Pontifícia Universidade Católica
do Rio de Janeiro

Daltro Simões Gama

Integração de um Sistema de Submissão Batch com um Ambiente de Computação em Nuvem

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC–Rio.

Orientadora : Prof^a. Noemi de La Rocque Rodriguez
Co–orientadora: Dr^a. Maria Julia de Lima

Rio de Janeiro
Dezembro de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Daltro Simões Gama

Integração de um Sistema de Submissão Batch com um Ambiente de Computação em Nuvem

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–Graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof[a]. Noemi de La Rocque Rodriguez
Orientadora
Departamento de Informática — PUC–Rio

Dr[a]. Maria Julia de Lima
Co–orientadora
Instituto Tecgraf

Dr. Renato Fontoura de Gusmão Cerqueira
IBM Research Brasil

Prof[a]. Ana Lúcia de Moura
Departamento de Informática — PUC-Rio

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 21 de dezembro de 2015

Daltro Simões Gama

Graduou-se em Ciência da Computação na Universidade Federal de Itajubá (UNIFEI, Minas Gerais) em 2004, tendo como projeto final um motor de inferência lógica para sistemas especialistas com bytecode próprio junto com uma linguagem de alto nível e seu compilador e IDE. Trabalhou na iniciativa privada desde então implementando inovações para tratamento massivo de bases de dados de roteatores legados de telefonia fixa comutada. Trabalhou como professor substituto na UERJ no primeiro semestre de 2012, lecionando linguagens formais e autômatos para graduação. Trabalha como desenvolvedor de software na Petrobras desde 2010, representando o Tecgraf/PUC-Rio desde 2013. Ingressou no mestrado da PUC-Rio em meados de 2013, na linha de Sistemas Distribuídos.

Ficha Catalográfica

Gama, Daltro Simões

Integração de um Sistema de Submissão Batch com um Ambiente de Computação em Nuvem / Daltro Simões Gama; orientadora: Prof^a. Noemi de La Rocque Rodriguez; co–orientadora: Dr^a. Maria Julia de Lima. — 2015.

83 f. : il. (color.); 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Sistemas Distribuídos. 3. Computação em Nuvem. 4. Microsoft Azure. 5. CSGrid. I. Rodriguez, Noemi de La Rocque. II. Lima, Maria Julia. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Agradecimentos

Impossível escrever um texto de agradecimento sobre um trabalho que encerra um ciclo tão rico em tantos aspectos sem se esquecer de alguém. Começo então agradecendo a todos os que ao menos conviveram comigo no pouco tempo em que me fazia de corpo presente entre idas e vindas de minha então vida dupla. Com certeza são pessoas que levarei pelo resto da vida.

Um agradecimento especial para o Carlos Cassino, que endossou a minha idéia de cursar o mestrado sem parar de trabalhar. Sem isso, este texto não existiria.

Agradeço a Prof^a Noemi pela orientação atenciosa e disciplinada, assim como também agradeço a Maria Julia por todo o apoio técnico e acadêmico dado ao longo da elaboração desta monografia.

Agradeço o departamento de informática da PUC-Rio, que me apoiou na realização do curso.

Agradeço os colegas parceiros de estudos, bons amigos que fiz.

Agradeço a minha família que, mesmo distante, nunca deixou de demonstrar apoio.

Agradeço a Amanda, que suportou meus momentos de procrastinação.

Agradeço aos meus amigos e colegas de trabalho que me entenderam quando reduzi a minha já curta quota de socialização em busca de realização pessoal.

Um brinde à vida.

Resumo

Gama, Daltro Simões; Rodriguez, Noemi de La Rocque; Lima, Maria Julia. **Integração de um Sistema de Submissão Batch com um Ambiente de Computação em Nuvem**. Rio de Janeiro, 2015. 83p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A computação em nuvem, com sua promessa de redução de custos de manutenção e facilidades de configuração, está despertando cada vez mais o interesse da comunidade científica que depende de muitas máquinas para executar seus programas. Neste trabalho implementamos uma nova integração para o sistema CSGrid, do Tecgraf/PUC-Rio, que o torna apto a submeter programas para execução no ambiente de nuvem pública Microsoft Azure, usufruindo assim dos benefícios da elasticidade de recursos computacionais. Para tal, apresentamos algumas medidas de desempenho para o caso de uso da nuvem pública Microsoft Azure pelo sistema CSGrid, no que se refere a custos de transferência de dados e provisionamento de máquinas virtuais. O objetivo com essa integração é avaliar os benefícios e as dificuldades que envolvem o uso de um modelo de execução em nuvem por um sistema tipicamente voltado a execução de aplicações de alto desempenho em clusters.

Palavras–chave

Sistemas Distribuídos; Computação em Nuvem; Microsoft Azure; CSGrid.

Abstract

Gama, Daltro Simões; Rodriguez, Noemi de La Rocque (Advisor); Lima, Maria Julia (Co-Advisor). **Integration of a Batch Submission System with a Cloud Computing Environment**. Rio de Janeiro, 2015. 83p. MsC. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Cloud computing appeals to those who need many machines to run their programs, attracted by low maintenance costs and easy configuration. In this work we implemented a new integration for the CSGrid system, from Tecgraf/PUC-Rio, enabling it to submit workloads to Microsoft Azure public cloud, thus enjoying the benefits of elastic computing resources. For this purpose, we present related works and some performance measures in the case of CSGrid's use of Microsoft Azure public cloud, with regard to costs on data transfers and provisioning of virtual machines. With this integration, we could evaluate the benefits and difficulties involved in using cloud resources in a system designed for the submission of HPC applications to clusters.

Keywords

Distributed Systems; Cloud Computing; Microsoft Azure; CSGrid.

Sumário

1
Introdução

Em essência fruto das tecnologias emergentes de virtualização, o paradigma de computação em nuvem veio para criar abstrações convenientes para quem procura elasticidade em hardware para processamento científico ou comercial. A possibilidade de se terceirizar tal infra-estrutura e ainda ter a comodidade de facilmente controlá-la desperta o interesse de uma grande gama de usuários.

Neste trabalho iremos explorar o aproveitamento da elasticidade das nuvens e seus novos serviços para apoiar os sistemas de submissão e gerência de programas para execução em lote através de exercícios com um ambiente de nuvem pública e através da integração com sistema CSGrid (Lima et al., 2006), do Tecgraf/PUC-Rio. O sistema de nuvem a ser estudado é o Microsoft Azure(Azure - Microsoft's Cloud Computing Platform), e a integração se dá através da implementação de um novo módulo para o CSGrid, denominado SGA Azure. O objetivo com essa integração é avaliar os benefícios e as dificuldades que envolvem o uso de um modelo de execução em nuvem por um sistema tipicamente voltado a execução de aplicações de alto desempenho em clusters.

Ferramentas de submissão de programas para execução são comuns quando se trabalha com computação de alto desempenho ou HPC. Como exemplo, podemos citar o PBS (Bayucan et al., 1999), com sua implementação Torque (TORQUE Resource Manager) ou HTCondor (Thain et al., 2005), que distribuem execuções de programas em máquinas remotas, monitoram seus recursos e a própria execução. Aplicações HPC, tipicamente científicas, são projetadas para tirar proveito de processadores de alta capacidade e usufruir de uma infra-estrutura de rede de alta velocidade. Aplicações HPC são normalmente executadas em supercomputadores ou clusters de máquinas próprias de quem os executa, e consequentemente as ferramentas tradicionais de submissão também são projetadas levando isso em conta. Com o hardware dimensionado e dedicado especificamente para os requisitos das aplicações, o desempenho das aplicações HPC tem atendido as expectativas de seus usuários.

Utilizar computação em nuvem para executar programas tradicionalmente submetidos em clusters HPC tem sido um desafio devido às perdas de desempenho que a computação em nuvem traz se comparado com os clusters dedicados. Alguns trabalhos têm proposto descrever ambientes híbridos que unem as vantagens de clusters dedicados de alto desempenho com recursos de

nuvem ((Mateescu et al., 2011) e (Tröger e Merzky, 2014)).

A arquitetura proposta para o SGA Azure foi motivada por resultados de medições apresentadas no Capítulo 4 e serve como estudo de caso para explorar as vantagens e desvantagens da elasticidade de recursos provisionados para a execução de programas que tipicamente utilizam o CSGrid em ambientes HPC.

A estrutura deste trabalho é dada a seguir. No Capítulo 2, descrevemos o sistema CSGrid e sua arquitetura. No Capítulo 3, introduzimos os conceitos relevantes de computação em nuvem e detalhamos aspectos específicos da Microsoft Azure. No Capítulo 4, descrevemos experimentos feitos com a nuvem Azure visando embasar decisões na elaboração do SGA Azure. No Capítulo 5, apresentamos outras soluções já dadas para o uso de computação em nuvem para submissão de programas em lote. No Capítulo 6, apresentamos o SGA Azure concebido. No Capítulo 7, mostramos resultados de experimentos feitos com o SGA Azure e discutimos as dificuldades enfrentadas na implementação. Finalmente, a conclusão é apresentada no Capítulo 8 junto com sugestões para trabalhos futuros.

2
O CSGrid

O CSGrid é um sistema construído como uma instância do framework CSBase (Lima et al., 2006), desenvolvido ao longo dos últimos dez anos no Instituto Tecgraf, da PUC-Rio. Trata-se de um sistema voltado para submissão de programas para execução remota em batch, provendo recursos adicionais para gestão e integração das aplicações em um ambiente computacional distribuído e heterogêneo, abstraindo a infra-estrutura para que usuários possam compartilhar recursos em seus projetos. O framework CSBase é utilizado por cinco sistemas diferentes, sendo o CSGrid um deles. Hoje o CSGrid é usado para submissão de programas HPC em diversos projetos em desenvolvimento e em produção em diferentes domínios de aplicações. O CSGrid oferece um repositório em que o usuário pode selecionar programas para execução em máquinas automaticamente designadas pelo sistema. Tais atributos tornam o CSGrid elegível para a integração com nuvens de forma a usufruir da elasticidade de recursos de que estas dispõe. Sua abstração de áreas de projetos para compartilhamento de dados, organização de recursos e os programas executáveis fornecem um grande apelo para os usuários. Na nomenclatura do CSGrid, os programas executáveis são denominados "algoritmos". Porém, nesta monografia, manteremos o termo "programas". Cada submissão de um programa com seus parâmetros é denominada um "comando". O sistema inclui facilidades de gestão de autenticação e autorização para o uso dos dados nas áreas de projetos, podendo assim ser usado por organizações que lidam com dados sensíveis ou programas com propriedade intelectual.

Nas áreas de projeto, o usuário disponibiliza arquivos a serem usados como entrada para um programa. Ao submeter um programa para execução, tal ação é sempre vinculada a alguma área de projeto. Um programa submetido, quando em execução, tem acesso apenas a arquivos na área designada e, de forma análoga, os arquivos gerados pelo programa durante sua execução só poderão ser gravados dentro da mesma área. As áreas de projetos também podem ser compartilhadas entre diferentes usuários do sistema.

Na arquitetura do CSGrid, um módulo servidor é responsável pela gerência das áreas de projetos, de programas, e pela submissão de comandos do usuário para sistemas remotos. Para essa submissão, o sistema contacta um módulo denominado Sistema de Gerência de Algoritmos (*SGA*). O SGA pode fazer o papel do próprio nó de execução, disparando os processos que executam o programa propriamente dito, ou pode ser uma fachada para outro sistema

de escalonamento, distribuindo tarefas em algum cluster ou sistema terceiro como PBS (Bayucan et al., 1999)/Torque (TORQUE Resource Manager) ou HTCondor(Thain et al., 2005). A Figura 2.1 ilustra a arquitetura básica do CSGrid com um SGA representando um cluster.

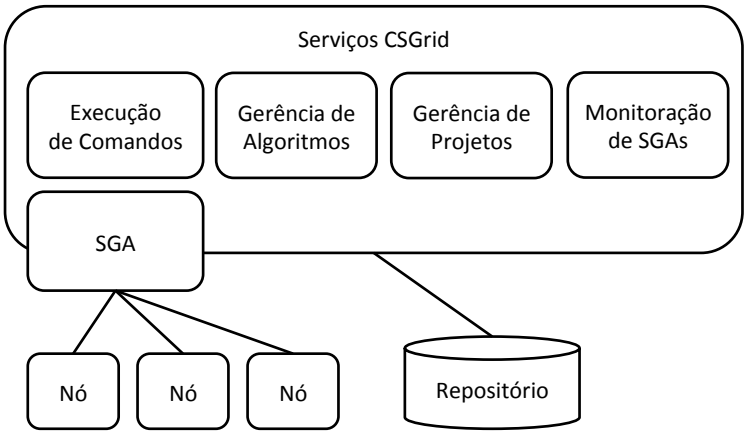

Figura 2.1: Estrutura básica do servidor CSGrid e sua relação com um SGA "cluster"

Dado que o ambiente de nuvem pública é composto por máquinas virtuais, seu uso do ponto de vista de configuração básica de sistema operacional e operação pode ser feito de forma análoga ao de um cluster de máquinas dedicadas, salvo questões ligadas ao desempenho. Assim, um ambiente configurado em um cluster próprio pode ser totalmente transferido para um ambiente de nuvem sem nenhuma mudança na implementação do sistema. Porém, neste trabalho, estamos interessados em que o SGA possa usufruir da elasticidade proporcionada pela infraestrutura de nuvem.

2.1 O SGA

O Sistema de Gerência de Algoritmos (SGA) é um componente que tem como finalidade disparar e gerenciar as execuções dos comandos. Na arquitetura do CSGrid, o SGA faz o papel do componente responsável por receber e gerenciar pedidos de execução de comandos em uma plataforma computacional específica. Além disso, o SGA é responsável por prover ao servidor central do CSGrid informações (memória, CPU, estados dos comandos em execução) a respeito das condições e configurações da plataforma que representa. Essas informações são utilizados pelos usuários na decisão dos recursos que podem ser utilizados ou para o próprio servidor central do CSGrid fazer o escalonamento de submissões de programas para os diferentes SGAs que estão disponíveis. Em alguns cenários, o SGA pode atuar como um intermediário, integrando-se a escalonadores terceiros como PBS/Torque ou

HTCondor. Quando um SGA representa um cluster, ele repassa os comandos submetidos pelo servidor CSGrid para a fila do escalonador em questão.

Um aspecto importante na arquitetura do CSGrid e sua interação com o SGA é o acesso aos arquivos da área de projeto do CSGrid e aos binários dos programas que serão executados. Para a execução de um comando, o CSGrid precisa disponibilizar os arquivos de entrada e scripts de execução para o nó, assim como garantir que os resultados estejam disponíveis na área de projeto ao final da execução. Não é o SGA quem busca os arquivos no servidor CSGrid, e sim o CSGrid que orquestra as transferências, indicando inclusive os caminhos para os arquivos. Sempre que parâmetros de comandos representam entrada ou saída de arquivos, esses arquivos são indicados como caminhos absolutos que o CSGrid assume que o nó de execução possui acessível. Portanto, o SGA é dependente da estrutura de diretórios imposta pelo CSGrid. Originalmente, existiam apenas duas formas para o nó de execução ter acesso aos arquivos de entrada e ao binário do programa:

1. Uso de NFS (Shepler et al., 2003): Caso todas as máquinas envolvidas se encontrem em uma mesma rede, os diretórios Linux/Unix podem ser compartilhados através do sistema de arquivos em rede NFS. Neste caso, os nós de execução terão acesso transparente a todos os dados que necessitam para executar os programas, ler arquivos de entrada e produzir insumos sem auxílio de alguma outra ferramenta externa.

2. CSFS (Santos, 2006): O CSFS é um programa que executa em um processo independente do servidor CSGrid e do nó de execução e tem como único objetivo a transferência de arquivos de e para os nós. Há processos CSFS dedicados tanto ao servidor CSGrid como ao nó, e é o SGA quem declara ao CSGrid necessitar do CSFS. Uma vez em uso, sempre que a execução de um comando for demandada, o servidor CSGrid cria uma *sandbox* no nó de execução, enviando todos os arquivos de entrada do programa para o nó, e só então executa o comando de fato. Ao término da execução do comando, os arquivos modificados e gerados no nó são copiados de volta para o servidor CSGrid através do mesmo mecanismo. O CSFS implementa diferentes protocolos de transferência, sendo que o uso de CORBA é o mais usado nas instalações atuais.

As duas formas de compartilhamento de dados oferecem prós e contras: Usando NFS, um comando de uso muito intenso de I/O pode permanecer durante bastante tempo bloqueado aguardando envio e recebimento de dados pela rede. Porém, se o volume de dados é menor, ou o comando tem um perfil de uso mais intensivo de CPU, a sobrecarga pode não ser significativa.

O uso do CSFS, por sua vez, faz com que os dados necessários para a execução do comando fiquem disponíveis diretamente no disco local do nó de execução através da criação de uma *sandbox*. Em contrapartida, como todos os dados de entrada são enviados para o nó antes do início da execução do comando, caso vários nós sejam iniciados para execuções sobre os mesmos dados simultaneamente, a concorrência para a cópia massiva de dados do servidor CSGrid pode impactar o desempenho.

O SGA possui as seguintes responsabilidades:

1. Indicar a necessidade de uso ou não de um mecanismo de transferência de arquivos como o CSFS;

2. Gerir a execução do programa no nó de execução ou monitorá-lo no escalonador terceiro;

3. Manter a responsividade e a capacidade de informar o status das execuções ao longo do processo, tornando explícitas todas as fases da execução.

O diagrama da Figura 2.2 ilustra o funcionamento do SGA em conjunto com o mecanismo de transferência de arquivos (por exemplo, o CSFS) para cada nó de execução, regidos pelo CSGrid. Na figura, itens mais escuros representam etapas executadas pelo mecanismo de transferência, quando aplicável, enquanto os demais representam etapas do próprio SGA. Tais estados representam as etapas responsáveis por garantir explicitamente um ou mais dos requisitos acima. Em detalhes, as etapas são as seguintes:

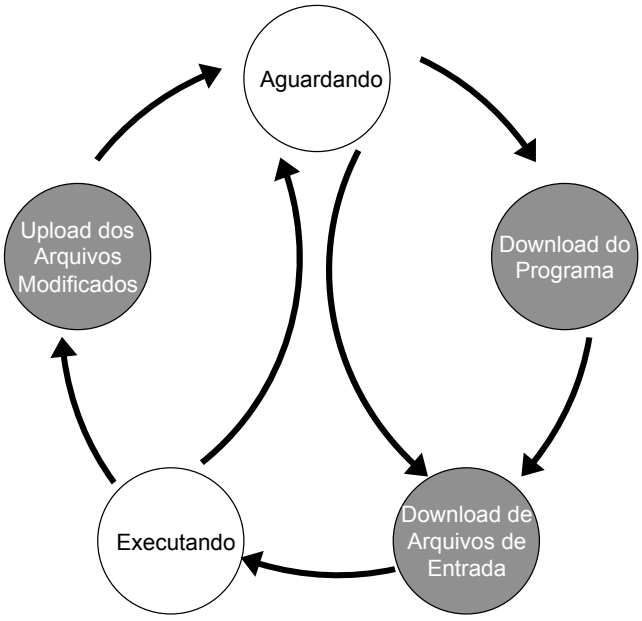

Figura 2.2: Diagrama de estados do nó de execução do SGA

1. Aguardando: Estado onde o SGA está ocioso, aguardando algum comando para executar;

2. Download do Programa: Nesta etapa, o binário do programa propriamente dito é transferido para o nó de execução pelo mecanismo de transferência caso o SGA tenha declarado precisar de tal;

3. Download dos Arquivos de Entrada: Nesta etapa, caso o SGA tenha declarado precisar de um mecanismo de transferência para os arquivos do programa e arquivos de entrada indicados no descritor do comando, estes serão enviados da área de projeto CSGrid para a área *sandbox* no nó de execução pelo mecanismo de transferência;

4. Executando: O processo do programa é iniciado no nó de execução e o SGA o monitora. Nesta etapa, o comando pode encerrar com falha ou não, ou ter sua execução cancelada pelo usuário.

5. Upload dos Arquivos Modificados: Uma vez encerrada a execução, caso o SGA tenha declarado precisar de um mecanismo de transferência, a área de projeto temporária (*sandbox*) do nó de execução é verificada em busca de arquivos criados ou modificados. Apenas os arquivos modificados ou criados são enviados para a área de projeto do servidor CSGrid.

2.2 Contratos de Plugins

O SGA descrito na seção anterior (desenvolvido em C++ e Lua) é executado como um processo daemon na própria máquina de execução (ou em um nó de um cluster) e que se conecta ao servidor CSGrid remotamente. Nesse modelo, a integração entre o SGA e o servidor CSGrid é feita através de CORBA(Group, 2006). Porém, novas demandas de integração com ambientes de execução surgiram e, com isto, o CSGrid passou a oferecer também uma outra forma de integração entre o servidor e o SGA através de um mecanismo de plugin. Nessa arquitetura, as responsabilidades delegadas pelo CSGrid para o SGA passam a poder ter implementações distintas que podem ser carregadas simultaneamente no servidor. O mecanismo de transferência, antes apenas CSFS, foi generalizado, abrindo caminho para o uso de outros protocolos de transferência, como por exemplo o *SSH*. A transformação do SGA em plugin diminuiu o acoplamento com o protocolo de comunicação entre o CSGrid e os nós de execução, que anteriormente exigia o uso de NFS ou CSFS para transferência e CORBA para execução e monitoração.

Os dois pontos de acoplamento no CSGrid através de plugins são:

– Plugin de Transferência: Neste contrato, o CSGrid instrui o plugin a copiar arquivos da *sandbox* para a área de projeto gerenciada pelo servidor CSGrid e vice-versa. Através desse ponto de acoplamento, além dos arquivos do projeto, são transferidos também os binários dos programas que serão executados.
– Plugin de Execução ou SGA: Neste contrato, o CSGrid instrui a execução de um comando, partindo do pressuposto de que todos os arquivos necessários já foram copiados para a *sandbox* do nó de execução (ou possuem acesso NFS). Uma vez submetido um comando para execução, este ponto de acoplamento também é usado para monitorar o andamento da execução e também para controlar seu possível cancelamento.

