Introdução de um mecanismo de *checkpointing* e migração em uma infra-estrutura para aplicações distribuídas

Jeane Cezário[1] e Alexandre Sztajnberg[1,2]

[1]Departamento de Informática e Ciência da Computação (DICC) / IME
[2]Programa de Pós-Graduação em Eletrônica (PEL) / FEN
Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro, RJ - Brasil

{jeane, alexszt}@ime.uerj.br

Abstract. *Highly distributed environments are characterized by the heterogeneity and availability variation of resources. Applications with non-functional requirements running in such environments require mechanisms to (i) monitor resource state and (ii) manage component distribution based on the state of the monitored resources in order to optimize its execution. The ZeliGrid middleware, a grid-based infrastructure, provides such mechanisms using contracts. In runtime ZeliGrid evaluates if a contract is being respected and decides if the diverse components remain executing in their current node or if they have to be reinitiated in another node, with more resources. In this work we introduce a mechanism for state persistence (checkpointing) and migration of the components, integrated to ZeliGrid, aiming to maintain the application running according to its non-functional requirements, with no need to reinitiate its components when a reconfiguration occurs.*

Resumo. *Ambientes altamente distribuídos são caracterizados pela heterogeneidade e variação da disponibilidade de seus recursos. Aplicações com requisitos não-funcionais executando em tais ambientes requerem mecanismos para (i) monitorar o estado dos recursos e (ii) gerenciar a distribuição de componentes, baseado no estado dos recursos monitorados, com o objetivo de otimizar a sua execução. O middleware ZeliGrid, uma infra-estrutura baseada em grades computacionais, provê estes mecanismos através de contratos. Durante a execução da aplicação ZeliGrid avalia se o contrato está sendo respeitado e dinamicamente decide se os diversos componentes continuam executando nos nós atuais ou se devem ser reiniciados em outro nó, com mais recursos. Neste trabalho introduzimos um mecanismo de persistência do estado (checkpointing) e migração dos componentes da aplicação, integrado à ZeliGrid, com o objetivo de manter a aplicação executando segundo seus requisitos não-funcionais, sem a necessidade de reiniciar seus componentes quando ocorre uma reconfiguração.*

1. Introdução

Ambientes altamente distribuídos, formados por nós de processamento ou clusters ligados por redes, geralmente sob administração de entidades diferentes, são caracterizados pela heterogeneidade e variação da disponibilidade de seus recursos. Por

exemplo, a capacidade de processamento e memória, e enlaces de rede (banda passante e atraso) podem se apresentar muito diferentes. Além disso, em geral não se pode realizar a reserva de recursos ou controlar, o tempo todo, a alocação dos mesmos, a menos que haja aderência a políticas de alocação e escalonamento restritivas. Assim sendo, os recursos disponíveis para uma aplicação distribuída pode variar muito durante sua execução.

Aplicações com requisitos não-funcionais executando em tais ambientes requerem mecanismos para (i) descrever seus requisitos de qualidade e recursos requeridos; (ii) monitorar o estado dos recursos utilizados e os recursos disponíveis, e (iii) gerenciar a distribuição de componentes, baseado no estado dos recursos monitorados, com o objetivo de otimizar a sua execução. Por exemplo, uma aplicação pode requerer uma determinada capacidade de processamento e memória disponíveis para a execução de seus componentes ou, ainda, enlaces com atrasos máximos observados para executar com a qualidade requerida.

Neste contexto, é importante que os componentes distribuídos de uma aplicação em execução possam ser relocados ou migrados para outro nó, motivados pela inferência de que uma falha está na iminência de ocorrer na rede ou em um nó de processamento, por alguma falha do componente, ou simplesmente porque o estado atual dos recursos não atende mais aos requisitos não-funcionais contratados. Para tornar a migração de componentes viável é importante a presença de um mecanismo de *checkpointing*, que permita salvar periodicamente o estado de um componente e retomar sua execução em outro nó da grade a partir do ponto salvo no último *checkpoint*.

