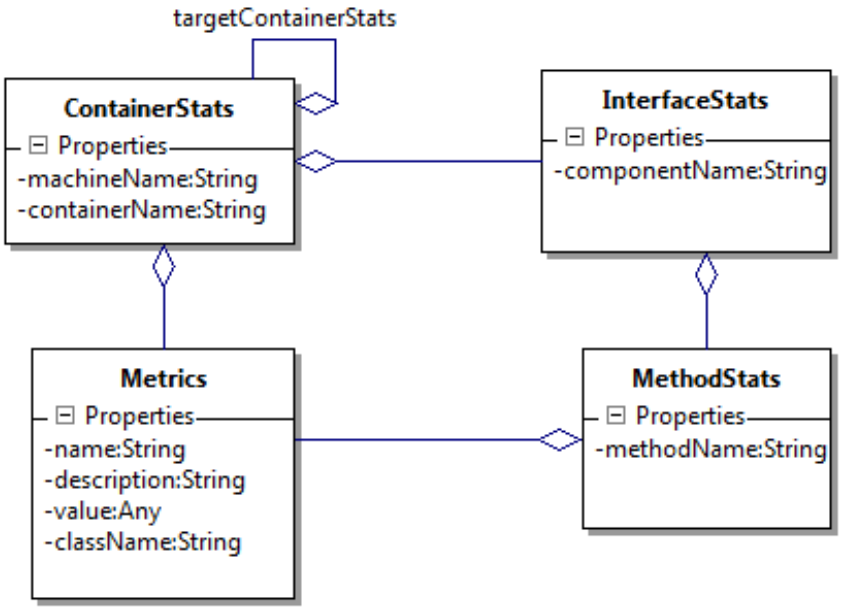

Figura 5.4: Estrutura de Dados

que foram chamados nos componentes de outros *containers*, e assim obter as métricas coletadas nessa comunicação. A seção 5.4.3 irá apresentar uma ferramenta capaz de apresentar as informações coletadas, nela torna-se mais clara a forma de navegação dessa estrutura.

Como podemos observar na figura 5.3 podemos ter tanto o *daemon* do container como os interceptadores publicando informações. No caso do *daemon* do *container*, ele realiza a coleta de todas as métricas que foram selecionadas e monta um objeto do tipo *containerStats* com os dados armazenados na coleção de métricas dessa estrutura. Já a coleta realizada a partir dos interceptadores, que obtém informações a respeito do fluxo das chamadas realizadas, irá armazenar as informações na coleção de componentes existente dentro do *containerStats*.

Ao se cadastrar no serviço de eventos, uma aplicação cliente da infraestrutura passará a receber objetos do tipo *containerStats*, e deverá navegar pela estrutura recebida para encontrar os dados coletados. A partir desse momento fica a cargo do consumidor desses eventos definir a forma como serão tratadas as informações recebidas.

5.4.2
Cliente Agregador

Todo cliente da arquitetura de monitoração poderá receber os eventos diretamente do canal de eventos onde os mesmos são publicados. No entanto, pode ser interessante para o cliente obter essas informações consolidadas, afinal, um canal pode receber informações de diversas máquinas e de diversos *containers*. Para consolidar as informações coletadas e oferecer uma estrutura organizada, implementamos um cliente agregador chamado *StatsCollection*.

O *StatsCollection* pode se conectar a quantos canais de eventos forem necessários, ele irá agregar informações de diversas máquinas, e poderá oferecer essas informações em uma única estrutura. Para isso definimos a estrutura *MachineStats* (listagem 5.6). Ela possui o nome da máquina *machineName*, uma coleção de *ContainerStats* chamada *ContainerStatsCollection*, que irá conter as métricas coletadas em cada *container* inicializado em uma determinada máquina.

```
1   struct MachineStats{
2        string machineName;
3        ContainersStatsSeq ContainerStatsCollection;
4   };
```

Listagem 5.6: Estrtura de Dados MachineStats

A interface *StatsCollection*, listagem 5.7, irá oferecer um método para o usuário receber a estrutura com todas as informações recolhidas até aquele momento. E também o método *connectToEventSource* que permite conectar o componente a um canal de eventos específico.

