6
Avaliação Experimental

O principal objetivo deste trabalho é oferecer uma arquitetura para monitoração do ambiente de execução de sistemas distribuídos baseados em componentes, de forma que o administrador do sistema possa acompanhar a execução de sua aplicação, e tomar medidas corretivas de acordo com as informações que forem coletadas. Alguns aspectos são de extrema importância quando avaliamos soluções criadas com esse objetivo, podendo destacar três delas: desempenho, facilidade de uso e flexibilidade. A arquitetura de monitoração insere alguns passos no fluxo normal de execução da aplicação, esses passos irão adicionar uma sobrecarga de desempenho nos nós onde a aplicação estará executando, cabendo ao desenvolvedor avaliar se essa sobrecarga será aceitável para sua solução. Caso o uso da estrutura não seja uma barreira, o desenvolvedor também deverá avaliar o impacto causado no desenvolvimento e as possibilidades de configuração que ela oferece.

Neste capítulo procuramos mensurar como a infraestrutura desenvolvida irá onerar o funcionamento do sistema e o resultado do uso das políticas de publicação implementadas. Para atingir esses objetivos implementamos duas aplicações utilizando o modelo SCS instrumentado. Uma delas é uma aplicação simples de produtor consumidor. A outra trata-se de uma aplicação distribuída chamada *MapReduce* [5]. Com ela podemos avaliar o uso da arquitetura em um ambiente que possui uma maior complexidade.

6.1
Produtor Consumidor

Para realizar as medições da sobrecarga inserida no sistema devido à utilização da arquitetura de coleta de dados, optamos por construir uma aplicação simples que atenda as principais características de programação

distribuída, como execução em diversos nós e troca mensagens entre os diversos componentes da aplicação. O fato de se tratar de um sistema simples facilita a apresentação do mesmo, e também a realização de modificações na aplicação para perceber mudanças no comportamento da infraestrutura de monitoração.

6.1.1

Descrição

O exemplo do produtor consumidor implementado trata-se de uma aplicação composta por três componentes, são eles: produtor, consumidor e o mediador. A idéia é que o mediador tenha uma lista de tarefas que deverão ser "produzidas"pelo produtor e posteriormente "consumidas"pelo consumidor. O mediador receberá uma lista de tarefas e irá repassá-las para o produtor que retornará para o mediador o resultado do trabalho produzido, que por sua vez irá encaminhar o resultado para o consumidor. A figura 6.1 mostra como os componentes ficarão conectados durante a execução da aplicação. Vale ressaltar que nessa figura optamos por enfatizar somente os componentes da solução. Por isso, não está representada toda a arquitetura do SCS instrumentado que pode ser vista na figura 4.1.

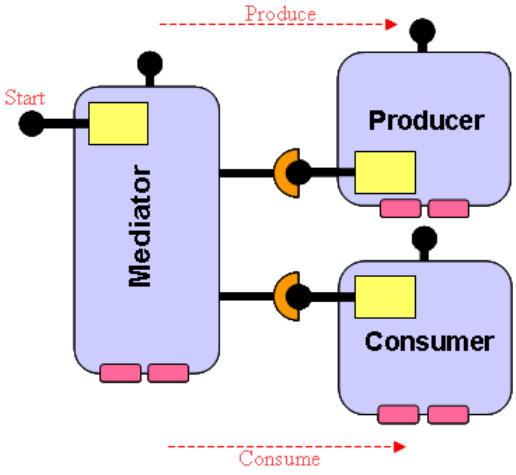

Figura 6.1: Produtor Consumidor

Cada componente irá executar em uma máquina distinta, logo, a troca de mensagens entre eles será feita de forma remota. Aqui podemos utilizar a flexibilidade do uso de canal de eventos para publicação das métricas, colocando-o em uma máquina isolada, assim, evitamos onerar o desempenho do sistema com tarefas de distribuição das mensagens. O mediador irá utilizar os serviços oferecidos pelo consumidor e produtor através de dois receptáculos

onde cada um deles será conectado. O mediador oferece uma faceta, que irá fornecer o método que inicializa a aplicação.

6.1.2 Implementação

Para implementar o produtor consumidor definimos a IDL que pode ser vista na listagem 6.1. A interface *Mediator*, exposta pelo mediador, oferece o método *start* que recebe um conjunto de inteiros representando as tarefas que devem ser executadas. A interface *ProducerConsumerServer* será oferecida tanto pelo componente produtor como pelo consumidor, dessa forma um componente que oferece essa faceta poderá ser um consumidor ou produtor, dependendo do receptáculo onde ele for conectado. O método *produce* recebe um inteiro representando a tarefa, e produz um conjunto de *strings* como resultado. O método *consume* recebe um conjunto de *strings* que representam o dados produzidos e consume essa informação.

```
1  module scs{
2    module demos{
3      module producerconsumer {
4
5        typedef sequence<long> task;
6        typedef sequence<string> producerResult;
7
8        interface Mediator {
9          void start(in task taskList);
10       };
11
12       interface ProducerConsumerServer {
13         producerResult produce(in long task);
14         void consume(in producerResult task);
15       };
16
17     };
18   };
19 };
```

Listagem 6.1: IDL ProducerConsumer

A listagem 6.2 mostra a implementação de cada um dos métodos oferecidos pelos componentes. O método *start* inicialmente obtém uma referência para os seu dois receptáculos, em seguida obtém uma referência para as facetas do consumidor e do produtor que devem estar conectados a esses receptáculos. Ele executa um *loop* chamando para cada inteiro recebido no vetor de tarefas o produtor, e em seguida enviando o resultado obtido ao consumidor.