Neste trabalho vamos explorar essa arquitetura de plugins quando usada para realizar a integração com uma nuvem pública. O trabalho também funcionará como uma avaliação dessa arquitetura. Serão apresentados os contratos citados em maiores detalhes, considerando aspectos importantes para a elaboração do novo plugin voltado para a integração com a nuvem Azure. O formato relatado da API se trata do estado da arte em Maio/2015.

2.2.1 Contrato do Plugin de Transferência

O contrato do plugin de transferência teve sua origem no mesmo contrato adotado pelo CSFS e, portanto, está presente quando os nós de execução não compartilham os diretórios com o servidor do CSGrid via NFS. Nestes casos, o CSGrid orquestra a criação de uma *sandbox* no nó de execução para cada comando submetido. O plugin de transferência deve ser capaz de responder às seguintes requisições vindas do servidor CSGrid:

– **getProjectsRootPath()**: O plugin informa ao CSGrid o nome do diretório, no nó de execução, pai da subestrutura de diretórios que receberá as *sandboxes* para executar programas;
– **getAlgorithmsRootPath()**: O plugin informa ao CSGrid o nome do diretório, no nó de execução, pai da subestrutura de diretórios que receberá os binários dos programas executáveis;
– **getLocalTimestamps(caminho da sandbox)**: Dado um caminho no disco local do servidor CSGrid, informa o momento da última modificação de acordo com o sistema de arquivos para todos os arquivos dentro da subestrutura;

- **getRemoteTimestamps(caminho da sandbox)**: Dado um caminho no disco do nó de execução, informa o momento da última modificação de acordo com o sistema de arquivos para todos os arquivos dentro da subestrutura;
- **checkExistence(caminho)**: Responde sim ou não sobre a existência ou não do arquivo indicado, no nó de execução;
- **createDirectory(caminho)**: Cria um diretório no nó de execução;
- **copyTo(origem, destino)**: Copia um arquivo (ou diretório, recursivamente) do servidor CSGrid para o nó de execução;
- **copyFrom(origem, destino)**: Copia um arquivo (ou diretório, recursivamente) do nó de execução para o servidor CSGrid;
- **remove(caminho)**: Exclui um arquivo (ou diretório, recursivamente) do nó de execução.

Podemos observar que a API enumerada acima não inclui como parâmetro algum identificador do nó de execução para onde os arquivos serão copiados. Isto se dá para manter a compatibilidade com o SGA original, onde cada instância correspondia a um nó de execução e indicava o endereço de seu servidor CSFS dedicado. Assim, não havia ambiguidade das chamadas que referenciavam nós de execução. Com a evolução do CSGrid, novos casos de uso reais foram mapeados para trabalhar com redes NFS, o que poupava a necessidade do plugin de transferência distinguir nós de execução de forma explícita. Para os casos onde o nó de execução devesse ser diferenciado, aspectos que pudessem ficar ambíguos pela não definição do nó de execução passaram a ser resolvidos na fase de submissão do comando, pelo plugin de execução.

Para este contrato, não há definições de chamadas no sentido oposto, ou seja, não é previsto que o plugin de transferência invoque o servidor CSGrid.

2.2.2 Contrato do Plugin SGA

O contrato do plugin SGA teve sua origem no mesma interface usada pelo SGA descrito na Seção 2.1. Neste contrato, tanto o servidor CSGrid realiza chamadas para o plugin como o plugin também aciona o CSGrid. Portanto, serão ilustrados aqui os contratos referentes aos dois sentidos da comunicação. O serviço do CSGrid que é acionado pelo plugin SGA é denominado `SGAService`.

O servidor CSGrid, através de seu contêiner de plugins, ao instanciar um plugin SGA, o faz obrigatoriamente informando como parâmetro uma referência para os serviços do servidor que poderão ser chamados pelo plugin. Em especial, é necessário extrair deste a referência para o serviço `SGAService`.

Os principais métodos do `SGAService` chamados pelo plugin SGA são descritos a seguir:

- **registerSGA(sgaReference, sgaName, sgaProperties)**: Todo SGA, quando iniciado, deve se registrar no servidor CSGrid através deste método. Deve ser informada a referência para o objeto que implementa o contrato principal do executor (`sgaReference`), um nome que identifique unicamente a instância do SGA no CSGrid (`sgaName`) e um mapa de chaves de configuração (`sgaProperties`) que indicarão ao servidor CSGrid informações como o nome do plugin de transferência a ser utilizado (ou não), a plataforma na qual o SGA executa e os nomes de diretórios a serem considerados nos nós de execução para o plugin de transferência;
- **unregisterSGA(sgaReference, sgaName)**: Desregistra a instância do SGA do servidor CSGrid, dada a referência para a implementação do contrato principal do plugin e do nome que identifica o plugin;
- **updateSGAInfo(sgaReference, sgaName, sgaProperties)**: Este método deve ser chamado periodicamente pelo plugin SGA para indicar ao CSGrid que se mantém ativo. Caso contrário, o servidor CSGrid irá decretar o SGA como fora do ar. Ao ser chamado, informando a referência da implementação do SGA e de seu nome, este método pode também atualizar as informações sobre o plugin no servidor CSGrid.
- **commandCompleted(sgaName, command, cmdId, info)**: Este método deve ser chamado pelo SGA quando este detecta o término da execução de um comando. Quando o CSGrid recebe esta chamada, ele se encarrega de orquestrar a finalização da execução, que pode incluir a transferência dos arquivos de saída de volta para a área de projeto do usuário. O parâmetro `sgaName` deve identificar o próprio SGA, `command` e `cmdId` são referências para o comando que foi concluído e `info` se trata de uma estrutura que descreve o tempo decorrido na execução e outras informações sobre a execução do comando;
- **commandRetrieved(sgaName, infos)**: Na inicialização do plugin SGA, este pode detectar comandos submetidos porém não completos e atuar no mecanismo de tolerância a falhas. Tal situação pode ocorrer caso o servidor CSGrid seja finalizado durante a execução de algum comando. Nestes casos, este método deve ser chamado para informar a existência de comandos pendentes ou em execução. O parâmetro `sgaName` deve identificar o próprio SGA, e `infos` detalha o comando pendente;

– **commandLost(sgaName, cmdId)**: Durante a execução de um comando, caso o SGA detecte que um comando foi perdido no sistema de escalonamento de terceiros, este método deverá ser chamado pelo SGA para informar o CSGrid do ocorrido. O parâmetro `sgaName` deve identificar o próprio SGA e o parâmetro `cmdId` identifica unicamente a execução.

O contrato do plugin SGA é bem simples e sua interação com o contêiner de plugins e o *SGAService* está ilustrado na Figura 2.3. Os métodos *start()* e *stop()* são chamados pelo contêiner para notificar eventos do ciclo de vida do plugin e o método *setProperties()* é usado para definir as propriedades de configuração do plugin. A chamada para `commandRetrieved()` é opcional.

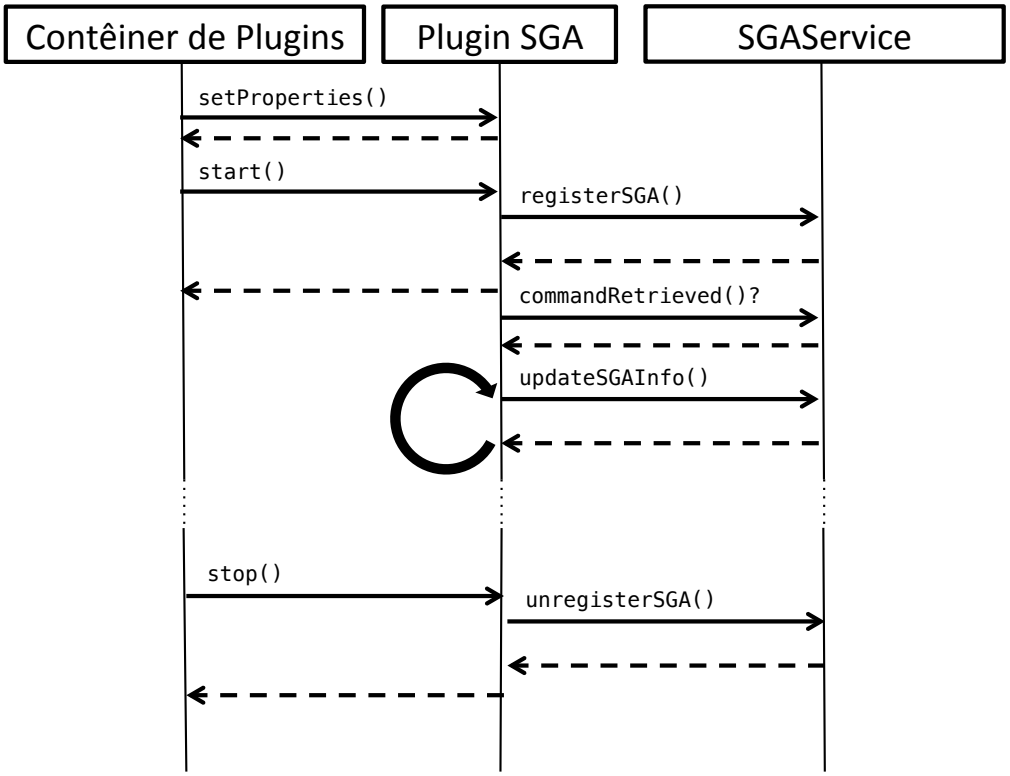

Figura 2.3: Interação básica entre o SGA, CSGrid e plugin de transferência

O plugin SGA deve implementar no contrato, além dos métodos *start*, *stop* e *setProperties* citados na Figura 2.3, também o método de submissão de comandos. Tal método é denominado *executeCommand*, e tem os parâmetros listados abaixo:

1. **command**: Parâmetro do tipo texto que indica a linha de comando que dispara um programa executável com os desejados parâmetros. Tipicamente o servidor CSGrid gera um script para Korn Shell (Bolsky e Korn, 1989) e passa neste parâmetro a chamada para o interpretador

(`/bin/ksh`) juntamente com o script gerado dinamicamente para executar o programa propriamente dito;

2. **cmdid**: Identificador único do comando. Trata-se de um texto curto contendo uma sequência única de caracteres usada internamente para identificar a execução, juntamente com o nome do usuário e do projeto CSGrid associados;

3. **extraParams**: Um conjunto de parâmetros no formato chave-valor, ambos em formato texto, representando outros parâmetros gerados pelo servidor CSGrid para executar o comando. Neste parâmetro, o servidor CSGrid informa os caminhos absolutos no sistema de arquivos do nó de execução onde se encontram os arquivos de entrada e o programa para execução.

A Figura 2.4 ilustra a interação entre o CSGrid, o SGA e o plugin de transferência. Inicialmente, o servidor CSGrid aciona o plugin de transferência para criar o diretório para a *sandbox* que será usada na execução do programa. Em seguida, solicita ao mesmo plugin que envie os arquivos, um por um, para o nó de execução. Tal procedimento de cópia inicial se aplica também para os binários dos programas que serão executados. Antes de qualquer cópia, o servidor CSGrid invoca os métodos *getLocalTimestamps()* e *getRemoteTimestamps()* para inferir a real necessidade da cópia. Caso os timestamps sejam idênticos, a cópia não é realizada. Após a cópia concluída do programa e dos arquivos de entrada para a *sandbox* no nó, o servidor CSGrid solicita a execução do comando para o plugin SGA, que retorna uma referência a ser usada futuramente para interrogar o andamento desta através do método *getRunningCommandInfo()*. Tal método é chamado periodicamente para a interrogação do status. Quando o término da execução é detectado, o servidor CSGrid então solicita ao plugin de transferência que traga apenas os arquivos modificados no nó de execução. Sabe-se quais são os arquivos modificados através da prévia chamada de *getLocalTimestamps()* e *getRemoteTimestamps()*.

O servidor CSGrid não delega a criação da *sandbox* de execução para o SGA. Para instruir a execução de um programa, o CSGrid antes orquestra a criação da *sandbox* através do plugin de transferência e, ao instruir a execução do programa, já informa os caminhos dos diretórios definitivos. Cabe então ao SGA executar o programa usando os diretórios informados, sob a premissa de que já estão carregados.

O CSGrid provê uma estrutura interna para parametrização dos programas pelo usuário final onde cada programa deve fornecer um arquivo em formato XML onde os possíveis parâmetros dos programas são descritos junto

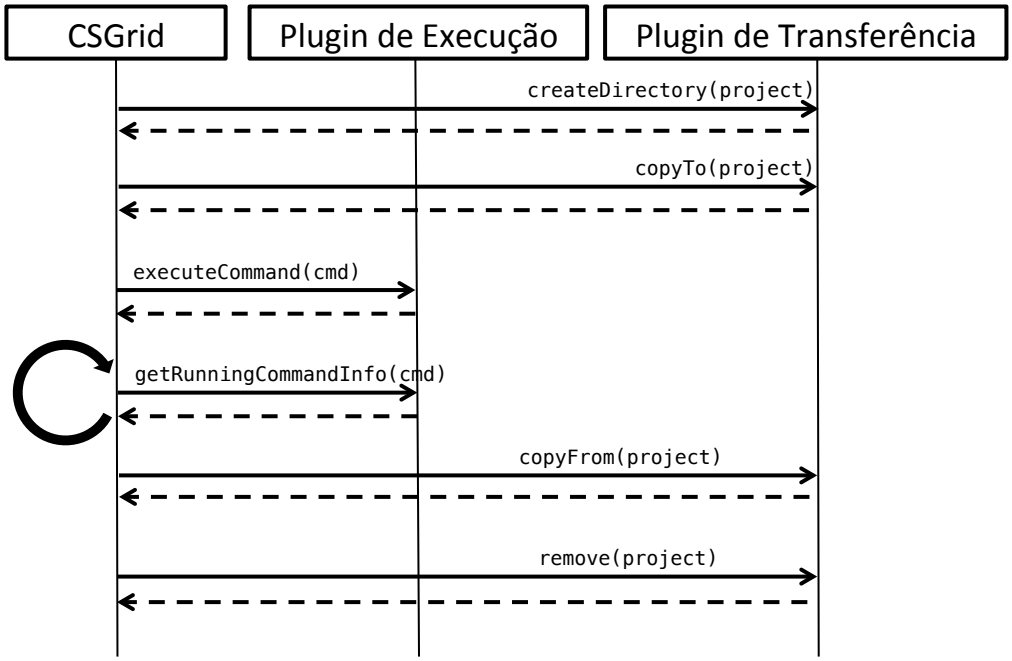

Figura 2.4: Interação básica entre o SGA, CSGrid e plugin de transferência

com seus respectivos domínios. Tais parâmetros são mapeados para a linha de comando do programa no momento da execução, e o formato dos parâmetros também é descrito no arquivo XML citado. O servidor CSGrid, no ato da submissão do comando, cria dinamicamente um script para Korn Shell onde toda a parametrização da execução é encapsulada. Os parâmetros definidos pelo usuário são embutidos neste script e o que o SGA executa é este script (vide parâmetro `command`, de `executeCommand`), e não o programa propriamente dito. Assim, o SGA não é informado diretamente sobre os parâmetros de entrada específicos do programa, a não ser que o SGA abra o arquivo de script e extraia os parâmetros. O parâmetro extra `input_files_path` (de *extraParams* em *executeCommand*) visa informar de forma explícita a listagem de todos os arquivos de entrada parametrizados para o programa, que serão informados na linha de comando em sua execução. Em adição, tal parâmetro informa também os demais arquivos de controle do CSGrid que foram copiados para a *sandbox*. Trata-se da maneira formal de informar o SGA sobre arquivos da *sandbox*. Assim, é possível que o SGA, na fase de submissão do comando, possa realizar ações com tais arquivos, possivelmente os enviando para um nó de execução mais específico, identificado só após a fase relacionada ao plugin de transferência.

O retorno esperado de `executeCommand` é uma referência que representa especificamente o comando criado no SGA, que pode ser executado imediatamente ou não, de acordo com a implementação do plugin de execução. Tal referência informada pelo SGA será usada pelo servidor CSGrid para acom-

panhar o andamento da execução do comando, assim como realizar ações específicas sobre o comando em questão. Os métodos que devem ser implementados pelo plugin de execução neste contrato que representa um comando específico são listados abaixo:

- **getRunningCommandInfo()**: Este método deve retornar um mapa relacionando chaves nomeadas a valores texto. As chaves do mapa em questão são pré-definidas e indicam o status da execução do comando e da máquina onde está sendo executado;
- **control(ação)**: Este método permite o cancelamento, a suspensão ou a retomada de um comando.

A API de plugins do CSGrid será amplamente utilizada neste trabalho visando um propósito para o qual ela não foi originalmente desenhada. No caso, trata-se de implementar um SGA cujos nós de execução são máquinas localizadas em uma nuvem pública. Com o exercício da implementação, foi produzida uma crítica sobre os contratos de acoplamento definidos entre o CSGrid e seus plugins. Através da experiência com um caso real, espera-se contribuir para a evolução do CSGrid em suas futuras versões da API de plugins.

Tendo apresentado a API de integração do servidor CSGrid com plugins de SGA e de transferência, serão apresentados agora aspectos relevantes do paradigma de computação em nuvem e, a seguir, resultados de medições de desempenho específicas com a nuvem Azure que serão levados em conta no projeto do plugin SGA Azure.

3
Computação em Nuvem

Considerando a intenção de uso do CSGrid integrado com uma nuvem pública, é importante salientar os conceitos úteis sobre computação em nuvem. Considerando a virtualização de máquinas, tecnologias como a da VMWare (VMWare Virtualization for Desktop and Server) ou o VirtualBox (Oracle VM VirtualBox) provêem soluções para desktops, onde o sistema operacional denominado "hospedeiro" é executado diretamente no hardware (Windows, Linux, MacOS, ...) e, através do programa virtualizador, outra máquina completa é simulada, rodando outra instância de um sistema operacional qualquer. As tecnologias de virtualização evoluíram para o ponto onde os próprios fabricantes de hardware contemplam instruções assembly dedicadas à virtualização, visando garantir o máximo desempenho e mínima sobrecarga com um ou mais sistemas operacionais executando dentro de suas *sandboxes* (Adams e Agesen, 2006). A computação em nuvem nada mais é do que a possibilidade de se alugar uma máquina virtual na internet para instalar qualquer sistema operacional ou aplicação, com a garantia de que ela estará logicamente isolada de máquinas de outros usuários.

Paralelamente, provedores de computação em nuvem também oferecem serviços de armazenamento dedicados de forma independente do aluguel de máquinas virtuais, além de outros serviços para customização de infra-estrutura como firewalls e balanceadores de carga. Tais serviços oferecidos por um provedor de nuvem são divididos nas categorias *infra-estrutura como serviço* (IaaS), *plataforma como serviço* (PaaS) e *software como serviço* (SaaS). Considerando uma pilha de tecnologias que vai desde a máquina física até a aplicação apresentada ao usuário final, a Figura 3.1 ilustra as diferenças entre cada uma das três categorias de serviços no que diz respeito à responsabilidade pela administração dos elementos envolvidos em uma aplicação.

A camada IaaS consiste principalmente na oferta de máquinas virtuais para livre utilização, privando o usuário de aspectos ténicos como lidar com hardware físico, localização geográfica precisa, particionamento de dados, backup, entre outros. O provedor de nuvem administra a rede, o armazenamento (físico), os servidores físicos e o sistema de virtualização. O sistema operacional das máquinas fica sob responsabilidade do usuário que solicitou a máquina virtual. Assim, todos os aspectos de segurança e configuração no nível do sistema operacional em diante não recaem sobre o provedor da nuvem em si. Também são oferecidos como IaaS serviços como armazenamento de blobs, firewalls,

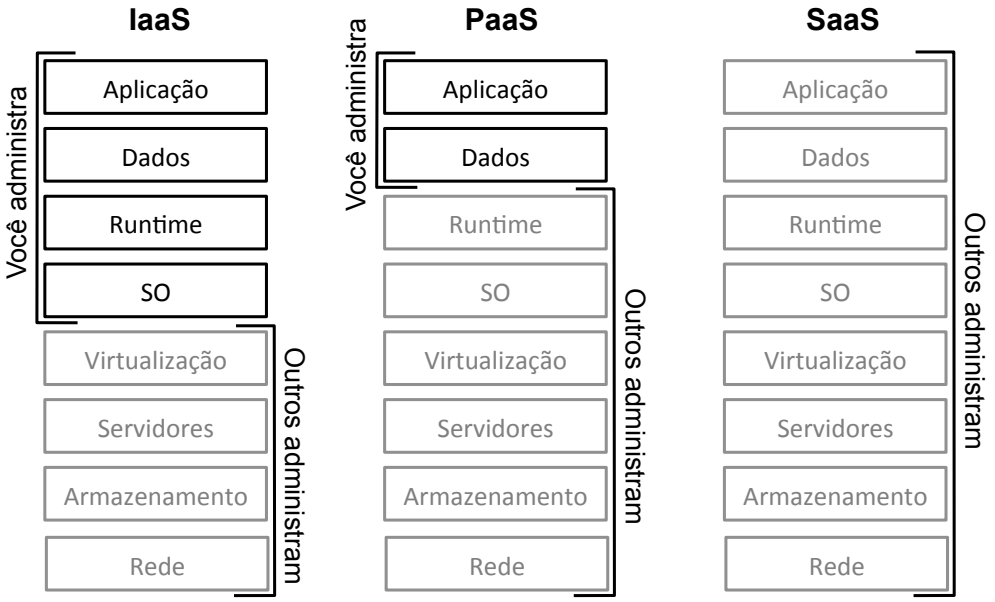

Figura 3.1: Níveis de serviços oferecidos por nuvens

VLANs e repositórios de imagens para máquinas virtuais.

Já na camada PaaS o usuário não possui acesso ao sistema operacional das máquinas que executam suas aplicações, ficando todas as garantias de segurança e estabilidade neste nível sob responsabilidade do provedor da nuvem. Um exemplo típico deste caso são hospedagens de sites onde o servidor http é administrado pelo provedor da nuvem, porém todo o site e seus dados são de responsabilidade do usuário da nuvem.