No *StatsCollection* cada métrica coletada a partir dos interceptadores, como o tempo de execução de um determinado método servidor, é somado aos valores recebidos anteriormente, como o *StatsCollection* mantém o número de vezes que cada método foi chamado é possível calcular a média do tempo de execução do método, o mesmo ocorre com a métrica do tempo de CPU utilizado pelo método durante o tratamento da requisição e o tempo de rede. Já algumas métricas como tempo total de uso de CPU do processo, que são coletadas a partir do *container*, o *StatsCollection* armazena somente o último valor publicado.

```
1   interface StatsCollection {
2     MachineStatsSeq getMachinesStats();
3     void connectToEventSource(in EventChannel channel);
4 };
```

Listagem 5.7: Interface do *StatsCollection*

O agregador de informações implementado neste trabalho serve como uma forma ilustrativa de representar a flexibilidade da arquitetura de coleta e publicação de informações. Novos agregadores podem ser definidos e os resultados podem ser publicados em novos canais de eventos onde clientes poderão obter informações de mais alto nível de acordo com o tipo de agregação que for realizada no agregador desenvolvido.

5.4.3 Monitor Gráfico

Após coletar as informações precisamos oferecer uma forma do usuário visualizar as informações obtidas. Implementamos um cliente gráfico do *StatsCollection* que recebe as informações e exibe na forma de árvore para o cliente. A figura 5.5 apresenta a interface gráfica do monitor de informações. As funcionalidades realizadas pelo monitor são:

- TreeView: Exibe todas as máquinas que estão sendo monitoradas, todos os *containers* presentes nessas máquinas, as instâncias de cada componentes que estão executando dentro de cada *container*, e finalmente os métodos chamados em cada componente. No nó comunicação é possível verificar para quais *containers* partiram requisições vindas desse *container*, a instância de componente que foi chamada, e o método invocado.
- Detalhamento: Na parte inferior estão as informações sobre as métricas coletadas, e a política que está associada àquela métrica.
- Atualizar: Atualiza o *treeview*, basicamente ele busca novos dados no *StatsCollection*.

Para o exemplo capturado na figura 5.5, utilizamos uma aplicação chamada *ProducerConsumer*, que foi utilizada na seção 6.1 para realizar testes com a arquitetura proposta. Nesse caso o exemplo foi executado em uma única máquina na qual três *containers* foram criados. O terceiro nível da árvore mostra esses *containers*, são eles: *ConsumerContainer*, *ProducerContainer*, *MediatorContainer*. Um nível abaixo encontram-se os componentes carregados dentro de cada *container* e um nó chamado comunicação. Para os *containers* *ConsumerContainer* e *ProducerContainer* esse nó está vazio, pois esses componentes não executaram nenhuma chamada remota. Já no *MediatorContainer* seus componentes realizaram chamadas para dois *containers* como pode ser

visto nos nós filhos do nó comunicação. Navegando nesse nó é possível verificar qual foi o container, o componente e o método chamado.

Já os nós irmãos de comunicação podem ser utilizados para verificar quais métodos foram servidos por determinado componente. No caso da figura 5.5, o *container MediatorContainer* está selecionado, e a tabela na parte inferior está mostrando as métricas coletadas nele, e a política que está associada a cada métrica.

