Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
Sistemas de Componentes de Software
Prof. Renato Cerqueira

Segundo Trabalho

Mecanismo de Tolerância a Falhas para a Implementação de Map-Reduce do SCS

2008/02

Bruno de Figueiredo Melo e Souza
Ernesto Augusto Thorp de Almeida
Rafael Silva Pereira

1. Introdução

Para ser suficientemente robusta e confiável, uma aplicação cuja premissa consiste em executar diversas tarefas simultaneamente em diversos locais, de forma distribuída, necessita de um mecanismo bastante eficiente de detecção e recuperação de falhas, de modo que a indisponibilidade de um nó não comprometa todo o sistema. Isto porque, de maneira geral, não podemos garantir que todos os nós irão executar as tarefas designadas de forma correta, sem que haja uma interrupção, intencional ou não, do processo. Problemas de comunicação entre os nós, problemas de hardware em um dos nós, queda de energia ou até mesmo uma falha de software que ocasione a queda do processo, podem impedir que uma determinada tarefa seja executada em um nó, o que, se não houver um plano de detecção e recuperação, pode gerar uma saída inconsistente ou até mesmo pode derrubar todas as demais tarefas em execução.

Uma aplicação onde a queda de um nó teria um grande impacto no resultado é o Map-Reduce. Nele, uma determinada entrada é quebrada em diversas partes, sendo que cada parte é inicialmente processada em um nó, gerando um mapeamento chave-valor. Esta é a etapa de mapping. Em seguida, as saídas do mapping são processadas nos reducers, para que os resultados possam ser consolidados. *(Figura 1)*

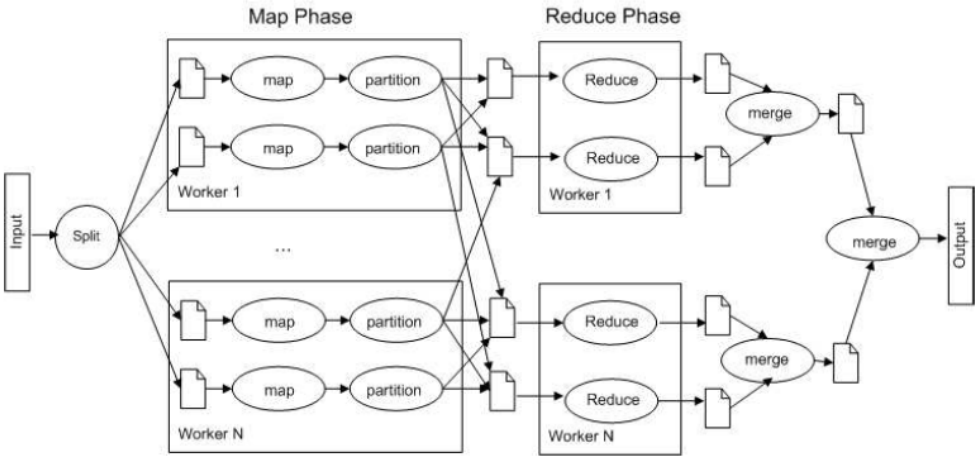

Figura 1: O processo de Map-Reduce

Como podemos ver, caso um nó que esteja realizando uma tarefa de mapping caia, então nosso mapeamento chave-valor estará incompleto, o que irá gerar uma consolidação inconsistente nos reducers. Da mesma forma, a queda de um reducer irá causar uma inconsistência na consolidação.

Desta forma, para que uma aplicação distribuída seja tolerante a falhas é imprescindível que seja realizada a detecção e recuperação de possíveis problemas, o que significa, de forma mais genérica, redistribuir nos nós ativos as tarefas anteriormente designadas à nós que apresentaram algum tipo de comportamento não desejado.

Considerando a grande importância de um mecanismo de tolerância a falhas eficiente, temos como objetivo implementar uma solução capaz de garantir uma maior robustez e capacidade de recuperação à aplicação de Map-Reduce disponibilizada pela distribuição do middleware SCS. Nesta implementação temos um nó Master que distribui as tarefas para outros nós, chamados de Workers, responsáveis pela execução das etapas de mapping e reduce.