SaaS já trata da aplicação final a ser oferecida para outros desenvolvedores ou para o usuário final, tipicamente aplicações com interface web. Tais aplicações são caracterizadas por serem softwares disponíveis sob demanda, não cabendo instalação na estação de trabalho do usuário final. Todos os dados da aplicação e a execução da aplicação em si ficam encapsulados na nuvem.

3.1 Microsoft Azure

A nuvem pública Microsoft Azure está, na atualidade, no grupo dos maiores provedores de nuvem pública, principalmente com relação aos seus recursos de IaaS e PaaS, provendo APIs programáticas via HTTP que permitem o controle de toda a infra-estrutura de máquinas virtuais e armazenamento.

Na camada de IaaS, a Azure provê a instanciação de máquinas virtuais onde toda a gestão pode e deve ser feita pelo usuário, desde o sistema operacional até as aplicações que lá irá executar. Há APIs HTTP para realizar as operações como criação e exclusão de máquinas virtuais do ambiente, tornando o manuseio agnóstico com relação à linguagem de programação que invoca a API. Desta forma, são implementadas bibliotecas em algumas linguagens de programação comumente usadas como Python, JavaScript, Java e .NET.

A nuvem Azure provê um catálogo fixo de tamanhos de máquinas virtuais, com preços distintos por hora de funcionamento. Ao se alugar uma máquina virtual, deve ser levada em conta sua localização física, pois o data center onde a máquina será instanciada faz parte dos parâmetros que são informados para a Azure, e conforme o local no globo selecionado, o preço varia. Mais a seguir veremos a lista dos data centers disponibilizados pela Azure com mais detalhes. A Tabela 3.1[1] apresenta configurações e preços de máquinas disponíveis. Os preços são apresentados em relação ao custo da configuração de máquina intitulada A1, sempre considerando o data center do leste dos EUA. Assim, por exemplo, sabemos que a máquina A4 custa oito vezes mais do que a A1. Os valores exatos, em moeda, podem ser obtidos no site oficial do serviço Azure. As máquinas da série D (D1, D2, etc...) se diferenciam das máquinas da série A (A0, A1, etc...) devido ao fato de possuírem discos SSD e núcleos virtuais cerca de 60% mais rápidos do que os da série A.

Ainda como IaaS, há um serviço de armazenamento próprio com acesso exclusivamente através de API HTTP. Tal serviço permite que aplicações sejam capazes de trabalhar com as seguintes estruturas de dados:

1. Blobs: Cada blob (*binary large objects*) consiste em uma unidade nomeada com armazenamento ilimitado organizado em uma estrutura não hierárquica. Cada unidade é comumente tratada como um arquivo. Não se trata de uma estrutura de diretórios, mas tal estrutura pode facilmente ser representada através da utilização de separadores de diretório dentro dos nomes dos blobs, o que é permitido;

[1]Dados coletados em Maio/2015

Nome	Núcleos	RAM	Disco	Preço Relativo
A0	1	0.75 GB	20 GB	0,33
A1	1	1.75 GB	70 GB	1,00
A2	2	3.5 GB	135 GB	2,00
A3	4	7 GB	285 GB	4,00
A4	8	14 GB	605 GB	8,00
A5	2	14 GB	135 GB	4,17
A6	4	28 GB	285 GB	8,33
A7	8	56 GB	605 GB	16,67
D1	1	3.5 GB	50 GB	1,57
D2	2	7 GB	100 GB	3,13
D3	4	14 GB	200 GB	6,27
D4	8	28 GB	400 GB	12,53
D11	2	14 GB	100 GB	3,97
D12	4	28 GB	200 GB	7,93
D13	8	56 GB	400 GB	14,28
D14	16	112 GB	800 GB	25,7

Tabela 3.1: Relação das máquinas disponíveis para provisionamento

2. Filas: O usuário pode criar um número arbitrário de filas que oferecem uma API própria para entrada e consumo de mensagens. Este serviço de armazenamento de filas é projetado para alta escalabilidade, podendo não garantir uma ordenação FIFO perfeita. Porém, o serviço garante que nenhuma mensagem é perdida. Cada mensagem postada neste serviço pode ter no máximo 64Kb. Não há formas de se inscrever como observador da fila de forma a receber notificações automáticas em caso de novas mensagens. Para se observar uma fila do serviço de armazenamento, é necessária uma abordagem iterativa e constante (polling);

3. Tabelas: Tabelas estruturadas de dados com índices distribuídos podem ser armazenadas diretamente no serviço de armazenamento de forma não-transacional. Voltado para a escalabilidade, é um serviço útil para armazenamento de dados que não requerem junções complexas, chaves estrangeiras e que possam ser desnormalizados para ganho de desempenho.

Sharan (Shahan et al.) documenta as metas de desempenho para o serviço de armazenamento da Microsoft Azure. Os principais indicadores para este trabalho são reproduzidos na Tabela 3.2[2].

A Azure oferece ainda, como serviço IaaS, bancos de dados relacionais proprietários, processadores de mídia e integrações com plataformas de celula-

[2]Dados coletados em Junho/2015

Recurso	**Limite**
Tamanho máximo de um blob	500TB
Número máximo de blobs	Ilimitado até que o tamanho de todos os blobs não exceda 500TB
Taxa de transferência para um único blob	Até 60MB/s ou 500 requisições por segundo
Taxa de transferência de entrada para uma conta do serviço de armazenamento em data centers dos Estados Unidos	20Gbps sem redundância geográfica
Taxa de transferência de saída para uma conta do serviço de armazenamento em data centers dos Estados Unidos	30Gbps sem redundância geográfica

Tabela 3.2: Metas de Escalabilidade do Serviço de Armazenamento da Microsoft Azure

res. Em particular, a Azure oferece um serviço denominado *Service Bus*, que é um servidor pub-sub para distribuição de mensagens através de tópicos ou filas independente do serviço de armazenamento citado acima. As filas de um serviço *Service Bus* oferecem uma garantia maior de ordenação FIFO, podendo ter até 256Kb em seu tamanho total, e o serviço permite o consumo de mensagens de forma bloqueante, sem a necessidade da realização de polling. Como se trata de um serviço independente do serviço de armazenamento, o *Service Bus* possui uma tarifação própria.

Em se falando de PaaS, a Azure oferece uma abstração chamada de *Cloud Service*, onde ao invés de máquinas virtuais totalmente gerenciáveis pelo usuário, entrega um serviço em maior grau de abstração. Um *Cloud Service* consiste na combinação de *Web Roles* e de *Worker Roles* que executam uma aplicação tipicamente de interface web. As instâncias denominadas *Web Roles* são os servidores web propriamente ditos, provendo acesso HTTP para a aplicação. O usuário apenas especifica o tamanho do servidor web em um catálogo de configurações e, se desejar, parametriza a elasticidade automatizada, que faz com que novos servidores HTTP sejam iniciados com o mesmo site em caso de aumento de carga. Não se pode atuar sobre o servidor web ou sobre o sistema operacional do mesmo, pois a configuração em baixo nível é de responsabilidade da Azure. Porém, o usuário conta com um catálogo de servidores de tecnologias distintas para usar. Uma vez a Azure administrando o servidor web, não recaem sobre o usuário os aspectos de segurança e desempenho próprios do contêiner. Como se trata de um serviço de

hospedagem web, também são providas facilidades para *deploy* e gestão destas aplicações. Através da API HTTP, é possível instruir a criação do espaço para o site com seus parâmetros de escalabilidade. Para o envio dos arquivos do site ou aplicação web, pode ser usado o protocolo FTP.

Os *Worker Roles*, por sua vez, são instâncias que consomem trabalhos de uma fila e também são gerenciados pela Azure, estando sujeitos a ajustes automáticos relacionados a elasticidade e segurança. Tais instâncias executoras trabalham consumindo dados de uma fila do serviço de armazenamento e executando um código desenvolvido obrigatoriamente na plataforma própria Microsoft, um framework baseado na tecnologia .NET. Tal restrição tecnológica torna a utilização dos *Worker Roles* indesejada para o uso planejado neste trabalho.

3.1.1 Sites

O serviço de nuvem Azure, assim como seus concorrentes, oferecem infraestrutura em vários data centers espalhados geograficamente, de forma que o usuário deve escolher sempre em qual data center as máquinas virtuais ou contas de armazenamento serão instanciados. Naturalmente, a localização geográfica das máquinas virtuais e das contas de armazenamento influencia diretamente no desempenho para transfência de dados na própria Azure e também entre uma estação de trabalho do usuário final e a máquina virtual como na PUC-Rio, por exemplo. A Tabela 3.3[3] lista da localidades onde a Microsoft Azure possui data centers.

Algumas das localidades são reservadas para uso governamental, já as demais são submetidas à legislação do país onde se encontram. Como os custos de manutenção de um data center variam conforme o país, o preço da locação de recursos da nuvem também varia conforme o data center selecionado. Por exemplo, em Julho de 2015, alugar uma máquina virtual com configuração A1 no Brasil custa 22% mais caro do que alugar a mesma máquina virtual no data center do leste dos Estados Unidos.

3.1.2 Tarifação

O uso dos recursos da nuvem envolvem custos financeiros. Sob um apelo de "pague apenas o quanto usar", os serviços de nuvem elencam seus recursos e geram tarifação de forma bastante semelhante, cobrando por tudo o que se usa: máquinas, rede, armazenamento e serviços. Para ilustrar com um exemplo, são

[3]Dados coletados em Junho/2015

Tabela 3.3: Relação das regiões geográficas da Microsoft Azure

Região Nomeada	**Localização Geográfica**
EUA Central	Iowa
Leste dos EUA	Virgínia
Leste dos EUA 2	Virgínia
Governo dos EUA de Iowa	Iowa
Governo dos EUA de Virgínia	Virgínia
Centro-Norte dos Estados Unidos	Illinois
Centro-Sul dos Estados Unidos	Texas
Oeste dos EUA	Califórnia
Norte da Europa	Irlanda
Europa Ocidental	Holanda
Ásia Oriental	Hong Kong
Sudeste da Ásia	Cingapura
Leste do Japão	Tóquio
Oeste do Japão	Osaka
Sul do Brasil	Estado de São Paulo
Austrália Oriental	Nova Gales do Sul
Sudeste da Austrália	Vitória

abordados os preços específicos da Microsoft Azure relacionados aos recursos convenientes para este trabalho:

- Máquina virtual: Paga-se pelo valor da máquina virtual por hora em que um hardware estiver alocado para ela. Uma máquina virtual também exige o espaço no serviço de armazenamento para guardar a imagem de seu disco local;
- Serviço de armazenamento: Paga-se pelo total armazenado, levando em conta a política de redundância escolhida, e pelo número de transações realizadas no serviço. São consideradas transações quaisquer chamadas para a API com o intuito de consultar, ler, apagar ou escrever. Por exemplo, realizar polling em uma fila pode implicar em cobrança por este critério;
- Tráfego de rede: Paga-se pelo volume de dados transferido através das fronteiras do serviço de nuvem. O tráfego interno não é tarifado.

Para ilustrar de forma mais concreta os preços totais do uso da computação em nuvem, vamos utilizar os valores em dólares americanos coletados em Junho/2015 da fonte oficial da Microsoft Azure:

- Máquina virtual D1: U$0,085/hora;
- Armazenamento de discos de máquinas virtuais: U$0,05/GB/Mês;
- Armazenamento de arquivos: U$0.024/GB/Mês;

– Tráfego de rede acima de 5GB/Mês: U$0,181/GB/Mês.

Para se criar uma máquina virtual, é necessário armazenar o seu disco no serviço de armazenamento, o que também gera tarifação. Vamos considerar uma máquina D1, que possui um disco de 50GB e um mês de 30 dias ou 720 horas. Ao criar uma máquina virtual D1, estaremos pagando o preço de seu disco pelas horas em que ela existir. É possível não excluir o disco do serviço de armazenamento de forma a reusá-lo quando uma nova máquina virtual for criada, mas não vamos trabalhar com esta hipótese agora. Com esses parâmetros, podemos inferir o preço total de uma máquina D1 por hora, que vai equivaler aos U$0,085 já citados mais $50GB \times (U\$0,05/720h) = U\$0,0884$, ou seja, quase nove centavos de dólar americano. Com mais máquinas virtuais, o valor cresce na proporção direta.

Mesmo que nenhuma máquina virtual esteja alocada, é provável que o serviço de armazenamento esteja sendo utilizado para guardar, por exemplo, imagens de discos virtuais a serem usadas para criar as máquinas virtuais ou arquivos a serem utilizados por executores ou projetos. Por isso, não devemos considerar que o preço de não se ter nada executando na nuvem é zero. Este custo de base irá depender diretamente da estratégia de execução de programas adotada e do número de imagens de máquinas virtuais armazenados de forma permanente ou reusável.

3.1.3 API Azure

Para compreender aspectos da implementação do plugin SGA voltado para a Azure, é necessário se ter uma visão mais detalhada da API programática disponibilizada pela nuvem pública em questão. A parte relevante da API será mostrada aqui, pois integralmente esta se faz muito extensa e muitas vezes fora do escopo deste trabalho. Os aspectos que serão apresentados aqui são os da API do sistema de armazenamento, do sistema de filas e tópicos (*Service Bus*) e do sistema de provisionamento de máquinas virtuais.

Toda a API é originalmente acessada através de protocolo HTTP, enviando e recebendo mensagens codificadas como XML. Independentemente do formato usado para transmitir as solicitações e recuperar os resultados, serão abordados aqui os principais parâmetros usados e seus domínios. Parâmetros considerados irrelevantes para o escopo deste trabalho não serão apresentados. A referência completa para os parâmetros se encontra na documentação oficial da API Azure (Azure - Microsoft's Cloud Computing Platform).

Criação de Máquinas Virtuais

Toda máquina virtual na Azure deve estar associada a um *Cloud Service*. Portanto, devemos inicialmente criar o *Cloud Service*.

O comando *CreateHostedService* admite os seguintes parâmetros:

- **Nome do *Cloud Service***: Nome do *Cloud Service*. Este nome será usado no nome DNS que será visível da Internet e deve ser único;
- **Localização**: Em qual zona geográfica será alocado, onde uma das zonas listadas na Seção 3.1.1 deve ser selecionada;

Uma vez criado o *Cloud Service*, podemos criar a máquina virtual propriamente dita. Os parâmetros informados para o comando *CreateVirtualMachineDeployment* são:

- **Nome do *Cloud Service***: Nome do *Cloud Service* onde a máquina virtual será alocada;
- **Nome**: Nome da máquina virtual, que a identificará unicamente dentro do *Cloud Service*;
- **Imagem VHD de Origem do Sistema Operacional**: Este parâmetro serve para indicar qual é a imagem a ser usada para a criação da nova máquina virtual. As imagens de sistemas operacionais ficam no serviço de blobs, porém devem constar listadas em um repositório específico para este fim;
- **Caminho para o VHD da máquina**: Uma nova imagem de disco rígido será criada para a nova máquina virtual, iniciando como uma cópia da imagem indicada como origem do sistema operacional. Deve ser informado o caminho onde a nova imagem será armazenada no serviço de blobs;
- **Sistema Operacional**: Este parâmetro indica se o sistema operacional da máquina virtual é Linux ou Windows e é necessário porque a nuvem acessa a máquina virtual na sua criação para configurar a rede e as credenciais de acesso remoto;
- **Credenciais para Acesso Remoto**: No caso de sistema operacional Linux, devem ser informadas as informações para acesso remoto via SSH, seja através de senha ou através de par de chaves;
- **Dados Customizados**: Trata-se de um campo aberto e opcional. Quando informado, um arquivo com nome pré-definido é criado no disco virtual da máquina cujo conteúdo é o que foi passado por este parâmetro.

Esse conteúdo pode ser útil para injetar dados na nova máquina virtual no momento de sua criação. Normalmente é usado para injetar senhas de acesso ou chaves de acesso que os programas de dentro da máquina virtual vão precisar;

– **Tamanho da Máquina Virtual**: Deve ser indicado o tamanho desejado para a máquina virtual de acordo com a Tabela 3.1.

Para se instanciar uma máquina virtual, portanto, basta chamar *CreateHostedService* e depois *CreateVirtualMachineDeployment*. É possível provisionar múltiplas máquinas virtuais dentro de um mesmo *Cloud Service*, porém não é possível provisioná-las simultaneamente. Criando um *Cloud Service* para cada máquina virtual viabiliza-se o provisionamento em paralelo.

Serviço de Blobs

No serviço de blobs, o usuário deve criar pelo menos uma entidade denominada *conta*. Uma *conta* é vinculada a uma localização geográfica (vide Tabela 3.3) e a um esquema de redundância. Em uma *conta*, espaços para armazenamento denominados *contêineres* devem ser criados para armazenar blobs. Todos os blobs estão obrigatoriamente em um *contêiner*.

A criação de um blob leva, além dos parâmetros triviais, um conjunto opcional de propriedades livres no formato "chave-valor" textuais que o usuário pode usar à sua conveniência.

O endereço oficial de um blob fica sendo:

`https://conta.blob.core.windows.net/contêiner/nome do blob`

Não é definida nenhuma relação hierárquica entre blobs, não cabendo o conceito de diretório. Porém, é possível nomear blobs usando caracteres de separação de diretórios como '/'. Assim, um blob cujo nome é "frutas/abacate" pode ser criado de forma a emular uma estrutura de diretórios dentro do *contêiner*. Devido a esta característica, no serviço de blobs dizemos que "frutas" é o *prefixo* do blob "frutas/abacate". É possível realizar buscas em *contêineres* baseadas em prefixos de blobs.

Service Bus

Assim como no serviço de blobs, deve ser criada uma *conta* no *Service Bus*. Dentro de uma *conta*, deve ser criado pelo menos um *namespace*, que por sua vez irá abrigar as filas e tópicos para transmissão de mensagens.

As entidades "filas" provêem um mecanismo de transmissão de mensagens com garantia FIFO (*first-in, first-out*) onde a recuperação de mensagens

pode ser feita por vários clientes simultaneamente de maneira bloqueante. Apenas um cliente receberá cada mensagem enviada para a fila.

As entidades do tipo "tópico" também provêem um mecanismo de transmissão de mensagens com garantia FIFO. Porém, devem ser criadas entidades denominadas "assinaturas". Cada assinatura é dedicada a um só tópico, e clientes devem recuperar as mensagens das assinaturas. Uma assinatura de um tópico é equivalente a uma caixa-postal onde as mensagens enviadas para o tópico são guardadas até que o cliente a recupere. Com isto, um tópico tem o efeito de distribuir cópias das mensagens para todas as assinaturas e, consequentemente, para vários clientes.

A API de envio de mensagens para tópicos ou filas, além dos parâmetros triviais e do próprio conteúdo da mensagem, traz a possibilidade de definir um tempo para que a mensagem expire e não seja mais recuperável, o que pode ser útil para a transmissão de mensagens de status voltadas para acompanhamento em tempo real: Só faz sentido que a mensagem seja recuperada caso tenha sido submetida recentemente.

No próximo capítulo, são apresentados experimentos feitos usando a API Azure e resultados úteis para a elaboração do novo SGA.

4
Medições de Desempenho da Microsoft Azure

O objetivo deste capítulo é apresentar medições de desempenho feitas sobre a arquitetura de nuvem Azure, visando embasar decisões de projeto feitas para sua utilização na execução de programas submetidos pelo CSGrid. Os custos aqui apresentados são primordialmente de ordem computacional, tendo seus custos monetários avaliados apenas quando necessário.

Para entender os custos computacionais associados à utilização da nuvem, executamos algumas medidas de desempenho utilizando a plataforma Azure. A escolha dos critérios e medições estão diretamente relacionados às principais etapas envolvidas na submissão de programas pelo CSGrid para execução na nuvem pública através de um SGA dedicado. Serão apresentadas nas Seções 4.1 e 4.2 resultados de medições baseadas do uso direto dos recursos da nuvem Azure. Em 4.3, já serão apresentados resultados de medições que envolvem uma instalação CSGrid e vários SGAs na nuvem Azure em sua forma original. Finalmente, na Seção 4.4 será abordada a eficiência de se movimentar dados armazenados no serviço de armazenamento de blobs.

4.1
Provisionamento de Máquinas Virtuais

Uma das principais vantagens do uso de nuvens é a possibilidade de criar e destruir máquinas virtuais sob demanda, pagando apenas pelo tempo em que as máquinas forem usadas. Porém, o provisionamento de uma nova máquina virtual leva tempo e, para aplicações HPC, este tempo pode ser relevante.

Na Azure, cada máquina virtual pertence a uma entidade chamada *Cloud Service*, já descrita na Seção 3.1. Cada *Cloud Service* é exposta na Internet pública com um nome DNS próprio, mapeado para uma rede interna virtual. Para um mesmo *Cloud Service*, não é possível executar provisionamentos simultâneos de máquinas virtuais, mas em *Cloud Services* independentes, é possível realizar o provisionamento em paralelo.

A Tabela 4.1 apresenta os tempos de criação de máquinas virtuais em diferentes medições ao longo de um dia. Os detalhes sobre as máquinas podem ser vistos na Tabela 3.1. O tamanho da máquina não influi no tempo de criação. O término do processo de criação indica que a máquina está ligada e pronta para uso. O tempo médio de 40 segundos observado para criação de uma máquina, considerando que criaremos máquinas virtuais de forma serializada, indica que o tempo esperado para a criação de um cluster de n

máquinas é de 40n segundos, o que indica que a criação de máquinas sob demanda não é conveniente em um cenário HPC, salvo por particularidades no escalonamento. Criando máquinas em paralelo este tempo pode ser bastante reduzido, porém se mantém significativo. Dado que esperamos provisionar as máquinas virtuais necessárias na véspera da execução do programa, no momento em que a execução for demandada pelo servidor CSGrid, poder provisionar várias máquinas em paralelo torna-se de fundamental importância para atender a demanda com o mínimo de sobrecarga relacionada à preparação do ambiente na nuvem.