O middleware ZeliGrid [Granja 2006], uma infra-estrutura de suporte para aplicações distribuídas baseada em grades computacionais, desenvolvido em nosso grupo, provê uma parte dos mecanismos mencionados através do conceito de contratos. Um contrato expressa, em alto-nível, uma política para a seleção/alocação de recursos e componentes, bem como descreve uma máquina de estados para a reconfiguração da aplicação. Durante a execução da aplicação, ZeliGrid monitora os recursos selecionados para a aplicação, e avalia se o contrato está sendo respeitado. Dinamicamente, ZeliGrid decide se os diversos componentes continuam executando nos nós atuais ou se devem ser reiniciados em outros nós, que atendam aos requisitos do contrato.

Neste trabalho introduzimos um mecanismo de *checkpointing,* com persistência do estado, integrado ao ZeliGrid. O objetivo é adicionar ao middleware a capacidade de manter a aplicação executando segundo seus requisitos não-funcionais, sem a necessidade de reiniciar seus componentes, limitação da versão atual. Assim, quando ZeliGrid detectar a necessidade da migração de um componente, o mesmo será reiniciado em um nó apto, e retomará sua execução a partir do estando do último *checkpoint*.

Os mecanismos de *checkpointing* e restauração propostos são baseados no mecanismo de serialização de objetos da plataforma Java. Estes são integrados a uma versão especializada do mecanismo de migração presente na arquitetura do *middleware*, permitindo recuperar as informações necessárias para restauração dinâmica do estado da instância do componente que deve ser restaurado. Desta forma melhoramos o desempenho do procedimento de reconfiguração de aplicações de ZeliGrid e provemos alguma tolerância à falhas ocorridas na execução dos componentes da aplicação.

O restante do texto está organizado da seguinte forma. Na Seção 2 introduzimos o *middleware* ZeliGrid. Na Seção 3, apresentamos algumas técnicas de *checkpointing* utilizadas em contextos similares ao nosso. Na Seção 4 apresentamos a proposta de implementação e a arquitetura da técnica de criação de *checkpoints* integrada ao ZeliGrid e discutimos a solução adotada. Por fim, na Seção 5, concluímos o texto, fazendo observações finais sobre o trabalho.

2. ZeliGrid

ZeliGrid [Granja 2006] é um *middleware* que oferece suporte à execução para aplicações distribuídas com requisitos não-funcionais dinâmicos em um ambiente de grades computacionais através de serviços que permitem à aplicação: (i) definir seus requisitos não-funcionais; (ii) descobrir os recursos disponíveis na grade e selecionar os nós da grade cujos recursos disponíveis melhor atendam sua necessidade de execução, (iii) implantar e executar componentes distribuídos; e provê ainda um (iv) serviço de reconfiguração, que possibilita à aplicação ter seus componentes dinamicamente adaptados quando os recursos disponíveis em algum nó utilizado pela aplicação já não atende às especificações de seus requisitos não-funcionais.

ZeliGrid integra em uma arquitetura de suporte (i) o Globus Toolkit [Globus (B) 2007], um middleware de serviços para Grades Computacionais que provê a base para os serviços de execução remota; e (ii) o Network Weather Service [Obertelli 2007], que coleta informações sobre diversos tipos de recursos computacionais presentes em sistemas distribuídos, realizando medições e previsões sobre a disponibilidade destes recursos.