No método *produce* o inteiro recebido é usado para criar um vetor de mesmo tamanho, contendo a frase *WORK DONE* em todas as suas posições. O método *consume* itera sobre o vetor recebido e não retorna nenhum valor ao mediador.

```
1   public void start(int[] taskList) {
2
3     Receptacle consumerRec = myComponent.getReceptacles()
4             .get("Consumer");
5     Receptacle producerRec = myComponent.getReceptacles()
6             .get("Producer");
7
8     ProducerConsumerServer producer = ProducerConsumerServerHelper
9             .narrow(consumerRec.getConnections().get(0).objref);
10
11    ProducerConsumerServer consumer = ProducerConsumerServerHelper
12            .narrow(producerRec.getConnections().get(0).objref);
13
14    for (int i = 0; i < taskList.length; i++) {
15      int taskLength = taskList[i];
16      String [] producerWork = producer.produce(taskLength);
17      consumer.consume(producerWork);
18    }
19  }
20
21  public String[] produce(int task) {
22    String work = "WORK DONE";
23
24    String [] resp = new String[task];
25    for(int i =0 ; i < task ; i++ )
26    {
27      resp[i] = work;
28    }
29    return resp;
30  }
31
32  public void consume(String[] task) {
33    for(int i =0 ; i < task.length ; i++ );
34  }
```

Listagem 6.2: Funções Básicas *ProducerConsumer*

Com esse exemplo podemos variar a quantidade de valores passados e o tamanho das tarefas, o que nos permitirá observar como será o comportamento da arquitetura para diferentes casos. Para executar os exemplos utilizamos quatro máquinas com a mesma configuração: dois processadores *3 GHz Intel Pentium IV* com 1 *GB* de memória RAM. Para configurar a coleta de informações, lançar os *ExecutionNodes*, os *Containers* e os componentes *ProducerConsumer* utilizamos um *script* Lua. A função *setupContainerInstrumentation* (listagem 6.3) recebe o *container* que se deseja instrumentar e o canal de eventos no qual as informações deverão ser publicadas. Com a faceta *InstrumentationManager* do *container* informamos qual canal de eventos o container deve usar para as publicações, linha 8. Os métodos que devem ser monitorados nos inter-

ceptadores são adicionados em seguida, linhas 10 - 12. Ao final inserimos um conjunto de métricas que desejamos coletar a partir do *container*, linhas 17 - 24.

Nesse ponto não utilizamos políticas de publicação pois elas podem evitar algumas publicações e nossa intenção inicial é validar o pior caso, que ocorre quando todas as métricas coletadas são publicadas. O intervalo usado entre cada coleta feita no *daemon* do *container* foi de 100 milissegundos, linha 15.

```
1  function setupContainerInstrumentation(container, channel)
2
3    local instMg =
4    container:getFacet("IDL:scs/instrumentation/InstrumentationManager:1.0")
5
6    instMg =
7    orb:narrow(instMg, "IDL:scs/instrumentation/InstrumentationManager:1.0")
8    instMg:setEventChannel(channel)
9
10   instMg:addInterceptedMethod("start")
11   instMg:addInterceptedMethod("produce")
12   instMg:addInterceptedMethod("consume")
13   instMg:setDataCollector(true)
14
15   instMg:setCollectInterval(100)
16
17   local p = {className="",policyName="", policyDesc = "",
18      id = 0, args = {""}}
19
20   local m = {name="CPUUsage",
21     description="Tempo de Uso da CPU, desde a ultima medicao (ms)",
22     value="", type="CONTAINER", policy=p ,
23     className="scs.instrumentation.metrics.CPUUsageCollector"}
24   instMg:addMetric(m);
25
26   ...
27
28 end
```

Listagem 6.3: Método que Inicializa a Instrumentação de um *Container*

6.1.3 Resultados

Para verificar a sobrecarga imposta ao sistema executamos o exemplo do produtor consumidor com a monitoração ativa, coletando todas as informações apresentadas na figura 5.2. Em seguida fizemos o mesmo com todos os serviços de instrumentação desativados. Também realizamos testes com a monitoração coletando dados somente pelos interceptadores e em seguida somente pelo *daemon* dos *containers*, buscando quais métricas são mais

custosas de coletar. Confrontando os dados obtidos podemos avaliar como a arquitetura de monitoração interfere no processamento do sistema.

Inicialmente executamos o teste passando para o mediador um vetor de tarefas com 125 inteiros, cada inteiro tinha o valor de 50.000, o que irá gerar 250 trocas de mensagens entre mediador, consumidor e produtor. Em seguida executamos o teste passando para o mediador vetores de tamanho 250, 375 e 500 tarefas. Como uma das formas pela qual realizamos a coleta de informações é através do uso de interceptadores, o aumento no número de mensagens trocadas irá impactar na sobrecarga imposta ao sistema, com esse teste podemos observar como a sobrecarga cresce à medida que o número de mensagens aumenta. Para garantir a qualidade dos valores levantados executamos cada teste 30 vezes [29] [30] para cada tamanho do vetor de tarefas. Para plotarmos o gráfico, calculamos as médias dos valores coletados em cada configuração e mostramos o intervalo de confiança (95%) para as medidas. O resultado pode ser visto na figura 6.2.