Tamanho	**Média**	**Desvio Padrão**
A0	41,7	2,8
A1	40,8	2,6
A2	40,8	2,0
A3	41,6	2,3
A4	42,6	3,7
A5	41,4	2,7
A6	41,1	2,6
A7	39,9	1,1
D1	40,3	1,3
D2	40,2	0,9
D3	41,7	3,0
D4	42	2,2
D11	40,5	1,9
D12	42,4	2,9
D13	40,3	1,3
D14	40,1	1,2

Tabela 4.1: Tempos medidos para criação e destruição das máquinas, em segundos

4.2 Transferência de Arquivos entre Máquinas e Data Centers

Considerando que a execução de programas na nuvem pública envolve a preparação do ambiente, e tal preparação envolve o envio dos binários dos programas, assim como dos arquivos de entrada para os programas, é importante se ter uma idéia da eficiência da transferência de dados entre um servidor CSGrid, que não está localizado na nuvem, até a nuvem. Analogamente, é importante se ter uma idéia do quão eficiente é trazer os resultados computados da nuvem de volta para o servidor CSGrid.

Nesta seção relatamos medições de taxas de transferência através de `ssh`:

1. Entre a PUC-Rio e os data centers da Azure distribuídos pelo globo;

2. Entre duas máquinas virtuais dentro do mesmo data center;

3. Entre data centers distintos.

Não consideramos o cenário onde se contrata um barramento de rede exclusivo de 40Gbit/s, pois esse torna o valor da hora-máquina cerca de 50% maior, além de só ser suportado por máquinas de 8 e 16 núcleos que só estão disponíveis em poucos data centers. Para ilustrar como amostra representativa de transferências, consideramos algumas combinações possíveis de data centers cujo desempenho pudesse representar as demais. Como se trata de um ambiente compartilhado, um valor mais exato está sujeito às condições de cada momento. Escolhemos combinações de forma a ilustrar a transferência entre a PUC-Rio e os data centers, e os data centers entre si de forma geograficamente distantes. Em dois casos, medimos o desempenho da cópia entre duas máquinas dentro do mesmo data center, incluindo o caso mais rápido com relação à PUC-Rio.

Cada medição levou em conta a média de três cópias de um arquivo de 256MB com conteúdo aleatório através de um canal criptografado SSH. Foram usadas máquinas virtuais de tamanho A1 ou small para tal, que possuem um núcleo de processamento e 1.75GB de RAM. A sobrecarga de entrada e saída para o disco das máquinas virtuais também foi considerada, pois as cópias envolvem leituras e escritas em disco.

A Figura 4.1 apresenta as taxas de transferência citadas.

Podemos observar que as taxas de transferência quando dentro do mesmo data center se aproximam de 10MB/s. Quando as transferências são entre data centers do mesmo continente, o desempenho cai para cerca de 1MB/s, e transferências entre data centers da Azure em continentes distintos apresenta queda de 50% com relação ao melhor desempenho. Quando se trata de enviar dados do campus da PUC-Rio para a Azure, o desempenho se mostrou superior ao de trazer dados para o Rio de Janeiro, relação esta que se confirmou em repetições do experimento feitas em dias e horários diferentes.

4.3 Medição de Desempenho para Transferências Concorrentes

Um cenário bastante comum em aplicações HPC é o de execuções concorrentes de programas em nós de execução distintos onde cada nó copia para si os dados de entrada para executar o programa e depois envia os resultados da computação de volta para uma entidade centralizadora. Considerando cenários em que vários nós de execução são iniciados simultaneamente, várias máquinas virtuais irão então acessar simultaneamente o mesmo repositório de dados para recuperar seus arquivos de entrada ou os próprios binários a serem executados.

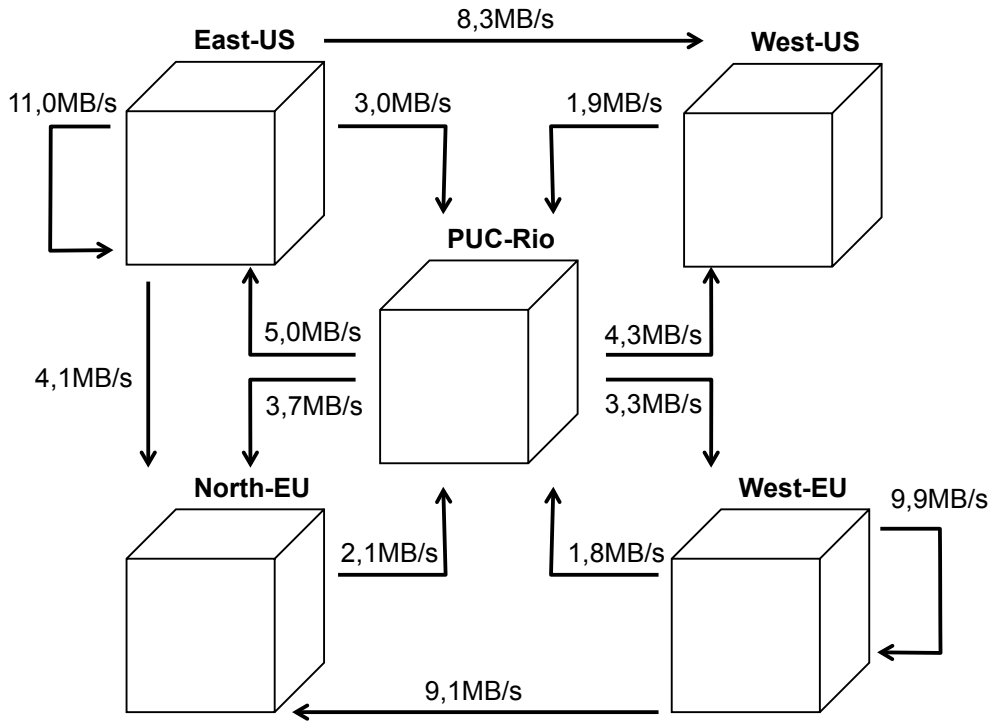

Figura 4.1: Medidas de tempo para transferência `ssh` entre diferentes localidades

Essa concorrência para o acesso de dados envolvida na inicialização de programas executados massiva e simultaneamente é o objeto de medição nesta seção. Com este experimento, procura-se estudar o impacto das diferentes formas de se transferir os arquivos do servidor CSGrid para os nós de execução de forma a embasar decisões de arquitetura do SGA Azure proposto mais a seguir. Trata-se de um caso específico onde um arquivo grande deve ser enviado e trazido do SGA (io-bound), portanto este experimento não visa retratar casos onde a quantidade de computação é muito grande perante o uso de I/O (cpu-bound).

O objetivo aqui é comparar o comportamento de diferentes técnicas de transferência de arquivos perante variações no grau de paralelismo para acesso de leitura e escrita. Para tal, foram construídos três cenários baseados no CSGrid no ambiente de nuvem com o seu SGA original. Particularmente neste experimento, o servidor CSGrid é também uma máquina virtual na nuvem, dentro do mesmo site físico em que as demais máquinas virtuais serão provisionadas como nós de execução. Assim, as transferências envolvidas acontecem apenas dentro da Azure, fazendo com que o servidor CSGrid e os nós de execução estejam sempre dentro da mesma rede privada. Os três cenários levados em conta para a experimentação estão a seguir:

1. Rede NFS, que é o caso mais comum de clusters CSGrid, onde as áreas de projeto e os programas são compartilhados entre o servidor CSGrid e os

nós de execução através de diretórios compartilhados via NFS. Utilizando esta configuração, todas as máquinas virtuais provisionadas são clientes NFS do servidor CSGrid ou todas as máquinas serão clientes NFS de um outro servidor dedicado a servir arquivos;

2. CSFS, que é implementado quando não é viável ter o compartilhamento NFS. Neste mecanismo, ao invés de diretórios compartilhados entre máquinas, usa-se a cópia integral dos dados e programas antes da execução e dos resultados no término da mesma. O protocolo utilizado no experimento para se transferir os dados foi CORBA;

3. Azure Blob Storage: Neste cenário, não utilizamos nem NFS e nem CSFS. Para recuperar os arquivos da área de projeto e executar comandos, o SGA recorre ao serviço de armazenamento provido pela infraestrutura do Windows Azure. Assim como o CSFS, os arquivos são recuperados do serviço da Azure antes da execução do comando, e depois são enviados de volta.

A idéia deste experimento é comparar o desempenho de um programa cujo objetivo é tentar comprimir um arquivo de 100Mb de conteúdo aleatório. Devido à entropia do arquivo, é esperado que a compressão não reduza em nada o tamanho original, mas o esforço computacional é levado em conta. Nota-se, no entanto, que o procedimento deve exigir um bom desempenho de rede para carregar a área de projeto no SGA.

Para comparar o desempenho na nuvem, o programa foi executado inicialmente em um só nó de execução, depois em dois nós de execução simultaneamente, depois em três, e assim por diante. Quanto mais nós simultâneos, maior a concorrência na rede para transferir o arquivo de 100Mb para a máquina virtual do nó e trazê-lo de volta.

Para o experimento, cada execução do programa foi separada em três fases:

- Fase 1: Montagem da *sandbox* com o arquivo de entrada no nó de execução. Esta etapa envolve sempre a transferência do arquivo uma vez até o diretório criado no disco local do nó de execução antes de executar o programa;
- Fase 2: Disparo do programa propriamente dito, executando-o na *sandbox* citada;
- Fase 3: Envio do arquivo de resultado de volta da *sandbox*. Esta etapa envolve uma transferência dos arquivos modificados ou criados durante

a execução do programa. No caso deste experimento, um único arquivo é transferido de volta.

Excepcionalmente para a implementação do cenário 3, com Azure Blob Storage, o programa de cópia e compressão foi feito utilizando o SDK Java da Azure para interação com o serviço de storage. Os demais são constituídos de um script shell Linux que executa um comando "tar" com fator de compressão gzip. O código-fonte Java usado para a medição no cenário 3 pode ser visto no Apêndice B.

As execuções na modalidade Azure e NFS foram feitas 10 vezes, e os tempos reportados são a média aritmética das execuções. No caso do CSFS, apenas uma execução foi feita devido a falhas não determinísticas apresentadas pela implementação no cenário dinâmico da nuvem[1].

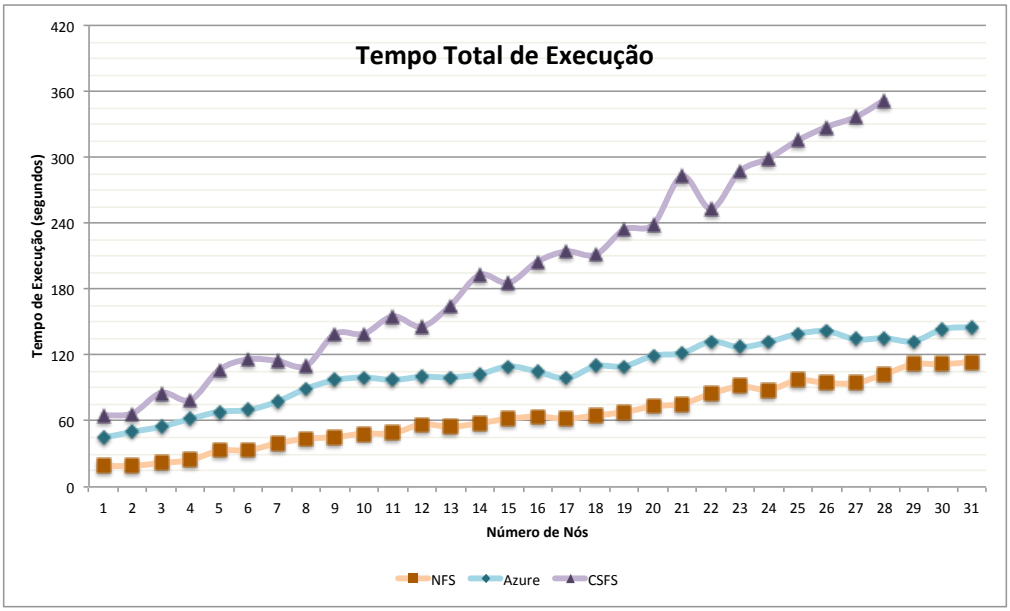

Figura 4.2: Medição do tempo total de execução simultânea do programa de compressão de 100Mb

A Figura 4.2 mostra o tempo a partir do início da execução do primeiro nó de execução até o término total da execução do último. Por exemplo, disparando 30 nós para a realização da compressão simultaneamente em um cluster NFS, todos os 30 completaram a tarefa depois de cerca de um minuto e quarenta segundos (100 segundos). Este é o tempo visível para o usuário final, que vai desde quando os comandos foram submetidos até o momento em que ele vê todos os comandos totalmente concluídos na interface gráfica do CSGrid.

[1]O mecanismo CSFS se mostrou instável em situações onde a máquina SGA foi reiniciada ou SGAs são removidos ou inseridos sem que o servidor CSGrid fosse reiniciado junto. Assim, desalocar e alocar máquinas para o experimento constantemente para a realização dos testes causou falhas que impediram a execução fluida dos testes em massa

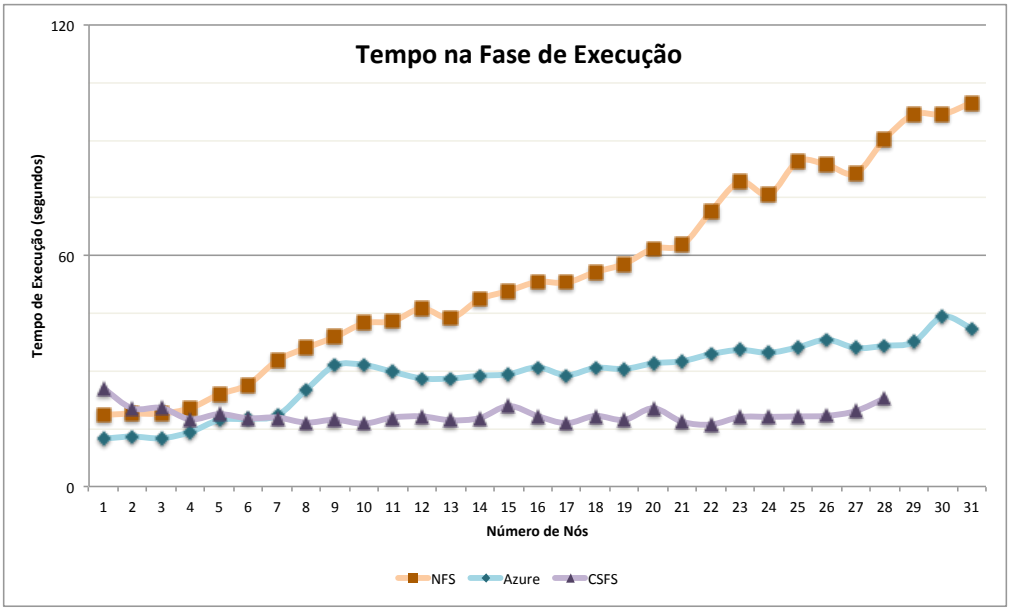

Figura 4.3: Medição do tempo de execução do programa de compressão de 100Mb em n nós

A Figura 4.3 mostra os tempos apenas da fase 2 descrita acima. Note que o tempo da execução do modelo NFS cresce conforme o número de máquinas virtuais concorrendo pelo arquivo. Isto porque a compressão neste cenário ocorre ao mesmo tempo em que o arquivo é recuperado. No caso do CSFS ou Azure, independente do tempo de download ou upload, o tempo isolado da compressão permanece estável. A diferença na curva do Azure para a curva do CSFS vem por conta da diferença da implementação dos programas utilizados (o Azure é um programa Java, enquanto os demais são um *shellscript*). Vale, assim, analisar o comportamento da curva ao invés dos valores absolutos: A única curva com comportamento ascendente é a do NFS, enquanto as demais não são influenciadas pelo aumento do número de máquinas virtuais concorrentes.

A Figura 4.2 mostra que o desempenho do modelo NFS e do Azure são semelhantes. Até o limite de 30 nós, a concorrência pelo mesmo dado impacta o desempenho geral de forma semelhante nos dois módulos. Tendo em mente que as soluções para escalabilidade em função de requisições concorrentes do serviço de armazenamento da Azure e de um servidor NFS são diferentes, podemos especular sobre o comportamento das curvas para um número bem maior de máquinas. No caso do NFS, utilizando um único servidor de arquivos, a degradação pode manter sua piora linear. O serviço de armazenamento da Azure oferece a garantia de desempenho para até 500 acessos por segundo e leituras e escritas a 60MB/s para um mesmo arquivo, como listado na Tabela 3.2. A degradação por parte da Azure se faz semelhante à degradação do

NFS para até 31 nós. Porém, no NFS, os arquivos eram recuperados pelos nós de execução na medida em que eram comprimidos, ou seja, a taxa de transferência exigida ao longo do processo foi diluída ao longo de toda a execução do programa, favorecendo a distribuição do tráfego entre os nós. Na estratégia com Azure, os downloads foram requisitados integralmente antes do início da compressão, exigindo dedicação exclusiva dos recursos para a recuperação e escrita dos arquivos, trazendo um maior stress para o servidor do arquivo. Como a degradação das duas curvas foram semelhantes, e dada a última diferença de comportamento citada, podemos intuir que o mecanismo de armazenamento Azure responde melhor a downloads simultâneos em uma escala maior, porém seguindo a recomendação de 500 requisições por segundo. Como recomendação do provedor de nuvem, caso a demanda por leituras simultâneas de um mesmo blob ultrapasse os 500 e a sobrecarga da recuperação se torne significativa, deve-se usar uma CDN(Buyya et al., 2008) para distribuir o conteúdo, ao invés de usar o serviço de armazenamento diretamente.

4.4 Medição de Desempenho do Serviço de Blobs

Na seção anterior, vimos que o serviço de armazenamento de blobs apresenta um desempenho satisfatório para transferências concorrentes entre o serviço em si e várias máquinas virtuais, o que representa impacto na etapa de inicialização de nós de execução. Visando melhorar a percepção do desempenho em particular do desempenho do serviço de armazenamento de blobs, fizemos medições dedicadas apenas a ele, sem levar em conta transferências concorrentes.

Para quantificar o desempenho com que blobs são recuperados e salvos no sistema de armazenamento, fizemos downloads e uploads sucessivos de arquivos de 1GB a partir e para o disco rígido de uma máquina virtual no data center do Leste dos Estados Unidos para um contêiner de blobs instanciado na mesma região, sem nenhuma configuração de redundância geográfica. Foram medidos uploads a uma taxa de 17,15MB/s, a um desvio padrão de 0,27MB/s. Downloads foram feitos a um pouco mais, à média de 21,88MB/s com desvio padrão de 1,10MB/s. Como se pode constatar, o desempenho de um único download ou upload de um único blob se apresenta aquém da meta ilustrada na Tabela 3.2. Porém, como discutido na Seção 4.3, o gráfico da Figura 4.2 ilustra como recuperações e envios de arquivos de forma concorrente influenciam no desempenho, mostrando que um mesmo blob acessado de forma concorrente por várias máquinas pode chegar a uma taxa de transferência total até maior do que meta estipulada, 60MB/s dependendo das condições da rede.

O mesmo teste também foi feito a partir do campus da PUC-Rio, e constatando taxas de upload para a Azure de 4,64MB/s, a um desvio padrão de 0,79MB/s, e taxas de download da Azure para a PUC de 1,08MB/s, com desvio padrão de 0,22MB/s. Nota-se, de acordo com o já relatado no experimento com `ssh` da Seção 4.2, que as taxas para transferências de dados para a Azure são privilegiadas com relação às transferências de dados da Azure.

Apesar do relatado na Seção 4.2, downloads via `ssh` se mostraram mais lentos do que o via Blob Store da Azure. Apesar de ambos possuírem criptografia (o Blob Store usa protocolo HTTPS), motivos relacionados à infra-estrutura de transferência, armazenamento e aos algoritmos específicos influem nos resultados. Ao se reproduzir os experimentos com `ssh` na mesma janela de tempo em que os experimentos com Blob Store, as taxas de download `ssh` se mantiveram abaixo de 1MB/s, enquanto as taxas de upload se mantiveram em números semelhantes.

A variação da taxa de transferência entre a PUC-Rio e a Azure do Leste dos Estados Unidos com relação a medições em datas e horários distintos reduz as conclusões sobre as taxas de transferência para respostas relativas: Sabemos então que enviar dados é bem mais rápido do que recuperar dados, assim como as taxas de envio e recebimento em ambos os protocolos são compatíveis. As taxas de transferência medidas apenas dentro dos próprios data centers da Azure (i.e., sem trafegar dados de ou para a PUC-Rio) não apresentaram variações significativas ao longo da reprodução dos testes.

4.5 Medição de Desempenho para Uso Intensivo de CPU

Uma vez apresentadas medições de desempenho baseadas em taxas de transferências de dados, é importante agora medir o desempenho relativo da computação intensiva de CPU, sem a interferência das sobrecargas de entrada e saída. Escolhemos um sistema real para se realizar um teste voltado para o desempenho do uso de CPUs das máquinas providas pela nuvem Azure. Com isto, esperamos criar uma boa intuição sobre o desempenho de um programa que não faz operações de entrada e saída e usa intensivamente todos os núcleos de CPU disponíveis quando comparado com uma máquina física. Para tal comparação, o mesmo experimento também foi executado em uma máquina desktop real de dentro do campus da PUC-Rio. O sistema escolhido é o Tatu, da Alis/Tecgraf/PUC-Rio, por se tratar de um simulador que faz uso de computação intensiva, pouco uso de disco, e nenhum uso de rede. O sistema Tatu foi desenvolvido para ajudar a resolver uma questão relevante relacionada à exploração de petróleo cujo método numérico pode despender

alguns minutos de CPU dependendo de seus parâmetros. Uma breve descrição sobre a computação realizada pelo TATU pode ser vista no Apêndice A.