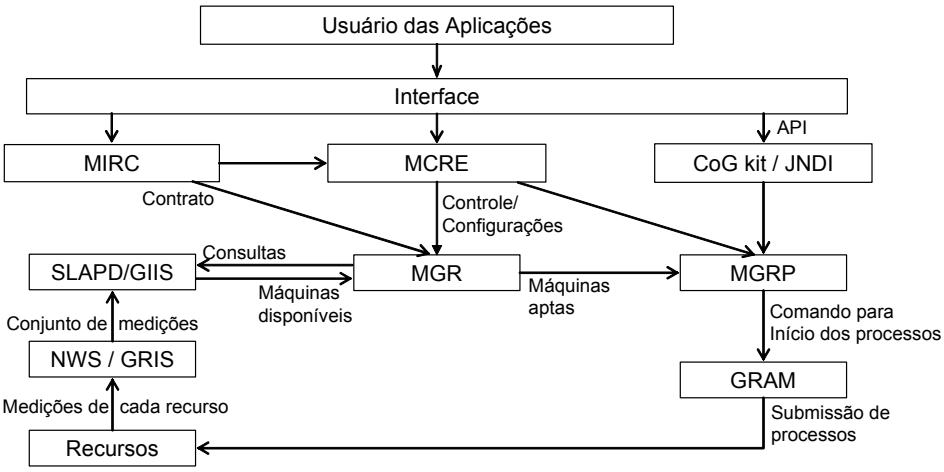

Figura 1. Arquitetura de suporte

Sobre esta arquitetura de suporte foi desenvolvida (i) uma arquitetura de controle, composta por quatro módulos e (ii) uma interface gráfica que permite ao usuário de ZeliGrid configurar e monitorar o funcionamento das aplicações distribuídas. A Figura 1 apresenta a arquitetura atual de ZeliGrid, onde podem ser identificados os módulos desenvolvidos: o MIRC (módulo de interpretação de regras de contratos); o MGR (módulo de gerência de recursos); o MGRP (módulo de gerência remota de processos); e o MCRE (módulo de controle e reconfiguração de experimentos), Na próxima subseção apresentamos brevemente os serviços e a arquitetura. Maiores detalhes em [Granja 2006].

2.1 Serviços de ZeliGrid

Descoberta e indexação de recursos. O módulo MGR é responsável pela descoberta, gerência e avaliação dos recursos. Para isso, ele oferece aos outros módulos, funções para a busca de recursos registrados em servidores NWS e MDS distribuídos pelos nós da grade. Dados sobre recursos estáticos (como arquitetura da CPU) são coletados pelo MDS (*Monitoring and Discovery System*), disponível no Globus Toolkit [Globus (A) 2007]. Dados sobre recursos dinâmicos (por exemplo, disponibilidade de tempo de CPU) são coletados através de sensores NWS. Além de realizar medições sobre recursos locais a cada nó, estes sensores interagem para realizar medições sobre o desempenho da rede (como tempo de *roundtrip* de pequenos pacotes TCP, por exemplo). O MDS é integrado ao NWS, conforme descrito em [Wolski 2007], através de um servidor LDAP (*sldap* [OpenLDAP 2007]), que age como um repositório de informações, coletando dados do NWS e fornecendo-os ao MDS. O módulo MGR também recebe do MIRC o contrato interpretado e avalia que nós da grade estão aptos para execução dos componentes, segundo as especificações dos perfis do contrato.

Contrato. ZeliGrid permite que a aplicação defina um contrato contendo requisitos não-funcionais. Na versão atual, um contrato possui dois perfis, e cada perfil possui uma lista de atributos, que descreve os requisitos não-funcionais da aplicação. No primeiro perfil são descritos os requisitos ideais para execução dos componentes da aplicação, enquanto o segundo perfil representa os requisitos mínimos para garantir seu funcionamento. O módulo MIRC é responsável pelo *parsing* e interpretação do contrato logo no início de suas atividades.

Implantação remota de processos. O módulo MGRP, responsável por submeter a aplicação para execução, consulta o MICR para obter a lista de máquinas aptas e, de acordo com os comandos do módulo MCRE, utiliza o GRAM para submeter os componentes para as máquinas remotas. O GRAM (*Globus Resource Allocation Manager*) [Globus (B) 2007] é a ferramenta do Globus responsável pelo controle da execução remota dos processos. Este serviço oferece uma API de suporte à submissão e implantação dos processos na Grade, além de gerenciar os recursos necessários à execução das aplicações.

Configuração e gerência da aplicação. O módulo MCRE é responsável pelo controle das aplicações executando em ZeliGrid. Este módulo permite a descrição da configuração da aplicação, possibilitando ao usuário especificar, entre outras informações, o número de máquinas necessárias para a execução do experimento e a localização dos servidores NWS e MDS a serem utilizados. Além disso, o MCRE utiliza funções disponibilizadas pelo MGR e pelo MGRP para coordenar a execução dos componentes da aplicação, realizar sua submissão para execução remota e também controlar a realização do processo de reconfiguração, bloqueando, cancelando e reativando componentes.