O *cluster* utilizado para realizar os testes não é de uso exclusivo, logo, outros usuários podem utilizá-lo durante a execução do teste. Durante o teste com instrumentação com o vetor de tarefas com 125 posições, uma das 30 medidas coletadas ficou muito acima da média, o mesmo ocorreu no teste com o vetor de tarefas com 500 posições. Por isso o intervalo de confiança percebido nesses casos ficou maior do que nos demais.

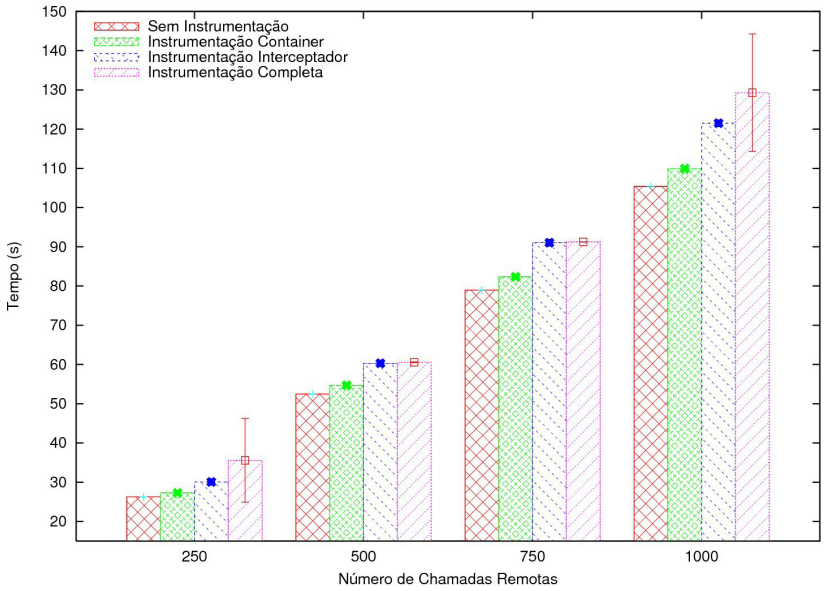

Figura 6.2: Tempo Médio das Execuções do Produtor Consumidor

Analisando o gráfico vemos que a sobrecarga imposta ao sistema se manteve em aproximadamente 15%, quando comparamos as execuções com e sem

instrumentação. Isso mostra que a sobrecarga percentual se mantém à medida que o número de mensagens trocadas vai aumentando. Nesse exemplo, para percebermos a sobrecarga gerada, montamos uma aplicação focada na troca de mensagens entre os componentes. Os métodos do produtor e consumidor usam pouco processamento, o que aumenta a disparidade entre os tempos coletados com e sem instrumentação. Caso aumentemos a quantidade de trabalho realizada pelo consumidor e pelo produtor, essa diferença percentual entre os valores coletados tende a diminuir. A figura 6.3 mostra resultados coletados utilizando o mesmo exemplo, agora cada posição do vetor de tarefas foi preenchida com o valor 100.000, o que irá gerar um maior processamento no produtor e no consumidor. Nesse caso a diferença percentual se mantém em aproximadamente 8%. Durante toda execução do teste outros usuários não utilizaram o *cluster*, logo, não existiu concorrência pelos recursos e não obtivemos nenhuma medida exageradamente fora da média, o que explica os pequenos intervalos de confiança.

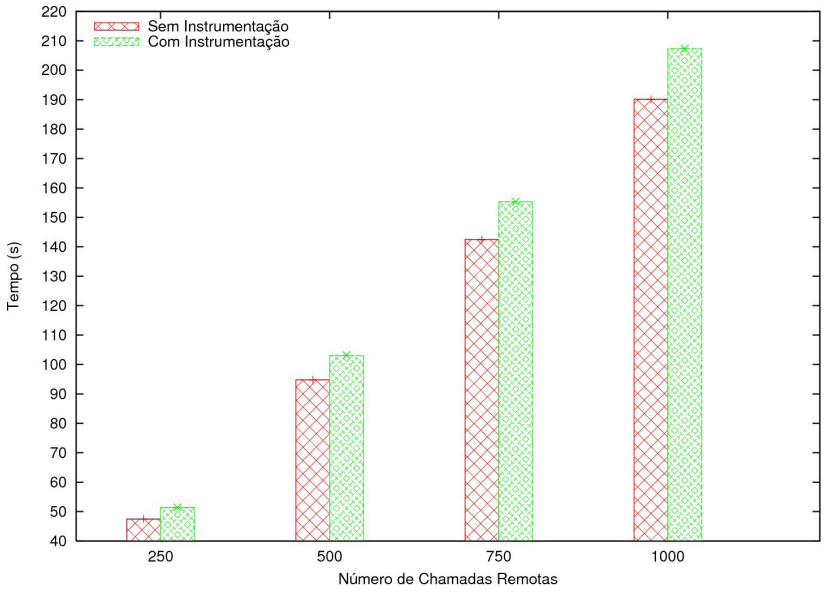

Figura 6.3: Tempo Médio para Tarefas de Tamanho 100.000

Se analisarmos as barras que representam os tempos gastos com instrumentação completa e com o uso apenas de interceptadores (figura 6.2), fica claro que o grande responsável pela sobrecarga no sistema são as métricas coletadas pelos interceptadores. Logo o uso da arquitetura de monitoração deve ser avaliado de acordo com o tipo de sistema que deseja-se monitorar. Caso o sistema distribuído possua uma alta taxa de troca de mensagens e baixo processamento em seus nós cabe uma análise de quais métodos realmente precisam ser monitorados evitando assim uma sobrecarga desnecessária. Caso contrário, ou seja, um sistema com grande carga de processamento e baixa troca de men-

sagens, as métricas coletadas pelo *container* devem ser reduzidas, evitando que o processamento realizado para coleta das métricas concorra com o processamento da aplicação.