O Tatu é um programa de uso intensivo de CPU, cujos arquivos de entrada são bastante pequenos (poucos KBs) e os arquivos de saída podem possuir alguns MBs. Cada execução do sistema é individual e não necessita de várias entradas independentes. O programa que executa a simulação possui implementação que explora ao máximo os núcleos disponibilizados pela máquina. Assim, é um projeto que pode ser executado na nuvem usando máquinas de vários núcleos para avaliação de desempenho. Caso sejam constatados cenários com mais execuções, os recursos da nuvem poderão ser melhor aproveitados.

Realizamos execuções usando dois tipos de máquinas virtuais da Azure, a série A e a série D, apresentadas na Seção 3.1. Para oferecer uma base de comparação, executamos o mesmo programa em uma máquina física, não virtual. O hardware nesse caso é uma máquina Intel Core2 Quad 2.83GHz, com 4GB de memória rodando Windows, localizada no Labpos, do Departamento de Informática da PUC-Rio.

O programa Tatu é compilado para plataforma Windows, explorando o paralelismo da CPU através da biblioteca OpenMP(Dagum e Enon, 1998). Desta forma, ele pôde ser executado em plataforma Linux utilizando o pacote Wine(Amstadt e Johnson, 1994), que possui a finalidade de executar programas compilados para Windows em Linux. Para o nosso experimento, a máquina desktop do Labpos executou a versão Windows diretamente, enquanto a versão na nuvem executou o binário com o adaptador Wine, que não apresenta sobrecarga devido ao formato do programa Tatu, que é voltado apenas para cálculo numérico, não interagindo com o sistema operacional.

Núcleos	A-Series	D-Series	PC-Labpós-8 cores
1	174,87	108,66	58,00
2	95,38	58,24	38,13
4	52,31	32,45	32,74
8	31,59	19,18	26,54

Tabela 4.2: Medição do tempo de execução do programa Tatu com diferentes números de threads, em segundos.

A Figura 4.4 mostra a comparação dos tempos em escala logarítmica. A Tabela 4.2 traz os valores que fundamentaram o gráfico. Para as medições usando Windows Azure, o número de threads é igual ao número de núcleos da máquina virtual alocada para executar o programa Tatu. No caso da máquina física, como o número de núcleos informado pelo sistema operacional é sempre oito (quatro núcleos com tecnologia hyper-threading), o número de threads do programa foi variado através de parâmetro específico para tal. O intúito

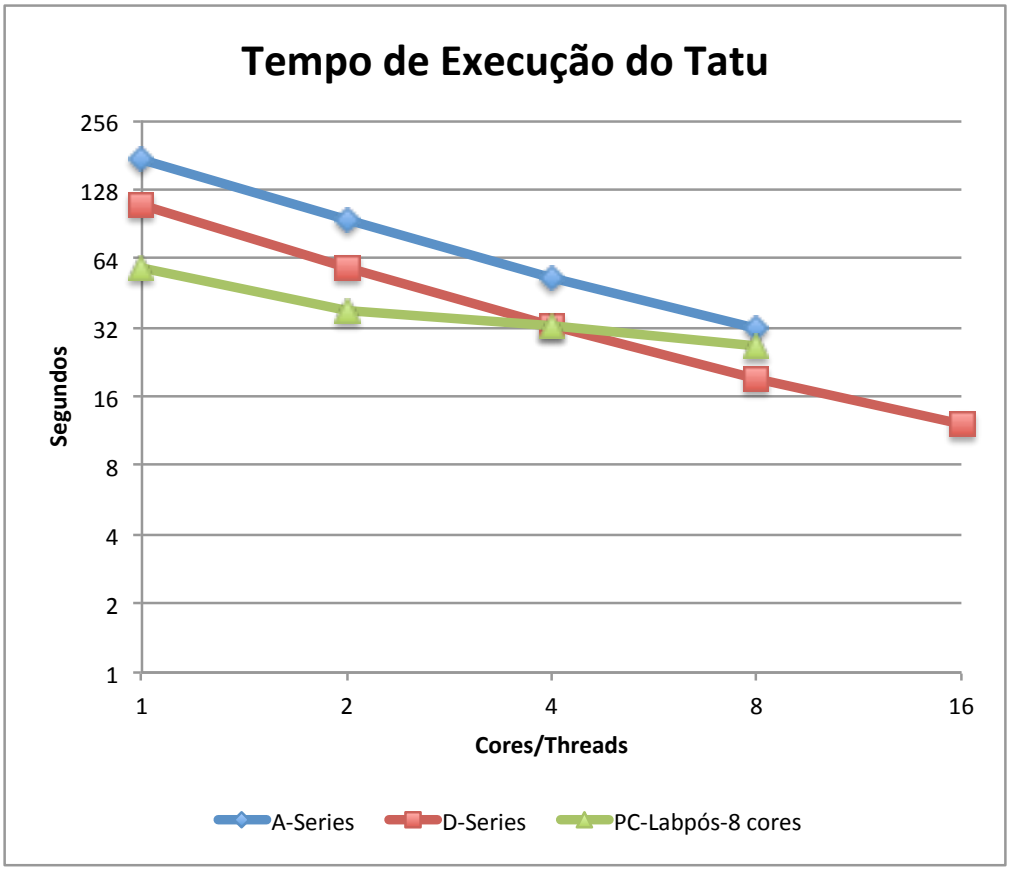

Figura 4.4: Medição do tempo de execução do programa Tatu com diferentes números de threads

da execução na máquina real citada não é o de fazer uma comparação de desempenho criteriosa, mas sim o de trazer a intuição sobre a diferença de desempenho entre as máquinas virtuais e uma máquina real com configuração tipicamente usada para executar o Tatu por seus usuários finais. Infelizmente, não foi possível desligar efetivamente os núcleos da máquina real, o que influenciou no formato de sua curva em particular.

A redução dos tempos de execução considerando apenas a mudança de máquina virtual de A-series para D-series se mostrou bastante significativa. Nos testes de desempenho, os tempos para execução nas máquinas D-series foram equivalentes a, em média, 61% do tempo de execução nas máquinas A-series com o mesmo número de núcleos. Com relação à máquina real para referência, o desempenho utilizando quatro núcleos se mostrou semelhante ao da máquina virtual D-series de também quatro núcleos, até que a máquina D-series com o mesmo número de núcleos da máquina real executou então o procedimento de maneira mais eficiente. Este resultado ilustra o baixo impacto que a virtualização pode ter sobre aplicações que fazem uso intensivo de CPU.

Observando a Tabela 3.1, podemos relacionar os preços das máquinas virtuais com o desempenho do programa Tatu. As máquinas D-series possuem

preço 57% maior do que a A-series em todos os tamanhos. Isso indica que o ganho de desempenho das máquinas D-series traz vantagem apenas com relação ao tempo de execução, mantendo o preço no mesmo patamar. Por exemplo, o programa levou em média 19,2 segundos para executar em uma máquina D-series enquanto levou 31,6 segundos na A-series, ambas com oito núcleos. O tempo da D-series equivale a 60,1% do tempo da A-series. Como o preço relativo da máquina D-series é 57% maior, a diferença dos valores pagos para executar os programas fica $\frac{19,2 \times 1,57}{31,6} \cong 95\%$. Ou seja, para oito núcleos, usar uma máquina D-series equivale a uma economia financeira de 5%. Para as execuções do Tatu neste experimento, a economia foi de cerca de 2% para máquinas virtuais de um núcleo, e valores intermediários entre 2% e 5% para máquinas de dois e quatro núcleos. Portanto, conclui-se que vale a pena utilizar as máquinas D-series para aplicações de uso intensivo de CPU mesmo financeiramente.

5
Trabalhos Relacionados

Para dar continuidade ao embasamento necessário para a construção do SGA Azure, fizemos uma prospecção por tecnologias já implementadas com propósitos semelhantes ou grande afinidade. Com isso, visamos absorver a experiência intrínseca de quem precisou resolver problemas semelhantes ao que o CSGrid resolve, porém utilizando recursos de nuvens públicas.

5.1
Virtualização com HTCondor

O sistema de escalonamento de comandos do HTCondor (nele denominados "jobs") possui uma forma de executar programas baseada na instanciação de máquinas virtuais. Quando parametrizado para executar programas no universo de máquinas virtuais, deve ser informada qual a imagem da máquina a ser utilizada e qual o sistema virtualizador que deverá ser utilizado. Deste modo, a execução se dá através da instanciação da imagem de máquina virtual indicada, que deverá estar pré-programada para executar a tarefa em sua inicialização e, ao término, se desligar automaticamente. O HTCondor só dará a execução como concluída quando a máquina virtual estiver desligada.

Nota-se, portanto, que não se trata necessariamente de uma nuvem, pois não há instanciação de máquinas virtuais com hardware sob demanda, há sim o uso direto de virtualizadores nos nós de execução. Um uso autêntico do paradigma de computação em nuvem aliado ao escalonador do HTCondor foi feito em outro projeto chamado Cloud Scheduler, que será apresentado a seguir.

5.2 Cloud Scheduler

O sistema Cloud Scheduler foi desenvolvido pela Universidade de Victoria visando aplicações de física de partículas e astronomia que envolvem grande movimentação de dados e simulações. Em sua descrição (Armstrong et al., 2010), são citadas execuções que duram seis horas e produzem 100GB de resultado.

O Cloud Scheduler utiliza um repositório de imagens de máquinas virtuais que pode ser usado na execução dos comandos na nuvem. Assim, o usuário deve fornecer junto dos demais parâmetros para a execução de seu programa, a referência para a imagem de máquina virtual necessária para se executar o comando, assim como o número de máquinas virtuais a serem utilizadas.

O mecanismo de escalonamento escolhido neste caso foi o HTCondor, por já possuir como requisito a utilização de máquinas heterogêneas de forma oportunista. Para usar o Cloud Scheduler, o *daemon startd* do HTCondor deve estar instalado nas imagens das máquinas virtuais que serão indicadas pelo usuário. O Cloud Scheduler executa em paralelo com o escalonador do HTCondor, consultando toda a fila de comandos periodicamente para se manter atualizado sobre as demandas. Com base na fila do próprio HTCondor, o Cloud Scheduler instancia as máquinas virtuais necessárias. Na inicialização das máquinas virtuais, o *daemon startd* do HTCondor registra a nova máquina automaticamente como um novo membro do cluster e o módulo executor do HTCondor faz todo o resto como se a execução fosse em um cluster comum, transferindo o programa executável e arquivos de entrada e saída. Da mesma forma que o CloudScheduler monitora a fila de comandos do HTCondor, ele também monitora as máquinas virtuais para desligar as que se declararem em estado de falha ou que não estejam mais sendo usadas.

Apesar de o escalonador de comandos do HTCondor ditar a ordem de execução, o Cloud Scheduler pode exercer influência sobre a ordem com a qual os comandos são executados através de variações na ordem da criação das máquinas virtuais, por exemplo, criando primeiro uma máquina virtual adequada para a execução de um comando que não é o primeiro da fila de execução. Assim, o Cloud Scheduler pode forçar quotas de recursos por usuário, não permitindo que um único usuário tome todos os recursos do sistema para si alocando suas próprias imagens de máquinas virtuais. Caso o Cloud Scheduler detecte que é necessário liberar recursos para favorecer a justiça entre os usuários, ele pode também desligar máquinas virtuais durante a execução de um comando. Neste caso, o escalonador do HTCondor reiniciará o comando

automaticamente de acordo com sua política.

O Cloud Scheduler é, portando, um sistema que aloca e desaloca máquinas virtuais que atuam como nós de execução para um HTCondor, usando imagens definidas na própria parametrização do comando HTCondor.

5.3 Azure Batch

Em Julho/2015, a Microsoft lançou a primeira versão oficial e aberta do sistema Azure Batch, que se propõe a ser um gerenciador de execução de comandos usando sua infra-estrutura de máquinas virtuais. Trata-se de um gerenciador de comandos com API HTTP que atua como escalonador e executor de comandos em máquinas virtuais. Desta forma, o Azure Batch se comporta de forma semelhante ao Cloud Scheduler, gerenciando uma fila de execuções e possibilitando a instanciação de máquinas virtuais sob demanda ou usando máquinas já existentes.

Apesar de trabalhar como o HTCondor, ainda há a restrição de que o sistema operacional dos nós de execução seja da família Windows, pois o sistema de execução que roda dentro de cada nó é baseado em tecnologia compatível apenas com este. Assim, apenas programas compilados para plataforma Windows podem ser submetidos ao Azure Batch.

Na parametrização de um comando a ser submetido para o escalonador Azure Batch, a indicação sobre os binários e arquivos de entrada são informadas sob a forma de referências para blobs guardados no serviço de armazenamento. Assim, quando os nós de execução são acionados, todos os arquivos necessários são recuperados e, então, o programa é executado. No ambiente de execução de cada nó, é disponibilizado um diretório que é compartilhado por todos os nós de forma que o resultado da computação possa ser agrupado. É de responsabilidade da aplicação organizar os arquivos de saída do programa, assim como transferí-lo de volta para o usuário final.

No que diz respeito à elasticidade, o Azure Batch permite definir o número de nós de execução através de uma fórmula, com linguagem própria, onde são definidas variáveis que podem ser usadas para computar o número de nós de execução dedicados. Tais variáveis envolvem métricas como uso de CPU médio entre os nós ativos, memória total, estatísticas de entrada e saída de disco e rede, fila de execução e data/hora. Uma vez definida uma fórmula, o Azure Batch reajusta o número de nós a cada quinze minutos baseado no resultado calculado.

5.4 Outro Uso do CSGrid para Executar Programas na Nuvem

Assim como a Microsoft Azure, há várias nuvens no mercado e algumas formas de se manusear seus recursos, dependendo do fornecedor. Muitas dessas tecnologias viabilizaram o que se chama de nuvem privada, ou seja, uma instituição possui seu próprio hardware, porém deseja que este abrigue sistemas operacionais virtualizados. Motivações para trabalhar desta forma incluem custo, possibilidade de melhor customização da infra-estrutura e sensibilidade dos dados com os quais se irá trabalhar. Com isso, tira-se vantagem da flexibilidade provida pelo paradigma de computação em nuvem, porém utilizando recursos próprios. Nestes casos, é comum haver apenas máquinas virtuais como recursos, sendo o restante da IaaS apresentada (filas e armazenamento) opcionais.

Em um universo onde existem nuvens privadas de diversas instituições, cabe a mesma motivação para compartilhamento de recursos que originou o que se conhece por computação em grade. Com isso, o problema de compartilhamento de recursos permanece, porém agora sob a ótica de virtualização. Compartilhar apenas hardware possibilitando que outros possam instanciar seus sistemas através de imagens de máquinas virtuais é vantajoso porque isenta o dono do hardware de dar manutenção no sistema operacional que executa as aplicações de terceiros.

Um exemplo de sistema para realizar a federação de nuvens privadas é o Fogbow (Barros et al., 2015), desenvolvido na Universidade Federal de Campina Grande (UFCG). Através de um gerenciador, uma instituição pode ceder hardware de sua nuvem privada para ser usado por terceiros interessados em instanciar máquinas virtuais por lá e depois se comunicar com elas através de um canal seguro.

No escopo do projeto EUBrazilCC[1] não consta o escalonamento e execução de programas. Foi implementado um SGA específico para integração do servidor CSGrid com o Fogbow, onde uma máquina é requisitada e toda a interação entre o servidor CSGrid e o nó de execução é feita através do protocolo SSH.

No SGA Fogbow, quando uma execução é demandada, uma nova máquina virtual é sempre provisionada. Em seguida, todos os binários são copiados para ela, após sua inicialização, utilizando o comando SCP (cópia de arquivos utilizando o protocolo SSH). Só então o programa é executado na máquina virtual através de comandos shell enviados pelo SGA através de SSH. Ao término da execução, o SGA copia os arquivos modificados na *sand-*

[1]Vide http://www.eubrazilcloudconnect.eu (capturado dia 02/12/2015)

box da máquina virtual através de SCP novamente. Finalmente, a máquina é desprovisionada, não sendo reaproveitada para um novo comando.

6
O SGA Azure

Neste capítulo, apresentamos a arquitetura do novo SGA dedicado ao uso da nuvem pública Azure, assim como as decisões de projeto feitas baseadas nos resultados expostos no Capítulo 4. O SGA Azure traz dois artefatos principais em sua arquitetura: O plugin do servidor CSGrid (tanto para cópia como para execução) e o programa *executor*, que é executado nos nós de execução. Na Seção 6.1, apresentamos uma visão geral do SGA Azure e as principais decisões de projeto tomadas com relação ao uso dos recursos da nuvem como blobs e provisionamento de máquinas virtuais. Na Seção 6.2, abordamos em maior detalhe aspectos técnicos do SGA Azure e seu *executor*.

6.1
Visão Geral

O plugin SGA Azure é o responsável por provisionar novas máquinas virtuais de acordo com a demanda. A política escolhida para tal consiste em alocar máquinas virtuais dedicadas à execução de cada comando, aumentando o número de máquinas virtuais no caso de maior demanda de comandos a serem executados. O *executor* é projetado para trabalhar sempre um comando por vez. Assim, não é possível que uma mesma máquina virtual execute mais de um comando simultaneamente. Vimos na Seção 4.1 que o tempo de provisionamento não é influenciado pelo tamanho da máquina. Porém, é mais vantajoso em termos monetários utilizar máquinas com o tamanho adequado para o programa em questão.

O plugin de transferência do SGA Azure realiza cópias de e para o serviço de armazenamento de blobs, que é usado como repositório central na nuvem para programas e arquivos de entrada e saída. Como o servidor CSGrid não se localiza dentro da nuvem, as transferências de dados envolvem a Internet e estão sujeitas a variações de desempenho. Já o executor que roda dentro das máquinas virtuais realiza transferências de e para o serviço de armazenamento de forma bem mais eficiente (cerca de 10MB/s, como descrito na Seção 4.2). Com isso, transferências redundantes entre o servidor CSGrid e o serviço de armazenamento são poupadas, e não nos preocupamos tanto com as transferências para as máquinas virtuais.

A Figura 6.1 mostra a interação entre o servidor CSGrid, o plugin SGA Azure e o plugin de transferência Azure durante a execução de um comando, estendendo o diagrama de sequência já apresentado na Figura 2.4. No

diagrama, “Azure” representa os serviços oferecidos pela Azure através de sua API programática. A coluna “Executor” se refere ao *daemon* executor que roda nas máquinas virtuais. Há simplificações no que diz respeito aos processos de polling, detalhados a seguir. Inicialmente, o servidor CSGrid aciona o plugin de transferência para carregar os arquivos de entrada do projeto, já destinados à *sandbox*. O plugin de transferência por sua vez executa as operações necessárias no serviço de armazenamento de blobs. Uma vez criada a *sandbox*, o servidor CSGrid solicita a execução do comando para o SGA Azure. Este cria uma mensagem com todos os dados sobre o comando e envia para a fila do *Service Bus* criada para tal fim. O executor, em uma máquina virtual ativa, recebe a mensagem e inicia o processo de execução, que passa por fazer o download dos artefatos necessários do serviço de armazenamento, executar o comando, e depois fazer o upload do resultado de volta para o serviço de armazenamento. Para cada etapa, o executor envia uma mensagem com o status para um tópico no *Service Bus* criado para este fim. O SGA Azure mantém linhas de execução para recuperar as mensagens do tópico que detalha o status das execuções. Quando o executor termina de enviar os resultados da execução de volta para o serviço de armazenamento, este envia uma mensagem de status indicando o término da execução. Ao receber esta mensagem de status, o plugin SGA Azure notifica o servidor CSGrid sobre o término da execução do comando. Neste ponto, o servidor CSGrid invoca o plugin de transferência para trazer de volta os resultados da computação realizada na máquina virtual.

6.2 Arquitetura Interna do Executor

Nesta seção, apresentamos em maior detalhe aspectos técnicos do SGA Azure e seu *executor*.

A Figura 6.2 ilustra a proposta de arquitetura do novo SGA Azure. Este SGA usufrui dos serviços específicos da Azure para provisionamento de máquinas virtuais, acesso às filas de mensagens e armazenamento.

Escolhemos como serviço de comunicação o sistema de filas do *Service Bus*, que possui bibliotecas para controle programático com espera bloqueante por novas mensagens. O SGA trabalha com mensagens em formato texto estruturadas em JSON (The JSON Data Interchange Standard) para troca de informações sobre comandos a executar e status dos executores.