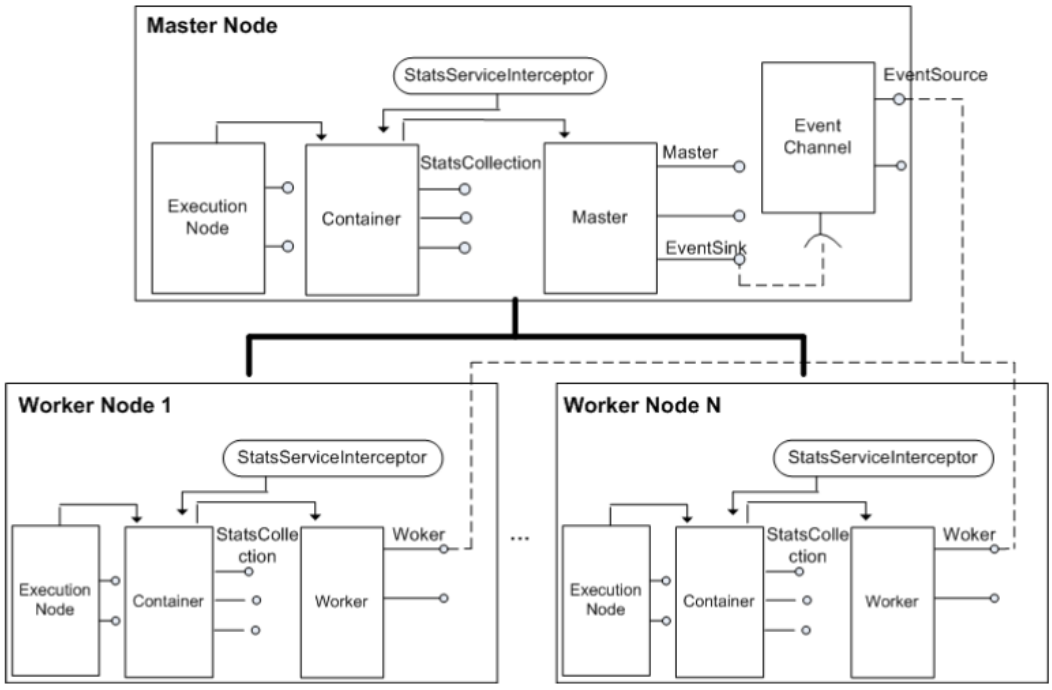

Figura 2: O Map-Reduce representado por componentes SCS

Basicamente, temos uma aplicação (MapReduceApp) que instancia um componente SCS MasterComponent, que inicializa os componentes Worker (WorkerComponent) nos nós do cluster, incluindo-os em uma lista de workers disponíveis, e também cria uma lista de tarefas a serem exeutadas. Em seguida, é criado um canal de eventos (EventChannel) para efetuar a comunicação entre o nó master e os workers, de modo que, ao finalizar uma tarefa, o worker lança um evento neste canal, que é capturado pelo nó master. Finalmente cada tarefa da lista de tarefas é designada e executada por um Worker disponível na lista de workers, concluindo assim o processo de map-reduce.

A seguir, iremos ver o que esperamos de um mecanismo de tolarância a falhas em sistemas distribuídos, e o que como incorporar este mecanismo na implementação de Map-Reduce do SCS.

2. Tolerância a Falhas em Sistemas Distribuídos

Como vimos anteriormente, para que um sistema distribuído seja suficientemente robusto é fundamental que sua arquitetura permita a indisponibilidade de um ou mais nós sem comprometer as funções para as quais o sistema se propõe à executar. Assim, durante o processo de desenvolvimento devemos nos preocupar em como identificar e tratar eventuais indisponibilidades, independentemente da etapa de processamento em que elas venham a ocorrer.

Um mecanismo eficiente de tolerância a falhas deve ser capaz de detectar qualquer tipo de comportamento adverso, e, além disso, deve tomar as ações necessárias para que a cadeia de processamento siga o seu fluxo sem interrupções, garantindo ainda a consistência das informações. Em outras palavras, isto significa que o próprio sistema deve decidir como recuperar o trabalho que estava sendo executado por um determinado nó.

No caso do Map-Reduce, por exemplo, devemos distribuir entre os nós ativos, as tarefas que estavam sendo processadas pelos nós que apresentaram problemas, sendo que, em uma situação ideal, as tarefas seriam apenas continuadas em outros nós, o que significa que o trabalho já executado não seria reinicializado [4].