A política adotada por ZeliGrid procura implantar os componentes da aplicação em nós que atendem o primeiro perfil. Se não existirem nós que atendam ao perfil ideal, nós que atendam pelo menos o segundo perfil são utilizados.

Durante a execução da aplicação, o MCRE, monitora periodicamente, segundo um intervalo de tempo configurado pelo usuário, os atributos de cada nó (utilizados ou não) e os avalia em relação aos perfis. Quando um componente está executando em um

nó que deixa de atender os requisitos especificados pelo perfil ideal, e existe um outro nó que atenda este perfil, o componente é reiniciado neste outro nó. Caso contrário, o segundo perfil do contrato é então avaliado para identificar se o componente pode continuar executando, ainda que no perfil mínimo, no mesmo nó, ou se também precisa ser reiniciado em outro nó apto para o segundo perfil. Se não existirem nós que atendam a algum dos dois perfis, um componente pode ser retirado de execução e aguardar que um nó apto fique disponível ou a aplicação é terminada por falta de recursos. O intervalo de tempo em que ocorrem as tentativas de reconfiguração da aplicação pode ser configurado pelo usuário através da interface gráfica do ZeliGrid.

O procedimento de reconfiguração consiste, então, em (i) suspender a execução do componente-alvo, (ii) localizar um nó cujos recursos computacionais disponíveis sejam capazes de atender a um dos perfis e (iii) implantar o componente neste nó, quando encontrado, e reiniciar completamente sua execução.

Algumas aplicações distribuídas, do tipo *bag-of-tasks*, ou aplicações de teste podem abrir mão da persistência do estado de componentes que serão migrados. O componente original é simplesmente reinstanciado. Entretanto, aplicações mais complexas precisam manter a consistência de seus componentes durante a reconfiguração, o que pode ser obtido através do salvamente do estado em *checkpoints* e posterior uso do último estado salvo durante a recuperação do componente. Além disso, não havendo a persistência do estado do componente toda computação já realizada durante sua execução será perdida.

3. *Checkpointing*

Em ambientes altamente distribuídos, como as grades computacionais, o emprego de *checkpointing*, aliado à migração de processos, apresenta vantagens [Lopes 2005]:

- balanceamento de carga: as aplicações podem ser movidas para máquinas com recursos ociosos, otimizando a utilização dos recursos disponíveis;
- tolerância à falhas: em caso de falha, o estado anteriormente salvo pode ser usado para reiniciar a execução da aplicação em outro nó;
- reconfiguração: se um nó torna-se indisponível ou, se seus recursos não atendem mais à demanda da aplicação, esta pode ser movida para outro nó.

A recuperação por retrocesso baseada em *checkpointing* é uma técnica adotada por vários ambientes de grades computacionais, como o Integrade [Camargo 2006] e o OurGrid [Silva 2006] e por sistemas de middleware como o Condor[Condor 2007].

As técnicas de *checkpointing* encontradas na literatura utilizam meios sofisticados para permitir a captura do estado de execução dos processos componentes da aplicação. As técnicas mais utilizadas são (i) pré-compiladores que inserem código adicional em pontos do código-fonte em que o *checkpoint* deve atuar ([Camargo 2006], [Ellahi 2006]) e (ii) mecanismos de captura dos dados da pilha de execução de aplicações desenvolvidas em linguagens como C ([Camargo 2006]), ou informações de estado dos processos na JVM ([Ellahi 2006], [Wang 2001]), para *checkpointing* de aplicações Java.