Dada a flexibilidade e a capacidade de escala discutida na Seção 4.3, o armazenamento provido pela IaaS será usado como ambiente para troca de dados entre o servidor CSGrid e os nós de execução. Assim, os arquivos da *sandbox* serão enviados para a nuvem antes mesmo da máquina virtual

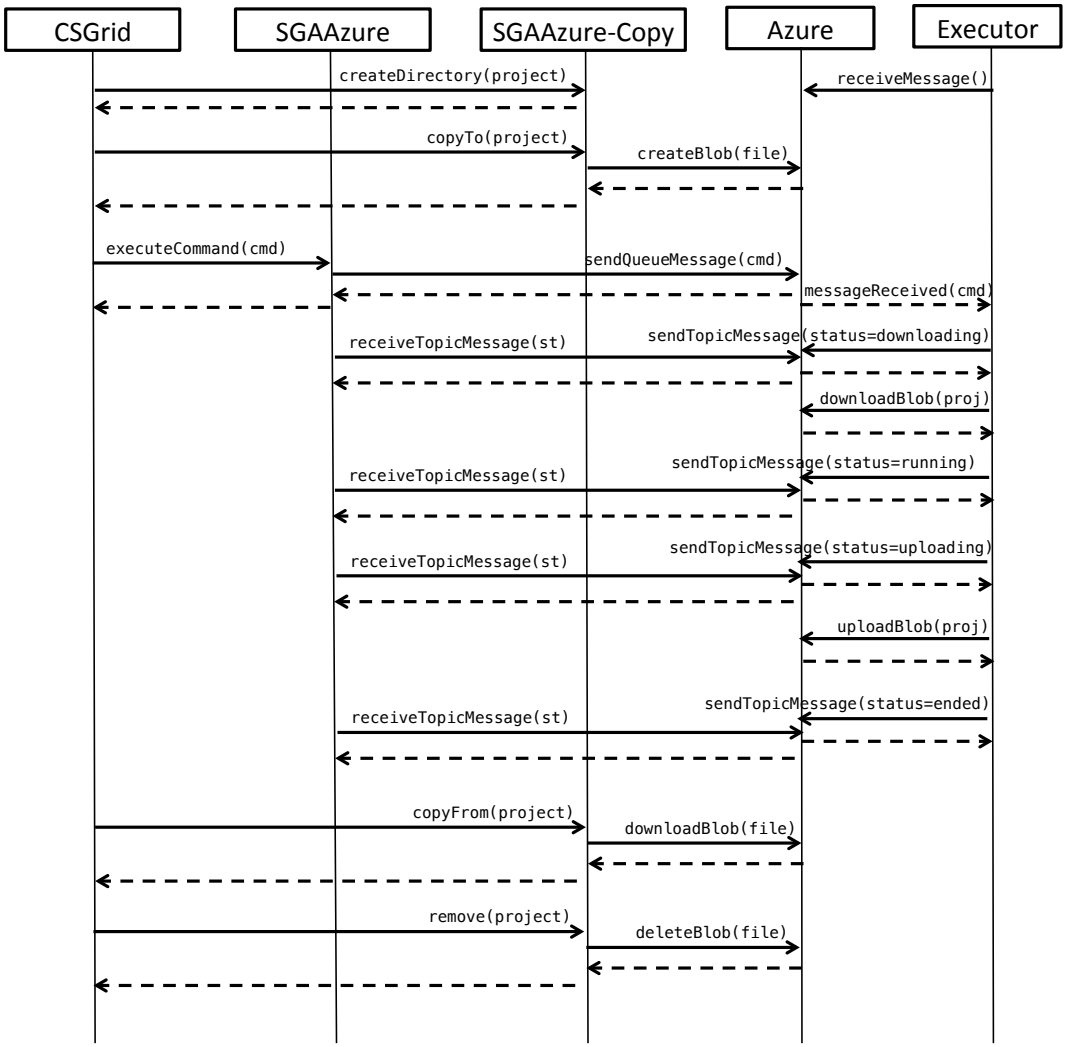

Figura 6.1: Diagrama de sequência sobre uma execução de comando no plugin SGA Azure

executora ser provisionada ou designada. Como o plugin de transferência não precisa aguardar pela disponibilidade da máquina virtual designada para só então realizar a cópia, viabiliza-se o fluxo de execução idêntico ao do SGA original (descrito na Seção 2.2.2), onde a cópia dos dados é feita antes da submissão do comando propriamente dita. Uma vez que a API do plugin de transferência discutida na Seção 2.2.1 não permite definir o nó de execução, o uso do serviço de blobs como o destino único nas transferências dos arquivos se torna bem adequado ao modelo da API do CSGrid.

A política para criação de novas máquinas virtuais se baseia em uma linha de execução independente do SGA Azure, implementada como uma *thread*. Nesta linha, o SGA interroga o serviço de nuvem sobre quantas máquinas virtuais existem na nuvem e sobre quantas mensagens estão na fila de comandos do *Service Bus* esperando serem consumidos. O SGA demanda então

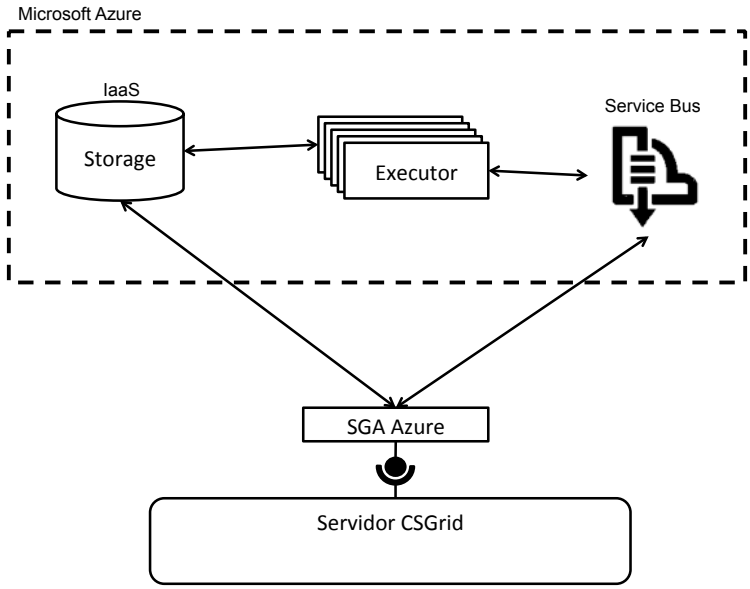

Figura 6.2: Estruturação da solução SGA dedicada ao Windows Azure usando *Service Bus*

a criação do número de máquinas virtuais equivalente ao número de mensagens pendentes na fila de execução. A demanda para criação das máquinas virtuais acontece em paralelo, tornando o aparecimento de novos nós de execução bastante eficiente. Vimos na Seção 4.1 que uma máquina virtual leva cerca de 40 segundos para ser provisionada. Assim, uma iteração de criação de várias máquinas virtuais também leva em torno deste mesmo tempo. O algoritmo abaixo ilustra a política de criação de novas máquinas virtuais:

```
1 A cada um minuto,
2   NumDeMensagens = [quantidade de mensagens na fila do
       Service Bus];
3   NumVMs = [quantidade de VMs ativas]
4   DemandaDeVMs = NumDeMensagens - NumVMs
5   Se DemandaDeVMs > 0,
6     Provisionar DemandaDeVMs em paralelo
```

Dada a monitoração constante feita pelo SGA Azure do número de máquinas virtuais instanciadas e também da fila de execução no *Service Bus*, quando um novo comando for submetido pelo CSGrid, na ausência de máquinas virtuais, o SGA Azure provisionará uma nova. Caso vários comandos sejam submetidos simultaneamente pelo CSGrid, várias máquinas virtuais serão criadas. Cada máquina virtual, em sua inicialização, executa automaticamente o *daemon executor* do SGA Azure, que consome o trabalho da fila do *Service Bus* para executar o comando associado. Após o término da execução de um comando, o *executor* aguarda por mais algum comando na fila por no máximo um minuto. Após tal tempo de ociosidade, o *executor* então instrui a Azure a desprovisionar sua máquina virtual, junto com todos os seus recursos

relacionados.

A estratégia de que o próprio executor se destrua é útil tanto para proteger o sistema contra falhas que levam o ambiente a reter máquinas virtuais ativas (*causando tarifação indevida*) quanto para promover o paralelismo nas operações de manutenção das máquinas virtuais. Assim o trabalho se mantém dividido: O plugin SGA Azure se encarrega de iniciar máquinas virtuais conforme a demanda aumenta, e as próprias máquinas virtuais se encarregam de se destruir quando a demanda cessa.

O *executor* que roda nos nós de execução também se encarrega de recuperar os arquivos binários do programa a ser executado, assim como os dados necessários da área de projeto. O executor monta a *sandbox* para o programa, o executa monitorando o status da máquina e, logo depois do término, submete os arquivos gerados de volta para o armazenamento do projeto na nuvem. O *executor* também se encarrega de enviar o seu status para o servidor CSGrid periodicamente através de um tópico no *Service Bus* destinado para tal.

Tal *executor* é um programa Python 2.7.6 pré-instalado nas máquinas virtuais que cumpre os requisitos citados na seção anterior. O programa submetido é executado na máquina virtual deste executor como um usuário limitado, sem direitos especiais, diferente do usuário relacionado ao executor propriamente dito.

A Figura 6.3 ilustra como estão organizados os componentes do *executor* que roda em cada máquina virtual dentro da nuvem. As responsabilidades dos componentes citados estão listadas abaixo:

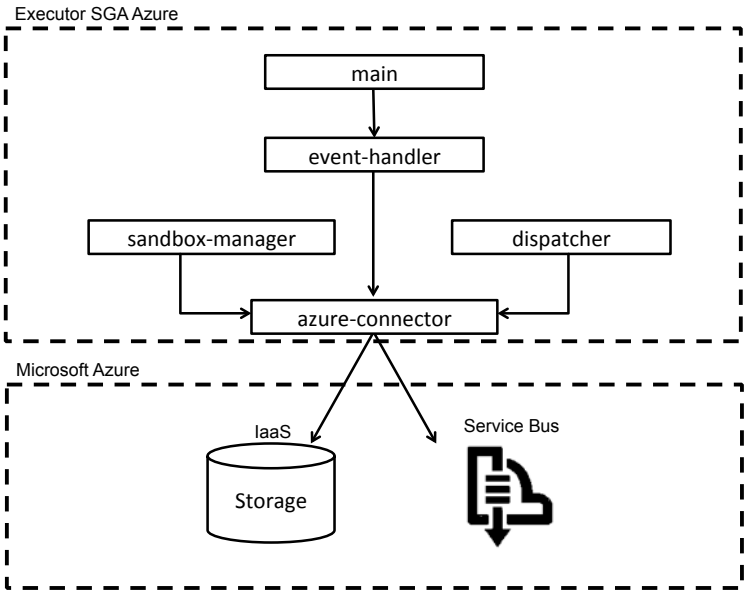

Figura 6.3: Arquitetura da solução para execução nas máquinas virtuais

1. main: Módulo responsável pela inicialização ou encerramento do serviço, conforme linha de comando utilizada. Ao iniciar o sistema, o loop principal é iniciado em um novo processo no sistema operacional e seu identificador (PID) é armazenado em disco para referência futura;

2. event-handler: Módulo responsável pelo loop principal do sistema. O executor do SGA Azure executa em apenas uma linha de execução, com um loop de eventos e uma máquina de estados pré-definida. O código-fonte para o loop principal de eventos do executor do SGA Azure pode ser visto no Apêndice C;

3. sandbox-manager: Módulo responsável pela recuperação dos arquivos de entrada para os programas, dos binários dos programas propriamente ditos, e também da identificação e envio dos arquivos criados ou modificados durante a execução de volta para o armazenamento principal do projeto;

4. dispatcher: Módulo responsável pelo disparo do programa no ambiente controlado com credenciais diferenciadas perante o sistema operacional;

5. azure-connector: Módulo responsável por toda a comunicação com a API da Azure. Todos os módulos previamente citados, quando precisam acessar recursos da nuvem, o fazem através de chamadas a este módulo.

Em particular, o módulo azure-connector é carregado dinamicamente, baseado em um parâmetro de configuração. Assim, uma vez respeitando o contrato a ser implementado, qualquer outra plataforma de computação em nuvem que ofereça serviços semelhantes à Azure pode ter seu conector implementado, com um ponto único de acoplamento.

7
Avaliação de Resultados

Há vários aspectos a serem analisados na integração entre o CSGrid e a nuvem pública Azure relacionados ao desempenho, flexibilidade e segurança. Neste trabalho nos concentramos em determinar o ganho de desempenho decorrente do uso da nuvem quando comparado a um ambiente estaticamente configurado executando de forma serializada. O uso de alocação dinâmica de máquinas e da comunicação de dados e controle via serviços Azure trazem um impacto em tempo de execução e throughput. Procuramos medir esse impacto.

Avaliamos o desempenho do SGA perante a elasticidade através do envio sucessivo e com frequência variada de comandos para o escalonador do CSGrid utilizando o SGA Azure. Para cada experimento, monitoramos o número de máquinas virtuais ativas, o número de comandos em execução e o número de comandos na fila do *Service Bus* ao longo do tempo.

Submetemos programas que simulam a execução de uma tarefa de 100 segundos, que recebe um arquivo de entrada e o retorna comprimido. A simulação consiste em aguardar 100 segundos através de uma primitiva *sleep* e então processar o arquivo de entrada. Vimos na Seção 4.5 que o desempenho da execução de um programa quando usada uma máquina virtual **D** é compatível com o da execução em uma estação de trabalho não-virtual comumente usada. Assim, ao executar simulações controladas, pudemos nos concentrar apenas nos detalhes referentes à elasticidade do sistema.

Neste capítulo apresentamos resultados e discussões sobre variações no padrão de submissão de instâncias do programa descrito acima. Na Seção 7.1, apresentamos o comportamento do CSGrid usando SGA Azure quando demandada a execução de apenas um comando com o perfil descrito. Na Seção 7.2, exploramos o comportamento quando submetemos ao escalonador do CSGrid uma rajada de vinte comandos de uma só vez. Na Seção 7.3, o comportamento é alterado para, além da rajada inicial de vinte comandos, outros comandos são submetidos de dez em dez segundos por vinte minutos. Na Seção 7.4, semelhante ao experimento anterior, após os vinte minutos de submissões, outra rajada de vinte comandos simultâneos é enviada. Finalmente, na Seção 7.5, descrevemos uma experiência onde a política de provisionamento foi modificada.

Continuando a apresentação de resultados, discutimos na Seção 7.6 os aspectos relacionados à API de plugins do CSGrid e ao comportamento do próprio que podem ser melhorados perante a nova demanda de adaptação ao

paradigma de computação em nuvem.

7.1 Execução de Um Só Comando

A execução única do teste onde o servidor CSGrid é iniciado e, imediatamente, é enviada uma execução do programa de 100 segundos descrito anteriormente e 9MB de entrada obteve os seguintes resultados de tempo, listados a seguir:

- Tempo total de execução, desde a submissão no CSGrid até o término completo: 254 segundos;
- Tempo de execução no executor (download, execução e upload): 102 segundos;
- Tempo de espera desde a submissão no CSGrid até o início da execução no executor: 126 segundos.

Nota-se nesta execução que o tempo total percebido pelo usuário é mais do que o dobro do esperado devido à sobrecarga total de provisionamento e inicialização da máquina virtual. Conclui-se aqui que o modelo de nuvem com alocação dinâmica de máquinas não é adequado para execuções com este perfil onde apenas uma execução de tempo relativamente curto é demandada isoladamente.

7.2 Rajada de Comandos

Nesta seção vamos mostrar como o SGA Azure se comporta quando submetemos 20 comandos logo na inicialização do servidor CSGrid, com o mesmo perfil da seção anterior: cada comando durando 100 segundos e recebendo um arquivo de entrada de 9MB. Com isto, pretendemos capturar o comportamento do sistema perante uma carga única de um conjunto de comandos e também como a arquitetura do CSGrid e a política de provisionamento do SGA Azure responde a tal demanda. A Figura 7.1 mostra o número de máquinas virtuais ao longo do tempo de execução:

Da Figura 7.1 pode ser observado que foram instanciadas nove máquinas virtuais. O tempo total para o término da execução dos 20 comandos foi cerca de 490 segundos. Considerando que cada execução levou 100 segundos, sem contar cópias de dados, temos $20 \times 100 = 2000$ segundos. Conseguimos então executar os 20 comandos em cerca de um quarto do tempo que os mesmos 20 demorariam para ser executados de forma serial, e considerando os tempos de provisionamento e cópia de dados.

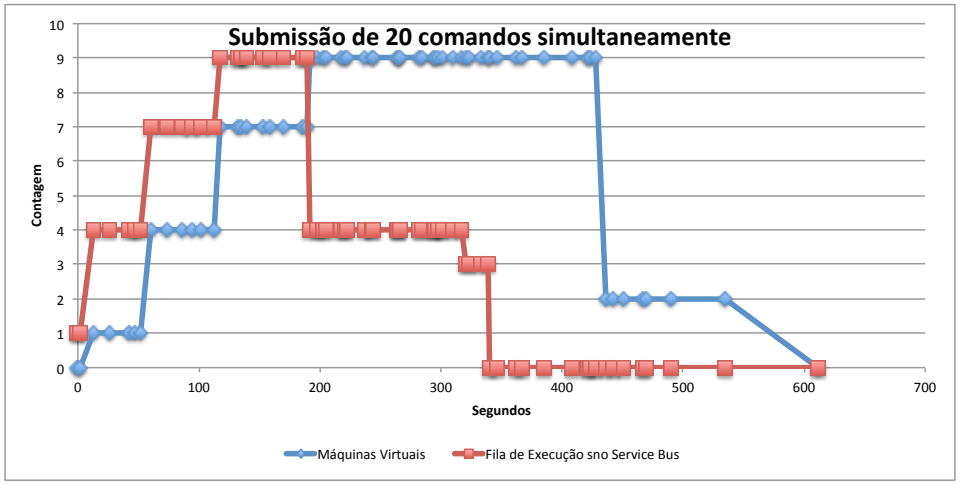

Figura 7.1: Resultado da submissão de 20 comandos na inicialização do CSGrid

Para uma estimativa do custo monetário envolvido apenas relacionado ao tempo de máquinas virtuais, fizemos uma estimativa pessimista que consiste em calcular a área do gráfico da Figura 7.1 com relação ao número de máquinas virtuais. Com isto, temos uma aproximação para o número de máquinas virtuais x segundos. Para o gráfico em questão, o valor da área é 3.268. Ou seja, se para o cenário onde os comandos são executados de forma serializada que seriam necessários 2.000 segundos, a sobregarga devido às operações de provisionamento e inicialização das máquinas e processos fez com que o tempo total de máquinas virtuais ativas fosse cerca de 63% maior do que o tempo necessário para executar os comandos de forma serializada. Assim, para se ganhar 75% do tempo de relógio para executar os comandos em lote, foi pago o preço de 63% de tempo de máquinas virtuais em sobrecargas relacionadas à elasticidade.

Observando o gráfico, nota-se também que a fila de execução do *Service Bus* não foi inicializada com as vinte execuções de uma só vez. Ao invés disso, foi crescendo gradualmente. Isto se deve ao fato de que o serviço de submissões do CSGrid opera sequencialmente e, além da sobrecarga do envio das mensagens para a fila do *Service Bus*, o CSGrid só considera que o comando foi submetido após ter copiado os arquivos de entrada através do plugin de transferência, não realizando assim submissões em paralelo. Cada submissão com 9MB trouxe então um impacto para o caso da submissão de vários comandos simultaneamente, pois apesar de se ter instruído ao CSGrid a submissão de 20 comandos de uma só vez, cada um foi informado ao SGA Azure na medida em que seus 9MB foram transferidos, de forma sequencial.

7.3 Rajada Seguida por uma Sequência de Comandos

Neste experimento, os arquivos de entrada foram reduzidos para 1MB. Foram submetidos 20 comandos logo na inicialização do servidor CSGrid. Logo após, mantivemos um fluxo no envio de comandos a cada cinco segundos para o escalonador do CSGrid durante 20 minutos, totalizando 260 submissões. Com este comportamento, obtivemos os resultados expressos no gráfico da Figura 7.2.

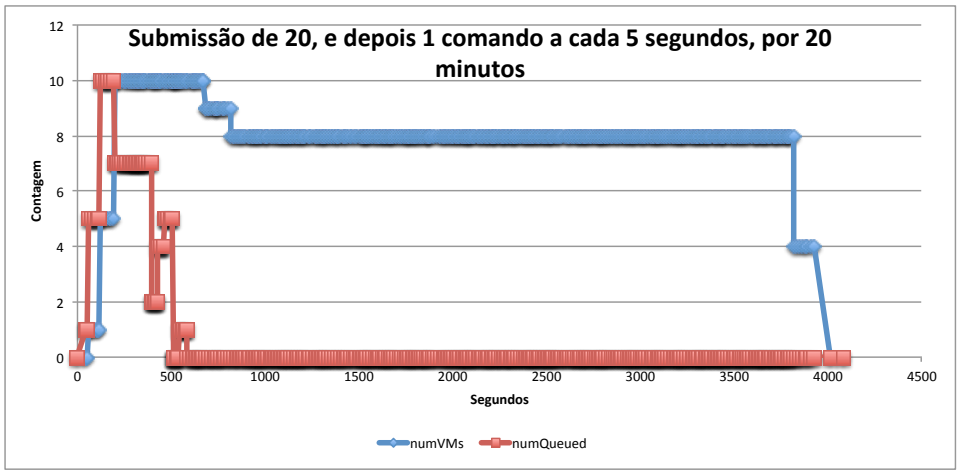

Figura 7.2: Resultado da submissão de 20 comandos e logo após 20 minutos de envios de 5 em 5 segundos

Dado que 260 execuções, se executadas de forma serial, levariam 26.000 segundos (*7h13m*), e o término da última execução no SGA Azure foi registrado após 3.889 segundos, com um pico de 10 nós de execução, temos que o tempo total de parede de execução do algoritmo foi cerca de 15% do tempo que as 260 execuções levariam para rodar sequencialmente.

7.4 Rajada, Sequência, e Outra Rajada de Comandos

Desta vez, repetimos o experimento da etapa anterior, porém agora executando após o término da sequência de envios de 20 minutos de 5 em 5 segundos, mais outra rajada de 20 submissões foi feita. Portanto, submentendo um total de 280 comandos. O resultado está no gráfico da Figura 7.3.

Dados que 280 execuções, se executadas de forma serial, levariam 280.000 segundos (*7h46m*), e o término da última execução no SGA Azure foi registrado após 4.160 segundos com um pico de 9 nós de execução, temos que o tempo total de parede de execução do algoritmo foi também cerca de 15% do tempo que as 280 execuções levariam para rodar sequencialmente.

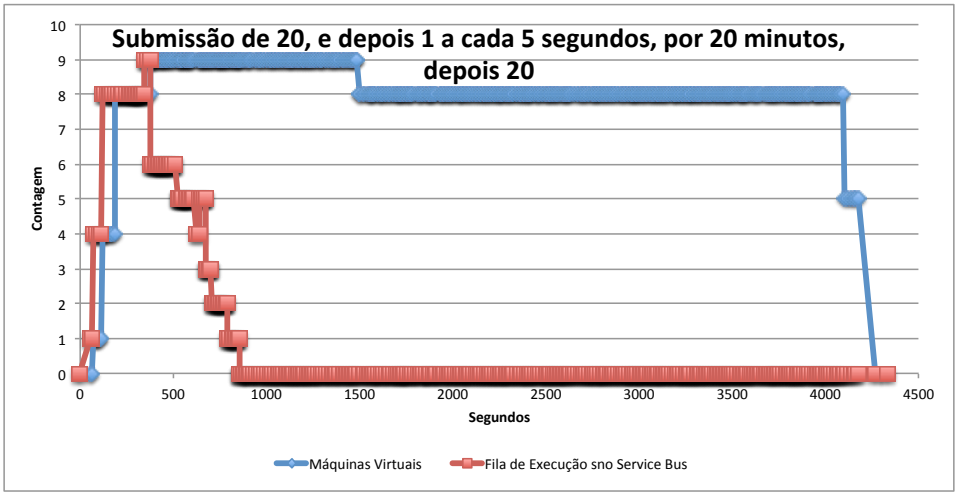

Figura 7.3: Resultado da submissão de 20 comandos, depois de 5 em 5 por 20 minutos, depois mais 20, na inicialização do CSGrid

Notamos neste experimento que a última rajada de 20 submissões mal apareceu como um pico no gráfico, o que se justifica pela pré-existência de máquinas virtuais suficientes para consumir a fila na medida em que o servidor CSGrid disponibilizava as submissões para o SGA. Mesmo com o arquivo de entrada do algoritmo reduzido para 1MB, a sobrecarga envolvida na preparação das *sandboxes* foi suficiente para que as oito máquinas virtuais ativas não permitissem o acúmulo de comandos na fila do *Service Bus*.