Outra medida que coletamos durante o teste foi o número de mensagens de publicação (figura 6.4) geradas durante a execução de cada teste. Esses valores nos mostram que mesmo com um número superior de mensagens geradas no exemplo das métricas coletadas pelo *container* (227980 mensagens), essa execução foi mais rápida se comparada com a que usou somente as métricas dos interceptadores (180120 mensagens). Isso mostra que o uso do canal de eventos como solução para publicação dos valores coletados impõe um impacto pequeno no desempenho da aplicação.

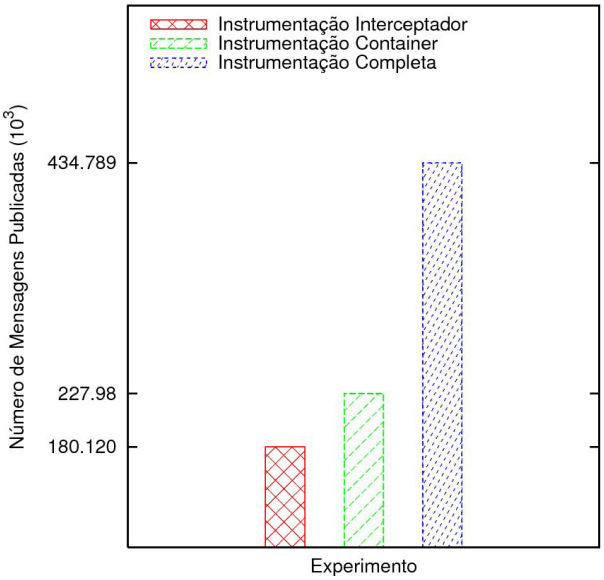

Figura 6.4: Número de Mensagens Publicadas em Cada Tipo de Intrumentação

6.2 Uso das Políticas

Para exemplificarmos o uso das políticas implementadas utilizamos a aplicação do produtor consumidor. Para cada política testada executamos o método *start* três vezes, passando um vetor de tarefas para o mediador com os valores: 100.000, 50.000 e 1. Como agora estamos interessados no comportamento das políticas e não das métricas, ativamos a coleta somente de uma métrica, a porcentagem de uso da CPU, e apenas no *container* onde está executando o produtor. O tempo entre as coletas realizadas no *daemon* do *container* foi ajustado para 2 segundos.

Inicialmente utilizamos as políticas AACD e ADI (seção 5.4.4). A figura 6.5 mostra os resultados obtidos com o uso da política AACD associado à métrica. A segunda coluna apresenta os resultados coletados pelo *daemon* do *container*. A coluna diferença mostra a quanto variou o valor coletado desde a última medição. A coluna *d_threshold* mostra o valor dinâmico calculado pela política, e na última coluna temos o valor publicado.

No uso dessa política o valor calculado dinamicamente sempre aumenta, pois, ele representa a média das diferenças dos valores anunciados, sendo que um valor só é anunciado quando a variação entre o valor coletado e o último publicado for maior que o *d_threshold*. Como o produtor inicia com uma tarefa grande, 100.000, o uso de CPU aumenta rapidamente no início levando o *d_threshold* para cima. Isso fez com que a política não publicasse uma série de coletas, linhas 5 a 13, da figura 6.5.

	Coletado	Diferença	d_threshold	Publicado (AACD)
1	0,017258			
2	0,023739	0,006480963	0,006480963	0,023738872
3	0,288815	0,265076607	0,135778785	0,288815479
4	0,019352	0,269463762	0,180340444	0,019351717
5	0,088977	0,06962505	0,180340444	
6	0	0,088976767	0,180340444	
7	0,001994	0,001994018	0,180340444	
8	0,003976	0,001982125	0,180340444	
9	0,183724	0,17974827	0,180340444	
10	0,125811	0,05791313	0,180340444	
11	0	0,125811283	0,180340444	
12	0	0	0,180340444	
13	0	0	0,180340444	
14	0,276342	0,276341948	0,19950289	0,276341948
15	0,003931	0,272410744	0,214084461	0,003931204
16	0,074766	0,070835151	0,214084461	
17	0,003982	0,070784275	0,214084461	

Figura 6.5: Resultados da política AACD

Agora executamos o mesmo exemplo, porém, dessa vez usando a política ADI (figura 6.6). A coluna DTI mostra o valor (em milissegundos) calculado dinamicamente pela política e a coluna *Timer* mostra o tempo decorrido (em milissegundos) desde o último anúncio. Como o valor coletado muda com bastante frequência o DTI calculado se mantém pequeno o que gerou poucas omissões, linhas 5,8,11,14 e 17.

Por fim avaliamos o uso da política híbrida ACTC (figura 6.7), definindo um *min_threshold* de 0,001. As colunas *Publica (AACD)* e *Publica (ADI)* indicam se o valor coletado seria ou não publicado com o uso dessas métricas.