Uma abordagem mais sofisticada de tolerância a falhas poderia executar estas tarefas de forma adaptativa, ou seja, permitindo que o mecanismo altere a tolerância a falhas do sistema através da adaptação a mudanças no ambiente computacional ou nas políticas de tolerância a faltas [1]. O modelo TFA-CCM - Tolerância a Faltas Adaptativa no Modelo de Componentes CORBA, por exemplo, fornece suporte a tolerância a faltas adaptativa totalmente transparente à aplicação sem a necessidade de mudanças no modelo de componentes e na sua implementação [2].

Baseado nestes conceitos, para que um mecanismo de tolerância a falhas mínimo seja implementado na aplicação de Map-Reduce do SCS, devemos construir soluções capazes de:

- Identificar a queda de um nó, ou seja, a indisponibilidade de acesso ao Execution Node;
- Identificar a queda de um Worker em um determinado nó, o que significa, por exemplo, um problema com o container onde o componente está instanciado;
- Identificar problemas na execução das tarefas, seja por problemas com a Thread responsável pela execução, seja por problema de acesso aos splits de arquivos, etc;
- Recuperar um container/worker em caso de falha;
- Recuperar e reagendar a tarefa em caso de falha;
- Excluir um worker ou nó com comportamento atípico;

De acordo com estes requisitos, o ideal seria construir um componente isolado de monitoração, capaz de detectar problemas não somente nos Workers, mas também no Master. Entretanto, esta abordagem exigiria uma alteração substancial na implementação existente, com a criação de novas interfaces e métodos que permitissem o acesso e o gerenciamento das filas, além do desenvolvimento do próprio componente de monitoração. Assim, optamos inicialmente por uma abordagem menos flexível, incluindo o mecanismo de monitoração no próprio componente Master.

Para garantir uma monitoração periódica, isolamos o mecanismo de monitoração em uma thread específica, que é inicializada durante o startup do componente Master, com o objetivo de mantê-lo ativo continuamente durante todo o processo de Map-Reduce.

A seguir, iremos detalhar cada um dos mecanismos de detecção e monitoração de falhas implementados.

3. Mecanismos de Detecção de Falhas

O processo de identificação de falhas deve levar em consideração os diversos níveis de indisponibilidade citados anteriormente, mas também deve ser suficientemente inteligente para detectar comportamentos que possam indicar problemas na execução de tarefas. Assim, devemos verificar a integridade de todos os níveis de processamento envolvidos na aplicação, desde o Execution Node até a thread que efetivamente executa a tarefa.

Uma verificação do Execution Node nos garante que o ambiente de execução está íntegro, ou seja, todos os componentes de conectividade, hardware e as camadas de mais baixo nível, como o Sistema Operacional, estão minimamente funcionais. Entretanto, não temos a garantia de que todos os requisitos necessários para a execução de uma tarefa estão disponíveis. Por exemplo, um mount NFS, necessário para que uma determinada operação seja executada, pode não estar devidamente configurado, ou um componente que provê serviços necessários para execução de uma tarefa pode não estar devidamente instanciado, etc. Assim, a validação dos Execution Nodes é o mecanismo mais básico de detecção de falhas, principalmente àquelas relacionadas ao ambiente de execução.

No caso da aplicação de Map-Reduce, por usar o middleware SCS, podemos utilizar de algumas facilidades para implementar esta verificação. Como temos no nó Master uma lista com todos os nós envolvidos no processamento, podemos simplesmente percorrê-la verificando se conseguimos ou não obter uma referência para o nó em questão. Abaixo podemos ver o trecho de código que realiza este procedimento:

```
//Checking if all Execution Node are UP / Reachable
reporter.report(1,"Checking Nodes");
for(int i = 0; i < execNodeList.length; i++){
    ExecutionNode execNode=initializer.getNode(execNodeList[i]);
    if (execNode == null) {
        //Node Down
        reporter.report(1,"Execution Node " + execNodeList[i]
                + " is unreachable. Rescheduling its tasks!");

         //Get Nodes's tasks to requeue
         Enumeration<Task> enu = (master.getWorkingOn()).keys();
         while (enu.hasMoreElements()) {
             task = (Task) enu.nextElement();
             if(task.getNode().equals(execNodeList[i])) {
                 ....
             }
         }
    }
}
```

