Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem Baseada em Testes Automáticos de Software para Diagnóstico de Faltas em Grades Computacionais

Alexandre Nóbrega Duarte

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Brasileiro
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

D812a Duarte, Alexandre Nóbrega.

Uma abordagem baseada em testes automáticos de software para diagnóstico de faltas em grades computacionais / Alexandre Nóbrega Duarte. - Campina Grande, 2010.

122f. : il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientador : Profº. Francisco Brasileiro.

1. Redes de Computadores. 2. Computação em Grades. 3. Diagnóstico de Faltas. 4. Testes de Software. I. Título.

CDU 004.7(043)

Resumo

Muito tem se pesquisado nos últimos anos sobre tratamento de falhas com o objetivo de aumentar a confiabilidade de infraestruturas de grade computacional. Idealmente, um usuário de grade deve ser capaz de submeter um conjunto de tarefas para execução em tal infraestrutura, aguardar pelo término da execução e, em seguida, obter o resultado das suas tarefas da mesma forma que faria caso utilizasse uma única máquina de grande poder computacional. Na prática, porém, não é isso o que vem ocorrendo com usuários das maiores grades computacionais disponíveis atualmente. Não é incomum observar altas taxas de falha nas tarefas submetidas para grades computacionais. Usuários de grades vêem a execução de suas tarefas falhar e não recebem qualquer informação que possa ajudá-los a entender porque suas tarefas falharam. Muitas vezes o usuário não consegue sequer identificar se sua tarefa falhou por um defeito no software de sua aplicação, ou por um problema em um serviço de grade localizado do outro lado do globo. Esta tese propõe e avalia um mecanismo baseado na utilização de testes automáticos de software para detectar falhas na execução de aplicações neste tipo de infraestrutura e diagnosticar as causas de tais falhas. Resultados experimentais demonstraram uma taxa de acerto de $93,99\% \pm 5,63\%$, com um nível de confiança de 95%, ou de $93,99\% \pm 7,52\%$, com um nível de confiança de 99%, nos diagnósticos efetuados por uma ferramenta implementada de acordo com o mecanismo proposto.

Palavras-chave: Diagnóstico de Faltas, Grades Computacionais, Testes de Software.

Abstract

A lot of research effort has being put to find better mechanisms for fault treatment on grid computing aiming at improving the reliability of such infrastructures. Ideally, a grid user should be able to submit a set of tasks to remote execution, wait until the execution is concluded, and then retrieve the results of its execution in the very same way she would do if using a single high-performance machine. In practice, however, this is not what is happening to users of the larger grid infrastructures available nowadays. It is not rare to observe high failure rates on tasks submitted for execution in a grid infrastructure. Grid users see their tasks failing and receive no feedback from the grid middleware that could possibly help them to figure out why their tasks failed. Must of the time the user is not even able to tell if the task failed due to a problem inside the user application or due to some faulty service located somewhere in the grid.This thesis proposes and evaluates a mechanism based on the utilization of automatic software tests to detect failures and to diagnose their causes during the execution of applications on this kind of infrastructure. Experimental results showed a success rate of $93.99\% \pm 5,63\%$, with a 95% cofidence level, or $93.99\% \pm 7.52\%$. with a 99% confidence level, for the dianosis of a tool implemented according to the proposed mechanism.

Keywords: Fault Diagnosis, Grid Computing, Software Tests.

Agradecimentos

Agradeço a todos os meus familiares e amigos que tanto me incentivaram a prosseguir com esta empreitada. Em particular, minha mãe, Emeide Nóbrega, que sempre foi uma fonte de inspiração para mim e para todos a sua volta.

Agradeço às "meninas da COPIN", Aninha e Vera pela ajuda operacional que me deram durante todo este tempo que por lá estive.

Agradeço ao meu orientador, Francisco Brasileiro (Fubica) pelas discussões sempre frutíferas sobre as direções e rumos a serem seguidos durante o desenvolvimento deste trabalho.

Agradeço a minha amada esposa Dayse, por sua paciência e companheirismo em todos os momentos que passamos juntos.

Finalmente, dedico este trabalho a minha pequena filha Sofia, que hoje, com seus quase 6 meses de idade e largo sorriso, já é a minha maior fonte de inspiração e razão para seguir sempre em frente.

Conteúdo

Lista de Símbolos

ATLAS : *A Toroidal LHC Apparatus*

BDII : *Berkeley Database Information Index*

CASTOR : *CERN Advanced STORage*

CERN : *European Organization for Nuclear Research*

CE : *Computing Element*

CMS : *Compact Muon Solenoid*

DPM : *Disk Pool Management*

EELA : *E-science grid facility for Europe and LatinAmerica*

EGEE : *Enabling Grids for E-sciencE*

FDS : *Failure Detection System*

GIIS : *Grid Index Information Services*

GLUE : *Grid Laboratory Uniform Environment*

GMA : *Grid Monitoring Architecture*

GOC : *Grid Operations Center*

GRIS : *Grid Resource Information Services*

GSI : *Grid Security Infrastructure*

JVM : *Java Virtual Machine*

LDAP : *Lightweight Directory Access Protocol*

LHC : *Large Hadron Collider*

LHCb : *Large Hadron Collider beauty*

LRMS : *Local Resource Management System*

LSF : *Load Sharing Facility*

MDS : *Monitoring and Discovery Service*

PBS : *Portable Batch System*

SAM : *Service Availability Monitoring*

SE : *Storage Element*

SOAP : *Simple Object Access Protocol*

SPMD : *Single Process Multiple Data*

SRM : *Storage Resource Manager*

SSL : *Secure Socket Layer*

VOMS : *Virtual Organization Management System*

WLCG : *Worldwide LHC Computing Grid*

WMS : *Workload Management System*

WQR : *Workqueue with Replication*

XML : *Extensible Markup Language*

gLite : *Lightweight Middleware for Grid Computing*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

O objetivo geral deste trabalho é investigar como testes automáticos de software podem ser utilizados para detecção de falhas e diagnóstico de faltas em serviços e aplicações de grades computacionais[1]. Visamos também prover mecanismos baseados na utilização de testes automáticos de aceitação para facilitar a localização de serviços defeituosos neste tipo de infraestrutura de computação distribuída.

No restante deste capítulo, nós discutimos a relevância deste trabalho (Seção 1.1), detalhamos seus objetivos gerais e específicos (Seção 1.2), descrevemos a metodologia que delineou este trabalho (Seção 1.3); resumimos nossas contribuições (Seção 1.4) e apresentamos a organização do restante do documento (Seção 1.5).

1.1 Justificativa e Relevância

Grades computacionais têm o potencial para revolucionar a computação de alto desempenho ao prover acesso ubíquo e sob demanda a serviços e recursos computacionais. Tais infraestruturas de computação distribuída visam permitir o acesso e composição sob-demanda de serviços fornecidos por diversos provedores, possivelmente pertencentes a diferentes domínios administrativos, prometendo níveis de paralelismo sem igual para aplicações de alto desempenho. Estas possibilidades criaram um terreno fértil para o desenvolvimento de uma nova gama de aplicações, muito mais ubíquas e adaptativas e com requisitos de processamento e armazenamento muito mais ambiciosos. Por outro lado, as características intrínsecas

[1]Neste trabalho utilizamos os termos falha e falta como traduções para os termos *failure* e *fault* do inglês.

de uma grade computacional, como sua alta heterogeneidade, complexidade e distribuição, muitas vezes englobando múltiplos domínios administrativos, criaram vários novos desafios técnicos que precisam ser superados para que essa visão seja realizada.

Muito tem se pesquisado nos últimos anos sobre tratamento de falhas com o objetivo de aumentar a confiabilidade de infraestruturas de grade computacional [Hwang e Kesselman 2003; Kola, Kosar e Livny 2004; Medeiros et al. 2003; Litke et al. 2007; Dabrowski 2009; Qian-Mu, Man-wu e Hong 2006; Duarte et al. 2006]. Idealmente, um usuário de grade deve ser capaz de submeter um conjunto de tarefas para execução em tal infraestrutura, aguardar pelo término da execução e, em seguida, obter o resultado das suas tarefas da mesma forma que faria caso utilizasse uma única máquina de grande poder computacional.

Na prática, porém, não é isso o que vem ocorrendo com usuários das maiores grades computacionais disponíveis atualmente. Não é incomum observar altas taxas de falha nas tarefas submetidas para grades computacionais de larga escala. Mecanismos deficientes não conseguem detectar se os diversos serviços oferecidos por uma grade computacional funcionam como deveriam. Usuários de grades vêem a execução de suas tarefas falhar e não recebem qualquer informação que possa ajudá-los a entender o porque da falha.

Etapas como a de diagnóstico de faltas, chave em qualquer estratégia de tratamento de falhas, precisam ser aprimoradas para permitir que todo o potencial das grades computacionais possa ser explorado eficientemente [Dabrowski 2009].

Atualmente, quando a execução de uma aplicação de grade não é bem sucedida, o processo para se descobrir a razão pela qual a aplicação falhou, i.e., a causa raiz da falha, é extremamente complexo [Medeiros et al. 2003]. A aplicação pode falhar devido a um erro de codificação na própria aplicação; ou por conta de um problema de configuração quando a aplicação é executada pela primeira vez em um novo nó da grade; ou devido a um erro de autenticação ou autorização do usuário gerado por um certificado com tempo de vida expirado; ou porque um disco rígido falhou em algum nó da grade. A lista de possibilidades é muito grande. Para começar, até mesmo saber se uma execução não foi bem sucedida por conta de um problema na própria aplicação ou por um problema em algum serviço da grade já é suficientemente difícil. Uma vez que uma grade computacional combina serviços e recursos localizados em múltiplos domínios administrativos, podem haver restrições na obtenção de informações sobre o estado dos serviços da grade. Mesmo quando tal informação está li-

vremente disponível, é preciso entender o que deveria estar acontecendo mas não está para poder fazer um diagnóstico correto. Para complicar um pouco mais o processo, mensagens de erro tendem a ser pouco úteis para elucidar as causas de um problema. Um estudo mostrou que até mesmo usuários especializados, como administradores de sistema, podem gastar até 25% do seu tempo seguindo pistas incorretas levantadas por mensagens de erro ambíguas ou pouco significativas [Barrett et al. 2004].

Sistemas computacionais fazem uso de abstrações para poder ocultar a complexidade. Para utilizar um serviço não é preciso saber como ele funciona, apenas o que ele faz. Infelizmente, quando ocorre uma falha, as abstrações perdem sua habilidade de esconder complexidade. É preciso entender como elas deveriam funcionar (ao invés de apenas o que elas deveriam fazer) para poder diagnosticar e corrigir o problema. No contexto das grades computacionais, isso significa entender o funcionamento interno dos vários serviços disponíveis. Tal entendimento requer *expertise* em várias tecnologias diferentes em termos de *middleware*, sistemas operacionais e hardware. Conhecimento demais para qualquer ser humano.

1.1.1 *Survey* com Usuários de Grades Computacionais

Consultamos os usuários de grades computacionais para tentar identificar o *status quo* do tratamento de falhas em grades [Duarte et al. 2006; Duarte 2009]. As seguintes questões foram reunidas em um questionário e enviadas para usuários ao redor do mundo:

- Quais são os tipos de falhas mais frequentes que você enfrenta ao utilizar uma grade computacional?
- Quais são os mecanismos utilizados para detectar, corrigir ou tolerar falhas?
- Quais são os maiores problemas encontrados quando é preciso se recuperar de uma falha?

Um questionário contendo essas questões foi disponibilizado na Web e anunciado em várias listas de discussão de usuários de grades computacionais. As respostas foram recebidas através do formulário Web e também por e-mail. Coletamos respostas para o questionário em três etapas. A primeira, em Abril 2003, resultou em 22 respostas, a segunda, em Abril

de 2005, resultou em 13 respostas e a terceira, em Abril de 2009, resultou em 83 respostas. Apesar do reduzido número de respostas, elas fornecem bons indícios sobre quais são os principais problemas relacionados ao tratamento de falhas enfrentados pela comunidade de usuários de grades computacionais.

Tipos de falhas

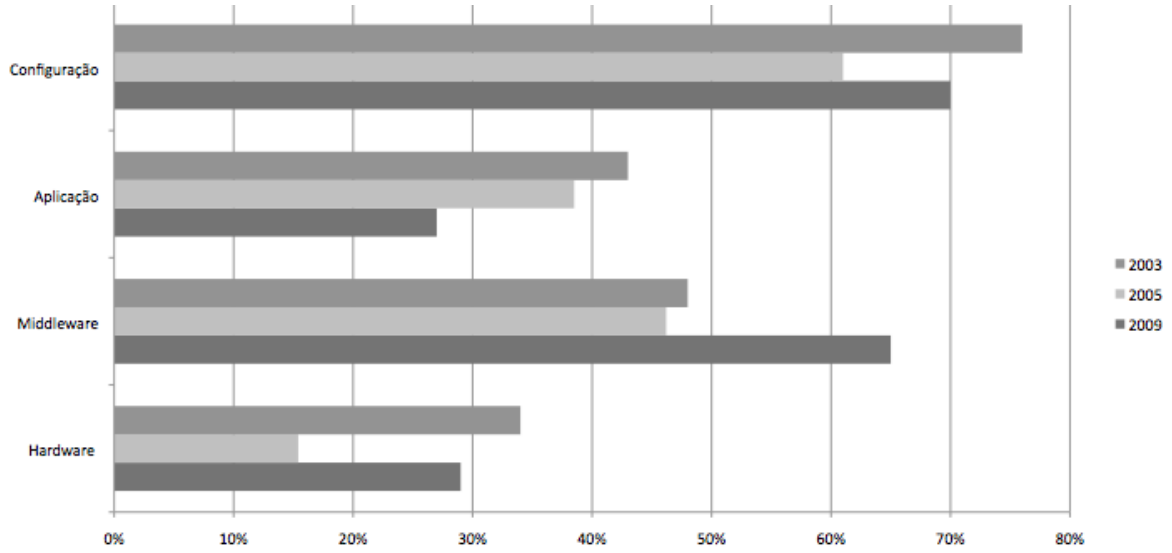


Figura 1.1: Tipos de Falhas

A Figura 1.1 ilustra as respostas recebidas para a pergunta *Quais são os tipos de falhas mais frequentes que você enfrenta ao utilizar uma grade computacional?* A partir dos dados coletados podemos dizer que os tipos de falhas mais frequentes em 2003 e 2005 permaneceram praticamente os mesmos em 2009. A principal causa apontada para as falhas na execução de aplicações em grades computacionais são problemas relacionados à configuração do ambiente de execução, como apontaram cerca de 75% dos usuários em 2003, pouco mais de 60% em 2005 e 70% em 2009. Em seguida vêm as falhas causadas por problemas no *middleware*, com pouco menos de 50% em 2003 e 2005 e cerca de 60% em 2009. Falhas da própria aplicação foram a terceira causa mais frequente em 2003 e 2005 mas em 2009 essa posição foi ocupada por falhas de hardware.

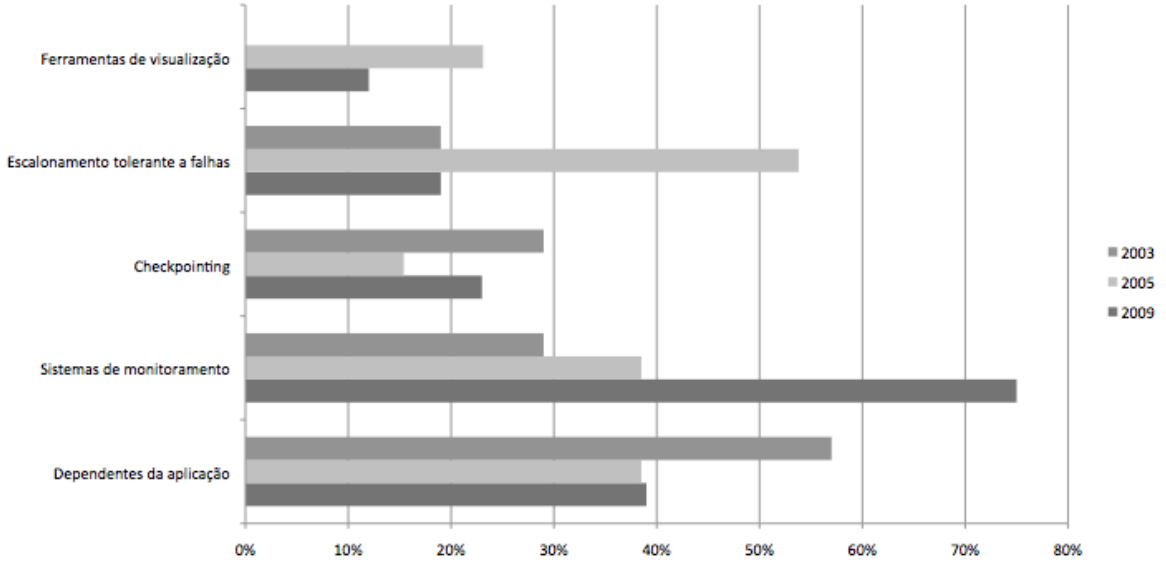


Figura 1.2: Mecanismos de Tratamento de Falhas

Mecanismos de tratamento de falhas mais utilizados

A Figura 1.2 ilustra as repostas recebidas para a pergunta *Quais são os mecanismos utilizados para detectar, corrigir ou tolerar falhas?* e mostra que existem diferenças interessantes entre os dados coletados nas diferentes etapas do *survey*. Em 2003, além de mecanismos *ad hoc*, baseados na análise de arquivos de *log* pelos usuários, haviam também mecanismos automáticos para lidar com falhas em seus sistemas. No entanto, 57% deles eram soluções dependentes da aplicação e 29% eram ferramentas de monitoramento. *Checkpointing* era um recurso utilizado por 29% dos usuários e escalonamento tolerante a falhas por 19%. Em alguns casos, diferentes mecanismos eram combinados. Em 2005, escalonamento tolerante a falhas foi a solução mais utilizada, segundo 53% das respostas. Soluções dependentes da aplicação e ferramentas de monitoramento eram utilizados por quase 40% dos usuários. 22% dos usuários mencionaram o uso de ferramentas de visualização para auxiliar no tratamento de falhas. Em 2009 os sistemas de monitoramento são a principal ferramenta utilizada pelos usuários para lidar com falhas em grades, como apontado por cerca de 75% dos participantes do *survey*. Em segundo lugar, com quase 40% estão soluções dependentes da aplicação, seguidas por *checkpointing* (22%), escalonamento tolerante a falhas (19%) e ferramentas de visualização (12%). Esses números indicam que mecanismos de tratamento de falhas são

importantes e utilizados pela grande maioria dos usuários de grade.

Maiores problemas para se recuperar de uma falha

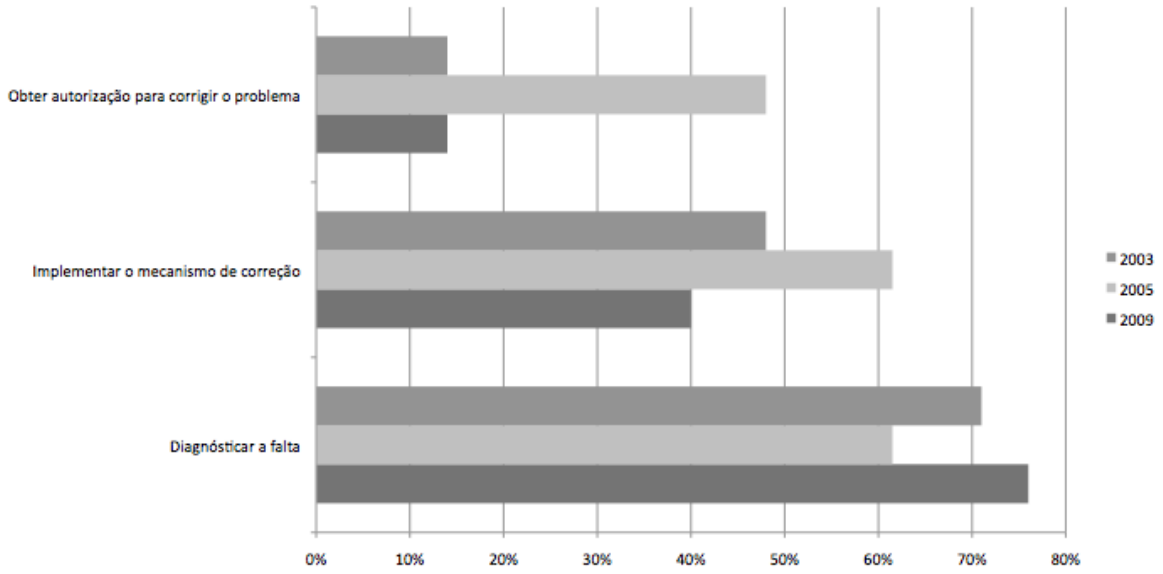


Figura 1.3: Maiores problemas para se recuperar de uma falha

A Figura 1.3 mostra as respostas recebidas para a pergunta *Quais são os maiores problemas encontrados quando é preciso se recuperar de uma falha?* e mostra resultados bem similares. Nas três rodadas do *survey* o principal problema enfrentado pelos usuários na hora de se recuperar de uma falha foi diagnosticar a causa da falha, como apontaram cerca de 71% dos usuários em 2003, 61% em 2005 e 75% em 2009.

A dificuldade em implementar mecanismos dependentes da aplicação para se recuperar de falhas foi o segundo maior problema para 48% dos usuários em 2003, 61% em 2005 e 40% em 2009. A impossibilidade de se obter acesso para corrigir um componente com problemas foi reportado nas três rodadas do *survey* por cerca de 13% em 2003 e 2009 e por cerca de 48% dos usuários em 2005, sendo o problema menos comum nas três rodadas do *survey*.

Perfil dos participantes do *survey*

Na última rodada do *survey*, aplicada em 2009, decidimos identificar o *middleware* e a(s) infraestrutura(s) de grade computacional utilizadas pelos participantes na pesquisa para ter-

mos uma melhor visão da abrangência das respostas recebidas. Por essa razão, adicionamos as seguintes questões ao formulário:

- Que middleware é utilizado em sua infraestrutura de grade?
- Você é usuário de quais grades computacionais?

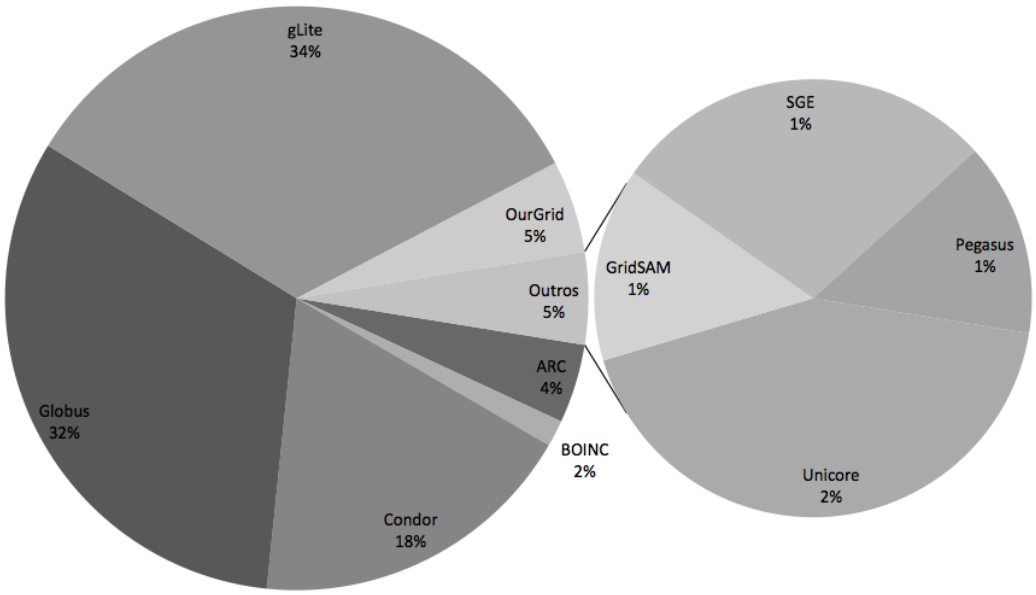


Figura 1.4: *Middleware* para computação em grade utilizados pelos participantes do *survey*

A Figura 1.4 identifica os 10 diferentes *middleware* para computação em grades utilizados pelos participantes do *survey* em 2009. *gLite* [Gagliardi 2005] (55%), *Globus* [Foster e Kesselman 1997] (53%) e *Condor* [Litzkow, Livny e Mutka 1988] (30%) são os principais *middleware* para computação em grades, segundo os usuários consultados, seguidos por *OurGrid* [Cirne et al. 2006] (8%) e ARC [Eerola et al. 2003] (7%). Também foram mencionados o *Unicore* [Erwin e Snelling 2001] (3%), *BOINC* [Anderson 2004] (2%), *Sun Grid Engine* [Gentzsch 2001] (2%), *GridSAM* [Wang et al. 2008] (1%) e Pegasus [Deelman et al. 2004] (1%).

A Figura 1.5 exibe as 16 infraestruturas de grades computacionais utilizadas pelos usuários que responderam ao *survey*. O *WLCG/EGEE* [Gagliardi 2005] é a infraestrutura de grades mais utilizada, servindo cerca de 41% dos usuários. Em seguida vem o *Open Science*

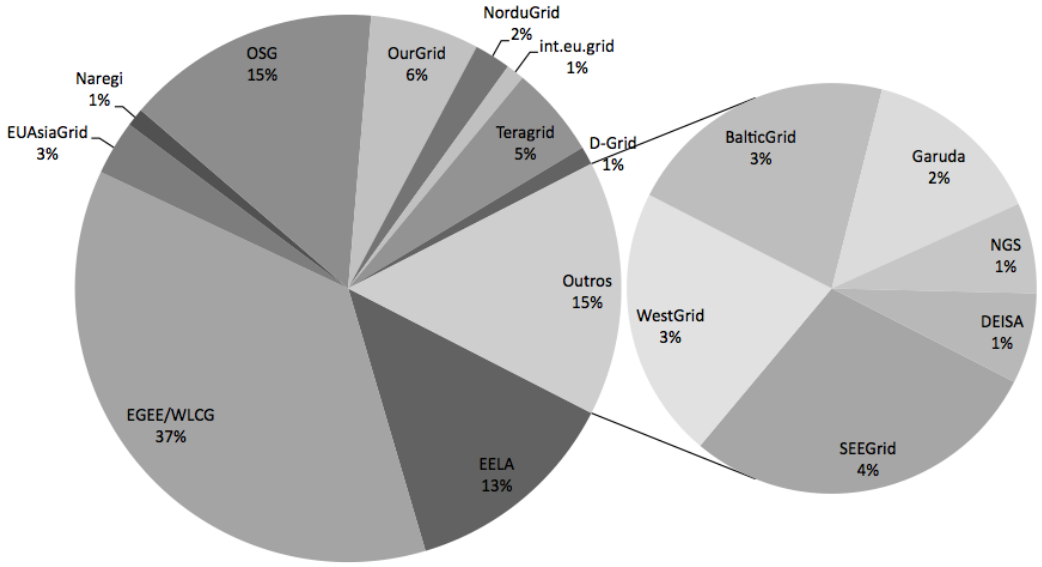


Figura 1.5: Infraestruturas de grade utilizadas pelos participantes do *survey*

Grid [Pordes et al. 2007], com cerca de 17%, seguida de perto pela infraestrutura criada pelo projeto *EELA* [EELA 2010], com quase 15%. Duas das três infraestruturas mais utilizadas, *WLCG/EGEE* e *EELA*, são baseadas no *middleware gLite* enquanto que o OSG utiliza o *Globus*, o que explica o fato destes dois serem os *middleware* mais populares como mostrado na Figura 1.4. Em seguida, com valores oscilando entre 7% e 4% aparecem as infraestruturas dos projetos *OurGrid* [Cirne et al. 2006], *TeraGrid* [Catlett 2005] e *SEEGrid* [SEEGrid 2010]. Várias outras infraestruturas foram mencionadas por menos de 3% dos participantes do *survey*, como: *EUAsiaGrid* [EuAsiaGrid 2010], *Naregi* [NAREGI 2010], *Nordugrid* [Eerola et al. 2003], *int.eu.grid* [int.eu.grid 2010], *D-Grid* [D-Grid 2010], *WestGrid* [WestGrid 2010], *BalticGrid* [BalticGrid 2010], *Garuda* [GARUDA 2010], *NGS* [NGS 2010] e *DEISA* [DEISA 2010].

Apesar de não podermos afirmar com certeza qual o percentual das infraestruturas de grade computacional ou de *middleware* para computação em grades existentes hoje cobertos pelo *survey*, podemos observar uma excelente cobertura geográfica. As infraestruturas de grade computacional citadas no *survey* atendem usuários da América Latina (*EELA*), América do Norte (*OSG, WestGrid*), Europa (*WLCG/EGEE, NorduGrid, int.eu.grid, D-Grid, Bal-*

ticGrid, NGS e DEISA) e Ásia (*EuAsiaGrid,Naregi, e Garuda*).