4. Integração de *checkpointing* ao ZeliGrid

Em nosso trabalho, consideramos inicialmente utilizar os mecanismos de *checkpoint* do Globus, base de ZeliGrid. Entretanto, o suporte oferecido é limitado à persistência de recursos como `STDIN` e `STDOUT`. Assim, optamos por investigar o uso da serialização de objetos [Sun 2008] da plataforma Java para realizar o salvamento do estado de um componente, transformado o mesmo em um fluxo, gravando o mesmo em uma memória persistente a cada intervalo de *checkpoint*. As próximas seções apresentam e discutem esta abordagem.

O mecanismo de serialização permite transformar um objeto em um fluxo de bits, que pode ser usado para transportar o objeto por uma conexão de rede, por exemplo, ou persisti-lo em uma memória não volátil, como um arquivo em disco. Da mesma forma, um objeto serializado pode ser reconstituído em uma nova instância. Desta forma, podemos fazer com que um objeto continue existindo mesmo após o encerramento da aplicação que o originou. A API de serialização de Java provê estes mecanismos [Sun 2008].

Nossa abordagem consiste em ser o menos intrusivo possível em ZeliGrid e no código da aplicação do usuário. Porém, este deverá adotar uma disciplina de programação, agregando o mecanismo de *checkpointing* através de herança em Java e aderindo a um padrão de codificação de componente. Esta prática é adotada em outros *frameworks*, onde se deseja que um módulo de software seja aderente a um modelo de componentes, com algumas características pré-determinadas. Por exemplo, J2EE, .NET, CORBA, RMI, etc.

Assim, (i) os módulos MGRP e MCRE de ZeliGrid devem ser adaptados para carregar e iniciar um componente parametrizado pela imagem serializada de seu estado, o mecanismo de migração é especializado; e (ii) o código do usuário deve estender a classe abstrata *Checkpoint* do módulo de *checkpointing* e aderir a uma disciplina de programação.

4.1. Arquitetura do módulo de checkpointing

A Figura 2 apresenta o módulo de *checkpointing*, na forma de um pacote Java, bem com a classe *Thread* e a interface *Serializable*, utilizadas. As Figuras 3 e 4 apresentam diagramas de seqüência com as interações entre as classes da Figura 2 e as classes dos módulos de ZeliGrid no salvamento e restauração de um componente, respectivamente.

1. A classe *Aplicação*, fornecida pelo programador, deve estender a classe abstrata *Checkpoint*, implementando em seu método *run()* todo o processamento a ser realizado durante sua execução. O método *main()* desta classe cria uma instância da classe *Aplicação*, e uma instância da classe *SingleJobExecutor,* que recebe como parâmetro a primeira (1:). Em seguida, ainda no método *main()* a *thread* da instância da classe *SingleJobExecutor* é ativada (ativando-se seu método *run()*), (2:).
2. Ao ter seu método *run()* ativado classe *SingleJobExecutor* instancia a classe *Temporizer*, passando a referência ao objeto da classe *Aplicação* (3:), recebida no passo 1, e inicia duas *threads*: a do objeto *Aplicação* e a do *Temporizer* (4: e 5:).
3. A *thread* do objeto da classe *Temporizer*, iniciado no passo 2, controla o intervalo de tempo em que um *checkpoint* é gerado. Esse intervalo de tempo é configurável

pelo usuário, através da interface gráfica do ZeliGrid. No momento de *checkpoint* a *thread* da *Aplicação* é suspensa pela *thread* de *Temporizer*, o objeto é serializado, persistido, e em seguida a *thread* da *Aplicação* é reativada (6:, 7: e 8:).

4. *CheckpointHandler* contém os métodos para serializar (*saveState*) e recuperar (*restoreState*) o estado do objeto que representa a classe da aplicação.