Mecanismo de Detecção de falhas nos Execution Nodes

O método *getNode* acima acessa a faceta *"scs::execution_node::ExecutionNode"* do Execution Node, de modo que, caso o mesmo esteja inacessível, uma exceção é

capturada, gerando uma mensagem de log, e forçando que o retorno da função seja null, o que, no mecanismo acima, é tratado como uma indisponibilidade. Neste caso, as tarefas que estavam sendo executadas no Execution Node que apresentou problemas devem ser identificadas e redistribuídas para os demais nós ativos. Este procedimento de recuperação será visto com maiores detalhes posteriormente.

Após verificar todos os nós do sistema o mecanismo de monitoração procura falhas nos componentes responsáveis pela execução das tarefas. Para isto, chamamos o método *ping()* da interface Worker do WorkerComponent. Esta interface verifica se a thread de execução das tarefas está efetivamente ativa. Um ponto importante é que está verificação só é realizada para os workers que possuem uma tarefa designada.

Esta é uma etapa fundamental para o processo de detecção de falhas pois ela pode identificar não apenas uma falha na thread de execução da tarefa, mas uma falha em todo WorkerComponent ou em seu container. Isto porque caso haja algum problema nestes dois pontos o método ping() não estará disponível, o que irá gerar uma exceção (*CORBA_COMM_FAILURE*). Desta forma, no tratamento desta exceção devemos verificar se o container está ativo, o que pode ser realizado através do método *getConainer()*, e se o WorkerComponent está devidamente instanciado, o que pode ser realizado através do método *getComponent()* da interface ComponentCollection do container. A seguir temos um trecho do código que realiza estas verificações:

```
Enumeration<Task> enu = (master.getWorkingOn()).keys();
while (enu.hasMoreElements()) {
    task = (Task) enu.nextElement();
    worker = (Worker) master.getWorkingOn().get(task);
    try {
        if(!worker.ping()) {
            ....
        }
    } catch (...) {
        ....
    }
```

Mecanismo de Detecção de falhas nos Workers

Apenas os mecanismos acima não nos garantem a detecção de todos os tipos de falha. Podemos ter algum problema na execução da thread da tarefa que faça com que a mesma fique executando indefinidamente. Um exemplo disto seria não termos dentro da implementação do processo de execução da tarefa um do seu tempo de execução para um procedimento específico. Em casos como este, o nó poderia demorar excessivamente para concluir a tarefa, ou até mesmo não concluí-la jamais.

Para isso, implementou-se um controle dos tempos de início e fim de execução de uma determinada tarefa de map ou reduce. A idéia é usar como parâmetro o tempo médio de execução das tarefas de mesma característica e definir um limite máximo de execução em cima dessa média.

```
interface Task {
            ...
            void setInitTime();
            long long getInitTime();
            void setKilled();
```

```
            boolean isKilled();
            };

...
Date now = new Date();
duration = now.getTime() - task.getInitTime();
...
case TaskStatus._MAP:
    // 1.2x threshold
    if ((duration > (1.2*meanMapTaskDuration)) &&
            (meanMapTaskDuration > 0)) {
        ...
    }
```

Mecanismo de Controle de Tempo de Execução

Para evitar estas circunstâncias, ou melhor, para detecta-las, poderíamos implementar uma esquema de timeout adaptativo para execução de tarefas, de modo a garantir que o tempo de execução de uma tarefa seja semelhante ao tempo que outros nós levaram para executá-la. Apesar de ser uma alternativa interessante, a utilização de timeouts em sistemas distribuídos é bastante questionável uma vez que nem sempre a disponibilidade de recursos é igual para todos os nós.

Outro ponto importante consiste na monitoração do nó Master. Da forma como os mecanismos foram implementados não é possível detectar uma falha no nó principal pois o próprio mecanismo de monitoração está integrado no mesmo. Neste caso, seria necessário a implementação de um componente independente, o que seria bastante desejável para garantir a integridade de todo o sistema.

Assim, uma vez que identificamos uma falha em um nó, ou na execução de alguma tarefa, devemos ter uma série de alternativas que nos permitam a recuperação e que nos garantam a execução de todo o processo. A seguir veremos alguns dos mecanismos implementados para estas situações.