1.1.2 Análise da Grade Computacional do Projeto WLCG/EGEE

Os números citados na seção anterior apresentam indícios de que, apesar de algum progresso ter sido feito, tratamento de falhas ainda está longe de ser um problema resolvido em se tratando de sistemas distribuídos complexos como grades computacionais.

Além dos indícios levantados pelo *survey* com os usuários de grades computacionais, analisamos também resultados de execuções submetidas por usuários da grade computacional mais utilizada pelos participantes do nosso *survey*. O WLCG/EGEE é uma das maiores grades computacionais em produção no momento, criada no contexto da série de projetos Europeus EGEE [Gagliardi 2005], e que a partir de 2010 será gerenciada pela *European Grid Initiative*[The European Grid Initiative 2010], uma organização criada para coordenar uma infraestrutura de grade para todo o continente Europeu.

Atualmente tal grade computacional possui mais de 68.000 unidades de processamento e 20 petabytes de capacidade de armazenamento, espalhadas em 250 sítios localizados em 50 países. Atende cerca de 10 mil usuários que submetem até 188 mil tarefas por dia [Gagliardi 2005]. Os usuários do WLCG/EGEE são organizados em grupos chamados Organizações Virtuais [Foster e Kesselman 1997]. Nós analisamos os resultados obtidos por três das maiores Organizações Virtuais do WLCG/EGEE (LHCb, CMS e Atlas), todas relacionadas a experimentos específicos do *Large Hadron Collider* [The Large Hadron Collider Collaboration 2010] do CERN [CERN 2010]. Usuários destas três Organizações Virtuais submeteram mais de 18 milhões de tarefas em 2007, o que corresponde a cerca de um terço da carga total anual da infraestrutura naquele ano.

Os resultados da análise são ilustrados na Tabela 1.1. Pode-se observar que a taxa de falhas para as tarefas submetidas pelos usuários da Organização Virtual LCHb foi de 60%. Deste montante, 35% tiveram causa desconhecida. As tarefas dos usuários da Organização Virtual CMS tiveram uma taxa de falha menor, de cerca de 30%, porém mantiveram uma percentagem semelhante (27%) de falhas com causa desconhecida. O resultado para as tarefas submetidas pelos usuários da Organização Virtual Atlas foi diferente dos resultados anteriores. A taxa de falhas foi de apenas 5%, o que seria um valor considerado normal para uma infraestrutura de grade, não fosse o fato de que 74% das tarefas submetidas tinham estado

Tabela 1.1: Análise WLCG/EGEE

LHCb	CMS	Atlas
Taxa de Falhas de 60%	Taxa de Falhas de 30%	Taxa de Falhas de 5%
35% das Falhas com Causa Desconhecida	27% das Falhas com Causa Desconhecida	74% das tarefas com estado final desconhecido
2.873.243 tarefas submetidas	3.646.985 tarefas submetidas	11.867.361 tarefas submetidas

final desconhecido, ou seja, não foi possível sequer saber se as tarefas foram executadas, muito menos se elas foram concluídas com sucesso ou não. O resultado final aponta para mais de **12 milhões de tarefas falhas ou em estado desconhecido**, implicando em uma taxa de sucesso de menos de 40%.

Um outro trabalho analisou noves meses de tarefas executadas nos centros de recursos do WLCG/EGEE localizados no sudoeste da Europa em 2007 e obteve resultados semelhantes aos nossos, determinando que apenas 48% das tarefas submetidas para estes centros de recursos foram executadas corretamente [Costa, Dikaiakos e Orlando 2007]. Outro resultado semelhante foi observado em 2005, quando foram observadas taxas de falha de até 35% em um experimento para encontrar a cura para uma variação da Malária [Nicolas et al. 2006].

O estudo realizado é bastante significativo mesmo sendo baseado em dados extraídos de uma única infraestrutura de grade computacional e representa bem o cenário atual na área, uma vez que o mesmo *middleware*, os mesmos serviços e os mesmos procedimentos operacionais adotados pelo WLCG/EGEE são também adotados por várias infraestruturas de grade regionais [Gagliardi 2005; The European Grid Initiative 2010; SEEGrid 2010; EELA 2010; BalticGrid 2010; NGS 2010; int.eu.grid 2010; D-Grid 2010; EuChinaGrid 2010; EuAsiaGrid 2010; The EUIndiaGrid Project 2010; ItalianGrid 2010] e nacionais [NGS 2010; ItalianGrid 2010] disponíveis atualmente, atendendo usuários espalhados por boa parte do globo.

Além disso, muitos esforços têm sido realizados nos últimos anos com o objetivo de permitir que grades baseadas em *middleware* diferentes possam interoperar e o *gLite* tem sido escolhido como o elo de ligação entre essas grades [Field e Schulz 2008; Brasileiro et al. 2008; Asvija et al. 2009; Wang, Cheng e Chen 2009; Nakada et al. 2008; Gronager et al.

2008; Riedel et al. 2008; Wang et al. 2007; Kacsuk, Farkas e Fedak 2008; Garzoglio et al. 2009].

Estes números ilustram bem a qualidade da informação fornecida pelo estado da prática em tratamento de falhas em grades computacionais e são corroborados por outros estudos.

1.2 Objetivos

O objetivo geral deste trabalho é investigar como testes automáticos de software podem ser utilizados para detecção de falhas e diagnóstico de faltas em serviços e aplicações de grades computacionais.

Para atingir este objetivo geral, definimos os seguintes objetivos específicos:

1. Investigar a possibilidade de se utilizar testes automáticos de software para detectar a ocorrência de falhas nos serviços oferecidos por uma grade computacional, em particular, falhas relacionadas à configuração do ambiente, apontada em nosso *survey* como a principal causa de falhas nas execuções de aplicações de grade.

2. Projetar um mecanismo para o diagnóstico automático de faltas em aplicações e serviços de grades computacionais, baseado no uso de testes automáticos.

3. Analisar a precisão dos diagnósticos efetuados automaticamente pela solução proposta através da realização de um experimento prático utilizando uma grade computacional em produção.

1.3 Metodologia

O desenvolvimento deste trabalho iniciou-se com a condução de uma consulta à comunidade de usuários de grades computacionais para identificar o *status quo* dos mecanismos de tratamento de falhas utilizados nesse tipo de infraestrutura de computação distribuída.

As consultas apontaram uma lacuna no tocante ao diagnóstico de faltas neste tipo de infraestrutura e nos levaram a iniciar uma análise da carga de trabalho de um grade computacional de grande porte, em produção, e utilizada diariamente por um grande número de usuários. Constatamos em nossa análise altas taxas de falhas apresentadas aos seus usuários

e mais uma vez observamos a falta de mecanismos apropriados para informar aos usuários da grade as causas das falhas na execução de suas aplicações.

Tendo trabalhado com testes automáticos de software durante os anos que antecederam este trabalho de doutorado, decidimos investigar a hipótese de que testes automáticos de software poderiam ser utilizados tanto para detectar falhas quanto para diagnosticar faltas em grades computacionais e propor uma solução para o problema reportado pelos usuários de grades e observado em nossa análise da carga de trabalho.

Realizamos um experimento inicial em ambiente controlado para constatar a viabilidade da nossa abordagem antes de partir para um experimento de grande porte, utilizando uma grade computacional em produção.

Para a realização do experimento de grande porte foi necessário desenvolver uma ferramenta capaz de detectar falhas em serviços de uma grade utilizando testes automáticos de software e em seguida criar um mecanismo capaz de correlacionar os resultados dos testes realizados pela ferramenta de detecção a fim de identificar a origem de uma eventual falha na execução de uma aplicação na grade.

Participamos do time que desenvolveu a ferramenta de detecção de falhas durante um estágio de 20 meses realizado no CERN. Tal ferramenta, batizada de *Service Availabily Monitoring*, ou simplesmente *SAM*, acabou sendo adotada como principal ferramenta de monitoramento em diversas infraestruturas de grade espalhadas pelo mundo.

Em seguida, já de volta ao Brasil, desenvolvemos um *framework* denominado *GridDoctor* para o dianóstico de faltas em grades e instanciamos esse *framework* em uma ferramenta para grades baseadas no *middleware gLite*. Tal ferramenta foi capaz de utilizar os resultados dos testes executados pelo *SAM* para diagnosticar a causa de uma falha na execução de uma aplicação do usuário, apontando o serviço da grade onde a falta que deu origem à falha ocorreu.

Utilizando estas duas ferramentas foi possível realizar um experimento de grande porte em uma grade computacional real e obter resultados experimentais que oferecem fortes evidências de que é possível utilizar com sucesso testes automáticos de software para detectar falhas e diagnosticar faltas em grades computacionais. Os resultados experimentais demonstraram uma taxa de acerto de $93,99\% \pm 5,63\%$, com um nível de confiança de 95%, ou de $93,99\% \pm 7,52\%$, com um nível de confiança de 99%, nos diagnósticos efetuados por uma

ferramenta implementada de acordo com o mecanismo proposto.

1.4 Contribuições

A principal contribuição deste trabalho é uma abordagem para o diagnóstico de faltas em grades computacionais baseada na utilização de testes automáticos de software. Tal abordagem se diferencia de outras abordagens existentes para o diagnóstico automático de faltas por:

- não requerer uma modelagem formal do sistema [McKeag e Macnaghten 1980; Jones 1990; Ortmeier e Reif 2004], o que seria impraticável para um sistema tão complexo, dinâmico e heterogêneo como uma grade computacional;
- não requerer que o usuário tenha acesso ao código fonte dos serviços da grade [DeMillo e Mathur 1995; Kuhn 1999];
- não requerer treinamento como o necessário para utilização de sistemas baseados em aprendizado de máquina [Chen et al. 2004; Podgurski et al. 2003; Bowring, Rehg e Harrold 2004; Mirgorodskiy, Maruyama e Miller 2006];
- não requerer qualquer modificação no código fonte da aplicação do usuário ou dos serviços disponíveis na grade;
- utilizar como motor de inferência o conjunto de testes automáticos do próprio software, um artefato que já é criado normalmente durante a fase de desenvolvimento.

1.5 Estrutura do Documento

O restante deste documento está organizado em seis capítulos. As seções seguintes apresentam um sumário de cada um deles.

Capítulo 2: Fundamentação Teórica

O Capítulo 2 contextualiza o problema tratado nesta tese, apresentando uma uniformização da nomenclatura utilizada em todo o trabalho e estabelece uma ligação entre gerência de

falhas em grades e testes de software.

Capítulo 3: Trabalhos Relacionados

O Capítulo 3 apresenta trabalhos relacionados à gerência de falhas em grades computacionais, particularmente trabalhos voltados à detecção e tolerância de falhas e ao diagnóstico de faltas.

Capítulo 4: Detecção de Falhas Baseada em Testes Automáticos de Software

O Capítulo 4 detalha a solução para a detecção de falhas em aplicações e serviços de grade desenvolvida neste trabalho. Também é apresentada uma ferramenta desenvolvida para monitorar os serviços de uma grade computacional de grande porte, que utiliza testes automáticos de software para detectar falhas nos serviços da grade.

Capítulo 5: Diagnóstico de Faltas Baseado em Testes Automáticos de Software

O Capítulo 5 apresenta a estratégia para o diagnóstico de faltas em aplicações e serviços de grade proposta neste trabalho, apresentando um *framework* para o diagnóstico de faltas utilizando testes automáticos de software e uma implementação de referência do *framework* capaz de diagnosticar faltas em grades criadas utilizando um determinado *middleware*.

Capítulo 6: Avaliação Experimental

Este capítulo apresenta detalhes sobre o experimento realizado com a ferramenta de diagnóstico detalhada no Capítulo 5 em uma grade computacional real e apresenta uma análise dos resultados obtidos com o uso da ferramenta para diagnosticar faltas em aplicações submetidas para execução nesta grade.

Capítulo 7: Conclusões e Trabalhos Futuros

Por fim, o Capítulo 7, encerra o trabalho com uma análise dos resultados obtidos e com uma discussão sobre direções para possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

No capítulo anterior apresentamos evidências de que grades computacionais ainda estão longe de serem consideradas sistemas confiáveis. A gerência de falhas em grades computacionais, uma área chave para o sucesso do modelo de computação em grade, ainda precisa receber bastante atenção para que tal modelo possa ser realizado de forma satisfatória. As altas taxas de falha observadas em grades em produção no momento aliadas à insatisfação dos usuários com os mecanismos de gerência de falhas em utilização por essas grades demonstram claramente que são necessárias melhores soluções para lidar com tais problemas.

Neste capítulo contextualizamos o problema tratado neste trabalho e discutimos alguns aspectos relacionados ao teste automático de software relevantes para o restante do trabalho.

2.1 Taxonomia

Segundo Avizienis *et al*, uma **falha** é um evento que ocorre quando o serviço fornecido por um sistema não corresponde à função que o sistema deveria desempenhar. Um serviço oferecido por um sistema é considerado falho quando ele não corresponde à especificação funcional do sistema ou quando a especificação não descreve corretamente a função do sistema. Uma vez que um serviço corresponde a uma seqüência de estados externos de um sistema, uma falha no serviço significa que pelo menos um dos estados externos do sistema se desviou de seu estado considerado correto. Tal desvio é chamado de **erro**. A causa determinada ou hipotética para um erro é chamada **falta** [Avizienis et al. 2004].

Ainda segundo Avizienis *et al*, as estratégias que podem ser adotadas para se aumentar

a confiabilidade em um sistema, ou seja, diminuir a possibilidade de que falhas venham a ocorrer, podem ser agrupadas em quatro categorias [Avizienis et al. 2004]:

- Prevenção de Faltas: prevenir a ocorrência ou introdução de faltas;
- Tolerância a Faltas: evitar que a ocorrências de faltas impliquem em falhas no sistema;
- Remoção de Faltas: reduzir o número ou severidade das faltas;
- Predição de Faltas: estimar o número existente de faltas, a possibilidade de ocorrências dessas faltas e seus potenciais efeitos.

O ciclo de vida de um sistema pode ser dividido em duas etapas: desenvolvimento e uso. A fase de desenvolvimento inclui todas as atividades realizadas entre a coleta dos requisitos junto ao usuário até a conclusão de que o sistema passou em todos os testes de aceitação e está pronto para ser usado em seu ambiente de produção. Durante o processo de desenvolvimento o sistema interage com seu ambiente de desenvolvimento e faltas de desenvolvimento podem ser introduzidas no sistema. O ambiente de desenvolvimento é formado pelos seguintes elementos:

- o mundo físico e seus fenômenos naturais;
- desenvolvedores humanos;
- ferramentas de desenvolvimento: o software e hardware utilizado pelos desenvolvedores;
- ambiente de testes, utilizado para checar o funcionamento do sistema em desenvolvimento.

A fase de uso do sistema se inicia quando o sistema é considerado pronto para ser usado e passa a oferecer serviços a seus usuários. Durante a fase de uso, o sistema interage com seu ambiente de produção e pode ser negativamente afetado por faltas originadas por outros elementos presentes no ambiente. O ambiente de produção consiste nos seguintes elementos:

- o mundo físico e seus fenômenos naturais;
- administradores (humanos ou outros sistemas), que têm autoridade para modificar e reparar o sistema;

- usuários (humanos ou outros sistemas) que consomem o serviço fornecido pelo sistema;
- provedores (humanos ou outros sistemas) que fornecem serviços para o sistema;
- a infraestrutura que fornece o suporte à execução do sistema.

A taxonomia de faltas proposta em [Avizienis et al. 2004] agrupa mais de 30 tipos de faltas em 3 conjuntos não-disjuntos:

- Faltas de Desenvolvimento: todas as faltas originadas na etapa de desenvolvimento;
- Faltas Físicas: todas as faltas que afetam o hardware;
- Faltas por Interação: todas as faltas originadas da interação do sistema com componentes externos.

Do ponto de vista do desenvolvedor de software, faltas físicas devem ser tratadas através da utilização de mecanismos de tolerância a faltas, uma vez que suas ocorrências independem da forma como o software foi desenvolvido. Por outro lado, faltas de desenvolvimento e faltas por interação estão profundamente ligadas à forma como o software foi concebido e, principalmente, à forma como o software foi testado.

Software para computação em grade apresenta um nível de complexidade a mais em relação a outros software para computação distribuída. Isso se deve a características relacionadas à heterogeneidade e ampla distribuição do ambiente para o qual foi projetado. Tal tipo de software, mesmo quando exaustivamente testado no ambiente de desenvolvimento, pode vir a falhar por uma série de fatores ligados ao seu ambiente de produção. Não é sem razão que os resultados do nosso *survey* [Duarte et al. 2006], resumidos no Capítulo 1, indicaram que a maioria das causas de falhas em aplicações e serviços de grade estão relacionadas ao ambiente de execução, ou seja, são em sua maioria causadas por Faltas por Interação.

2.2 Testes de Software

Testes são uma parte fundamental do processo de desenvolvimento de qualquer software. Exemplos de desastres causados por software mal testado não são raridades na literatura [Ben-Ari 1999; Mellor 1994].

Existem duas estratégias principais para se testar um software: testes de caixa preta e testes de caixa branca [Myers et al. 2004].

Testes de caixa branca, também chamados de testes estruturais, são utilizados para avaliar o comportamento interno do software. Por conta disso, para serem criados, testes de caixa branca exigem bom conhecimento sobre o funcionamento interno do software. Tais testes são utilizados para checar o funcionamento interno do componente de software sendo testado, geralmente envolvendo aspectos como testes de condições lógicas, testes de caminhos lógicos e checagem de trechos de código nunca executados.

Um exemplo prático desta técnica é o uso da ferramenta de testes JUnit [Gamma e Beck 1999], utilizada para criar e executar testes automáticos de unidade [Jeffries, Anderson e Hendrickson 2000] para programas escritos em linguagem Java. Testes de unidade são criados para testar individualmente pequenos blocos de código, por exemplo, métodos de uma classe em um programa Java, exigindo, portanto, conhecimento sobre detalhes do funcionamento e implementação do código para selecionar casos de testes apropriados. Testes de caixa branca são geralmente utilizados para testar funcionalidades individuais (testes de unidade) e funcionalidades obtidas com a cooperação de diferentes módulos do sistema (testes de integração) [Myers et al. 2004].

Testes de caixa preta, também conhecidos como testes funcionais, por sua vez, são utilizados para avaliar o comportamento externo do sistema, ignorando por completo qualquer detalhe específico de sua implementação [Myers et al. 2004]. São conhecidos também como testes de entrada e saída uma vez que seu funcionamento é baseado no fornecimento de dados de entrada para os quais a saída já é conhecida e comparação dessa saída conhecida com a saída produzida pelo programa sendo testado. Testes de caixa preta podem ser utilizados em todas as fases de teste, desde os testes de unidade e integração, que testam partes do sistema de forma isolada ou não, até testes de sistema e testes de aceitação, que visam testar o sistema como um todo, reproduzindo o mais fielmente possível a experiência que o usuário terá ao utilizar o software [Myers et al. 2004].

A metodologia de desenvolvimento *Extreme Programming* [Jeffries, Anderson e Hendrickson 2000] prega que os testes devem ser criados antes mesmo do próprio código a ser testado, consistindo em um excelente exemplo de testes de caixa preta uma vez que não requer sequer que o código do sistema exista e muito menos que exista qualquer conhecimento

sobre o mesmo para que os testes sejam escritos.

Existem também os chamados testes de caixa cinza, uma abordagem que mescla características dos testes de caixa branca e de caixa preta ao desenvolver os testes no nível do desenvolvedor, com total acesso aos detalhes do código fonte da aplicação mas conduzi-los no nível do usuário, sem qualquer contato com as partes internas da aplicação.

2.2.1 Fases do Processo de Testes

O processo de testes pode ser dividido em várias fases e estas fases podem ser agrupadas por seu posicionamento em uma determinada metodologia de desenvolvimento de software ou pelo nível de funcionalidade e granularidade do código testado.

A Figura 2.1 ilustra o relacionamento entre funcionalidades testadas e granularidade do código para diferentes fases do processo de testes descritas nas seções a seguir.

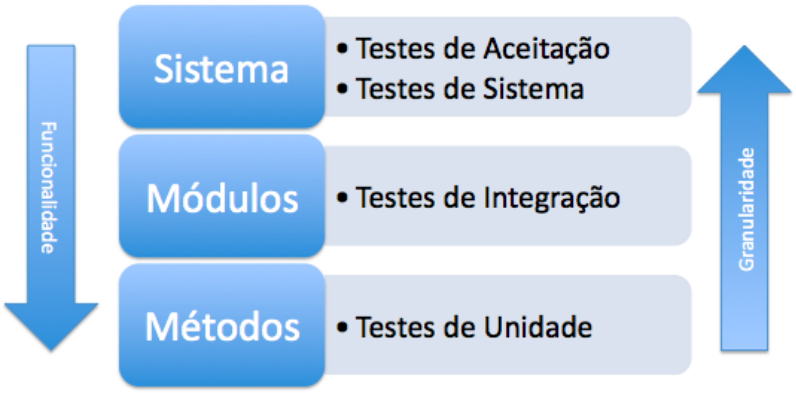


Figura 2.1: Funcionalidades testadas e granularidade dos testes

Testes de Unidade

Testes de unidade se referem a testes que checam funcionalidades específicas de um determinado seguimento de código, geralmente no nível de funções ou métodos, no caso de linguagens orientadas a objetos [Binder 2000]. Este tipo de teste é geralmente escrito pelos desenvolvedores enquanto escrevem o código da aplicação (testes de caixa branca) ou antes do código da aplicação ser escrito [Jeffries, Anderson e Hendrickson 2000] (testes de caixa preta). Testes de unidade isoladamente não podem checar a funcionalidade de um pedaço de

software. Eles são utilizados para garantir que cada bloco fundamental do programa funciona corretamente quando isolados dos demais blocos.

Testes de Integração

Testes de integração são testes que visam checar a interação entre diferentes componentes presentes em um software. Pode ser feito de forma iterativa, tão logo os componentes vão sendo integrados, ou de única vez quando todos os componentes estiverem juntos. Normalmente a abordagem iterativa é considerada mais eficiente uma vez que permite detectar problemas de integração mais cedo no processo de desenvolvimento do software. Testes de integração são utilizados para expor defeitos na interface de comunicação entre diferentes módulos de um sistema [Binder 2000].

Testes de Sistema

Testes de sistema são testes realizados após a integração de todos os módulos do programa e são realizados para checar se o sistema atende aos requisitos funcionais [Geraci 1991] levantados durante a fase de análise do sistema. São testes de caixa preta portanto não exigem qualquer conhecimento prévio sobre o funcionamento interno do sistema.

Testes de Aceitação

Testes de aceitação são testes de caixa preta executados no sistema como um todo antes de sua liberação para uso por seus usuários finais [Black 2002]. Em geral, testes de aceitação executados pelo desenvolvedor do sistema são diferentes de testes de aceitação realizados pelo usuário ou cliente. Os testes de aceitação executados pelo cliente muitas vezes são utilizados como um atestado de que o sistema atingiu os objetivos esperados.

Testes de Regressão

Testes de regressão são executados com o objetivo de detectar se uma alteração em uma determinada parte do código causou ou não efeitos colaterais em outras partes do sistema. Seu objetivo é garantir que defeitos que foram removidos no passado não regressem ao código. Um método comum para testes de regressão envolve executar todos os testes de unidade e

integração do sistema sempre que uma nova funcionalidade é adicionada e sempre que um defeito é removido.

2.2.2 Teste Automático

Um conjunto de testes (*test suite*) que captura o comportamento esperado para um software pode ser utilizado para checar se uma determinada implementação funciona de acordo com o esperado. Adicionalmente, uma vez corrigidos os possíveis defeitos detectados pelo conjunto de testes, esse mesmo conjunto pode ser utilizado também para checar se as modificações realizadas não tiveram efeitos colaterais em alguma outra parte do código.

Conjuntos de testes são ainda mais benéficos no desenvolvimento de software de grande porte, o que geralmente envolve vários times de desenvolvedores. Nessas situações, um conjunto de testes pode ser utilizado por um time de desenvolvimento para checar se um módulo do software fornecido por uma outra equipe funciona como esperado sem que se precise investigar detalhes da implementação do módulo [Jeffries, Anderson e Hendrickson 2000].

Para garantir sua efetividade, o processo de testes deve ser realizado de forma automatizada, pois cada conjunto de testes pode ser executado dezenas ou centenas de vezes durante a fase de desenvolvimento do software, por exemplo, toda vez que uma nova funcionalidade é adicionada ou algum defeito é corrigido [Myers et al. 2004; Jeffries, Anderson e Hendrickson 2000].

Middleware para computação em grades são artefatos de software particularmente complexos e que precisam ser testados exaustivamente durante a fase de desenvolvimento para aumentar as chances de que eles venham a funcionar corretamente uma vez iniciada suas fases de uso [Meglio 2007]. Por conta disso, testes automáticos de software vêm sendo utilizados para melhorar a qualidade dos componentes de *middleware* para computação em grades [Dantas, Cirne e Saikoski 2006; Bégin et al. 2007] e para tentar detectar, ainda durante a fase de desenvolvimento, falhas por interação, que de outra forma só seriam detectadas depois de iniciada a fase de uso [Duarte et al. 2006; Duarte et al. 2005; Duarte et al. 2006; Meglio et al. 2008].

2.3 Considerações Finais do Capítulo

Neste capítulo apresentamos a taxonomia adotada no restante do trabalho, definindo os conceitos de falta, falha e erro e das diferentes fases do ciclo de vida de uma aplicação [Avizienis et al. 2004].

Testes são um dos principais mecanismos utilizados na Engenharia de Software para localizar defeitos durante a fase de desenvolvimento. Testes automáticos são ainda mais eficientes nesta tarefa ao permitir que um conjunto de testes possa ser executado repetidas vezes de forma rápida e eficiente, ajudando a diminuir a probabilidade de falhas na aplicação, sem eliminar, contudo, a chance de que o software venha a falhar quando estiver em sua fase de uso [Jeffries, Anderson e Hendrickson 2000]. Testes automáticos de software são importantes também para o desenvolvimento de *middleware* para computação em grade ao possibilitar a detecção de problemas de concorrência causados pela execução de múltiplos *threads* [Dantas, Cirne e Saikoski 2006] ou para expor o quanto antes possíveis falhas de interação, causadas por problemas de configuração do ambiente [Duarte et al. 2006; Duarte et al. 2005; Duarte et al. 2006; Meglio et al. 2008], o tipo mais comum de falha segundo os participantes do nosso *survey*.

Capítulo 3

Trabalhos Relacionados

O capítulo anterior contextualizou o problema abordado, apresentando a taxonomia adotada no trabalho e fazendo a ligação entre a gerência de falhas e testes automáticos de software.

Neste capítulo apresentamos uma revisão de literatura sobre gerência de falhas em grades apontando soluções existentes para detecção e tolerância a falhas e para o diagnósticos de faltas.

3.1 Gerência de Falhas em Grades Computacionais

Na última década, sistemas distribuídos têm evoluído rapidamente, de simples aplicações cliente/servidor executando em uma rede local para infraestruturas de grades computacionais cobrindo vários países localizados muitas vezes em continentes distintos e envolvendo uma vasta gama de arquiteturas de hardware e software, executado uma diversidade de serviços. Apesar desse aumento de popularidade e de escopo, projetar, desenvolver e operar tais sistemas continuam sendo tarefas das mais desafiadoras.

A gerência de falhas em grades computacionais se dedica a fornecer ferramentas para auxiliar na operação desse tipo de infraestrutura, provendo, entre outros, mecanismos para detectar falhas no funcionamento e violações de propriedades desejadas para os serviços existentes, tolerar a ocorrência de falhas durante a execução das tarefas de seus usuários e diagnosticar, no caso de uma falta implicar em uma falha na execução da aplicação do usuário, a causa raiz do problema. Além disso, evidências forenses devem ser mantidas para permitir uma possível auditoria na infraestrutura para identificar mau uso ou falhas de

segurança.

Existem duas técnicas principais para lidar com falhas em grades computacionais. A primeira envolve tolerar a ocorrência das falhas e permitir que o sistema continue a funcionar mesmo que com uma possível degradação de seu desempenho. As soluções mais frequentes para tolerar falhas em grades envolvem redundância temporal (re-submissão) ou redundância espacial (replicação, re-escalonamento e *checkpointing*). A segunda técnica se dedica a fornecer ao usuário o maior nível possível de detalhe sobre a ocorrência de falhas para possibilitar o diagnóstico da causa raiz do problema [Zhou 2010].

As duas estratégias se baseiam na existência de detectores de falhas [Gupta, Chandra e Goldszmidt 2001] capazes de sinalizar com maior ou menor grau de confiança a ocorrência de uma falha na execução de uma tarefa do usuário ou em algum dos serviços da grade utilizados na execução da tarefa.

Nas seções a seguir apresentamos trabalhos que abordam diversos aspectos da gerência de falhas em grades computacionais que são relacionados aos objetivos propostos neste trabalho.

3.1.1 Detecção de Falhas

Tomando por base trabalhos anteriores voltados para a detecção de falhas em sistemas distribuídos [Bianchini e Buskens 1991; Gupta, Chandra e Goldszmidt 2001], pesquisadores têm investigado métodos e modelos para detectar falhas em infraestruturas de grade computacional.