	Coletado	DTI	Timer	Publicado (ADI)
1	0,017982018			
2	0,003972195	2004	0	
3	0,212864415	2083	2162	0,212864415
4	0,087037037	2108	2159	0,087037037
5	0,091089109	2086	2021	
6	0	2069	4024	0
7	0,009564802	2073	2092	0,009564802
8	0,00198906	2064	2010	
9	0,175736395	2056	4013	0,175736395
10	0,114122682	2062	2105	0,114122682
11	0,001981179	2057	2017	
12	0,003992016	2052	4021	0,003992016
13	0	2053	2063	0
14	0,185587364	2051	2027	
15	0,0499002	2048	4030	0,0499002
16	0,022695035	2052	2115	0,022695035
17	0,001993024	2049	2007	

Figura 6.6: Resultados da política ADI

Ao contrário do que ocorre quando usamos a política AACD, nesse caso o *d_threshold* pode diminuir como ocorre entre as linhas 7 e 8. Isso devido a ocorrência de uma publicação feita pela política ADI e o valor publicado ter sido menor do que o *d_threshold*. Na linha 6, embora a política ADI indicasse que a métrica deveria ser publicada, isso não ocorreu. Isso acontece, pois, o último valor anunciado foi 0, linha 4, e caso ocorresse esse anúncio o valor seria 0 novamente. Como o *min_threshold* foi configurado para 0,001, a métrica não foi publicada.

	Coletado	Diferença	d_threshold	DTI	Timer	Publica (AACD)	Publica (ADI)	Publicado (ACTC)
1	0,018682858							
2	0,007633588	0,01104927	0,011049271	2086	0	SIM	NÃO	0,007633588
3	0,314299658	0,30666607	0,158857671	2068	2050	SIM	NÃO	0,314299658
4	0	0,31429966	0,210671667	2054	2027	SIM	NÃO	0
5	0,086656819	0,08665682	0,210671667	2048	2031	NÃO	NÃO	
6	0	0,08665682	0,210671667	2039	4034	NÃO	SIM	
7	0,00397812	0,00397812	0,210671667	2034	2012	NÃO	NÃO	
8	0,1234445	0,11946638	0,188864875	2031	4022	NÃO	SIM	0,1234445
9	0,081960627	0,04148387	0,159388674	2088	2488	NÃO	SIM	0,081960627
10	0,138957816	0,05699719	0,159388674	2080	2014	NÃO	NÃO	
11	0	0,13895782	0,146484	2072	4017	NÃO	SIM	0
12	0,005813953	0,00581395	0,146484	2071	2066	NÃO	NÃO	
13	0,176258993	0,17044504	0,15073757	2084	4289	SIM	SIM	0,176258993
14	0,015429122	0,16082987	0,151999108	2083	2073	SIM	NÃO	0,015429122
15	0,086359176	0,07093005	0,151999108	2080	2038	NÃO	NÃO	
16	0	0,08635918	0,136824665	2075	4046	NÃO	SIM	0
17	0,001981179	0,00198118	0,136824665	2071	2019	NÃO	NÃO	

Figura 6.7: Resultados da política ACTC

6.3 Map Reduce

Com o objetivo de validar o uso da arquitetura implementamos uma aplicação utilizando um estilo arquitetural para programação distribuída chamado *MapReduce* [5]. Implementamos uma aplicação baseada nesse modelo e realizamos as medições da sobrecarga imposta pela arquitetura na execução do sistema. A versão do *MapReduce* apresentada nessa tese passou por algumas versões; em [31] apresentamos um estudo da arquitetura proposta nesse trabalho em uma das suas versões preliminares.

6.3.1 Descrição

O *MapReduce* trata-se de um estilo arquitetural voltado para o processamento de grandes volumes de dados, que facilita a distribuição do processamento desses dados em um grande conjunto de máquinas. Atualmente existem *frameworks* [32] que oferecem serviços para implementação de sistemas segundo a arquitetura do *MapReduce*. Tal framework baseia-se em dois conceitos provenientes de linguagens funcionais para expressar algoritmos que fazem uso intensivo de dados: funções de mapeamento (map) e redução (reduce). A função de mapeamento processa um arquivo de entrada e gera um conjunto de pares $<chave/valor>$ intermediários. A função de redução, então, combina os pares intermediários que possuem a mesma chave. Como essa arquitetura define bem os papéis e o relacionamento entre cada componente, o processo de distribuição dessa aplicação em diversas máquinas fica facilitado.

Na estrutura de execução do *MapReduce* (figura 6.8) destacamos o papel de dois componentes, são eles: o *Master* e o *Worker*. O primeiro é responsável por coordenar a execução dos diversos *Workers*, encarregando-se de garantir que cada um receba o conjunto de pares $<chave/valor>$ que deve tratar. Já os *Workers* são os responsáveis por realizar as funções de *Map* e *Reduce* implementadas pelo usuário. Inicialmente o *Master* deve dividir o arquivo inicial em partes menores e atribuir a cada worker o processamento de um desses subconjuntos de dados. Cada *Worker* recebe sua parte e gera um conjunto de arquivos intermediários que serão entregues a novos *Workers*, que dessa vez irão realizar a operação de *reduce* sobre os dados. Ao final são gerados arquivos com os resultados produzidos após cada *reduce*.