7.5 Rajadas, Sequências e Fator de Provisionamento

Para este experimento, introduzimos o conceito de fator de provisionamento, que até então tem sido trabalhado com o valor um. Tal fator pode ser considerado um parâmetro para a política de escalonamento do SGA Azure que influencia na criação das máquinas virtuais através da sequinte modificação no algoritmo descrito na Seção 6.2:

```
1 A cada um minuto,
2   NumDeMensagens = [quantidade de mensagens na fila do
      Service Bus];
3   NumVMs = [quantidade de VMs ativas]
4   DemandaDeVMs = NumDeMensagens - NumVMs
5   Se DemandaDeVMs > 0,
6     Provisionar DemandaDeVMs * FatorDeProvisionamento
        em paralelo
```

Nesta versão, apenas a linha onde *FatorDeProvisionamento* aparece foi modificada. Acreditamos que, com isto, em padrões de submissão intensivos, o SGA Azure antecipe o envio de comandos e os consuma mais rapidamente, uma vez que mais máquinas virtuais serão provisionadas com relativa antecedência

e, na ocasião da chegada de comandos, haverá mais chance de todo o processo de boot e inicialização de mais máquinas já terem sido concluídos.

Para os experimentos a seguir, o mesmo algoritmo de 100 segundos foi usado, porém com um arquivo de entrada de tamanho simbólico para minimizar o efeito da serialização da transferência e submissão pelo servidor CSGrid. Assim, cada comando possui sua duração fixa, variando apenas a forma como foram submetidos e o fator de provisionamento programado no SGA Azure.

Inicialmente, executamos o experimento para uma rajada única de vinte submissões com o fator de provisionamento 1 e 2. Os respectivos gráficos são apresentados na Figuras 7.4 e na Figura 7.5.

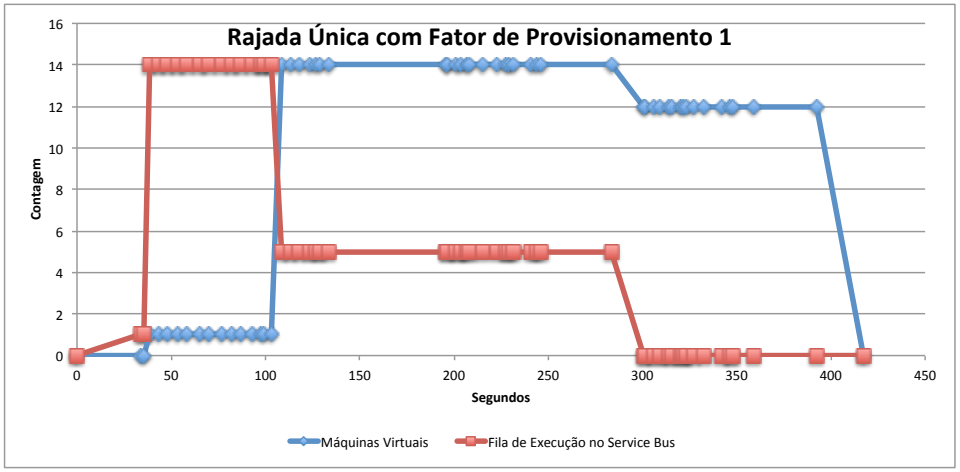

Figura 7.4: Resultado da submissão de 20 comandos com fator de provisionamento 1

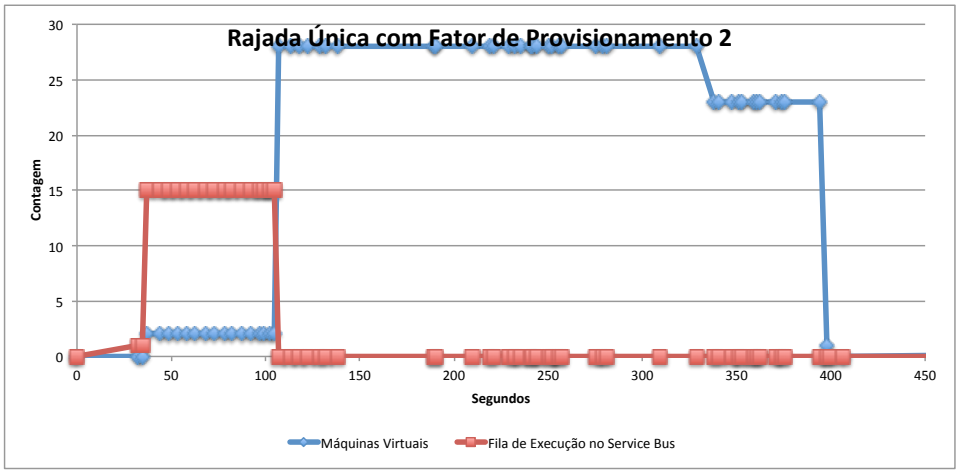

Figura 7.5: Resultado da submissão de 20 comandos com fator de provisionamento 2

Observamos que a fila é consumida integralmente em menos tempo quando usado o fator 2, o que era esperado. Porém a medição do número de máquinas virtuais evidenciou a queda no número aproximadamente no mesmo

momento. A falta de precisão temporal na medida do número de máquinas virtuais se deve ao número total delas e ao processo de polling feito pelo SGA Azure. Quanto maior for o número de máquinas ativas, mais demorado é o ciclo de polling que monitora as máquinas virtuais devido ao número de chamadas feitas à API.

Para ilustrar a melhoria no ritmo em que os comandos são consumidos com o aumento do fator de provisionamento, realizamos também o experimento que envia apenas comandos de dez em dez segundos, por vinte minutos, sem as rajadas inicial ou final usadas nos experimentos anteriores. Os resultados estão nos gráficos da Figura 7.6 e da Figura 7.7.

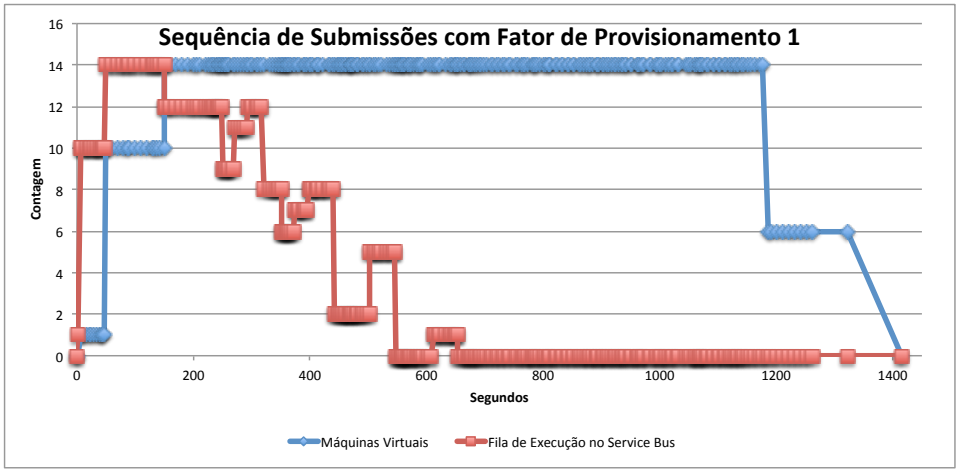

Figura 7.6: Resultado da submissão de comandos por 20 minutos com fator de provisionamento 1

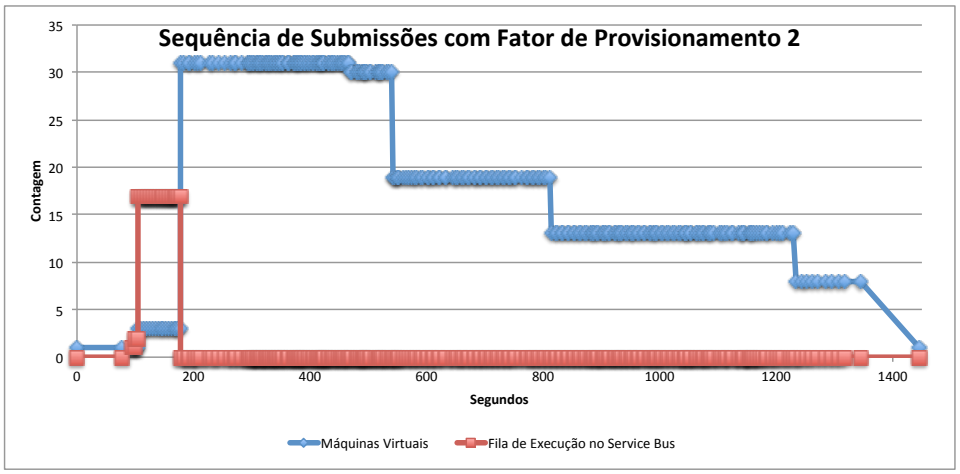

Figura 7.7: Resultado da submissão de comandos por 20 minutos com fator de provisionamento 2

Observando a Figura 7.6, observamos o pico de submissões feitas no início para só então ocorrer o decaimento esperado. Isto ocorre porque enquanto o

CSGrid está submetendo comandos de dez em dez segundos, as máquinas virtuais ainda estão sendo provisionadas e inicializadas. Durante este tempo, os comandos vão se acumulando naturalmente na fila do *Service Bus*. Após a primeira iteração de provisionamento, começa a ser notado o início do decaimento da série que representa o número de comandos na fila. Com o fator de provisionamento 1, o ritmo em que os comandos são consumidos e executados faz o efeito "escada" que aparece no gráfico. Já quando se usa o fator de provisionamento 2, o grande número de máquinas virtuais provisionadas inicialmente consegue consumir integralmente a fila de maneira muito rápida. O natural excesso de máquinas virtuais faz com que, ao longo do tempo, os nós de execução ociosos sejam desprovisionados sem permitir que a fila do *Service Bus* acumule comandos.

O experimento com fator de provisionamento traz uma boa ilustração do comportamento causado pela política de criação e destruição de máquinas virtuais adotada pelo SGA Azure.

7.6 Sobre a API de Plugins do CSGrid para Computação em Nuvem

O CSGrid, como apresentado no Capítulo 2, não foi originalmente concebido para interagir com os SGAs através de uma API de plugins. Historicamente, o modelo de submissão de comandos do CSGrid se especializou nos cenários de HPC em cluster de tamanho fixo e com a implementação original de seu SGA. Nesta seção são propostas metas de melhoria para a API de plugins do CSGrid partindo da descrição da API do CSGrid apresentada na Seção 2.2, da API da Azure apresentada na Seção 3.1.3, e de reflexões sobre os resultados dos experimentos com o SGAAzure apresentados nas demais seções deste capítulo.

7.6.1 Fluxo de Execução

Quando o servidor CSGrid solicita a execução de um comando a um SGA que requer um mecanismo de transferência, a *sandbox* para execução é criada sob orquestração exclusiva do servidor CSGrid. Tal orquestração cria uma estrutura de diretórios cujos nomes têm como parte o identificador único do comando e, para cada comando, uma cópia independente dos arquivos de entrada é feita. Trata-se de um comportamento coerente no cenário onde cada SGA é um nó de execução independente e nenhum deles está conectado ao servidor CSGrid através de NFS.

No entanto, no cenário do SGA Azure onde foram feitas várias submissões que levam o mesmo arquivo de entrada, a imposição da orquestração do CSGrid com réplicas de tais arquivos para cada comando inviabilizou otimizações no processo de preparação das *sandboxes*. No caso, o SGA Azure poderia submeter o arquivo de entrada apenas uma vez para o serviço de armazenamento, evitando transferências redundantes de arquivos de entrada reusados entre diferentes comandos. Uma vez no serviço de armazenamento, cada nó de execução traria o arquivo para si, limitando transferências múltiplas dos mesmos arquivos à rede interna da nuvem, o que já se mostrou bem mais eficiente.

O cenário NFS ilustra como não são necessárias cópias independentes dos arquivos de entrada para cada comando que os utilize, pois os programas são executados nos nós acessando diretamente os arquivos do repositório de projeto no servidor CSGrid. Ou seja, não são criadas réplicas dos arquivos de entrada para uma *sandbox* exclusiva do comando. Entende-se, portanto, que a criação de *sandboxes* só faz sentido quando não se tem acesso NFS, e é necessário fazer com que o programa acredite que se tem. No caso do SGA Azure, vimos que tal objetivo pode ser atingido de forma mais eficiente através do uso do serviço de armazenamento de blobs.

Uma estratégia que pode ser mais adequada é delegar para o próprio SGA todo o procedimento para a criação de *sandbox* de execução dos comandos. Assim, se for mais aplicável a cópia ativa do servidor CSGrid para cada nó de execução, o plugin SGA o fará, criando a *sandbox* da maneira mais eficiente e conveniente de acordo com seu domínio. No caso de nuvem, por exemplo, o SGA poderá administrar a sincronização da área de projeto com os nós de execução utilizando um repositório intermediário como um serviço de blobs ou uma máquina virtual dedicada.

Da mesma forma, o modelo atual onde o CSGrid pressupõe os caminhos absolutos onde os arquivos estarão no nós de execução não é aplicável para todos os cenários. O *executor* do SGA Azure ilustra isto, pois ao receber os arquivos para a sua *sandbox*, precisa realizar a substituição textual no arquivo de script fornecido pelo servidor CSGrid para que os caminhos absolutos representados internamente se apliquem aos da máquina virtual. Tal necessidade de subterfúgio na execução de comandos para garantir a compatibilidade sugere que o modelo atual de submissão deve ser ajustado para dar maior autonomia neste quesito ao SGA.

7.6.2 Uso do Escalonador do CSGrid

O CSGrid possui um escalonador que vem sofrendo melhorias constantemente. O serviço de escalonamento do CSGrid conta com uma fila interna de comandos para submissão onde a ordem de envio para os SGAs registrados pode variar conforme prioridade declarada ou compatibilidade com os SGAs disponíveis. Para tal, o serviço de escalonamento faz a análise de sua fila de execução e trabalha com a premissa de que cada instância de SGA pode executar um número finito de comandos simultaneamente.

SGAs que representam escalonadores terceiros, uma vez que encaminham as execuções para a fila interna de seu próprio escalonador, são tidos pelo CSGrid como sempre disponíveis para receberem novos comandos. Assim, a fila do escalonador do CSGrid permanece sempre vazia, pois todos os comandos submetidos são imediatamente encaminhados para tais SGAs.

O SGA Azure atua dessa forma. Todos os comandos enviados para o CSGrid são imediatamente passados para o SGA Azure, que os encaminha para sua fila privada no *Service Bus*. Esse modelo dificulta que o escalonador do próprio CSGrid possa ser usado para adotar políticas mais eficientes de uso da fila e controle de envio de comandos para o SGA.

Para usufruir dos benefícios do escalonador do CSGrid, é necessário que SGAs que representem clusters não admitam comandos incondicionalmente para que, uma vez retidos na fila própria do CSGrid, seu escalonador possa tomar suas decisões de maneira mais consistente. Formas de se atuar sobre o modelo de submissão incluem inverter o controle da requisição de comandos (o SGA solicitar comandos para o CSGrid, ao invés do CSGrid enviar comandos para o SGA) ou o SGA negar explicitamente um comando enviado pelo escalonador, dando oportunidade deste reencaminhar o comando ou tomar qualquer outra decisão sob o domínio do CSGrid.

7.6.3 Parametrização de Submissões

O SGA Azure é responsável pelo provisionamento de máquinas virtuais de forma automática de acordo com a demanda medida pela ocupação da fila de submissões do *Service Bus*. Atualmente, a máquina provisionada sempre utiliza a mesma imagem pré-configurada de sistema operacional com o *daemon executor*. Por convenção, o SGA Azure sempre provisiona máquinas de tamanho "D1" (vide Tabela 3.1). O que seria mais adequado é que o tamanho esperado para a máquina a ser usada no comando seja declarada ou suge-

rida como parâmetro da submissão para o SGA. Este parâmetro da submissão poderia ser feito de duas formas:

1. O SGA informa um catálogo de perfis de hardware disponíveis onde cada nome corresponde a uma configuração. De posse deste catálogo, o servidor CSGrid permite com que o usuário selecione em uma lista fechada a configuração de hardware desejada para executar o comando;

2. O CSGrid informa para o SGA a configuração de hardware mínima para executar o comando em termos de número de núcleos, disco e memória RAM. O SGA deve então descobrir em seu catálogo de tamanhos de máquinas virtuais qual é a menor máquina virtual que seja maior do que a esperada.

A primeira alternativa é um consenso para nuvens públicas, enquanto a segunda opção é mais generalista. Ter formalmente o parâmetro sobre o tamanho esperado da máquina para um comando traz a possibilidade da implementação do SGA gerir com maior acurácia os provisionamentos de máquinas virtuais.

De forma mais abrangente, é interessante que todos os parâmetros esperados pela Microsoft Azure voltados para o provisionamento de novas máquinas possam ser informados pelo usuário final do CSGrid no ato da submissão. Assim, outro parâmetro que se faz de vital importância é o identificador da imagem de origem para o sistema operacional.

Da mesma forma, cada programa diferente pode requerer pacotes de bibliotecas pré-instaladas na imagem do sistema operacional. Tendo um catálogo de imagens pré-montadas para o SGA Azure, um novo parâmetro para a execução do comando serviria para identificar a imagem a ser usada para a máquina que vai executar o comando. De forma análoga ao cenário anterior, este parâmetro pode ser informado de duas formas:

1. O SGA informa um catálogo com nomes de imagens para popular algum dicionário no servidor CSGrid. Com isto, o servidor CSGrid interroga o usuário final de maneira mais objetiva, apresentando uma lista de opções;

2. O SGA tem para si um catálogo das imagens de sistemas operacionais de executores e, associado a cada um deles, a lista de pacotes pré-instalados. Com isto, espera-se que a parametrização do programa leve apenas a coleção dos pacotes pré-instalados necessários. O SGA então deverá encontrar imagens em seu repositório interno cujos pacotes sejam compatíveis.

Os parâmetros relevantes envolvidos no provisionamento de máquinas virtuais foram apresentados na Seção 3.1.3.

7.7 Sobre a API da Azure

Apesar de não estar no escopo deste trabalho a avaliação da API da Azure, a experiência que tivemos no seu uso para o desenvolvimento do SGA Azure pode ajudar outros desenvolvedores que precisem utilizar essa mesma API para desenvolver novos serviços ou aplicações na nuvem. Nesta seção descrevemos algumas considerações com relação aos aspectos da API experimentados durante o trabalho.

7.7.1 Serviço de Armazenamento

No serviço de armazenamento da Azure, para a implementação correta do método *getRemoteTimestamps* no plugin de transferência, foi necessário usar o recurso de associação de pares chave-valor a cada blob enviado. Tal recurso é citado na Seção 3.1.3. Ao se copiar um arquivo do servidor CSGrid para o serviço de blobs, o timestamp referente ao arquivo não era enviado corretamente, fazendo com que uma implementação ingênua de *getRemoteTimestamps* sempre levasse o CSGrid a crer que o arquivo foi modificado remotamente, causando transferências desnecessárias.

Para resolver esta questão, sempre que o plugin de transferência do SGA azure é chamado para copiar arquivos do servidor CSGrid para um blob, é criado um metadado chave-valor no mesmo contendo o timestamp do arquivo no servidor CSGrid. Analogamente, a implementação de *getRemoteTimestamps* checa primeiramente o metadado do blob para, caso não exista, considerar o seu timestamp oficial.

7.7.2 Provisionamento de Máquinas Virtuais

Apesar do provisionamento de máquinas virtuais aparentar ser simples, o uso da API da Azure se mostrou bastante prolixo. Como já discutido, alguns dos parâmetros podem fazer parte do contrato de plugins como o tamanho da máquina virtual ou o endereço da imagem a ser usada com o sistema operacional devidamente instrumentado com o *executor*. Porém, há outros como credenciais para autenticação na nova máquina (no caso, login e senha para acesso SSH), parâmetros de rede interna e discos virtuais extras que são fixos por convenção no SGA Azure.

Em particular, o modelo da Azure define *Cloud Services*. Cada *Cloud Service* pode conter uma ou mais máquinas virtuais compartilhando mesmo IP na Internet. O aspecto relevante para este trabalho, no entanto, é que não é possível provisionar mais de uma máquina virtual simultaneamente dentro de um mesmo *Cloud Service*. Como tal requisito foi considerado crítico no projeto, o SGA Azure então cria um *Cloud Service* para cada máquina virtual.

Este uso da API da Azure torna o processo de provisionamento e monitoração mais verboso, uma vez que não é possível interrogar diretamente a Azure sobre todas as máquinas virtuais existentes. A API define apenas a consulta sobre todos os *Cloud Services* existentes e a consulta sobre as máquinas virtuais dentro de um *Cloud Service*. Não há busca por todas as máquinas virtuais diretamente. Assim, para provisionar uma máquina virtual, deve-se então criar primeiro seu *Cloud Service* para depois sim instruir a criação da máquina propriamente dita. Para monitorar a máquina virtual, é necessário questionar a API através de seu *Cloud Service*, um por um. Em um cenário de uso de dezenas de máquinas virtuais, deverão ser feitas tantas requisições para a API quantas forem os *Cloud Services*, o que pode se tornar caro.