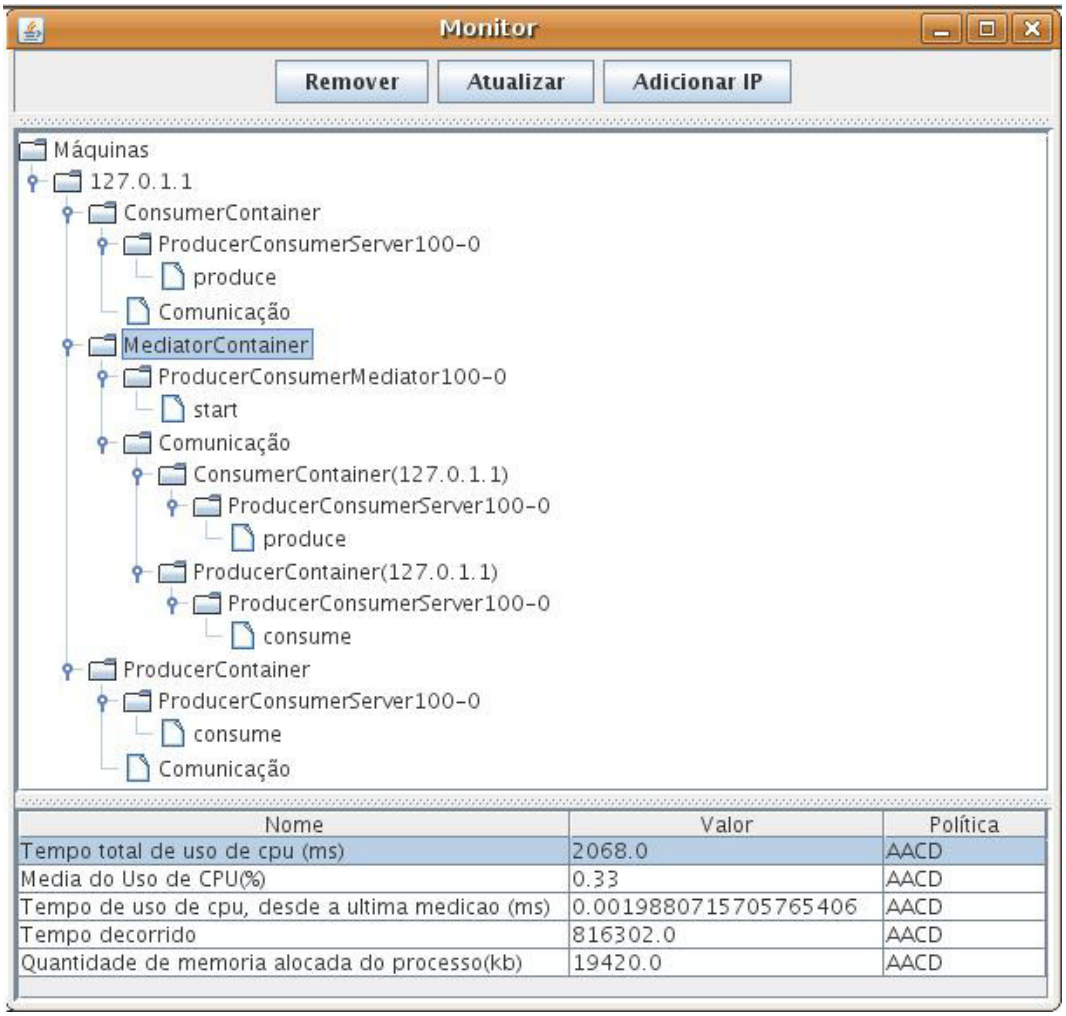

Figura 5.5: Interface Gráfica da Ferramenta Monitora

5.4.4 Políticas de Publicação

Uma das desvantagens da escolha de um canal de eventos para publicação de informações é o grande fluxo de mensagens que ele poderá gerar. Ao enviar os dados contendo as métricas coletadas, inevitavelmente estaremos con-

sumindo banda de rede que pode estar sendo compartilhada com a aplicação. Caso o número de mensagens contendo dados coletados comece a aumentar, a aplicação poderá sofrer com problemas de comunicação entre seus nós.

Para tentar minimizar esse custo implementamos algumas políticas de publicação. Essas têm como objetivo evitar o anúncio desnecessário de informações. Um exemplo que podemos usar para ilustrar uma situação onde a publicação não se faz necessária seria: supondo que a única métrica coletada seja o uso de memória de um determinado processo, e que a infraestrutura está configurada para realizar a coleta de informações de dois em dois segundos, porém, o uso de memória entre uma medição e outra não variou. Dessa forma, pode ser razoável que não seja publicada a segunda medição realizada, dado que esta será feita com o mesmo valor da anterior.

Para identificar o momento correto de publicar as informações coletadas implementamos as políticas de publicação. Essas têm como objetivo definir quando um valor coletado deve ser informado ao observador, evitando informar valores redundantes ou variações irrelevantes para o administrador da aplicação. O grande desafio é manter um equilíbrio entre a periodicidade dos anúncios e a taxa de variação dos valores.

Se por um lado utilizar políticas que diminuam o número de anúncios de informações que tenham alta taxa de mudanças pode deixar o cliente desatualizado sobre o real comportamento do recurso, por outro lado, realizar anúncios com intervalos muito pequenos para métricas que variam pouco poderá levar a um excesso de mensagens na rede, além de aumentar as chances de publicações irrelevantes. Dessa forma, a política ideal que deve ser usada, dependerá do tipo de uso que será dado a essa métrica e principalmente do tipo do recurso que está sendo monitorado.