Vários sistemas de monitoramento para este tipo de infraestrutura foram propostos nos últimos anos uma vez que os métodos disponíveis para detecção de falhas em outros tipos de sistemas distribuídos não são considerados adequados para uso em grades. Uma das razões para isso é que a maioria das soluções existentes se baseia no conhecimento detalhado sobre a infraestrutura de rede utilizada pelo sistema [Case et al. 1993]. Este tipo de informação geralmente não está disponível em uma infraestrutura como uma grade computacional, amplamente distribuída, altamente dinâmica e envolvendo múltiplos domínios administrativos e de segurança.

A *GMA (Grid Monitoring Architecture)* [Tierney et al. 2002] é um padrão aberto para o monitoramento de falhas em grades proposto por um dos grupos de trabalho do *Open Grid*

Forum [The Open Grid Forum 2009]. Sua arquitetura, ilustrada na Figura 3.1, é formada por quatro tipos de componentes, chamados *Directory Service*, *Producer*, *Consumer* e *Intermediary*.

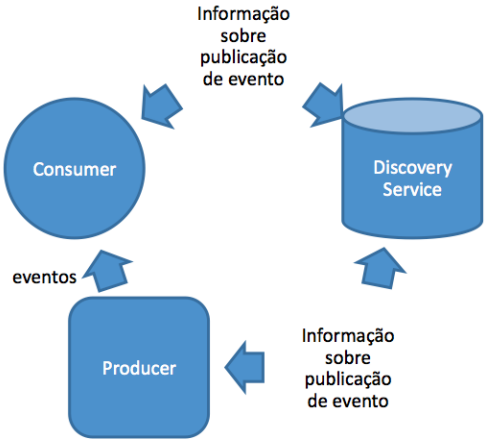


Figura 3.1: *GMA: Grid Monitoring Architecture*
[Tierney et al. 2002]

O *Directory Service* permite a publicação de informação e o descobrimento de novos serviços. *Producers* são entidades que disponibilizam informação de gerência através da publicação de eventos. *Consumers* recebem eventos contendo informação de gerência e os processam apropriadamente. *Intermediaries* são ao mesmo tempo *Producers* e *Consumers* e fornecem serviços especializados de transformação de informação. Por exemplo, um *Intermediary* pode consumir eventos de vários *Producers* e produzir novos eventos derivados dos eventos recebidos. Um serviço chamado *ReGS* [Aridor et al. 2003] especifica dois tipos de *Intermediaries* para o monitoramento de aplicações: um *Intermediary* para filtrar eventos e outro para armazená-los. O serviço de detecção de falha, *FDS* [Hwang e Kesselman 2003] oferece um mecanismo de notificação capaz de prover informações detalhadas sobre o estado de tarefas submetidas a uma grade. A flexibilidade do *GMA* fornece uma plataforma interessante para a construção de soluções para a detecção de falhas e para o monitoramento de serviços de grade, porém não oferece componentes específicos para facilitar a localização do componente onde a falta que causou uma determinada falha foi originada.

O *MapCenter* [Bonnassieux, Harakaly e Primet 2002], desenvolvido no contexto do projeto *DataGrid* [CERN 2010] é uma ferramenta de monitoramento e visualização da dispo-

nibilidade e distribuição de serviços em uma grade computacional, funcionando como uma ferramenta administrativa para localizar problemas de disponibilidade de serviços. O *MapCenter* constrói e atualiza periodicamente um modelo de rede dos serviços disponíveis na grade e oferece várias opções diferentes para visualizar a informação através de uma interface *web*. É importante ressaltar que o *MapCenter* só apresenta informações referentes à disponibilidade dos serviços, sem considerar detalhes de configuração e/ou utilização dos recursos. Portanto, utilizando o *MapCenter* o usuário de uma grade computacional pode descobrir quais serviços estão disponíveis na grade mas não há como ele descobrir se os serviços disponíveis estão configurados corretamente para executar tarefas de uma determinada Organização Virtual [Foster e Kesselman 1997].

O *GridICE* [Andreozzi et al. 2005] foi criado no contexto do projeto *DataTag* [CERN 2001] com o objetivo de facilitar o trabalho dos administradores de infraestruturas de grade computacional. Ele apresenta informações sobre o estado e a utilização em diferentes níveis de granularidade, como Organização Virtual, sítio ou recurso, além de mostrar também estatísticas básicas derivadas de análises históricas e de alertas recebidos em tempo real. Oferece também uma interface web, ilustrada na Figura 3.2.

O *GridICE* tem uma arquitetura centralizada em que um servidor principal consulta periodicamente um conjunto de máquinas para extrair informações sobre o estado da grade computacional, da rede e da utilização dos recursos. Esse servidor principal é baseado no *Nagios* [Imamagic e Dobrenic 2007], uma ferramenta de código aberto, para o monitoramento de máquinas e serviços de rede que pode ser facilmente estendida com o uso de *plugins* para monitorar serviços específicos. As informações coletadas são armazenadas em uma base de dados e utilizadas para criar estatísticas, alertas e para configurar dinamicamente o *Nagios* para monitorar qualquer novo serviço recém descoberto na grade. O *GridICE* lembra mais uma ferramenta de monitoramento industrial do que um sistema de monitoramento de grades. Apesar de prover uma visão dos dados de monitoramento, o seu público alvo são os administradores dos provedores de recursos da grade, mais interessados em encontrar e resolver problemas em seus próprios recursos, e não os usuários, que querem saber porque a execução da sua aplicação foi interrompida abruptamente.

O *Monitoring and Discovery Service* [Fitzgerald et al. 1997], previamente conhecido como *Metacomputing Directory Service* constitui o sistema de informação do *Globus tool-*

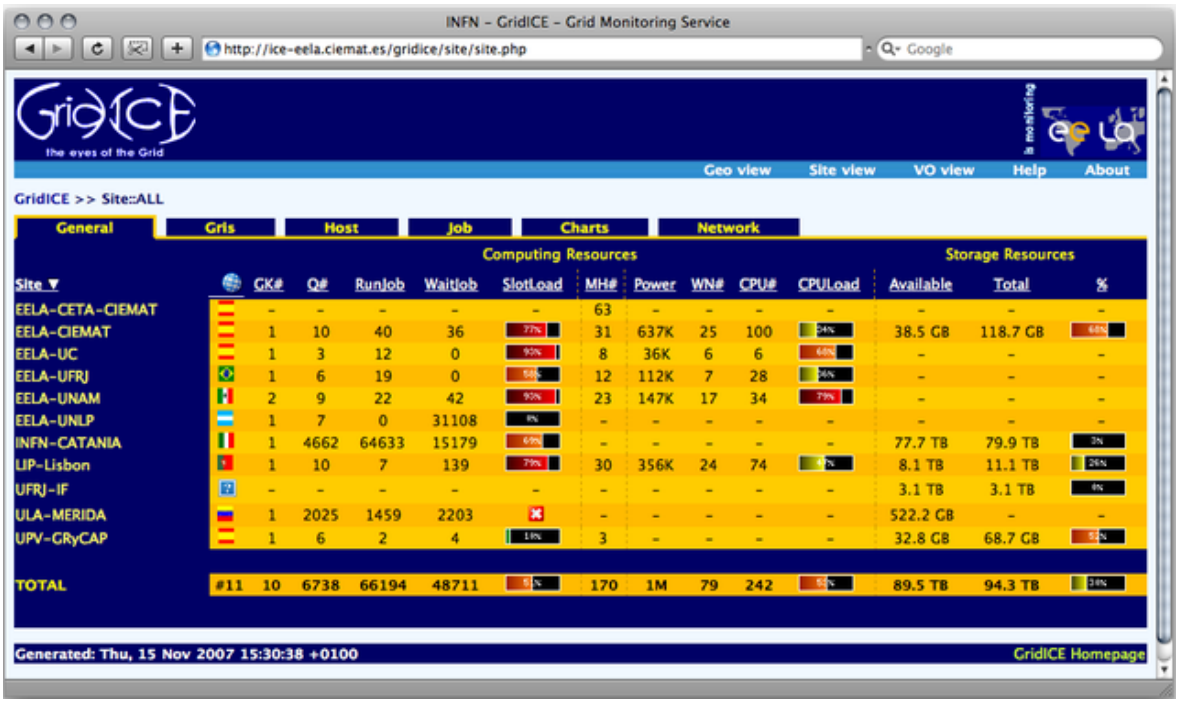


Figura 3.2: Interface web do *GridICE*

[Andreozzi et al. 2005]

kit [Foster e Kesselman 1997]. O MDS é baseado em dois protocolos principais: o *Grid Information Protocol* (GRIP) e o *Grid Registration Protocol* (GRRP). O primeiro descreve como consultar, responder e buscar por informações na grade enquanto que o segundo possibilita que o estado seja mantido e compartilhado entre diferentes componentes utilizando o *Lightweight Directory Access Protocol* (LDAP) [Wahl, Howes e Kille 1997] como modelo de dados e como protocolo de transporte. O MDS possui dois tipos de componentes principais: provedores de informação *Grid Resource Information Services* (GRIS) ou produtores e os *Grid Index Information Services* (GIIS) ou re-publicadores. Tanto os produtores quanto os re-publicadores são implementados em cima de uma versão de código aberto de um servidor LDAP.

A Figura 3.3 ilustra uma hierarquia de produtores (GRIS) e republicadores (GIISs) para uma grade computacional que possui duas regiões, com dois sítios da grade em cada região e com cada sítio fornecendo dois tipos de serviços de grade. Nessa hierarquia, os produtores coletam informações dos serviços monitorados e repassam essas informações utilizando o GRIP para os re-publicadores no nível local, responsáveis pelo sítio, para que elas sejam propagadas para os componentes consumidores interessados na informação. Os republicadores formam uma hierarquia em que cada nível agrega as informações fornecidas pelos re-publicadores nos níveis inferiores (ou por produtores no caso do primeiro nível de re-publicadores). No entanto, apesar deste tipo de solução para o monitoramento de grades ser bastante flexível e extensível, ela não pode ser utilizada para checar a cadeia completa de interação entre os diferentes serviços necessários para executar uma aplicação do usuário.

O *GStat* [The GStat Team 2010], desenvolvido no contexto do projeto EGEE [Gagliardi 2005], é uma ferramenta projetada para monitorar um dos principais serviços de uma grade computacional: seu sistema de informação central. A característica mais interessante do *GStat* é sua capacidade de gerar gráficos, parcialmente exibidos na Figura 3.4, fornecendo informações importantes sobre diferentes níveis de alerta e várias outras métricas relacionadas aos serviços existentes em uma grade. Tais gráficos permitem, por exemplo, analisar a estabilidade de um sítio ao apresentar de forma concisa a variação apresentada para os recursos disponibilizados. As informações coletadas dos GIISs, localizadas em cada sítio da grade, descrevem todos os recursos existentes. A quantidade de recursos disponíveis varia com o tempo devido à natureza dinâmica de uma grade, mais especificamente por conta

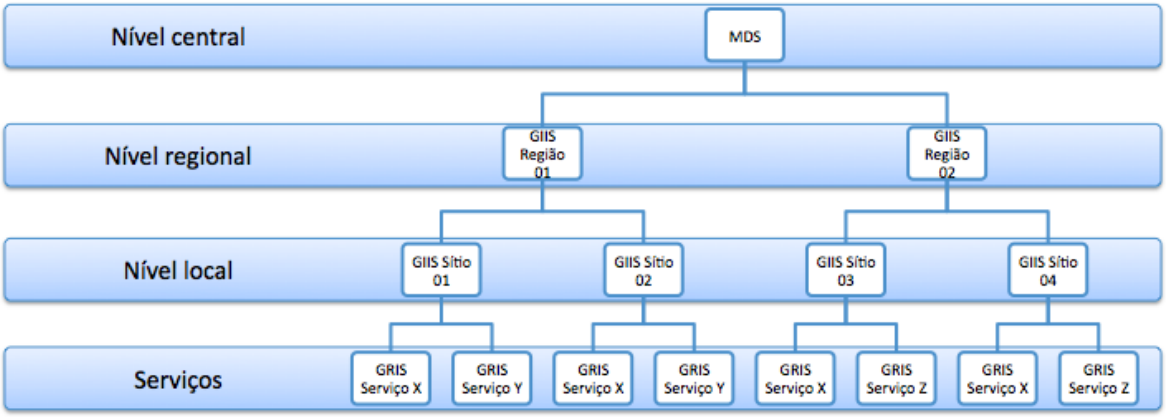


Figura 3.3: Exemplo de hierarquia de GIISs e GRIIs

de mudanças na configuração ou devido a alterações nos ambientes de execução disponibilizados para os usuários das várias organizações virtuais suportadas por um determinado centro de recursos. Isso significa que a quantidade de recursos em um centro de recursos pode flutuar e mesmo assim seu sistema de informação (GIIS) ser considerado como correto e atualizado. Porém, em algumas situações a quantidade de recursos cai abruptamente para 0 ou para um valor muito próximo disso, talvez por algum problema na rede ou uma falha no próprio GIIS. O *GStat* é uma ferramenta utilizada atualmente por várias infraestruturas de grade computacional mas sua funcionalidade precisa ser complementada por outras ferramentas uma vez que ele é uma ferramenta especializada para um determinado tipo de serviço.

3.1.2 Tolerância a Falhas

De forma geral, soluções para o monitoramento de serviços de grade se concentram apenas em coletar informações oriundas dos componentes da grade. Apesar de serem bastantes úteis para detecção de falhas, tais serviços não são capazes de ajudar na superação da barreira cognitiva associada ao diagnóstico de faltas em sistemas altamente complexos como as grades computacionais.

Assim como em qualquer outro tipo de sistema distribuído, tolerância a falhas em grades computacionais é baseada em redundância. Existem duas formas de redundância a serem consideradas. Redundância temporal envolve tentar re-executar repetidas vezes uma tarefa

Figura 3.4: Interface web do *GStat*

[The GStat Team 2010]

que falhou em uma tentativa de execução anterior. Já a redundância espacial é utilizada para explorar a existência de múltiplas instâncias de um mesmo serviço na grade. Essas duas formas de redundância são amplamente utilizadas para tolerar falhas em infraestruturas de grade. No entanto, uma vez que grades computacionais, por definição, fornecem múltiplas instâncias de todos os seus serviços, soluções explorando redundância espacial têm recebido mais atenção.

Existem três técnicas mais comuns baseadas em redundância espacial: *i*) replicação, *ii*) re-escalonamento e *iii*) *checkpointing*.

Soluções utilizando replicação de tarefas têm sido utilizadas para trocar ciclos de CPU por informação. Ao invés de submeter uma única tarefa para a grade e aguardar até que a informação sobre sua falha seja detectada de alguma forma, soluções de tolerância a falhas por replicação de tarefas submetem um número limitado de réplicas de uma mesma tarefa e aguardam até que uma das réplicas conclua sua execução corretamente, encerrando neste momento a execução das demais réplicas da tarefa.

Litke *et al* utilizaram a função de confiabilidade de Weibull [Xie e Lai 1996], que é dada

pelo complemento de sua distribuição acumulada de probabilidades, para estimar a quantidade mínima de réplicas necessárias para garantir um determinado grau de tolerância a falhas na execução de aplicações de grade [Litke et al. 2007]. *GALLOP* [Weissman 1998] e *WQR* [Cirne et al. 2007] são exemplos de escalonadores de tarefas para grades computacionais que oferecem tolerância a falhas utilizando replicação e re-escalonamento. O *GALLOP* replica a execução de tarefas do tipo SPMD (*simple-program-multiple-data*) em vários pontos da grade, dentro da mesma organização virtual, enquanto que o *WQR* é um escalonador de tarefas para aplicações do tipo *Bag-of-tasks* [Cirne et al. 2006] que re-escalona automaticamente tarefas com execução sem sucesso. O *Legion* [Grimshaw et al. 1999] e o *Condor* [Litzkow, Livny e Mutka 1988] toleram a ocorrência da falhas utilizando mecanismos de *checkpointing*, onde estados corretos da execução da aplicação são salvos gradativamente para que, no caso da ocorrência de uma falha, possam ser recuperados para que a execução continue do último ponto salvo. No caso do *Legion*, este mecanismo é fornecido no nível das aplicações que são executadas na grade enquanto que no *Condor* o *checkpointing* é uma das funcionalidades oferecidas pelo *middleware*.

Todas as ferramentas e soluções descritas na seção 3.1.1 são utilizadas para detectar a existência de uma falha na execução de uma aplicação na grade. Soluções como o *Grid-WFS* [Hwang e Kesselman 2003] e o *Phoenix* [Kola, Kosar e Livny 2004] fazem mais que isso ao reagir ativamente à detecção de uma falha para permitir que a execução da aplicação do usuário possa continuar utilizando recursos e/ou serviços localizados em outra parte da grade.

Grid-WFS provê tolerância a falhas através de um esquema flexível de recuperação. Construído sobre um sistema de detecção de falhas, o *Grid-WFS* utiliza uma técnica para mascarar a ocorrência de falhas através de novas tentativas de execução. Diferentemente do que ocorre com ferramentas de replicação, o *Grid-WFS* só re-submete uma tarefa caso a tentativa anterior de execução não tenha sucesso.

O *Phoenix*, por outro lado, é uma solução capaz de detectar e classificar falhas em aplicações de grade que fazem uso intenso de transferência e armazenamento de dados. Ele utiliza uma estratégia probabílistica para detectar falhas na transferência de arquivos. Uma vez que uma falha é detectada, um componente chamado *Failure Agent* é utilizado para classificar a falta que afeta a transferência em transiente ou permanente. Quando a falta é considerada

transiente o sistema pode tentar realizar a transferência novamente. Apesar da classificação de faltas oferecida pelo *Phoenix* ser útil para impedir que novas tentativas de transferência sejam realizadas na ocorrência de faltas permanentes, ele não oferece qualquer suporte para auxiliar na correção ou diagnóstico das causas para este tipo de falta.

Checkpointing também tem sido utilizado para permitir que o poder computacional de dispositivos móveis, cuja dinamicidade e heterogeneidade é ainda maior do que de recursos convencionalmente encontrados em grades computacionais, possa ser explorado para executar tarefas de uma grade [Chu e Humphrey 2004]. Para que isso ocorra sem que haja demasiado desperdício de recursos, é preciso salvar o estado das tarefas periodicamente e movê-las para outras máquinas ou dispositivos móveis sempre o que dispositivo em que a tarefa estava executando sai da área de cobertura da grade ou começa a executar uma tarefa que exige todos os recursos do dispositivo.

3.1.3 Diagnóstico de Faltas

Mecanismos para tolerância a falha são, sem dúvida, muito úteis e estão se tornando mais populares, como apontaram as respostas do nosso *survey*. No entanto, todas as soluções discutidas anteriormente se baseiam em esconder do usuário a ocorrência de falhas. Diagnosticar a causa das falhas não é um requisito para que essas ferramentas funcionem corretamente. Porém, caso a falta original seja permanente, tais mecanismos nem sempre conseguem manter o usuário afastado do processo, uma vez que será preciso diagnosticar a causa do problema para que ele seja sanado corretamente.

Mesmo sendo apontado pelos usuários que responderam o nosso *survey* como o maior problema no momento em que eles precisam se recuperar de uma falha na execução de uma aplicação na grade, o diagnóstico de faltas em grades computacionais é um tópico de pesquisa que têm recebido menor atenção do que as demais áreas da gerência de falhas. Um extenso *survey* [Dabrowski 2009] sobre gerência de falhas em grades publicado em 2009 listou apenas dois trabalhos [Qian-Mu, Man-wu e Hong 2006; Duarte et al. 2006] tratando do diagnóstico de faltas em grades, sendo um deles de nossa autoria e refente ao trabalho apresentado neste documento.

Abordagens clássicas para o diagnóstico de faltas se baseiam desde especificações formais do sistema [McKeag e Macnaghten 1980; Jones 1990; Ortmeier e Reif 2004] até a ge-

ração automática de casos de teste a partir do código fonte da aplicação [DeMillo e Mathur 1995; Kuhn 1999]. Tais abordagens não são as mais apropriadas para a utilização em grades computacionais uma vez que especificar formalmente um sistema tão amplamente heterogêneo e largamente distribuído seria tão ou até mais difícil do que diagnosticar manualmente suas faltas, além disso, na maioria dos casos não se tem acesso ao código fonte dos sistemas por trás dos serviços oferecidos na grade.

O diagnóstico de faltas em grades computacionais ainda é uma tarefa realizada de forma manual e *ad hoc*, consumindo muito do tempo útil de seus utilizadores, porém existem alguns esforços no sentido de automatizar, pelo menos parcialmente, tal processo.

[Smallen et al. 2004] propõem a utilização de um *framework* baseado em testes automáticos de software para a verificação de acordos de interoperabilidade entre os diferentes sistemas em interação em uma grade.

Sistemas baseados no aprendizado de máquina têm sido aplicados com sucesso a vários problemas de classificação [Chen et al. 2004; Podgurski et al. 2003], por exemplo, para identificar o comportamento de um sistema baseado em dados sobre sua execução [Bowring, Rehg e Harrold 2004] ou para diagnosticar anomalias em um conjunto de processos através do monitoramento de chamadas de procedimentos [Mirgorodskiy, Maruyama e Miller 2006].

Qian-mu, Man-wu e Hong [Qian-Mu, Man-wu e Hong 2006] descrevem uma abordagem para detectar a falta que originou uma falha baseada em princípios da imunologia, muito utilizados no campo da biônica [Rosen 1963]. O sistema proposto trata eventos de monitoramento dos serviços da grade como peptídeos e utiliza o princípio imunológico da seleção positiva para correlacioná-los. Em seguida, os comportamentos mais frequentes, determinados pela quantidade de eventos relacionados, são analisados para apontar a provável origem de uma falha.

Por fim, técnicas de correlação de eventos, como o raciocínio baseado em modelos [Davis e Hamscher 1988], têm sido utilizadas tanto para o diagnóstico de faltas em aplicações de grades computacionais [Hofer e Fahringer 2008] quanto para o diagnóstico de faltas em redes de transmissão e distribuição de energia elétrica [Sauvé et al. 2005; Duarte 2003], outro exemplo de sistema altamente heterogêneo e largamente distribuído.

No entanto, nenhuma das soluções propostas anteriormente foi até o momento adotada como uma solução geral e genérica para o diagnóstico de faltas nas grades computacionais

em produção atualmente e todas requerem, em maior ou menor grau, alterações nos serviços disponibilizados na grade e/ou nas aplicações dos usuários, aumentando assim a resistência frente a sua adoção, sendo potencialmente esta a razão para o diagnóstico de faltas ser o principal problema encontrado pelos participantes do nosso *survey* na hora em que precisam se recuperar de uma falha na execução de uma aplicação na grade [Duarte et al. 2006; Duarte 2009].

3.2 Considerações Finais do Capítulo

Apresentamos neste capítulo trabalhos relacionados na área de gerência de falhas em grades, focando soluções para detecção e tolerância de falhas e diagnóstico de faltas. Em particular, constatamos que existe uma lacuna em relação ao diagnóstico de faltas em grades, apesar (ou talvez justamente por isso) desse ser apontado pelos participantes do nosso *survey* como o maior problema para se recuperar de uma falha na execução de uma aplicação na grade [Duarte et al. 2006; Duarte 2009]. Uma outra evidência dessa lacuna é o fato de um extenso estudo [Dabrowski 2009] sobre mecanismos para tratamento de falhas em grades computacionais, publicado em 2009, ter listado apenas dois trabalhos [Qian-Mu, Man-wu e Hong 2006; Duarte et al. 2006] voltados para o diagnóstico de faltas em grades, sendo um deles de nossa autoria e relativo ao trabalho descrito neste documento.

A hipótese que pretendemos verificar com este trabalho é que os mesmos testes de aceitação utilizados durante a fase de desenvolvimento para aferir quando o software está pronto ou não para ir para o ambiente de produção podem ser utilizados também para detectar falhas em serviços de grade e servir de base para um serviço de localização e diagnóstico de faltas.

Uma vez que testes automáticos de software já são produzidos durante a fase de desenvolvimento dos componentes presentes em um *middleware* para computação em grades, como discutido no capítulo anterior, pretendemos analisar a possibilidade de se utilizar esses mesmos testes para detectar falhas e diagnosticar faltas durante a fase de uso.

Capítulo 4

Detecção de Falhas Baseada em Testes Automáticos de Software

Antes de poder diagnosticar as causas de uma falha nos serviços de uma grade computacional é necessário detectar a ocorrência dessa falha. Com este objetivo, participamos, durante um estágio realizado no CERN [CERN 2010], da equipe de desenvolvimento de uma solução para a detecção de falhas em grades computacionais baseada no uso de testes automáticos de software.

Tal trabalho foi financiado pelo projeto WLCG/EGEE [Gagliardi 2005] e por essa razão seu desenvolvimento foi baseado no leque de serviços ofertados pelo *middleware* utilizado na grade computacional mantida pelo projeto: o *gLite*.

Neste capítulo apresentamos um sumário executivo dos serviços disponibilizados pelo *middleware gLite* e em seguida detalhamos nossa solução para detecção de falhas em grades computacionais através do uso de testes automáticos de software. Tal solução, batizada *Service Availability Monitoring* ou simplesmente SAM, é hoje a principal solução para detecção de falhas na maioria das grades criadas com o *gLite*.

4.1 O *Middleware gLite*

O *gLite* é um *middleware* para computação em grades criado por uma colaboração envolvendo cerca de 80 pessoas em 12 diferentes centros de pesquisa acadêmica e industrial no contexto do projeto WLCG/EGEE [Gagliardi 2005].

O *gLite* é hoje utilizado por diversas grades computacionais espalhadas por boa parte do globo [Gagliardi 2005; The European Grid Initiative 2010; SEEGrid 2010; EELA 2010; BalticGrid 2010; NGS 2010; int.eu.grid 2010; D-Grid 2010; EuChinaGrid 2010; EuAsiaGrid 2010; The EUIndiaGrid Project 2010; ItalianGrid 2010]. Além disso, tem sido tratado por muitos pesquisadores como um elo de ligação entre grades computacionais criadas com diferentes tipos de *middleware*, como atestam os vários trabalhos sobre interoperabilidade encontrados na literatura [Field e Schulz 2008; Brasileiro et al. 2008; Asvija et al. 2009; Wang, Cheng e Chen 2009; Nakada et al. 2008; Gronager et al. 2008; Riedel et al. 2008; Wang et al. 2007; Kacsuk, Farkas e Fedak 2008; Garzoglio et al. 2009]

O *gLite* oferece um *framework* para criação de aplicações distribuídas, possibilitando que tais aplicações explorem poder de processamento e capacidade de armazenamento localizados em qualquer parte de uma grade computacional. Sua primeira versão, *gLite* 1.0, foi liberada em Abril de 2005. Após a versão 1.5, liberada em Janeiro de 2006, o *gLite* foi combinado com um outro *middleware* para computação em grades chamado LCG [Lamanna 2004], usado para processar os dados coletados pelo Grande Colisor de Hadrons (*Large Hadron Collider* ou LHC) do CERN. Em Maio de 2006 foi liberada a versão 3.0 do *gLite*, combinando todos os serviços da versão 2.7 do LCG com vários componentes do *gLite* 1.5. A partir de então os dois *middleware* passaram a ser considerados como um só, com o nome de *gLite* e que se tornou o *middleware* padrão para várias infraestruturas de computação em grades espalhadas pelo mundo, incluindo a grade utilizada pelos projetos EGEE e WLCG, a colaboração mundial criada em torno do LHC [Gagliardi 2005; Campana et al. 2008].

Nas seções a seguir apresentamos os principais serviços presentes em uma grade *gLite*, ilustrados na Figura 4.1 [D-Grid 2010], e que foram utilizados no experimento descrito no Capítulo 6.

4.1.1 Autenticação e Autorização

Os usuários de uma grade computacional criada utilizando o *gLite* são agrupados em Organizações Virtuais [Foster, Kesselman e Tuecke 2001]. Para serem autenticados e autorizados a utilizar os recursos disponíveis na grade, os usuários precisam primeiro ingressar em uma organização virtual.