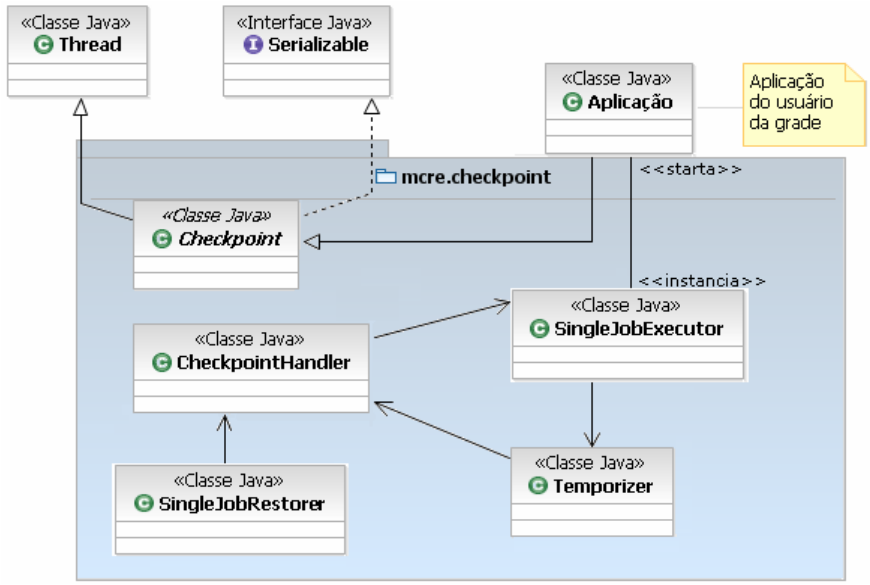

Figura 2. Diagrama de Classes

5. *CheckpointHandler.saveState()* é invocado pela *thread* de *Temporizer*, no momento de checkpoint. Esse método contém todo código responsável por serializar e persistir a aplicação do usuário.

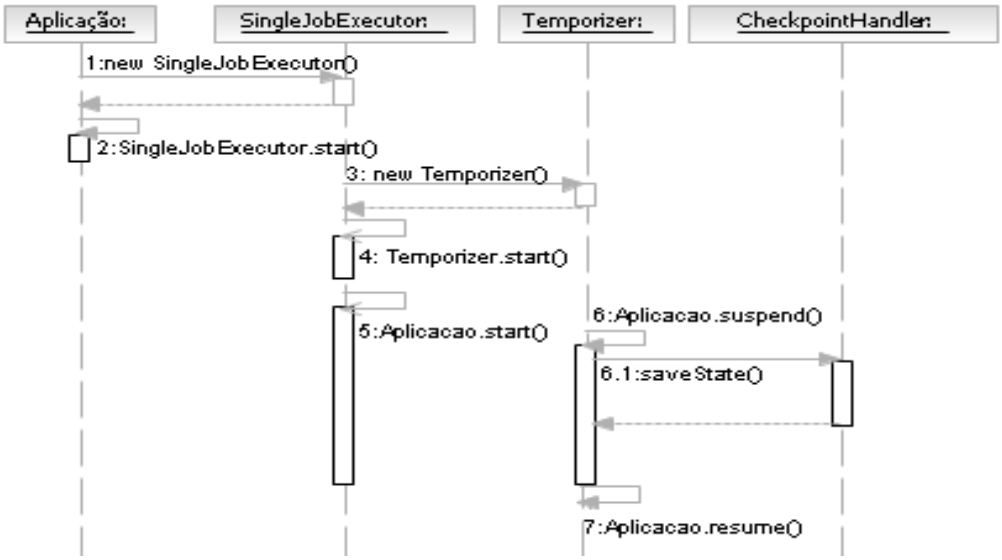

Figura 3. Diagrama de Seqüência – Salvamento de Estado

6. *CheckpointHandler.restoreState()* é o método responsável por recuperar o objeto serializado e passar sua referência para a classe *SingleJobExecutor*, que realiza novamente as ações descritas no item 2, mas utilizando como parâmetro o objeto

reconstruído a partir do arquivo com a imagem serializada do objeto *Aplicação* (seqüência do diagrama da Figura 4). Dessa maneira, a execução do componente é reiniciada a partir do estado salvo no último *checkpoint*.