4. Mecanismos de Recuperação

Os mecanismos de recuperação, acionados a partir da detecção de uma falha, são de fundamental importância para garantir o funcionamento do sistema, o processamento correto, e também para manter os níveis de serviço dentro de uma margem esperada. Sem eles, um determinado nó com problemas poderia ser simplesmente excluído, de modo que os demais nós seriam responsáveis por absorver as tarefas anteriormente designadas para o nó problemático. Numa situação como esta, o tempo total de execução de um procedimento poderá aumentar substancialmente a medida que temos um número razoável de falhas.

Entretanto, podem existir falhas cujo um mecanismo de recuperação não é capaz de lidar. Uma queda de energia, ou de conectividade, por exemplo, gera uma falha onde a única alternativa de recuperação automática está na tentativa periódica.

Com o objetivo de mantermos uma qualidade de serviço aproximadamente constante durante o processamento de uma tarefa, devemos implementar mecanismos eficientes para que, caso uma falha aconteça, o sistema seja prontamente recuperado. No caso específico do Map-Reduce disponível no SCS, isto significa reinicializar uma tarefa, um worker, um container ou até mesmo um Execution Node. Em todos estes casos, para que a tarefa volte a ser executada, seja pelo mesmo nó, seja por outro, devemos incluí-

la novamente na fila de tarefas a serem processadas (*taskQueue*).

Entretanto, durante os testes de recuperação identificamos que simplesmente recolocar a tarefa na fila não era o suficiente. Isto porque, no caso de o WorkerComponent ser indevidamente finalizado (com um kill por exemplo), a thread de execução poderá continuar ativa, mesmo estando orfã. Neste caso, ela continuaria escrevendo o seu arquivo de saída. Voltando esta tarefa para a fila de tarefas a serem processadas, um outro worker poderia tentar executar a mesma, o que geraria uma inconsistência dos dados, pois ambas as threads estariam escrevendo no mesmo arquivo, simultaneamente. A solução para este problema consiste na criação de uma nova tarefa, igual àquela que estava sendo executada, porém com id`s e arquivos de saída diferentes (*recreateTask*). Nesta situação, todo o output da tarefa inicial seria simplesmente ignorado. Abaixo podemos ver como a recuperação é realizada no monitor de falhas:

```
try {
    master.getWorkingOn().remove(task);
    task = initializer.recreateTask(task.getInput(),
                    task.getStatus(), task.getIndex(),
                        master.getInputToReducers());

    master.addTaskQueue(task);
    task.setNode("");
    task.setWorkerId(-1);
} catch (Exception ex) {
    reporter.report(1,"Error rescheduling task");
}
```

Mecanismo de Recuperação de Tarefas

O método recreateTask recria uma tarefa de acordo com as características da original, ou seja, se é uma tarefa de Map ou Reduce, com o mesmo conteudo de entrada, e com o mesmo índice, que é utilizado para as tarefas de reduce.

Além da recuperação de tarefas, no caso de falha do container ou do WorkerComponent, é necessário reinicializar parte do ambiente de execução. Nestes casos, devemos efetivamente nos conectar no execution node do nó defeituoso, e refazer o processo de startup do nó, recriando o container e reinstanciando o componente. Para isto, utilizamos os mesmos métodos de inicialização do sistema.

Uma vez criado o container e carregado o componente, incluimos o mesmo na fila de worker disponíveis, para que ele volte a processar as tarefas normalmente. Vejamos:

```
if (!initializer.createContainer(containerName, execNode)) {
    reporter.report(1,"Erro criando o container");
} else {
    container = execNode.getContainer(containerName);
    try {
        container.startup();
        Worker newWorker =
            initializer.reinitializeWorker(container);

        if(newWorker != null) {
            reporter.report(0,"Novo Worker Criado");
            newWorker.setNode(execNodeRef);
```

```
            newWorker.setId(workerId);
            workerQueue.add(newWorker);
        }

    } catch (Exception exc) {
        exception = LogError.getStackTrace(exc);
        reporter.report(0,"Erro no startup do worker no
             container " + containerName + ".\n" + exception);
    }
}
```

Mecanismo de Recuperação de Containers e Workers

Um ponto importante que devemos lembrar é que em todos os casos de falha o par tarefa/worker é retirado da lista de workingOn, para que então as devidas ações de recuperação sejam tomadas.