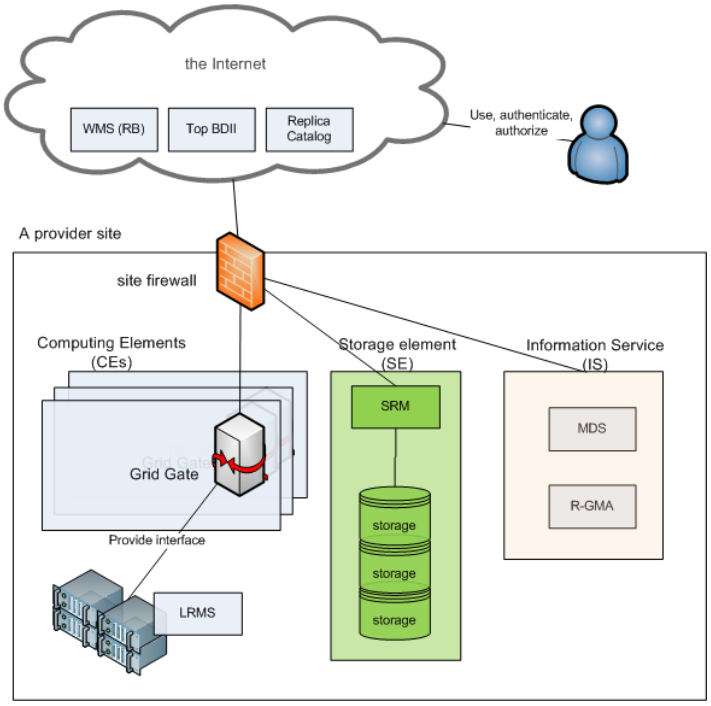


Figura 4.1: Arquitetura dos principais serviços do *middleware gLite*

A Infraestrutura de Segurança de Grade (*Grid Security Infrastructure* ou GSI) [Foster e Kesselman 1997] de grades *gLite* provê autenticação através de uma rede aberta como a Internet, sendo baseada em um sistema de criptografia de chave pública que usa o padrão $X.509$ [ITU-T 2000] e na utilização do protocolo de comunicação *Secure Sockets Layer* (SSL), adicionado de extensões para delegação de identidade.

Cada usuário de uma grade *gLite* deve possuir um certificado digital $X.509$ emitido por uma Autoridade Certificadora reconhecida [IGTF 2010] pelas instituições que fornecem os recursos computacionais da grade.

A autorização de um usuário para utilizar um determinado serviço da grade é feita utilizando o *Virtual Organization Membership Service*(VOMS) [Cecchini et al. 2003], que autentica o usuário checando a validade de seu certificado junto à Autoridade Certificadora que o emitiu e autoriza o uso do recurso checando se o usuário é realmente membro da organização virtual da qual ele atesta ser.

4.1.2 Interface do Usuário

O acesso a uma grade *gLite* é feito através de uma máquina configurada com um conjunto de aplicações cliente para os diversos serviços da grade. Ao instalar seu certificado digital nesta máquina o usuário pode se autenticar e receber a autorização para utilizar os serviços e recursos disponíveis na grade. As funcionalidades básicas da grade, listadas a seguir, são oferecidas através de uma interface de linha de comando:

- listar todos os recursos capazes de executar uma determinada tarefa;
- submeter uma tarefa para execução;
- cancelar a execução de uma tarefa;
- recuperar os arquivos de saída de uma determinada tarefa;
- consultar o estado de uma tarefa ainda não concluída;
- copiar arquivos da máquina do usuário para a grade;
- copiar arquivos da grade para a máquina do usuário;
- remover arquivos da grade;
- replicar arquivos na grade;
- consultar o sistema de informação da grade para saber o estado dos diferentes serviços.

4.1.3 *Computing Element*

Computing Element (CE) é a terminologia utilizada para definir um conjunto de recursos computacionais, um *cluster*, por exemplo, localizado em um provedor de recursos para uma grade *gLite*.

Um *Computing Element* inclui uma interface padrão para o acesso remoto aos recursos locais *Grid Gate*, um gerenciador local de recursos (*Local Resource Management System* - LRMS), e os recursos propriamente ditos, que aparecem como uma coleção de nós (*Worker Nodes*) onde as tarefas são executadas.

O *gLite* oferece duas implementações diferentes da interface para acesso remoto aos recursos (*gLiteCE* e textitLGCCE), um sintoma remanescente da integração com *middleware* do projeto LCG descrita anteriormente, e os provedores de recursos da grade podem escolher livremente qual das interfaces instalar, o que acaba causando uma certa confusão para seus usuários, obrigando que saibam utilizar os aplicativos cliente para as duas interfaces.

O *Grid Gate* é o responsável por receber as tarefas submetidas pelos usuários da grade e escaloná-las para execução nos recursos computacionais locais através de um dos LRMS suportados (OpenPBS/PBSPro [PBS Works], LSF [Zhou et al. 1992], Maui/Torque [Staples 2006; Cluster Resources 2010] e Condor [Litzkow, Livny e Mutka 1988]).

Um trabalho recente [Brasileiro et al. 2008] permitiu estender a funcionalidade do *Computing Element* para que ele possa repassar tarefas recebidas da grade para recursos gerenciados pelo *middleware OurGrid*, possibilitando a disponibilização de recursos não-dedicados em uma grade *gLite*.

4.1.4 *Storage Element*

O *Storage Element* (SE) fornece acesso uniforme a recursos de armazenamento de dados. Um SE pode controlar desde servidores de disco até dispositivos de armazenamento de massa utilizando fitas magnéticas. Todo provedor de recursos em uma grade *gLite* oferece pelo menos um *Storage Element* para gerenciar seus recursos de armazenamento.

Assim como os recursos de processamento são gerenciados localmente por um LRMS abaixo do *Computing Element*, a maioria dos recursos de armazenamento são gerenciados localmente por um *Storage Resource Manager* (SRM) [J., Bakken e Petravick 2004], um serviço capaz de prover migração transparente de arquivos de fita magnética para disco, reserva de espaço e outras funcionalidades importantes para este tipo de serviço.

Da mesma forma que para o LRMS, existe uma variedade de implementações de SRM suportadas pelo *gLite*. O *Disk Pool Manager* (DPM) [CERN 2009] é utilizado geralmente por provedores de recursos menores, onde o espaço de armazenamento geralmente é todo composto por discos magnéticos, enquanto que o CASTOR [Baud et al. 2003] fornece acesso transparente a grandes dispositivos de armazenamento em fitas magnéticas. *dCache* [Fuhrmann 2004] é uma implementação intermediária que pode ser utilizada pelos dois tipos de provedores de recursos.

4.1.5 Serviço de Informação

O Serviço de Informação (*Information Service* - IS) fornece detalhes sobre os recursos existentes em uma grade *gLite* e sobre seu estado. Este serviço é muito importante para o funcionamento da grade como um todo pois é através dele que são descobertos novos recursos adicionados à grade. A informação publicada é também utilizada para monitoração dos serviços da grade e para fazer contabilidade referente à utilização dos recursos e é descrita segundo um esquema chamado GLUE (*Grid Laboratory Uniform Environment*) [Foster e Kesselman 1997]. GLUE define um modelo de dados conceitual para ser utilizado no monitoramento e descoberta de serviços.

O Serviço de Informação do *gLite* é baseado em conceitos do *Globus* MDS [Fitzgerald et al. 1997]. No entanto, o GRIS e GIIS do MDS foram substituídos pelo *Berkeley Database Information Index* ou simplesmente BDII, que é basicamente um servidor LDAP [Wahl, Howes e Kille 1997] cujas informações podem ser atualizadas por um processo externo.

4.1.6 *Workload Management System*

O objetivo do *Workload Management System* (WMS) é selecionar o *Computing Element* mais apropriado para executar uma tarefa do usuário.

Os usuários de uma grade *gLite* escrevem suas tarefas utilizando uma linguagem de descrição de tarefa (*Job Description Language* ou JDL), que especifica, por exemplo, que programa executar e quais parâmetros utilizar, arquivos para serem movidos da máquina do usuário para o *Computing Element* onde a tarefa será executada, arquivos de entrada necessários para executar a tarefa e qualquer requisito necessário para que um *Computing Element* seja capaz de executar a tarefa.

O WMS escolhe para qual *Computing Element* enviar a tarefa do usuário através de um processo chamado *match-making*, que seleciona um *Computing Element* dentre todos os disponíveis que seja capaz de atender a todos os requisitos especificados pelo usuário em seu arquivo de descrição de tarefa. A tarefa então é repassada para o *Computing Element* escolhido que trata do escalonamento local para um de seus *Worker Nodes* através de um LRMS.

4.2 *Service Availability Monitoring*

O *Service Availability Monitoring* ou SAM é uma ferramenta de detecção de falhas em grades computacionais que utiliza testes automáticos de aceitação para checar o funcionamento dos diversos serviços existentes em uma grade computacional [Duarte et al. 2007; Duarte et al. 2008]. O SAM foi criado para substituir o *Site Functional Tests* (SFT) [Neocleous et al. 2007], a ferramenta utilizada até então para monitorar a grade computacional do projeto WLCG/EGEE [Gagliardi 2005], considerada muito restrita e com um modelo de dados muito complexo e de difícil manutenção.

As principais funcionalidades do SAM são:

- componentes de software chamados sensores capazes de executar os testes automáticos de aceitação de um determinado serviço da grade;
- modelo de dados simples e de fácil manutenção para armazenamento dos resultados dos testes;
- integração com outras ferramentas operacionais utilizadas no projeto WLCG/EGEE (FCR [CERN 2010], *CIC Portal* [IN2P3 2010] e *GridView* [Guangbao, Jie e Bo 2004]);
- cálculo automático de métricas de disponibilidade para os serviços da grade.

Nossa participação no desenvolvimento do SAM se restringiu à concepção e implementação dos sensores e do *framework* de submissão, utilizados para executar os testes automáticos de aceitação dos serviços da grade. As outras funcionalidades do sistema foram implementadas pelos demais membros da equipe de desenvolvimento no CERN.

4.2.1 Arquitetura

A arquitetura do sistema é exibida na Figura 4.2. O sistema tem um número considerável de componentes especializados que podem ser agrupados em três camadas independentes: entrada, armazenamento/processamento e saída (ou apresentação).

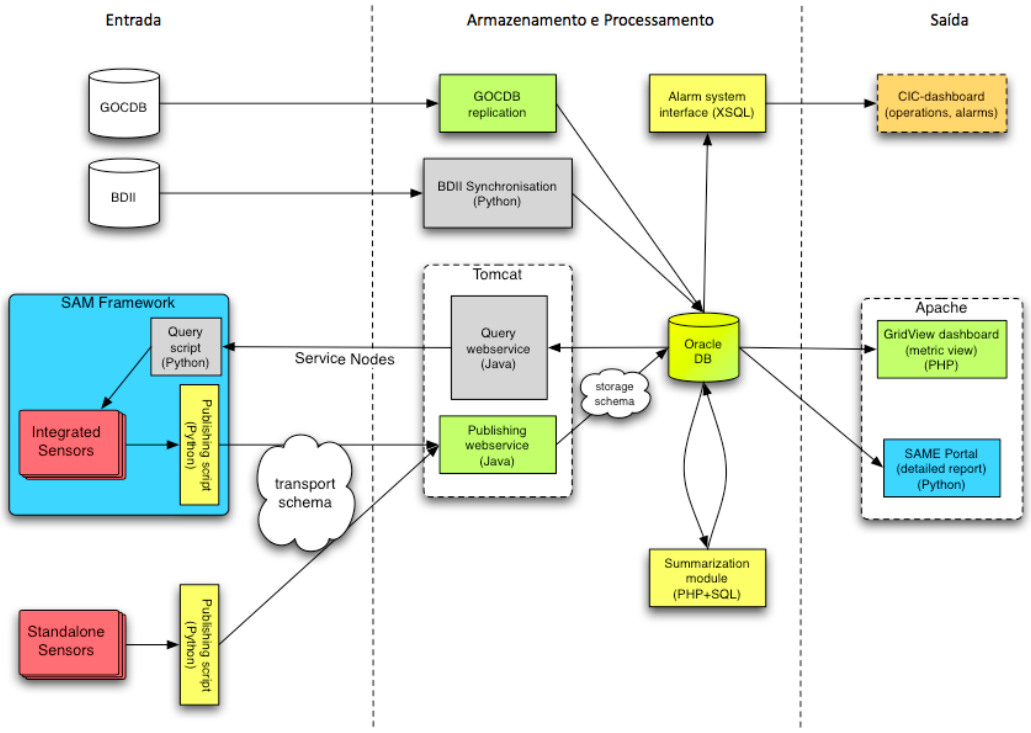


Figura 4.2: Arquitetura do SAM

Camada de Entrada

A camada de entrada engloba os componentes responsáveis por executar os testes de aceitação dos serviços da grade e por publicar seus resultados. Os testes tanto podem ser executados utilizando o *framework* de submissão de testes do SAM (*SAM Submission Framework*) quanto por aplicações independentes.

O *framework* de submissão do SAM é um pacote que oferece uma plataforma uniforme para a execução dos testes e publicação dos resultados na base de dados do sistema. A comunicação entre o *framework* de submissão e a base de dados do sistema é feita através de um *web service* acessado por um *script* escrito em *Python* e que oferece duas operações simples:

- consultar a base de dados para obter informações sobre a infraestrutura de grade (centros de recursos, nós, serviços, organizações virtuais);
- publicar os resultados dos testes utilizando um esquema de transporte de dados (*transport data schema*) bem simples.

Todos os sensores do SAM são *plugins* que se comunicam com o *framework* através de um protocolo bem simples. Dessa forma, adicionar novos sensores é extremamente fácil, bastando para isso escrever um programa que executa de alguma forma os testes automáticos de um determinado serviço de grade e que se comunica com o *framework* de acordo com o protocolo.

A arquitetura do *framework* de submissão do SAM pode ser vista na Figura 4.3. O processo de teste de um serviço da grade é realizado pelos seguintes passos:

1. a rotina de preparação para a execução dos testes do sensor é invocada uma única vez antes da execução dos testes;
2. o *framework* consulta a base de dados para descobrir quantas instâncias do serviço a ser testado existem e agenda a execução dos testes do sensor para cada uma delas;
3. o escalonador interno do *framework* solicita que o sensor teste uma determinada instância do serviço;
4. o sensor executa cada um dos testes automáticos disponíveis para o serviço;
5. o sensor publica os resultados dos testes na base de dados do SAM.

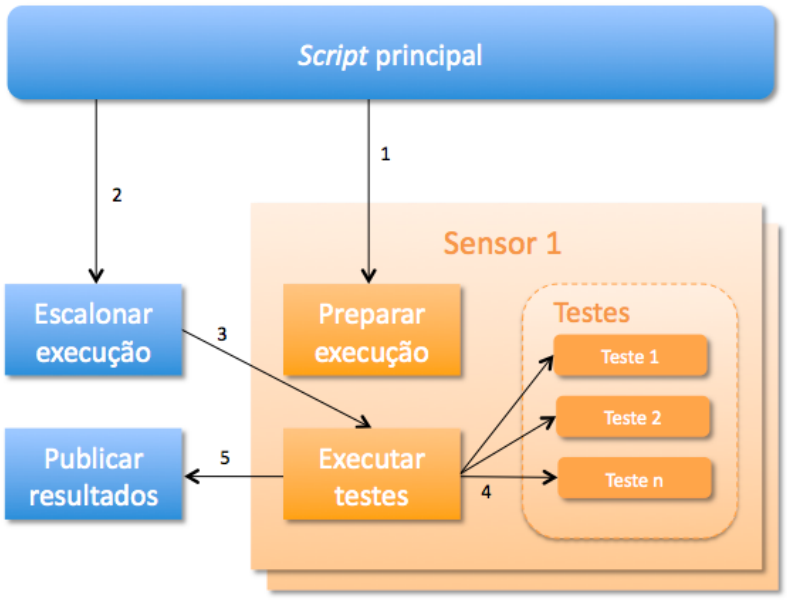


Figura 4.3: *Framework* de submissão do SAM

Camada de Armazenamento e Processamento

Os componentes desta camada do SAM são responsáveis pela coleta, armazenamento e pós-processamento dos resultados dos testes. O núcleo da camada é um sistema de banco de dados relacional utilizado para armazenar todas as informações sobre as entidades da grade computacional (provedores de recursos, nós, serviços e organizações virtuais), os relacionamentos entre estas entidades, os resultados dos testes e as métricas de disponibilidade calculadas com base nos resultados dos testes.

Os componentes que interagem diretamente com a base de dados do sistema são os seguintes:

- *web services* SOAP [Curbera et al. 2002] que fornecem serviços para consultar remotamente a base de dados em busca de informações sobre a infraestrutura (*Query web service*) e para publicar os resultados da execução dos testes (*Publishing web service*);
- o *script* de sincronização com o BDII da infraestrutura para possibilitar o descobrimento de novos serviços e novos provedores de recursos;
- o *script* de replicação da *GOC DB* [Cordier et al. 2006], utilizado apenas para a obtenção de dados sobre a infraestrutura do WLCG/EGEE;
- um módulo de sumário para definir o estado de um serviço ou de um provedor de recursos com base nos resultados dos testes;
- um sistema de geração de alarmes baseado em procedimentos e gatilhos da própria base de dados, utilizado para alimentar a interface de operação do EGEE/WLCG.

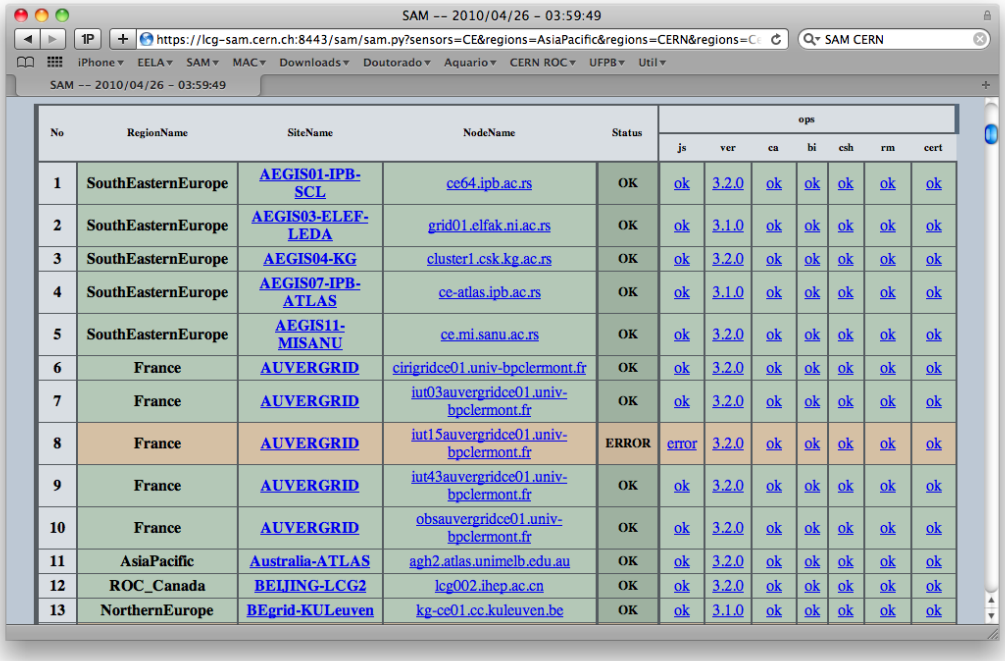
Camada de Saída

A camada de apresentação do SAM contem um série de componentes que acessam a base dados de forma direta ou indireta para fornecerem informações para outros serviços ou para os usuários da grade. Os componentes mais importantes são os seguintes:

- o *SAM Portal*, uma interface *web*, exibida na Figura 4.4, mostrando os resultados dos testes executados nos *Computing Elements* da grade do projeto WLCG/EGEE; esse componente foi escrito em *Python* e é utilizado para exibir os resultados dos testes

agrupados de diversas formas diferentes e com a possibilidade de exibir resultados históricos;

- *GridView*, uma ferramenta de visualização que é utilizada para exibir gráficos de disponibilidade de serviços baseados nos resultados dos testes executados com o SAM;
- *CIC Dashboard*, um portal utilizado na operação da grade do projeto WLCG/EGEE.



No	RegionName	SiteName	NodeName	Status	ops						
					js	ver	ca	bi	csh	rm	cert
1	SouthEasternEurope	AEGIS01-IPB-SCL	ce64.ipb.ac.rs	OK	ok	3.2.0	ok	ok	ok	ok	ok
2	SouthEasternEurope	AEGIS03-ELEF-LEDA	grid01.elfak.ni.ac.rs	OK	ok	3.1.0	ok	ok	ok	ok	ok
3	SouthEasternEurope	AEGIS04-KG	cluster1.csk.kg.ac.rs	OK	ok	3.2.0	ok	ok	ok	ok	ok
4	SouthEasternEurope	AEGIS07-IPB-ATLAS	ce-atlas.ipb.ac.rs	OK	ok	3.1.0	ok	ok	ok	ok	ok
5	SouthEasternEurope	AEGIS11-MISANU	ce.mi.sanu.ac.rs	OK	ok	3.2.0	ok	ok	ok	ok	ok
6	France	AUVERGRID	cirigridce01.univ-bpclermont.fr	OK	ok	3.2.0	ok	ok	ok	ok	ok
7	France	AUVERGRID	iut03auvergridce01.univ-bpclermont.fr	OK	ok	3.2.0	ok	ok	ok	ok	ok
8	France	AUVERGRID	iut15auvergridce01.univ-bpclermont.fr	ERROR	error	3.2.0	ok	ok	ok	ok	ok
9	France	AUVERGRID	iut43auvergridce01.univ-bpclermont.fr	OK	ok	3.2.0	ok	ok	ok	ok	ok
10	France	AUVERGRID	obsauvergridce01.univ-bpclermont.fr	OK	ok	3.2.0	ok	ok	ok	ok	ok
11	AsiaPacific	Australia-ATLAS	agh2.atlas.unimelb.edu.au	OK	ok	3.2.0	ok	ok	ok	ok	ok
12	ROC_Canada	BEIJING-LCG2	lcg002.ihep.ac.cn	OK	ok	3.2.0	ok	ok	ok	ok	ok
13	NorthernEurope	BEgrid-KULeuven	kg-ce01.cc.kuleuven.be	OK	ok	3.1.0	ok	ok	ok	ok	ok

Figura 4.4: Interface *web* do SAM

4.2.2 Disponibilidade

O SAM define o estado de um provedor de recursos e de seus serviços baseado nos resultados dos testes armazenados em sua base de dados. Além disso, ele calcula métricas de disponibilidade para todos os serviços monitorados.

O estado de cada serviço é obtido através de uma operação de *E* lógico sobre todos os resultados dos testes para o serviço. Dessa forma, cada serviço é inequivocamente definido como *disponível*, quando passou em todos os testes ou *indisponível*, quando falhou em pelo menos um dos testes.

Depois de calcular o estado de cada serviço, o SAM calcula também o estado geral do provedor de serviços, levando em consideração todos os serviços do provedor. O provedor de serviços é considerado *disponível* se pelo menos uma instância de cada um dos serviços disponibilizados pelo provedor passaram em todos os testes do SAM e *indisponível* caso contrário.

Todos os estados são então armazenados na base de dados e representam um retrato do estado da grade. Baseado nesses retratos, o módulo de sumário atualiza as métricas de disponibilidade para cada serviço de cada um dos provedores de recursos. A disponibilidade é definida então como o percentual dos retratos do estado da grade em que determinado serviço ou determinado provedor de recursos foi considerado *disponível* em um dado período de tempo (dia, semana, mês).

4.3 Considerações Finais do Capítulo

A necessidade de se ter uma ferramenta que oferecesse uma escalabilidade melhor para a detecção de falhas em uma grade computacional que vivenciava um crescimento muito rápido, com um repentino aumento do número de provedores de recursos, levou ao desenvolvimento de uma ferramenta de detecção de falhas chamada *Service Availability Monitoring* ou SAM.

Por ocasião de um estágio realizado no CERN no qual participamos da equipe de monitoramento e qualidade, propusemos que a nova ferramenta utilizasse os testes automáticos de aceitação do *middleware* adotado pela grade do projeto WLCG/EGEE, o *gLite*, para detectar falhas na grade.

Tal abordagem se mostrou apropriada e acabou colaborando para que o SAM passasse a ser a principal ferramenta de detecção de falhas em todas as infraestruturas de grade criadas com o *middleware gLite*.

O processo de desenvolvimento e validação do *SAM* resultou na apresentação de um trabalho no *GMW'07: Workshop on Grid Monitoring* [Duarte et al. 2007] e na publicação de um artigo no *Journal of Physics: Conference Series* [Duarte et al. 2008] a partir de um trabalho apresentado na conferência *Computing for High-Energy Physics*.

Porém, apesar de o SAM ter se mostrado uma ferramenta apropriada para detectar falhas em uma grade de grande porte como a do projeto WLCG/EGEE, ele não é capaz de apontar

corretamente as causas para as falhas detectadas nos serviços da grade.

O Apêndice A apresenta um exemplo de um resultado completo emitido pelo SAM no momento em que é detectada uma falha na execução do teste de submissão de tarefa em um *Computing Element*.

É possível observar que o resultado do teste apresenta, além de informações sobre a tarefa submetida e detalhes sobre as tentativas de execução da tarefa no *Computing Element*, um resumo sobre a possível causa da falha, exibido na Figura 4.5.

```
**************************************************************
BOOKKEEPING INFORMATION:
Status info for the Job : https://wms208.cern.ch:9000/k5OkrDH0-mHdlFSxqayEOA
Current Status:     Aborted
Logged Reason(s):
    - Job got an error while in the CondorG queue.
    - Job got an error while in the CondorG queue.
Status Reason:      hit job shallow retry count (1)
Destination:        iut15auvergridce01.univ-bpclermont.fr:2119/jobmanager-lcgpbs-ops
Submitted:          Mon Apr 26 04:07:59 2010 CEST
**************************************************************
```

Figura 4.5: Razão apontada pelo SAM como possível causa da falha na execução de uma tarefa

A razão apontada pelo SAM para a falha na execução do teste foi *Job got an error while in CondorG queue.* Este é um exemplo de mensagem de erro que o *gLite* emitiria para um usuário após tentar sem sucesso executar a sua tarefa, mas que não representa necessariamente a razão para sua aplicação ter falhado, pois essa falha pode ter sido causada por uma falta localizada em um outro componente do qual este depende.

A partir deste ponto, ao receber a mensagem de erro e perceber que a tarefa falhou, a pessoa responsável por corrigir o problema ou o usuário que apenas tentava executar sua tarefa terá que iniciar um processo de diagnóstico manual da causa da falha. Tal pessoa mergulhará nos detalhes de execução da tarefa e nos caminhos percorridos por todos os arquivos entre a máquina de onde a tarefa foi submetida até a máquina onde a tarefa seria executada para tentar desvendar o mistério por trás da falha na execução da tarefa. Um processo que não só consome muito tempo como também requer um nível de conhecimento muito maior do que a maioria dos usuários de grades tem ou gostaria de ter.

No próximo capítulo nós descrevemos nossa abordagem para o diagnóstico automáticos de faltas em grades que utiliza os resultados de testes automáticos de software como evidências para tentar localizar o real culpado pela falha na execução de uma aplicação na

grade.

Capítulo 5

Diagnóstico de Faltas Baseado em Testes Automáticos de Software

Os resultados apresentados no Capítulo 1, referentes ao *survey* com usuários de grades computacionais, indicaram que o maior problema para se recuperar de uma falha na execução de uma aplicação em uma grade, reportado por 70% dos usuários na última consulta realizada em 2009, é a dificuldade encontrada na hora de diagnosticar a falta que originou a falha.

O diagnóstico de faltas é um dos passos básicos em qualquer estratégia de gerenciamento de falhas. Hoje, é extremamente difícil para um usuário de grades descobrir porque a execução da sua tarefa na grade não foi bem sucedida. Nossa análise das taxas de falhas nas tarefas executadas na grade computacional do projeto WLCG/EGEE [Gagliardi 2005] em 2007 corroboram as respostas obtidas no *survey*. Verificamos taxas de erro de até 60% na execução das tarefas de uma determinada Organização Virtual e um número equivalentemente grande de falhas sem qualquer explicação para a sua causa.

Como discutido no Capítulo 3, várias soluções para o monitoramento de grades computacionais vêm sendo propostas ao longo dos anos [Aridor et al. 2003; Baker e Smith 2002; Smith 2001; Stelling et al. 1998; Tierney et al. 2000; Tierney et al. 2002; Waheed et al. 2000; Duarte et al. 2007; Duarte et al. 2008]. Certamente, tais ferramentas são úteis, uma vez que possibilitam a detecção de falhas e a coleta de informações referentes ao problema. Porém, tais soluções não apresentam mecanismos para diagnosticar a falta que originou o problema. Também foram propostos mecanismos para recuperação de falhas [Grimshaw et al. 1999; Hwang e Kesselman 2003; Kola, Kosar e Livny 2004; Litzkow, Livny e Mutka 1988;

Medeiros et al. 2003; Weissman 1998]. Tais mecanismos possibilitam a recuperação de falhas por omissão e por parada. Apesar de serem capazes de mascarar alguns tipos de falhas, tais mecanismos também não são capazes de diagnosticar as faltas e restaurar completamente o sistema. Por fim, foram apresentadas algumas abordagens para o diagnóstico de faltas em grades [Smallen et al. 2004; Chen et al. 2004; Podgurski et al. 2003; Bowring, Rehg e Harrold 2004; Hofer e Fahringer 2008; Mirgorodskiy, Maruyama e Miller 2006; Qian-Mu, Man-wu e Hong 2006] mas nenhuma delas conseguiu até o momento ser incluída como ferramenta padrão nas infraestruturas de grade em operação hoje por apresentarem requisitos que dificultam ou muitas vezes inviabilizam sua implantação.