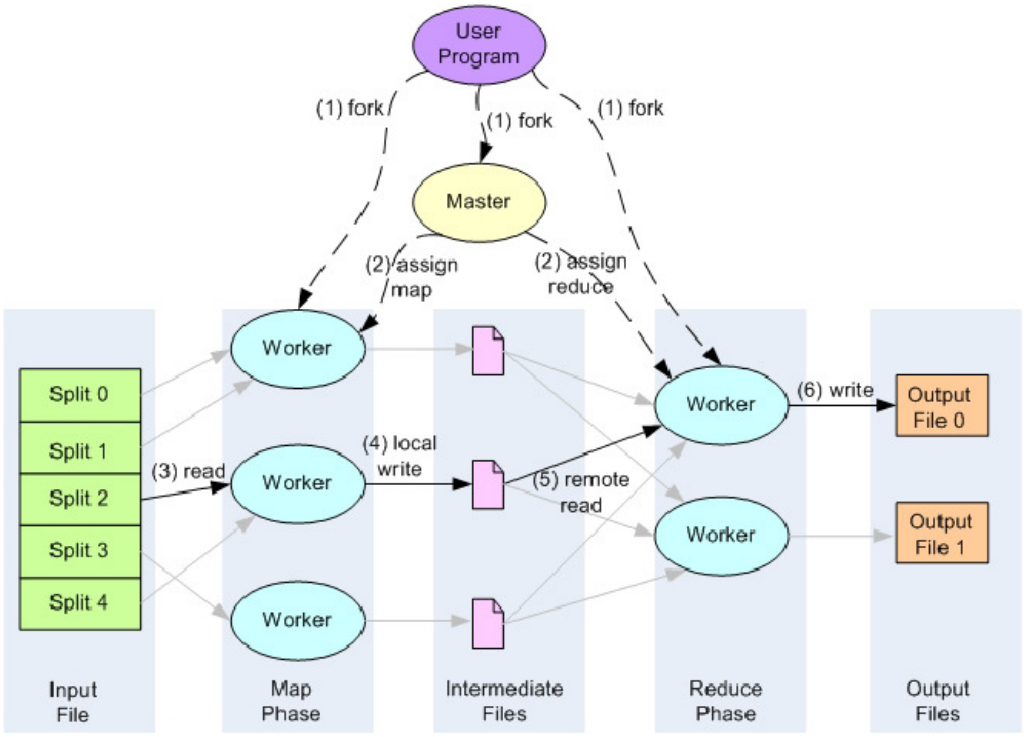

Figura 6.8: MapReduce [5]

Um dos usos dessa arquitetura é na contagem de ocorrências de palavras, ou conjunto de palavras, em grandes arquivos de *log* gerados em servidores *web*. Para esse exemplo as funções *Map* e *Reduce* produzidas poderiam ser as representadas pelo pseudo-código na listagem 6.4.

```
1
2   map(String key, String value):
3   // key: document name
4   // value: document contents
5     for each word w in value:
6     EmitIntermediate(w, "1");
7
8   reduce(String key, Iterator values):
9   // key: a word
10  // values: a list of counts
11    int result = 0;
12    for each v in values:
13      result += ParseInt(v);
14    Emit(AsString(result));
```

Listagem 6.4: Pseudo-Código *MapReduce*[5]

A função *map* recebe o par <*chave/valor*> contendo o nome do documento e conjunto de palavras que o compõem. Para cada palavra encontrada ele gera um par <*chave/valor*> onde a chave é a palavra encontrada e o valor é o número 1. A função *reduce* irá receber os pares e somar todos os valores encontrados para cada chave, gerando no final o total de ocorrências daquela chave.

6.3.2
Implementação

A figura 6.9 ilustra a implementação de um *framework* MapReduce usando componentes SCS. De maneira geral, o *framework* é composto por dois componentes principais: *Master* e *Worker*. O componente *Master* executa em um *container* próprio, controlando todo o fluxo de execução da aplicação, escolhendo *workers* desocupados e atribuindo a estes funções de mapeamento ou redução. Este componente implementa uma faceta *Master*, através da qual usuários submetem trabalhos (listagem 6.5, linhas 9-16). Para tanto, é necessário invocar o método *submitJob* fornecendo como parâmetro o arquivo de configuração do trabalho. Este arquivo define propriedades como: localização do arquivo de entrada, tamanho (em bytes) da partição na qual o arquivo será fragmentado, identificação dos nós *master* e *workers* e outras.

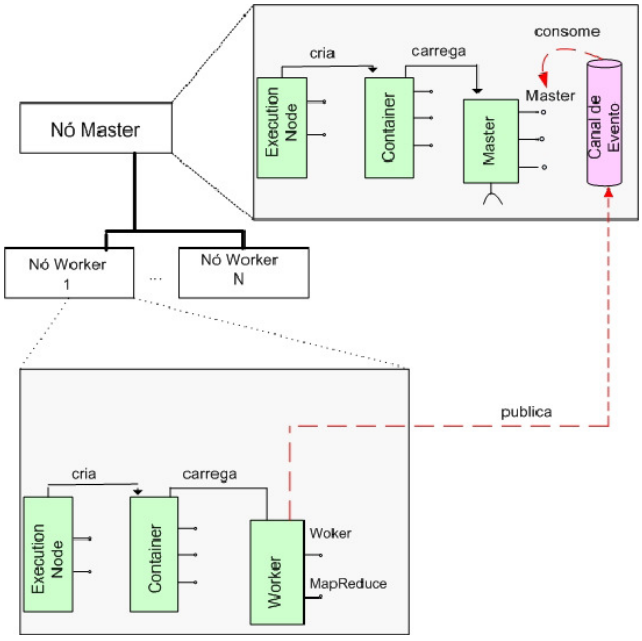

Figura 6.9: *Famework* MapReduce Implementado com Componentes SCS

Componentes *Workers* são instanciados em diferentes nós em *containers* individuais e executam funções de mapeamento ou redução definidas pelo usuário. Um componente *Worker* oferece duas facetas: *Worker* e *MapReduce* (listagem 6.5, linhas 18-24 e 26-29, respectivamente). A primeira oferece serviços para configuração e gerenciamento do *worker* (como, por exemplo, um método *ping* para verificar se o *worker* está ativo), bem como um método *execute*, através do qual o *worker* recebe a descrição de uma tarefa. A segunda faceta cria a tarefa a partir da descrição recebida e a executa efetivamente. A interface *TaskDescription* (listagem 6.5, linhas 1-7) representa a descrição de