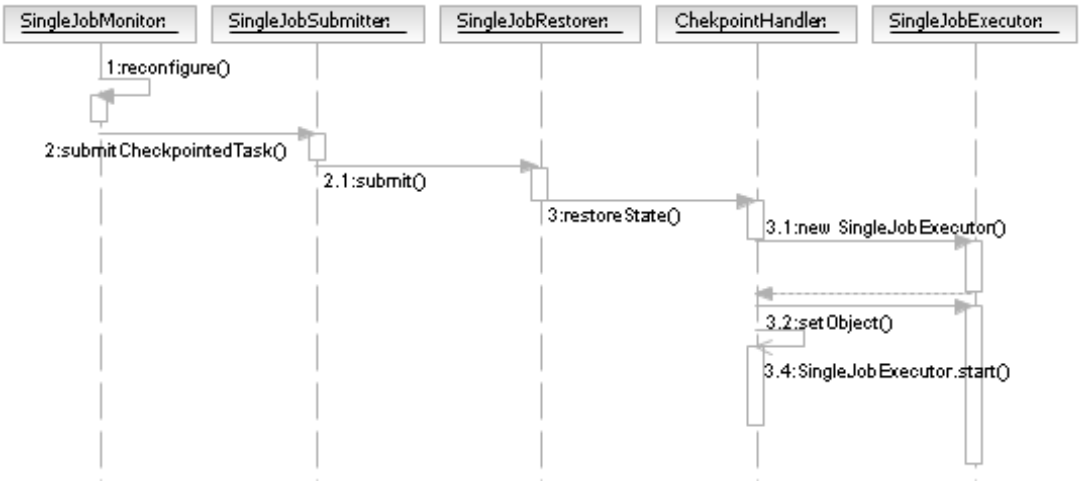

Figura 4: Diagrama de Seqüência – Restauração de Estado

Observe-se que os passos 5 e 6 são parte da especialização do mecanismo de migração de ZeliGrid. Ao identificar que um componente precisa ter sua execução reiniciada em outro nó, o método *restoreState()* é executado em um nó diferente daquele onde o método *saveState()* salvou o estado.

4.2 Avaliação preliminar

A modularidade da arquitetura de ZeliGrid facilitou a integração do mecanismo de *checkpointing* e migração. Além da adaptação do módulo de interface para especificar o intervalo de *checkpoint*, outra modificação significativa se deu no processo de reconfiguração. Este processo é controlado pela classe *SingleJobMonitor* do módulo MCRE de ZeliGrid, que agora verifica se o usuário deseja ou não usar *checkpointing* e se o componente a ser carregado já passou por algum *checkpoint*. Na carga inicial do componente o objeto original é utilizado. Caso contrário, a imagem serializada no último *checkpoint* é utilizada. Neste caso, a classe *SingleJobRestorer* é submetida para execução remota e invoca o método *CheckpointHandler.restoreState()*, que restaura a execução do componente a partir de uma referência ao arquivo que contém o último *checkpoint*.

Atualmente estamos aperfeiçoando o código que adapta o mecanismo de reconfiguração do *MCRE*, para que nos casos em que ocorre *checkpointing* dos componentes da aplicação, a classe *SingleJobRestorer* seja submetida para execução remota, no lugar da classe original do usuário, permitindo assim a restauração da execução da aplicação, conforme descrito no parágrafo anterior.

Em paralelo, estamos definido o sistema de diretórios para manter as imagens serializadas de *checkpoint* dos componentes. Trabalhamos com duas possibilidades (i) assumir que os nós da grade compartilham, através de um sistema de arquivos distribuídos ou serviço centralizado, o diretório em que as imagens serão salvas ou (ii) transferir ponto-a-ponto os arquivos necessários à restauração da execução da aplicação utilizando o utilitário GridFTP, do Globus, permitindo a realização de operações de

transferência de arquivo entre os nós da grade com segurança. A solução centralizada é simples e pode ser robusta se projetada para ser tolerante a falhas. Mas, apresenta problemas potenciais de escalabilidade. A solução distribuída, em que a imagem serializada de um componente é transmitida do nó que não está mais apto para um outro nó, tende a ser mais complexa e não permite a recuperação do componente se o nó de origem efetivamente falhar. Porém, deve escalar melhor. Para realizar os testes, estamos utilizando neste primeiro momento a primeira opção.