Neste capítulo discutimos as lições aprendidas com nosso *survey* com os usuários de grades computacionais (Seção 5.1) e apresentamos nossa abordagem para o diagnóstico de faltas em grades computacionais (Seção 5.2). Nossa abordagem é baseada na utilização de testes automáticos de software para determinar se uma falha apresentada por um componente *C* foi originada por uma falta no próprio componente *C* ou em algum outro componente que fornece um serviço a *C*. Na Seção 5.3 nós introduzimos um *framework* criado para materializar a abordagem proposta e que foi substanciado com a implementação de uma ferramenta para o diagnóstico automático de faltas em grades criadas com um determinado *middleware*. Ao fim do capítulo (Seção 5.4) fazemos algumas considerações finais sobre a abordagem proposta e as ferramentas desenvolvidas.

5.1 A Barreira Cognitiva

A partir das respostas coletadas pelo nosso *survey* com usuários de grades computacionais [Duarte et al. 2006], resumidas no Capítulo 1, é possível inferir que, apesar de algum progresso ter sido feito, gerência de falhas continua sendo um dos principais problemas da computação em grades. Particularmente, o diagnóstico de faltas ainda é um problema que requer bastante atenção. Acreditamos que tal fato está profundamente relacionado com a forma como o software de grades é organizado: uma coleção de camadas, cada uma fornecendo um nível de abstração diferente. Os desenvolvedores de aplicações para grades utilizam as abstrações fornecidas pelo middleware para simplificar o desenvolvimento. De forma semelhante, os desenvolvedores do middleware utilizam as abstrações fornecidas pelo Sistema

Operacional para facilitar seu trabalho. Está é uma forma excelente de lidar com complexidade e heterogeneidade quando tudo funciona corretamente. Quando um componente de software falha, ele geralmente afeta os componentes que o usam. Tal propagação acaba chegando até o usuário, que percebe o problema. Para entender o que realmente aconteceu o usuário precisa mergulhar nas camadas de abstração até encontrar a falta que desencadeou todo o processo.

Em suma, o problema se resume a, quando tudo funciona, é preciso saber apenas o que um determinado componente de software faz, quando algo não funciona, é preciso saber como o componente funciona. A Figura 5.1 mostra um exemplo de uma arquitetura de grade com múltiplas camadas [Foster, Kesselman e Tuecke 2001]. A Figura mostra também o usuário e administradores do software em cada uma das camadas.

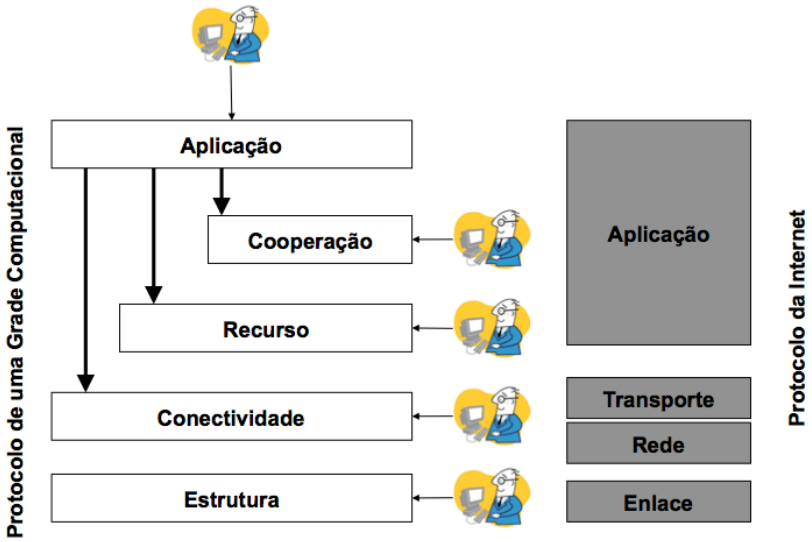


Figura 5.1: Arquitetura em Camadas de uma Grade Computacional [Foster, Kesselman e Tuecke 2001]

Imagine que em uma instância dessa arquitetura ocorreu uma falta na camada *Estrutura*. É possível que esta falta venha a causar uma falha em uma aplicação do usuário sendo executada na grade. Quando o usuário recebe uma mensagem de erro indicando que aconteceu uma falha durante a execução de sua aplicação, ele não sabe se a causa da falha está na sua própria aplicação ou em algum serviço da grade. É possível que o usuário tenha acesso ao histórico da execução do software nas diversas camadas para analisar e descobrir o que houve de errado. Inicialmente, ele verifica o histórico da sua própria aplicação para tentar

descobrir algo útil para identificar a causa do problema. Em seguida, depois de gastar algum tempo na primeira análise sem achar nada errado, o usuário começa a suspeitar que o problema pode estar na infra-estrutura. Neste momento o usuário está prestes a iniciar uma tarefa extremamente penosa. Ele precisa analisar todos os históricos de execução disponíveis para tentar diagnosticar uma falta em um sistema que é imensamente complexo e que é executado em um ambiente altamente heterogêneo. Se o usuário realmente conseguir ter acesso aos históricos de execução, pode chegar a ver mensagens de erro das camadas *Cooperação*, *Recurso*, *Conectividade* e *Estrutura*, ou até mesmo de camadas mais baixas do sistema operacional. No entanto, ele não tem como checar o software para saber em qual componente está o problema. Uma vez que a falta na camada *Estrutura* pode ter se propagado para camadas superiores, o usuário pode nunca vir a descobrir a falta que causou a falha em sua aplicação.

Apesar de não ser algo que aconteça exclusivamente com grades computacionais, a situação descrita no parágrafo anterior é muito mais grave neste tipo de sistema do que em plataformas tradicionais de computação distribuída. Isso se deve ao fato das grades serem muito mais complexas e heterogêneas e incluírem um número muito maior de tecnologias do que outras plataformas. Por exemplo, uma grade pode possuir recursos cuja arquitetura seja completamente desconhecida para o usuário. Portanto, o usuário não sabe nada sobre esses recursos. Não sabe como eles deveriam funcionar. Não sabe onde os históricos de execução são armazenados. Existe uma enorme barreira cognitiva entre a detecção da falha e o diagnóstico da falta.

Para superar esta barreira cognitiva, o diagnóstico de faltas deve ser conduzido de forma automática, com um mínimo de intervenção do usuário. Para que isso seja possível, o software precisa possuir o conhecimento necessário para diagnosticar suas próprias faltas. Infelizmente, o software disponível atualmente não está pronto para fornecer tal nível de informação [Maglio e Kandogan 2004]. Tudo que os sistemas oferecem no momento são os sintomas das falhas, sob a forma de mensagens de erro, fazendo com que o diagnóstico de faltas seja uma tarefa totalmente dependente do usuário. Portanto, uma solução para lidar com a complexidade envolvida no tratamento de falhas em grades é adicionar mecanismos para diagnóstico automático de faltas ao software disponível atualmente. Fazer isso não é uma tarefa fácil uma vez que existem importantes aspectos a serem considerados para for-

necer uma solução genérica para o diagnóstico de faltas em grades que possa ser adotada na prática por grades computacionais em produção, dentre os quais:

- **Intrusividade:** a solução não deve requerer modificações no código fonte do software existente, uma vez que tal código pode não estar disponível ou ser muito complexo para ser alterado.
- **Simplicidade:** Soluções complexas podem adicionar um outro grau de incerteza, complicando ainda mais o problema ao invés de resolvê-lo; uma forma de tornar uma solução mais confiável é garantir que ela seja tão simples que qualquer desenvolvedor possa entendê-la por completo.
- **Manutenabilidade:** Todo software possui um ciclo de vida, a solução para diagnóstico de faltas deve estar pronta para evoluir juntamente com o software para manter sua utilidade;
- **Flexibilidade:** para ser genérica, uma solução para diagnóstico de faltas precisa ser flexível o suficiente para poder ser utilizada por diferentes usuários e diferentes aplicações.

5.2 Diagnóstico Colaborativo de Faltas

Em um ambiente de produção, cada camada de software de uma grade computacional fornece um nível de abstração e possui uma equipe (desenvolvedores de aplicação, administrador de middleware ou pessoal de suporte de sistemas) responsável por, entre outras coisas, lidar com falhas. Portanto, se uma falha é detectada em uma camada superior mas sua origem é uma falta em uma camada de nível mais baixo, o pessoal responsável pela camada onde o defeito foi originado deve ser acionado para corrigir o problema. O desafio é justamente identificar o nível certo para a ação corretiva, permitindo que as partes envolvidas colaborem de forma controlada para resolver o problema. Nossa abordagem para solucionar este problema consiste em utilizar testes automáticos para checar a corretude do software em seu ambiente de produção e identificar a camada (componente ou serviço) onde a falta foi originada.

Mesmo que o software seja exaustivamente testado em seu ambiente de desenvolvimento antes de entrar em produção, é possível que defeitos não sejam detectados devido a vícios no ambiente utilizado para os testes. O conjunto de testes criado durante o desenvolvimento constitui um artefato valioso que pode ser reaproveitado para diagnosticar faltas uma vez que o software começa a ser executado em produção.

Ao executar o conjunto de testes de um determinado software em seu ambiente de produção usuários de uma determinada camada do software da grade podem descobrir se uma determinada falha foi causada por uma falta na sua própria camada ou em alguma das camadas abaixo dela sem precisar saber como qualquer uma das camadas funciona. Esta prática pode ajudar a revelar defeitos que não foram previamente detectados durante o desenvolvimento.

A funcionalidade de uma determinada camada é fornecida por um conjunto de componentes de software. Propomos a criação de componentes de diagnóstico, chamados *Doctors*, capazes de executar os testes automáticos dos componentes que implementam as funcionalidades apresentadas por cada uma das camadas de software de uma grade computacional. Para seguir esta estratégia, o desenvolvedor de um componente de software deve disponibilizar o conjunto de testes automáticos juntamente com o próprio componente [Duarte et al. 2006].

Os componentes de diagnóstico, *Doctors*, seguem uma interface comum que possibilita que os testes sejam executados de forma colaborativa, permitindo diagnosticar faltas em todo o sistema. Quando o *Doctor* de um determinado componente *C* é ativado, ele executa os testes automáticos do componente *C* e retorna um diagnóstico. Se todos os testes de *C* forem executados corretamente o *Doctor* de *C* infere que não há problemas em *C* e retorna tal informação. Se algum dos testes de *C* falha, o *Doctor* retorna um diagnóstico indicando a causa da falha apontada pelo teste falho. No entanto, neste caso, o *Doctor* precisa também descobrir se a falha em *C* aconteceu por conta de uma falta no próprio componente *C* ou em algum outro componente utilizado por *C*. Para fazer isso o *Doctor* de *C* deve checar também todos os componentes *C'* utilizados por *C*, ativando seus respectivos *Doctors*. Se pelo menos um desses *Doctors* retornar um diagnóstico para uma falta em um componente *C'* então o diagnóstico gerado pelo *Doctor* tratando de *C* incluirá tal informação. Portanto, quando um diagnóstico informa que apenas *C* está com problemas então uma falta em *C* é a provável

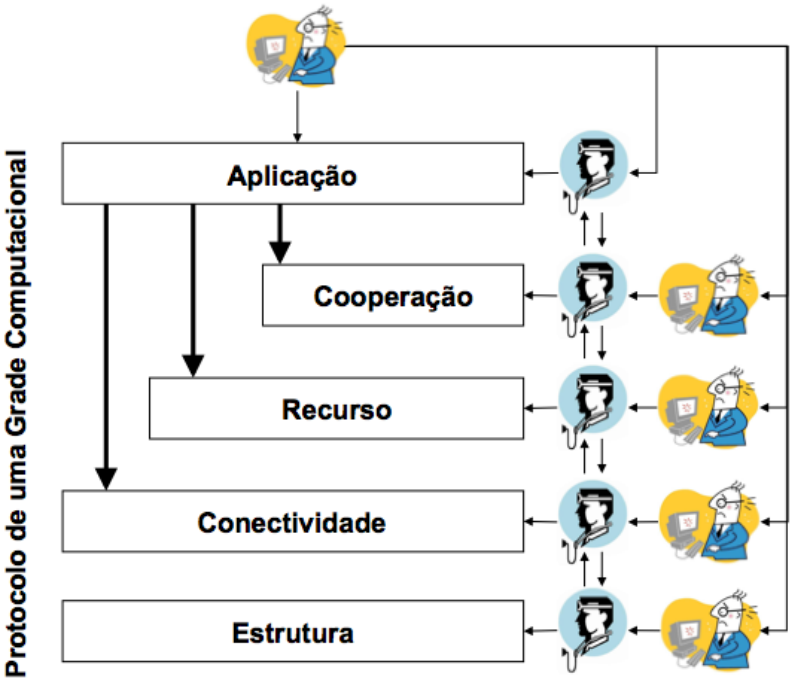


Figura 5.2: *Doctors* na Arquitetura em Camadas de uma Grade Computacional

causa da falha detectada.

A Figura 5.2 ilustra uma arquitetura em camadas de software para computação em grades e mostra o posicionamento de um *Doctor* para a aplicação do usuário, assim como *Doctors* para cada componente que implementa parte das funcionalidades de cada camada. Tão logo o usuário é notificado de um problema, ele pode acionar o *Doctor* associado a sua aplicação para que este aponte o componente onde a falha foi originada. Esta ação pode desencadear um efeito cascata, envolvendo os *Doctors* localizados nas camadas inferiores, até que algum deles retorne um diagnóstico para a falta que originou a falha observada pelo usuário.

A abordagem proposta não é intrusiva, uma vez que é baseada na utilização de componentes de software externos, os *Doctors*, não requerendo, portanto, qualquer modificação no código fonte do software utilizado. No entanto, como foi discutido anteriormente, uma ferramenta de diagnóstico deve ser simples, de fácil manutenção e bastante flexível. Nossa abordagem implica em ter um *Doctor* para cada componente de software. Isto faz com que o *Doctor* seja bastante simples, uma vez que ele é responsável por um único componente de software e não por todo o sistema. É possível argumentar que essa proliferação de componentes de diagnósticos pode agravar o problema de manutenção do software uma vez que

implicam em vários novos componentes de software para serem atualizados. É importante lembrar que testes são parte do software fornecido pelos desenvolvedores, e cada desenvolvedor precisa se preocupar apenas com os testes do seu próprio software. Portanto, a complexidade em manter o software atualizado é compartilhada por todos os desenvolvedores, fazendo com que o problema seja tratável mesmo que haja um número razoavelmente grande de componentes. Finalmente, a abordagem poderia fazer com o que os *Doctors* não fossem uma solução flexível, uma vez que cada um é responsável por diagnosticar faltas em um único componente de software. No entanto, a existência de uma interface bem conhecida entre os *Doctors* faz com que tais módulos possam cooperar para diagnosticar faltas envolvendo os mais diversos tipos de componentes de software.

5.3 O *Framework GridDoctor*

A Figura 5.3 ilustra a arquitetura proposta para o nosso *framework* de diagnóstico de faltas. Ao perceber que uma de suas tarefas não foi executada corretamente, o usuário envia ao *GridDoctor* o arquivo com o *log* da execução da tarefa, com todas as mensagens de erro recebidas ao fim da execução. Tal arquivo é utilizado para identificar os componentes envolvidos na tentativa de execução da tarefa do usuário, possibilitando que os *Doctors* responsáveis por cada um desses componentes sejam ativados e comecem a colaborar para descobrir a origem da falha na execução da aplicação do usuário. Cada um dos *Doctors* envolvidos no processo de diagnóstico é então utilizado para checar o estado de um determinado serviço da grade através da execução dos testes automáticos disponibilizados junto com o serviço.

O Código Fonte 5.1 mostra a principal classe do *framework*, que precisa ser estendida para que a solução possa ser utilizada para diagnosticar faltas em serviço de uma determinada grade. Esta classe depende da implementação de duas interfaces do *framework*: *Parser* e *GraphGenerator*, que são específicas para cada tipo de grade.

Código Fonte 5.1: Classe abstrata *GridDoctor.java*

```
1 /**
2  * Classe principal do framework GridDoctor
3  * Esta classe deve ser extendida para toda instanciação do framework uma
4  * vez que o parsing do log de execução da tarefa assim como a geração
5  * do grafo de dependências são dependentes do middleware.
6  * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
```

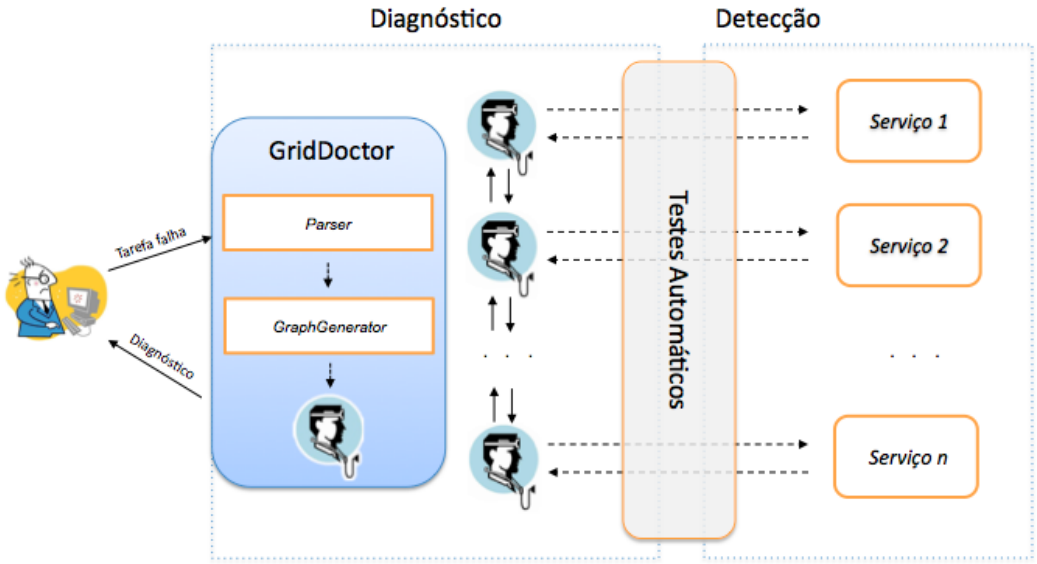


Figura 5.3: Arquitetura do *framework GridDoctor*

```
7   */
8  public abstract class GridDoctor {
9          /** Responsável por analisar o arquivo de log da tarefa.
10          */
11         private Parser parser;
12         /** Responsável por gerar o grafo de dependência dos doctors.
13          */
14         private GraphGenerator graph;
15         /**
16          * @param filename - path para o arquivo com o log da execução da tarefa.
17          */
18         public GridDoctor( String filename ) {
19                 parser = createParser(filename);
20                 graph = createGraphGenerator();
21         }
22         /**
23          * Método que inicia o diagnóstico.
24          * @return a possível causa para o problema.
25          * @throws ImpossibleToDiagnoseException
26          */
27         public Cause diagoseFault() throws ImpossibleToDiagnoseException {
28                 try {
29                         AppDoctor app = graph.generateGraph(parser.getServices());
30                         return app.getCause();
31                 } catch (Exception e) {
32                         throw new ImpossibleToDiagnoseException(e);
33                 }
34         }
```

```
35          /**
36           * @return um gerador de grafo de dependência específico para um determinado
37           * middleware e grade.
38           */
39          protected abstract GraphGenerator createGraphGenerator();
40          /**
41           * @return um parser de arquivo de log de tarefa específico para um determinado
42           * middleware e grade.
43           */
44          protected abstract Parser createParser(String filename);
45  }
```

Uma implementação do *Parser*, cuja interface é exibida no Código Fonte 5.2, é responsável por analisar o arquivo de *log* de erros da tentativa de execução da tarefa para identificar os serviços envolvidos. Tal análise é dependente do *middleware* utilizado na grade e, portanto, precisa ser fornecida por cada instanciação do *framework*.

Código Fonte 5.2: Interface *Parser.java*

```
1   import java.io.IOException;
2   import java.util.Set;
3
4   /**
5    * Interface que define o comportamento esperado de um Parser.
6    * Um Parser deve ser capaz de processar um arquivo contendo o log da
7    * tentativa de execução de uma tarefa do usuário na grade e extrair
8    * dele o conjunto de serviços de grade envolvidos.
9    * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
10   */
11  public interface Parser {
12
13          /**
14           * @return Um conjunto contendo todos os serviços grade envolvidos
15           * na tentativa de execução da tarefa do usuário de acordo com o
16           * arquivo de log da execução
17           * @throws FileNotFoundException Se o arquivo de log não puder ser
18           * acessado.
19           */
20          public Set<GridService> getServices() throws IOException;
21
22  }
```

Depois de definidos os serviços envolvidos na tentativa de execução, a classe *Parser* retorna uma estrutura de dados contendo todos os serviços da grade. Tal estrutura é utilizada por uma implementação da interface *GraphGenerator*, exibida no Código Fonte 5.3, para

Código Fonte 5.3: Interface *GraphGenerator.java*

```
1  import java.util.Set;
2  /**
3   * Interface que indica que uma classe é capaz de gerar um grafo de dependências para um
        conjunto de Doctors.
4   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
5   */
6  public interface GraphGenerator {
7          /**
8           * Geração de um grafo de dependências para um conjunto de Doctors
9           * @param services - Conjunto dos serviços envolvidos na tentativa de execução da
                aplicação
10          * @return - um grafo de dependência entre Doctors cujo nó inicial é um AppDoctor.
11          */
12         AppDoctor generateGraph(Set<GridService> services);
13 }
```

criar o grafo de dependência entres os *Doctors* do sistema. Neste grafo existe um nó para cada um dos *Doctors* e um arco ligando os *Doctors* dos serviços *C* e *C'* sempre que *C* depender de *C'*.

Cada *Doctor* obedece uma interface bem simples, como a ilustrada no Código Fonte 5.4, para permitir que o processo de colaboração se dê sem maiores problemas. O método *isFaulty()* retorna *True* caso o serviço sendo testado falhe em pelo menos um dos testes, enquanto que o método *getCause()* retorna a provável causa para a falha naquele serviço, que pode estar no próprio serviço ou em algum serviço do qual ele dependa. O método *addDoctor()* adiciona uma dependência entre dois *Doctors*, criando um arco no grafo de dependência entre os *Doctors*.

Tal associação pode ser realizada de forma dinâmica, consultando algum sistema de informação da grade, ou de forma estática, baseada em um grafo pré-calculado com as dependências entre os diferentes tipos de serviços oferecidos pelo *middleware* utilizado na grade. A única restrição quanto ao grafo de dependências entre os *Doctors* é que ele não pode possuir ciclos, o que poderia levar o sistema a entrar em um *loop* infinito. Com o grafo de dependências construído, a tarefa do *GridDoctor* se resume a disparar o esforço colaborativo descrito anteriormente para que o componente onde a falha foi originada seja encontrado.

O processo de diagnóstico se inicia pelo *Doctor* da aplicação do usuário, cuja implementação é fornecida pelo *framework* como mostrado pelo Código Fonte 5.5, cujo método

Código Fonte 5.4: Interface *DoctorIF.java*

```
1  /**
2   * Interface que define o comportamento esperado de um Doctor
3   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
4   */
5  public interface DoctorIF {
6          /** @returns true caso o componente falhe a pelo menos um dos testes e false caso
               contrário.
7           */
8          public boolean isFaulty();
9          /** @returns a provável causa para a falha no caso do método isFaulty() ter
               retornado true.
10          */
11         public Cause getCause();
12         /**
13          * Adiciona um Doctor à lista de dependências.
14          * @param aDoctor - um outro Doctor.
15          */
16         public void addDoctor(DoctorIF aDoctor);
17 }
```

isFaulty(), por definição, retorna sempre *True* uma vez que o processo de diagnóstico só é iniciado no caso de uma tentativa frustrada de execução de uma aplicação na grade. Esse comportamento padrão é adotado para evitar que os usuários sejam obrigados a fornecer testes automáticos para suas aplicações e um *Doctor* capaz de executar tais testes. Porém, tal simplificação pode implicar em diagnósticos equivocados de faltas na aplicação uma vez que caso nenhuma falha seja encontrada nos serviços utilizados em uma tentativa frustrada de execução, a culpa pela falha será atribuída, talvez erroneamente, à própria tarefa.

O *AppDoctor* consulta os *Doctors* dos serviços dos quais a aplicação depende para ser executada, que por sua vez invocam os *Doctors* dos serviços dos quais eles próprios dependem, à procura de uma causa para a falha. Caso todos os *Doctors* informem que os serviços pelos quais são responsáveis funcionam corretamente, o *AppDoctor* conclui que a falha foi causada por um problema na própria aplicação. Por outro lado, se um dos *Doctors* encontrar um falha nos testes automáticos do serviço pelo qual é responsável tal serviço é apontado como culpado pela falha na execução da aplicação.

Código Fonte 5.5: Classe *AppDoctor.java*

```
1  import java.util.*;
```

```
2  /**
3   * Implementação da interface DoctorIF para diagnosticar a causa da falha na execução
4   * de uma aplicação do usuário.
5   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
6   */
7  public class AppDoctor implements DoctorIF {
8          /** Lista com os doctors dos serviços dos quais este Doctor depende.
9           */
10         private List<DoctorIF> doctors;
11         /** Construtor padrão
12          */
13         public AppDoctor() {
14                 doctors = new Vector<DoctorIF>();
15         }
16         /** @returns A causa da falha. Se algum dos Doctors do grafo de dependência
               detectar uma falha
17          * ele deve diagnosticar sua causa. Se não houve falhas em nenhum dos serviços
                utilizados
18          * pela aplicação, consideramos que a falha foi causada por um defeito na própria
                aplicação.
19          */
20         public Cause getCause() {
21                 Iterator<DoctorIF> it = doctors.iterator();
22                 while(it.hasNext()) {
23                         DoctorIF d = (DoctorIF) it.next();
24                         if( d.isFaulty() ) return d.getCause();
25                 }
26                 return new Cause("The problem is in the Application itself.");
27         }
28         /** @returns true uma vez que o diagnóstico só é iniciado quando a execução da
               aplicação falhou.
29          */
30         public boolean isFaulty() {
31                 return true;
32         }
33         /**
34          * @param aDoctor um outro Doctor para a lista de dependências.
35          */
36         public void addDoctor(DoctorIF aDoctor) {
37                 doctors.add(aDoctor);
38         }
39 }
```

5.3.1 Instanciação do *framework GridDoctor*: o *gLiteGridDoctor*

O *framework* descrito na seção anterior foi instanciado para o *middleware gLite* para que a abordagem proposta pudesse ser avaliada através de um experimento prático.

Escolher o *gLite* como plataforma para realização do experimento prático foi a alternativa mais apropriada por conta da existência do SAM, descrito no capítulo anterior, o que facilitaria significativamente a implementação do que batizamos de *gLiteGridDoctor*, uma instanciação do *GridDoctor* para diagnóstico de faltas em grades criadas com o *gLite*. A implementação aproveitou sempre que possível código e funcionalidades presentes no SAM.

A Figura 5.4 mostra a arquitetura do *framework GridDoctor* instanciada para o *gLite*.

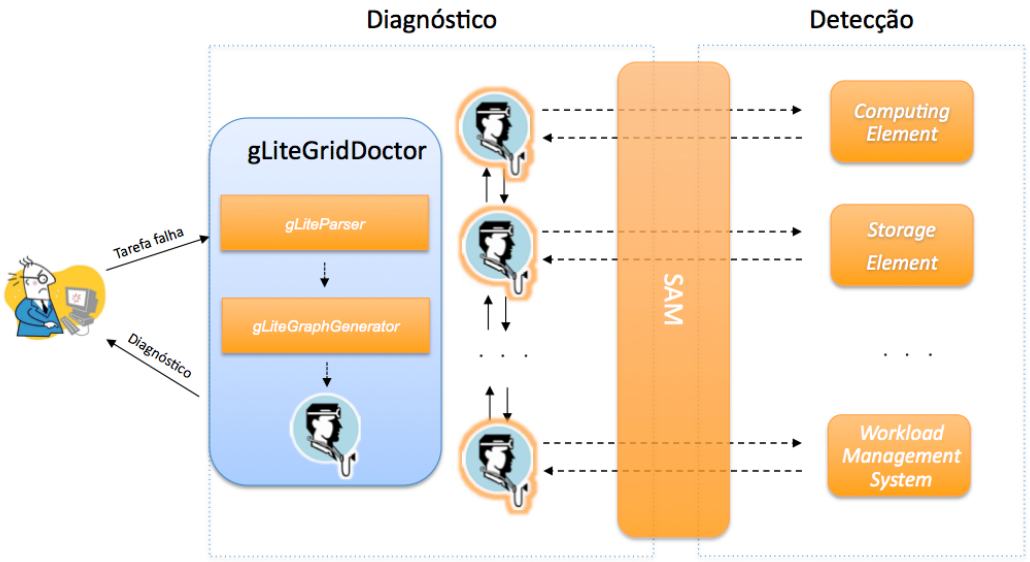


Figura 5.4: Arquitetura do *gLiteGridDoctor*

A classe principal do *gLiteGridDoctor*, vista no Código Fonte 5.6, se limita a implementar os métodos abstratos da classe *GridDoctor* e a fornecer um método principal para execução da aplicação.