uma tarefa. Ao final da execução de uma tarefa, o *worker* responsável por sua execução publica em um canal de eventos a identificação da tarefa seguida do *status* de término: completada com sucesso ou interrompida devido a erros. Estes eventos são posteriormente tratados pelo *master*.

A figura 6.10 ilustra as principais classes e interfaces Java que implementam o *framework*. A Classe *MasterServant* implementa a interface *Master*. Adicionalmente, três classes compõem *MasterServant*: *WorkerLauncher*, *Dispatcher* e *EventSink*. A primeira é responsável por instanciar os componentes *workers* e as descrições das tarefas a serem realizadas. *Dispatcher* é responsável pela atribuição de tarefas, enquanto que *EventSink* trata os eventos gerados pelos componentes *workers*. As classes *WorkerServant* e *MapReduceServant* implementam as interfaces *Worker* e *MapReduce* respectivamente. De acordo com a descrição recebida, a classe *MapReduceServant* executa uma tarefa *MapTask*, que implementa toda a fase de mapeamento, ou uma tarefa *ReduceTask*, a qual implementa a fase de redução.

```
1   interface TaskDescription {
2      long getId();
3      void setStatus(in TaskStatus status);
4      TaskStatus getStatus();
5      StringSeq getInput();
6      StringSeq getOutput();
7   };
8
9   interface Master {
10     void submitJob(in string confFileName)
11          raises (PropertiesException, ConnectionException,
12          ChannelException,
13        WorkerInstantiationException,
14        TaskInstantiationException,
15        SchedulerException);
16  };
17
18  interface Worker{
19    boolean start(in string configFileName, in string nodeName,
20      in scs::org::omg::CosEventChannelAdmin::EventChannel channel);
21    void execute (in TaskDescription t);
22    string getNode();
23    boolean ping();
24  };
25
26  interface MapReduce {
27    void map (in TaskDescription t) raises (IOMapReduceException);
28    void reduce (in TaskDescription t) raises (IOMapReduceException);
29  };
```

Listagem 6.5: Interfaces IDL Usadas no *Framework* MapReduce

Na fase de mapeamento, os dados provenientes de uma partição do arquivo de entrada são formatados para registros intermediários usando-se pares *chave-valor* iniciais. A classe *RecordReader* é responsável por tal

formatação. Em seguida, um objeto do tipo *Mapper*, definido pelo usuário, reprocessa os pares gerados pelo *RecordReader*, mapeando-os para novos pares *Chave-Valor*, usando a função *map*. O par gerado é então coletado por um objeto do tipo *Collector*. Ao final da fase, estes pares são distribuídos e ordenados em função dos valores contidos em suas chaves e os grupos obtidos são gravados em arquivos de dados intermediários.

Na fase de redução, os grupos gerados na fase anterior são lidos e reprocessados por um objeto do tipo *Reducer*, definido pelo usuário, usando-se o método *reduce*. Este método gera os pares *Chave-Valor* finais que são coletados por um objeto do tipo *Collector*. Esse coletor persiste os pares em disco, gerando os aquivos de saída. A persistência é implementada pela classe *RecordWriter*.

6.3.3 Resultados

Para realizar o teste com o *MapReduce*, instanciamos o *framework* para contar o número de ocorrências de cada palavra em um determinado arquivo, simulando os arquivos de *log* encontrados nos servidores *web*. Para o teste realizado escolhemos um arquivo de 100 *MB* e utilizamos 5 máquinas com processadores *3 GHz Intel Pentium IV* outra com 1 *GB* de memória. Inicializamos o *Master* em uma máquina, três workers em máquinas distintas e os canais de eventos e *StatsCollection* em uma máquina a parte.

Assim como no caso do produtor consumidor executamos o *Mapreduce* 30 vezes com instrumentação e 30 vezes sem. Os resultados foram plotados em um gráfico e o resultado está apresentado na figura 6.11, que mostra o intervalo de confiança (95%) da amostra. Como no caso do *MapReduce* existe uma grande quantidade de escrita e leitura em disco, e não existem muitas trocas de mensagens a sobrecarga imposta foi de aproximadamente 4%.