5. Conclusões

A técnica de checkpointing proposta, utilizando a serialização de objetos, embora simples, impõe ao desenvolvedor a disciplina de programação, mencionada na Seção 4.1. Estamos também cientes de algumas limitações desta técnica. Por exemplo, usando a serialização de objetos não é possível determinar que trechos do código do componente foram executados até o momento de *checkpoint*, dado que a pilha de execução não é salva. Isto permitiria "saltar" instruções já processadas. Mesmo assim, segundo demonstram testes iniciais, aplicações cujo processamento baseia-se em estruturas de repetição (*loops*) funcionam bem, como por exemplo, aplicações científicas.

Entre as atividades a serem realizadas estão os testes para avaliar o desempenho do mecanismo de *checkpoint* e o de migração de componentes, bem como a avaliação do uso do mecanismo sobre o tempo total de execução da aplicação.

Acreditamos que a utilização de escalonamento de componentes, baseada em políticas associadas aos requisitos não-funcionais das aplicações, em conjunto com o monitoramento do contexto, já oferecidos por ZeliGrid, aliado ao mecanismo de *checkpointing* proposto, pode tornar mais eficiente a execução de aplicações e o uso de recursos computacionais em ambientes altamente distribuídos.

Agradecimentos. Os autores gostariam de agradecer o apoio parcial da Faperj e CNPq. Jeane Cezário gostaria de agradecer o PIBIC UERJ.

Referências

Camargo, R. Y., Goldchleger, A., Kon, F., Goldman, A., "Checkpointing BSP parallel applications on the InteGrade Grid middleware", Concurrency and computation-practice & experience, Hoboken, NJ, EUA, Vol. 18, No. 6, pp. 567-579, 2006.

Condor Manual. "Condor's Checkpoint Mechanism", Dezembro, 2007. http://www.cs.wisc.edu/condor/manual/v6.8/4_2Condor_s_Checkpoint.html

Ellahi, T. N., Hudzia, B., McDermott, L., Kechadi, T., "Transparent Migration of Multi-Threaded Applications on a Java Based Grid", The IASTED International Conference on Web Technologies, Applications, and Services (WTAS 2006), Alberta, Canada, Julho, 2006.

Globus Alliance (A), "MDS 2.2 User´s Guide", Novembro, 2007. htpp://www-fp.globus.org/mds/mdsusersguide.pdf

Globus Project (B), "The Globus Toolkit Documentation", http://www.globus.org, Setembro, 2007.

Granja, S. R., Sztajnberg, A., "Zeligrid: uma arquitetura para a implantação de aplicações com requisitos não-funcionais dinâmicos em Grades Computacionais", IV WCGA / SBRC 2006, Curitiba, Junho, 2006.

Lopes, R. F., Silva, F. J. da S., "Migration Transparency in a Mobile Agent Based Computational Grid", Proceedings of the 5th WSEAS Int. Conf. on Simulation, Modeling and Optimization, 1st WSEAS International Symposium on GRID COMPUTING. Corfu, Greece, pp. 31-36, Agosto, 2005.

Obertelli, G., "Network Weather Service User´s Guide", Novembro, 2007. http://nws.cs.ucsb.edu/users_guide.html

OpenLDAP Foundation, "Software Man Pages: slapd", Novembro, 2007. http://www.openldap.org/software/man.cgi?query=slapd

Silva, H., Siqueira, T. F. de, Dalpiaz, L. R., Jansch-Pôrto, I. E. S., Weber, T. S. "Implementação de um Mecanismo de Recuperação por Retorno para o Ambiente de Computação OurGrid", WTF 2006 / SBRC 2006, Curitiba, 2006.

Sun Microsystems, "Java Object Serialization Specification", Janeiro, 2008. http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html

Wang, H., Zeng G., Lin S. "A strong migration method of mobile agents based on Java", The Sixth International Conference on Computer Supported Cooperative Work in Design, Ontário, Canadá, pp. 313–318, 2001.

Wolski, R., "Lecture Notes", Novembro, 2007. http://www.cs.ucsb.edu/~rich/class/cs290I-grid/notes/