Código Fonte 5.6: Classe *gLiteGridDoctor.java*

```
1  /**
2   * Classe principal do gLiteGridDoctor, a instanciação do GridDoctor
3   * para diagnósticar faltas em grades gLite.
4   * @author Alexandre Nóbrega Duarte – alexandrend@gmail.com
5   */
6  public class gLiteGridDoctor extends GridDoctor {
7          /**
```

```
8           * Método principal da classe gLiteGridDoctor
9           * @param args - filenane - o programa recebe como argumento da linha de comando
10          * o nome do arquivo com o log da execução da tarefa.
11          * @throws ImpossibleToDiagnoseException
12          */
13          public static void main(String args[]) throws ImpossibleToDiagnoseException {
14                  GridDoctor g = new gLiteGridDoctor(args[1]);
15                  Cause c = g.diagoseFault();
16                  System.out.println(c);
17          }
18          /**
19           * Construtor padrão.
20           * @param filename
21           */
22          public gLiteGridDoctor(String filename) {
23                  super(filename);
24          }
25          /**
26           * Instanciação do método createGraphGenerator para o middleware gLite
27           */
28          protected GraphGenerator createGraphGenerator() {
29                  return new gLiteGraphGenerator();
30          }
31          /**
32           * Instanciação do método createParser para o midleware gLite
33           */
34          protected Parser createParser(String filename) {
35                  return new gLiteParser(filename);
36          }
37  }
```

Uma vez que o SAM já executa periodicamente os testes automáticos de aceitação dos serviços da grade, armazena seus resultados em uma base de dados e oferece uma interface simples para consultar o estado de um determinado serviço, a implementação dos *gLiteDoctors* se resumiu a uma consulta à base de dados especificando apenas o serviço monitorado.

A implementação do *gliteDoctor*, exibida no Código Fonte 5.7, foi baseada no *script* fornecido pelo SAM para consultar o *web service* de acesso à base de dados, como apresentado na Figura 4.2. O método *runQuery()* executa o *script* de consulta do SAM passando como parâmetros o nome do campo que se deseja consultar na base de dados, o tipo de serviço sendo consultado e o nome da máquina onde o serviço está sendo executado. Uma consulta pelo campo *servicestatus* retorna um valor entre *Up*, caso o serviço tenha passado em todos os testes do SAM e *Down*, caso contrário. Uma consulta pelo campo *testresultdetails* retorna

detalhes sobre o resultado do teste, em um formato semelhante ao apresentado no Apêndice A. O método *getCause()* implementa o comportamento colaborativo proposto. O *Doctor* primeiro verifica se algum dos *Doctors* dos componentes dos quais ele depende identificaram a causa da falha. Se isso acontece, ele considera tal componente como o culpado pela falha e informa isso para quem invocou inicialmente seu método *getCause()*. Caso nenhum dos componentes dos quais ele depende acusarem uma falha, ele próprio se acusa como culpado.

Código Fonte 5.7: Classe *gLiteDoctor.java*

```
1  import java.io.IOException;
2  import java.util.*;
3  /**
4   * Implementação da interface DoctorIF para o middleware gLite.
5   * Para checar o estado de um serviço fazemos uma simples consulta à base
6   * de dados do SAM utilizando o script de linha de comando same-query
7   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
8   */
9  public class gLiteDoctor implements DoctorIF {
10         /** Script de consulta à base de dados
11          */
12         private static final String script="/opt/lcg/same/client/bin/same-query";
13         /** Serviço gLite ao qual este doctor está associado.
14          */
15         private gLiteService service;
16         /** Lista com os doctors dos serviçso dos quais este Doctor depende.
17          */
18         private List<DoctorIF> doctors;
19         /**
20          * @param gLiteService - Um objeto que representa um serviço gLite
21          */
22         public gLiteDoctor(gLiteService service) {
23                 this.service = service;
24                 doctors = new Vector<DoctorIF>();
25         }
26         /** @returns A causa da falha. Se algum dos Doctors do grafo de dependência
               detectar uma falha
27          * ele deve diagnosticar a causa. Caso contrário, o serviço é apontado como causa
28          * para a falha na execução da tarefa e causa é obtida por uma consulta a base de
               dados
29          * do SAM para obter os últimos resultados dos testes para este Doctor.
30          */
31         public Cause getCause() {
32                 Iterator<DoctorIF> it = doctors.iterator();
33                 while(it.hasNext()) {
34                         DoctorIF d = (DoctorIF) it.next();
```

```
35                      if( d.isFaulty() ) return d.getCause();
36              }
37              return new Cause(service, runQuery("testresultdetails"));
38      }
39      /** @returns true caso o serviço tenha sido considerado "down" pelo SAM.
40       */
41      public boolean isFaulty() {
42              return runQuery("servicestatus").equalsIgnoreCase("Down");
43      }
44      /**
45       * Executa uma consulta à base de dados do SAM
46       * @param what - o que se deseja consultar
47       * @return - um string com o resultado da consulta.
48       */
49      private String runQuery( String what ) {
50              StringBuffer sb = new StringBuffer();
51              try {
52                      Scanner s = new Scanner(Runtime.getRuntime().exec(script + " " +
                            what + "serviceabbr=" + service.getType() + " nodename=" +
                            service.getName()).getInputStream());
53                      while( s.hasNext() ) sb.append(s.next());
54              } catch (IOException e) {
55              }
56              return sb.toString();
57      }
58      /**
59       * @param aDoctor um outro Doctor para a lista de dependências.
60       */
61      public void addDoctor(DoctorIF aDoctor) {
62              doctors.add(aDoctor);
63
64      }
65  }
```

A classe *gLiteParser*, exibida no Código Fonte 5.8, é responsável por extrair do arquivo de *log* da tentativa de execução da tarefa do usuário os nomes dos serviços envolvidos na falha da execução. Sua implementação também foi bastante facilitada uma vez que o SAM oferece um mecanismo de consulta que permite identificar o tipo de um determinado serviço a partir de seu nome. O método *getServices()* varre todo o arquivo de *log* a procura de nomes de máquinas e sempre que encontra um consulta a base de dados do SAM para tentar identificar se a máquina é ou não um serviço da grade. Se a máquina estiver presente na base de dados do SAM o método *getServiceType()* retornará seu tipo, caso contrário a máquina será ignorada.

Código Fonte 5.8: Classe *gLiteParser.java*

```
1  import java.util.*;
2  import java.io.*;
3  import java.net.*;
4  /**
5   * Parser capaz de extrair todos os serviços gLite mencionados em um arquivo de log de
        execução de uma tarefa.
6   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
7   */
8  public class gLiteParser implements Parser {
9          /** Script de consulta à base de dados
10          */
11         private static final String script="/opt/lcg/same/client/bin/same-query";
12         /** Nome do arquivo de log
13          */
14         private String filename;
15         /**
16          * Construtor
17          */
18         public gLiteParser(String filename) {
19                 this.filename = filename;
20         }
21         /**
22          * Extrai todos os serviços de grade de um arquivo de log de execução de tarefa.
23          * @return - um conjunto com todos os serviços da grade mencionados no arquivo de
                log
24          * @throws IOException - se o arquivo de log não puder ser acessado.
25          */
26         public Set<GridService> getServices() throws IOException {
27                 Set<String> already = new HashSet<String>();
28                 Set<String> trash = new HashSet<String>();
29                 Set<GridService> services = new HashSet<GridService>();
30                 Scanner s = new Scanner( new InputStreamReader(new FileInputStream(filename
                       )));
31                 while(s.hasNext()) {
32                         String hostname = s.next();
33                         if( !already.contains(hostname) && !trash.contains(hostname)) {
34                                 try {
35                                         InetAddress.getByName(hostname);
36                                         int type = getServiceType(hostname);
37                                         if(type != gLiteService.OTHER ) {
38                                                 services.add(new gLiteService(hostname,
                                                        type));
39                                                 already.add(hostname);
40                                         }
41                                 } catch( UnknownHostException e) {
42                                         trash.add(hostname);
```

```
43                                  }
44                          }
45                  }
46                  return services;
47          }
48          /**
49           * Executa uma consulta à base de dados do SAM para descobrir o tipo de serviço
50           * executado em uma determinada máquina
51           * @param hostname - o nome de host do provável serviço
52           * @return - Pode retornar CE, WMS, BDII, SE ou OTHER se o serviço não for
                 encontrado.
53           */
54          private int getServiceType( String hostname ) {
55                  StringBuffer sb = new StringBuffer();
56                  try {
57                          Scanner s = new Scanner(Runtime.getRuntime().exec(script + "
                                serviceabbr " + " nodename=" + hostname ).getInputStream());
58                          while( s.hasNext() ) sb.append(s.next());
59
60                          if( sb.toString().equals("CE")) return gLiteService.CE;
61                          if( sb.toString().equals("SE")) return gLiteService.SE;
62                          if( sb.toString().equals("BDII")) return gLiteService.BDII;
63                          if( sb.toString().equals("WMS")) return gLiteService.WMS;
64
65                  } catch (IOException e) {
66                  }
67                  return gLiteService.OTHER;
68          }
69  }
```

Finalmente, temos a classe *gLiteGraphGenerator*, exibida no Código Fonte 5.9, responsável pela geração do grafo de dependência entre os *gLiteDoctors*, que possibilita o diagnóstico colaborativo da origem da falha. Decidimos por mapear de forma estática as relações de dependência entre os diferentes tipos de serviços do *gLite* uma vez que tais relacionamentos não são alterados com frequência, e usar esse mapeamento para estabelecer as dependências entre os serviços envolvidos na execução da tarefa do usuário.

A Figura 5.5 ilustra o grafo estático de dependência entre os principais tipos de serviços da grade *gLite*, que possuem serviços equivalentes na maioria dos *middleware* de grade computacional existentes hoje, e a aplicação do usuário. Note que o grafo não apresenta ciclos o que implica que grafos de dependência criados a partir deste grafo estático também não terão ciclos, como requerido pelo *framework*.

A aplicação do usuário depende do WMS para que a tarefa seja escalonada para um *Computing Element* capaz de atender todos os seus pré-requisitos e do próprio *Computing Element* para que a tarefa possa ser efetivamente executada em um de seus *Worker Nodes*. O WMS, por sua vez, depende do Sistema de Informação da grade (BDII) para poder obter as informações sobre os *Computing Elements* existentes e ser capaz de fazer o *match-making* dos requisitos da tarefa com as propriedades dos *Computing Elements* disponíveis. Já o *Computing Element* depende do *Storage Element* para armazenar e recuperar dados necessários para a execução da tarefa do usuário e do Sistema de Informação para encontrar um *Storage Element* que possua o espaço de armazenamento requisitado pela tarefa ou que possua os dados necessários para sua execução.

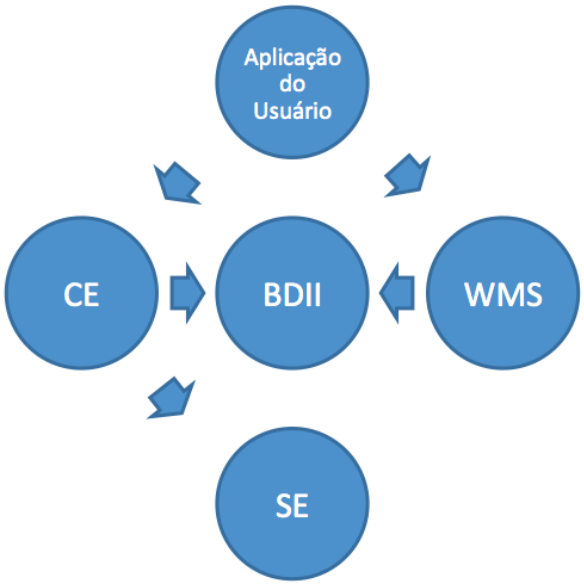


Figura 5.5: Grafo de Dependência para os Serviços do gLite

Código Fonte 5.9: Classe *gLiteGraphGenerator.java*

```
1  import java.util.*;
2  /**
3   * Implementação da interface GraphGenerator, capaz de gerar
4   * o grafo de dependência para os Doctors de uma grade gLite.
5   * @author Alexandre Nóbrega Duarte - alexandrend@gmail.com
6   */
7  public class gLiteGraphGenerator implements GraphGenerator {
8          /**
9           * Geração de um grafo de dependências para um conjunto de serviços gLite.
10          * @param services - Conjunto dos serviços gLite envolvidos na tentativa de
                execução da aplicação
```

```
11          * @return - um grafo de dependência entre gLiteDoctors cujo nó inicial é um
                AppDoctor.
12          */
13         public AppDoctor generateGraph(Set <GridService> services) {
14
15                 AppDoctor appDoctor = new AppDoctor();
16
17                 gLiteService gServices[] = (gLiteService[]) services.toArray();
18                 gLiteDoctor gDoctors[] = new gLiteDoctor[gServices.length];
19
20                 //Criação de um gLiteDoctor para cada serviço.
21                 for(int i = 0; i < gServices.length; i++)
22                         gDoctors[i] = new gLiteDoctor(gServices[i]);
23
24                 //Dependências da Aplicação do usuário (AppDoctor - CE e WMS)
25                 for(int i = 0; i < gServices.length; i++)
26                         if( gServices[i].getType() == gLiteService.CE || gServices[i].
                                getType() == gLiteService.WMS )
27                                 appDoctor.addDoctor(gDoctors[i]);
28
29                 //Dependências dos demais serviços
30                 for( int i = 0; i < gServices.length; i++) {
31                         for( int j = 0; j < gServices.length; j++ ){
32                                 //Dependências do Computing Element (BDII e SE)
33                                 if( gServices[i].getType() == gLiteService.CE) {
34                                                 if( gServices[j].getType() == gLiteService.
                                                        BDII || gServices[j].getType() ==
                                                        gLiteService.SE )
35                                                         gDoctors[i].addDoctor(gDoctors[j]);
36                                 }
37                                 //Dependências do WMS (BDII)
38                                 else if( gServices[i].getType() == gLiteService.WMS ) {
39                                                 if( gServices[j].getType() == gLiteService.
                                                        BDII )
40                                                         gDoctors[i].addDoctor(gDoctors[j]);
41                                 }
42                         }
43                 }
44                 return appDoctor;
45         }
46 }
```

5.4 Considerações Finais do Capítulo

A abordagem descrita neste capítulo resultou na criação de um *framework* para o diagnóstico de faltas em grades computacionais, na instanciação desse *framework* para a criação de uma ferramenta capaz de diagnosticar causas de falhas em uma grade computacional tomando por base os resultados produzidos pela ferramenta de monitoração descrita no Capítulo 4 e na publicação de dois artigos científicos.

O primeiro trabalho foi apresentado na *20th International Conference on Advanced Information Networking and Applications* [Duarte et al. 2006] e descreveu um protótipo inicial implementado em Java do *framework* para diagnóstico de faltas que serviu para aferir a viabilidade da implementação da solução apresentada neste capítulo. O protótipo desenvolvido foi testado em duas grades computacionais baseadas em dois *middleware* diferentes, o *OurGrid* [Cirne et al. 2006] e o *Globus* [Foster e Kesselman 1997] e deu indícios iniciais de que a abordagem proposta seria viável, possibilitando que o restante do trabalho pudesse ser desenvolvido.

Após a execução deste experimento inicial e da obtenção de indicação de que a solução proposta seria viável, partimos para a elaboração de um experimento maior e mais representativo, descrito em detalhes no Capítulo 6, e que pudesse nos dar resultados claros da efetividade da nossa solução para o diagnóstico de faltas em grades computacionais.

O segundo trabalho, descrevendo um refinamento da solução inicial proposta no trabalho anterior, foi apresentando no *Student Forum* do *Fourth Latin-American Symposium on Dependable Computing*.

Capítulo 6

Avaliação Experimental

Decidimos conduzir um experimento para avaliar a efetividade da nossa proposta para o diagnóstico da faltas em grades computacionais baseada no uso de testes automáticos de software.

Aproveitando o trabalho que realizamos para a detecção de falhas utilizando testes automáticos de software para grades baseadas no *middleware gLite*, apresentado no Capítulo 4, decidimos criar uma instância do *framework GridDoctor* capaz de diagnosticar faltas em serviços de grades baseadas neste mesmo *middleware*: o *gLiteGridDoctor*. Apresentamos a seguir detalhes sobre o experimento realizado com o *gLiteGridDoctor* em uma grade real e discutimos os resultados obtidos com o experimento.

6.1 Definição do Experimento

O experimento foi realizado utilizando a grade computacional criada no contexto do projeto $EELA-2$ (*E-Infrastructure shared between Europe and Latin America*) [EELA 2010].

A grade do projeto $EELA-2$ foi criada utilizando o *middleware gLite* e oferece cerca de 3.000 processadores e 7 petabytes de espaço de armazenamento para usuários de 40 instituições, localizados em 13 países da Europa e América Latina (Figura 6.1) [EELA 2010].

A infraestrutura de grade do projeto $EELA-2$ é monitorada pelo SAM, que checa os estados de todos os serviços da grade e publica os resultados das checagens em sua base de dados. Na grade do $EELA-2$, assim como em várias outras grades, o SAM testa todos os serviços da grade em intervalos de uma hora.

Figura 6.1: Infraestrutura de Grade do Projeto EELA-2

Decidimos focar o experimento apenas nos serviços periféricos da grade (*Computing Element* e *Storage Element*), encontrados em todos os provedores de recursos da grade, por serem estes os pontos críticos no processo de diagnóstico. Serviços centrais como o WMS são encontrados me menor número na grade e utilizados por um grande número usuários simultâneamente, o que facilita o processo de identificação de uma falta nestes serviços. Além disso, serviços centrais geralmente são gerenciados por equipes mais experimentes, com habilidade para detectar e corrigir rapidamente potenciais problemas. Por outro lado, os serviços localizados nos provedores de recursos muitas vezes são gerenciados por equipes inexperientes, que não têm ferramentas ou conhecimento para descobrir e corrigir problemas tão logo eles ocorram. Além disso, pela pouca experiência, a probabilidade da ocorrência de falhas de configuração aumenta sensilvelmente. Finalmente, esses dois tipos de serviços são geralmente encontrados na maioria dos *middleware* de computação em grade, o que torna mais fácil a reprodução do experimento utilizando outras soluções e infraestruturas disponíveis.

Criamos para este experimento dois tipos de aplicações de grade. A primeira aplicação copia um arquivo do *Computing Element* onde está sendo executada a aplicação para um *Storage Element*, em seguida copia de volta o arquivo do *Storage Element* para o *Computing Element* com um outro nome e faz uma comparação para checar se o arquivo recebido é idêntico ao arquivo enviado. Na situação em que não ocorram falhas no *Computing Element*

ou no *Storage Element* utilizados, a execução da aplicação deve ser concluída com sucesso. A segunda aplicação faz as mesmas transferências de dados da primeira mas apresenta uma falta proposital na comparação dos arquivos. Com isso, a execução de aplicações do segundo tipo sempre falham, independentemente da existência ou não de falhas nos serviços utilizados.

O primeiro tipo de aplicação é utilizado para checar o diagnóstico de faltas nos serviços da grade enquanto que o segundo tipo serve para checar se o sistema é capaz de diagnosticar uma falta existente na própria aplicação.

Criamos então quatro tipos de submissões envolvendo estas duas aplicações e ocorrência de falhas em no máximo um dos serviços utilizados, a saber:

1. **Submissão com Falha na Aplicação**: submetemos a aplicação incorreta para execução utilizando um *Computing Element* e um *Storage Element* considerados disponíveis (*Up*) pelo SAM nas três últimas checagens.

2. **Submissão com Falha no *Computing Element***: submetemos a aplicação correta para execução utilizando um *Computing Element* considerado indisponível (*Down*) pelo SAM nas três últimas checagens e um *Storage Element* considerado disponível (*Up*) pelo SAM nas três últimas checagens.

3. **Submissão com Falha no *Storage Element***: submetemos a aplicação correta para execução utilizando um *Storage Element* considerado indisponível (*Down*) pelo SAM nas três últimas checagens e um *Computing Element* considerado disponível (*Up*) pelo SAM nas três últimas checagens.

4. **Submissão Correta**: submetemos a aplicação correta para execução utilizando um *Computing Element* e um *Storage Element* considerados disponíveis (*Up*) pelo SAM nas três últimas checagens.

Não achamos necessário criar um quinto cenário com falhas simultâneas nos dois serviços porque nesta situação um diagnóstico apontando qualquer um dos dois serviços como culpado pela falha na execução da aplicação do usuário estaria correto.

O artifício de considerar como disponíveis ou indisponíveis apenas os serviços considerados como tal em três checagens consecutivas do SAM serve para diminuir a chance de

classificarmos como indisponível um serviço que teve apenas uma falha transiente, voltando a funcionar em seguida, ou de considerarmos como disponível um serviço com problemas mas que por alguma razão é considerado disponível de forma intermitente. Essas medidas são também utilizadas pelos operadores de infraestruturas de grade, responsáveis por seu monitoramento, antes de considerarem um problema previamente detectado resolvido e antes de enviarem uma notificação para o administrador de um provedor de recursos solicitando uma ação face à detecção de um novo problema.

As tarefas foram submetidas sempre 10 minutos após o SAM ter realizado a sua última checagem e, para o caso de tarefas ainda não concluídas, canceladas 40 minutos depois da submissão, 10 minutos antes do SAM realizar sua próxima checagem. Uma vez que as tarefas em si não precisam de mais do que alguns poucos minutos para serem executadas, uma execução não ser concluída após 40 minutos significa que o *Computing Element* está sobrecarregado e que a tarefa encontra-se na fila de espera do LRMS. Tal medida foi tomada para evitar diferenças nos dados disponíveis na base de dados do SAM entre o momento da submissão das tarefas e o momento da realização dos diagnósticos. Após a recuperação do *log* da execução de cada tarefa, o *gliteGridDoctor* foi utilizado para diagnosticar a causa de cada falha.

A situação de que uma tarefa não foi executada 40 minutos após sua submissão pode acontecer por várias razões. Por exemplo, o *Computing Element* escolhido pode estar sobrecarregado de trabalho e a tarefa pode permanecer na fila do LRMS por muito tempo.

Para cada tarefa submetida foram armazenados a data da submissão, o identificador da tarefa na grade e o resultado esperado para sua execução, definido de acordo com o tipo da tarefa submetida. Após a conclusão da execução da tarefa, com ou sem falha, foram armazenados também o *log* final da execução e o diagnóstico efetuado pelo *gLiteGridDoctor* para os casos em que a execução não foi concluída com sucesso.

Submetemos para execução na grade do projeto $EELA-2$ (no máximo) uma tarefa de cada um dos quatro tipos mencionados acima por hora entre os dias $15/12/2009$ e $05/01/2010$ e entre os dias $04/02/2010$ e $28/02/2010$, resultando em 4.238 tarefas executadas nos 47 dias do experimento, o que nos dá uma média de $3,76$ tarefas por hora. Esse número é menor do que os 4.0 esperados por diversos fatores, incluindo o cancelamento de tarefas que excederam os 40 minutos máximos de execução e a não submissão de tarefas por

inexistência de *Computing Elements* ou *Storage Elements* com as características necessárias para o tipo de tarefa.

O experimento precisou ser suspenso entre os dias $06/01/2010$ e $03/02/2010$ por conta de um problema com a máquina na qual o SAM estava instalado e de onde eram submetidas as tarefas do experimento.

6.2 Resultados Experimentais

A análise dos resultados do experimento considera três tipos de resultados. O primeiro é o *Resultado Esperado* para o conjunto de tarefas submetidas, baseado no diagnóstico pré-definido para cada tipo de tarefa no momento da submissão. O segundo é o *Resultado Obtido*, baseado no diagnóstico realizado pelo *gLiteGridDoctor* e o último é o *Resultado Real*, baseado na análise manual dos *logs* das execuções das tarefas.

O *Resultado Esperado* serve como uma estimativa do resultado que seria obtido com o experimento no caso dos estados dos serviços da grade não serem alterados entre a última avaliação realizada pelo SAM e o momento em que a tarefa é executada.

Definimos a taxa de acertos de nossa abordagem para o diagnóstico de faltas como sendo o percentual de faltas que são corretamente diagnosticadas:

$$Taxa\ de\ Acerto = \frac{Diagnosticos\ Corretos}{Total\ de\ Diagnosticos} \tag{6.1}$$

Um diagnóstico incorreto pode ser também um *Falso Positivo* ou um *Falso Negativo*. Um *Falso Positivo* ocorre quando não ocorre falha na execução da tarefa e mesmo assim o sistema emite um diagnóstico de falta para algum serviço de grade enquanto que um *Falso Negativo* ocorre quando acontece uma falha na execução da tarefa e o sistema não emite qualquer diagnóstico.

Apresentamos nesta seção os resultados experimentais obtidos para cada um dos tipos de tarefas submetidas seguidos por uma análise geral do experimento.

6.2.1 Tarefa tipo 1: Submissão com Falha na Aplicação

Durante a duração do experimento foram executadas 1.052 tarefas do tipo 1, cujo diagnóstico esperado seria *Falha na Aplicação* uma vez que este tipo de tarefa foi submetido utilizando

Tabela 6.1: Submissão com Falha na Aplicação

	Falha App.	Falha SE	Falha CE
Resultado Esperado	1.052	0	0
Resultado Obtido	1.052	0	0
Resultado Real	991	37	24

um *Computing Element* e um *Storage Element* considerados disponíveis nas últimas três checagens realizadas pelo SAM.

A Tabela 6.1 apresenta os resultados obtidos para este tipo de tarefa. O *Resultado Esperado* era ter 1.052 diagnósticos de falha na própria aplicação, 0 diagnósticos de falhas no *Storage Element* e 0 diagnósticos de falha no *Computing Element*. O *Resultado Obtido* foi exatamente igual ao *Resultado Esperado*, o que aparentemente indicaria uma taxa de sucesso de 100%, porém, ao analisar manualmente o *log* das execuções para definir o *Resultado Real*, observamos algumas discrepâncias. Das 1.052 tarefas falhas, apenas 991 falharam por conta do erro proposital na aplicação. 37 delas falharam por um problema com o *Storage Element* selecionado e 24 por um problema com o *Computing Element*. Tais erros de diagnóstico foram cometidos porque um desses dois serviços falhou entre duas checagens do SAM, deixando o sistema em estado inconsistente, sem conhecimento do real estado desses serviços. Por conta disso, os *gLiteDoctors* responsáveis por diagnosticar a falta no *Computing Element* ou no *Storage Element* falharam. No final, o *gLiteGridDoctor* conseguiu uma taxa de acertos de $94,20\%$, errando em apenas 61 dos 1.052 diagnósticos realizados para este tipo de tarefa. Tanto o número de *Falsos Positivos* quanto o número de *Falsos Negativos* foram iguais a 0.

6.2.2 Tarefa tipo 2: Submissão com Falha no *Storage Element*

Foram submetidas 1.061 tarefas do tipo 2, preparadas para utilizar um *Storage Element* considerado indisponível e um *Computing Element* considerado disponível pelo SAM. Nesse contexto o *Resultado Esperado* seria 1.061 diagnósticos de falha por problema no *Storage Element* porém o *Resultado Obtido* mostrou apenas 1.044 falhas por problemas no *Storage Element* (Tabela 6.2) quando o *Resultado Real* seria 1.014 falhas por problemas no *Storage Element* e 30 falhas por problemas no *Computing Element* . O desvio nos resultados obtidos em relação aos esperados foi causado por transições no estado dos *Storage Elements*,

Tabela 6.2: Submissão com Falha no *Storage Element*

	Falha App.	Falha SE	Falha CE
Resultado Esperado	0	1.061	0
Resultado Obtido	0	1.044	0
Resultado Real	0	1.014	30

que foram considerados indisponíveis antes da submissão mas voltaram a funcionar no momento da execução da tarefa e também por mudanças no estado dos *Computing Elements*, que foram considerados disponíveis antes da submissão da tarefa mas falharam antes de sua execução. Por conta dos 30 diagnósticos equivocados dentre os 1.044 realizados, a taxa de sucesso do *gLiteGridDoctor* para este tipo de tarefa foi de $97,13\%$. Novamente não houve registro de *Falsos Positivos* ou *Falsos Negativos*.

6.2.3 Tarefa tipo 3: Submissão com Falha no *Computing Element*

Os resultados para a tarefa de tipo 3, onde o componente faltoso é o *Computing Element*, foram muito próximos dos resultados para a tarefa tipo 2.

Foram submetidas 1.064 tarefas do tipo 3 e o *Resultado Esperado* seria 1.064 falhas por problemas no *Computing Element*. Porém, ao invés disso, o *Resultado Obtido* trouxe apenas 1.031 (Tabela 6.3) diagnósticos apontando problemas no *Computing Element* quando o *Resultado Real* seria 998 falhas por essa razão e 33 falhas por problemas no *Storage Element*.

Mais uma vez, a razão para os 33 diagnósticos incorretos entre os 1.031 realizados foi alteração no estado dos serviços, fazendo com que 33 tentativas de execução que deveriam falhar fossem completadas e que outras 33 tarefas falhas cujo diagnóstico emitido apontou uma falha no *Computing Element* tenham falhado na verdade por problemas no *Storage Element*.

Devido aos 33 erros cometidos, a taxa de sucesso para este tipo de tarefa foi de $96,80\%$ e também não houve registro de *Falsos Positivos* ou *Falsos Negativos*.