A configuração da estrutura de monitoração pode ser feita de forma independente da construção dos componentes que compõem a aplicação, bastando para isso conhecer os *ExecutionNodes* responsáveis por inicializar os *containers* da aplicação. Dessa forma a arquitetura de monitoração fica transparente para o desenvolvedor dos componentes. A sobrecarga da estrutura de monitoração não se mostrou como um impeditivo para execução da aplicação. Assim como ocorreu com o caso do produtor consumidor, não foi necessário alterar o código

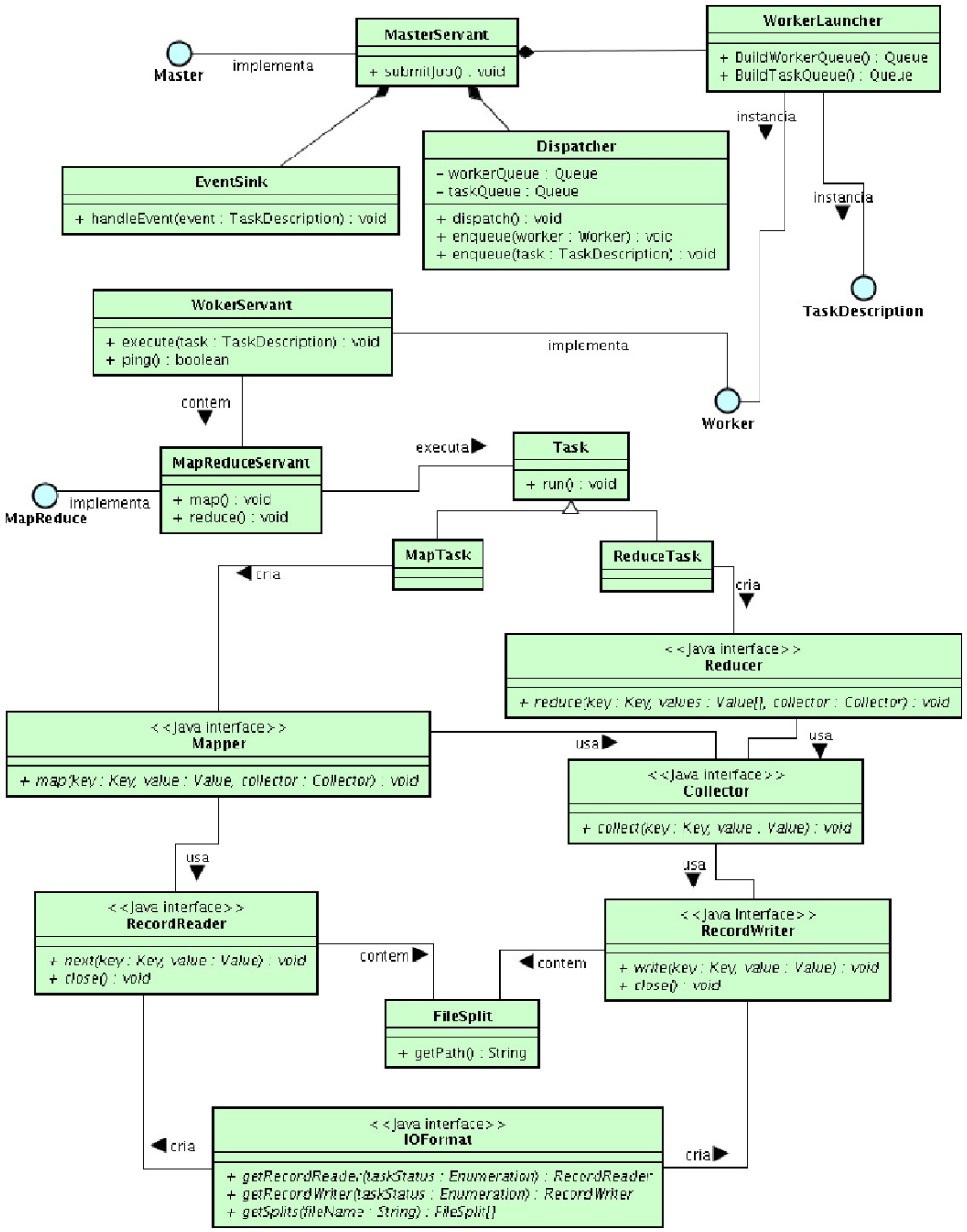

Figura 6.10: Diagrama de Classes para o *Framework* MapReduce

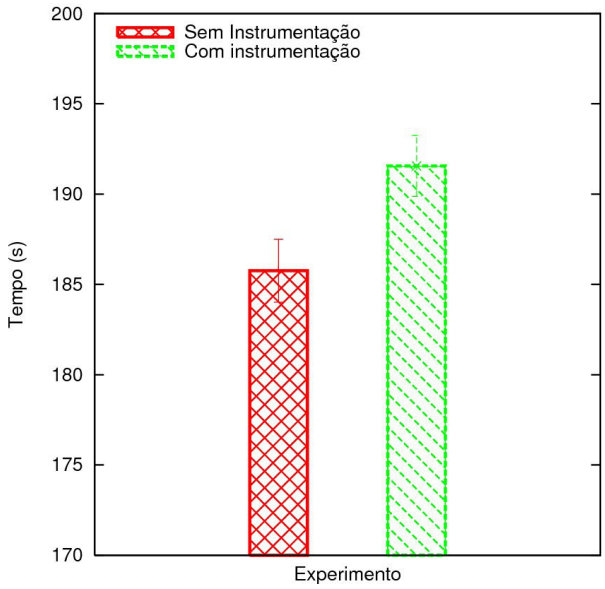

Figura 6.11: Tempos Coletado no *MapReduce*

dos componentes da aplicação para realizar a coleta de informações.

Em [12], os dados coletados a partir da arquitetura de monitoração implementada nesta dissertação são usados para alimentar um modelo de estatísticas capaz de inferir violações de SLO (*Service Level Objective*). Dessa forma através da análise dos dados coletados é possível localizar a causa de problemas de desempenho do sistema, e a partir disso gerar modificações que possam melhorar esse desempenho.