Para implementar as políticas de publicação definimos na *idl* do SCS uma estrutura chamada *PublishPolicy* (listagem 5.8). Essa estrutura possui o nome da política *policyName*; um vetor de *strings*, *args*, com os parâmetros de entrada da política; a propriedade *className* que recebe o nome da classe que implementa a política; e a propriedade *policyDesc* que possui uma descrição da política.

```
1    struct PublishPolicy{
2       string policyName;
3       string policyDesc;
4       string className
5       StringSeq args;
6    };
```

Listagem 5.8: Estrutura da Política de Publicação

Como podemos ver na listagem 5.4, cada métrica possui uma política associada a ela. Dessa forma podemos definir políticas de publicação diferentes para cada métrica que é coletada. Neste trabalho implementamos seis políticas de publicação distintas. Basicamente podemos dividi-las em três grupos: um que leva em consideração o tempo entre cada publicação, outro que observa o valor coletado a cada medição e um que procura observar tanto tempo como valor coletado.

O primeiro grupo de políticas procura definir um intervalo de tempo ideal entre as publicações, observando para isso o número de mudanças ocorridas na métrica coletada durante um certo intervalo de tempo. Já o segundo grupo possui políticas que irão observar a diferença entre os valores coletados e a partir daí definir se essa variação deve ou não ser informada. Finalmente, o terceiro grupo irá juntar as duas formas de publicação e assim irá anunciar novos valores tanto devido ao tempo como por consequência de alterações no valor coletado.

As políticas implementadas neste trabalho foram propostas em [2], artigo citado na seção 2.2. A seguir vamos avaliar cada uma das políticas implementadas.

Políticas Baseadas no Valor

Políticas baseadas em valor levam em consideração a variação do valor coletado entre a última medição e o último anúncio para determinar se a métrica deve ser publicada. Nesse tipo de política utilizam-se duas abordagens: uma chamada *Announce-On-Change*(AOC), que envia informações caso ocorra qualquer tipo de mudança no dado, e a segunda *Announce-Absolute-Change* (AAC), que publica informações de acordo com o tamanho da variação ocorrida.

Na política AOC, cada valor anunciado será armazenado, quando ocorrer uma coleta o valor obtido será comparado ao que foi anunciado anteriormente. Caso esses valores sejam diferentes ocorrerá a publicação do novo valor que também será armazenado para comparação durante a próxima medição. O principal foco dessa política é evitar publicações redundantes.

Na política AAC, o valor coletado será publicado caso a diferença entre ele e a coleta anterior seja superior a um valor previamente estipulado. Esse valor poderá ser estático ou dinâmico, o que levou a criação de duas políticas

independentes para tratar cada caso. No caso do valor estático (AACS) o cliente informa o valor, esse valor nunca muda, assim qualquer variação que supere o valor informado irá gerar uma publicação.

No caso do valor dinâmico (AACD) o que teremos é uma média das últimas variações publicadas, dessa forma procura-se obter quanto pode ser considerada uma modificação significativa para um determinado recurso. Ou seja, caso o recurso tenha constantemente grandes variações, o valor especificado para que a publicação ocorra será alto, caso contrário ele ficará menor. Esse valor será calculado dinamicamente de acordo com as variações apresentadas entre as coletas.

O valor calculado dinamicamente será inicializado em 0, e seu valor será modificado de acordo com o valor das coletas. Assumindo que A1,A2,..,An, sejam os sucessivos anúncios feitos, e que $AVC_i = |A_i - A_{i-1}|$, podemos calcular o valor dinâmico, que chamaremos de $d\_threshold$ da seguinte forma:

$$d\_threshold = \frac{1}{NA} \times \sum_{i=1}^{NA} AVC_i$$

A figura 5.6 mostra o caso de coleta de informações sobre o uso de memória livre, nesse caso o valor fixo utilizado para o AACd foi 50. Nessa figura fica claro como pode variar o número de anúncios de acordo com a política escolhida.