A exigência de um *Cloud Service* por máquina virtual para viabilizar o provisionamento em paralelo também traz consequências para o processo de desprovisionamento. Quando uma máquina virtual instrui o seu desprovisionamento, ela é incapaz de solicitar a exclusão de seu *Cloud Service*, pois ela não mais existirá para fazer a segunda solicitação. A responsabilidade por fazer a exclusão dos *Cloud Services* já sem nenhuma máquina virtual recai sobre o SGA Azure, que a faz automaticamente como parte do processo de monitoração das máquinas virtuais. Convenientemente, as operações envolvendo criação e exclusão de *Cloud Services* são rápidas e não causam sobrecarga significativa ao sistema.

Infelizmente a Azure impõe um limite de 200 *Cloud Services* por assinatura do serviço de nuvem. O mesmo já não ocorre com o número de máquinas virtuais. Tal limitação, em conjunto com a necessidade de se provisionar em paralelo, levam a demandas para trabalhos futuros que envolvem o reuso inteligente de *Cloud Services* para provisionamento de mais máquinas virtuais dentro de um mesmo *Cloud Service* sem comprometer o desempenho do sistema devido à serialização no provisionamento. Com um número já grande de *Cloud Services* instanciados, é possível recorrer a um dos já existentes ao invés de continuar instanciando novos.

7.7.3 Service Bus

O *Service Bus* é atraente especialmente por sua garantia FIFO e requisições bloqueantes para o recebimento de mensagens. Porém, por se tratar de um serviço avulso dentro do catálogo da Azure, sofre tarifação diferenciada. Além disso, sua API HTTP/REST é limitada com relação às demais. Por exemplo, não é possível instanciar serviço de filas programaticamente pela API REST.

Outra limitação diz respeito à sobrecarga das chamadas bloqueantes, que são sujeitas à latência da Internet, possuem criptografia SSL e recuperam apenas uma mensagem por vez. Na ocorrência de um número grande de mensagens para se recuperar, a API do *Service Bus* pode se tornar o gargalo do sistema.

O *Service Bus* provê uma implementação para o recebimento de mensagens em lote de forma mais eficiente. Porém até o momento da implementação do SGA Azure tal implementação estava disponível apenas para a plataforma .NET em protocolo fechado, inviabilizando o uso no SGA Azure.

Mesmo assim, o *Service Bus* se mostrou satisfatório, podendo ocasionar atrasos significativos apenas com um grande número de nós de execução interagindo simultaneamente com o servidor CSGrid.

7.8 Implementações da API

Apesar de boa parte da API da Azure ser baseada em REST/HTTP, há implementações disponíveis abertamente em várias linguagens de programação para que o uso dos recursos da nuvem não precise passar por tal implementação de mais baixo nível. O SGA Azure utilizou duas implementações da API: A versão para Java, usada no plugin SGA Azure e a versão para Python, usada na implementação do *executor*.

A implementação em Python foi usada com grande facilidade devido à sua simplicidade.

A implementação em Java é também de grande facilidade de uso. Porém, a arquitetura de plugins usada na implementação do SGA Azure encontrou alguns problemas no uso da implementação Java pelo CSGrid.

No CSGrid, os plugins são carregados dinamicamente. Na tecnologia Java, carregamentos dinâmicos envolvem a instanciação de espaços lógicos distintos para o armazenamento de tais objetos carregados. Esses espaços são denominados *classloaders*(Liang e Bracha, 1998). Qualquer sistema que envolva carregamento dinâmico em Java utiliza este artifício. Infelizmente,

o artifício tecnológico usado na implementação da API Azure para Java é incompatível com o carregamento dinâmico. Não podendo ser carregadas em um *classloader* individual, as bibliotecas referentes à implementação padrão da API Azure tiveram de ser incorporadas ao próprio CSGrid. Uma vez carregadas no *classloader* do CSGrid, o plugin SGA Azure não apresentou problemas para executar.

Dada tal limitação, pode ser necessária a reimplementação da API REST/HTTP da Azure de forma compatível com carregamento dinâmico em Java.

8
Conclusão e Trabalhos Futuros

Tomamos como objetivo inicial deste trabalho a especificação e prototipação de um novo módulo especializado na nuvem pública Microsoft Azure, com a finalidade de tornar o CSGrid capaz de adquirir novos recursos automaticamente através da instanciação de máquinas virtuais na medida em que forem demandados comandos para executar. Com o desenvolvimento deste protótipo, tivemos a oportunidade de exercitar a então nova infra-estrutura de plugins provida pelo CSGrid e, através da experiência, indicar pontos de atenção e melhoria. Para embasar o uso da nuvem e avaliar seu desempenho, realizamos experimentos que quantificaram e possibilitaram a comparação com máquinas reais no que diz respeito à transferência de dados, instanciação e remoção de recursos da nuvem, assim como com a capacidade de processamento propriamente dita de forma superficial. Com os modelos de transferência de dados e execução baseados no CSGrid, implementamos o novo módulo utilizando a infra-estrutura de plugins do CSGrid, nos fazendo valer de decisões de projeto com base nos experimentos previamente realizados e em conhecimento adquiridos através de tecnologias já existentes. A implementação realizada do nomeado SGA Azure foi submetida então a testes de stress que enviam comandos em série e monitoram o comportamento elástico do sistema, tornando evidente a redução de até 85% do tempo de computação com relação ao cenário serializado. Apresentamos, finalmente, na Seções 7.6 e 7.7 o relato sobre as dificuldades e sugestões de melhorias baseadas na experiência adquirida.

8.1
Trabalhos Futuros

Este trabalho abriu caminho para os seguintes trabalhos futuros:

- Pesquisar a influência da parametrização do SGA sobre o ganho de desempenho global em execuções múltiplas. O tempo de ociosidade das máquinas virtuais antes do suicídio e os tempos de *polling* do servidor CSGrid para demandar novos provisionamentos certamente têm influência sobre os custos de tempo e de dinheiro, mas a escolha de valores convenientes depende da carga submetida;
- Em função da Microsoft Azure impor um limite de 200 *Cloud Services*, porém um limite muito maior para máquinas virtuais, implementar

políticas de provisionamento de máquinas virtuais em diferentes *Cloud Services* sem perder os benefícios do provisionamento em paralelo;

– Aplicar o modelo do SGA Azure para outras nuvens públicas;

– Trabalhar variações do escalonamento interno do SGA de nuvem, uma vez que comandos diferentes demandando tamanhos e configurações de máquinas diferentes podem ter sua ordem de execução melhorada para favorecer o uso mais racional de recursos.

– Pesquisar políticas de escalonamento que não submetam todos os comandos da fila interna do CSGrid para o SGA, mantendo a possibilidade de melhor gestão de prioridade e balanceamento de carga entre diferentes SGAs ou usuários.

9
Referências Bibliográficas

ADAMS, K.; AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. **SIGARCH Computer Architecture News**, ACM, New York, NY, USA, v. 34, n. 5, p. 2–13, out. 2006. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/1168919.1168860>. 23

AMSTADT, B.; JOHNSON, M. K. Wine. **Linux Journal**, Belltown Media, v. 1994, n. 4es, p. 3, 1994. 43

ARMSTRONG, P. et al. Cloud Scheduler: a resource manager for distributed compute clouds. **CoRR**, abs/1007.0050, 2010. Disponível em: <http://arxiv.org/abs/1007.0050>. 47

Azure - Microsoft's Cloud Computing Platform. `http://azure.microsoft.com`. Accessed: 2012-04-11. 9, 30

BARROS, A. et al. Using fogbow to federate private clouds. In: **XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [S.l.: s.n.], 2015. 49

BAYUCAN, A. et al. **Portable batch system: External reference specification**. [S.l.], 1999. 9, 12

BOLSKY, M. I.; KORN, D. G. The kornshell command and programming language. **Englewood Cliffs: Prentice Hall, 1989**, v. 1, 1989. 19

BUYYA, R.; PATHAN, M.; VAKALI, A. **Content Delivery Networks**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 9783540778868. 41

DAGUM, L.; ENON, R. OpenMP: an industry standard API for shared-memory programming. **Computational Science & Engineering, IEEE**, IEEE, v. 5, n. 1, p. 46–55, 1998. 43

GROUP, O. M. **CORBA Component Model 4.0 Specification**. [S.l.], April 2006. Disponível em: <http://www.omg.org/docs/formal/06-04-01.pdf>. 15

LIANG, S.; BRACHA, G. Dynamic class loading in the java virtual machine. **ACM SIGPLAN Notices**, ACM, v. 33, n. 10, p. 36–44, 1998. 70

LIMA, M. J. de et al. CSBase: A framework for building customized grid environments. In: IEEE. **Enabling Technologies: Infrastructure for Collaborative**

Enterprises, 2006. WETICE'06. 15th IEEE International Workshops on. [S.l.], 2006. p. 187–194. 9, 11

MATEESCU, G.; GENTZSCH, W.; RIBBENS, C. J. Hybrid computing—where HPC meets grid and cloud computing. **Future Generation Computer Systems**, v. 27, n. 5, p. 440 – 453, 2011. ISSN 0167-739X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X1000213X>. 10

ORACLE VM VirtualBox. `http://www.virtualbox.org`. Accessed: 2014-12-11. 23

SANTOS, M. N. d. **GridFS - Um servidor de arquivos para grades e ambientes distribuídos heterogênos**. Dissertação (Mestrado) — PUC-Rio, 2006. 13

SHAHAN, R. et al. **Azure Storage Scalability and Performance Targets**. `https://azure.microsoft.com/en-us/documentation/articles/storage-scalability-targets/`. Accessed: 2015-07-14. 26

SHEPLER, S. et al. Network file system (NFS) version 4 protocol. **Network**, 2003. 13

THAIN, D.; TANNENBAUM, T.; LIVNY, M. Distributed computing in practice: the Condor experience. **Concurrency - Practice and Experience**, v. 17, n. 2-4, p. 323–356, 2005. 9, 12

THE JSON Data Interchange Standard. `http://www.json.org/`. Accessed: 2012-04-11. 52

TORQUE Resource Manager. `http://www.adaptivecomputing.com/products/open-source/torque/`. Accessed: 2012-04-11. 9, 12

TRöGER, P.; MERZKY, A. Towards standardized job submission and control in infrastructure clouds. **Journal of Grid Computing**, Springer Netherlands, v. 12, n. 1, p. 111–125, 2014. ISSN 1570-7873. Disponível em: <http://dx.doi.org/10.1007/s10723-013-9275-2>. 10

VMWare Virtualization for Desktop and Server. `http://www.vmware.com/`. Accessed: 2014-12-11. 23

A
Apêndice: Descrição do Sistema TATU

O TATU é um sistema de simulação numérica voltado para aspectos geomecânicos desenvolvido pela Alis/Tecgraf/PUC-Rio e foi escolhido para realização de um teste comparativo de desempenho por fazer uso intensivo e praticamente exclusivo de CPU.

Quando se extrai petróleo de um poço terrestre, o espaço ocupado pelo petróleo embaixo da terra pode ser substituído por outro fluido durante o processo de extração. Mesmo assim, a mudança de densidade envolvida e as pressões exercidas pela própria terra em volta do poço levam a deslocamentos que culminam em modificações na superfície, tipicamente afundamentos. O projeto Tatu visa simular as transformações sofridas na superfície dado o conhecimento sobre o tipo de solo e rocha encontrados ao redor dele, assim como em todo o trajeto entre o poço e o reservatório.

B
Apêndice: Código-fonte do programa Azure para medição de tempos usando CSGrid

Na Seção 4.3, foram apresentados resultados de medições usando o CSGrid, sendo que um programa diferenciado teve de ser desenvolvido para que a transferência de dados fosse feita usando o sistema de storage da Azure e a medida de tempo pudesse ser discriminada no experimento. Para os casos NFS e CSFS, a medição foi feita usando o próprio CSGrid e seu SGA lua padrão, tendo os marcadores de início e fim de cópia de dados inseridos no código do próprio CSGrid.

Abaixo o código-fonte do programa executado no CSGrid para medir os tempos expressos na Figura 4.2 e 4.3:

```
1  import java.io.File;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.io.PrintWriter;
7  import java.net.InetAddress;
8  import java.net.URISyntaxException;
9  import java.security.InvalidKeyException;
10 import java.util.zip.GZIPOutputStream;
11
12 //Include the following imports to use blob APIs.
13 import com.microsoft.azure.storage.CloudStorageAccount;
14 import com.microsoft.azure.storage.StorageException;
15 import com.microsoft.azure.storage.blob.CloudBlobClient;
16 import com.microsoft.azure.storage.blob.
       CloudBlobContainer;
17 import com.microsoft.azure.storage.blob.CloudBlockBlob;
18
19 /*
20  * Simple GZIP algorithm to CSGrid using Azure Storage
21  */
22 public class Benchmark {
23
24   //Define the connection-string with your values
25   public static final String storageConnectionString =
26       "DefaultEndpointsProtocol=http;" +
27       "AccountName=csgridprojects;" +
28       "AccountKey=someKeyWasHereISwareButItRunAway";
29
30   public static void main(String[] args)
31       throws InvalidKeyException, URISyntaxException,
32       StorageException, FileNotFoundException,
             IOException {
33
```

```
34      System.out.println("I am "+InetAddress.getLocalHost()
35          .getHostName());
36
37      String myName = InetAddress.getLocalHost().
           getHostName();
38      String myBlobName = myName+"/result.bin";
39      String downloadBlobName = "random.bin";
40      File projectDir =
41          new File(System.getenv("CSBASE_PROJ_DIR"));
42      if (!projectDir.isDirectory()){
43        throw new FileNotFoundException(
44            "Directory not found: "
45            +projectDir.getAbsolutePath());
46      }
47      // Retrieve storage account from connection-string.
48      CloudStorageAccount storageAccount =
49          CloudStorageAccount.parse(storageConnectionString
               );
50      // Create the blob client.
51      CloudBlobClient blobClient = storageAccount
52          .createCloudBlobClient();
53      // Writer to log that will be used to data
           aggregation
54      PrintWriter logOut = new PrintWriter(
55          new File(projectDir, myName+".upDownTimes.csv"));
56      // Get a reference to a container.
57      // The container name must be lower case
58      CloudBlobContainer container =
59          blobClient.getContainerReference("projects");
60      CloudBlockBlob downloadBlob =
61          container.getBlockBlobReference(downloadBlobName)
               ;
62      // Timer
63      long startTimer = System.currentTimeMillis();
64      // Mark download start
65      logOut.println(myName+",START_DOWNLOAD,"+startTimer);
66      // Do download
67      File outFile = File.createTempFile("azure-io-b", ".
           bin");
68      System.out.println("Downloading blob "+
           downloadBlobName
69          +" to "+outFile.getAbsolutePath());
70      FileOutputStream out = new FileOutputStream(outFile);
71      downloadBlob.download(out);
72      out.flush();
73      out.close();
74      // Mark download end time
75      long downTimer = System.currentTimeMillis();
76      // Now starting GZIP stage
77      logOut.println(myName+",START_GZIP,"+downTimer);
78      File outFileGz =
79          File.createTempFile("azure-io-b", ".bin.gz");
80      System.out.println("GZip compressing from "
81          +outFile.getAbsolutePath()
82          +" to "+outFileGz.getAbsolutePath());
```

```
83      FileOutputStream outGz = new FileOutputStream(
            outFileGz);
84      GZIPOutputStream gzOut = new GZIPOutputStream(outGz);
85      // Write do GZIP stream with a 1M buffer
86      FileInputStream fIn = new FileInputStream(outFile);
87      byte[] buffer = new byte[1024 * 1024];
88      int tam;
89      while ((tam = fIn.read(buffer))>0){
90        gzOut.write(buffer, 0, tam);
91      }
92      gzOut.flush();
93      outGz.flush();
94      outGz.close();
95      fIn.close();
96      // Mark GZIP end time
97      long gzTimer = System.currentTimeMillis();
98      logOut.println(myName+",START_UPLOAD,"+gzTimer);
99      System.out.println("Uploading "+outFileGz.
            getAbsolutePath()
100          +" to blob "+myBlobName);
101     CloudBlockBlob uploadBlob =
102         container.getBlockBlobReference(myBlobName);
103     // Do upload using Azure API
104     uploadBlob.uploadFromFile(outFileGz.getAbsolutePath()
            );
105     // Mark upload end time
106     long upTimer = System.currentTimeMillis();
107     logOut.println(myName+",END,"+upTimer);
108     logOut.flush();
109     logOut.close();
110     // Cleanup
111     outFileGz.delete();
112     outFile.delete();
113     System.out.println("ok.");
114   }
115 }
```

C
Apêndice: Código-fonte do Loop Principal do Executor do SGA Azure

Abaixo o código-fonte do programa executado em cada nó de execução do SGA Azure:

```
from daemon import Daemon
from sga_shared_data_structures import CommandMetadata
from SandboxManager import SandboxManager
from subprocess import Popen
import executor
import ConfigParser, imp, socket

import time, random
from datetime import datetime

sga_debug = True

def trace(message):
    if sga_debug:
        print message

class SgaDaemon(Daemon):

    def run(self):
        running_process = None
        execution_metadata = None
        self.execution_stdout = None

        trace("Reading config...")
        config = ConfigParser.ConfigParser()
        config.read("sga.ini")

        # The OS user name, to be used to delegate the
           program execution
        self.os_execution_username = config.get("sga", "
           os_execution_username")

        # The name of the executor itself
        machine_name = socket.gethostname()
        config.set("sga", "name", machine_name)

        # Load the connector module
        trace("Loading connector...")
        connector_module = imp.load_source("connector",
           config.get("sga", "connector"))
        self.connector = connector_module.create(config)

        self.sandbox_manager = SandboxManager(config,
           self.connector)
```

```
41
42          # General polling configuration
43          polling_time_secs = config.getfloat("sga", "
                polling_time_secs")
44
45          try:
46              # Main event loop
47              while True:
48
49                  # If there is no current running job,
50                  # let's check on the queue to something
                        to run
51                  if running_process is None:
52                      trace("Looking for new job...")
53                      execution_metadata = self.
                            __check_new_tasks()
54                      if not execution_metadata is None:
55                          trace("New job found. Starting
                                execution...")
56                          running_process = self.
                                __start_execution(
                                execution_metadata)
57                          if running_process is None:
58                              print("Can't execute job.
                                    Aborting.")
59                              execution_metadata = None
60                      else:
61                          trace(" Nothing to do.")
62                          self.__shutdown_myself()
63
64                  else:
65                      if running_process.poll() is not None
                            :
66                          # Job completed.
67                          trace("Job completed. Finalizing.
                                ")
68                          self.__end_execution(
                                execution_metadata)
69                          execution_metadata = None
70                          running_process = None
71
72                      else:
73                          # Job still running.
74                          trace("Job is running...")
75                          if self.__check_cancel_request(
                                execution_metadata):
76                              trace("Cancell command
                                    detected. Interrupting
                                    process...")
77                              self.__send_status(
                                    execution_metadata, "
                                    Canceling")
78                              self.__cancel_execution(
                                    running_process)
```

```
79                              self.__send_status(
                                    execution_metadata, "
                                    Canceled")
80                          else:
81                              # Job running normally. Wait
                                    plus 1 second to iterate
82                              # main loop again
83                              time.sleep(1)
84
85                  time.sleep(polling_time_secs)
86
87          except KeyboardInterrupt:
88              trace("Bye bye.")
89
90      def __check_new_tasks(self):
91          return self.connector.check_new_tasks()
92
93      def __start_execution(self, execution_metadata):
94
95          self.__send_status(execution_metadata, "
                Downloading")
96          start = time.time()
97          algorithm_dir = self.sandbox_manager.
                get_algorithm(execution_metadata.
                algorithm_bin_file)
98          project_sandbox_dir = self.sandbox_manager.
                get_project(
99              project_prfx=execution_metadata.project_prfx,
100             project_input_files=execution_metadata.
                    project_input_files)
101
102         if self.execution_stdout is not None:
103             self.execution_stdout.close()
104
105         self.execution_stdout = open("/tmp/out.txt", "w")
106
107         execution_metadata.time_download = (time.time() -
                start)
108
109         execution_metadata.time_start_run = time.time()
110         self.__send_status(execution_metadata, "Running")
111         return executor.execute(executor.Execution(
112             algorithm_executable=execution_metadata.
                    algorithm_executable_name,
113             algorithm_params=execution_metadata.
                    algorithm_parameters,
114             algorithm_dir=algorithm_dir,
115             project_sandbox_dir=project_sandbox_dir,
116             os_username=self.os_execution_username,
117             stdout=self.execution_stdout,
118             stderr=self.execution_stdout
119         ))
120
121     def __end_execution(self, execution_metadata):
```

```
122         execution_metadata.time_run = (time.time() -
               execution_metadata.time_start_run)
123         start_upload = time.time()
124         self.__send_status(execution_metadata, "Uploading
               ")
125         out_prfx = self.sandbox_manager.
               upload_project_modified_files(
               execution_metadata.project_prfx)
126         execution_metadata.time_upload = (time.time() -
               start_upload)
127         if not out_prfx is None:
128             self.sandbox_manager.upload_log_project_file(
                   execution_metadata.project_prfx,
                   execution_metadata, out_prfx)
129         self.__send_status(execution_metadata, "Ended")
130
131     def __cancel_execution(self, running_process):
132         counter = 30
133         if running_process.poll() is not None:
134             if counter > 0:
135                 running_process.terminate()
136             else:
137                 running_process.kill()
138             time.sleep(1)
139         return
140
141     def __check_cancel_request(self, execution_metadata):
142         return
143
144     def __send_status(self, execution_metadata,
           main_status):
145         # Format the JSON message
146         status_message = "{cmdId:\"" + execution_metadata
               .command_id + "\", status:\""+main_status+"\",
                vmName:\""+self.connector.myMachineName+"\"}"
147         self.connector.send_status(status_message)
148
149     def __shutdown_myself(self):
150         # shutdown the virtual machine
151         self.connector.shutdown_myself()
```