No caso de uma tarefa de map ou reduce estar levando muito tempo para se concretizar, ou seja, ultrapassando os limites máximos pré-estabelecidos, vale uma política de contorno baseada em abortar a execução no nó lento e tentar reagendar a mesma num nó com tempo de resposta menor. Ou até, diminuir a quantidade de tarefas simultâneas no mesmo nó visando uma serialização na execução das tarefas agregando com um tempo de processamento (ao invés de paralelizar com um aumento de tempo exponencial).

```
// 2x threshold
if ((duration > (2*meanReduceTaskDuration)) &&
(meanReduceTaskDuration > 0)) {
    // TODO: implement a better policy
    reporter.report (1,"WARNING: Reduce task (ID = " +
task.getId()         + ") running time on " + "execution node "
+ task.getNode()         + " is taking too long: " + duration +
" ms. Current mean: "         + meanReduceTaskDuration + "
ms.");
    task.setKilled();
    // Re-schedule task
    ...
    // Make container unavailable
    try {
        reporter.report(1,"Timed out!!! Stopping container");
        String execNodeRef = task.getNode();
        if (!masterHost.equals(execNodeRef)) {
            ExecutionNode execNode =
                initializer.getNode(execNodeRef);
            String containerName = MAP_REDUCE_CONTAINER +
                workerId;
            IComponent container =
                execNode.getContainer(containerName);
            execNode.stopContainer(containerName);
        }
    } catch (Exception ex) {
        reporter.report(1,"Error stopping container");
    }
}
```

Indo mais além, poderíamos ainda implementar um mecanismo de recuperação mais eficiente se houvesse uma persistência de estado das tarefas. Mantendo constatemente o estado de processamento de uma tarefa, em caso de falha, não seria necessário reinicializar todo o trabalho. Nesta circunstância poderíamos apenas retomar o trabalho interrompido em outro nó a partir do ponto onde houve a interrupção. Esta é uma funcionalidade fortemente desejável, já que reduz o tempo de recuperação em caso de falha.

Um outro ponto de recuperação importante está relacionado à falha do Execution Node, que pode cair sem que o ambiente de execução de baixo nível tenha sido afetado, ou seja, podemos ter uma falha no execution node mesmo que o hardware, o SO, e a rede, por exemplo, tenham apresentado algum problema. Neste caso, seria bastante interessante que o mecanismo de recuperação tentasse reinicializar o mesmo, o que significa acessar o host do nó em questão e inicializar a aplicação ExecutionNodeApp.

Finalmente, com uma detecção de problemas no nó master, seria necessário implementar um mecanismo capaz de recomeçar todo o processo de Map-Reduce, reiniciando o core da aplicação. Neste caso específico, a manutenção do estado da aplicação seria imprenscindível para reduzir os impactos da falha no tempo total de processamento.

Por restrições de tempo nem todas essas funcionalidades foram implementadas, porém, conseguimos atingir um nível de tolerância a falhas aceitável para o propósito desta implementação. Os resultados obtidos e os testes realizados com o mecanismo desenvolvido serão apresentados a seguir.

5. Resultados Obtidos

Anteriormente apresentamos os mecanismos de detecção e recuperação de falhas desenvolvidos para a aplicação de Map-Reduce do middleware SCS. Para validar seu funcionamento e eficiência realizamos uma série de testes, simulando diferentes tipos de falhas que podem ocorrer em um ambiente real.

Para verificar o mecanismo de detecção de problemas no execution node, geramos uma indisponibilidade em um dos nós, interrompendo a conectividade do mesmo, de modo que ele ficasse inacessível a partir do nó master. Neste caso, o mecanismo de detecção de falhas nos execution nodes identificou o problema e excluiu o nó em questão dos nós disponíveis para execução de tarefas. Além disso, a tarefa associada à ele foi recriada e incluída na fila de tarefas e serem processadas. Finalmente, um outro nó, ativo, obteve a tarefa referida na lista, e executou a mesma com sucesso, gerando o output esperado, idêntico àquele obtido em uma execução sem falhas. Este teste foi realizado nas diversas etapas do Map-Reduce.