Timestamp	Memfree(kb)	change	d_threshold	AOC	AACs	AACd
13:49:13	4408	N/A	0.0	N/A	N/A	N/A
13:49:14	4404	4	4.0	Y	N	Y
13:49:15	4404	0	4.0	N	N	N
13:49:16	4404	0	4.0	N	N	N
13:49:17	4412	8	6.0	Y	N	Y
13:49:18	4608	196	69.3	Y	Y	Y
13:49:19	4604	4	69.3	Y	N	N
N/A				4	1	3

Figura 5.6: Políticas AOC, AACS, e AACD

Políticas Baseadas no Tempo

Nesse tipo de política leva-se em consideração o tempo entre um anúncio e outro. Duas formas de calcular o intervalo de tempo foram implementadas: *Announce-Regular-Interval* (ARI) e *Announce-Dynamic-Interval* (ADI).

No caso da ARI o tempo entre os avisos será fixo e determinado pelo usuário. Assim, caso o cliente queira a informação mais atualizada possível ele poderá ajustar um valor pequeno para usar entre as publicações. É interessante observar que um tempo muito curto irá gerar uma grande quantidade de mensagens, e um tempo grande pode gerar perda de informações.

O tempo utilizado pelo *daemon* do *container* entre uma coleta e outra é definido na interface do *InstrumentationManager* (seção 5.3) do *container*. Dessa forma de nada adiantará ajustar um tempo para o intervalo entre as publicações menor do que a periodicidade em que a coleta é realizada. Por exemplo, se a infra-estrutura de monitoração foi ajustada para realizar a coleta de métricas de segundo em segundo, e a política associada a uma das métricas coletadas estipular um tempo de meio segundo para publicação, certamente serão realizadas publicações redundantes.

Para a política ADI calcula-se o tempo entre os anúncios dinamicamente, baseado no número de variações de valor do recurso monitorado num determinado período de tempo. Supondo que T0,T1,...,Tn, sejam os momentos que o valor foi modificado, e que NC seja o número de mudanças ocorridas, pode-se definir o intervalo entre os anúncios com a seguinte fórmula:

$$DTI = \frac{1}{NC} \times \sum_{i=1}^{NC} (T_i - T_{i-1}) = \frac{1}{NC} \times (T_{NC} - T_0)$$

O principal objetivo dessa política é perceber a frequência de alterações de uma determinada métrica. Caso uma métrica mude valor com grande frequência faz sentido que o valor seja publicado com maior periodicidade e caso contrário que o dado seja anunciado com intervalos de tempo maiores.

A figura 5.7 mostra um exemplo de uso das métricas baseadas em tempo utilizadas. Nesse caso o tempo utilizado para política ARI foi 5 segundos. A coluna DTI (*Dynamic Time Interval*) mostra o tempo calculado dinamicamente. Já as colunas *TIMER ari* e *TIMER adi* indicam o valor do contador das políticas ARI e ADI respectivamente.

Política Híbrida

Essa política utiliza uma combinação das políticas apresentadas anteriormente. Agora, tanto tempo quanto valor serão considerados no momento de realizar o anuncio. Essa política será chamada de *Announcing with Change*

Timestamp	Memfree(kb)	Ti	DTI	$Timer_{ARI}$	ARI	$Timer_{ADI}$
13:49:13	4408	T0	N/A	5	N	∞
13:49:14	4404	T1	1	4	N	1
13:49:15	4404	0	1	3	N	1
13:49:16	4404	0	1	2	N	1
13:49:17	4412	T2	2	1	N	2
13:49:18	4608	T3	1	5	Y	1
13:49:19	4604	T4	1	4	N	1
N/A					1	

Figura 5.7: Políticas ARI e ADI

and Time Consideration(ACTC) (anúncio considerando mudanças e tempo).

Essa política utiliza os mesmos algoritmos implementados nas políticas AACd e ADI. Dessa forma os anúncios serão feitos por tempo, caso o valor de tempo calculado dinamicamente expire, ou devido a alterações no valor, no caso do valor ser maior do que a medida calculada dinamicamente. Como podem existir recursos onde pequenas variações não precisam ser informadas, mesmo que a política de tempo determine o anúncio, foi introduzido nesse método uma variável que indica o valor mínimo para que a publicação ocorra. Assim, caso o tempo expire, a métrica somente será publicada caso a variação de valor seja maior que esse valor mínimo informado pelo usuário.