Tabela 6.3: Submissão com Falha no *Computing Element*

	Falha App.	Falha SE	Falha CE
Resultado Esperado	0	0	1.064
Resultado Obtido	0	0	1.031
Resultado Real	0	33	998

Tabela 6.4: Submissão Correta

	Falha App.	Falha SE	Falha CE
Resultado Esperado	0	0	0
Resultado Obtido	68	0	0
Resultado Real	0	29	39

6.2.4 Tarefa tipo 4: Submissão Correta

Finalmente, também foram submetidas 1.061 tarefas do tipo 4 durante os 47 dias do experimento. Uma vez que não há erro na aplicação em execução e para cada submissão foram selecionados *Computing Elements* e *Storage Elements* considerados disponíveis nas últimas três checagens realizadas pelo SAM, o *Resultado Esperado* seria que todas as tarefas fossem executadas corretamente e que nenhum diagnóstico fosse realizado. Porém, novamente aconteceram alterações no estado dos serviços selecionados e 68 das 1.061 tarefas acabaram falhando, como exibido na Tabela 6.4.

O *gLiteGridDoctor* diagnosticou erroneamente a causa das 68 falhas como sendo problemas na própria aplicação, uma vez que tanto o *Computing Element* quanto o *Storage Element* apareciam na base de dados do SAM como disponíveis, fazendo com que a resposta padrão fosse utilizada, acusando a própria aplicação de ser a origem do problema. Analisando os *logs* das tentativas falhas de execução constatamos que na verdade 29 delas falharam por conta de problemas no *Computing Element* e 39 por conta de problemas no *Storage Element*, fazendo com que o *gLiteGridDoctor* obtivesse uma taxa de sucesso de 0%, não acertando nenhum dos diagnósticos efetuados para este tipo de tarefa. Também não houve registro de *Falsos Positivos* ou *Falsos Negativos*.

É importante reforçar que este resultado se deve diretamente à decisão de considerar a própria aplicação como falha sempre que os serviços da grade utilizados por ela em uma

Tabela 6.5: Resumo dos Diagnósticos Realizados

Tipo de Tarefa	Diagnósticos	Acertos	Erros	Taxa de Acerto
1	1.052	991	61	94,20%
2	1.044	1.014	30	97,12%
3	1.031	998	33	96,80%
4	68	0	68	0%
Total	3.195	3.003	192	93,99%

tentativa frustrada de execução forem considerados disponíveis. Tal decisão foi tomada com o objetivo de não obrigar o usuário do *gLiteGridDoctor* a ter que fornecer um conjunto de testes automáticos para sua aplicação. Porém, na existência de tal conjunto de testes seria possível alterar a class *AppDoctor* para que seu método *isFaulty()* só retornasse *True* caso um dos testes da aplicação falhasse. No entanto, mesmo existindo tais testes para a aplicação do usuário, a taxa de acertos nesse cenário se manteria a mesma uma vez que ao invés de diagnosticar erroneamente a aplicação como culpada pela falha, o *gLiteGridDoctor* não iria emitir nenhum diagnóstico já que os testes da aplicação, em tese, não falhariam.

Este é um exemplo claro dos efeitos da assincronia encontrada em um sistema distribuído real, como a grade computacional utilizada durante os experimentos, e demostra uma limitação prática para soluções que visam o diagníostico de faltas neste tipo de infraestrutura.

6.2.5 Análise dos Resultados

A Tabela 6.5 mostra um resumo dos resultados dos experimentos para os quatro tipos de tarefas. Observamos o diagnóstico de 3.195 falhas nas 4.238 tarefas submetidas. A taxa de falhas nas tarefas submetidas foi de $75,38\%$, muito próxima dos 75% esperados uma vez que 3 dos 4 tipos de tarefas criadas possuem falhas.

Destes 3.195 diagnósticos, 3.003 foram considerados diagnósticos corretos e 192 foram considerados diagnósticos incorretos.

O gráfico exibido na Figura 6.2 mostra a evolução da taxa média de acerto com o número de diagnósticos efetuados. É possível observar que após cerca de 1600 diagnósticos efetuados a taxa de média de acertos começa a convergir para 94% e a partir dai pouco oscila em relação a este valor, mostrando que a quantidade de diagnósticos realizados foi o suficiente

para termos uma boa média da taxa de acerto.

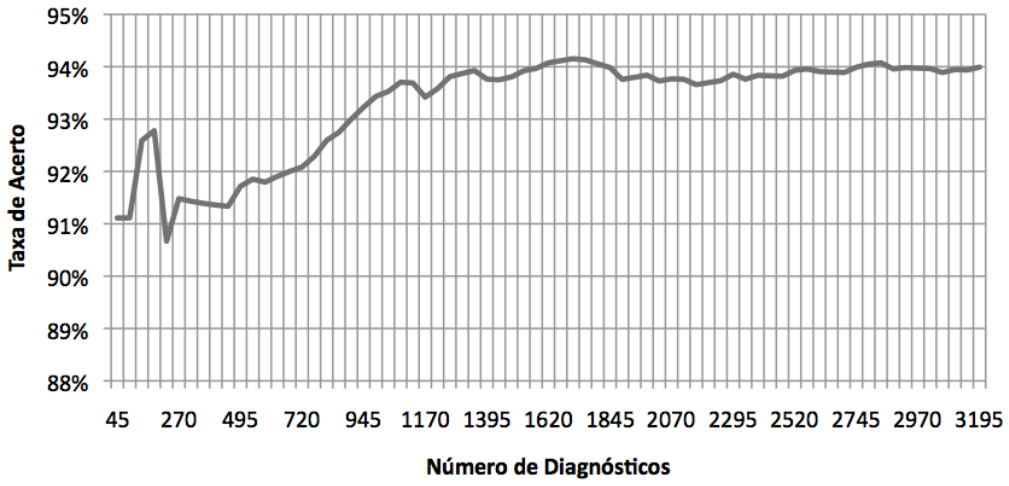


Figura 6.2: Evolução da Taxa Média de Acertos

Agrupamos os 3.195 diagnósticos em 45 amostras independentes de 71 diagnósticos cada. O histograma exibido na Figura 6.3 mostra a distribuição de frequência para a taxa de acerto das 45 amostras. Podemos ver pelo histograma que na grande maioria das amostras (89%), a taxa de acerto foi maior ou igual a 92%, sendo 92% e 93% as taxas de acerto mais frequentes, aparecendo cada uma em 20% das amostras.

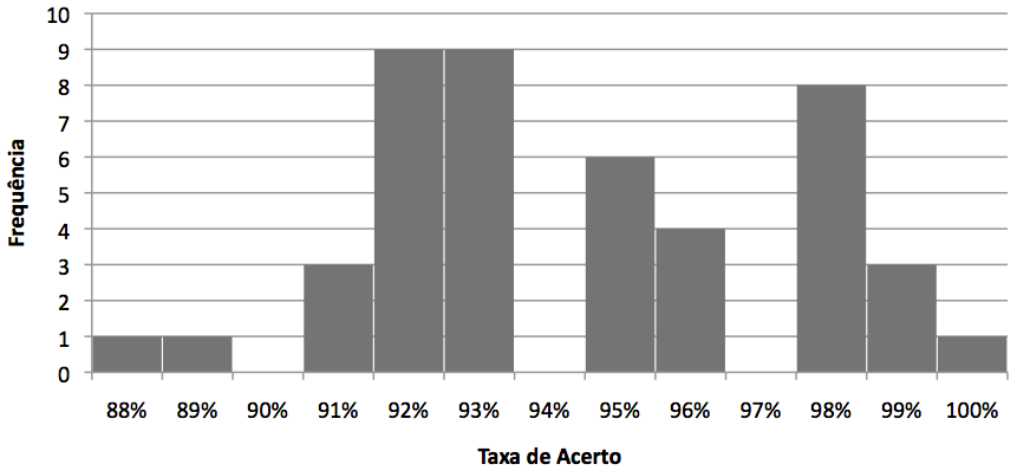


Figura 6.3: Distribuição de Frequência para a Taxa de Acertos

Realizamos então uma análise da distribuição de probabilidades para a taxa de acerto das 45 amostras e identificamos que a taxa de acertos segue uma distribuição normal,

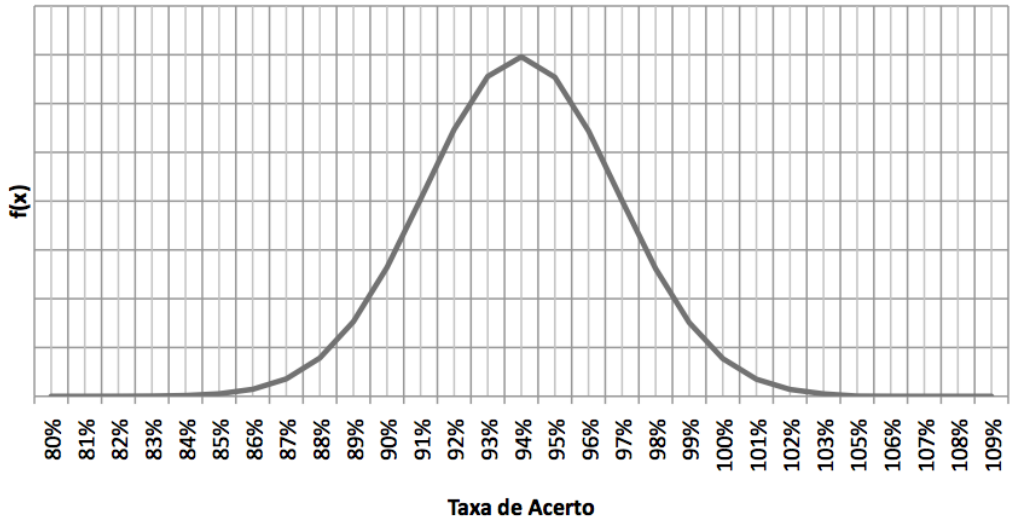


Figura 6.4: Distribuição de Probabilidades para a Taxa de Acertos

como ilustrado na Figura 6.4, com média $\mu = 93,99\%$ e desvio padrão $\sigma = 2,92\%$. Com isso, pudemos definir o intervalo de confiança com nível de confiança de 95% para taxa de acertos do nosso experimento como sendo [Barbetta, Reis e Bornia 2008]: $IC(Taxa\ de\ Acertos, 95\%) = \mu \pm 1,96 \times \sigma$ ou $IC(Taxa\ de\ Acertos, 95\%) = 93,99\% \pm 5,63\%$. Resultando em um intervalo variando de $88,27\%$ a $99,62\%$.

Definimos também o intervalo de confiança com nível de confiança de 99%, que é dado pela equação: $IC(Taxa\ de\ Acertos, 99\%) = \mu \pm 2,576 \times \sigma$ ou $IC(Taxa\ de\ Acertos, 99\%) = 93,99\% \pm 7,52\%$, resultando em um intervalo variando de $86,47\%$ a 100%.

Capítulo 7

Conclusões e Trabalhos Futuros

Apresentamos neste trabalho uma abordagem para a detecção de falhas e diagnóstico de faltas em grades computacionais baseadas na utilização de testes automáticos de software, criados juntamente com o software durante sua fase de desenvolvimento.

Tal abordagem foi substanciada pela criação de uma ferramenta para detecção de falhas e de um *framework* para o diagnósticos de faltas, ambos baseados na utilização de testes automáticos de software.

A ferramenta de detecção de falhas denominada *Service Availability Monitoring*, ou simplesmente SAM, tornou-se a principal ferramenta de monitoramento e operação em várias das maiores infraestruturas de grade computacional existentes hoje [Gagliardi 2005; The European Grid Initiative 2010; SEEGrid 2010; EELA 2010; BalticGrid 2010; NGS 2010; int.eu.grid 2010; D-Grid 2010; EuChinaGrid 2010; EuAsiaGrid 2010; The EUIndiaGrid Project 2010; ItalianGrid 2010; NGS 2010; ItalianGrid 2010].

O *framework* para o diagnóstico automático de faltas em grades, batizado *GridDoctor*, apresenta características importantes em uma solução genérica para o diagnóstico de faltas:

- **Baixa Intrusividade:** não requer qualquer alteração no código fonte dos serviços da grade nem das aplicações dos usuários;
- **Simplicidade:** a solução é extremamente simples, tendo sido implementada em poucas centenas de linhas de código, possibilitando uma fácil compreensão de seu funcionamento;

- **Fácil Manutenção:** a solução está automaticamente atualizada sempre que novas versões dos serviços de grade, juntamente com seu conjunto de testes automáticos, são disponibilizados;

- **Alta Flexibilidade:** a solução se baseia na existência de componentes *Doctors* capazes de diagnosticar faltas em um único tipo de serviço, podendo ser facilmente estendida com a criação de novos componentes de diagnóstico.

Para fins de validação, o *framework* proposto foi instanciado para o mesmo *middleware* de computação em grades para o qual o SAM foi inicialmente projetado, chamado *gLite*, dando origem a uma ferramenta chamada *gLiteGridDoctor*. Tal ferramenta foi utilizada em um experimento prático realizado em uma grade computacional real e em operação.

O experimento foi conduzido durante 47 dias, entre os dias $15/12/2009$ e $05/01/2010$ e entre os dias $04/02/2010$ e $28/02/2010$, resultando na submissão de 4.238 tarefas. Foram criados duas aplicações simples, que fazem transferências de arquivos entre *Computing Elements* e *Storage Elements* e que comparam os arquivos enviados e recebidos nessas transferências. A diferença entre as duas aplicações esta na injeção de uma falha em uma delas para garantir que sua execução não será bem sucedida, independentemente dos serviços da grade utilizados. Tais aplicações foram então submetidas para execução na grade utilizando combinações de serviços considerados funcionais e não funcionais, podendo ser agrupadas como descrito a seguir:

- **Submissão com Falha na Aplicação**: submetemos a aplicação com a falha para execução utilizando um *Computing Element* e um *Storage Element* considerados disponíveis (*Up*) pelo SAM;

- **Submissão com Falha no *Computing Element***: submetemos a aplicação correta para execução utilizando um *Computing Element* considerado indisponível (*Down*) e um *Storage Element* considerado disponível (*Up*);

- **Submissão com Falha no *Storage Element***: submetemos a aplicação correta para execução utilizando um *Storage Element* considerado indisponível (*Down*) e um *Computing Element* considerado disponível (*Up*);

- **Submissão Correta**: submetemos a aplicação correta para execução utilizando um *Computing Element* e um *Storage Element* considerados disponíveis (*Up*) pelo SAM.

3.195 dentre as 4.238 tarefas submetidas falharam, valor muito próximo da taxa esperada de 75% de falhas, uma vez que 3 entre cada 4 tarefas submetidas continham uma falha na própria aplicação ou em um dos serviços utilizados. O *gLiteGridDoctor* foi utilizado para diagnosticar a causa dessas falhas, tendo conseguido apontar corretamente a causa de 3.003 das 3.195 falhas.

Agrupamos os 3.195 diagnósticos em 45 amostras independentes de 71 diagnósticos cada. A análise de distribuição de probabilidades para as 45 amostras coletadas durante o experimento apontou os seguintes intervalos de confiança para a taxa de acerto da ferramenta: $93,99\% \pm 5,63\%$ para um nível de confiança de 95% e $93,99\% \pm 7,52\%$ para um nível de confiança de 99%.

É importante ressaltar que estes resultados foram obtidos em um experimento realizado em uma grade computacional real e em produção. Por conta disso, foram afetadas diretamente pela assincronia existente neste tipo de infraestrutura. Em uma infraestrutura largamente distribuída como uma grade computacional é impossível ter certeza sobre o estado dos serviços antes de efetivamente tentar utilizá-los. Os diagnósticos incorretos realizados pela ferramenta desenvolvida durante o experimento se devem exatamente a esta assincronia, que faz com que o estado da grade no momento em que o sistema é checado pela ferramenta de monitoração seja diferente do seu estado no momento em que as tarefas são submetidas para execução, que por sua vez é diferente do estado da grade no momento em que a ferramenta de diagnóstico tenta identificar a falta que originou a falha na execução da aplicação do usuário.

Os resultados obtidos comprovam a validade da nossa hipótese de que os mesmos testes de aceitação utilizados durante a fase de desenvolvimento para aferir quando o software está pronto ou não para ir para o ambiente de produção podem ser utilizados também para detectar falhas em serviços de grade e servir de base para um serviço de localização e diagnóstico de faltas.

Apesar de não termos encontrado na literatura trabalhos e resultados que pudessem ser comparados com os resultados que obtivemos como nosso experimento, acreditamos que 94% de acerto seja um excelente resultado e que uma ferramenta com este nível de preci-

são pode efetivamente facilitar a vida dos usuários das grades computacionais e ser útil no momento em que estes forem obrigados a identificar a razão pela qual a execução de sua aplicação na grade não foi bem sucedida.

7.1 Trabalhos Futuros

Como possível trabalho futuro vislumbramos um refinamento da implementação do *gLiteGridDoctor* e uma possível integração da ferramenta com máquinas de submissão (*Submission Machines*) utilizadas por muitos usuários e organizações virtuais para gerenciar a submissão de tarefas para execução em uma grade *gLite*.

Tal integração facilitaria ainda mais o uso da solução proposta, implicando na imediata adoção por um grande número de usuários, e possibilitaria a coleta de uma enorme quantidade de dados sobre o desempenho da ferramenta, permitindo uma análise ainda mais precisa de sua precisão e dos benefícios obtidos com sua utilização.

Além disso, pretendemos disponibilizar livremente para a comunidade de usuários de grades computacionais tanto o *framework* criado quanto a ferramenta de diagnóstico desenvolvida para fomentar a instanciação do *framework* para outros *middleware* e outras infraestruturas de grade, aumentando significativamente o tamanho da comunidade potencialmente beneficiada por sua utilização.

Uma outra direção que pretendemos seguir é o desenvolvimento de uma solução para recuperação automática em cenários de falha baseada nos diagnósticos emitidos pelo *GridDoctor*. Dessa forma, o usuário poderia ser totalmente removido do processo de diagnóstico e ressubmissão e poderia concentrar-se apenas na produção e análise dos seus resultados.

Planejamos também executar novos experimentos envolvendo diretamente os usuários de grades computacionais para avaliar os possíveis impactos no seu fluxo de trabalho e em sua produtividade por conta da utilização de uma ferramenta para o diagnóstico automático de faltas.

Finalmente, pretendemos continuar com a condução dos *surveys* sobre gerência de falhas em grades computacionais para podemos traçar um panorama histórico da evolução da área com a possível adoção de ferramentas de diagnóstico automático de faltas, reduzindo

o número de reclamações sobre este que é apontado como o maior problema enfrentado pelos usuários de grades computacionais quando precisam lidar com a ocorrência falhas na execução de suas aplicações.

Bibliografia

[Anderson 2004]ANDERSON, D. P. BOINC: A System for Public-Resource Computing and Storage. In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004. p. 4–10. ISBN 0-7695-2256-4. Disponível em: <http://dx.doi.org/10.1109/GRID.2004.14>.

[Andreozzi et al. 2005]ANDREOZZI, S. et al. GridICE: a Monitoring Service for Grid Systems. *Future Generation Computer Systems*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 21, n. 4, p. 559–571, 2005. ISSN 0167-739X.

[Aridor et al. 2003]ARIDOR, Y. et al. *Reporting Grid Services (ReGS) Specification*. [S.l.], 2003.

[Asvija et al. 2009]ASVIJA, B. et al. Realizing Interoperability among Grids: a Case Study with GARUDA Grid and the EGEE Grid. *Production Grids in Asia*, Springer, p. 175–184, 2009.

[Avizienis et al. 2004]AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, p. 11–33, 2004. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1335465>.

[Baker e Smith 2002]BAKER, M.; SMITH, G. GridRM: A resource Monitoring Architecture for the Grid. *Lecture Notes in Computer Science*, Springer, Berlin, v. 2536, p. 268–273, 2002.

[BalticGrid 2010]BalticGrid. *The BalticGrid Project*. 2010. http://www.balticgrid.org/.

[Barbetta, Reis e Bornia 2008]BARBETTA, P. A.; REIS, M. M.; BORNIA, A. C. *Estatística para Cursos de Engenharia e Informática*. São Paulo, SP, Brasil: Atlas, 2008. 410 p. ISBN 978-85-224-4989-7.

[Barrett et al. 2004]BARRETT, R. et al. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In: *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2004. p. 388–395. ISBN 1-58113-810-5.

[Baud et al. 2003]BAUD, J. et al. *CASTOR Status and Evolution*. [S.l.], 2003.

[Ben-Ari 1999]BEN-ARI, M. The Bug that Destroyed a Rocket. *Journal of Computer Science Education*, v. 13, n. 2, p. 15—16, 1999.

[Bianchini e Buskens 1991]BIANCHINI, R.; BUSKENS, R. An Adaptive Distributed System-level Diagnosis Algorithm and its Implementation. In: *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*. [S.l.: s.n.], 1991. p. 222–220.

[Binder 2000]BINDER, R. *Testing Object-oriented Systems: Models, Patterns, and Tools*. [S.l.]: Addison-Wesley Professional, 2000.

[Black 2002]BLACK, R. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. [S.l.]: Wiley-Interscience, 2002.

[Bonnassieux, Harakaly e Primet 2002]BONNASSIEUX, F.; HARAKALY, R.; PRIMET, P. MapCenter: an Open GRID Status Visualization Tool. In: *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems*. [s.n.], 2002. Disponível em: <citeseer.ist.psu.edu/bonnassieux02mapcenter.html>.

[Bowring, Rehg e Harrold 2004]BOWRING, J.; REHG, J.; HARROLD, M. J. Active Learning for Automatic Classification of Software Behavior. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*. [S.l.: s.n.], 2004.

[Brasileiro et al. 2008]BRASILEIRO, F. et al. An Approach for the Co-existence of Service and Opportunistic Grids: The EELA-2 Case. In: *Proceedings of the Latin-American Grid Workshop*. [S.l.]: LNCC, 2008. v. 1, p. 1–8.

[Bégin et al. 2007]BéGIN, M.-E. et al. Build, Configuration, Integration and Testing. Tools for Large Software Projects : ETICS. *Rapid Integration of Software Engineering Techniques*, v. 81–97, n. 4401, 2007.

[Campana et al. 2008]CAMPANA, S. et al. Testing and Integrating the WLCG/EGEE Middleware in the LHC Computing. In: INSTITUTE OF PHYSICS PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2008. v. 119, p. 062020.

[Case et al. 1993]CASE, J. et al. *Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*. [S.l.], 1993.

[Catlett 2005]CATLETT, C. TeraGrid: A Foundation for US Cyberinfrastructure. *Network and Parallel Computing*, p. 1, 2005. Disponível em: <http://dx.doi.org/10.1007/11577188_1>.

[Cecchini et al. 2003]CECCHINI, A. et al. VOMS, an Authorization System for Virtual. In: *Proceedings of the 1st European Across Grids Conference*. [S.l.: s.n.], 2003. p. 13–14.

[CERN 2001]CERN. *DataTAG*. 2001. http://datatag.web.cern.ch/datatag/.

[CERN 2009]CERN. *Disk Pool Manager: DPM*. 2009. https://twiki.cern.ch/twiki/bin/view/LCG/DpmGeneralDescription.

[CERN 2010]CERN. *CERN Web Site*. 2010. http://www.cern.ch.

[CERN 2010]CERN. *Freedom of Choice for Resources*. 2010. https://savannah.cern.ch/projects/fcr/.

[CERN 2010]CERN. *The European DataGrid Project*. 2010. Http://eu-datagrid.web.cern.ch/.

[Chen et al. 2004]CHEN, M. et al. Failure Diagnosis using Decision Trees. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. [S.l.: s.n.], 2004.

[Chu e Humphrey 2004]CHU, D. C.; HUMPHREY, M. Mobile OGSI.NET: Grid Computing on Mobile Devices. In: *Fifth IEEE/ACM International Workshop on Grid Computing*. [S.l.]: IEEE Computer Society, 2004. p. 182–192.

[Cirne et al. 2006]CIRNE, W. et al. Labs of the World, Unite!!! *Journal of Grid Computing*, Springer, New York, USA, v. 4, n. 3, p. 225–246, 2006. ISSN 1570-7873.

[Cirne et al. 2007]CIRNE, W. et al. On the Efficacy, Efficiency and Emergent Behavior of Task Replication. *Parallel Computing*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 33, n. 3, p. 213–234, 2007. ISSN 0167-8191.

[Cluster Resources 2010]Cluster Resources. *Maui*. 2010. http://www.clusterresources.com/products/maui-cluster-scheduler.php.

[Cordier et al. 2006]CORDIER, H. et al. Grid Operations: the evolution of operational model over the first year. In: *Proceedings of Computing in High Energy and Nuclear Physics*. [S.l.: s.n.], 2006.

[Costa, Dikaiakos e Orlando 2007]COSTA, G. D.; DIKAIAKOS, M. D.; ORLANDO, S. Nine months in the life of EGEE: a look from the South. In: *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. [S.l.: s.n.], 2007. p. 281–287.

[Curbera et al. 2002]CURBERA, F. et al. Unraveling the Web services web: an introduction to SOAP, WSDL, andUDDI. *IEEE Internet computing*, v. 6, n. 2, p. 86–93, 2002.

[D-Grid 2010]D-Grid. *Die Deustch Grid-Initiative*. 2010. http://www.d-grid.de/.

[Dabrowski 2009]DABROWSKI, C. Reliability in Grid Computing Systems. *Concurrency and Computation: Practice & Experience*, John Wiley & Sons, Inc, 605 Third Ave, New York, NY, 10016, USA,, v. 21, n. 8, p. 927–959, 2009.

[Dantas, Cirne e Saikoski 2006]DANTAS, A.; CIRNE, W.; SAIKOSKI, K. Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society*, v. 11, p. 21–35, 2006.

[Davis e Hamscher 1988]DAVIS, R.; HAMSCHER, W. *Exploring artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. 297–346 p. ISBN 0-934613-67-2.

[Deelman et al. 2004]DEELMAN, E. et al. Pegasus: Mapping Scientific Workflows onto the Grid. *Lecture Notes in Computer Science : Grid Computing*, p. 11–20, 2004. Disponível em: <http://www.springerlink.com/content/95rj5e2fgqqpkaha>.

[DEISA 2010]DEISA. *Distributed European Infrastructure for Supercomputing Applications*. 2010. http://www.deisa.eu/.

[DeMillo e Mathur 1995]DEMILLO, R.; MATHUR, A. *Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TEX*. [S.l.], 1995.

[Duarte 2003]DUARTE, A. *Tratamento de Eventos em Redes Elétricas: Uma Ferramenta*. DissertaÁ„o (Mestrado) — Departamento de Sistemas e Computação, Universidade Federal de Campina Grande, 2003.

[Duarte 2009]DUARTE, A. Fault Diagnosis in Grids through Automated Tests. In: *LADC '09: Proceedings of the Fourth Latin-American Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2009. ISBN 978-1-4244-4678-0.

[Duarte et al. 2006]DUARTE, A. et al. Collaborative Fault Diagnosis in Grids through Automated Tests. In: *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*. Washington, DC, USA: IEEE Computer Society, 2006. p. 69–74. ISBN 0-7695-2466-4-01.

[Duarte et al. 2005]DUARTE, A. et al. Using the Computational Grid to Speed up Software Testing. In: *Proceedings of 19th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2005.

[Duarte et al. 2006]DUARTE, A. et al. GridUnit: Software Testing on the Grid. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006. p. 779–782. ISBN 1-59593-375-1.

[Duarte et al. 2007]DUARTE, A. et al. Global grid monitoring: the EGEE/WLCG case. In: *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*. New York, NY, USA: ACM, 2007. p. 9–16. ISBN 978-1-59593-716-2.

[Duarte et al. 2008]DUARTE, A. et al. Monitoring the EGEE/WLCG grid services. *Journal of Physics Conference Series*, v. 119, n. 119, 2008.

[Duarte et al. 2006]DUARTE, A. et al. Multi-environment Software Testing on the Grid. In: *PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. New York, NY, USA: ACM, 2006. p. 61–68. ISBN 1-59593-414-6.

[EELA 2010]EELA. *E-Science Grid Facility for Europe and Latin America*. 2010. http://www.eu-eela.eu/.

[Eerola et al. 2003]EEROLA, P. et al. The NorduGrid Production Grid Infrastructure, Status and Plans. *grid*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, 2003. Disponível em: <http://dx.doi.org/10.1109/GRID.2003.1261711>.

[Erwin e Snelling 2001]ERWIN, D.; SNELLING, D. UNICORE: A Grid Computing Environment. *Euro-Par 2001 Parallel Processing*, p. 825–834, 2001. Disponível em: <http://dx.doi.org/10.1007/3-540-44681-8_116>.

[EuAsiaGrid 2010]EuAsiaGrid. *The EUAsiaGrid Initiative*. 2010. http://www.euasiagrid.org/.

[EuChinaGrid 2010]EuChinaGrid. *The EUChinaGrid Initiative*. 2010. http://www.euchinagrid.org/.

[Field e Schulz 2008]FIELD, L.; SCHULZ, M. Grid Interoperability: the Interoperations Cookbook. *Journal of Physics: Conference Series*, 2008.