Outro teste efetuado foi a simulação de falha no WorkerComponent, através da interrupção do processo relacionado ao mesmo. Neste teste, o Execution Node e o Container continuam íntegros, porém o componente não está mais disponível, o que gera uma exceção na chamada no método ping. Este teste foi realizado considerando dois cenários: um onde o processo relacionado ao componente era interrompido juntamente com a thread de execução de uma tarefa, e um onde a thread não era interrompida. Nos dois casos o mecanismo detectou a falha corretamente, e recuperou a tarefa, retornando-a para a fila, e o container/componente, recolocando o mesmo disponível para processamento.

Finalmente, simulamos uma falha na thread de execução das tarefas, de modo que o

método ping não identifica uma thread ativa. Neste caso, apenas o mecanismo de recuperação de tarefas é invocado, e todo o processamento transcorre normalmente.

Com estes testes fomos capazes de validar os mecanismos implementados, de modo que, mesmo em caso de falhas, a aplicação concluí suas operações de maneira correta, com um pequeno impacto no tempo de execução, causado justamente pela não manutenção do estado das tarefas.

6. Conclusão

Conforme foi possível verificar ao longo do processo de pesquisa e desenvolvimento dos mecanismos de tolerância a falhas descrito neste trabalho, a robustez e a confiabilidade de uma aplicação distribuída também estão fortemente relacionadas com a capacidade que a mesma possui em se recuperar de problemas inerentes ao ambiente distribuído. Em outras palavras, a distribuição de tarefas em diversos nós, para execução paralela, introduz uma complexidade bastante grande às aplicações, o que aumenta significativamente a possibilidade de ocorrência de problemas. Nestas situações, para evitar uma indisponibilidade geral da aplicação, é de fundamental importância que os componentes com problemas sejam excluídos ou recuperados, de modo que o resultado final esperado não seja impactado, seja em sua corretude, seja no tempo necessário para obtê-lo.

Por outro lado, um sistema distribuído com um bom mecanismo de tolerância a falhas integrado é, sem dúvida alguma, mais robusto que um sistema onde todas as tarefas são processadas em um único local, isto porque eliminamos um ponto único de falha. Entretanto, é bastante importante que o mecanismo citado consiga recuperar falhas não apenas nos nós secundários, mas também no nó principal.

Outra característica fundamental para minimizar os impactos de uma falha é a manutenção do estado de processamento de toda aplicação, o que permite que as tarefas sejam apenas retomadas, e não reiniciadas, em caso de falha.

Como resultado de todo este processo, conseguimos implementar os mecanismos básicos e imprescindíveis de detecção e recuperação de falhas, integrado à aplicação de Map-Reduce do SCS. Por restrições de escopo, necessárias em função da grande gama de mecanismos passíveis de implementação, não foi possível isolar os mecanismos de monitoração em um componente SCS, o que certamente permitira um reuso mais eficiente em outras aplicações. Além disso, alguns dos procedimentos de recuperação estão fortemente atrelados ao Map-Reduce, o que tornaria o esforço de isolamento das funcionalidades grande o suficiente para que a implementação da mesma fosse preterida.

Como trabalhos futuros, também incluímos a manutenção dos estados das tarefas, que, como foi dito ao longo do trabalho, é de suma importância para a manutenção da qualidade de serviço no que diz respeito ao tempo de execução. Além disso, a monitoração e recuperação do nó master é fortemente desejável, já que uma falha neste ponto geraria uma indisponibilidade da aplicação.

7. Referências

[1] Kim, K.H., Lawrence, T.(1990). *Adaptive Fault Tolerance: Issues and Approaches*. In 2nd IEEE Workshop on Future Trends of Distributed Computing Systems. p. 38-46, Cairo, Egypt. IEEE Computer Society.

[2] Fábio Favarim, Joni Fraga, Frank Siqueira. *Tolerância a Faltas Adaptativa em um Modelo de Componentes.* IV Workshop de Testes e Tolerância a Falhas, Universidade Federal de Santa Catarina, UFSC Florianópolis, SC, Brasil

[3] Fault tolerant Hadoop Job Tracker - http://issues.apache.org/jira/browse/HADOOP-4586

[4] Jeffrey Dean, Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters* - http://labs.google.com/papers/mapreduce-osdi04.pdf

[5] Map Reduce - http://www.cs.colostate.edu/~cs575dl/lects/MapReduce.ppt