A figura 5.8 indica os momentos onde ocorreram publicações com uso dessa política. Nesse exemplo assume-se um valor mínimo de 5. No instante 13:49:18 mesmo sem o tempo expirar o anúncio é feito devido à grande variação ocorrida. E no instante 13:49:19 não ocorre publicação devido ao valor ser inferior ao valor mínimo estipulado.

Timestamp	Memfree(kb)	change	d_threshold	TimerADI	ACTC
13:49:13	4408	N/A	4.0	∞	N
13:49:14	4404	4	4.0	1	Y
13:49:15	4404	0	4.0	1	N
13:49:16	4404	0	6.0	1	N
13:49:17	4412	8	69.3	2	Y
13:49:18	4608	196	69.3	1	Y
13:49:19	4604	4	53	1	N
N/A					3

Figura 5.8: Política ACTC

Definição de Novas Políticas

Neste trabalho implementamos algumas políticas para publicação de informações, porém, podem existir casos específicos no qual o usuário deseje criar uma nova política. Dependendo do tipo de aplicação, pode não ser

possível interrompê-la para que a nova política seja inserida. A arquitetura de monitoração desenvolvida possibilita ao usuário inserir uma política de publicação dinamicamente na aplicação.

Para definir a nova política o usuário deverá estender a classe *Policy* (listagem 5.9) definida no pacote *scs.instrumentation.policies*. Essa classe define um construtor que receberá um objeto do tipo *MetricCollector*, que é o responsável por coletar a métrica, com ele a política pode ter acesso ao último valor coletado. Ele também recebe uma sequência de *Strings* que contém os parâmetros que serão usados pela política, o desenvolvedor deve implementar o método *setArgs(String [] args)* que recebe essa sequência e deve se encarregar de tratá-la da forma adequada. A classe *Policy* também define o método *publish()* que deverá ser implementado pelo usuário com a política que ele deseja utilizar; ao final ele deve retornar um *boolean* indicando se deve, ou não, ocorrer a publicação.

```
1  public abstract class Policy {
2    MetricCollector collector = null;
3    protected Policy(MetricCollector collector,String [] args)
4    {
5      this.collector = collector;
6      setArgs(args);
7    }
8    public abstract boolean publish();
9    public abstract void setArgs(String [] args);
10 }
```

Listagem 5.9: Classe Policy

Em seguida o usuário deve compilar a classe e colocar o arquivo *.class* gerado junto com as demais políticas do *container* onde ele deseja utilizá-la. Para associar a política recém criada a uma métrica específica, o usuário deve utilizar a faceta *InstrumentationManager* do *container*, mais especificamente o método *setPolicy* (listagem 5.5), que receberá a métrica que será utilizada e um objeto *PublishPolicy* que contém a descrição da política.

Para facilitar a compreensão montamos um pequeno *script* (listagem 5.10) em Lua que adiciona uma política implementada neste trabalho a uma métrica de um determinado *container*. Nesse *script* assumimos que existe uma aplicação executando em diversos *containers* e que desejamos adicionar a política AACs 5.4.4 para a métrica que coleta o uso de memória no *container* chamado *container1*.

```
1 --enFacet e o objeto que representa o executionNode de onde esta
2 --rodando o container.
3 --Obtem uma referencia para o container
4 local CCJava = enFacet:getContainer("container1")
5
```

```
6  -- Obtem a faceta InstrumentationManager do container1
7  local instMg =
8    CCJava:getFacet("IDL:scs/instrumentation/InstrumentationManager:1.0")
9
10 --Obtem as metricas que estao sendo coletadas nesse container
11 local metrics = instMg:getMetrics()
12
13 --Criacao do objeto PublishPolicy
14 local p = {policyName="AACS",
15   policyDesc = "Valor fixo para verificar publicacao",
16   className = "scs.instrumentation.policys.AACSPolicy", args = {"50"}}
17
18 --Supondo que metrica desejada esteja na posicao 1.
19 --Adicionamos a politica criada a metrica.
20  instMg:setPolicy(metrics[1],p)
```

Listagem 5.10: Script Lua para Determinar a Política de Publicação

Na criação da política passamos no campo *args* o valor que desejamos utilizar no algoritmo, nesse caso 50, linha 16. Após executar esse *script* somente será publicada a métrica quando ocorrer uma variação maior que 50 no valor coletado.