[Fitzgerald et al. 1997]FITZGERALD, S. et al. A Directory Service for Configuring High-Performance Distributed Computations. In: *Proceedings of the Sixth IEEE Symposium on High Performance Distributed Computing*. [S.l.]: IEEE Computer Society Press, 1997. p. 365–375.

[Foster e Kesselman 1997]FOSTER, I.; KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, v. 11, n. 2, p. 115, 1997.

[Foster, Kesselman e Tuecke 2001]FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, v. 15, p. 200–222, 2001.

[Fuhrmann 2004]FUHRMANN, P. dCache: The Commodity Cache. In: *Proceedings of the 21st IEEE Conference on Mass Storage Systems and Technologies*. [S.l.: s.n.], 2004.

[Gagliardi 2005]GAGLIARDI, F. The egee european grid infrastructure project. In: DAYDÉ, M. et al. (Ed.). *High Performance Computing for Computational Science: VECPAR 2004: 6th International Conference Valencia, Spain, June 28–30, 2004 Revised Selected and Invited Papers*. [s.n.], 2005. v. 3402, p. 194–203. ISBN 3-540-25424-2. ISSN 0302-9743. Disponível em: <http://www.springerlink.com/openurl.asp?genre=issueissn=0302-9743volume=3402; http://www.springerlink.com/openurl.asp?genre=volumeid=doi:10.1007/b106965>.

[Gamma e Beck 1999]GAMMA, E.; BECK, K. Junit: A cook's tour. *Java Report*, v. 5, n. 4, p. 27–38, 1999.

[GARUDA 2010]GARUDA. *India's National Grid Computing Initiative*. 2010. http://www.garudaindia.in/.

[Garzoglio et al. 2009]GARZOGLIO, G. et al. Definition and Implementation of a SAML-XACML Profile for Authorization Interoperability across Grid Middleware in OSG and EGEE. *Journal of Grid Computing*, Springer, v. 7, n. 3, p. 297–307, 2009.

[Gentzsch 2001]GENTZSCH, W. Sun Grid Engine: Towards Creating a Compute Power Grid. In: *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*. [s.n.], 2001. p. 35–36. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=923173>.

[Geraci 1991]GERACI, A. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. Piscataway, NJ, USA: IEEE Press, 1991. ISBN 1559370793.

[Grimshaw et al. 1999]GRIMSHAW, A. S. et al. Wide-Area Computing: Resource Sharing on a Large Scale. *IEEE Computer*, v. 35, p. 29–37, 1999.

[Gronager et al. 2008]GRONAGER, M. et al. Interoperability between ARC and gLite - Understanding the Grid-Job Life Cycle. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2008 Fourth IEEE International Conference on eScience*. [S.l.], 2008. p. 493–500.

[Guangbao, Jie e Bo 2004]GUANGBAO, N.; JIE, M.; BO, L. GridView: A Dynamic and Visual Grid Monitoring System. In: IEEE COMPUTER SOCIETY. *Proceedings of the Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region*. [S.l.], 2004. p. 89–92.

[Gupta, Chandra e Goldszmidt 2001]GUPTA, I.; CHANDRA, T. D.; GOLDSZMIDT, G. S. On Scalable and Efficient Distributed Failure Detectors. In: ACM. *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*. [S.l.], 2001. p. 179.

[Hofer e Fahringer 2008]HOFER, J.; FAHRINGER, T. Grid Application Fault Diagnosis Using Wrapper Services and Machine Learning. *International Journal of Cooperative Information Systems (IJCIS)*, v. 3, n. 17, 2008.

[Hwang e Kesselman 2003]HWANG, S.; KESSELMAN, C. A Flexible Framework for Fault Tolerance in the Grid. *Journal of Grid Computing*, v. 1, n. 3, p. 251–272, 2003. Disponível em: <http://dx.doi.org/10.1023/B:GRID.0000035187.54694.75>.

[IGTF 2010]IGTF. *The International Grid Trust Federation*. 2010. http://www.italiangrid.org/.

[Imamagic e Dobrenic 2007]IMAMAGIC, E.; DOBRENIC, D. Grid Infrastructure Monitoring System based on Nagios. In: *GMW '07: Proceedings of the 2007 workshop on Grid monitoring*. New York, NY, USA: ACM, 2007. p. 23–28. ISBN 978-1-59593-716-2. Disponível em: <http://dx.doi.org/10.1145/1272680.1272685>.

[IN2P3 2010]IN2P3. *The CIC Portal*. 2010. https://operations-portal.in2p3.fr.

[int.eu.grid 2010]int.eu.grid. *The Interactive Grid Initiative Project*. 2010. http://www.i2g.eu/.

[ItalianGrid 2010]ItalianGrid. *The Italian Grid Infrastructure*. 2010. http://www.italiangrid.org/.

[ITU-T 2000]ITU-T. *Draft Revised ITU-T Recommendation X.509*. [S.l.], 2000.

[J., Bakken e Petravick 2004]J., T. P.; BAKKEN; PETRAVICK, D. Storage Resource Manager. In: *Proceedings of the Conference on Computing for High Energy Physics*. [S.l.: s.n.], 2004.

[Jeffries, Anderson e Hendrickson 2000]JEFFRIES, R. E.; ANDERSON, A.; HENDRICKSON, C. *Extreme Programming Installed*. [S.l.]: Addison-Wesley, 2000.

[Jones 1990]JONES, C. *Systematic Software Development using VDM*. [S.l.]: Prentice Hall, 1990.

[Kacsuk, Farkas e Fedak 2008]KACSUK, P.; FARKAS, Z.; FEDAK, G. Towards Making BOINC and EGEE Interoperable. In: *IEEE Fourth International Conference on eScience*. [S.l.: s.n.], 2008. p. 478–484.

[Kola, Kosar e Livny 2004]KOLA, G.; KOSAR, T.; LIVNY, M. Phoenix: Making Data-Intensive Grid Applications Fault-Tolerant. In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004. p. 251–258. ISBN 0-7695-2256-4.

[Kuhn 1999]KUHN, D. R. Fault Classes and Error Detection in Specification Based Testing. *ACM Transactions on Software Engineering Methodology*, v. 4, n. 8, p. 411—424, 1999.

[Lamanna 2004]LAMANNA, M. The LHC computing grid project at CERN. In: *Proceedings of the IXth International Workshop on Advanced Computing and Analysis Techniques in Physics Research*. [S.l.: s.n.], 2004. p. 1–6.

[Litke et al. 2007]LITKE, A. et al. Efficient task replication and management for adaptive fault tolerance in mobile Grid environments. *Future Generation Computing. Systems*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 23, n. 2, p. 163–178, 2007. ISSN 0167-739X.

[Litzkow, Livny e Mutka 1988]LITZKOW, M.; LIVNY, M.; MUTKA, M. Condor – A Hunter of Idle Workstations. In: *Proceedings of the 8th International Conference of Distributed Computing Systems*. [S.l.: s.n.], 1988. p. 104–111.

[Maglio e Kandogan 2004]MAGLIO, P.; KANDOGAN, E. Error Messages: What's the Problem? Real-world Tales of Woe Shed some Light. *ACM Queue*, v. 2, p. 51–55, 2004.

[McKeag e Macnaghten 1980]MCKEAG, R. M.; MACNAGHTEN, A. M. *On the Construction of Programs*. New York, NY, USA: Cambridge University Press, 1980. ISBN 052123090X.

[Medeiros et al. 2003]MEDEIROS, R. et al. Faults in Grids: Why are they so bad and What can be done about it? In: *GRID '03: Proceedings of the 4th International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2003. p. 18. ISBN 0-7695-2026-X.

[Meglio 2007]MEGLIO, A. D. *Grid Software Engineering Challenges*. [S.l.], 2007.

[Meglio et al. 2008]MEGLIO, A. di et al. ETICS: the International Software Engineering Service for the Grid. *Journal of Physics Conference Series*, v. 119, n. 119, 2008.

[Mellor 1994]MELLOR, P. CAD: Computer-Aided Disaster. *Highly Integrated Systems*, v. 1, n. 2, p. 101–156, 1994.

[Mirgorodskiy, Maruyama e Miller 2006]MIRGORODSKIY, A. V.; MARUYAMA, N.; MILLER, B. P. Problem Diagnosis in Large-Scale Computing Environments. In: *Proceedings of the ACM/IEEE Supercomputing'06 Conference*. [S.l.: s.n.], 2006.

[Myers et al. 2004]MYERS, G. J. et al. *The Art of Software Testing*. [S.l.]: Wiley, 2004.

[Nakada et al. 2008]NAKADA, H. et al. Job Invocation Interoperability between NAREGI Middleware Beta and gLite. *Journal of Physics Conference Series*, n. 119, 2008.

[NAREGI 2010]NAREGI. *National Institute of Informatics*. 2010. http://www.naregi.org/.

[Neocleous et al. 2007]NEOCLEOUS, K. et al. Failure Management in Grids: The Case of the EGEE Infrastructure. *Parallel Processing Letters*, v. 17, n. 4, p. 391–410, 2007.

[NGS 2010]NGS. *The National Grid Service*. 2010. http://www.ngs.ac.uk/.

[Nicolas et al. 2006]NICOLAS, J. et al. Demonstration of In Silico Docking at a Large Scale on Grid Infrastructure. In: IOS PR INC. *Challenges and opportunities of HealthGrids: proceedings of Healthgrid 2006*. [S.l.], 2006. p. 155.

[Ortmeier e Reif 2004]ORTMEIER, F.; REIF, W. *Failure-sensitive Specification - A Formal Method for Finding Failure Modes*. [S.l.], 2004.

[PBS Works]PBS Works. *OpenPBS Web Site*. Http://www.openpbs.org.

[Podgurski et al. 2003]PODGURSKI, A. et al. Automated Support for Classifying Software Failure Reports. In: *Proceedings of 25th International Conference on Software Engineering*. [S.l.: s.n.], 2003. p. 465—-475.

[Pordes et al. 2007]PORDES, R. et al. The Ppen Science Grid. *Journal of Physics: Conference Series*, Institute of Physics Publishing, v. 78, n. 1, p. 012057+, 2007. ISSN 1742-6596. Disponível em: <http://dx.doi.org/10.1088/1742-6596/78/1/012057>.

[Qian-Mu, Man-wu e Hong 2006]QIAN-MU, L.; MAN-WU, X.; HONG, Z. A Root-fault Detection System of Grid Based on Immunology. In: *GCC '06: Proceedings of the Fifth International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, 2006. p. 369–373. ISBN 0-7695-2694-2.

[Riedel et al. 2008]RIEDEL, W. et al. Improving e-Science with Interoperability of the e-Infrastructures EGEE and DEISA. In: *International Convention on Information and Communication Technology, Electronics and Microelectronics*. [S.l.: s.n.], 2008. p. 225–231.

[Rosen 1963]ROSEN, R. Relational Biology and Bionics. *Military Electronics, IEEE Transactions on*, v. 7, n. 2, p. 160–162, 1963. Disponível em: <http://dx.doi.org/10.1109/TME.1963.4323065>.

[Sauvé et al. 2005]SAUVé, J. P. et al. Maintenance Techniques for an Intelligent Alarm Processing System. *Engineering Intelligent Systems*, v. 13, n. 4, p. 213–222, 2005.

[SEEGrid 2010]SEEGrid. *The SEEGrid Project*. 2010. http://www.see-grid.org/.

[Smallen et al. 2004]SMALLEN, S. et al. The Inca Test Harness and Reporting Framework. In: *Proceedings of the ACM/IEEE Supercomputing'04 Conference*. [S.l.: s.n.], 2004.

[Smith 2001]SMITH, W. *A Framework for Control and Observation in Distributed Environments*. [S.l.], 2001.

[Staples 2006]STAPLES, G. TORQUE Resource Manager. In: ACM. *Proceedings of The 2006 ACM/IEEE Conference on Supercomputing*. [S.l.], 2006. p. 8.

[Stelling et al. 1998]STELLING, P. et al. A Fault Detection Service for Wide Area Distributed Computations. In: *Proceedings of the 7th IEEE Symposium On High Performance Distributed Computing*. [S.l.: s.n.], 1998. p. 268–278.

[The EUIndiaGrid Project 2010]The EUIndiaGrid Project. *Joining European and Indian Grids for e-Science Network Community*. 2010. http://www.euindiagrid.org/.

[The European Grid Initiative 2010]The European Grid Initiative. *The European Grid Initiative Web Site*. 2010. Http://egi.eu.

[The GStat Team 2010]The GStat Team. *Grid Statistics (GStat) Description*. 2010. http://goc.grid.sinica.edu.tw/gstat/filter_help.html.

[The Large Hadron Collider Collaboration 2010]The Large Hadron Collider Collaboration. *LHC Web Site*. 2010. http://lhc.web.cern.ch/lhc/.

[The Open Grid Forum 2009]The Open Grid Forum. *The Open Grid Forum Web Site*. 2009. http://www.ogf.org.

[Tierney et al. 2002]TIERNEY, B. et al. *A grid Monitoring Architecture*. [S.l.], 2002.

[Tierney et al. 2000]TIERNEY, B. et al. A Monitoring Sensor Management System for Grid Environments. In: *Proceedings of the IEEE High Performance Distributed Computing Conference*. [S.l.: s.n.], 2000. p. 97–104.

[Waheed et al. 2000]WAHEED, A. et al. An Infrastructure for Monitoring and Management in Computational Grids. In: *Proceedings of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*. [S.l.: s.n.], 2000. p. 235–245.

[Wahl, Howes e Kille 1997]WAHL, M.; HOWES, T.; KILLE, S. Lightweight Directory Access Protocol. In: *RFC 2251 http://www.ietf.org/rfc/rfc2251.txt*. [S.l.: s.n.], 1997.

[Wang et al. 2008]WANG, L. et al. On-Demand Build a Virtual e-Science Workflow. In: *Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08.*

The 3rd International Conference on. [s.n.], 2008. p. 93–98. Disponível em: <http://dx.doi.org/10.1109/GPC.WORKSHOPS.2008.46>.

[Wang, Cheng e Chen 2009]WANG, Y.; CHENG, Y.; CHEN, G. Interoperability between gLite and GOS. *Production Grids in Asia*, Springer, p. 155–173, 2009.

[Wang et al. 2007]WANG, Y. et al. Interconnect EGEE and CNGRID e-infrastructures through interoperability between gLite and GOS middlewares. In: *International Grid Interoperability and Interoperation Workshop*. [S.l.: s.n.], 2007. p. 553–560.

[Weissman 1998]WEISSMAN, J. *Fault Tolerant Computing on the Grid: What are My Options?* [S.l.], 1998.

[WestGrid 2010]WestGrid. *Western Canada Research Grid*. 2010. http://www.westgrid.ca/.

[Xie e Lai 1996]XIE, M.; LAI, C. Reliability analysis using an additive Weibull model with bathtub-shaped failure rate function. *Reliability Engineering and System Safety*, Elsevier, v. 52, n. 1, p. 87–93, 1996.

[Zhou et al. 1992]ZHOU, S. et al. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software Practice and Experience*, 1992.

[Zhou 2010]ZHOU, W. *Fault Management in Distributed Systems*. [S.l.], 2010.

Apêndice A

Resultado Detalhado de uma Falha Detectada pelo SAM

Este Apêndice apresenta o resultado completo emitido pelo SAM no momento em que foi detectada uma falha na execução do teste de submissão de tarefa em um *Computing Element*

SAM test: *CE-sft-job*
Submitter VO: *ops*
Node: *iut15auvergridce01.univ-bpclermont.fr*
Execution time: *26-Apr-2010 03:24:54*

```
Mon Apr 26 02:07:56 UTC 2010
Submitting from host: sam211.cern.ch
DN: /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
```

Generating JDL file:

```
Executable = "/bin/sh";
Arguments = "-x testjob.sh";
StdOutput = "testjob.out";
StdError = "testjob.out";
InputSandbox = {"testjob.sh","testjob.tgz","same.conf"};
OutputSandbox = {"testjob.out","testjob-results.tgz"};
Requirements = other.GlueCEInfoHostName == "iut15auvergridce01.univ-bpclermont.fr";
RetryCount = 0;
ShallowRetryCount = 1;
```

content of *testjob.sh*

```
#!/bin/sh
tar xzf testjob.tgz
if [ $? -ne 0 ] ; then
    echo "Problem unpacking SAM Framework on WN: $(hostname)"
    exit 1
fi
export SAME_WORK=`pwd`/work; bin/same-exec -c same.conf --nodetest testjob iut15auvergridce01.univ-bpclermont.fr -- CE-1272247276 sam
```

Submitting a job

```
+ glite-wms-job-submit -a --vo ops -o testjob.jid testjob.jdl
Warning - --vo option ignored
Connecting to the service https://wms208.cern.ch:7443/glite_wms_wmproxy_server
====================== glite-wms-job-submit Success ======================
The job has been successfully submitted to the WMProxy
Your job identifier is:
https://wms208.cern.ch:9000/k5OkrDH0-mHdlFSxqayEOA
The job identifier has been saved in the following file:
/home/samops/.same/CE/nodes/iut15auvergridce01.univ-bpclermont.fr/testjob.jid
==========================================================================
+ set +x
```

Job status:

```
**********************************************************
BOOKKEEPING INFORMATION:
Status info for the Job : https://wms208.cern.ch:9000/k5OkrDH0-mHdlFSxqayEOA
Current Status:     Aborted
Logged Reason(s):
    - Job got an error while in the CondorG queue.
    - Job got an error while in the CondorG queue.
Status Reason:      hit job shallow retry count (1)
Destination:        iut15auvergridce01.univ-bpclermont.fr:2119/jobmanager-lcgpbs-ops
Submitted:          Mon Apr 26 04:07:59 2010 CEST
**********************************************************
```

Job failed with logging info:

```
**************************************************************************
LOGGING INFORMATION:
Printing info for the Job : https://wms208.cern.ch:9000/k5OkrDH0-mHdlFSxqayEOA
        ---
Event: RegJob
- Arrived                   =    Mon Apr 26 04:07:59 2010 CEST
- Host                      =    wms208.cern.ch
- Ns                        =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Nsubjobs                  =    0
- Source                    =    NetworkServer
- Src instance              =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Timestamp                 =    Mon Apr 26 04:07:59 2010 CEST
- User                      =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
        ---
Event: RegJob
- Arrived                   =    Mon Apr 26 04:07:59 2010 CEST
- Host                      =    wms208.cern.ch
- Ns                        =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Nsubjobs                  =    0
- Source                    =    NetworkServer
- Src instance              =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Timestamp                 =    Mon Apr 26 04:07:59 2010 CEST
- User                      =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
        ---
Event: Accepted
- Arrived                   =    Mon Apr 26 04:08:01 2010 CEST
- From                      =    NetworkServer
- From host                 =    sam211.cern.ch
- Host                      =    wms208.cern.ch
- Source                    =    NetworkServer
- Src instance              =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
```

```
- Timestamp                =    Mon Apr 26 04:08:01 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
        ---
Event: EnQueued
- Arrived                  =    Mon Apr 26 04:08:01 2010 CEST
- Host                     =    wms208.cern.ch
- Queue                    =    /var/glite/workload_manager/jobdir
- Result                   =    START
- Source                   =    NetworkServer
- Src instance             =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Timestamp                =    Mon Apr 26 04:08:01 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
        ---
Event: EnQueued
- Arrived                  =    Mon Apr 26 04:08:02 2010 CEST
- Host                     =    wms208.cern.ch
- Queue                    =    /var/glite/workload_manager/jobdir
- Result                   =    OK
- Source                   =    NetworkServer
- Src instance             =    https://wms208.cern.ch:7443/glite_wms_wmproxy_server
- Timestamp                =    Mon Apr 26 04:08:02 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka
        ---
Event: DeQueued
- Arrived                  =    Mon Apr 26 04:08:02 2010 CEST
- Host                     =    wms208.cern.ch
- Queue                    =    /var/glite/workload_manager/jobdir
- Source                   =    WorkloadManager
- Src instance             =    28945
- Timestamp                =    Mon Apr 26 04:08:02 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Match
- Arrived                  =    Mon Apr 26 04:08:04 2010 CEST
- Dest id                  =    iut15auvergridce01.univ-bpclermont.fr:2119/jobmanager-lcgpbs-ops
- Host                     =    wms208.cern.ch
- Source                   =    WorkloadManager
- Src instance             =    28945
- Timestamp                =    Mon Apr 26 04:08:04 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                  =    Mon Apr 26 04:08:04 2010 CEST
- Host                     =    wms208.cern.ch
- Queue                    =    /var/glite/jobcontrol/jobdir
- Result                   =    START
- Source                   =    WorkloadManager
- Src instance             =    28945
- Timestamp                =    Mon Apr 26 04:08:04 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                  =    Mon Apr 26 04:08:04 2010 CEST
- Host                     =    wms208.cern.ch
- Queue                    =    /var/glite/jobcontrol/jobdir
- Result                   =    OK
- Source                   =    WorkloadManager
- Src instance             =    28945
- Timestamp                =    Mon Apr 26 04:08:04 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: DeQueued
- Arrived                  =    Mon Apr 26 04:08:05 2010 CEST
- Host                     =    wms208.cern.ch
- Local jobid              =    unavailable
- Queue                    =    /var/glite/jobcontrol/jobdir
- Source                   =    JobController
- Src instance             =    unique
- Timestamp                =    Mon Apr 26 04:08:05 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                  =    Mon Apr 26 04:08:05 2010 CEST
- Dest host                =    localhost
- Dest instance            =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid               =    unavailable
- Destination              =    LogMonitor
- Host                     =    wms208.cern.ch
- Reason                   =    unavailable
- Result                   =    START
- Source                   =    JobController
- Src instance             =    unique
- Timestamp                =    Mon Apr 26 04:08:05 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                  =    Mon Apr 26 04:08:05 2010 CEST
- Dest host                =    localhost
- Dest instance            =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid               =    2445508
- Destination              =    LogMonitor
- Host                     =    wms208.cern.ch
- Reason                   =    unavailable
- Result                   =    OK
- Source                   =    JobController
- Src instance             =    unique
- Timestamp                =    Mon Apr 26 04:08:05 2010 CEST
- User                     =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Accepted
- Arrived                  =    Mon Apr 26 04:08:10 2010 CEST
- From                     =    JobController
- From host                =    localhost
- From instance            =    unavailable
```

```
- Host                    =    wms208.cern.ch
- Local jobid             =    2445508
- Source                  =    LogMonitor
- Src instance            =    unique
- Timestamp               =    Mon Apr 26 04:08:10 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                 =    Mon Apr 26 04:23:38 2010 CEST
- Dest host               =    unavailable
- Dest instance           =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid              =    unavailable
- Destination             =    LRMS
- Host                    =    wms208.cern.ch
- Reason                  =    10 data transfer to the server failed
- Result                  =    FAIL
- Source                  =    LogMonitor
- Src instance            =    unique
- Timestamp               =    Mon Apr 26 04:23:38 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Done
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Exit code               =    1
- Host                    =    wms208.cern.ch
- Reason                  =    Job got an error while in the CondorG queue.
- Source                  =    LogMonitor
- Src instance            =    unique
- Status code             =    FAILED
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Resubmission
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Host                    =    wms208.cern.ch
- Reason                  =    unavailable
- Result                  =    WILLRESUB
- Source                  =    LogMonitor
- Src instance            =    unique
- Tag                     =    unavailable
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Host                    =    wms208.cern.ch
- Queue                   =    /var/glite/workload_manager/jobdir
- Reason                  =    unavailable
- Result                  =    START
- Source                  =    LogMonitor
- Src instance            =    unique
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Host                    =    wms208.cern.ch
- Queue                   =    /var/glite/workload_manager/jobdir
- Reason                  =    unavailable
- Result                  =    OK
- Source                  =    LogMonitor
- Src instance            =    unique
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: DeQueued
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Host                    =    wms208.cern.ch
- Queue                   =    /var/glite/workload_manager/jobdir
- Source                  =    WorkloadManager
- Src instance            =    28945
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Resubmission
- Arrived                 =    Mon Apr 26 04:23:45 2010 CEST
- Host                    =    wms208.cern.ch
- Reason                  =    token still exists
- Result                  =    SHALLOW
- Source                  =    WorkloadManager
- Src instance            =    28945
- Tag                     =    /var/glite/SandboxDir/k5/https_3a_2f_2fwms208.cern.ch_3a9000_2fk5OkrDH0-mHdlFSxqayEOA/token.txt
- Timestamp               =    Mon Apr 26 04:23:45 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Match
- Arrived                 =    Mon Apr 26 04:23:46 2010 CEST
- Dest id                 =    iut15auvergridce01.univ-bpclermont.fr:2119/jobmanager-lcgpbs-ops
- Host                    =    wms208.cern.ch
- Source                  =    WorkloadManager
- Src instance            =    28945
- Timestamp               =    Mon Apr 26 04:23:46 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                 =    Mon Apr 26 04:23:46 2010 CEST
- Host                    =    wms208.cern.ch
- Queue                   =    /var/glite/jobcontrol/jobdir
- Result                  =    START
- Source                  =    WorkloadManager
- Src instance            =    28945
- Timestamp               =    Mon Apr 26 04:23:46 2010 CEST
- User                    =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
```

```
Event: EnQueued
- Arrived                    =    Mon Apr 26 04:23:46 2010 CEST
- Host                       =    wms208.cern.ch
- Queue                      =    /var/glite/jobcontrol/jobdir
- Result                     =    OK
- Source                     =    WorkloadManager
- Src instance               =    28945
- Timestamp                  =    Mon Apr 26 04:23:46 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: DeQueued
- Arrived                    =    Mon Apr 26 04:23:47 2010 CEST
- Host                       =    wms208.cern.ch
- Local jobid                =    unavailable
- Queue                      =    /var/glite/jobcontrol/jobdir
- Source                     =    JobController
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:23:47 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                    =    Mon Apr 26 04:23:47 2010 CEST
- Dest host                  =    localhost
- Dest instance              =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid                 =    unavailable
- Destination                =    LogMonitor
- Host                       =    wms208.cern.ch
- Reason                     =    unavailable
- Result                     =    START
- Source                     =    JobController
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:23:47 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                    =    Mon Apr 26 04:23:47 2010 CEST
- Dest host                  =    localhost
- Dest instance              =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid                 =    2445573
- Destination                =    LogMonitor
- Host                       =    wms208.cern.ch
- Reason                     =    unavailable
- Result                     =    OK
- Source                     =    JobController
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:23:47 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Accepted
- Arrived                    =    Mon Apr 26 04:23:51 2010 CEST
- From                       =    JobController
- From host                  =    localhost
- From instance              =    unavailable
- Host                       =    wms208.cern.ch
- Local jobid                =    2445573
- Source                     =    LogMonitor
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:23:51 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Transfer
- Arrived                    =    Mon Apr 26 04:49:15 2010 CEST
- Dest host                  =    unavailable
- Dest instance              =    /var/glite/logmonitor/CondorG.log/CondorG.1272241459.log
- Dest jobid                 =    unavailable
- Destination                =    LRMS
- Host                       =    wms208.cern.ch
- Reason                     =    10 data transfer to the server failed
- Result                     =    FAIL
- Source                     =    LogMonitor
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:49:15 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Done
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Exit code                  =    1
- Host                       =    wms208.cern.ch
- Reason                     =    Job got an error while in the CondorG queue.
- Source                     =    LogMonitor
- Src instance               =    unique
- Status code                =    FAILED
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Resubmission
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Host                       =    wms208.cern.ch
- Reason                     =    unavailable
- Result                     =    WILLRESUB
- Source                     =    LogMonitor
- Src instance               =    unique
- Tag                        =    unavailable
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Host                       =    wms208.cern.ch
- Queue                      =    /var/glite/workload_manager/jobdir
- Reason                     =    unavailable
- Result                     =    START
- Source                     =    LogMonitor
- Src instance               =    unique
```

```
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: EnQueued
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Host                       =    wms208.cern.ch
- Queue                      =    /var/glite/workload_manager/jobdir
- Reason                     =    unavailable
- Result                     =    OK
- Source                     =    LogMonitor
- Src instance               =    unique
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: DeQueued
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Host                       =    wms208.cern.ch
- Queue                      =    /var/glite/workload_manager/jobdir
- Source                     =    WorkloadManager
- Src instance               =    28945
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Abort
- Arrived                    =    Mon Apr 26 04:49:27 2010 CEST
- Host                       =    wms208.cern.ch
- Reason                     =    hit job shallow retry count (1)
- Source                     =    WorkloadManager
- Src instance               =    28945
- Timestamp                  =    Mon Apr 26 04:49:27 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=wlapka/CN=623537/CN=Wojciech Lapka/CN=proxy/CN=proxy
        ---
Event: Clear
- Arrived                    =    Mon Apr 26 05:14:47 2010 CEST
- Host                       =    wms208.cern.ch
- Reason                     =    USER
- Source                     =    NetworkServer
- Src instance               =    NS
- Timestamp                  =    Mon Apr 26 05:14:47 2010 CEST
- User                       =    /DC=ch/DC=cern/OU=computers/CN=wms208.cern.ch
**************************************************************************
```

Contact: If you have any questions, please submit a GGUS ticket, assigned to the test submitter VO
